MLL normalization and transitive closure: circuits, complexity, and Euler tours

Harry Mairson
Problem: Given a proofnet in multiplicative linear logic (MLL), what is the computational complexity of determining its normal form? 
*sensitive to size of output*

Decision problem: Given two proofnets in multiplicative linear logic, what is the computational complexity of determining if they have the same normal form? 
*output insensitive - more interesting*
Why study the complexity of normalization?

*Linearity* is the key ingredient in understanding the complexity of type inference.

**Approximation:** every occurrence of a bound variable has the same type (simple types, ML, intersection types). Thus *linearity subverts approximation:* it renders type inference synonymous with *normalization.*
Why study the complexity of normalization?

MLL is a baby programming language:
- ⊗ is a linear pairing of expressions (cons) expression and continuation (@)
- ⊘ is a linear unpairing of expressions (π, π’) expression and continuation (λ)

complexity of normalization = complexity of interpreter
Plan...

Preliminaries
Complexity of normalization: some results...
Some technical details...
Some preliminaries...
Sequent rules for multiplicative linear logic

\[
\frac{\alpha, \alpha}{\alpha, \alpha} \quad \frac{\Gamma, \alpha \downarrow, \Delta}{\Gamma, \Delta, \alpha \downarrow, \Delta} \quad \frac{\Gamma, \alpha \downarrow, \Delta, \beta}{\Gamma, \alpha \downarrow, \Delta, \alpha \otimes \beta} \quad \frac{\Gamma, \alpha, \beta}{\Gamma, \alpha, \beta}
\]

An ML(L) metalanguage: MLL proofs, written in linear ML

Note: $\alpha \rightarrow \beta = \alpha \downarrow \otimes \beta$, $\alpha \downarrow \downarrow = \alpha$, two sided sequents

($\alpha \downarrow$ on left is like $\alpha$ on right), etc.

\[
\frac{x : \alpha \vdash x : \alpha}{\Gamma \vdash E : \alpha \quad \Delta \vdash F : \beta}
\]

\[
\frac{\Gamma, \Delta \vdash (E,F) : \alpha \otimes \beta}{\Gamma \vdash E : \alpha \quad \Delta \vdash F : \beta}
\]

\[
\frac{\Gamma \vdash F : \alpha \otimes \beta \quad \Delta, x : \alpha, y : \beta \vdash E : \sigma}{\Gamma, \Delta \vdash \text{let } (x,y) = F \text{ in } E : \sigma}
\]

\[
\frac{\Gamma \vdash \lambda x. E : \alpha \rightarrow \beta}{\Gamma \vdash E : \alpha \rightarrow \beta \quad \Delta \vdash F : \alpha}
\]

\[
\frac{\Gamma \vdash E : \alpha \rightarrow \beta \quad \Delta \vdash F : \alpha}{\Gamma, \Delta \vdash E \ F : \beta}
\]
Proofnets for multiplicative linear logic
How complex is an MLL proof?

Are the axiom formulas atomic or non-atomic?
(A is a propositional variable; $\alpha$ is an arbitrary formula.)

\[
\begin{align*}
A \vdash A, & \quad \text{Ax} \\
A \vdash A, & \quad \text{Ax}
\end{align*}
\]

What is the logical depth of cut formulas?

\[
\begin{align*}
\Gamma, \alpha \quad \alpha \vdash, \Delta, & \quad \text{Cut} \\
\Gamma, \Delta & \\
\Gamma \vdash E: \alpha \quad \beta & \quad \Delta \vdash F: \alpha \\
\Gamma, \Delta \vdash E F: \beta
\end{align*}
\]

The answers to these questions affect the computational complexity of normalization, and the expressive power of the logic.
η-expansion:
(linear in formula size, possibly exponential in formula depth)

\[ \alpha \otimes \beta, \alpha^\perp \otimes \beta^\perp \]

(programming language equivalents:)
\[ x : A \times B \]
\[ f : A \rightarrow B \]
\[ \lambda y : A. \ fy \]
Normalization (Computation)

\[
\frac{\Gamma, \alpha^\perp \quad \Delta, \beta^\perp}{\Gamma, \Delta, \alpha^\perp \otimes \beta^\perp} \quad \frac{\Sigma, \alpha, \beta}{\Sigma, \alpha \otimes \beta} \quad \Rightarrow \quad \frac{\Delta, \beta^\perp \quad \Sigma, \alpha, \beta}{\Gamma, \Delta, \Sigma}
\]

\[
\frac{\alpha, \alpha^\perp \quad \alpha, \alpha^\perp}{\alpha, \alpha^\perp} \quad \Rightarrow \quad \alpha, \alpha^\perp
\]

(Normalization always makes the proof get smaller.)

ML(L) programming language analogs (none for Ax-Cut ...)

let (x,y)=(U,V) in E  \Rightarrow  E[U/x,V/y]

(\lambda x.E)F  \Rightarrow  E[F/x]
Normalization (Computation) -- proofnet version

\[ \pi_1 \rightarrow \pi_2 \rightarrow \pi_3 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \pi_3 \]

Nice, easy, friendly, parallelizable, local...

\[ \pi_1 \rightarrow \pi_2 \rightarrow \pi_1 \rightarrow \pi_2 \]

Computationally worrysome: especially with non-atomic axioms...

Transitive closure on edges: not local!
Computationally problematic if done repeatedly.
When does this happen?
Transitive closure on edges: not local!
Computationally problematic if done repeatedly.

When does this happen?

\[ (\lambda x. x)((\lambda x. x)((\lambda x. x)(\ldots((\lambda x. x) y)\ldots))) \]

[dual to] \[ \ldots(((\lambda x. x) (\lambda x. x)) (\lambda x. x))\ldots) (\lambda x. x) y \]

f = (\lambda x. x) = \[ also (and more interesting!) parity function, permutation, unbounded fan-in Boolean operation, transitive closure,... \]

(Complexity/expressiveness: what can you compute with a “flat” proofnet?)
\((\lambda x.x)((\lambda x.x)((\lambda x.x)(...((\lambda x.x) \ y)...))))\)

local, parallel

global, sequential (costly?)
**MLLu:** MLL with unbounded fanout [Terui, 2004]

\[
\frac{\Gamma_1, A_1 \quad \Gamma_2, A_2 \quad \ldots \quad \Gamma_n, A_n}{\Gamma_1, \Gamma_2, \ldots, \Gamma_n, \otimes_n (A_1, A_2, \ldots, A_n)} \otimes_n
\]

\[
\frac{\Gamma, A_1, A_2, \ldots, A_n}{\Gamma, \otimes_n (A_1, A_2, \ldots, A_n)} \otimes_n
\]

*Why MLLu is needed:* to simulate unbounded **fanout** in circuits (and not unbounded fanin!) in constant depth

[computational behavior (theorems, complexity results) virtually identical to MLL.]
Complexity of normalization: some results...
For MLL with atomic axioms, normalization of a proof is complete for LOGSPACE.

**Containment:** Given a proof of size \( n \), its normal form can be computed in \( O(\log n) \) space. (Research: proved, and reproved?)

**Hardness:** An arbitrary computation requiring \( O(\log n) \) space (say, on a Turing machine with input of size \( n \)) can be compiled (reduced) to an MLL proof of size \( O(n^k) \) -- whose normalization (simulating the Turing machine calculation) takes \( O(\log n) \) space.

The reduction must use less resources than \( O(\log n) \) space, for example \( NC_1 \) -- polynomial-sized circuits of depth [time] \( O(\log n) \).
For MLL with non-atomic axioms, normalization of a proof is complete for PTIME.

**Containment:** Given a proof of size $n$, its normal form can be computed in $O(n^k)$ time. (A trivial observation.)

**Hardness:** An arbitrary computation requiring $O(n^k)$ time (say, on a Turing machine with input of size $n$) can be compiled (reduced) to an MLL proof of size $O(n^{ck})$ and depth $O(n^k)$ -- whose normalization (simulating the Turing machine calculation) takes $O(n^{ck})$ time.

The reduction must use less resources than PTIME, for example LOGSPACE.
**Circuits + TC = MLLu:** For MLLu with non-atomic axioms and formula depth $d$, normalization of a proof is equivalent to evaluating Boolean circuits with depth $\Theta(d)$, modulo a logic gate for constant-time transitive closure operation on acyclic graphs (using adjacency matrices).

**Circuits + TC ⊆ MLLu:** A Boolean circuit of size $n$ and depth $d$, including TC gates, can be simulated with uniform types by an MLLu proof of size $O(n)$ and formula depth $O(d)$. [TC on acyclic graphs is a constant-depth MLLu computation.]

**MLLu ⊆ Circuits + TC:** An MLLu proof of size $n$ and (formula) depth $d$ can be simulated by a Boolean circuit with TC gates, of size $O(n^k)$ and depth $O(d)$. (Terui, 2004)

The reduction must use less resources, for example $\text{NC}_1$ -- polynomial-sized circuits of depth time $O(\log n)$. 

What’s Old?

Papers by Kazushige Terui (LICS 2004) and myself (ICTCS 2003), others...

What’s New?

**MLLu Boolean computations “without garbage”**: A size- and depth-preserving coding of Boolean circuits in MLLu which does not create garbage (output with extra terms and a computation-dependent type). This coding improves a garbage-dependent coding without uniform types of Boolean logic in MLLu, due to Terui (2004).

**LOGSPACE-hardness**: MLL normalization with atomic axioms is as hard as any problem requiring LOGSPACE. (Reduction from the permutation problem.)

**Constant-depth transitive closure in MLLu**: An MLLu proof whose normalization simulates transitive closure on an \(n\)-node acyclic graph. The proof has a constant formula depth (not dependent on \(n\)), and is a baroque elaboration of the LOGSPACE-hardness argument.
Some technical details...
Boolean logic à la Church: linear, but affine
Very, very, old...

- 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

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

- fun K x y= x;
val K = fn : 'a -> 'b -> 'a

- fun Bizarre p= K (Same p) (Not p);
val Bizarre = fn :
   (('a -> 'a -> 'a) -> ('b -> 'b -> 'b) -> 'c) -> 'c
- Bizarre True;
val it = fn : 'a -> 'a -> 'a
- Bizarre False;
val it = fn : 'a -> 'a -> 'a
Paradise regained: copying and linearity

- fun Copy p= p (True,True) (False,False);

val Copy = fn
  : (('a -> 'b -> 'a) * ('c -> 'd -> 'c)
      -> ('e -> 'f -> 'f) * ('g -> 'h -> 'h) -> 'i)
      -> 'i

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

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

- fun nonBizarre p=
    let val (p',p'')= Copy p in K (Same p') (Not p'') end;

val nonBizarre = fn
  : (('a -> 'b -> 'a) * ('c -> 'd -> 'c)
      -> ('e -> 'f -> 'f) * ('g -> 'h -> 'h)
      -> (('i -> 'j -> 'i) -> ('k -> 'l -> 'l) -> 'm)
      * (('n -> 'o -> 'o) -> ('p -> 'q -> 'p) -> 'r))
      -> 'm

- nonBizarre True;
val it = fn : 'a -> 'b -> 'a

- nonBizarre False;
val it = fn : 'a -> 'b -> 'b
Continuation-passing style

- fun Notgate p k = k (Not p);
val Notgate = fn
  : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e) -> ('e -> 'f) -> 'f

- fun Andgate p q k = k (And p q);
val Andgate = fn :
  ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a -> ('d -> 'e) -> 'e

- fun Orgate p q k = k (Or p q);
val Orgate = fn :
  (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c -> ('d -> 'e) -> 'e

- fun Copygate p k = k (Copy p);
val Copygate = fn :
  (('a -> 'b -> 'a) * ('c -> 'd -> 'c)
    -> ('e -> 'f -> 'f) * ('g -> 'h -> 'h) -> 'i)
  -> ('i -> 'j) -> 'j
Circuit evaluation: type inference is normalization

- fun Circuit e1 e2 e3 e4 e5 e6=
  (Andgate e2 e3 (fn e7=>
  (Andgate e4 e5 (fn e8=>
  (Andgate e7 e8 (fn f=>
  (Copygate f (fn (e9,e10)=>
  (Orgate e1 e9 (fn e11=>
  (Orgate e10 e6 (fn e12=>
  (Orgate e11 e12 (fn Output=> Output))))))))))))));

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 -> 's -> 's) * ('t -> 'u -> 'u)
  -> 'c * ('v -> 'w -> 'v) -> 'x -> 'f)
  -> 'h -> ('y -> ('z -> 'ba -> 'ba) -> 'k) -> 'y -> 'x -> 'g

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

NB: Normalization and type inference are here synonymous.
The problem of garbage

Now try and make the calculations non-affine, as in MLL...

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

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

- **fun I** x= x;
  
  val I = fn : 'a -> 'a

- **fun Compose** (f,g) x= f (g x);
  
  val Compose = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- **fun Erase** b= let val (u,v)= b(I,I) in Compose (u,v) end;
  
  val Erase = fn
  : ('a -> 'a) * ('b -> 'b) -> ('c -> 'd) * ('e -> 'c) -> 'e -> 'd

- **Erase True**;
  
  val it = fn : 'a -> 'a

- **fun Or** p q= let val (u,v)= p (True,q) in (Erase v) u end;
  
  val Or = fn
  : ('a * 'b -> 'a * 'b) * 'c
  -> 'd * ('e -> 'e) * ('f -> 'f) -> ('g -> 'h) * ('d -> 'g))
  -> 'c -> 'h

- **Or True False**;
  
  val it = fn : 'a * 'b -> 'a * 'b
The problem of garbage (2)

Now try and let True and False have a uniform type, so that type inference is not synonymous with normalization. (Then the type of Or does not depend on the type/normal form of its arguments...)

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

- fun Or (p, q)= let val (u,v)= p (True,q) in (Erase v) u end;  
  val Or = fn : (('a * 'a -> 'a * 'a) * 'b -> 'c * (('d -> 'd) * ('e -> 'e) -> ('f -> 'g) * ('c -> 'f))) -> 'b -> 'g
- Or True False;  

```
stdIn:15.1-15.14 Error: operator and operand don't agree [circularity]
```

Partial solution: maintain and create garbage during the calculation...

- fun Or (p,p') (q,q')= let val (u,v)=p (True,q) in (u,(p',q',v)) end;  
  val Or = fn : (('a * 'b -> 'a * 'b) * 'c -> 'd * 'e) * 'f 
            -> 'c * 'g -> 'd * ('f * 'g * 'e)

...but can we eliminate this garbage? ...
Booleans, the non-affine variation, *without* garbage

- fun TT (x:'a,y:'a)= (x,y);
  val TT = fn : 'a * 'a -> 'a * 'a
- fun FF (x:'a,y:'a)= (y,x);
  val FF = fn : 'a * 'a -> 'a * 'a

- val True= (TT: ('a * 'a -> 'a * 'a), FF: ('a * 'a -> 'a * 'a));
  val True = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
- val False= (FF: ('a * 'a -> 'a * 'a), TT: ('a * 'a -> 'a * 'a));
  val False = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)

- fun Show (u,v)= (let val (x,y)= u(true,false) in x end,
  let val (x,y)= v(true,false) in x end);
  val Show = fn :
    (bool * bool -> 'a * 'b) * (bool * bool -> 'c * 'd) -> 'a * 'c
- Show True;
  val it = (true,false) : bool * bool
- Show False;
  val it = (false,true) : bool * bool
Symmetric logic gates

And \((p,p')(q,q') = (p \land q, p' \lor q') = (p \land q, \neg(p \land q))\)

- \(\text{fun And \((p,p') (q,q') = \)}\)
  \(\text{let val ((u,v),(u',v')) = (p (q,FF), p' (TT,q'))}\)
  \(\text{in (u,Compose (Compose (u',v),Compose (v',FF)))) end;}\)

val And = fn
  : ('a * ('b * 'b -> 'b * 'b) -> 'c * ('d -> 'e))
    * (('f * 'f -> 'f * 'f) * 'g -> ('e -> 'h) * ('i * 'i -> 'd))
    -> 'a * 'g -> 'c * ('i * 'i -> 'h)

An occurrence:
\((a \land a) \land (a \land a) \land (a \land a) \land (a \land a) \land (a \land a) \land (a \land a)\)

If \(B = (a \land a), this is (B \land B \rightarrow B)[B/'a]\)

- And True False;
val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
- Show (And True True);
val it = (true,false) : bool * bool
- Show (And True False);
val it = (false,true) : bool * bool
- Show (And False True);
val it = (false,true) : bool * bool
- Show (And False False);
val it = (false,true) : bool * bool
Why is there no garbage?

And \((p,p')(q,q')\) = \((p \land q, p' \lor q')\) = \((p \land q, \sim (p \land q))\)

```
fun And (p,p') (q,q')=
    let val ((u,v),(u',v')) = (p (q,FF), p' (TT,q'))
    in (u,Compose (Compose (u',v),Compose (v',FF))) end;
```

\((u,v) = (p (q,FF))\)
\((u',v') = (p'(TT,q'))\)

When \(p=TT,\) \(u,v) = (q, FF)\)
\((u',v') = (q',TT)\)

When \(p=FF,\) \(u,v) = (FF, q)\)
\((u',v') = (TT,q')\)

Thus \(\{v,v'\} = \{TT,FF\},\) and

\(\text{Compose (v,Compose(v',FF))} = \text{TT (identity function)}\)
\(\text{Compose (Compose (u',v),Compose (v',FF))} = u'\)

“Symmetric garbage is self-annihilating”
Symmetric logic gates (2)

\[ \text{Or} (p, p')(q, q') = (p \lor q, p' \land q') = (p \lor q, \neg (p \lor q)) \]

- fun Or (p,p') (q,q') =
  let val ((u,v),(u',v')) = (p (TT,q), p' (q',FF))
  in (u,Compose (Compose (u',v),Compose (v',FF))) end;

val Or = fn
  : ('a * 'a -> 'a * 'a) * 'b -> 'c * ('d -> 'e)
  * ('f * ('g * 'g -> 'g * 'g) -> ('e -> 'h) * ('i * 'i -> 'd))
  -> 'b * 'f -> 'c * ('i * 'i -> 'h)
- Or True False;
val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)

- Show (Or True True);
val it = (true,false) : bool * bool
- Show (Or True False);
val it = (true,false) : bool * bool
- Show (Or False True);
val it = (true,false) : bool * bool
- Show (Or False False);
val it = (false,true) : bool * bool
Symmetric logic gates (3)

- fun Not (x,y)= (y,x);
val Not = fn : 'a * 'b -> 'b * 'a

- Not True;
val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
- Show (Not True);
val it = (false,true) : bool * bool
- Show (Not False);
val it = (true,false) : bool * bool
Symmetric logic gates (4)

- fun Copy (p,p')= (p (TT,FF), p' (FF,TT));
val Copy = fn
  : (('a * 'a -> 'a * 'a) * ('b * 'b -> 'b * 'b) -> 'c)
  * (('d * 'd -> 'd * 'd) * ('e * 'e -> 'e * 'e) -> 'f)
  -> 'c * 'f

Set 'a = 'b = 'd = 'e and 'c = 'f = ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a):

  (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) ->
   ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
* (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) ->
   ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))

[ p= TT]:  Copy (p,p')= ((TT,FF), (TT,FF))  [second component reversed]
[ p= FF]:  Copy (p,p')= ((FF,TT), (FF,TT))  [first component reversed]

- let val (p,q)= Copy True in (Show p, Show q) end;
val it = ((true,false),(true,false)) : (bool * bool) * (bool * bool)
- let val (p,q)= Copy False in (Show p, Show q) end;
val it = ((false,true),(false,true)) : (bool * bool) * (bool * bool)
Symmetric logic gates (5)

- fun Copy4 (p,p')=
  let val (((s,s'),(p,p')),((q,q'),(r,r')))=
    (p ((TT,TT),(FF,FF)), p' ((FF,FF),(TT,TT)))
  in ((s,p),(s',p'),(q,r),(q',r')) end;
val Copy4 = fn
  : ((('a * 'a -> 'a * 'a) * ('b * 'b -> 'b * 'b))
      * (('c * 'c -> 'c * 'c) * ('d * 'd -> 'd * 'd))
      -> ('e * 'f) * ('g * 'h))
      * (('i * 'i -> 'i * 'i) * ('j * 'j -> 'j * 'j))
      * (('k * 'k -> 'k * 'k) * ('l * 'l -> 'l * 'l))
      -> ('m * 'n) * ('o * 'p))
      -> ('e * 'g) * ('f * 'h) * ('m * 'o) * ('n * 'p)

- let val (s,p,q,r)=Copy4 True in (Show s,Show p,Show q,Show r) end;
val it = ((true,false),(true,false),(true,false),(true,false)) :
  (bool * bool) * (bool * bool) * (bool * bool) * (bool * bool)
Symmetric logic gates (6)

- fun Circuit e1 e2 e3 e4 e5 e6 =
  (Andgate e2 e3 (fn e7=>
  (Andgate e4 e5 (fn e8=>
  (Andgate e7 e8 (fn f=>
  (Copygate f (fn (e9,e10)=>
  (Orgate e1 e9 (fn e11=>
  (Orgate e10 e6 (fn e12=>
  (Orgate e11 e12 (fn Output=> Output))))))))))))));
val Circuit = fn
  : < big type... >

- Circuit True False False False False True;
val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
- Show (Circuit True False False False False False True);
val it = (true,false) : bool * bool
Unbounded fan-in

The And of an arbitrary number of Boolean values can be computed in a constant depth MLL proof:

- fun Showtype x F= K x (F x);
  val Showtype = fn : 'a -> ('a -> 'b) -> 'a

- Showtype And (fn A=> (A True (And False (And True True))));
  val it = fn
    : (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
      -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
      -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
    -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)

- Showtype And (fn A=> (A True (A False (A True True))));
  val it = fn
    : (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
      -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
      -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
    -> ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)
Unbounded fan-in: Associate the binary
operator to the left: depth grows linearly
(size grows exponentially) under η-expansion.

Showtype And
(fn A=> (And (And (A True False) True) True));
val it = fn
: ((('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  (* ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  (* ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)))

-> (((('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  (* ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  (* ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))));
Following paths in a proof of $\phi$: atomic MLL normalization in \textsc{Logspace}

\textbf{Idea:} Use the \textit{Geometry of Interaction} (GoI), following paths in a \textit{proofnet} representing an MLL proof.

When axioms are atomic, paths can only be \textit{polynomial} in length (to be explained).

When paths become \textit{superpolynomial} in length

[example: non-atomic axioms, formula depth $O(\log^2 n)$, paths can have length $2^{\log n}$], attempts to parallelize following paths (in $\text{NC}^k$) will fail.
Following paths in a proof of $\phi$: atomic MLL normalization in LOGSPACE

A LOGSPACE operation: at axiom link $p$, guess the link $q$ such that the path $(p, \text{cut})$ and $(q, \text{cut})$ are isomorphic [modulo the gates]. Storing pointers $p, q$, etc. requires $O(\log n)$ bits -- four pointers suffice.

Isomorphic paths must enter nodes with same left/right orientation:

$T_1$ and $T_2$ are isomorphic, with $\otimes$ in place of $\otimes$ (links at leaves are different)

The usual path-following of the geometry of interaction is only modified by eliminating the stack (with height bounded by formula depth).
Terui’s LOGSPACE algorithm (2002): the Euler tour

1st Stage
- Copy axioms and links above conclusions.
- Find a cut and place two pointers on it.
- Make a trip as follows:

  ![Diagram of Euler tour](image)

- When two pointers arrive at axioms, connect the two axioms visited and continue the trip.
- When they come back to the original cut, finish.

2nd Stage. Apply axiom reductions.

Since both are logspace and logspace functions compose, the overall algorithm runs in logspace.
MLL normalization is as hard as LOGSPACE: the permutation problem

Problem: Given a permutation $\pi$ on $\{1, 2, \ldots, n\}$ and $1 \leq i, j \leq n$, are $i$ and $j$ on the same cycle of $\pi$?

Why this problem is LOGSPACE-hard: A LOGSPACE computation can be compiled into permutations on Turing machine IDs, with two cycles: the accept cycle, and the reject cycle. Which cycle is the initial ID on?

Why a binary tree? Without loss of generality, every ID is reached by a either the tape head moving left or moving right from a unique previous ID.

LOGSPACE = polynomial number of IDs/nodes (compiler from TM problem to permutation problem has limited resources).

To get a permutation, take an Euler tour of the tree.

Compiling this problem into MLL: check if $i, j$ are on the same cycle, using a proof/program of constant depth (formula complexity). A programming problem...
Transitive closure on permutations, in constant-depth MLL (without $u$)

- val P=(False: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  False: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  False: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  False: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a));
val P = ((fn,fn),(fn,fn),(fn,fn),(fn,fn))
  : <type omitted>

- fun Perm (S: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  P: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  Q: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  R: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))=
  (Q,P,S,R);
val Perm = fn
  : (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
    -> (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
      * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
      * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
      * (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
Transitive closure on permutations, in constant-depth MLL (2)

- fun Insert ((s,s'):
  ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  P: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  Q: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a),
  R: ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))=
  ((TT: 'a * 'a -> 'a * 'a, Compose (s,s'))),P,Q,R);
val Insert = fn
  : (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  -> (('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))
  * ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a))

Notice that (s,s') is either (TT,FF) or (FF,TT), but in either case, Compose (s,s')=FF. Thus linearity is preserved because the garbage is symmetric.
Transitive closure on permutations, in constant-depth MLL (3)

- Show True;
  val it = (true,false) : bool * bool
- fun Showvector (S,P,Q,R)= (Show S,Show P,Show Q,Show R);
  val Showvector = fn
    : <type omitted>

- Showvector P;
  val it = ((false,true), (false,true), (false,true), (false,true))
    : (bool * bool) * (bool * bool) * (bool * bool) * (bool * bool)
- Showvector (Insert P);
  val it = ((true,false), (false,true), (false,true), (false,true))
    : (bool * bool) * (bool * bool) * (bool * bool) * (bool * bool)
- Showvector (Insert (Perm (Insert P)));
  val it = ((true,false), (false,true), (true,false), (false,true))
    : (bool * bool) * (bool * bool) * (bool * bool) * (bool * bool)

For a permutation on \{1,2,...,n\}, iterate Compose(Perm,Insert) \(n\) times. Then index \(j\) is True when 1 and \(j\) are on the same cycle.
For a permutation on \( \{1,2,...,n\} \), iterate \( \text{Compose}(\text{Perm},\text{Insert}) \) \( n \) times. Then index \( j \) is True when 1 and \( j \) are on the same cycle.

- \textit{Showvector} \( \text{Insert}(\text{Perm}(\text{Insert} P)) \);
  \[
  \text{val it = ((true,false), (false,true), (true,false), (false,true))}
  : \text{(bool * bool) * (bool * bool) * (bool * bool) * (bool * bool)}
  \]

- \textit{fun ExtractR} \( s,p,q,(r',r'') \)=
  \[
  (r', \text{Compose}(r'', \text{Compose}(\text{Compose}(FF, \text{Compose} s),
  \text{Compose}(\text{Compose} p,\text{Compose} q)))));
  \]
  \[
  \text{val ExtractR = fn}
  : (('a -> 'b) * ('c -> 'a)) * (('d -> 'c) * ('e -> 'd)) * (('f -> 'e) * ('g -> 'f)) * (('b -> 'h) * 'i) -> ('g -> 'h) * 'i
  \]

[Recall: \( \text{Compose} \text{ True} = \text{Compose} (\text{TT,FF}) = \text{FF} = \text{Compose} \text{ False} \)]

- \textit{ExtractR} \( \text{Insert}(\text{Perm}(\text{Insert} P)) \);
  \[
  \text{val it = (fn,fn) : ('a * 'a -> 'a * 'a) * ('a * 'a -> 'a * 'a)}
  \]

- \textit{Show (ExtractR (Insert (Perm (Insert P))))};
  \[
  \text{val it = (false,true) : bool * bool}
  \]
Transitive closure on acyclic graphs in constant depth MLLu

Boolean circuit gate

Constant-depth MLL proof

\[
\begin{align*}
\text{number of nodes in graph} & \quad n \\
\text{size of adjacency matrix} & \quad m = \frac{n(n-1)}{2} \\
\text{enumeration of adjacency matrix} & \quad (i_t,j_t) \\
\text{vertices} & \quad r,s \\
\text{Is there a path from vertex } r \text{ to vertex } s?
\end{align*}
\]
Lemma. Let $G=(V,E)$ be an undirected, acyclic graph, and let $\sigma$ be an arbitrary enumeration of its edges. Each edge defines a transposition on $V$. Then the composition of the enumerated transpositions forms a cyclic permutation (Euler tour!) on the vertices of each connected component.

Example:

\[
\begin{align*}
\sigma &= (P,Q) (S,P) (Q,R) \\
&= s \quad p \quad q \quad r \\
(PQ) &= s \quad q \quad p \quad r \\
(SP) &= q \quad s \quad p \quad r \\
(QR) &= q \quad s \quad r \quad p \\
(PQ) \cdot (SP) \cdot (SR) &= (SQRP)
\end{align*}
\]

Proof by induction on $|V|$. (It fails, easily, for cyclic graphs.)

Application: to check if two vertices are connected, use a fixed permutation (on edges) to broadcast message from one to the other.
To check if $R$ is connected to $S$, use a fixed permutation (on edges) to broadcast message from $R$ to $S$

Graph adjacency matrix $e_{i1j1} e_{i2j2} e_{imjm}$ ($m=n^2$, Assume $n$ even)

Swap data in position of $p$- and $q$-vector given by edge $(u,v)$
[equal to True=$\text{TT,FF}$ or False=$\text{FF,TT}$]

fun Swap\text{ij} (u,v) (p1,...,pn,q1,...,qn) =
  let val ((p'i,p'j),(q'i,q'j)) = (u(pi,pj), v(qi,qj))
  in (p1,...,p'i,...,p'j,pn,...,q1,...,q'i,...,q'j,...,qn)

Insert True in position $pr$, False in position $qr$:
fun Insert\text{r} (p1,...,pr,...,pn,q1,...,qr,...,qn) =
  (p1,...,(Compose pr, TT),...,pn,q1,...,(TT, Compose qr),...,qn)
To check if $R$ is connected to $S$, use a fixed permutation (on edges) to broadcast message from $R$ to $S$ (2)

Graph adjacency matrix $e_{i_1 j_1} \ e_{i_2 j_2} \ e_{i_m j_m}$ ($m=n^2, n$ even)

Make $n$ copies of each edge (**big** fanout):

let val (  
  $(e_{i_1 j_1 1}, e_{i_1 j_1 2}, \ldots, e_{i_1 j_1 n}),$
  $(e_{i_2 j_2 1}, e_{i_2 j_2 2}, \ldots, e_{i_2 j_2 n}),$
  \ldots,
  $(e_{i_m j_m 1}, e_{i_m j_m 2}, \ldots, e_{i_m j_m n})) =$

$(\text{Copy}_n \ e_{i_1 j_1}, \text{Copy}_n \ e_{i_2 j_2}, \ldots, \text{Copy}_n \ e_{i_m j_m}))$

in ...
To check if R is connected to S, use a fixed permutation (on edges) to broadcast message from R to S (2)

Graph adjacency matrix $e_{i1j1} e_{i2j2} e_{imjm}$ ($m=n^2$, Assume $n$ even)

let val (F1,...,Fk,...,Fn)= (...,

(fn V=>

(Swap$_{i1j1}$ e$_{i1j1}$k)(Swap$_{i2j2}$ e$_{i2j2}$k)...(Swap$_{imjm}$ e$_{imjm}$k)

...)

Insert$_r$ V), ...),

in let val (p1,...,pn,q1,...,(q’s,q’’s),...,qn)=

F1(F2(...(Fn (True,..., True, False,...,False))...))

in (Compose (q’s, (Compose p1)$^o$...$^o$(Compose pn)),

Compose (q’’s, (Compose q1)$^o$...$^o$(Compose qs-1)

$^o$(Compose True)$^o$(Compose qs+1)...$^o$(Compose qn)))
Summary

*Parity* is everywhere -- it is the canonical programming technique in MLL. A close cousin is the *symmetry of garbage*.

*de Morgan symmetry:* the key to computing Boolean functions in MLL with a uniform type, and without garbage.

**Circuits + \( TC = MLL_u \):**

- **\( u \):** Unlike Boolean circuits, MLL *does not support* constant-depth unbounded Boolean *fan-out* (it has constant-depth unbounded Boolean *fan-in*).
- **\( TC \):** Unlike MLL, Boolean circuits do not have constant-depth *transitive closure* on acyclic graphs.

*Complexity and geometry of interaction:* following paths is the key to the semantics. It is also the key to the LOGSPACE-completeness of MLL with atomic axioms.

*Transitive closure* on acyclic graphs: the natural consequence of programming in bounded depth (*parallelism*). Transitive closure, coded in MLL, *creates its own transitive closure problem.* (In fact, the same one -- it is an *analog computation.*)
Proof of lemma by induction on $|V|$. Assume $G$ is a single connected component with $n+1$ vertices, including edge $(j,n+1)$:

$\sigma = \ldots (j,n+1) \ldots$

Without edge $(j,n+1)$

(by induction):

\[
\begin{align*}
1 & \quad 2 & \quad \ldots & \quad \pi^{-1}(k) & \quad \ldots & \quad n \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
\pi(1) & \quad \pi(2) & \quad \ldots & \quad k & \quad \ldots & \quad \pi(n)
\end{align*}
\]

Cycle:

$(\ldots \pi^{-1}(k) \ k \ \pi(k) \ \ldots)$

With edge $(j,n+1)$:

\[
\begin{align*}
1 & \quad j & \quad k & \quad \pi^{-1}(k) & \quad n & \quad n+1 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
k & \quad n+1 & \quad \pi(n) & & & \quad k \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
\pi(1) & \quad \pi(j) & \quad \pi(k) & \quad n+1 & \quad \pi(n) & \quad k
\end{align*}
\]

Cycle:

$(\ldots \pi^{-1}(k) \ n+1 \ k \ \pi(k) \ \ldots)$
The lemma fails for a cyclic graph

\[ \sigma = \text{acebdf} \]

\[ \begin{align*}
&\text{ABCDEF} \\
&\text{abcdef} \\
&\text{bacdef} \\
&\text{badcfe} \\
&\text{badacfe} \\
&\text{bda}f\text{ce} \\
&\text{edaf}f\text{cb} \\
&\prod = (\text{aec})(\text{bdf})
\end{align*} \]
Transitive closure on acyclic graphs in constant depth \text{MLLu}

(this you cannot do with a Boolean circuit in constant depth, because you cannot compute parity in constant depth)

There is a path from $P$ to $Q$ when the parity is odd...