On the transposition of computer programs

L. De Feo
(joint work with É. Schost)

Projet TANC, LIX, École Polytechnique

COSEC Seminar, **b**-it
Bonn, June 10, 2010

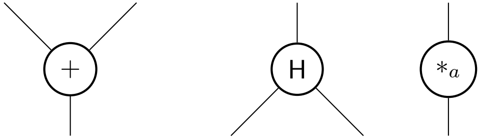
Plan

Arithmetic circuits

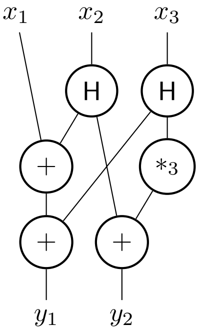
Algebraic complexity

Fix a ring R. We construct circuits that evaluate arithmetic functions.

Three gates: Addition, duplication, multiplication by a fixed element $a \in R$.



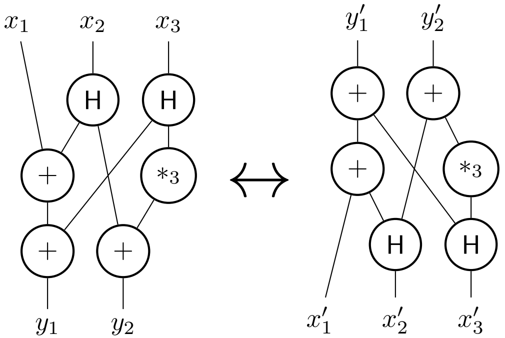
Arithmetic circuits



$$y_1 = x_1 + x_2 + x_3$$
$$y_2 = x_2 + 3x_3$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix}$$

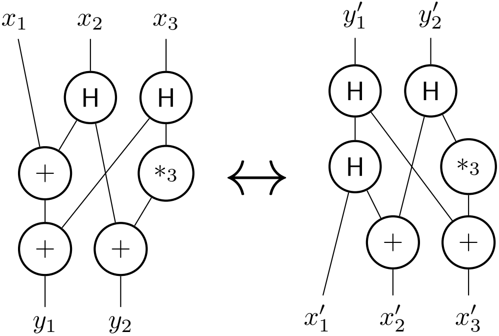
Transposition of an arithmetic circuit



$$y_1 = x_1 + x_2 + x_3$$
$$y_2 = x_2 + 3x_3$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix}$$
$$\updownarrow$$
$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 3 \end{pmatrix}$$

Transposition of an arithmetic circuit



$$y_1 = x_1 + x_2 + x_3$$
$$y_2 = x_2 + 3x_3$$

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix}$$
$$\updownarrow$$
$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 3 \end{pmatrix}$$

Arithmetic Circuits: uniform vs. non-uniform

Definition (Circuit family)

A *circuit family* is a family $(C_0, C_1, \ldots)$ of circuits indexed by $\mathbb{N}$ such that C_n has n inputs.

- Any Turing-undecidable problem has a trivial *polynomial-size* circuit family deciding it.

Definition (Uniform circuit family)

A circuit family $(C_0, C_1, \ldots)$ is said to be *uniform* if there is a $\log n$-space bounded touring machine which on input 1^n outputs a representation of C_n.

- We will extend the definition to allow families to be indexed by any (countable) set $\mathcal{P}$, called the *parameter space*.

From non-uniform to uniform circuits?

- The transposition theorem easily generalises to non-uniform circuits.
- It can even be directly applied to computer programs (under certain hypotheses).

```
for i = 0 to n-2 do
  a[i+1] = a[i] + a[i+1]
  a[i] = 0
end for
```

$$\begin{pmatrix} 0 & \cdots\cdots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \cdots\cdots & 0 \\ 1 & \cdots\cdots & 1 \end{pmatrix}$$

From non-uniform to uniform circuits?

```
a[1] = a[0] + a[1]
a[0] = 0
a[2] = a[1] + a[2]
a[1] = 0
...
a[n-1] = a[n-2] + a[n-1]
a[n-2] = 0
```

```
a[n-2] = 0
a[n-2] = a[n-2] + a[n-1]
...
a[1] = 0
a[1] = a[1] + a[2]
a[0] = 0
a[0] = a[0] + a[1]
```

```
for i = n-2 to 0 do
  a[i] = 0
  a[i] = a[i] + a[i+1]
end for
```

$$\begin{pmatrix} 0 & \cdots\cdots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \cdots\cdots & 0 \\ 1 & \cdots\cdots & 1 \end{pmatrix}$$

Plan

Minimal polynomials [Shoup '95]

Linear recurring sequences

An algebraic element $\sigma \in \mathbb{L} = \mathbb{K}[X]/f(X)$, the σ^i's satisfy a linear recurrence

$$\sigma^n = c_{n-1}\sigma^{n-1} + \cdots + c_1\sigma + c_0$$

Let $\ell \in \mathbb{L}^*$ a linear form, then

$$\ell(\sigma^n) = c_{n-1}\ell(\sigma^{n-1}) + \cdots + c_1\ell(\sigma) + c_0\ell(1)$$

Power projection

Given ℓ and σ, the power projection problem asks to compute

$$\ell(1), \ell(\sigma), \ell(\sigma^2), \ldots, \ell(\sigma^{n-1})$$

For fixed σ, it is a $\mathbb{K}$-linear map, its transpose is the map

$$g \mapsto g(\sigma) \bmod f$$

hence its transpose problem is *modular composition*.

Applications

Uses and generalisations of power projection

- Minimal polynomials in towers of extension fields [Shoup '95].
- Change of order in triangular sets.
- Change of order in Artin-Schreier towers [D.F., Schost '09], application to isogeny computation.

Other applications

- Generation of irreducible polynomials.
- Complexity bounds on evaluation/interpolation.
- Reverse mode in automatic differentiation.

Other motivations

"The author has not been able to find an example of a linear operator that is easy to apply but whose transpose is difficult to apply."

[Wiedemann '86]

"The transposition principle is very useful for proving the existence of algorithms, but actually coming up with an explicit, practical algorithm requires a bit more effort."

[Shoup '95]

"We offer no other proof of correctness other than the validity of this transformation technique (and the fact that it does indeed work in practice)."

[Shoup '95]

"Oulala ! Vous avez encore utilisé votre magie noire !"

François Morain, personal communication

Other motivations

```
void reduc_doit(GF2X& A0, GF2X& A1, const GF2X& A,
long init, long d, bool plusone){
  if (d <= 2){
    A0 = GF2X(0, coeff(A,init));
    A1 = GF2X(0, coeff(A,init+1));
    return;
  }

  long dp = d/2;
  GF2X A10, A11;

  reduc_doit(A0, A1, A, init, dp, plusone);
  reduc_doit(A10, A11, A, init+dp, dp, plusone);

  ShiftAdd(A0, A11, 1);
  if (plusone) A0 += A11;
  A1 += A10 + A11;

  long i = 1;
  bool even = true;
  while (2*i != d){
    ShiftAdd(A0, A10, i);
    ShiftAdd(A1, A11, i);
    i = 2*i;
    even = !even;
  }

  if (plusone && !even) {
    A0 += A10;
    A1 += A11;
  }
}
```

```
void treduc_doit(GF2X& A, const GF2X& A0, const GF2X& A1, long d,
bool plusone){
  if (d <= 2){
    SetCoeff(A, 0, coeff(A0, 0));
    SetCoeff(A, 1, coeff(A1, 0));
    return;
  }

  long dp = d/2;
  long hdp = dp/2;

  GF2X A00, A01, A10, A11;
  A00 = trunc(A0, hdp);
  A01 = trunc(A1, hdp);

  A10 = A01;
  if (plusone) A11 = A00;
  else A11 = 0;
  A11 += A01 + RightShift(trunc(A0, hdp+1), 1);
  long i = 1;
  bool even = true;
  while (2*i != d){
    A10 += RightShift(trunc(A0, hdp+i), i);
    A11 += RightShift(trunc(A1, hdp+i), i);
    i = 2*i;
    even = !even;
  }

  if (plusone && !even) {
    A10 += trunc(A0, hdp);
    A11 += trunc(A1, hdp);
  }

  GF2X B0, B1;
  treduc_doit(B0, A00, A01, dp, plusone);
  treduc_doit(B1, A10, A11, dp, plusone);
  A = B0 + LeftShift(B1,dp);
}
```

Other motivations

In developing transposed code for our ISSAC '09 paper, a very tricky mistake slowed down performances by more than a constant factor. The bug was so subtle that in the first place we didn't even think there was one; a machine wouldn't have made the mistake.

A striking similarity with reversible computation and quantum circuits.

A never (so I thought) enough studied relationship with automatic differentiation.

A long history of re-discoveries and many different formulations.

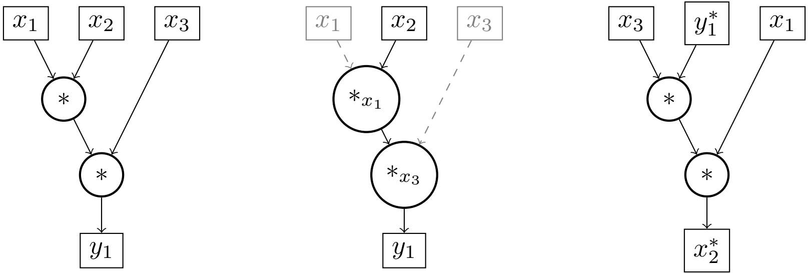
History

History

- Originally discovered in *electrical network theory* [Bordewijk '56] (only works for $\mathbb{C}$);
- [Bürgisser, Clausen, Shokrollahi] attribute the discovery to Tellegen, Bordewijk's director, but this is debated;
- Graph-theoretic approach by Fettweis (1971) for *digital filters*;
- [Fiduccia '73] and [Hopcroft, Musinski '73]: transposition of *bilinear chains*, the most complete formulation (non-commutative rings);
- Special case of *automatic differentiation* [Baur, Strassen '83];
- In *computer algebra*, popularized by Shoup, von zur Gathen, Kaltofen,...
- [Bostan, Lercerf, Schost '03] improve algorithms for polynomial evaluation and solve an open question on space complexity.

Plan

Multilinearity



- Almost anytime we want to transpose, we end-up *linearising* a circuit with multiplication nodes.
- Other constructs such as `if` statements and `for` loops need to be linearised too.

Circuit emulation

```
karatsuba [] y n = []

karatsuba x [] n = []

karatsuba x y n =
  if n <= 0
  then []
  else if n == 1
       then [(x!!0) * (y!!0)]
       else
         let h = n / 2 in
         let (a0, a1) = split x h in
         let (b0, b1) = split y h in
         let x0 = karatsuba a0 b0 h in
         let x2 = karatsuba a1 b1 (n-h) in
         let xx1 = karatsuba (a1 + a0) (b1 + b0) (n-h) in
         let x1 = xx1 + ((x0 + x2) * (- one)) in
         (shift x2 n) + (shift x1 h) + x0
```

Circuit emulation

```
karatsuba x n =
  if n <= 0
  then []
  else if n == 1
       then \y -> [(x!!0) * (y!!0)]
       else
         let h = n / 2 in
         let (a0, a1) = split x h in
         let x0 = karatsuba a0 h in
         let x2 = karatsuba a1 (n-h) in
         let xx1 = karatsuba (a1 + a0) (n-h) in
         let sp = \y -> split y h in
         let sh1 = \y -> shift y n in
         let sh2 = \y -> shift y h in

         \y -> ....
```

Circuit emulation

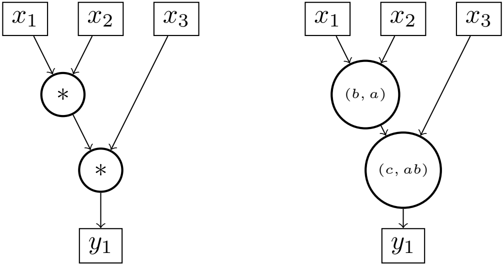
```
let h = n / 2 in
let (a0, a1) = split x h in
let x0 = karatsuba a0 h in
let x2 = karatsuba a1 (n-h) in
let xx1 = karatsuba (a1 + a0) (n-h) in
let sp = Circuit(\y -> split y h) in
let sh1 = Circuit(\y -> shift y n) in
let sh2 = Circuit(\y -> shift y h) in

proc y -> do
  (y0, y1) <- sp -< y
  s0 <- x0 -< y0
  s2 <- x2 -< y1
  ss1 <- (id <+> id).xx1 -< (y0, y1)
  s1 <- id <+> ((id <+> id) <*> (- one)) -< (ss1, (s0, s2))
  z <- sh1 <+> sh2 <+> id -<  (s2, (s1, s0))
  returnA -< z
```

Plan

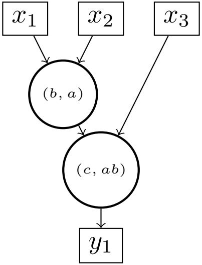
Automatic differentiation of circuits [Baur, Strassen '83]

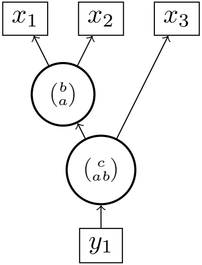
Transformation technique on circuits $R^n \to R^m$:
node $\mapsto$ its Jacobian at point x
(we are overly vulgarising, there's more than that in reality)



- By the chain rule, the result computes the Jacobian of the circuit.
- Evaluating on a vector gives a directional derivative
- n directional derivatives yield the whole Jacobian.

Reverse mode





Reverse mode

- After transposition, m directional derivatives suffice to compute the Jacobian,
- In particular, for a map $R^n \to R$, only one evaluation is needed to compute the gradient.

Transposing with AD tools

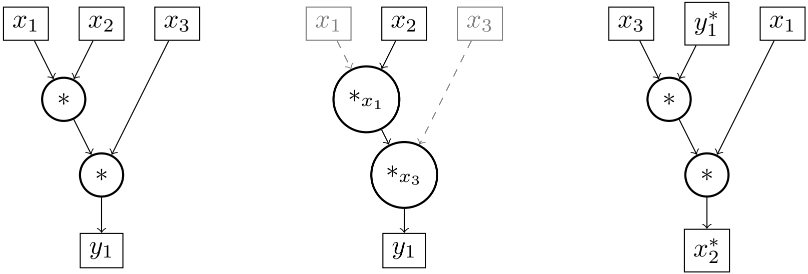
AD tools work on straight line programs, they implicitly implement transposition in reverse mode

But !

- They cost too much in space, because they have to precompute the circuit,
- Iterative statements can cause memory swell,
- They are only useful in the case $R^n \to R$,
- They can't handle multilinearity,
- They can't handle recursive calls (as far as I have seen).

Plan

Multilinearity



- Can we automatically deduce any possible linearisation of a program?
- Type inference systems can help us

Linearity inference

Suppose given a type `R` implementing a ring. We want to define types `L` (for *linear*) and `S` (for scalar) such that the following equations hold

```
plus :: L -> L -> L
plus :: S -> S -> S
times :: L -> S -> L
times :: S -> S -> S
zeroR :: L
zeroR :: S
oneR :: S
```

Linearity inference

Here's the solution

```
data L = L R
data S = S R

class Ring r where
  zero :: r
  (<+>) :: r -> r -> r
  neg :: r -> r
  (<*>) :: r -> S -> r

one = S oneR
(S a) == (S b) = a == b
```

Let's check it in a shell

Plan

The Transposable Algebraic Language

Algebraic types

- **Prototypes**: `Ring`, `Module`, (optionally `Algebra`, . . .)
- Declaring an algebraic type:

```
type Ring R
type Module(R) M
```

Declaring a function

```
fun (linear M A, const m)f(linear M Z, const M z, const n):
```

Other constructs

- Standard types (`int`, `bool`, . . .)
- `if`, `match`, recursion, `let` binding,
- Algebraic operators $+$, $\times$, projection/injection `a[n]`.

Automatic transposition: the general algorithm

```
fun (linear R res)scalar(linear M a, const n):
  if n = 0:
    res = 0
  else:
    res = a[n] + scalar(a, n-1)
```

The algortithm

- First run the algorithm in the normal direction to compute all the const values,
- then run the algorithm backwards transposing each instruction.

```
fun (linear M a)scalar^T(linear R res, const n):
  if n = 0:
    nop
  else:
    a[n] = res
    a += scalar^T(res, n-1)
```

Scalar prediction and tail recursion

- Permuting the order of the instructions may break tail/head recursion,
- this implies loss of efficiency,
- equivalently, in `for` loops we have to precompute all the `const` values of the loop,
- this seems to increase the space requirements of the algorithm, but does not affect the number of arithmetic operations.

Scalar prediction and tail recursion

```
fun (R a, R b)f(R c, R d):
  if d > 0:
    x, y = f(c, d - 1)
    a, b = x * y, y + 1
  else:
    a, b = c, d
```

```
fun (R c, R b)fT(R a, R d):
  # Forward sweep
  if (d > 0):
    _, y = f(a, d - 1)
    b = y + 1
  else:
    b = d

  # Reverse sweep
  if (d > 0):
    x = a * y
    c, y = fT(x, d - 1)
  else:
    c = a
```

Plan

A Python implementation of TransAL

`http://transalpyne.gforge.inria.fr/`

- Compiler/interpreter written in python,
- python-like syntax,
- automated constness inference,
- smart handling of array sizes,
- will compile to other languages (Haskell? OCaml?).

Karatsuba in `transalpyne`

```
def (M c)karatsuba(M a, M b, n):
  if n == 1:
    tmp = M.zero()
    tmp[0] += a[0]*b[0]
    c = tmp
  elif n > 1:
    a0, a1 = split(a, n/2, n)
    b0, b1 = split(b, n/2, n)
    x0 = karatsuba(a0, b0, n/2)
    x2 = karatsuba(a1, b1, n - n/2)
    x1 = karatsuba((a1 + a0), (b1 + b0), n - n/2) - x0 - x2
    c = shift(x2, n, n+1) + shift(x1, n/2, n+1) + x0
```

Karatsuba in transalpyne

```
(M b)karatsubaT(M a, M c, n)
  # Forward sweep
  if (n == 1):
    pass
  elif n > 1:
    a0, a1 = split(a, n / 2, n)
  # Reverse sweep
  if (n == 1):
    tmp = c
    _transAL_tmp_0[0] += a[0] * tmp[0]
    b = _transAL_tmp_0
  elif n > 1:
    x2 = trans shift(c, n, n + 1)
    x1 = trans shift(c, n / 2, n + 1)
    x0 = c
    b1 = trans karatsuba(x1, a1 + a0, n - n / 2)
    b0 = b1
    x0 += - x1
    x2 += - x1
    b1 += trans karatsuba(x2, a1, n - n / 2)
    b0 += trans karatsuba(x0, a0, n / 2)
    b = trans split(b0, b1, n / 2, n)
```

Bibliography

W. Baur and V.Strassen.
The complexity of computing partial derivatives.
Theoretical Computer Science 22, pp. 317–330, 1983.

J. L. Bordewijk.
Inter-reciprocity applied to electrical networks
Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods 6, pages 1–74, 1956.

A. Bostan, G. Lecerf & E. Schost,
Tellegen's Principle into Practice.
Proceedings of ISAAC 2003.

P. Bürgisser, M. Clausen & M. A. Shokrollahi,
Algebraic Complexity Theory.
Springer, 1997.

L. De Feo and É. Schost.
Fast arithmetics in Artin-Schreier towers over finite fields.
In *ISSAC'09*, pages 127–134. ACM, 2010.

Bibliography

C. M. Fiduccia.
On the algebraic complexity of matrix multiplication
PhD Thesis, Brown University, 1973.

J. Hopcroft and J. Musinski.
Duality applied to the complexity of matrix multiplication and other bilinear forms.
SIAM Journal on Computing, vol. 2, pp. 159–173, 1973.

E. Kaltofen.
Challenges of symbolic computation: my favorite open problems.
Journal of Symbolic Computation, 29(6):891–919, 2000.

V. Shoup.
A new polynomial factorization algorithm and its implementation.
J. Symb. Comp., 20(4):363–397, 1995.

D. H. Wiedemann.
Solving Sparse Linear Equations Over Finite Fields.
IEEE Trans. Inf. Theory, vol. IT-32:54–62, 1986.