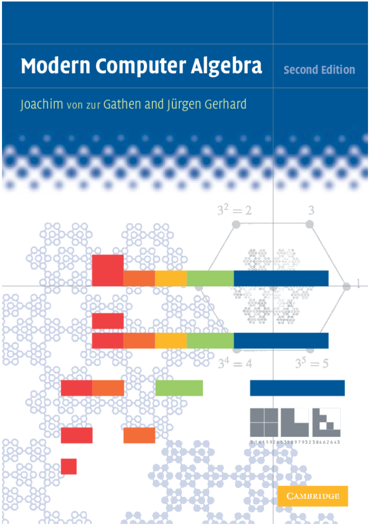
transalpyne: a language for automatic transposition

L. De Feo[1] and É. Schost[2]
(part of this talk is joint work with M. Boespflug[1])

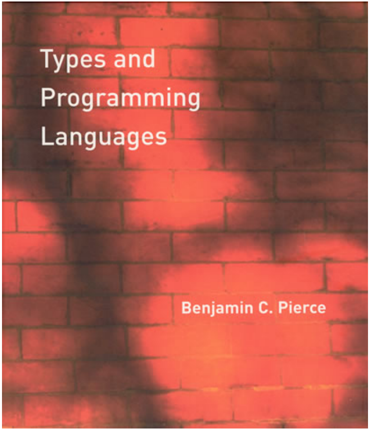
[1]LIX, École Polytechnique
[2]SCL, University of Western Ontario

PLMMS @CICM 2010
Conservatoire National des Arts et Métiers
Paris, July 8, 2010

Or, "The day I discovered I suffer from schizophrenia"



Mr. Jekyll is a computer algebraist (he'll eventually become Dr.)



Mr Type wastes precious time committed to thesis writing, by reading about types and categorical semantics

Matrices represented by computer programs?!

The black box model

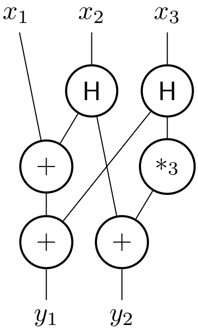
If A is a sparse or structured matrix, it is cheaper to restrict to algorithms that only query a black box:

$$b \longrightarrow \blacksquare \longrightarrow A \cdot b$$

Applications

- The good ol' Power iteration method to find the largest eigenvalue. Used by Google page ranking algorithm [Page, Brin, Motwani, Winograd '99].
- Wiedemann's algorithms for minimal/characteristic polynomial, determinant, rank, inversion. [Wiedemann '86]
- ...

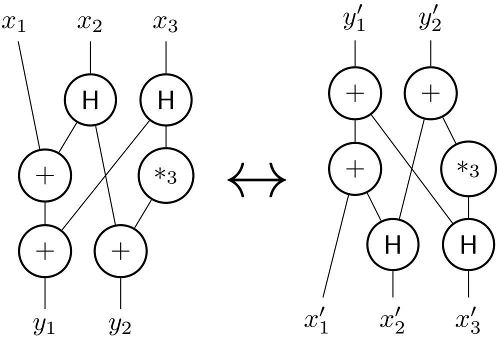
Arithmetic circuits



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

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

Transposition of an arithmetic circuit



$$y_1 = x_1 + x_2 + x_3$$

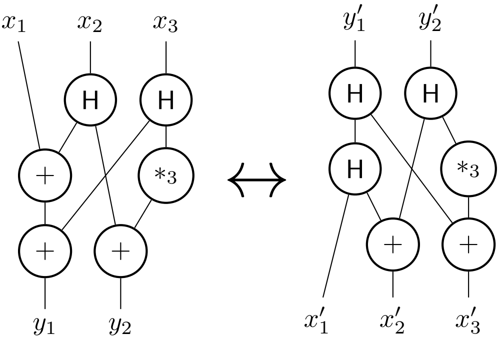
$$y_2 = x_2 + 3x_3$$

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

$$\updownarrow$$

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

Transposition of an arithmetic circuit



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

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

Transposition of straight line programs

Straight line programs $=$ Arithmetic 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
```

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

```
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]
```

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

What is transposition of programs useful for?

Power projection: $(\mathbb{K}/k)^* \to k[X]^*$

$$\ell \mapsto \sum_{i>0} \ell(\sigma^i) X^i$$

Modular composition: $k[X] \to \mathbb{K}/k$

$$g \mapsto g(\sigma)$$

Power projection = transposed modular composition

- 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 of transposition

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

Why *automatic* transposition?

```
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);
}
```

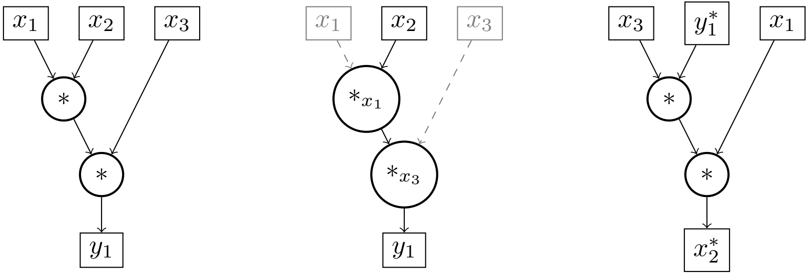
Why *automatic* transposition?

- Algorithms are hard to transpose, transposed algorithms are hard or impossible to understand;
- How to be confident that a transposed algorithm is well implemented if no one understands it?
- When proving programs with a proof assistant, why should we do the work twice?

Previous work

- Originally discovered in *electrical network theory* [Bordewijk '56] (only works for $\mathbb{C}$); some authors attribute the discovery to Tellegen, Bordewijk's director, but this is debated;
- [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.

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.
- 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 -> L -> L
times :: S -> S -> S
```

```
zero :: L
zero :: S
```

```
one :: S
```

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
```

$$\forall \alpha \in \{L, S\}.\alpha \to \alpha \to \alpha$$

```
times :: L -> S -> L
times :: S -> L -> L
times :: S -> S -> S
```

$$\forall \alpha \in \{L, S\}.\alpha \to S \to \alpha$$

```
zero :: L
zero :: S
```

$$\forall \alpha \in \{L, S\}.\alpha$$

```
one :: S
```

Linearity inference

The solution in Haskell

```
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
```

To treat `times :: S -> L -> L`, we extend the Hindley-Milner type inference to handle lists of acceptable unifications.

transalpyne

Algebraic types

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

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

Declaring a function

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

Other constructs

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

Automatic transposition: the scalars first!

```
def (linear R c)f(linear R a, const R b):
  d = b * b
  c = a * d
```

- Run the algorithm backwards transposing each instruction.

Wrong!

```
def (linear R a)fT(linear R c, const R b):
  a = c * d
  d = b * b
```

Automatic transposition: the scalars first!

```
def (linear R c)f(linear R a, const R b):
  d = b * b
  c = a * d
```

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

```
def (linear R a)fT(linear R c, const R b):
  # Forward sweep
  d = b * b

  # Reverse sweep
  a = c * d
```

Automatic transposition: if's and function calls

```
def (linear M a)f(linear M b, n):
  if n > 0:
    a = f(b, n - 1)
    a[n] += R.Z(n) * b[n]
```

- if's stay the same, the values appearing in the test must be scalar,
- (recursive) functions get their linear input and output parameters swapped, scalar arguments do not move.

```
def (linear M b)fT(linear M a, n):
  # Reverse sweep
  if n > 0:
    b[n] += R.Z(n) * a[n]
    b += fT(a, 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 scalar values of the loop,
- this seems to increase the space requirements of the algorithm, but does not affect the number of arithmetic operations.

Lazy evaluation in the forward sweep

```
def (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
```

```
def (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
```

Lazy evaluation in the forward sweep

```
def (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
```

```
def (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
```

Conclusion

What we achieved

- Transposition of multilinear/recursive code.
- An (almost complete) python implementation of transposition in the form of a compiler/interpreter.
- `transalpyne` can be easily used on top of CAS that have a python interface.
- Other CAS will be able to use `transalpyne` as we will add more languages to the output of the compiler (OCaml and Haskell look easy, C is somewhat harder).

Limitations

- We trust the user not to introduce side effects.
- No formal proof of correctness... Do I trust my compiler?
- It would be nice to have types checked statically $\rightarrow$ Implementation as a Haskell extension?

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 -
    c = shift(x2, n, n+1) + shift(x1, n/2, n+1) + x0
```

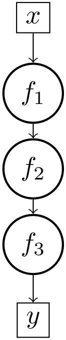
Karatsuba in transalpyne

```
(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)
```

Proof of the transposition theorem

$$
\begin{tikzcd}
\boxed{x} \arrow[d] \\
f_1 \arrow[d] \\
f_2 \arrow[d] \\
f_3 \arrow[d] \\
\boxed{y}
\end{tikzcd}
$$

$$(M_{f_1} M_{f_2} \cdots M_{f_n})^T = M_{f_n}^T \cdots M_{f_2}^T M_{f_1}^T$$

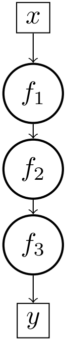


$$\begin{tikzcd} A \arrow[d, "f_1"] \\ A \arrow[d, "f_2"] \\ A \arrow[d, "f_3"] \\ A \end{tikzcd}$$

$$(M_{f_1} M_{f_2} \cdots M_{f_n})^T = M_{f_n}^T \cdots M_{f_2}^T M_{f_1}^T$$

Transposition is a contravariant functor

Proof of the transposition theorem



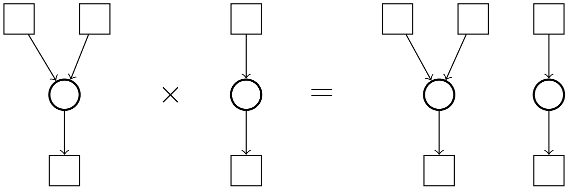
$$
\begin{tikzcd}
 & A \arrow[dl, "f_1"'] \arrow[dr, "g_1"] & \\
A \arrow[d, "f_2"'] & & A \arrow[d, "g_2"] \\
A \arrow[dr, "f_3"'] & & A \arrow[dl, "g_3"] \\
 & A &
\end{tikzcd}
$$

$$(M_{f_1} M_{f_2} \cdots M_{f_n})^T = M_{f_n}^T \cdots M_{f_2}^T M_{f_1}^T$$

Transposition is a contravariant functor

Categorical semantics

- Diagrams have a *semantic* in any category,
- Circuits have a *semantic* in any Cartesian category.



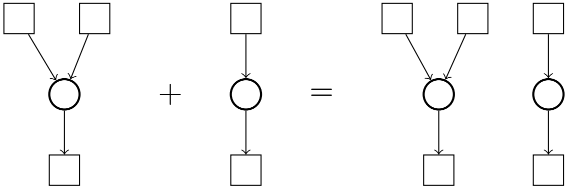
- Haskell's `Control.Arrow` calls this operation `***`,
- `Control.Arrow` also gives `&&&`, which is akin to our H port.

Semantics and cosemantics

- Transposition is just applying the functor to circuits (diagrams).
- But, wait a minute, transposition is contravariant (and continuous and cocontinuous)!

$$\prod \stackrel{?}{=} \coprod$$

- Not in general: circuits have another *semantic* in any Cocartesian category. Call it *cosemantic*.



- Haskell's `ArrowChoice` calls this operation `+++`.
- `ArrowChoice` also gives `|||`, which is akin to our $+$ port.

Additive categories

But $R\mathrm{-}\mathbf{Mod}$ is an additive category:

$\times \simeq +$ $\quad\Rightarrow\quad$ cosemantic $\simeq$ semantic

Can we implement a type class synthesising these properties?

```
class AdditiveArrow (~>) where
  (&&&) :: (a ~> b) -> (a ~> c) -> (a ~> (Plus b c))
  (|||) :: (a ~> c) -> (b ~> c) -> ((Plus a b) ~> c)
  (***) :: (a ~> b) -> (c ~> d) -> ((Plus a c) ~> (Plus b d))
```

Additive categories

But $R{-}\mathbf{Mod}$ is an additive category:

$\times \simeq + \quad \Rightarrow \quad$ cosemantic $\simeq$ semantic

Can we implement a type class synthesising these properties?

```
class AdditiveArrow (~>) where
  (&&&) :: (a ~> b) -> (a ~> c) -> (a ~> (Plus b c))
  (|||) :: (a ~> c) -> (b ~> c) -> ((Plus a b) ~> c)
  (***) :: (a ~> b) -> (c ~> d) -> ((Plus a c) ~> (Plus b d))
```

We tried, but we failed!

(this is essentially due to the limited support for dependent types in Haskell)

Future work

- Earn Mr. Jekyll a doctoral degree.
- Finish the implementation of `transalpyne` and release it at `http://transalpyne.gforge.inria.fr/`.
- Write `AdditiveArrow` in CoqMT.


The End?

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.

L. Page, S. Brin, R. Motwani and T. Winograd.
The PageRank Citation Ranking: Bringing Order to the Web.
Technical Report, Stanford InfoLab, 1999,
`http://ilpubs.stanford.edu:8090/422/`.

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.