Laboratoire d'Informatique de l'X
École Polytechnique
route de Saclay,
91128 Palaiseau, France

`transalpyne`: a language for automatic transposition

Luca De Feo
LIX, École Polytechnique

Éric Schost
CSD, University of Western Ontario

ISSAC 2010, Munich, July 25–28

Computer Science Department
University of Western Ontario
Middlesex College
London, Ontario, Canada, N6A 5B7

1 Introduction

The *transposition principle* says that by "reversing" the flow of a *linear arithmetic circuits* one obtains a circuit that computes the transposed map.

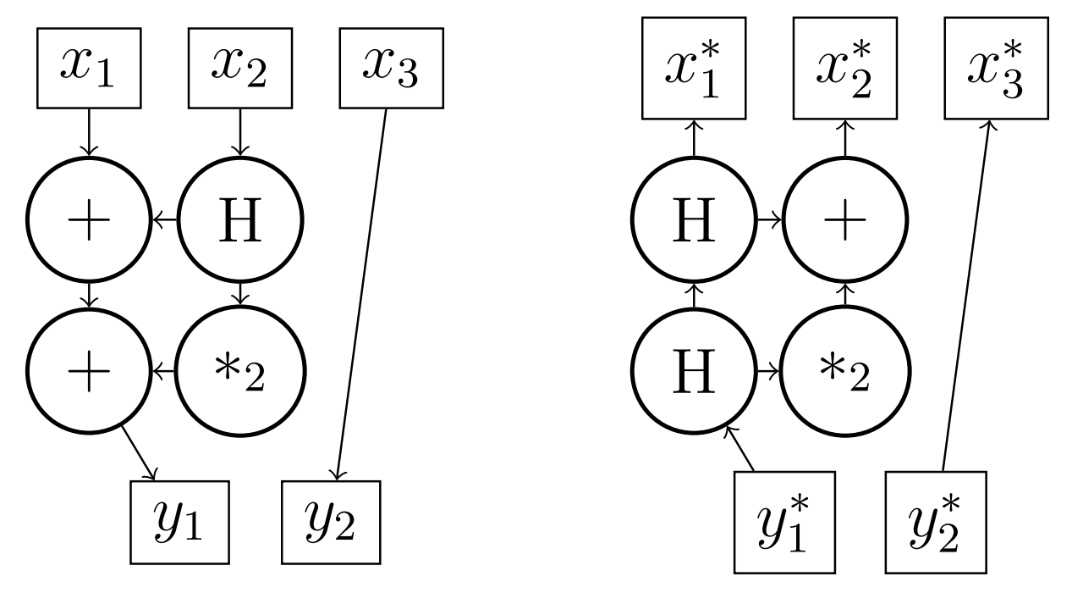

Figure 1: Two linear arithmetic circuits. The linear map $y_1 = x_1 + 3x_2, y_2 = x_3$ is computed by the circuit on the left and its transpose is computed by the circuit on the right.

1.1 History of the transposition principle

- Originally discovered by Bordewijk [1] in *electrical network theory* (only works for $\mathbb{C}$); some authors attribute the discovery to Tellegen, Bordewijk's director, but this is debated;
- Fiduccia [2] and Hopcroft & Musinski [3]: transposition of *bilinear chains*, the most complete formulation (non-commutative rings);
- Special case of Baur & Strassen's *differentiation of circuits* [4];
- In *computer algebra*, popularized by Shoup, von zur Gathen, Kaltofen [5],. . .
- Bostan, Lecerf & Schost [6] improve algorithms for polynomial evaluation and solve an open question on space complexity.

1.2 Why is it useful?

Let $\mathbb{K}/k$ be a field extension and let $\mathbb{K}^*$ be the dual space of $\mathbb{K}$. Fix $\sigma \in \mathbb{K}$ and consider the k-linear maps

$$\begin{aligned} C_\sigma : k[X] &\to \mathbb{K}, & P_\sigma : \mathbb{K}^* &\to k[[X]], \\ g &\mapsto g(\sigma), & \ell &\mapsto \sum_{i>0} \ell(\sigma^i)X^i. \end{aligned}$$

If we identify $k[[X]]$ to the dual space of $k[X]$, these are one other's dual. P_σ is called the *power projection.* When σ is algebraic, $\mathbb{C}_\sigma$ is called the *modular composition* because its is computed as $g \circ h \bmod f$ where h is a representation for σ in $k[X]/f(X)$.

- Modular composition is a well-known problem; the most famous algorithm is Brent and Kung's [7],
- Shoup [8] was the first to realize that applying the transposition principle to modular composition could yield efficient algorithms for power projection.
- As a consequence, one can compute efficiently
  - minimal polynomials in towers of extensions [8, 5],
  - change of order in triangular sets [9],
  - arithmetics in Artin-Schreier towers [10].
- Other applications of the transposition principle include
  - generation of irreducible polynomials [11],
  - complexity bounds on evaluation/interpolation [6],
  - reverse mode in automatic differentiation [12].

1.3 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?

2 Methods

2.1 Linearization

Algorithms usually involve non linear operations, the most common being multiplication. Arithmetic circuits involving multiplication nodes can be *linearized* by *fixing* some nodes as parameters, then the transposition principle can be applied to the linearized circuit. Of course, there may be more than one linearization for a given circuit.

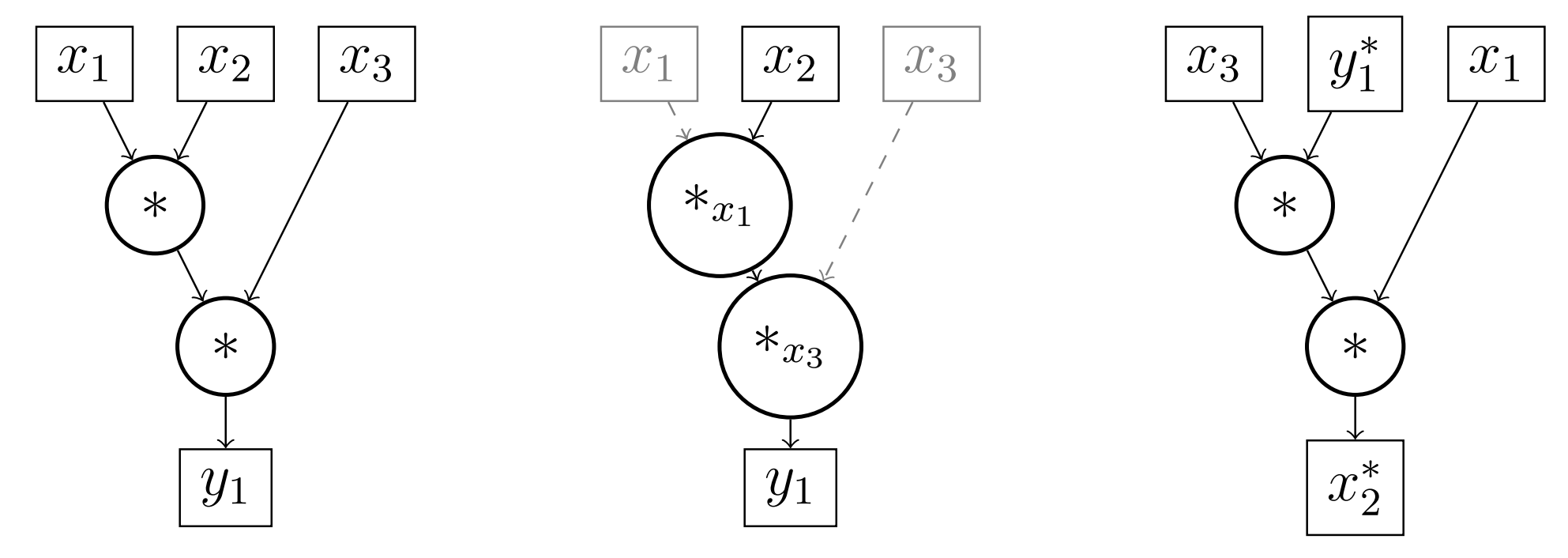

Figure 2: A linearization of a multilinear circuit and its transposition.

Similarly, computer programs containing multiplication, conditionals, loops, etc. may be linearized by partially evaluating their parameters. The result is a straight line program that can be transposed.

2.2 Linearity inference

All the possible linearizations of a computer program can be inferred automatically by type checking, starting from the signature of the elementary functions $0, 1, +, \times$, etc.

We define the type ℓ of elements used linearly and the type s of elements used as scalars. We give the following quantified types to the elementary functions

$$\begin{aligned} 0 &: \forall\alpha \in \{\ell, s\}.\alpha & 1 &: s \\ + &: \forall\alpha \in \{\ell, s\}.\alpha \to \alpha \to \alpha & & \\ \times &: \forall\alpha \in \{\ell, s\}.\alpha \to s \to \alpha & \times &: s \to \ell \to \ell \end{aligned}$$

It is easy to verify that for any validly typed program, the program obtained by partially evaluating all the arguments having type different from ℓ is a linearized program.

If we forget the type $\times : s \to \ell \to \ell$, all these type constraints can be expressed in Haskell's type system as

```
data L = L R;    data S = S R
class Ring r where
  zero :: r
  plus :: r -> r -> r
  neg :: r -> r
  times :: r -> S -> r
instance Ring L where ...
instance Ring S where ...
one = S oneR
```

Thus linearizations can be inferred by Hindley-Milner type inference [13] in this restricted setting. A minor modification to the Hindley-Milner algorithm permits to treat the case $\times : s \to \ell \to \ell$ as well.

2.3 Transposition

Once a possible linearization has been inferred, we partially evaluate the program in what we call a *forward sweep.* This is similar to the forward sweep of reverse mode in automatic differentiation [14].

The resulting partially evaluated program is transposed going from the bottom to the top line-by-line, swapping the inputs and the outputs of each function as in [6].

The algebraic time complexity of the resulting transposed program is the same as for the original one. There is an increase in space complexity due to the need to store data for the partial evaluation; however this increase tends to be relatively small since usual programs have few non-linear variables.

3 `transalpyne`

We are currently implementing our ideas in a python-like language called `transalpyne`. Its main features are

- python-like syntax,
- dynamically typed, except for algebraic variables,
- pure functional semantics,
- linearity inference, automatic transposition,
- on-the-fly interpretation inside python, or compilation to python,
- open source CeCILL licence.

The first release will be distributed at the url `http://transalpyne.gforge.inria.fr/`. As an *avant-goût* we give here an implementation of Karatsuba's algorithm in `transalpyne` and the generated transposed code.

```
def (M c)karatsuba(const 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
```

```
def (M b)_transAL_T_karatsuba(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)
```

References

[1] J. Bordewijk. Inter-reciprocity applied to electrical networks. *Applied Scientific Research, Section B*, 6(1):1–74, December 1957.

[2] C. M. Fiduccia. *On the algebraic complexity of matrix multiplication.* PhD thesis, Providence, RI, USA, 1973.

[3] J. Hopcroft and J. Musinski. Duality applied to the complexity of matrix multiplications and other bilinear forms. In *STOC '73*, pp. 73–87, 1973. ACM.

[4] W. Baur and V. Strassen. The complexity of partial derivatives. *Theor. Comput. Sci.*, 22(3):317–330, 1983.

[5] E. Kaltofen. Challenges of symbolic computation: my favorite open problems. *J. Symb. Comput.*, 29(6):891–919, 2000.

[6] A. Bostan, G. Lecerf, and Schost. Tellegen's principle into practice. In *ISSAC '03*, pp. 37–44, 2003. ACM.

[7] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *J. ACM*, 25(4):581–595, 1978.

[8] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comput.*, 20(4):363–397, 1995.

[9] C. Pascal and É. Schost. Change of order for bivariate triangular sets. In *ISSAC '06*, pp. 277–284, 2006. ACM.

[10] L. De Feo and É. Schost. Fast arithmetics in artin-schreier towers over finite fields. In *ISSAC '09*, pp. 127–134, 2009. ACM.

[11] J. F. Canny, E. Kaltofen, and L. Yagati. Solving systems of non-linear polynomial equations faster. In *ISSAC '89*, pp. 121–128, 1989. ACM.

[12] S. B. Gashkov and I. B. Gashkov. On the complexity of calculation of differentials and gradients. *Discr. Math. Appl.*, 15(4):327–350, 2005.

[13] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82*, pp. 207–212, 1982. ACM.

[14] A. O. Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pp. 83–108. Kluwer Academic Publishers, 1989.