Transposition Principle and Applications

L. De Feo

Projet TANC, LIX, École Polytechnique

Université Paul Sabatier, Toulouse
June 18, 2009

"From every *linear algorithm* computing a linear application we can deduce another *linear algorithm* computing the transpose application using *about* the same space and time resources."

History, motivations

History

- Originally discovered by Tellegen (1950), Bordewijk (1956) for *electrical network theory* and by Kalman (1960) for *control theory*;
- Graph-theoretic approach by Fettweis (1971) for *digital filters*;
- Fiduccia (1972): transposition of *bilinear algorithms*;
- Special case of reverse mode in *automatic differentiation*: Baur & Strassen (1983);
- In *computer algebra*, popularized by Shoup, von zur Gathen, Kaltofen,...
- [Bostan, Lercerf, Schost '03] improve algorithms for polynomial evaluation.

Motivations

- Existence result in *complexity theory*;
- *Code transformation* technique;
- Improve $M^T \Leftrightarrow$ Improve M;
- **Divides by 2 the number of algorithms yet to be discovered.**

Classical proofs

Linear algebra

M computed as a sequence of *simple* linear applications U_i

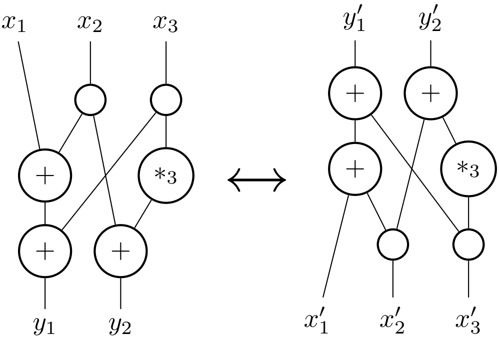
$$M(v) = U_1 \circ U_2 \circ \cdots \circ U_n(v) \qquad \Leftrightarrow \qquad M^T(v) = U_n^T \circ \cdots \circ U_2^T \circ U_1^T$$

Graph-theoretic approach

- *Compile* the algorithm in a DAG;
- reverse the arrows of the DAG.

This works only for straight-line programs !

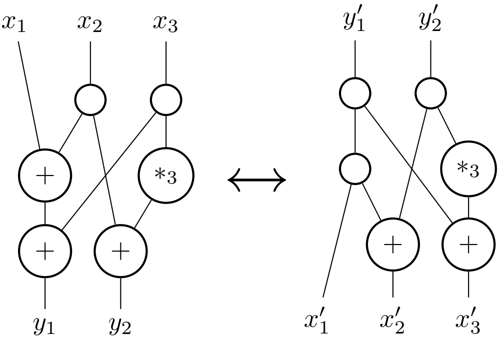
Graph-theoretic approach (cont'd)



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

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

Graph-theoretic approach (cont'd)



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

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

Category theory, a gentle introduction...

- Category $\mathscr{C}$
- Objects $\mathrm{ob}(\mathscr{C})$

$$A \qquad B$$

$$C$$

$$D \qquad E$$

Category theory, a gentle introduction...

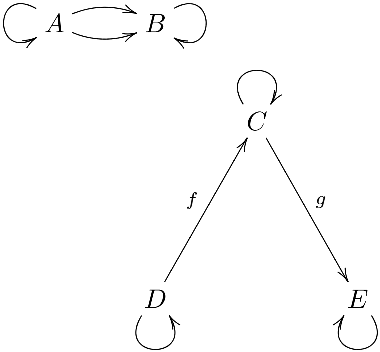
- Category $\mathscr{C}$
- Objects $\mathrm{ob}(\mathscr{C})$
- Arrows $\hom(\mathscr{C})$, $\mathrm{Hom}(A, B)$

$$\begin{tikzcd} A \arrow[r, bend left] \arrow[r, bend right] & B \end{tikzcd}$$

$$\begin{tikzcd} & C \arrow[dr, "g"] & \\ D \arrow[ur, "f"] & & E \end{tikzcd}$$

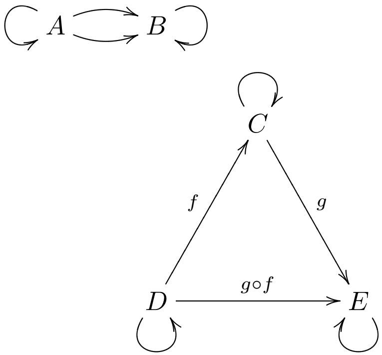
Category theory, a gentle introduction...

- Category $\mathscr{C}$
- Objects $\mathrm{ob}(\mathscr{C})$
- Arrows $\mathrm{hom}(\mathscr{C})$, $\mathrm{Hom}(A, B)$
- Identities id_A



Category theory, a gentle introduction...

- Category $\mathscr{C}$
- Objects $\mathrm{ob}(\mathscr{C})$
- Arrows $\hom(\mathscr{C})$, $\mathrm{Hom}(A, B)$
- Identities id_A
- Composition $g \circ f$



Category theory, a gentle introduction...

- Category $\mathscr{C}$
- Objects $\mathrm{ob}(\mathscr{C})$
- Arrows $\hom(\mathscr{C})$, $\mathrm{Hom}(A, B)$
- Identities id_A
- Composition $g \circ f$

$$\begin{tikzcd} A \arrow[loop left] \arrow[r, bend left] \arrow[r, bend right] & B \arrow[loop right] \end{tikzcd}$$

$$\begin{tikzcd} & C \arrow[loop above] \arrow[dr, "g"] & \\ D \arrow[loop below] \arrow[ur, "f"] \arrow[rr, "g\circ f"] & & E \arrow[loop below] \end{tikzcd}$$

Example : $\mathbf{FMod}_R$

- $\mathrm{ob}(\mathscr{C}) = R^n$ free R-modules,
- $\hom(\mathscr{C}) =$ linear applications.

Category theory, Functors

Covariant functor
$F : \mathscr{C} \to \mathscr{D}$

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\ & C \end{tikzcd} \quad \mapsto \quad \begin{tikzcd} F(A) \arrow[r, "F(f)"] \arrow[dr, "F(h)"'] & F(B) \arrow[d, "F(g)"] \\ & F(C) \end{tikzcd}$$

Contravariant functor
$F : \mathscr{C} \to \mathscr{D}$

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\ & C \end{tikzcd} \quad \mapsto \quad \begin{tikzcd} F(A) & F(B) \arrow[l, "F(f)"'] \\ & F(C) \arrow[ul, "F(h)"] \arrow[u, "F(g)"'] \end{tikzcd}$$

Equivalence, duality

- Equivalence if $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ covariant
- Duality if $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ contravariant

and $\quad F \circ G \simeq \mathrm{Id}_{\mathscr{D}} \quad$ and $\quad G \circ F \simeq \mathrm{Id}_{\mathscr{C}}$.

Tellegen's principle

"From every *linear algorithm* computing a linear application we can deduce another *linear algorithm* computing the transpose application using *about* the same space and time resources."

An example

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

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

Computations

Language, size

- Set of *instructions* $\mathscr{L} \subset \hom(\mathscr{C})$,
- Size function $\|.\| : \mathrm{ob}(\mathscr{C}) \to \mathbb{N}$.

Computation

Sequence $C_1 : b \to c$ of instructions.

$$\begin{tikzcd}
 & {} \arrow[r] \arrow[d, phantom, "C_2"] & {} \arrow[dr] & & {} \arrow[r] \arrow[d, phantom, "C_1"] & {} \arrow[dr] & \\
a \arrow[ur] \arrow[rrr, dashed, "f_2"'] & {} & & b \arrow[ur] \arrow[rrr, dashed, "f_1"'] & {} & & c
\end{tikzcd}$$

Time and space cost

- $\mathrm{t}(C) =$ length of the computation,
- $\mathrm{s}(C) = \max_{o \in C} \|o\|$.

The case $\mathbf{FMod}_R$

[Bostan, Lercerf, Schost '03] :

$$\begin{aligned} +_1 : R^2 &\to R^2 \\ (p,q) &\mapsto (p+q,q) \end{aligned} \qquad \begin{aligned} +_2 : R^2 &\to R^2 \\ (p,q) &\mapsto (p,p+q) \end{aligned} \qquad \begin{aligned} *_a : R &\to R \\ p &\mapsto ap \end{aligned}$$

$$\begin{aligned} \pi : R &\to 0 \\ p &\mapsto 0 \end{aligned} \qquad \begin{aligned} \iota : 0 &\to R \\ 0 &\mapsto 0 \end{aligned}$$

$$\mathscr{L} = \left\{ \mathrm{Id}_n \times \mathrm{op} \times \mathrm{Id}_m \middle| n, m \in \mathbb{N}, \mathrm{op} \in \{+_1, +_2, *_a, \pi, \iota \mid a \in R\} \right\}.$$

$$\|R^n\| = n$$

Our example

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

$$\mathrm{Id}_i \times +_2 \times \mathrm{Id}_{n-2-i}$$
$$\mathrm{Id}_i \times *_0 \times \mathrm{Id}_{n-1-i}$$

$$R^n \xrightarrow{+_2 \times \mathrm{Id}_{n-2}} R^n \xrightarrow{*_0 \times \mathrm{Id}_{n-1}} R^n \cdots\cdots\cdots\cdots R^n$$

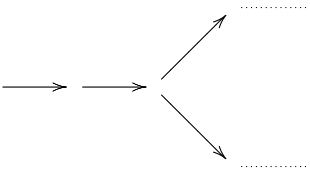
Our example

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

$$\mathrm{Id}_i \times +_2 \times \mathrm{Id}_{n-2-i}$$
$$\mathrm{Id}_i \times *_0 \times \mathrm{Id}_{n-1-i}$$

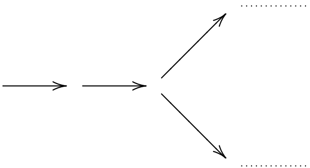
$$R^n \xrightarrow{+_2 \times \mathrm{Id}_{n-2}} R^n \xrightarrow{*_0 \times \mathrm{Id}_{n-1}} R^n \cdots\cdots\cdots\cdots R^n$$

Branchings



```
if a = (0,...,0) then
  ...
else
  ...
endif
```

Branchings



```
if n = 0 then
  ...
else
  ...
endif
```

Algorithm

Parameter space

Par a recursively enumerable set

For example, $\quad \mathrm{Par} = \mathbb{N}$

Algorithm

A function $\qquad A : \mathrm{Par} \to \mathscr{C}_{\to} \qquad (\mathscr{C}_{\to} =$ the computations$)$

Algorithm

Parameter space

Par a recursively enumerable set

For example, $\mathrm{Par} = \mathbb{N}$

Algorithm

A function $A : \mathrm{Par} \to \mathscr{C}_{\to}$ ($\mathscr{C}_{\to}$ = the computations)

1

Algorithm

Parameter space

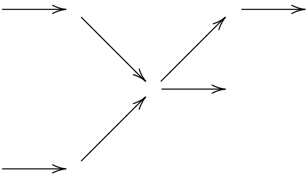
Par a recursively enumerable set

For example, $\mathrm{Par} = \mathbb{N}$

Algorithm

A function $A : \mathrm{Par} \to \mathscr{C}_{\to}$ ($\mathscr{C}_{\to}$ = the computations)

1
2



Algorithm

Parameter space

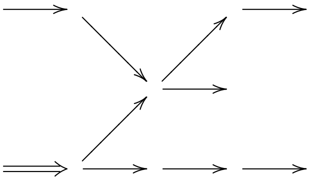
Par a recursively enumerable set

For example, $\quad \mathrm{Par} = \mathbb{N}$

Algorithm

A function $\quad A : \mathrm{Par} \to \mathscr{C}_{\to} \quad$ ($\mathscr{C}_{\to}$ = the computations)

1
2
3
$\vdots$



Complexity

Time complexity

$A : \mathrm{Par} \to \mathscr{C}_{\to}$ induces a function $\mathrm{t}_A : \mathrm{Par} \to \mathbb{N}$ given by

$$\mathrm{t}_A(x) = \mathrm{t}(A(x))$$

Space complexity

$A : \mathrm{Par} \to \mathscr{C}_{\to}$ induces a function $\mathrm{s}_A : \mathrm{Par} \to \mathbb{N}$ given by

$$\mathrm{s}_A(x) = \mathrm{s}(A(x))$$

Our example

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

$$R^n \xrightarrow{+_2 \times \mathrm{Id}_{n-2}} R^n \xrightarrow{*_0 \times \mathrm{Id}_{n-1}} R^n \cdots\cdots\cdots\cdots R^n$$

Our example

```
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
```

$$n \quad \mapsto \quad R^n \xrightarrow{+_2 \times \mathrm{Id}_{n-2}} R^n \xrightarrow{*_0 \times \mathrm{Id}_{n-1}} R^n \cdots\cdots\cdots\cdots R^n$$

Tellegen's theorem

T-functor

A functor $F : \mathscr{C} \to \mathscr{D}$ is said to be a T-functor if $F(\mathscr{L}_{\mathscr{C}}) \subset \mathscr{L}_{\mathscr{D}}$.

Tellegen's theorem

- $F : \mathscr{C} \to \mathscr{D}$ a T-functor
- Par a parameter space
- $A : \mathrm{Par} \to \mathscr{C}_{\to}$ an algorithm

$F \circ A$, noted $F(A)$ is an algorithm $\mathrm{Par} \to \mathscr{D}_{\to}$ such that

- $\mathrm{t}_{F(A)} = \mathrm{t}_A$,
- $\mathrm{s}_{F(A)} \leq B(\mathrm{s}_A)$ if $B : \mathbb{N} \to \mathbb{N}$ is an upper bound for F.

The case $\mathbf{FMod}_R$

We know a (contravariant) functor $T : \mathbf{FMod}_R \to \mathbf{FMod}_R$ given by matrix transposition.

Tellegen's theorem for linear algebra

T is a T-functor for the language $\mathscr{L}$ we gave before.

$$T(+_1) = +_2 \qquad T(*_a) = *_a \qquad T(\pi) = \iota$$

Our example

```
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]
```

Our example

```
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
```

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

Other examples: Quantum Computing

The QC category $\mathscr{Q}$

$$\mathrm{ob}(\mathscr{Q}) = \{\left(\mathbb{C}^2\right)^{\otimes n} \mid n \in \mathbb{N}\} \qquad \hom(\mathscr{Q}) = U \text{ unitaries } (U^* = U^{-1})$$

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}, \qquad R = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i} \end{pmatrix}, \qquad CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Language, size

$$\mathscr{L} = \left\{ \mathrm{Id}_2^{\otimes n} \otimes \mathrm{op} \otimes \mathrm{Id}_2^{\otimes m} \middle| n, m \in \mathbb{N}, \mathrm{op} \in \{H, R, R^*, CNOT\} \right\} \qquad \|(\mathbb{C}^2)^{\otimes n}\| = n$$

The functor

$$* : U \mapsto U^*$$

Other examples: Extension and restriction of scalars

A, B two rings, $f : A \to B$ a morphism,

- $E_f : M_A \mapsto M_A \otimes_A B$ maps A-modules to B-modules;
- $R_F : M_B \mapsto M_B$ maps B-modules to A-modules (by the law $am \equiv f(a)m$).

Extension

- Every algorithm written for $\mathbb{R}$-modules works for $\mathbb{C}$-modules;
- Every algorithm written for $\mathbb{Z}$-modules works for any module;
- Every algorithm written for $\mathbb{K}[X]$-modules works for $K[X]/P(X)$ modules.

Restriction

- Less important than extension;
- its adjoint, not its inverse (extension of scalars may loose information).

Application: transposed multiplication

Dual space

$\mathbb{L}/\mathbb{K}$ a field extension, $\mathbb{L}^*$ the space of $\mathbb{K}$-linear forms over $\mathbb{L}$

$$V \text{ basis of } \mathbb{L} \;\rightarrow\; V^* \text{ basis of } \mathbb{L}^*$$
$$\ell \in \mathbb{L}^* \;\rightarrow\; (\ell(V_0), \dots, \ell(V_n))$$

Multiplication

Let $x \in \mathbb{L}$, multiplication by x is a linear application $\mathbb{L} \rightarrow \mathbb{L}$ with matrix M_x:

$$\begin{pmatrix} \quad M_x \quad \end{pmatrix} \begin{pmatrix} y \end{pmatrix} \mapsto \begin{pmatrix} xy \end{pmatrix}$$

Transposed multiplication

Let $x \in \mathbb{L}$, $\ell \in \mathbb{L}^*$, middle product, noted $x \cdot \ell$, is the linear operation

$$\begin{pmatrix} \quad x \cdot \ell \quad \end{pmatrix} \begin{pmatrix} y \end{pmatrix} = \begin{pmatrix} \quad \ell \quad \end{pmatrix} \begin{pmatrix} \quad M_x \quad \end{pmatrix} \begin{pmatrix} y \end{pmatrix} \mapsto \begin{pmatrix} \quad \ell \quad \end{pmatrix} \begin{pmatrix} xy \end{pmatrix} = \ell(xy)$$

- Hence M_x^T is the linear application computing $x \cdot \ell$ from ℓ.
- We can transform an algorithm for multiplication into one for middle product.
- Used by [Bostan, Lercerf, Schost '03] to improve polynomial interpolation.

Application: basis change in integral extensions

Setting

Let A, B be integral domains, $B = A[b]$ integral extension, then

- B free A-module of rank $n+1$, $U = (1, b, \dots, b^n)$ a basis,
- $Q(b) = 0$, Q irreducible and separable,
- Q is the minimal polynomial of M_b,
- for $c \in B$ we set $\mathrm{Tr}(c) = \mathrm{Tr}(M_c)$,
- let $C \in A[X]$ s.t $c = C(b)$, then by linear algebra

$$\mathrm{Tr}(c) = \mathrm{Tr}(C(b)) = \mathrm{Tr}(c_0 + c_1 b + \cdots + c_n b^n) = \sum_{Q(\beta)=0} C(\beta).$$

Claim

Let V be a basis of B as an A-module and let M_V be the complexity of multiplication in this basis. From any algorithm with complexity C_V computing the change of basis $U \to V$ we can deduce an algorithm with complexity $O(C_V + M_V)$ computing the inverse change of basis.

Application: basis change in integral extensions

Trace formulae

Let $C \in A[X]$, $c = C(b)$, set $\ell_c = c \cdot \mathrm{Tr}$, then

$$\sum_{i>0} \ell_c(b^i) T^i = \frac{\operatorname{rev} \sum_{Q(\beta)=0} C(\beta) \prod_{\beta' \neq \beta} (T - \beta'^i)}{\operatorname{rev} Q(T)}$$

observe that $Q'(T) = \sum_{Q(\beta)=0} \prod_{\beta' \neq \beta} (T - \beta'^i)$, then

$$C(T) \equiv \frac{\sum_{Q(\beta)=0} C(\beta) \prod_{\beta' \neq \beta} (T - \beta'^i)}{Q'(T)} \mod Q(T)$$

The algorithm

- Computing Tr in the basis V^*. $O(M_V)$
- Knowing c in the basis V, computing $\ell_c = c \cdot \mathrm{Tr}$ is middle product. M_V
- computing the first $n+1$ coefficients of $\sum \ell_c(b^i) T^i$ is the same as move ℓ_c from V^* to U^*. C_V
- Truncated multiplication, Newton inversion, modular multiplication. $O(M_V)$

Conclusions

Towards theory

- Is this new point of view more enlightening?
- Are there any other interesting T-functors?

Towards practice

- Is it always possible to find a good parameter space?
- How much does the non-linear code (conditionals and loops) change when we transpose?
- Can we dream of a working AT tool?

Automatic transposition of code

```
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);
}
```

Automatic transposition of code

I want a compiler that automatically transposes my code !

The problem

- Fix two computational categories and a T-functor F,
- the languages have to be reasonable,
- composition has to be trivial,
- deciding the image of an instruction by F must be feasible.

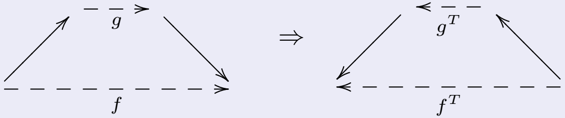
Straight line programs

- Easy.
- Read the program upside-down or bottom-up (depending if the F is covariant or contravariant),
- substitute each instruction with its dual.

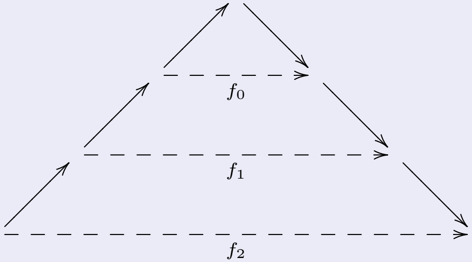
Subroutines

Subroutines

```
f(x, y, z) {
  ...
  a = g(x, y);
  ...
}
```



Recursion



Conditionals, loops

Conditionals

```
if n > 0 then
  ...
else
  ...
endif
```

- n **must be** in the parameter space,
- the conditional is left unchanged.

Loops

```
for i = 0 to n do
  ...
end for
```

- n **must be** in the parameter space,
- the loop is turned upside down (from n to 0).
- It also works for nested loops.

Are there any more complicated patterns ?

Who's in the parameter space?

The minimal parameter space

- Some variables **must** be in the parameter space.
- How do we find them ?

The maximal parameter space

- Any variable **can** be in the parameter space.
- Any decision problem can expressed in this category:

$$\begin{tikzcd} Y \arrow[loop left, "\mathrm{id}_Y"] \end{tikzcd} \qquad \begin{tikzcd} N \arrow[loop right, "\mathrm{id}_N"] \end{tikzcd}$$

- Even when we fix $\mathbf{FMod}_{\mathbb{Z}}$, there is a similar construction.
- All the variables are in the parameter space.

Who's in the parameter space?

The case of multiplication

```
Mult(x, y) {
  return x * y;
}
```

- Multiplication is not linear (it is bilinear),
- But *Transposed multiplication* (aka *Middle product*) is a very important operation :
- fix `x`, then $\mathtt{Mult}_\mathtt{x}$ is linear.

The solution

Put `x` in the parameter space !

How do we automatically find it ?

Bibliography

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

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

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

F. Rouillier.
Solving zero-dimensional systems through the Rational Univariate Representation.
Appl. Alg. in Eng. Comm. Comput., 9(5):433–461, 1999.