

# 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, Lecerf, 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  $U_i$

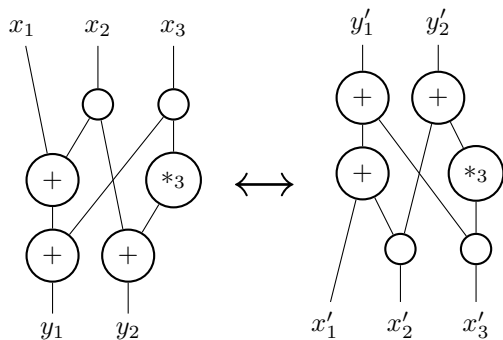
$$M(v) = U_1 \circ U_2 \circ \cdots \circ U_n(v) \quad \Leftrightarrow \quad M^T(v) = U_n^T \circ \cdots \circ U_2^T \circ U_1^T$$

## 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)



$$y_1 = x_1 + x_2 + x_3$$

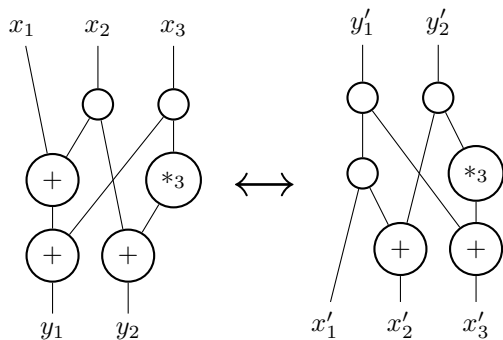
$$y_2 = x_2 + 3x_3$$

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



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

# Graph-theoretic approach (cont'd)



$$y_1 = x_1 + x_2 + x_3$$

$$y_2 = x_2 + 3x_3$$

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

# Category theory, a gentle introduction...

- Category  $\mathcal{C}$
- Objects  $\text{ob}(\mathcal{C})$

$A$

$B$

$C$

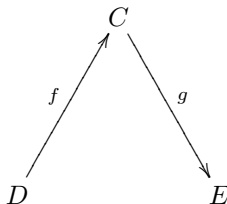
$D$

$E$

# Category theory, a gentle introduction...

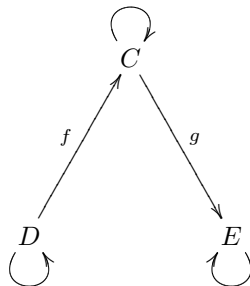
- Category  $\mathcal{C}$
- Objects  $\text{ob}(\mathcal{C})$
- Arrows  $\text{hom}(\mathcal{C})$ ,  
 $\text{Hom}(A, B)$

$$A \begin{array}{c} \xrightarrow{\quad} \\ \xleftarrow{\quad} \end{array} B$$



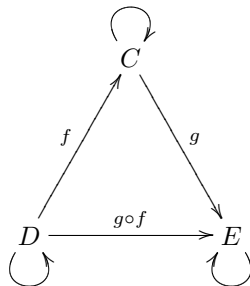
# Category theory, a gentle introduction...

- Category  $\mathcal{C}$
- Objects  $\text{ob}(\mathcal{C})$
- Arrows  $\text{hom}(\mathcal{C})$ ,  
 $\text{Hom}(A, B)$
- Identities  $\text{id}_A$



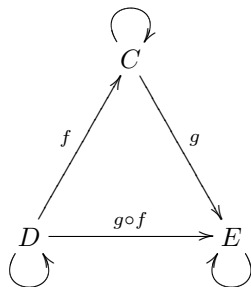
# Category theory, a gentle introduction...

- Category  $\mathcal{C}$
- Objects  $\text{ob}(\mathcal{C})$
- Arrows  $\text{hom}(\mathcal{C})$ ,  $\text{Hom}(A, B)$
- Identities  $\text{id}_A$
- Composition  $g \circ f$



# Category theory, a gentle introduction...

- Category  $\mathcal{C}$
- Objects  $\text{ob}(\mathcal{C})$
- Arrows  $\text{hom}(\mathcal{C})$ ,  $\text{Hom}(A, B)$
- Identities  $\text{id}_A$
- Composition  $g \circ f$



## Example : $\mathbf{FMod}_R$

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

# Category theory, Functors

Covariant functor

$$F : \mathcal{C} \rightarrow \mathcal{D}$$

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow h & \downarrow g \\ & & C \end{array} \quad \mapsto \quad \begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ & \searrow F(h) & \downarrow F(g) \\ & & F(C) \end{array}$$

Contravariant functor

$$F : \mathcal{C} \rightarrow \mathcal{D}$$

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow h & \downarrow g \\ & & C \end{array} \quad \mapsto \quad \begin{array}{ccc} F(A) & \xleftarrow{F(f)} & F(B) \\ & \swarrow F(h) & \uparrow F(g) \\ & & F(C) \end{array}$$

## Equivalence, duality

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

and  $F \circ G \simeq \text{Id}_{\mathcal{D}}$  and  $G \circ F \simeq \text{Id}_{\mathcal{C}}$ .

“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
```

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

# Computations

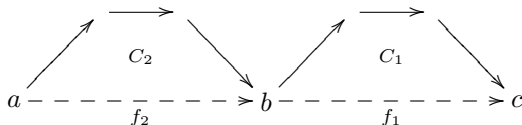
## Language, size

- Set of *instructions*
- Size function

$$\mathcal{L} \subset \text{hom}(\mathcal{C}),$$
$$\|\cdot\| : \text{ob}(\mathcal{C}) \rightarrow \mathbb{N}.$$

## Computation

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



## Time and space cost

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

# The case $\mathbf{FMod}_R$

[Bostan, Lercerf, Schost '03] :

$$+_1 : R^2 \rightarrow R^2$$

$$(p, q) \mapsto (p + q, q)$$

$$+_2 : R^2 \rightarrow R^2$$

$$(p, q) \mapsto (p, p + q)$$

$$*_a : R \rightarrow R$$

$$p \mapsto ap$$

$$\pi : R \rightarrow 0$$

$$p \mapsto 0$$

$$\iota : 0 \rightarrow R$$

$$0 \mapsto 0$$

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

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

# Our example

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

$$\begin{aligned} & \text{Id}_i \times +_2 \times \text{Id}_{n-2-i} \\ & \text{Id}_i \times *_0 \times \text{Id}_{n-1-i} \end{aligned}$$

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

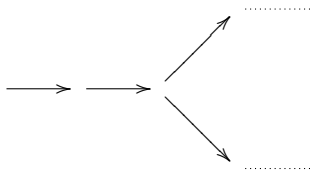
# Our example

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

$$\begin{aligned} & \text{Id}_i \times +_2 \times \text{Id}_{n-2-i} \\ & \text{Id}_i \times *_0 \times \text{Id}_{n-1-i} \end{aligned}$$

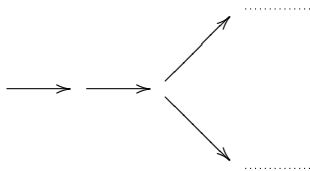
$$R^n \xrightarrow{+_2 \times \text{Id}_{n-2}} R^n \xrightarrow{*_0 \times \text{Id}_{n-1}} R^n \dots R^n$$

# 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,  $\text{Par} = \mathbb{N}$

## Algorithm

A function  $A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  ( $\mathcal{C}_{\rightarrow} =$  the computations)

# Algorithm

## Parameter space

Par a recursively enumerable set

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

## Algorithm

A function  $A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  ( $\mathcal{C}_{\rightarrow} =$  the computations)

1



# Algorithm

## Parameter space

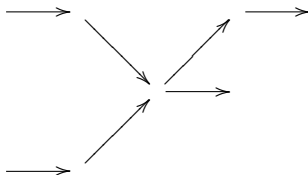
Par a recursively enumerable set

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

## Algorithm

A function  $A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  ( $\mathcal{C}_{\rightarrow}$  = the computations)

1  
2



# Algorithm

## Parameter space

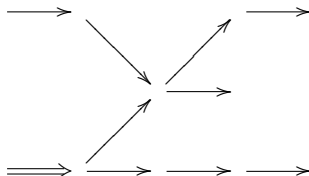
Par a recursively enumerable set

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

## Algorithm

A function  $A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  ( $\mathcal{C}_{\rightarrow}$  = the computations)

1  
2  
3  
⋮



# Complexity

## Time complexity

$A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  induces a function  $t_A : \text{Par} \rightarrow \mathbb{N}$  given by

$$t_A(x) = t(A(x))$$

## Space complexity

$A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  induces a function  $s_A : \text{Par} \rightarrow \mathbb{N}$  given by

$$s_A(x) = s(A(x))$$

# Our example

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

$$R^n \xrightarrow{+2 \times \text{Id}_{n-2}} R^n \xrightarrow{*0 \times \text{Id}_{n-1}} R^n \dots R^n$$

# Our example

$$a[1] = a[0] + a[1]$$

$$a[0] = 0$$

$$a[2] = a[1] + a[2]$$

$$a[1] = 0$$

...

$$a[n-1] = a[n-2] + a[n-1]$$

$$a[n-2] = 0$$

$$n \mapsto R^n \xrightarrow{+2 \times \text{Id}_{n-2}} R^n \xrightarrow{*0 \times \text{Id}_{n-1}} R^n \dots R^n$$

# Tellegen's theorem

## T-functor

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

## Tellegen's theorem

- $F : \mathcal{C} \rightarrow \mathcal{D}$  a T-functor
- $\text{Par}$  a parameter space
- $A : \text{Par} \rightarrow \mathcal{C}_{\rightarrow}$  an algorithm

$F \circ A$ , noted  $F(A)$  is an algorithm  $\text{Par} \rightarrow \mathcal{D}_{\rightarrow}$  such that

- $t_{F(A)} = t_A$ ,
- $s_{F(A)} \leq B(s_A)$  if  $B : \mathbb{N} \rightarrow \mathbb{N}$  is an upper bound for  $F$ .

# The case $\mathbf{FMod}_R$

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

## Tellegen's theorem for linear algebra

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

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

# 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
```

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

# Other examples: Quantum Computing

## The QC category $\mathcal{Q}$

$$\text{ob}(\mathcal{Q}) = \{(\mathbb{C}^2)^{\otimes n} \mid n \in \mathbb{N}\}$$

$$\text{hom}(\mathcal{Q}) = U \text{ unitaries } (U^* = U^{-1})$$

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

## Language, size

$$\mathcal{L} = \left\{ \text{Id}_2^{\otimes n} \otimes_{\text{op}} \text{Id}_2^{\otimes m} \mid n, m \in \mathbb{N}, \text{op} \in \{H, R, R^*, CNOT\} \right\} \quad \|(\mathbb{C}^2)^{\otimes n}\| = n$$

## The functor

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

# Other examples: Extension and restriction of scalars

$A, B$  two rings,  $f : A \rightarrow B$  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 lose information).

# Application: transposed multiplication

## Dual space

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

$V$  basis of  $\mathbb{L} \rightarrow V^*$  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  $M_x$ :

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

## Transposed multiplication

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

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

- Hence  $M_x^T$  is the linear application computing  $x \cdot \ell$  from  $\ell$ .
- We can transform an algorithm for multiplication into one for middle product.
- Used by [Bostan, Lecerf, 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  $\text{Tr}(c) = \text{Tr}(M_c)$ ,
- let  $C \in A[X]$  s.t  $c = C(b)$ , then by linear algebra

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

## 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  $U \rightarrow V$  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  $\ell_c = c \cdot \text{Tr}$ , then

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

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

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

## The algorithm

- Computing  $\text{Tr}$  in the basis  $V^*$ .  $O(M_V)$
- Knowing  $c$  in the basis  $V$ , computing  $\ell_c = c \cdot \text{Tr}$  is middle product.  $M_V$
- computing the first  $n + 1$  coefficients of  $\sum \ell_c(b^i) T^i$  is the same as move  $\ell_c$  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**.

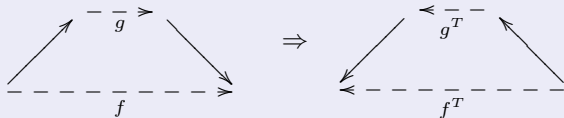
## Straight line programs

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

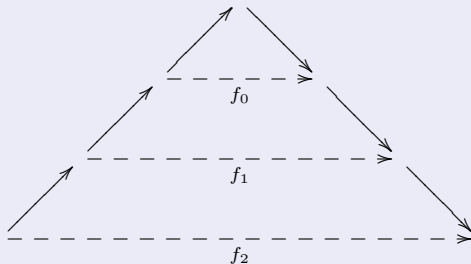
# Subroutines

## 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 be expressed in this category:

$$\text{id}_Y \circlearrowleft Y \quad N \circlearrowright \text{id}_N$$

- 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  $\text{Mult}_x$  **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.