On the paradigm of functional programming: Functionals as hardware

Stanislaw Ambroszkiewicz

LambdaDays, February 9-10, 2017
Kraków

What is functional?

- Is it a term? Or does the term denote a functional?
- **The current paradigm in IT:** only symbolic computation (term rewriting) is possible for higher order objects
- Hardware technology is very close to break the paradigm. Functionals may be envisioned as generic mechanisms for the management of dynamic connections in reconfigurable huge arrays of functional units (elementary first order functions)

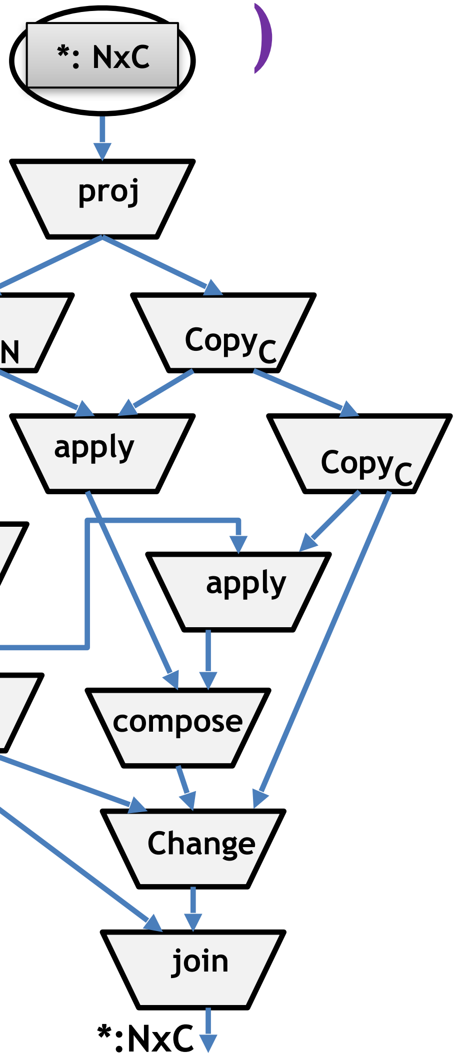
What is functional?

- Conceptually, functional is a **fully pipelined data-flow (directed acyclic graph)** with nodes corresponding to primitive functionals, and edges corresponding to flow of data from output of one node to the input of the next node.
- For complex functionals (with recursion) some nodes are dynamically unfolded

data-flow acyclic graph F

input: $c = (c(1), c(2), \dots , c(n), \dots)$

(n,c)



C denotes $N \rightarrow (A \rightarrow A)$;

D denotes $N \times C$

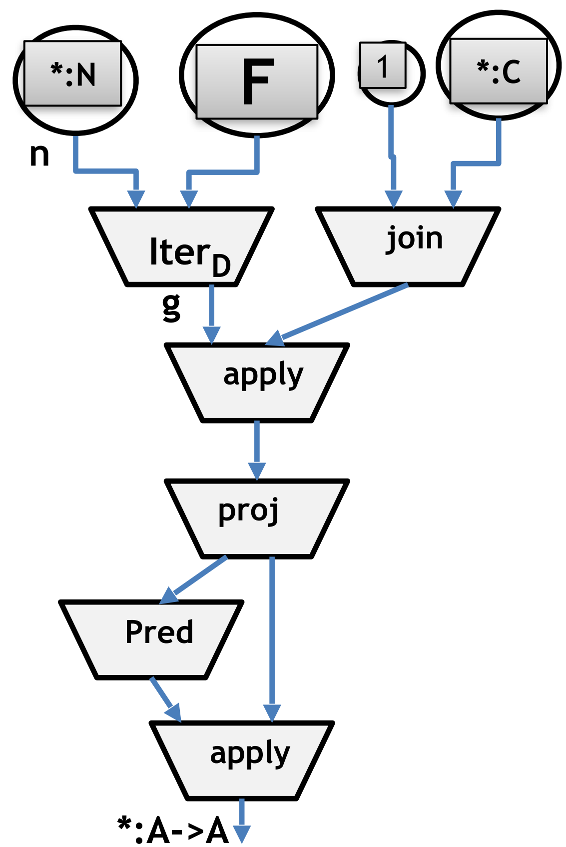
$F: D \rightarrow D$

$$c^o(n+1) = c(n) \circ c(n+1)$$

output: $(n+1, \ (c(1), c(2), \dots , c(n), c^o(n+1),$

Higher order primitive recursion schema

C denotes N → (A → A);   D denotes N×C

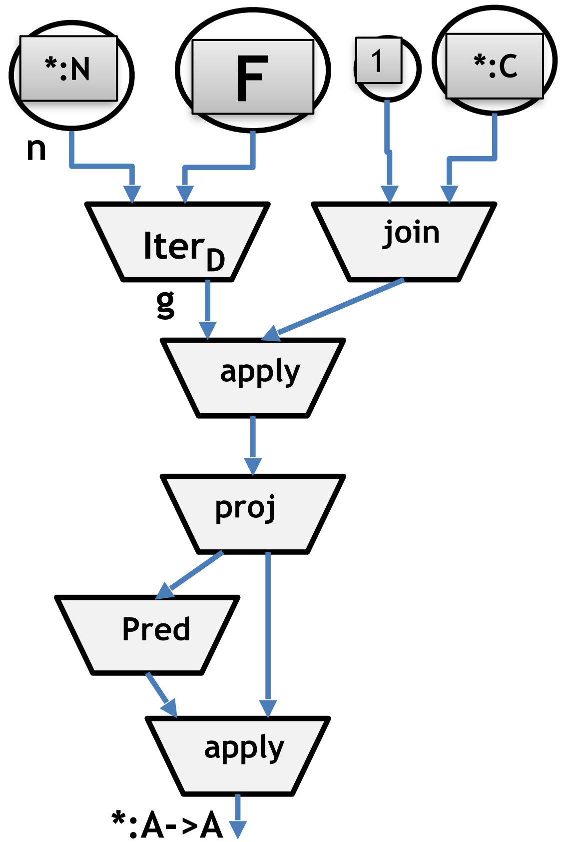


R^A: (N;C) → (A → A)

C denotes N → (A → A); D denotes N x C; op: D → D

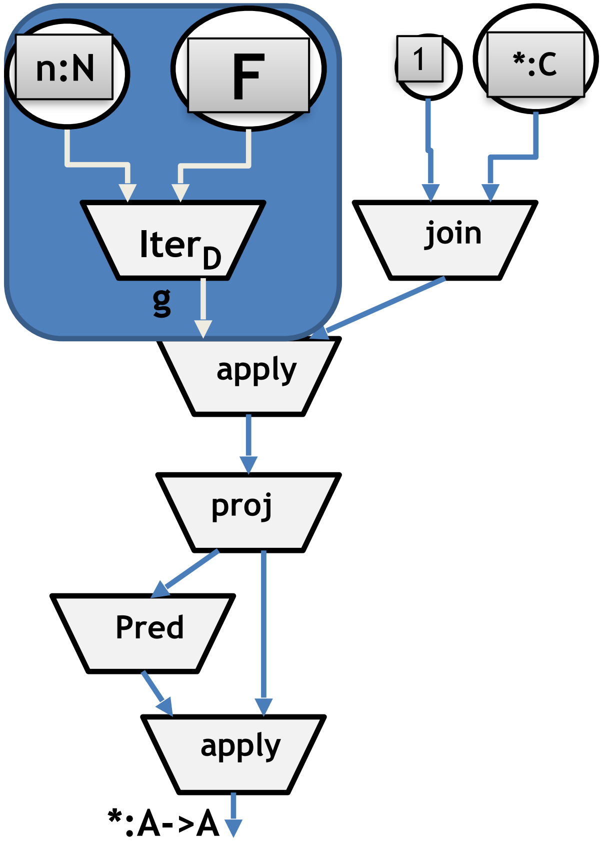
input: n ; c

c = (c(1), c(2), ... , c(n), ...)

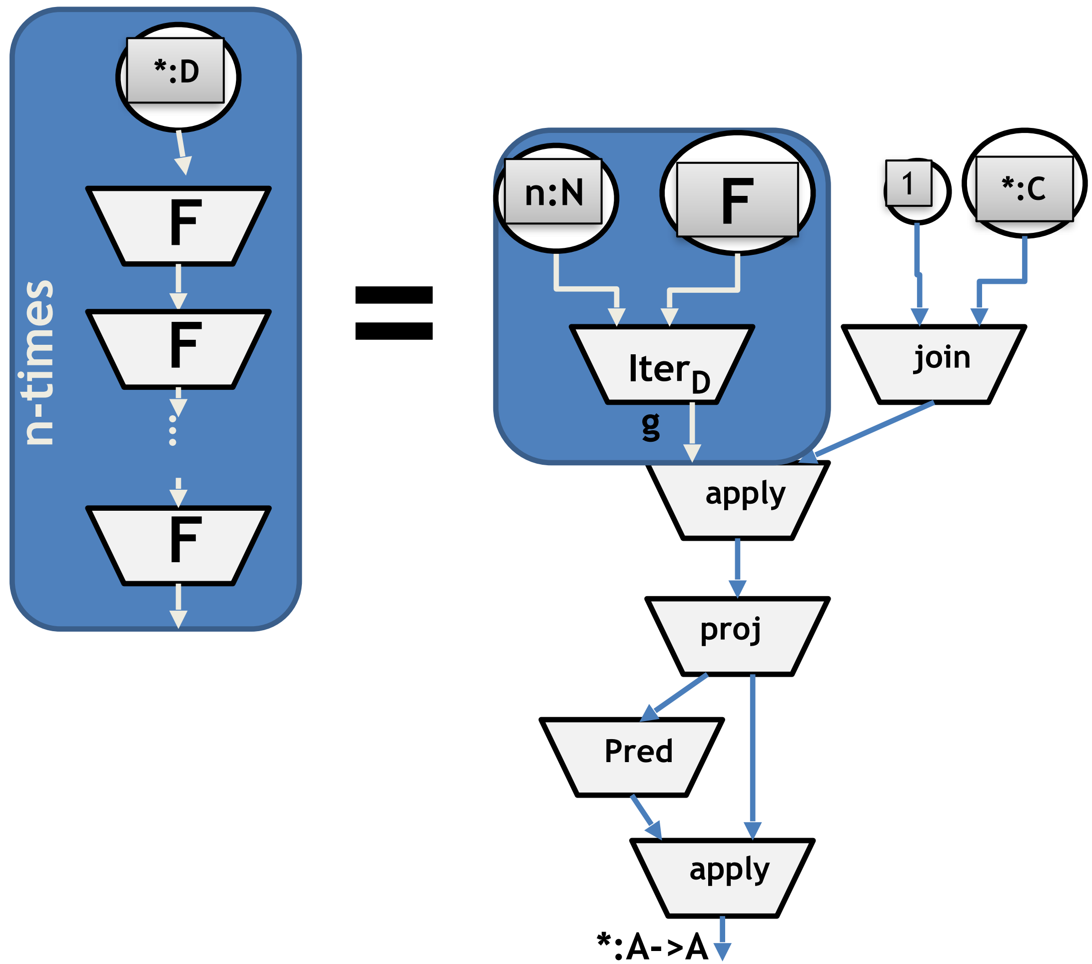


$c^o(n) = c(1) \circ c(2) \circ \ldots \circ c(n)$

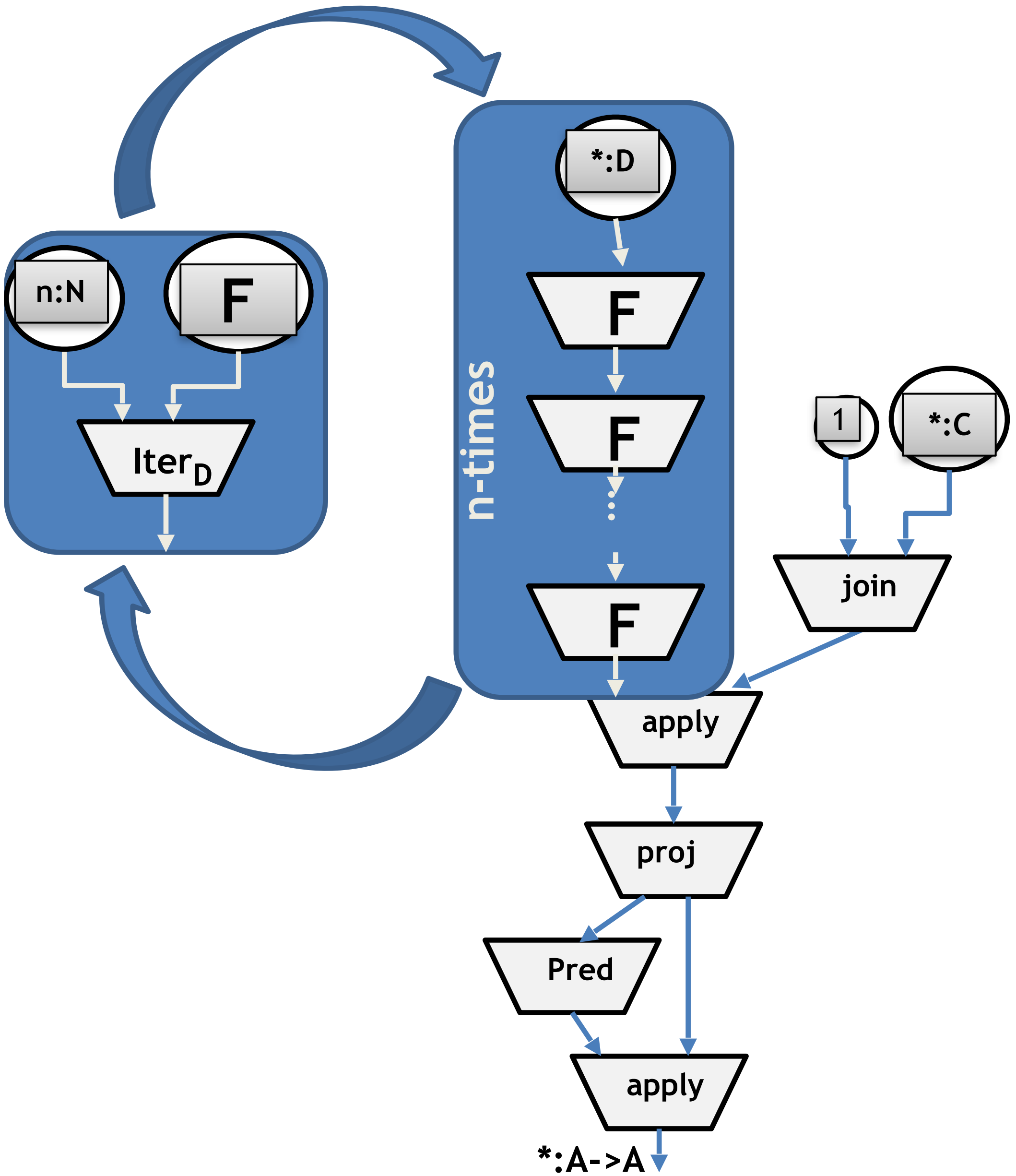
output: $c^o(n)$

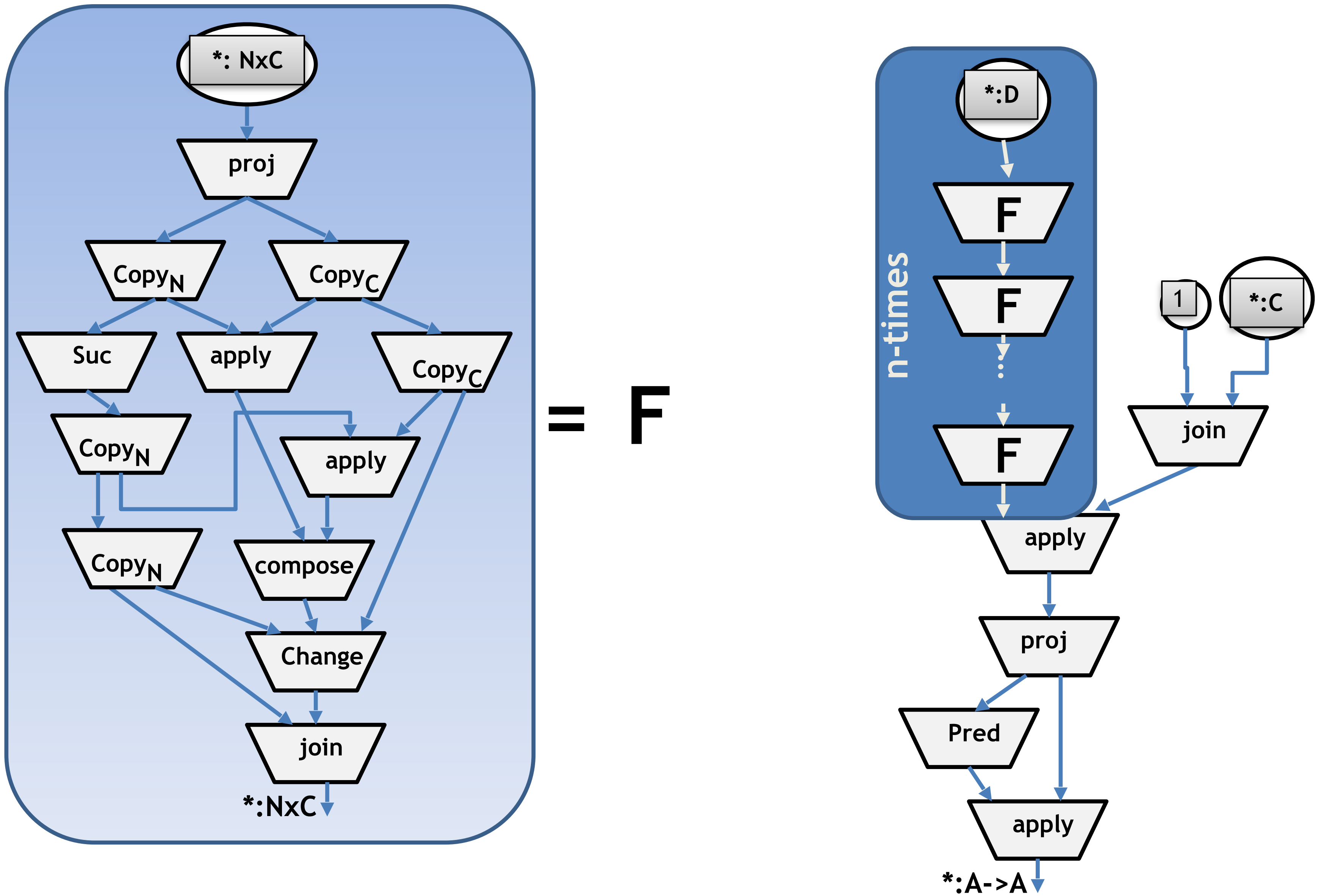


n:N
F
1
*:C
Iter$_D$
join
g
apply
proj
Pred
apply
*:A->A



*:D
F
F
F
n-times
=
n:N
F
Iter_D
g
1
*:C
join
apply
proj
Pred
apply
*:A->A

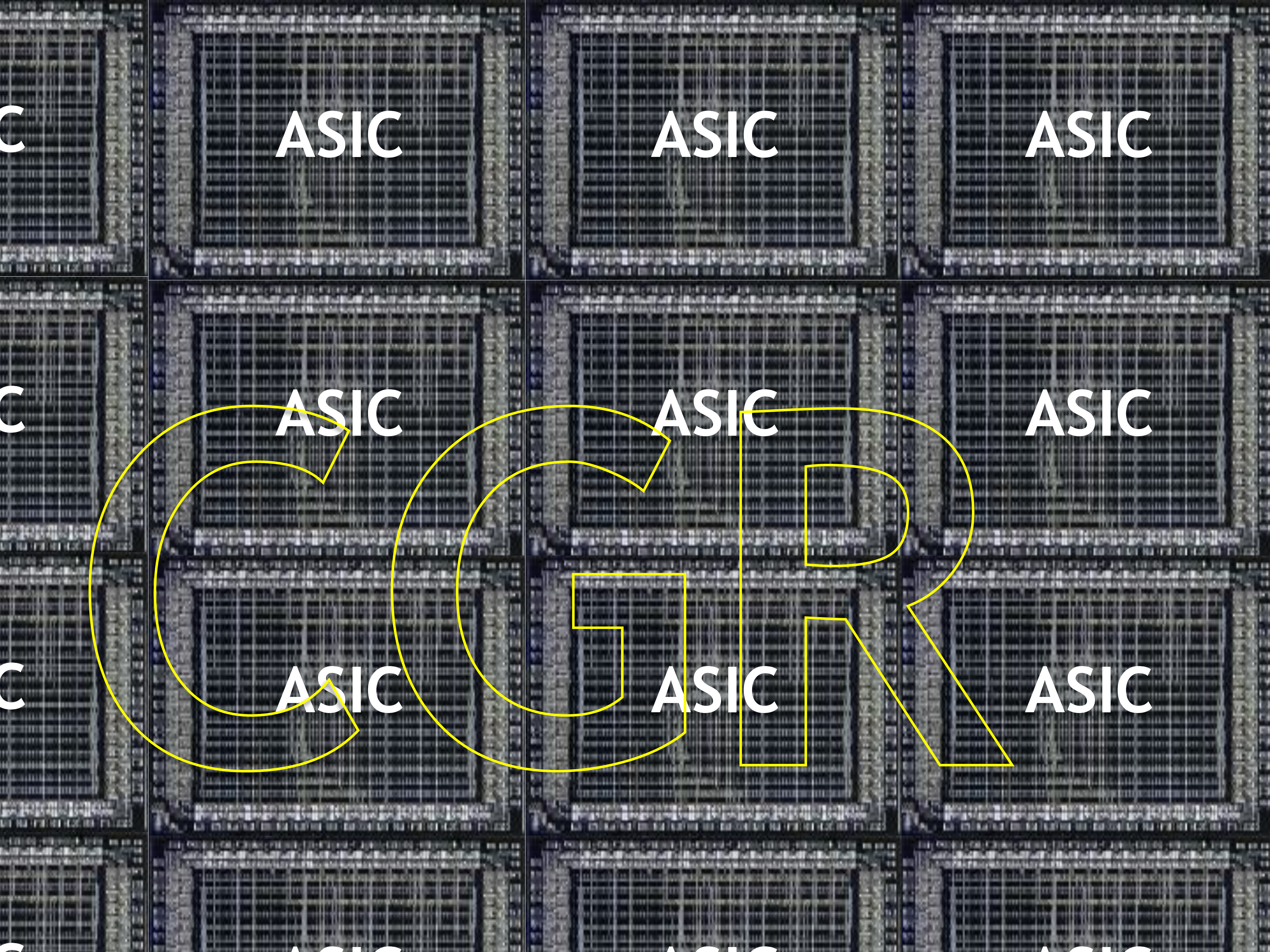

n:N
F
Iter$_D$
*:D
F
F
F
n-times
1
*:C
join
apply
proj
Pred
apply
*:A->A

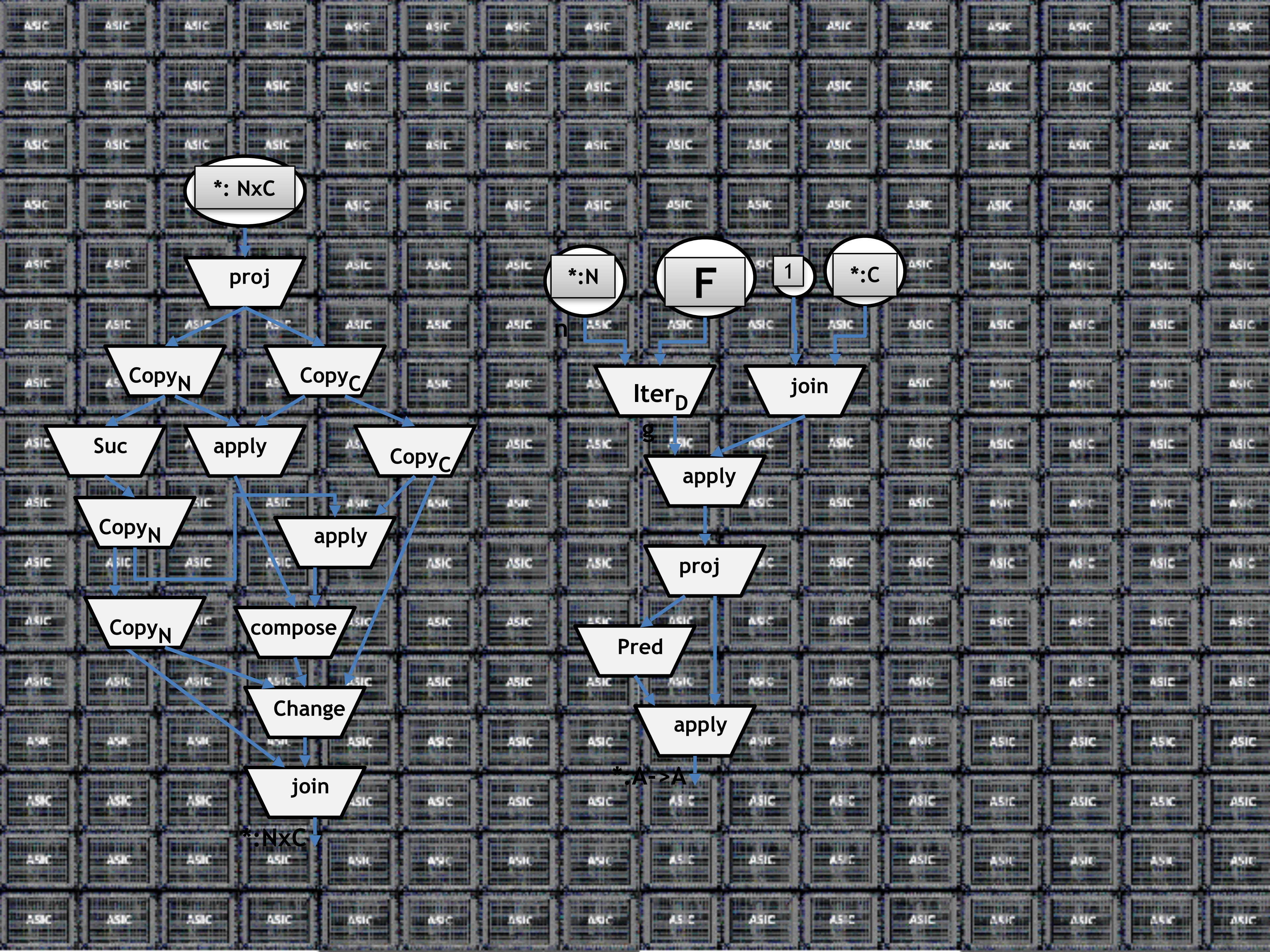


*: NxC
proj
CopyN
CopyC
Suc
apply
CopyC
CopyN
apply
CopyN
compose
Change
join
*:NxC
= F
*:D
F
F
F
n-times
1
*:C
join
apply
proj
Pred
apply
*:A->A

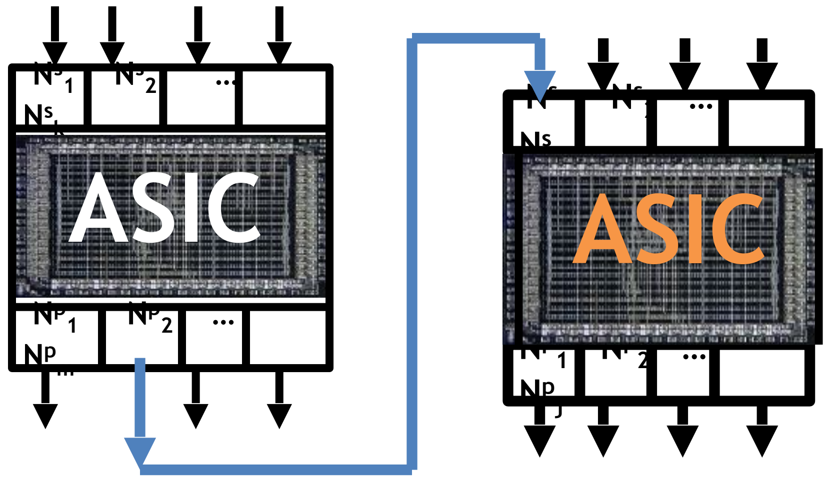
Functionals as transformations of acyclic directed graphs

mapping to CGRA (coarse-grained reconfigurable array) if all nodes of th graph are first order functions


CGR
ASIC
ASIC
ASIC
ASIC
ASIC
ASIC
ASIC
ASIC
ASIC

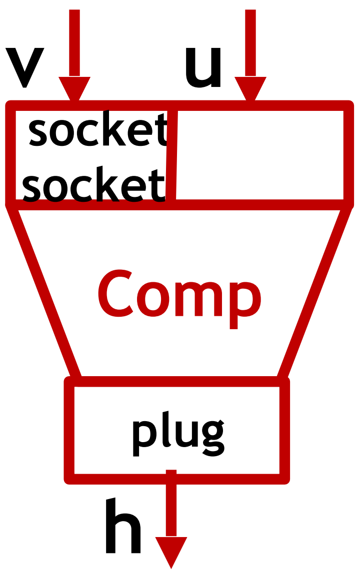

*: NxC
proj
CopyN
CopyC
Suc
apply
CopyC
CopyN
apply
CopyN
compose
Change
join
*:NxC
*:N
F
1
*:C
n
IterD
join
g
apply
proj
Pred
apply
*:A->A

Abstract view

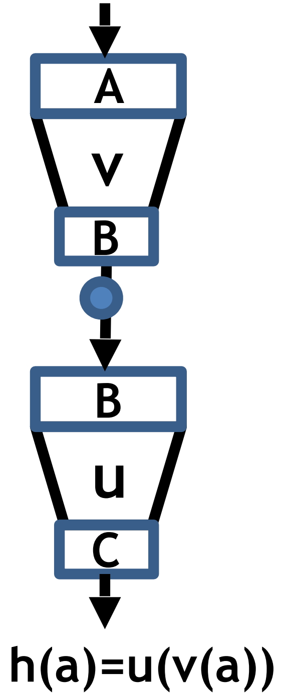


Connection between two chips is composition of two first order functions

composition as a functional

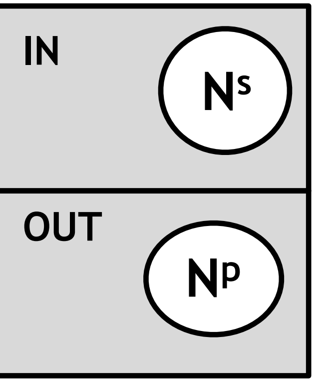


- Abstraction:
  Comp(u;v) = h
- two functions are arguments; value is a function
- functions **u** and **v** are put into the sockets; the result is at the plug as **h**



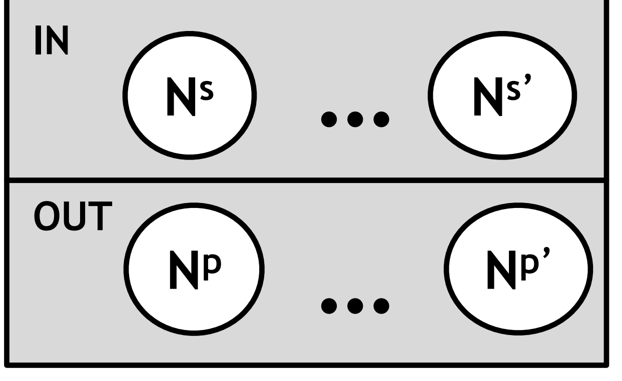
Sockets, plugs and types

- New notions:
  - Input of a function as a socket
  - Output of a function as a plug
- natural numbers N as primitive data type
- Function type $f{:}N^s \rightarrow N^p$
  - N^s is a socket of type N, and N^p is a plug of type N
  - type $N^s \rightarrow N^p$ as a concrete object, i.e. the following board



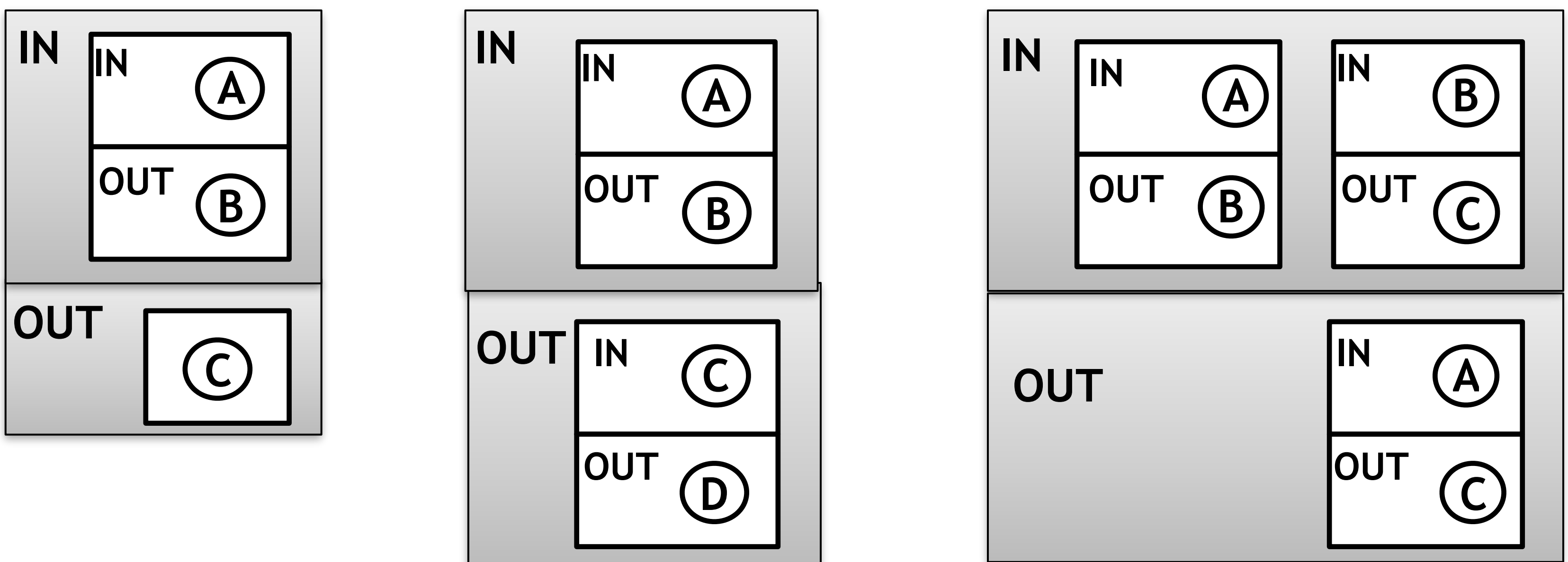
- Type of a function as the board

  sockets → plugs



Sockets, plugs and higher order types

- $(A \rightarrow B) \rightarrow C$
- $(A \rightarrow B) \rightarrow (C \rightarrow D)$
- $((A \rightarrow B); (B \rightarrow C)) \rightarrow (A \rightarrow C)$

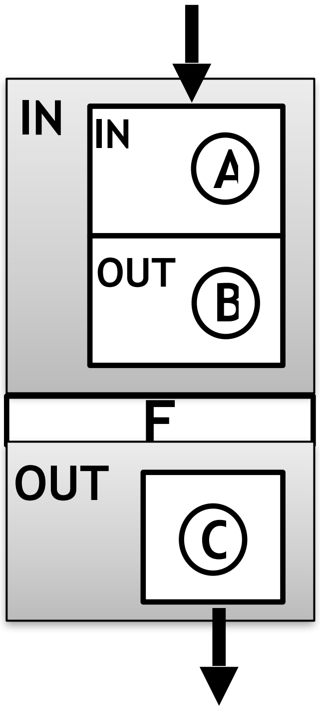


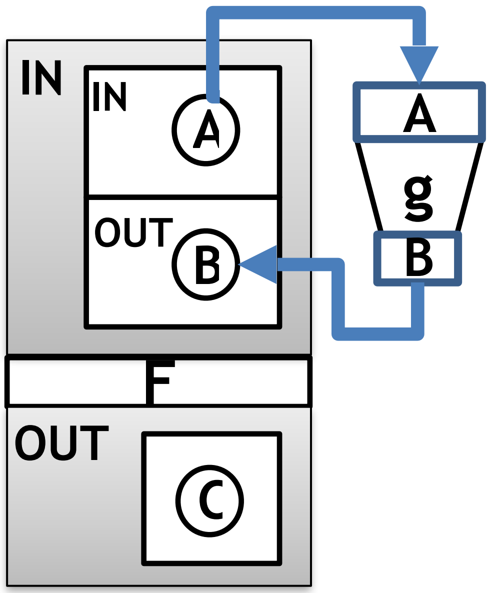
Links (connections)

- between plugs and sockets of <u>the same type</u>.
- A link is always directed, i.e. it determines the direction of data flow
- Generic mechanisms for dynamic creation and reconfiguration of links between plugs and sockets as higher order computations
- **Actually, the generic mechanisms are the functionals**

Computable functionals 1

- Application of a functional **F** of type **(A → B) → C** to a function **g: A → B**
- Note that **A→B** is the socket of the functional **F**. The application is done (see on the right) by establishing appropriate directed connections (links).
- The link between the socket **A** of the socket of **F** and the socket **A** of **g**, and the link between the plug **B** of **g** and the plug of the socket of **F**.
- The result , i.e. **F(g)** is of type **C**.

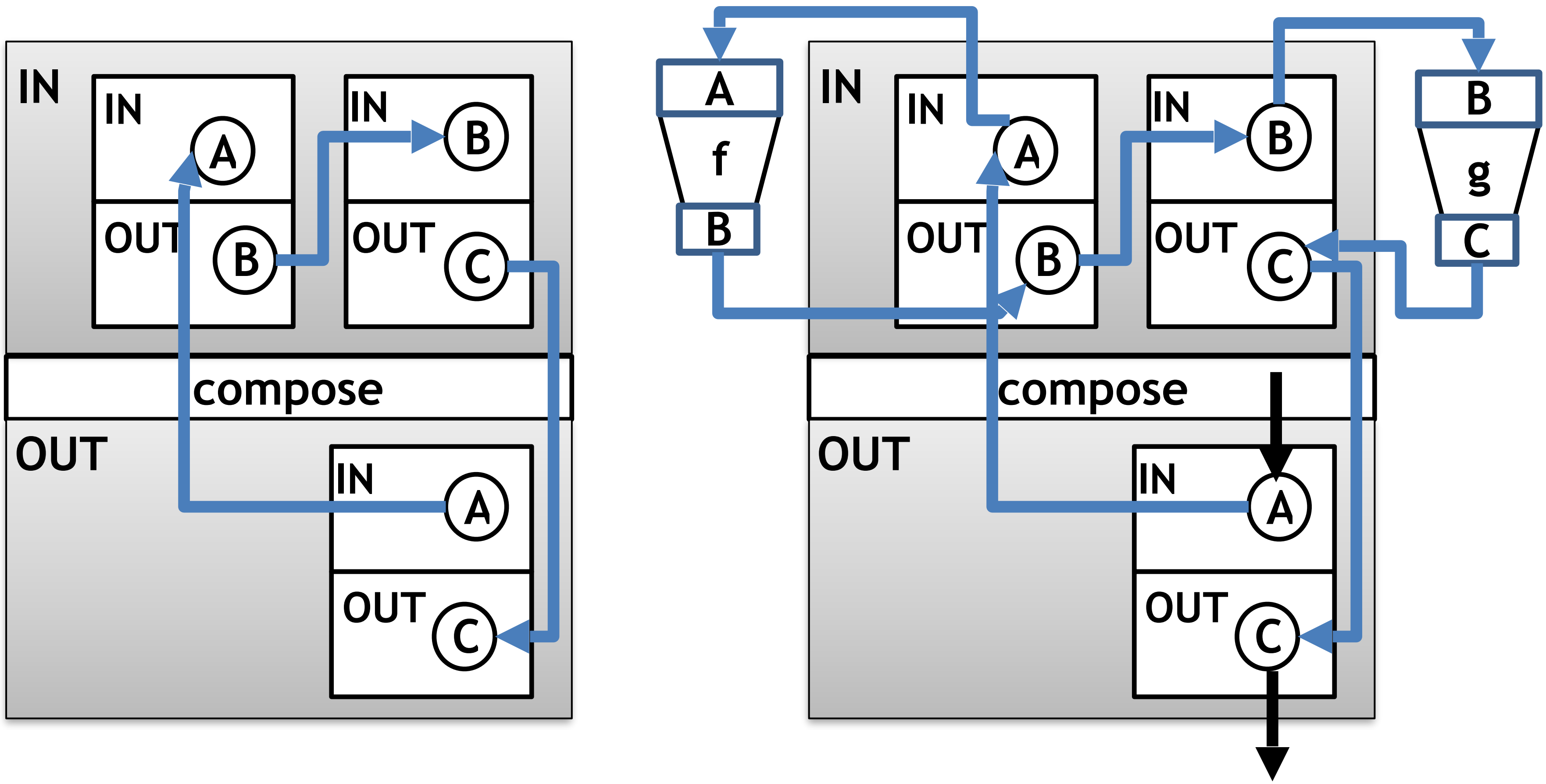




Computable functionals 2a

- The functional **compose**$_{ABC}$ **: ((A → B); (B→C)) → (A → C)**
- for composition of two functions (**f** of type **A→B**, and **g** of type **B→C**)
- realized as two boards with appropriate links (see the next slide).
- To check that **compose**$_{ABC}$ **(f;g),** i.e. application of **compose**$_{ABC}$ to **f** and **g**, is the composition, just follow the links. The result is of type **A→C**

Computable functionals 2b



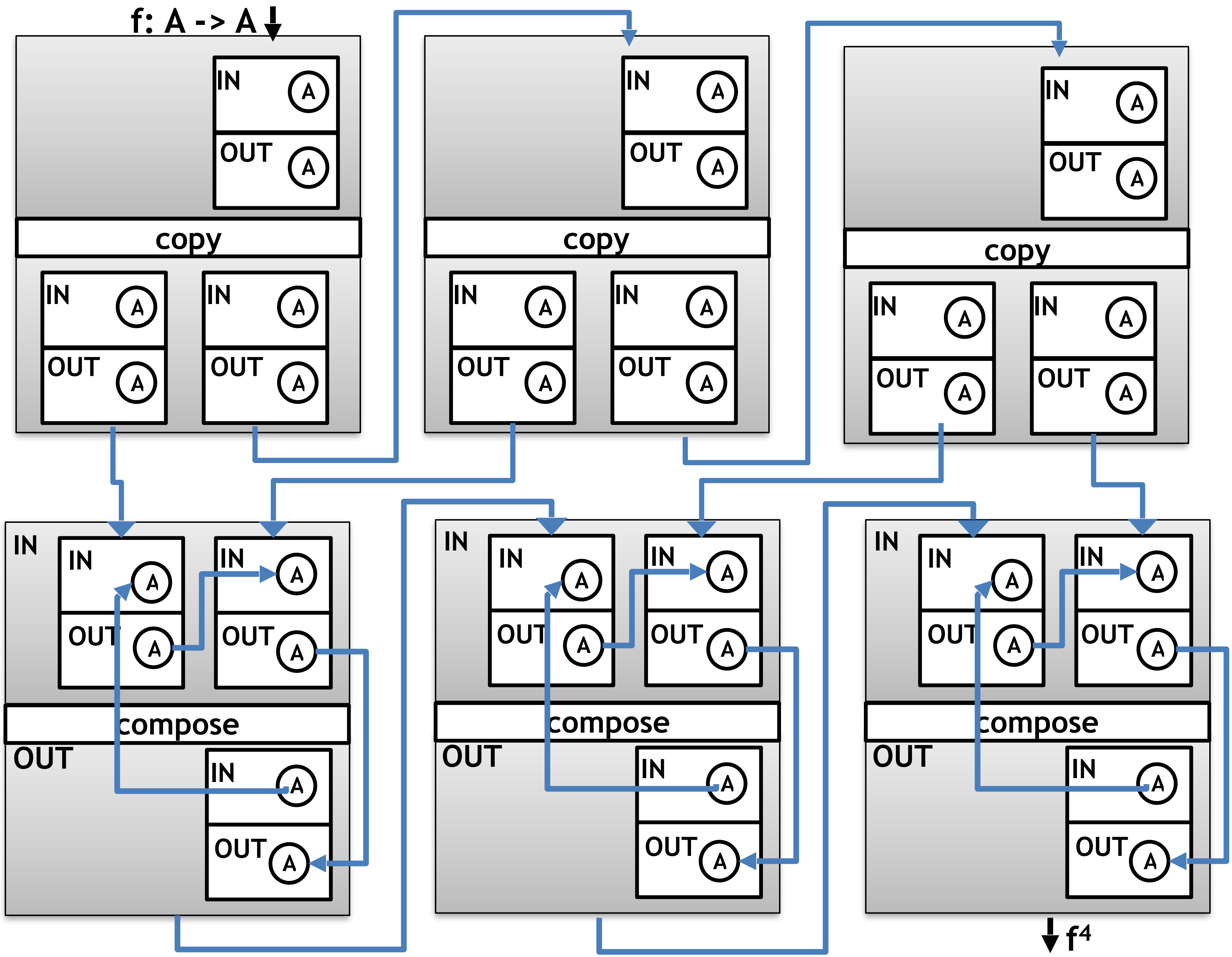
Computable functionals 3

- higher order application and composition are constructed just by providing some links between sockets and plugs.
- Functionals are constructed by dynamic creation and reconfiguration of links between sockets and plugs
- **Primitive type:** **natural numbers N**
- **Primitive type constructions:**
  - **product, disjoin union, arrow (for function types), dependent types**
- **Primitive operations**
  - **-apply, compose, Copy, Iter, Change, Successor, Predecessor, … primitive relations**

- **Computational power of the functionals: second order intuitionistic Arithmetic**

Operation $Iter(4;*)$ applied to f

$Iter_A : (N;(A \to A)) \to (A \to A)$

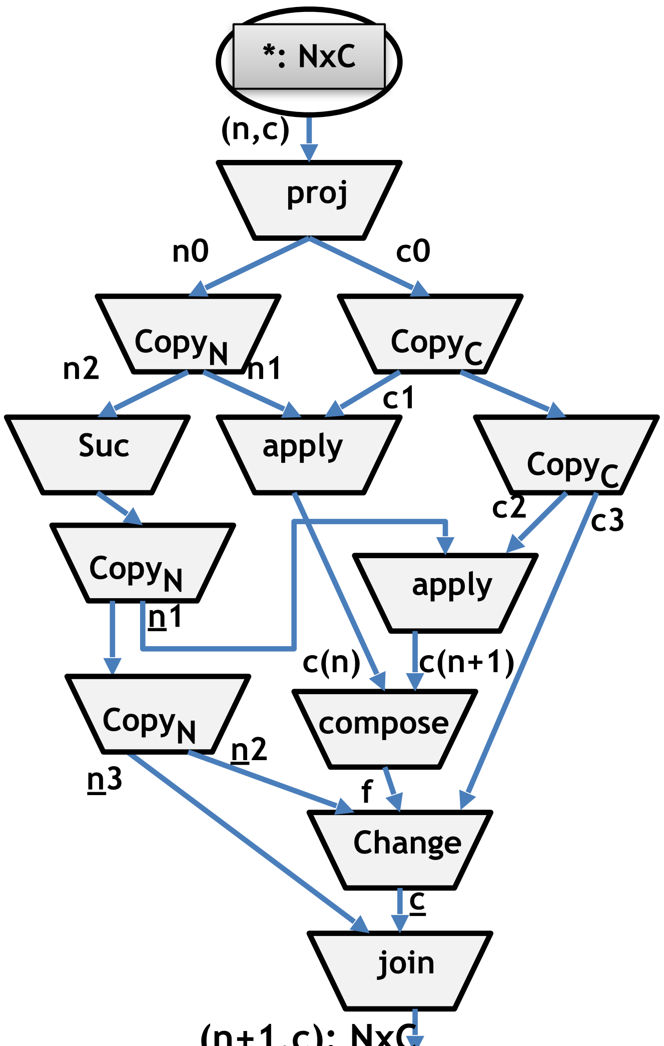


The end

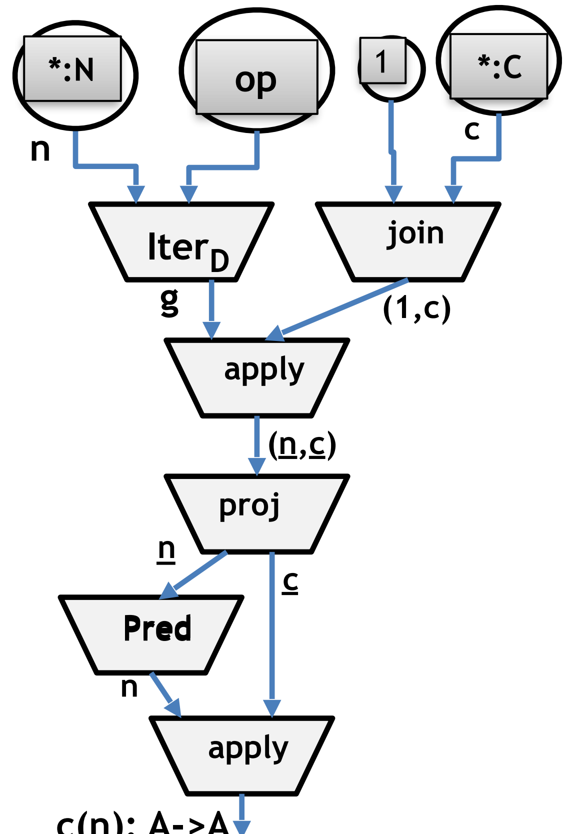
- more in „Types and operation v4” available at google arXiv Ambroszkiewicz

Higher order primitive recursion schema

C denotes N → (A → A);   D denotes NxC



op: D →



R^A: (N;C) → (A → A)