Compiling conventional languages to unconventional architectures

Dan R. Ghica

joint work with Alex Smith and Olle Fredriksson

OASIS 10th
24 November 2014

with or without game semantics

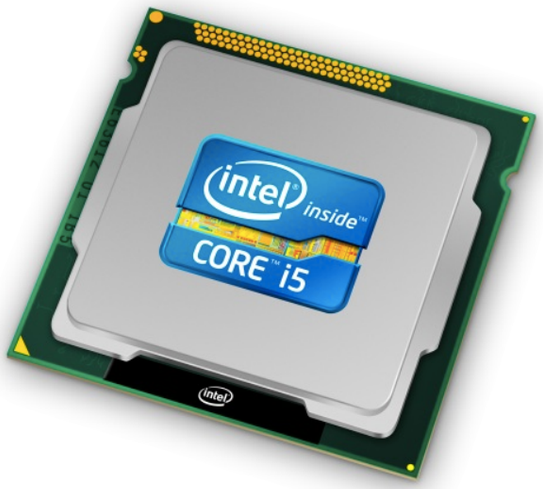
Compiling conventional languages to unconventional architectures

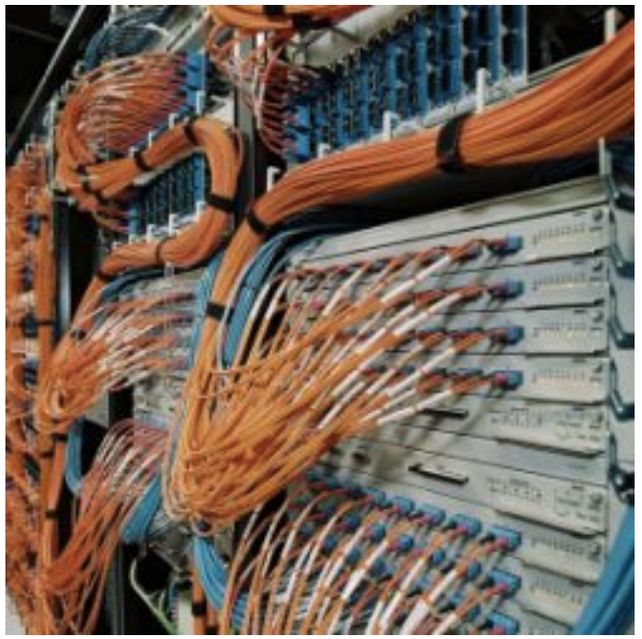
Dan R. Ghica

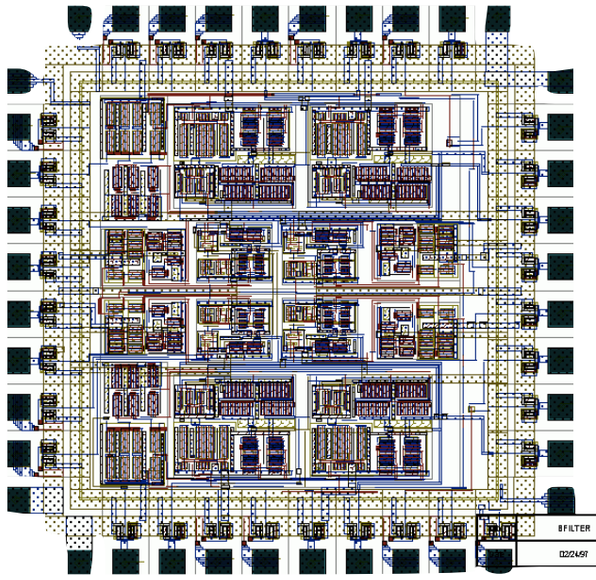
joint work with Alex Smith and Olle Fredriksson

OASIS 10th
24 November 2014

Unconventional architectures







Computation

Communication

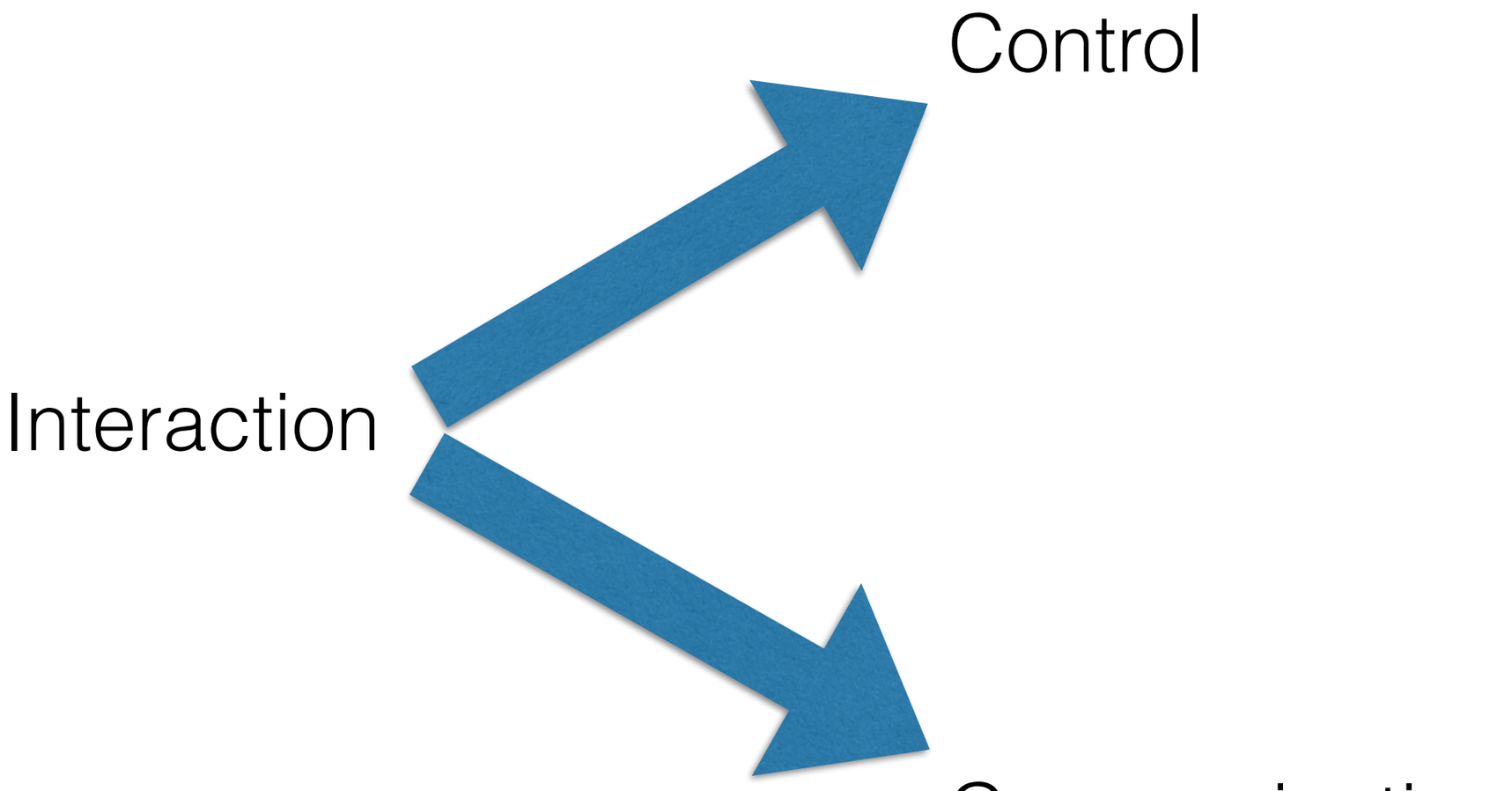
Control vs. communication

CPS-calculus	Appel's datatype `cexp`
$x\langle y_1 \ldots y_j \rangle$	`APP(VAR` x`,[VAR` y_1`,...,VAR` y_j`])`
$M\{n\langle x_1 \ldots x_j \rangle = N\}$	`FIX([(`n`,[`x_1`,...,`x_j`],`N`)],`M`)`

The binding of continuations in CPS can be implemented not only by "passing" (using the λ-calculus), but equally by "sending" (π-calculus) or "grabbing" (using `callcc`)

CPS and self-adjointness
H. Thielecke, 1996

Interaction semantics



Interaction semantics

Logic Colloquium '88
Ferro, Bonotto, Valentini and Zanardo (Editors)

GEOMETRY OF INTERACTION 1 : INTERPRETATION OF SYSTEM F

jean–yves girard
équipe de logique, UA 753 du CNRS
mathématiques, Université Paris VII

conception of execution, where actions at distance (*global time*) are forbidden, but what is found is always consistent with syntax, if not always the same. This difference with syntax must be seen as the disparition of any kind of central unit in a network of parallel computers trying to compute a sequential algorithm. Coming back to Λ^*, it turns out that this algebra manipulates very simple finite electronic circuits, and it is therefore likely that this modelisation could work on a concrete *ad hoc* machine.

Functions as Processes

Les Fonctions vues comme des Processus

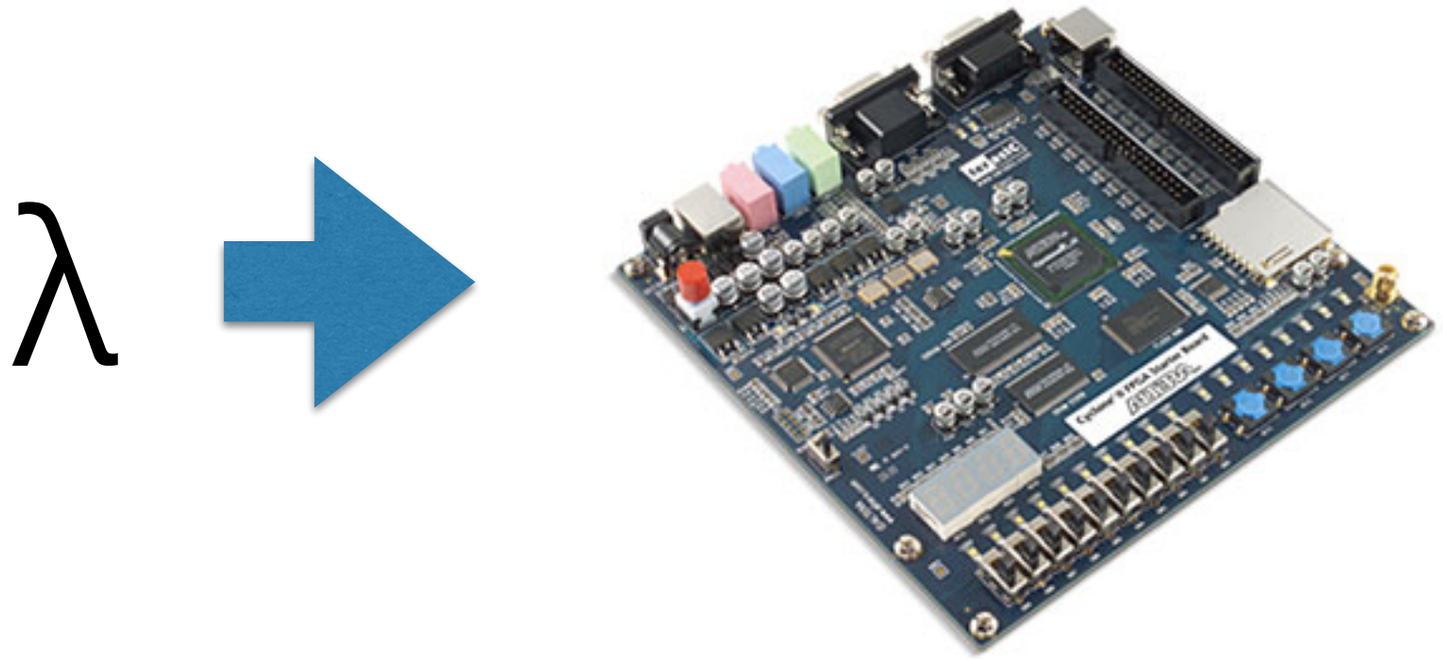
Robin Milner
University of Edinburgh
June 1989

Abstract This paper exhibits accurate encodings of the λ-calculus in the π-calculus. The former is canonical for calculation with functions, while the latter is a recent step [12] towards a canonical treatment of concurrent processes. With quite

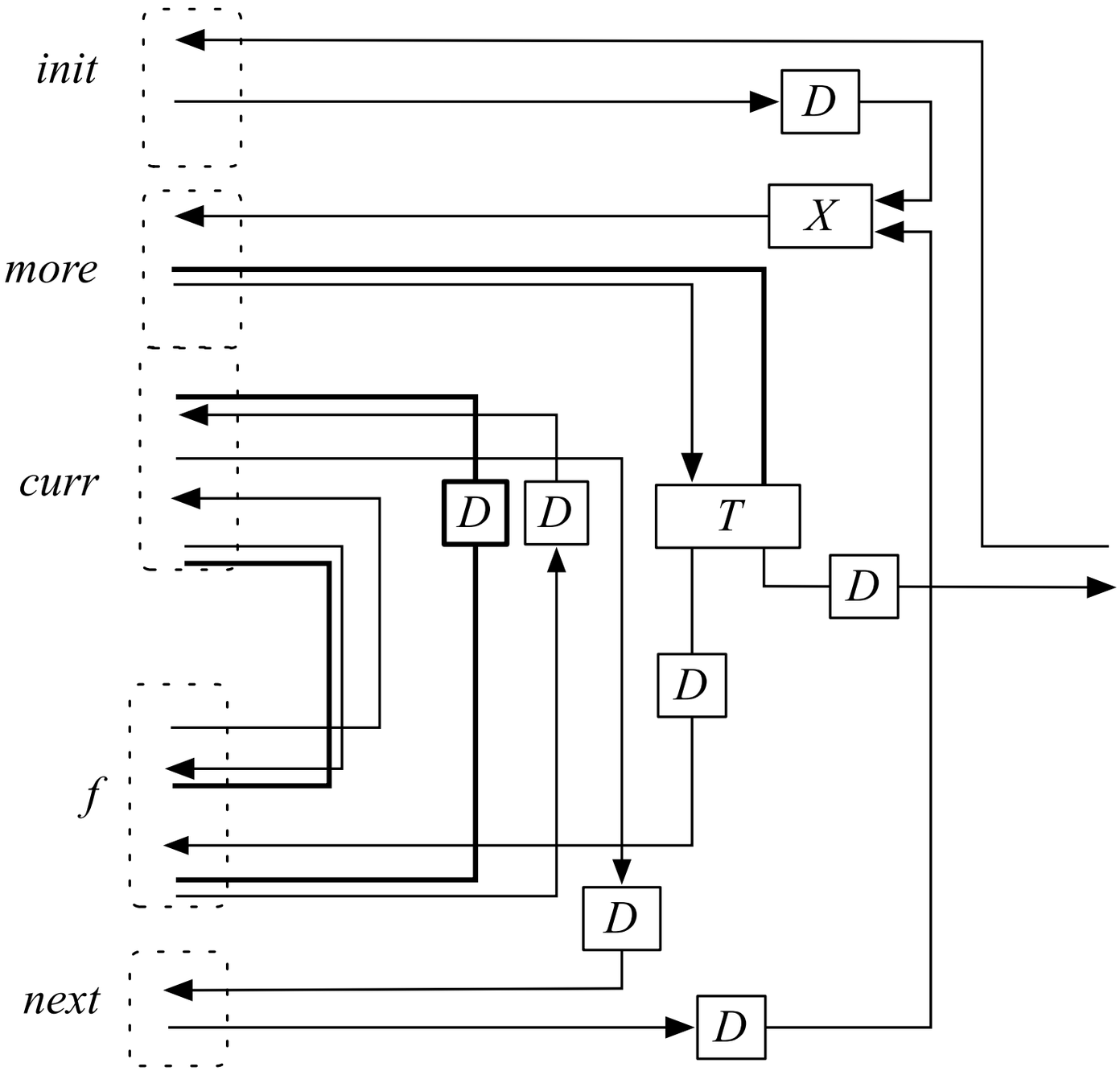
$$[\![x := M]\!] \stackrel{\text{def}}{=} !\,x(w).[\![M]\!]w$$

In passing, note particularly the replication. This is not needed if M will be called at most once; therefore the *linear* λ-calculus, in which each variable x must occur exactly once in its scope, may be encoded in the fragment of π-calculus without replication. The link with Girard's 'of course' connective '!' of linear logic [8] should be explored; his notation for it has been chosen here deliberately.

High-level synthesis with Game Semantics



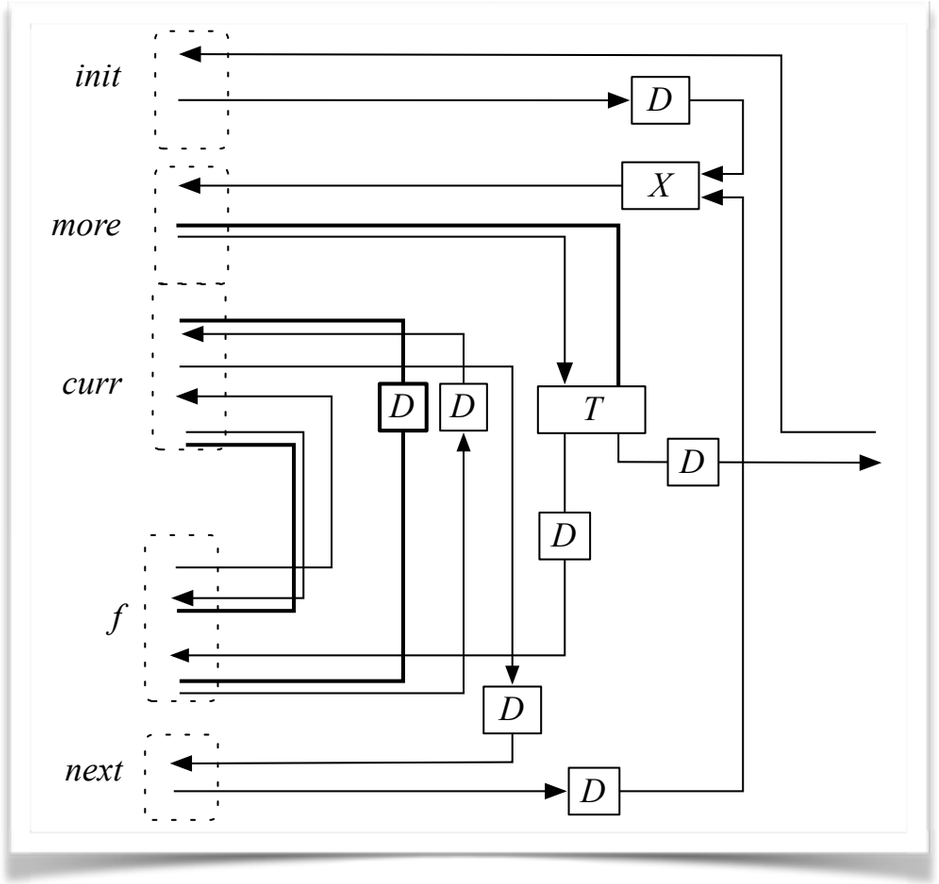
λf : exp $\rightarrow$ exp.init; while(more)(curr := f(!curr); next) : com,



HLS via GS

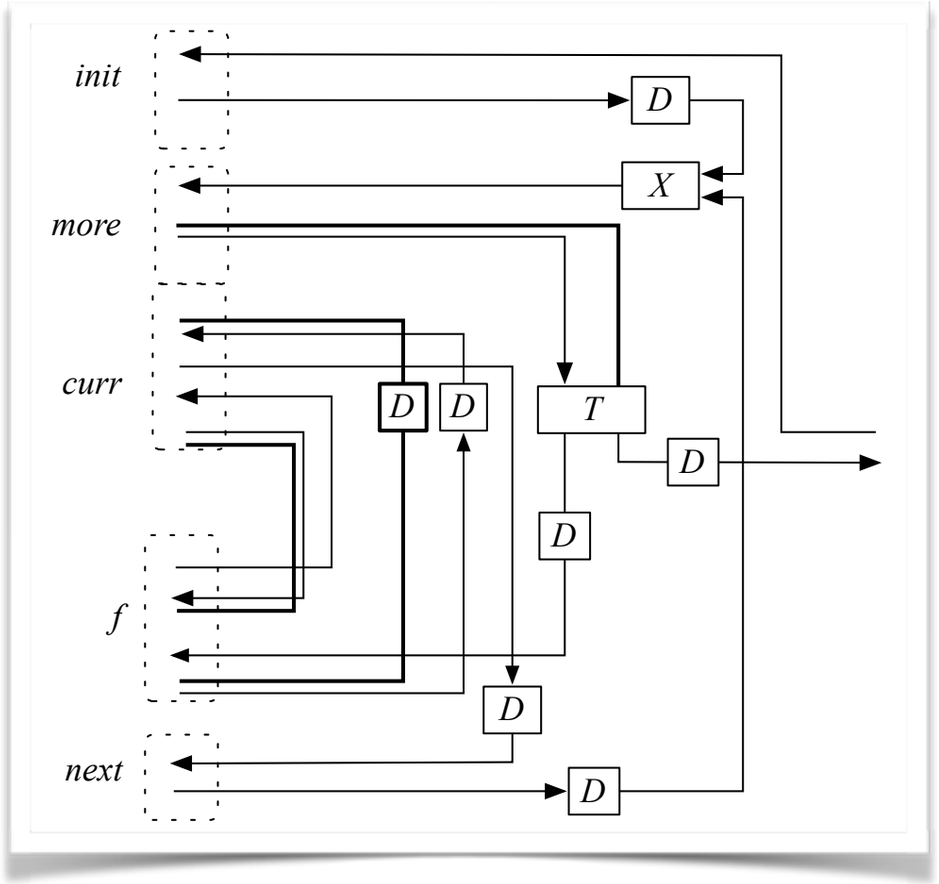
Ghica, Singh, Smith, ICFP‘11

An evaluation of HLS from GS



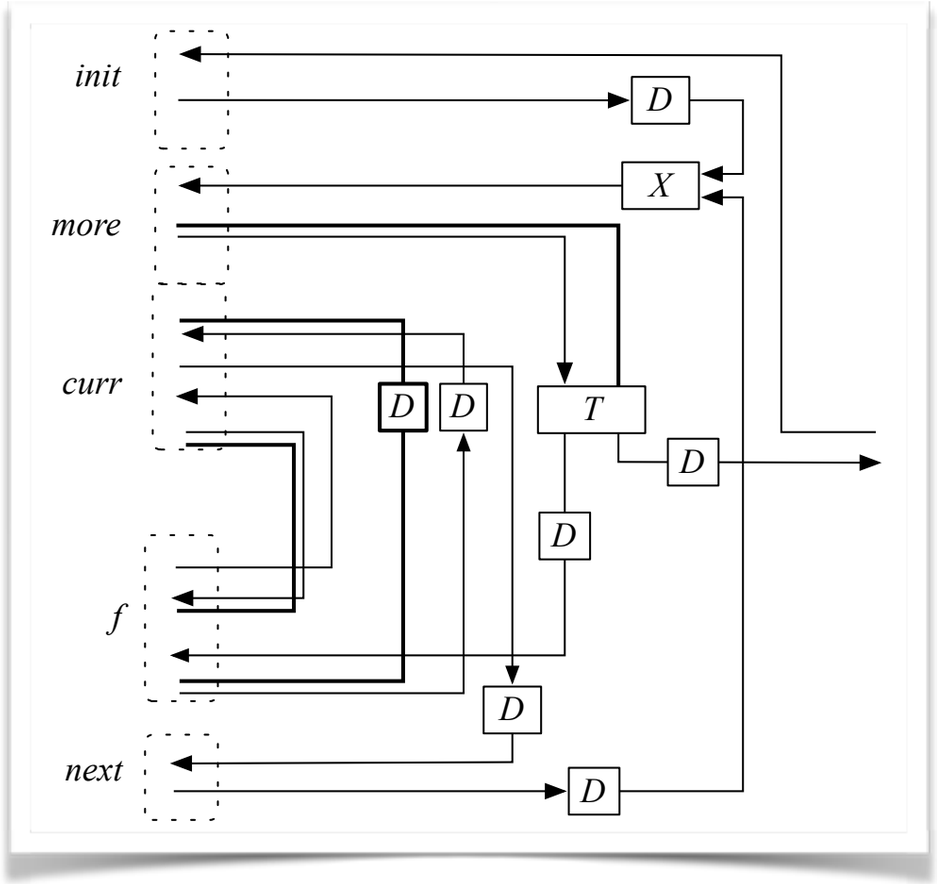
An evaluation of HLS from GS

- monoidal structure (+)



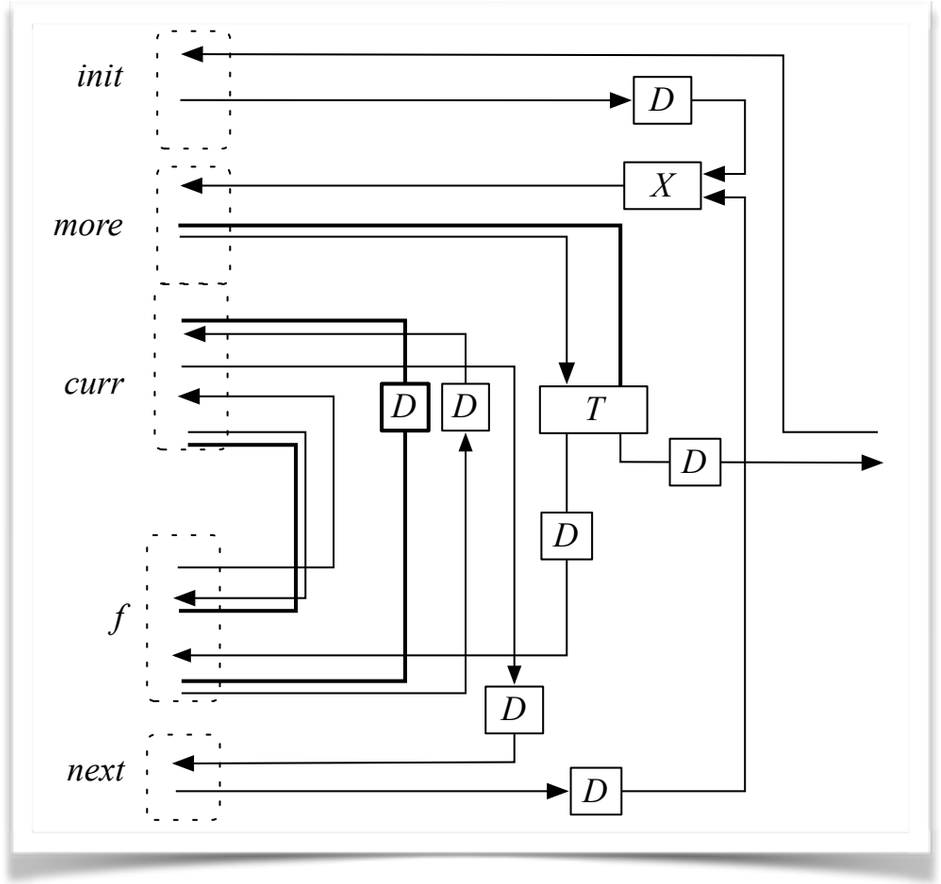
An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)



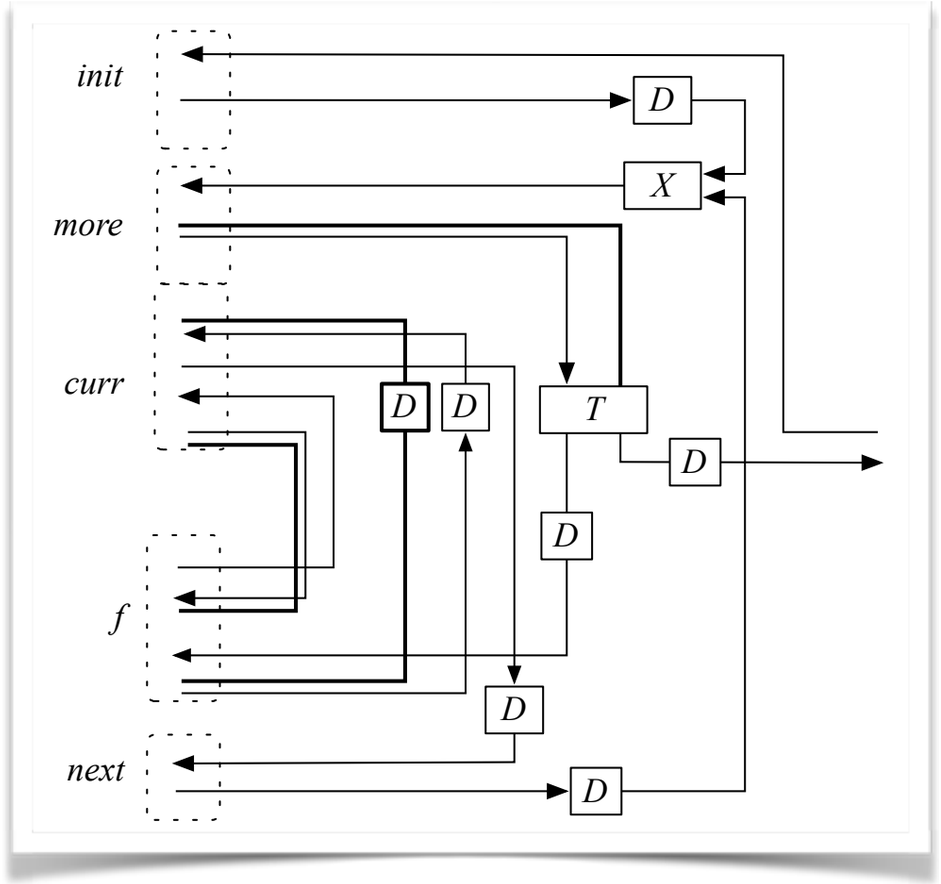
An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)
    - concurrency / parallelism (+)



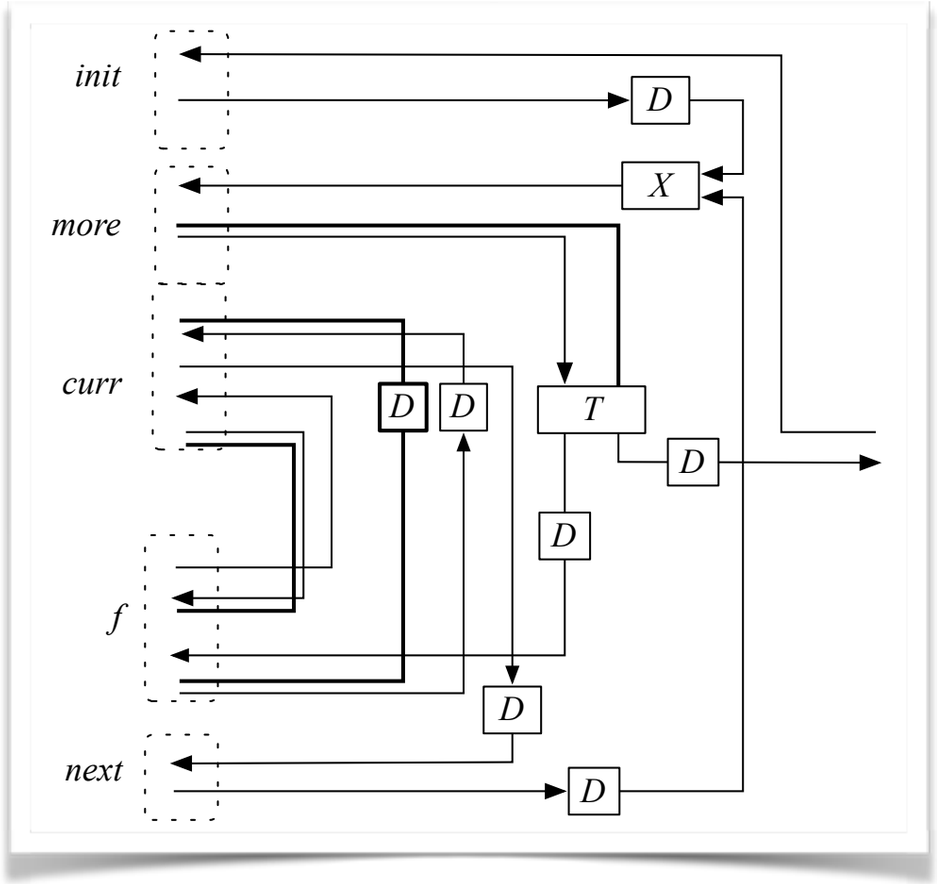
An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)
  - concurrency / parallelism (+)
- flexibility w.r.t. type systems (+)



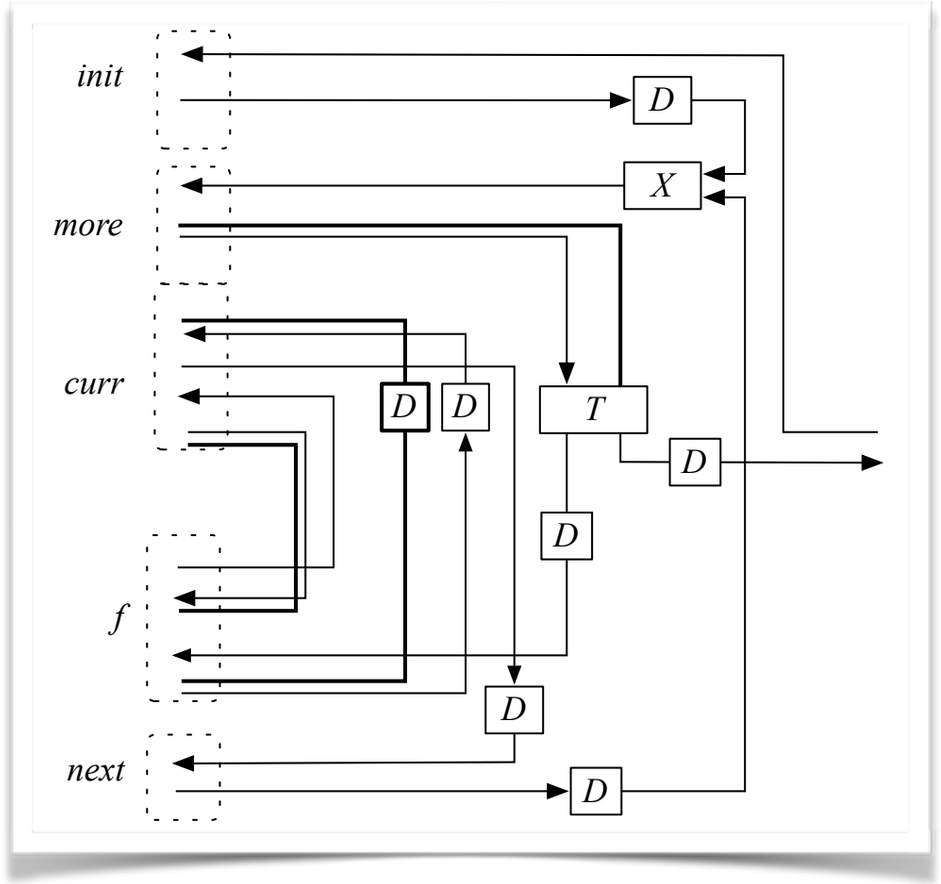
An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)
  - concurrency / parallelism (+)
- flexibility w.r.t. type systems (+)
- efficiency seems good (+)



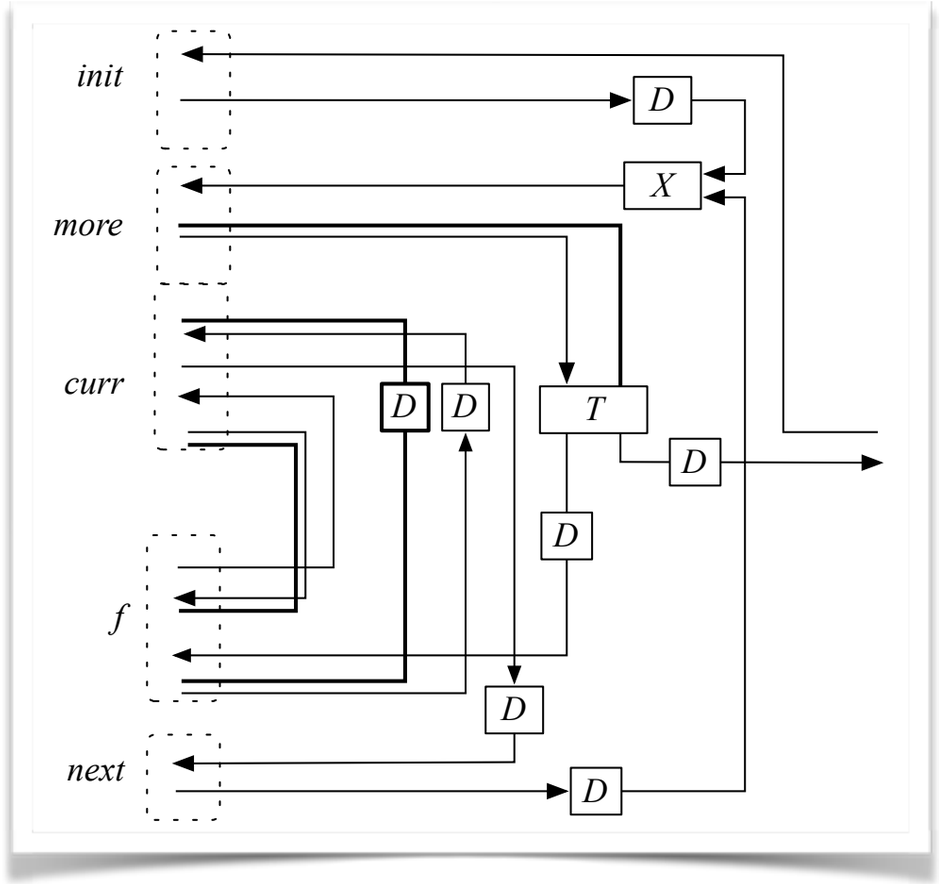
An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)
  - concurrency / parallelism (+)
- flexibility w.r.t. type systems (+)
- efficiency seems good (+)
- some optimisations required (?)



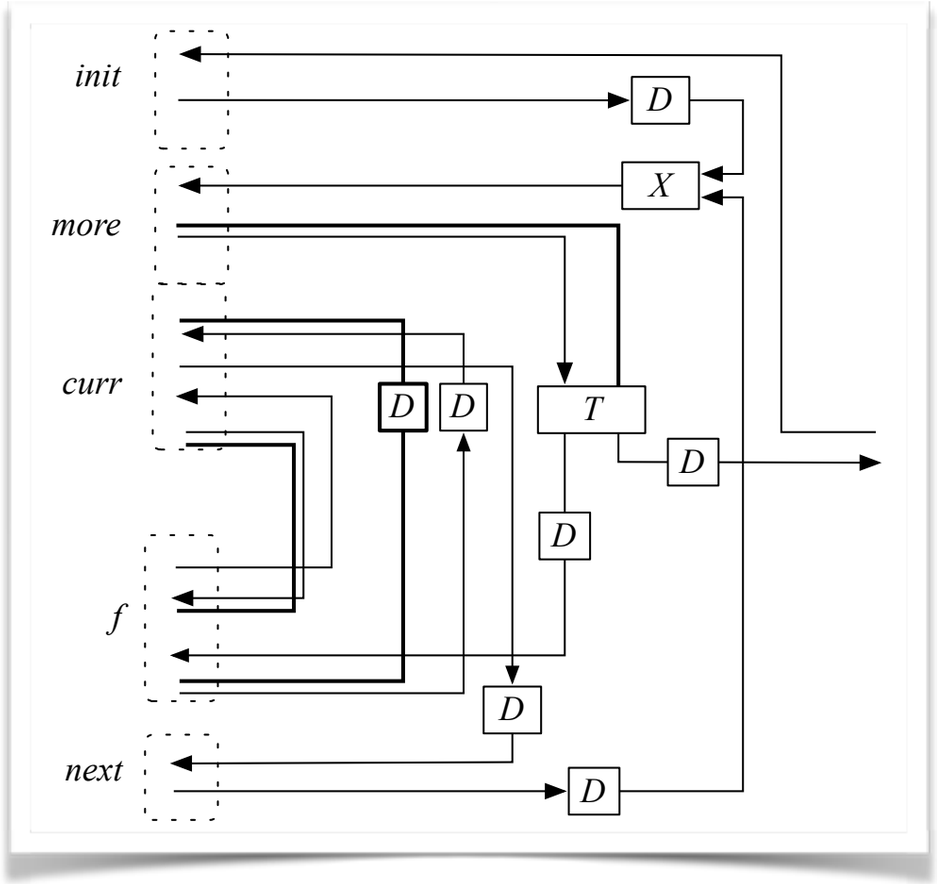
An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)
  - concurrency / parallelism (+)
- flexibility w.r.t. type systems (+)
- efficiency seems good (+)
- some optimisations required (?)
- extensible to heterogeneous compilation (+)



An evaluation of HLS from GS

- monoidal structure (+)
- interpretation of constants (+)
  - concurrency / parallelism (+)
- flexibility w.r.t. type systems (+)
- efficiency seems good (+)
- some optimisations required (?)
- extensible to heterogeneous compilation (+)
- a unique success: failed before many times



Distributed compilation of λ

The masquerade

$$(\mathbf{let}\, f\, =\, (\lambda x.\, x * x)\, @\, C\ \mathbf{in}\ (f\, 3 + f\, 4)\, @\, B)\, @\, A$$

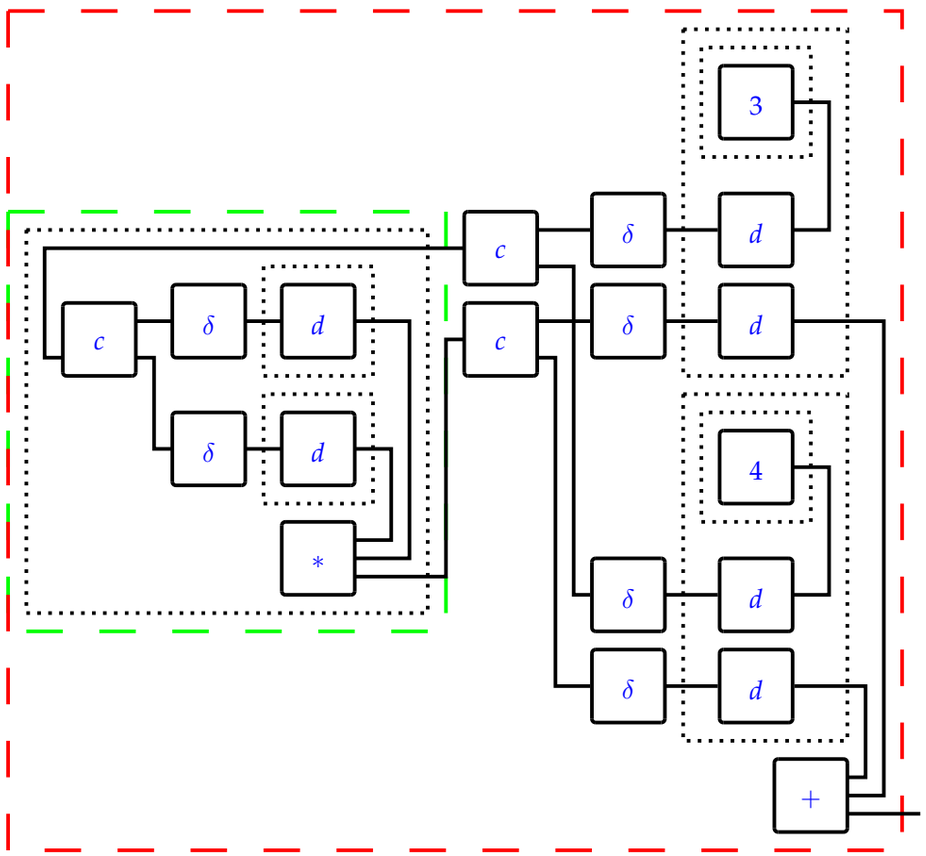
becomes this:

```
c() -> receive {Pid, X} -> Pid ! X * X end, c().
 b(A_pid, C_pid) ->
   receive
     request0 -> C_pid ! {self(), 3}, receive X -> A_pid ! X end;
     request1 -> C_pid ! {self(), 4}, receive X -> A_pid ! X end
   end,

  b(A_pid, C_pid).
main() ->
  C_pid = spawn(f2, c, []),
  B_pid = spawn(f2, b, [self(), C_pid]),
  B_pid ! request0,
  receive X -> B_pid ! request1, receive Y -> X + Y end
  end.
```

Compilation from GoI

$$(let\ f = (\lambda x.\ x * x)@B\ in\ f\ 3 + f\ 4)@A$$



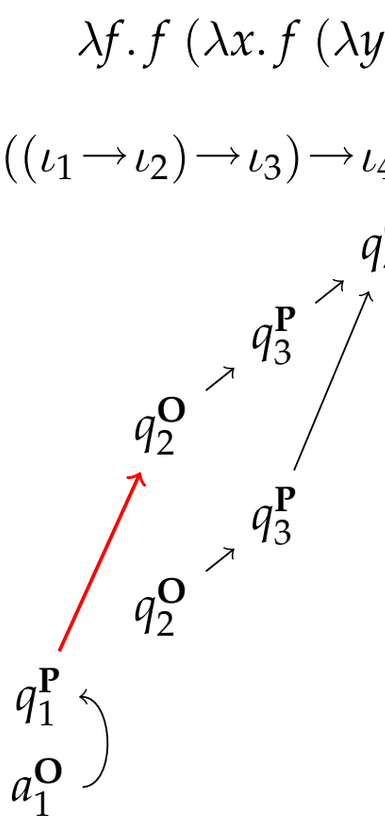
$$combine(M_1, M_2) = \langle (P_1 \cup P_2) \upharpoonright (\pi(M_1)\,\Delta\,\pi(M_2)),$$
$$(L_1 \cup L_2)[\mathtt{send}\ p/\mathtt{jump}\ P(p) \mid p \in \pi(M_1) \cap \pi(M_2)]\rangle.$$

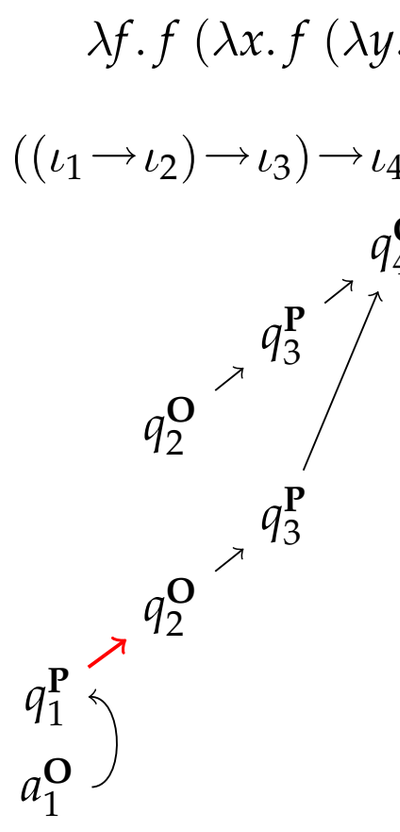
Fredriksson and Ghica. TGC'12

Good things	Bad Things
elegant and easy to implement	sequential (?)
extensible with a range of constants	call-by-name (?)
algebraic (peephole) optimisations	the stack is sent around

Compilation using games

Games are a trace semantics…



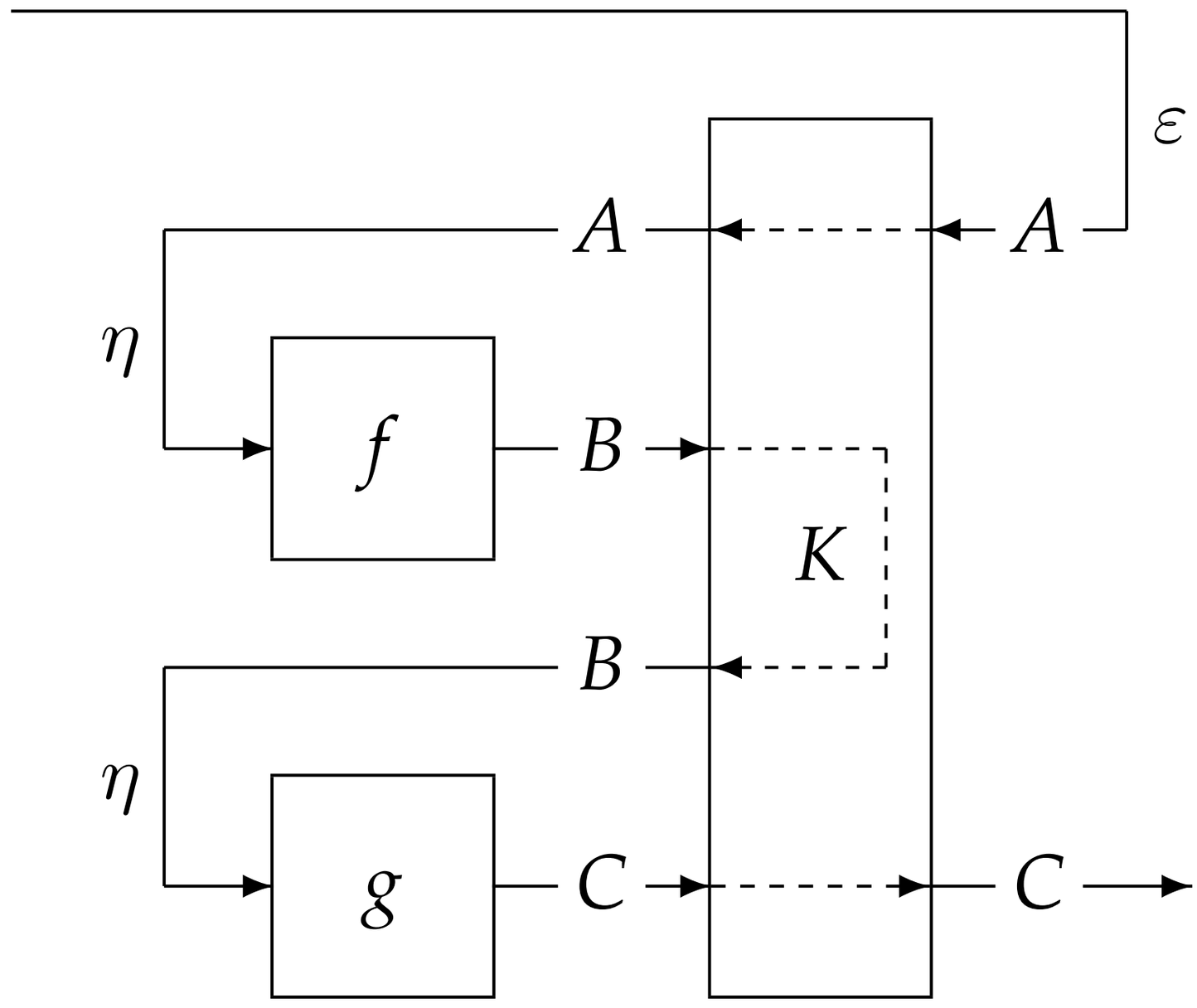


… of what machine?

$$
\begin{array}{rcl}
\text{Instr} & ::= & i \leftarrow \texttt{new}\, j, k \\
& \mid & i, j \leftarrow \texttt{get}\, k \\
& \mid & \texttt{update}\, i, j \\
& \mid & \texttt{free}\, i \\
& \mid & \texttt{flip}\, i, j \\
& \mid & i \leftarrow \texttt{set}\, j \\
& \mid & \texttt{ifzero}\, i\, c_1\, c_2 \\
& \mid & \texttt{spark}\, a \\
& \mid & \ldots
\end{array}
$$

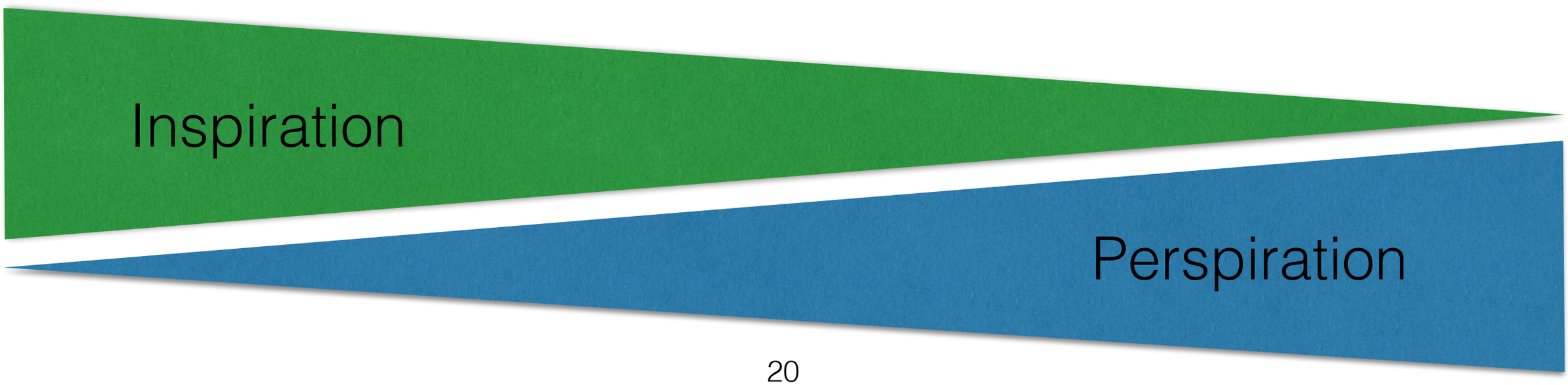
Fredriksson and Ghica. LICS'13

Composition wot?



Good things	Bad Things
parallelism / concurrency	awkward composition
virtually no runtime (gc)	call-by-name (?)
algebraic (peephole) optimisations	too much indirection
no GC	too much alloc and free

Distributing conventional abstract machines



Krivine & SECD machines

- add remote versions of all operations
- extend data structures with remote pointers
- prove correctness via simulation with single node machine
- machine-checked so we can sleep at night (Agda)

Fredriksson and Ghica. ICFP’14

Single-node performance

	`arith`	`fib`	`root`
Baseline	100% (0.34s)	100% (0.094s)	100% (0.009s)
GOI	3,042% (10.3s)	2,832% (2.7s)	20,222% (0.18s)
GAMC	765% (2.6s)	395% (0.53s)	356% (0.032s)
DKrivine	131% (0.44s)	141% (0.13s)	233% (0.021s)

Distribution overheads

	`arith`			`fib`			`root`		
	time	avg. size	max. size	time	avg. size	max. size	time	avg. size	max size
GOI	114% (11.6s)	107	172	4,017% (3.8s)	302	444	19,422% (1.7s)	717	1,312
GAMC	193% (5.0s)	20	24	1,481% (7.8s)	20	24	22,872% (7.3s)	20	24
DKrivine	140% (0.62s)	32	40	238% (0.32s)	32	40	890% (0.19s)	32	40

Good things	Bad Things
much more efficient	call by name
virtually no runtime	sequential (?)
less exotic	hard to extend

SECD (call-by-value)

	`trees`	`nqueens`	`qsort`	`primes`	`tak`	`fib`
FLOSKEL	91.2s	12.2s	9.45s	19.3s	16.5s	10.0s
`ocamlopt`	43.0s	3.10s	3.21s	6.67s	2.85s	1.68s
relative	2.12	3.94	2.94	2.9	5.77	5.95

Figure 1. Single-node performance.

	`trees`	`nqueens`	`qsort`	`primes`	`tak`	`fib`
μs/remote call	618	382	4.77	13.4	6.94	6.87
B/remote call	1490	25.8	28.1	27.0	32.0	24.0

Figure 2. Distribution overheads.

Good things	Bad Things
much more efficient	serialise/send data structs
local garbage collection	sequential (?)
less exotic	

Conclusions

Conclusions

- interaction-based compilation is inefficient

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad
    - too much jumping around (CPS-style)

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad
    - too much jumping around (CPS-style)
    - for HLS it doesn’t matter

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad
    - too much jumping around (CPS-style)
        - for HLS it doesn’t matter
- easier to prove correctness

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad
    - too much jumping around (CPS-style)
        - for HLS it doesn’t matter
- easier to prove correctness
    - compositionality

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad
    - too much jumping around (CPS-style)
        - for HLS it doesn’t matter
- easier to prove correctness
    - compositionality
    - seamless switch control / communication

Conclusions

- interaction-based compilation is inefficient
    - memory overhead not too bad
    - too much jumping around (CPS-style)
        - for HLS it doesn’t matter
- easier to prove correctness
    - compositionality
    - seamless switch control / communication
- opportunities for optimisation not explored enough

“More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all higher level programming languages.”

Go To Statement Considered Harmful
Edsger W. Dijkstra

“More recently I discovered why the use of the send statement has such disastrous effects, and I became convinced that the *send* statement should be abolished from all higher level programming languages.”

— the end —