Can Distribution Be (Statically) Calculated?

UNU/IIST — Seminar n.º 5
14 May 1997
Macau

J. Nuno Oliveira
University of Minho / INESC (Group 2361)
Braga

email: jno@di.uminho.pt

Abstract

- A potential application of software calculi such as SETS concerns **horizontal refinement**, that is, the decomposition of large software systems in terms of re-usable **components**.
- Horizontal refinement brings not only re-usability but also **distribution** and **concurrency** among the refined objects.
- Many laws of SETS work in this direction, particularly the ones which *push "$\times$" outwards*, possibly leading to exponentiation.
- This talk reports on on-going research which seems to provide evidence that the "amount of concurrency" gained in a **horizontal refinement** step can, if expressed in SETS, be calculated and reasoned about.
- Hash-tables are provided as an illustration of the exponentiation type of concurrency. A hashed data-structure is factored into a collection of n-similar (sub) structures — a kind of "fractal" effect on data structuring — which can be regarded truly concurrent processes.
- Regarding the hash-effect as mere data optimization is thus too limited a view. Instead of "data hashing" one should talk about "process hashing".
- This talk has to do with on-going research on extending SETS to processes, hopefully leading to a SETS-based **Process Refinement** calculus.

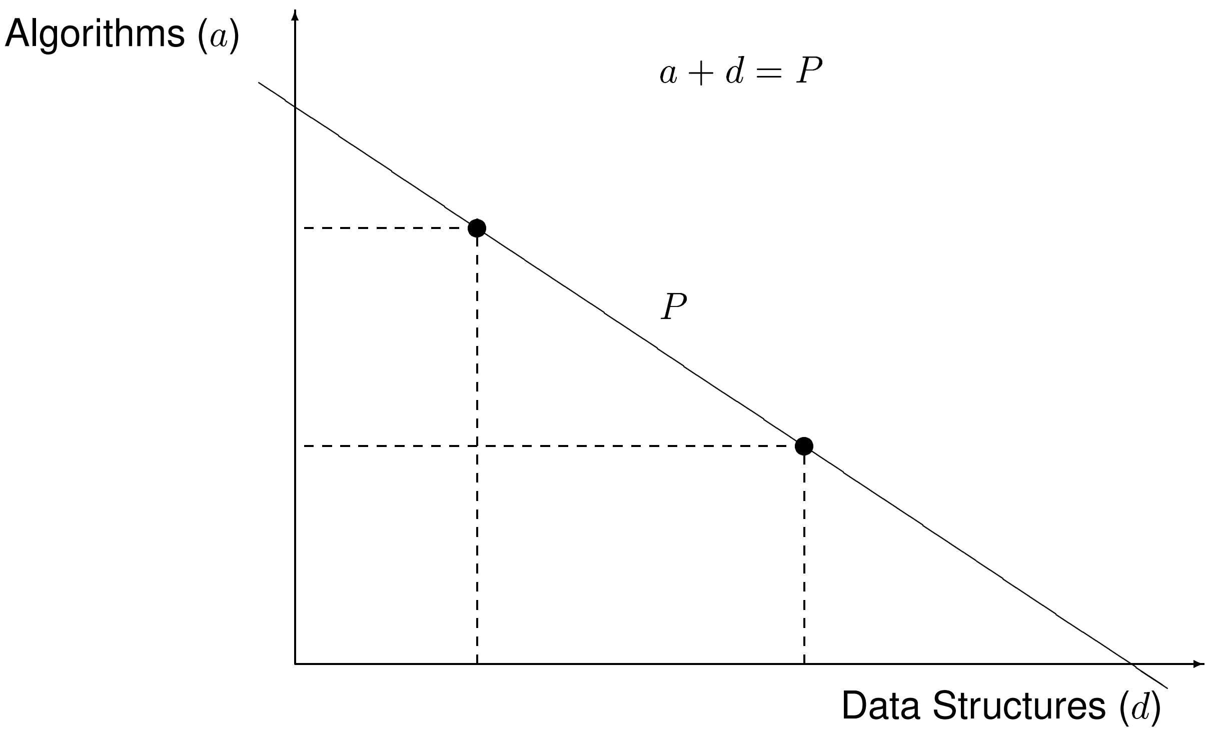
Motivation

- SETS is a data-structuring calculus.
- We want to move from static data to processes and develop a SETS-extension towards process calculation.
- Is this sound? Do processes have anything to do with data?

The $a + d = P$ equation

Recall title of Niklaus Wirth's famous book:

Algorithms + *Data Structures* = *Programs [Wir76]*



We may read it as an information-preserving "linear" equation...

$$a + d = P$$

...meaning:

> For a given problem P we want to solve by software, the "sum" of the overall "algorithmic load" (a) and the "data structuring load" (d) remains the same whatever correct solution you devise for P.

Well-known Example

LL parsing: the following are two i/o-equivalent realizations of the same specification, that of *parsing every sentence of LL-language $\mathcal{L}$*:

- The d-biased version ($a \rightarrow 0$) — a very small predicative parser (a stack automaton) + a large and smart parsing-table.
- The a-biased version ($d \rightarrow 0$) — the recursive descent parser, *i.e.*, a large system of mutually-recursive functions + insignificant data structures such as, *e.g.*, `cur_token`.

A better Example

Heard from Chris George at UNU/IIST: computers are so finite when combinatorial problems are at stake that the following realizations of factorial are i/o-equivalent:

- a-biased version ($d \rightarrow 0$) — a rather sophisticated, doubly recursive algoritm running on a multiprocessor,

$$fac(n) \stackrel{\text{def}}{=} df(1, n)$$

$$df(a, b) \stackrel{\text{def}}{=} \begin{cases} a \geq b & \Rightarrow \quad b \\ a < b & \Rightarrow \quad let \quad m = div(a + b, 2) \\ & \qquad\; in \quad df(a, m) \times df(m + 1, b) \end{cases}$$

  + no data structures.

- d-biased version ($a \to 0$) — a large but still finite look-up table (filled in as much as `maxint` allows us...),

n	$n!$
0	1
1	1
2	2
3	6
$\vdots$	$\vdots$

+ the trivial array indexing operation.

Conjecture

(Or "Lavoisier principle" re-stated):

$$\boxed{\text{Data-structures and algorithms are inter-changeable.}}$$

But, is this true? What about considering...

... the $e = mc^2$ "equation"?

In Portuguese,

- *estado* means *state*
- *comportamento* means *behaviour*

So,

$$e = mc^2$$

brings, by (a highy conjectural and unlikely) analogy, something like

$$\boxed{\text{State and behaviour are (quadratically) proportional.}}$$

to mind.

Not very good news — if true — for "state-minded" people like the author of these slides: behaviour would be quadratically more expressive than data-structuring...

State versus Behaviour

Certainly state static structure (Q) induces behaviour:

$$behaviour(Q; \{pre/post_1, \ldots, pre/post_n\}) \stackrel{\text{def}}{=}$$

the most liberal operation activation sequencing (set of traces) such that neither the state invariant nor any pre-condition is violated.

So,

- **Monotonicity**: imposing (or strengthening) a state invariant ϕ ($Q_\phi \subseteq Q$) will have a proportionally negative impact on behaviour (if it does not cancel any of the operations!):

  $\boxed{\text{the smaller the state the more constrained the behaviour}}$

- **Infimum**: $Q \cong 0$ exhibits no behaviour at all — recall CCS [Mil89], *etc.*
- *etc.*

Main idea

Behavioural refinement induced by SETS-based state-refinement, something like

$$\frac{A \trianglelefteq B}{b_\Sigma(A) \leq b_\Sigma(B)}$$

where $b_\Sigma(Q)$ abbreviates $behaviour(Q; \Sigma)$.

In summary,

- We want $\trianglelefteq$ to induce $\leq$, whatever this is — it certainly has to be more than *action refinement* [AH88].
- Put in another way:

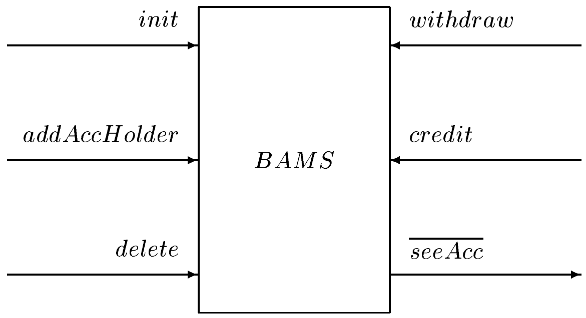
  Want to refine c such that $c = b_\Sigma(A)$?

  Ok, just do your calculation of B such that $A \trianglelefteq B$, and then calculate $b_\Sigma(B)$.

Tentative Illustration

Recall $BAMS$ from on-going SETS course equipped with the following set Σ of operations:

- $addAccHolder$ — we have seen this
- $init$ — it initializes the database
- $credit$ — deposit of an amount of money in some valid account
- $withdraw$ — it withdraws money from some valid account with an appropriate balance
- $delete$ — it removes accounts with balance=0
- $seeAcc$ — it shows all the information about a given account



We sketch the behaviour of $BAMS$ under Σ-operations as follows (please accept the abuse of notation):

$$b_\Sigma(BAMS) = init \bullet (\mu c. \sum_{\sigma \in \Sigma \wedge \text{pre-}\sigma} \sigma \bullet c)$$

Now, recall the refinement of $BAMS$ — for our purposes, it is enough to stop at level 3:

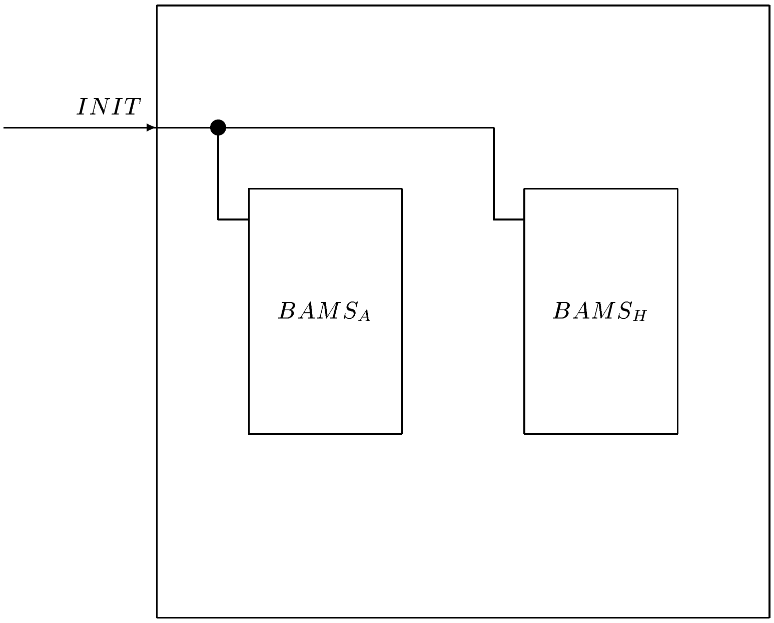
$$\begin{array}{rcl} BAMS & = & AccNr \hookrightarrow (2^{AccHolder} \times Amount) \\ & \cong_1 & AccNr \hookrightarrow (Amount \times 2^{AccHolder}) \\ & \cong_2 & AccNr \hookrightarrow (Amount \times (AccHolder \hookrightarrow 1)) \\ & \trianglelefteq_3 & \underbrace{(AccNr \hookrightarrow Amount)}_{BAMS_A} \times \underbrace{((AccNr \times AccHolder) \hookrightarrow 1)}_{BAMS_H} \\ & = & BAMS_3 \end{array}$$

What do we know at this point?

Not only that

$$BAMS_3 \cong BAMS_A \times BAMS_H$$

holds, but also that we have just started to calculate a distributed version of $BAMS$:



Intuitively, there is now more "room" for behaviour — for instance, events such as

$$withdraw \times addAccHolder$$

— recall the $f \times g$ construct, our abstract notation for "f concurrently with g" — or such as

$$credit \times addAccHolder$$

can be added to the original $BAMS$ behaviour, but not

$$credit \times delete$$

that is, we have added extra behaviour to the system at refinement level 3:

$$b_\Sigma(BAMS_3) =$$
$$init \bullet \mu c.((\sum_{\sigma \in \Sigma \wedge \text{pre-}\sigma} \sigma) + (withdraw + credit) \times addAccHolder + \cdots) \bullet c$$

Calculating Added Behaviour

The illustration above is a particular case of a general situation in SETS:

- All laws of SETS which "push the product construct ($\times$) outwards" generate concurrency or distribution, because they help in factorizing complex state structures of monolithic components into collections of simpler and relatively independent structures.
- What is the gain in concurrency? Can it be statically calculated? In some degree, yes:

  Stop refinement at each application of a law in the above situation, *e.g.*

  $$A \hookrightarrow D \times (B \hookrightarrow C) \trianglelefteq (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{1}$$

  or

  $$A \hookrightarrow (B \times C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C)$$

  For each operation σ of starting specification, the use functorial calculus to see if, at such as data-structure splitting level, one obtains:

– $\sigma = id \times \sigma_2$, for some σ_2 — in this case, from this stage downwards, σ is an operation "private" to the right-hand component, because, whatever abstraction function f is applied in subsequent calculations, we will always be the case that $id \circ f = f$.
– $\sigma = \sigma_1 \times id$, for some σ_1 — the same as above, with respect to the left-hand component.
– $\sigma = \sigma_1 \times \sigma_2$ and both σ_1 and σ_2 are not id functions — in this case we have a "shared" or "synchronizing" operation.

- extra behaviour consists of all $f \times g$ resulting from this process such that f is private to the left-hand side component and g is private to the right-hand side component.

Important: such extra behaviour is observable.

Technical Details

- Wherever A is reified into a product $B \times C$, we also have that $B \times C \trianglelefteq (B+1) \times (C+1) \cong 1 + B + C + B \times C$.
- The choice in the above co-product may be understood *dynamically*: interaction with the whole system ($B \times C$) or just with one of its components (B or C) is possible, provided, in the last case, one restricts oneself to the operations which do not interfere with the other component.
- By a similar argument, a product of n factors may be refined into a coproduct of all the possible partial products. Given the the set of (simulations) of the associated events, our first step consists of formalizing the notion of subsets of events indexed by the state space factors involved in its computation.
- This may be expressed as a *fibration* based on Set from a category E_M whose objects are the events in the model M and all their simulations in the concrete state space and arrows are pairs of functions that make the simulation diagrams commute.

Non-observable $\trianglelefteq$-induced Behaviour

Is this the only extra behaviour obtained by $\trianglelefteq$-refinement?

No, because of data redundancy. The 1_Q function, for instance — which is allways defined for whatever component we may think of — is now implemented by whatever low-level operation σ_1 which statisfies

$$f_Q(\sigma_1(q_1)) = f_Q(q_1)$$

for f_Q the state abstraction function:

$$\begin{tikzcd} Q \arrow[r, "id"] & Q \\ Q_1 \arrow[u, "f_Q"] \arrow[r, "\sigma_1"'] & Q_1 \arrow[u, "f_Q"'] \end{tikzcd}$$

For instance, in the "sets represented by lists" refinement, via $elems$, both the list-operation which removes **duplicates** from the state or the one which **orders** the state list according to same ordering are not observable at specification level, that is, both refine the identity function on sets.

Because

$$1_{A\times B} = 1_A \times 1_B$$

the number of combinations of different "house-keeping" concurrent operations which, at low-level, may run without being perceived at high-level grows as abstraction functions become less injective.

Other dimensions of process refinement

From a purely static standpoint, it can be shown [Oli97] that $\trianglelefteq$-calculations can only increase:

- data redundancy

- operation definition
- operation determinism

Exponential Distribution and the Hash Effect

Fact

$$\underbrace{A \times \ldots \times A}_{n} \;\cong\; A^n \tag{2}$$

invites us to iterate our intuitions towards networks of processes “of the same type” (A).

Hash tables — $(2^A)^n$ — provide a good example of this type of concurrency. In fact, the scope of the “hash effect” is much wider that $(2^A)^n$: it can be applied to any data structure $\mathcal{G}$ which satisfies the following clause:

$$\mathcal{G}(n \times A) \;\trianglelefteq\; (\mathcal{G}(A))^n \tag{3}$$

for $n > 0$ and a given hash function $H : A \longrightarrow n$, the basis for local invariant

$$\mathcal{G}(\lambda\langle i, a\rangle . i = H(a))$$

But we know that

$$(\mathcal{G}(A))^n \;\cong\; \underbrace{\mathcal{G}(A) \times \ldots \mathcal{G}(A)}_{n}$$

which means that a hashed structure, regarded as a monolithic structure, can at compile-time be factored into a collection of n similar (sub) structures — a kind of “fractal” effect on data structuring.

From another perspective, this can be viewed as a step in the *horizontal refinement* [Gog86] of a single state-based software component into a network of n “smaller” components (each behaving like its ancestor), which can be regarded as truly concurrent processes.

How much “concurrency” do we gain with this kind of *horizontal refinement*?

A lot, in particular due to the local invariant we have imposed relative to a fixed hash function H.

The extra (observable) behaviour is the fact that operations on disjoint collision sub-structures are independent and can overlap in time (*e.g.* the operation of adding a synonym of hash index i can proceed in parallel with another one which is removing a synonym of hash index j, for $i \neq j$).

In summary, regarding the hash-effect as mere data optimization is a limited view of hashing. Instead of "data hashing" one should talk about "process hashing", a topic addressed in [OBM97].

Final Topics

"Row" Concurrency versus "Column" Concurrency

SETS may also play a rôle in characterizing different types of distribution and concurrency:

- The line of inquiry sketched above may be called "column" distribution, by this meaning that (maximal) sets of columns in the original product are identified, autonomous processes being assigned to compute the services intended over those "chunks" of the state space.

  So we are dealing with "range" columns:

  $$A \hookrightarrow (B \times C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C)$$

- There is however a different way to introduce concurrency in a model, which relies in the fact that the behaviour of some models is uniform over the structure of the state space.

  For example, wanting to sum all numbers in the range of a large mapping $A \hookrightarrow \mathbb{N}$, one may think of a "divide-and-conquer" algorithm:

find a good way of splitting A into A_1 and A_2 such that these are disjoint. Then set up two concurrent processes, one taking care of the sum operation concerning A_1 and the other concerning A_2, and recursively repeat this as long as the underlying hardware can support it.

The binary split process is captured by

$$A \hookrightarrow B \trianglelefteq (A_1 \hookrightarrow B) \times (A_2 \hookrightarrow B)$$

Since A_1 and A_2 are disjoint, we have $A_1 \cup A_2 \cong A_1 + A_2$. So the SETS law supporting "divide-and-conquer" parallelism is

$$(A + C) \hookrightarrow B \trianglelefteq (A \hookrightarrow B) \times (C \hookrightarrow B)$$

Note the $+$ versus $\times$ duality

(This kind of concurrency is typical of *e.g.* image processing)

Conclusions

There is a lot of intuition about concurrency which SETS can formalize provided we find the right category for that — maybe SETS will be one day called TOPOS [McL85].

Related Work

We are also trying to extend SETS functoriality to processes.

But, what is a process?

- process algebras ?

- Petri nets?
- temporal logic ?
- *etc.*

Acknowledgement

The contribution of Luís Barbosa to the ideas presented in this talk is gratefully acknowledged.

References

[AH88] L. Aceto and M. Hennessy. Towards action-refinement in process algebras. Technical Report CS 3/880, University of Sussex, 1988.

[Gog86] J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.

[McL85] C. McLarty. *Elementary Categories, Elementary Toposes*. Oxford Logic Guides n.º 21. Calendron Press, Oxford, 1^{st} edition, 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.

[OBM97] J. N. Oliveira, L. S. Barbosa, and F. S. Moura. Can distribution be (statically) calculated? Technical report, DI/INESC, 1997. (In preparation).

[Oli97] J. N. Oliveira. A note on $\trianglelefteq$-calculation, nondeterminism and implementation bias. Technical report, DI/INESC, 1997. (in preparation).

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.