DFRO - 0

EWD435 - 0

Uncorrected flat typing errors

Associons: an effort towards accommodating potentially ultra-high concurrency.

by

Edsger W.Dijkstra, W.H.J.Feijen and M.Rem

The following is a research proposal to investigate the linguistic consequences when it is desired to program for a machine that could work with an ultra-high degree of concurrency. The subject has been suggested by the advent of LSI-techinques that store large amounts of information on essentially active components: once in the far future we may be invited to think of algorithms instructing machines, such that the major part of the logical manipulations will take place distributed all through "the store".

It is felt that the above invitation could have a deep linguistic consequence: if the purpose of the whole game is a drastic reduction of computation time, we may expect both a reduction of the number of "instructions" to be executed and of the number of "instructions" to be written down. One way of trying to achieve a similar goal is the introduction of large fancy data-types, upon which numerous, powerful operations are defined, the implementation of which allows a considerable --"internal", if you wish-- concurrency. Efforts along those lines that we are aware of, are, however, on this macroscopic scale still purely sequential, at each moment only a very small number of such fancy operands will be actually involved in the computational process. We would like to go a considerable step further, at each moment involving "all of the storage contents" so to speak in the activity.

To achieve the high degree of concurrency by asking the programmer to regulate --i.e. synchronize etc.-- explicitly the co-operation between a huge number of possibly all different concurrent sequential processes, seems a dead alley in the sense that the implied programming task will quickly exceed our abilities. It seems more attractive to look for a simple and systematic instruction repertoire such that each "instruction" can interfere in a homogeneous fashion with the total contents of the store. Its interference

must be homogeneous, because otherwise there is no hope that we, as programmers, will be able to manage the beast.

In order to avoid misunderstandings, we would like to point our that it is most emphatically not our intention to design a machine. Our "instruction repertoire" can, of course, be interpreted as the functional specifications of a machine; in choosing this instruction repertoire, however, we do not feel ourselves obliged to restrict ourselves in such a way that the corresponding machine could be built realistically with current-day technology. We are perfectly willing to consider functional specifications that would make most hardware designers shudder at the thought of having to build something meeting them! Our primary concern is that the machine will be manageable from the programmers point of view.

In one specific aspect we feel ourselves obliged to be kind to the poor hardware engineer or --to put in another way-- to leave the door open: our goal is to write down understandable programs under control of which a highly concurrent activity is possiblie, but not obligatory. An (obvious) underlying desire is to arrange our programs in such a fashion that, in spite of the high potential concurrency, most of the in the mean time developed techniques for the programming of sequential processes remain applicable. (We expect to maintain the "semicolons" but would like powerful "statements" in between!)

As understandability is one of our primary concerns, we shall certainly at the start allow ourselves the luxury that is characteristic of the so-called "higher programming languages", viz. arbitrarily complicated expressions as components of our commands. In classical environments it is well-known how the evaluation of an arbitrarily complicated (algebraic) expression can be broken down sucg that a (very) finite arithmetic unit can produce the result. For the time being we just hope that the "breaking down" of our expressions will not present unsurmountable logical difficulties.

* * *

Highly associative techniques seem indicated for two reasons. Firstly, associative techniques are a way of distributing an activity (viz. comparing)

all through the store; secondly, the potential but not obligatory concurrency could very well make the storage management problem as is cuased by a store consisting of explicitly addressed cells, unmanageable. The desire to use something that smells like "association" is a direct consequence of our desire to broadcast commands that all through the store will be processed in a "homogeneous" fashion.

Having abolished addresses, we have to introduce "names". We assume the machine able to deal with (values of) variables of type "name", we assume the cardinality of this type to be high enough. "To deal with" will certainly include the ability to test the equality of two names (as with all types!). Assuming all entities to be referred to, to be indentified by mutually distinct names, any relation between some of these entities can be represented by a relation between their names. If "p_1" through "p_{100}" stand for the distinct names of 100 different persons, we may have the relations

fatherof(p_i, p_j) meaning: "p_i is the father of p_j" and

olderthan(p_i, p_j) meaning: "p_i is older than p_j" .

(We then know, for instance, that for any i and j

fatherof(p_i, p_j) $\Rightarrow$ olderthan(p_i, p_j)

buth not the other way round!)

Instead of representing "all sorts of relations" --such as "fatherof" or "olderthan"-- we choose the more general --and neutral!-- technique of considering different relations as "named entities" as well --e.g. named by "fatherof" and "olderthan" respectively--, leaving us with a single universal relations --which, therefore, can remain anonymous-- and represent "(fatherof, p_i, p_j)" and "(olderthan, p_i, p_j)".

(Note that we could also have "(implies, fatherof, olderthan)" ! This is to stress that what from one point of view was regarded as "a relation", from another point of view can be regarded as "an argument".)

Such an ordered n-tuple of names is called: an "associon". The contents of the store is considered to be an unordered set of (different) associons.

The presence of an associon in store will be interpreted as the truth of the universal relation applied to its arguments, i.e. the members of the n-tuple.

Note 1. For the time being we shall not worry about arithmetic. If so desired we could postulate --i.e. bring into store-- the following collection of associons:

(integer, zero)	(nought, zero)
(integer, one)	(suc, zero, one)
(integer, two)	(suc, one, two)
etc.	etc.

as many as we like. (End of note 1.)

Note 2. Many relations are symmetric. As the possibilities are numerous, we shall not make up our mind now. For the time being, we can assume that whenever "(asoldas, p_i, p_j)" is in store, "(asoldas, p_j, p_i) will be in store as well. (Alternatively, we could store besides the specific associon "(asoldas, p_i, p_j)" the general "(twosym, asoldas)".) (End of note 2.)

Names occurring as elements of associons in store may occur as constants in our program texts: this facility should enable us to refer to subclasses of associons, e.g. all 3-tuples with "fatherof" in the leading position. We may expect that as the computation proceeds, new names have to be generated: such a new name will be different from any name occurring anywhere in an associon in store or as a constant in our program. We can restrict the rule to "different from any name occurring in an associon in store" when reading in a program , in which the name "fatherof" occurs, gives rise to an 1-tuple associon "(fatherof)" in store. As the creation of associons must anyhow be something that can be ordered by a program, its opening statement could start by greating these associons, thereby reserving the unique meaning of its constants.

* * *

The presence of associons in store is interpreted as recorded truths of facts. The evaluation of a computation is view as the creation of new associons, recording the truths of facts that are implied by already known truths. One truth is fairly universla, it is the truth of "true"; this will

be recorded by the irrevocable presence in store of the empty associon "()". When we refer to "an empty store", this means "a store containing only "()"".

Besides creating new associons, we shall also --"to save storage space!"-- cater for the destruction of associons. This may represent the abolishment of records of truths still valid, but no longer of any interest --e.g. at the end of a computation, an abolishment, very similar to the traditional block exit--, it may also represent that, from now onwards, a (transient) interpretation is no longer valid. In all probability, this second need for associon destruction will only emerge as soon as explicit repetitive mechanisms are introduced.

Note. The insertion "to save storage space" was not made jokingly: destruction of information seems characteristic of all non-trivial machine usage. (End of not.)

Above, we have said that the creation of new associons would take place as a recording of truths implied by already recorded truths. To do justice to this observation, we propose --as a rather fundamental language construct-- to use the implication for that purpose. Creating the two associons "(fatherof)" and "(olderthan)" could take place by

() => (fatherof) and (olderthan)

or by

() => (fatherof); ()=> (olderthan) .

The idea is that, upon completion of such a statement the "stated implication" holds. If the implication holds to start with, it will act as the empty statement, if not, however, it will react to it by creating missing associons as mentioned at the right-hadn side and not by removing "()". (This is not unlike the asymmetry of the ALGOL 60 assignment statement "x:= y".)

Note. For the time being we assume that the semicolon indicates successive execution in the usual way. (End of note.)

The Sheffer Stroke suggests, that we must be able to negate as well; the negation of the left-hand side presents no problem, the creation of

"(fatherof)" could then also be prescribed by the (longer) statement:

non (fatherof) => (fatherof) .

If "(fatherof)" is present, the implication holds; otherwise, in order to make the right-hand side true, the associon "(fatherof)" is created, whereafter the implication holds. (That in the mean time, the left-hand side has become false, is an admissible side-effect, that does not invalidate that the implication still holds.)

* * *

The above suggests a notation for destruction of associons, viz. precede by a negation at the right.hand side, e.g.

() => non (fatherof) and non (olderthan)

or () => non (fatherof); () => non (olderthan) .

Just as the creation can be described without using the universal truth "()", so can the destruction, viz.

(fatherof) => non (fatherof) .

* * *

In our previous examples the liberty we had when positive terms at the right-hand side could be created or negative terms at the right-hand side could be destroyed was, that as an equation, the implication had only one solution. But what, for constants "A", "B" and "C", about

(A) => (B) ?

If initially (A) holds, and (B) doesn't, an associon "(B)" will be created, but when initially (A) is false, both presence and absence of (B) would satisfy the implication. Viewing (B) as stating a fact that can be concluded from the truth of (A), it is not allowed to create (B) --establish that truth!-- when (A) does not hold. If, however, the truth of (B) had already been established otherwise, it will remain so. In other words: if (A) does not hold, (B) is left as it is and the little program:

(A) => (C); (B) => (C)

it equivalent to (A) or (B) => (C) .

In other words (A) => (B) will only change the store contents, when initially

(A) and non (B) holds. As long as constant are involved, the above shows that we do not need the or as a connective at the left-hand side. The connective or at the right-hand side, e.g.

(A) => (B) or (C)

is still more misplaced: even if the implication is true, we may not arbitrarily conclude either the truth of (B) or (C) or of both. For the time being, the connective "or" will not be used anymore. (The introduction of or at the right-hand side is certainly not a good way of introducing non-determinacy.)

* * *

Also the and at the right-hand side should be used with precaution, such as is shown by the trivial example

() => (A) and non (A) .

For the time being we shall therefore not use the and at the right-hand side anymore. (Time will show!)

Our format for the left-hand side is now reduced to a conjunction of terms, for the right-hand side to a single term, where a term is a possibly negated associon.

* * *

(To be continued.)
18th July 1974