Separation of Correctness and Complexity in Algorithm Design

Michel Chaudron, chaudron@cs.leidenuniv.nl
Department of Computing Science
Leiden University
P.O. Box 9512, 2300 RA Leiden

11 May 1994

1 Introduction

In this paper we propose a new approach to the design of algorithms. This approach is based on the view that all algorithms are composed of a computation and a control component, and that these components can be designed separately. The computation component is responsible for the correctness of an algorithm. It embodies the computational knowledge about a problem domain that is needed to solve the corresponding problem. The control component governs complexity aspects of the solution method by directing the usage of the computational knowledge. This bisection allows the problem of how to construct an algorithm to be split into two smaller problems: "What are the elementary units of computational knowledge for the problem at hand?" and "In what order should these units be used to (efficiently) obtain a solution." This way, the concerns of correctness and complexity are separated. We assert that there are many advantages in separating correctness and complexity in algorithm design. Hence, we examine properties of programming formalisms that influence the possibility of separating the design of the computation and control components of an algorithm.

The paper has the following structure. Section 2 illustrates that an algorithm contains parts that are solely related to the correctness, and parts that are primarily related to the complexity of algorithms. In section 3 we argue that the proposed view of separately designing computation and control leads to an improvement over the existing methods. In section 4 we investigate in what form computation and control appear in existing programming paradigms. We look at an example program expressed in different formalisms and explore which features of programming languages influence the entanglement of computation and control. This leads to the identification of aspects of programming languages that cause this entanglement.

2 Exposing the Distinction between Computation and Control

In this section we study some examples that illustrate the possibility of identifying a computation component and a control component in algorithms. These examples are summation and sorting. The notation of quantified operators is copied from [CM88].

2.1 Separating Computation from Control in Summation

We look at the problem of computing the sum of n numbers: $s = (+i : 1 \leq i \leq n :: a_i)$. It is clear that being able to compute the sum of two numbers is sufficient to solve the problem of computing the sum of n numbers. Informally speaking, we postulate that the elementary unit of computational knowledge is captured by the following rule:

"Select any pair of numbers, and compute their sum."

Application of this rule decreases the amount of numbers to be added by one. The problem is solved by repeatedly computing sums of pairs of numbers until there is only a single number left. This remaining number must be equal to the sum of the numbers that made up the initial sequence.

Even though we now know what has to be done to solve the problem, there are still many ways to do it. We present some solutions, where we use bracketing to indicate the evaluation order of the expression:

low to high	$(\ldots ((a_1 + a_2) + a_3) + a_4) + .. \ a_n)$	**(1)**
high to low	$(a_1 + (\ldots (a_{n-3} + (a_{n-2} + (a_{n-1} + a_n))) \ldots)$	**(2)**
recursive doubling	$(\ldots ((a_1 + a_2) + (a_3 + a_4)) + ((.. + a_n)))$	**(3)**
random	$(.. ((a_5 + a_1) + ((a_3 + a_2) + a_7) + .. \ a_4))$	**(4)**

In the expressions **(1)**, **(2)** and **(3)** we recognize some pattern. This pattern is the result of systematically guiding the application of the rule of computation. In **(1)**, the result is obtained by adding the element with the next smallest index to the sum of all previously added numbers. Evaluation of **(2)** starts with computing the sum of the elements with the highest indices and consecutively adds the element with the next highest index. The ordering of **(3)** represents a scheme of computation that is known as recursive doubling. This is a recursive scheme that computes the sums of intervals that repeatedly double in size. Random application of the addition rule is represented

by **(4)**. Randomly application of the addition rule yields the same result as all of the systematically guided computations **(1)**, **(2)** and **(3)**. This fact indicates that the strictly ordered computations determine aspects of the solution method that are not essential for the correctness of the computation.

If only a single rule can be applied at a time, then the time complexities of the above examples are the same, viz. $O(n)$. It is more interesting to examine the parallel evaluation of computational rules. The ordering of **(1)** and **(2)** is linear and, as a consequence, cannot benefit from parallel evaluation. The third ordering **(3)** does allow multiple additions to be computed at the same time. The number of additions that can be computed in parallel decreases exponentially, but still allows for an overall time complexity of $O(lg\ n)$. We cannot make any statement about the complexity of the random ordering **(4)** other than that any execution will take time somewhere between the best case, $O(lg\ n)$, and the worst case, $O(n)$. Example **(4)** shows that a well chosen control strategy may exploit the parallelism that is inherently present in the solution method. More importantly, it illustrates that the complexity aspect is determined by each of the systematic methods **(1)**, **(2)** and **(3)**, while this is still unresolved in the random **(4)** computation.

For now, we conclude that it is possible to describe a correct solution, without also determining its complexity. In section 4, we argue that the complexity of a solution method is determined by the ordering of computations and the selection of data.

2.2 Imposing Control over Computation in Sorting

We established that it is possible to specify a correct solution without settling complexity-related issues. If we have a complexity-free representation of a computation, then this may be understood as a non-deterministic specification. Different choices for the removal of the non-determinism lead to different algorithms. Consequently, such a non-deterministic specification represents a class of more determined algorithms.

An identical observation was made in the area of logic programming. The following quotation is from an article by K.L. Clark on logic programming [Cl82]

> "A logic program with control unspecified is a non-deterministic algorithm. It is the family of all the deterministic algorithms that can be obtained by adding specific computation and choice rule control. By partially specifying this control we obtain a new non-deterministic algorithm that is a subfamily of this family of algorithms."

We show that the ordering of computations can be specified explicitly and independent of the computations themselves. Correctness and complexity aspects of a solution method are, when expressed in an existing programming language, represented by a single text. We make the conscious decision of separating the representations of these aspects. When something is changed that concerns the complexity of the solution method, the representation of the correctness related issues remains unchanged.

Different ways of introducing control — or removal of non-determinism — lead to more detailed descriptions of algorithms that perform the same task. This will be illustrated by presenting different control components for sorting algorithms that are all based on the same computational component.

We design a computation component for sorting a sequence, and reason about the correctness of this solution. We continue by showing that it is possible to superimpose different controlling strategies on the same computation component. Additional knowledge about the problem domain can be exploited by the control component to gain a lower time complexity.

Input to the sorting problem is a sequence $A = \langle a_0 , a_1 , ..., a_n \rangle$. A solution is a sequence $B = \langle b_0 , b_1 , ..., b_n \rangle$ such that B is a permutation of A, and the elements of B are in increasing order: $(\forall\ i, j : 0 \leq i < j \leq n :: b_i \leq b_j)$.

In a sorted sequence smaller values precede larger values. If we take a pair of values from a sequence, of which the larger value precedes the smaller one, and exchange their positions, we get a sequence that is better sorted. This single rule is called *swap* (also known as *compare-exchange*). This knowledge provides the elementary rule of computation for solving sorting problems:

> "Select two elements that are in reverse order relative to each other, and interchange the positions of these elements."

This rule decreases the amount of numbers that are in reverse order relative to each other, and consequently increases the amount of numbers that respect their proper ordering. Repeated application of this rule must ultimately result in a sequence in which all numbers are in the correct order.

This rule can be defined formally by a pre- and a postcondition.

$$\{ a_i > a_j \wedge i<j \wedge \text{Perm}(A, A_0) \} \quad swap(i, j) \quad \{ a_i < a_j \wedge i<j \wedge \text{Perm}(A, A_0)\}$$

Execution of *swap*(*i*, *j*) results in values a_i and a_j swapping positions in the sequence if they are in reverse order relative to each other, otherwise nothing changes. In the case of sorting, the computation component consists of this single rule only. We explain in detail that this single rule suffices for solving the sorting problem.

The rule *swap* changes the position of the values in the sequence, but does not change any values themselves. Thus if previous to the execution of *swap* the sequence is a permutation of *A*, then execution of *swap* leaves this invariant.

A combination of two elements a_i and a_j such that $a_i > a_j$ and $i < j$ is called an *inversion*. The maximum number of inversions that can be present in a sequence of *n* elements is $\frac{1}{2}n(n-1)$. The sorted sequence is characterized by the fact that it contains no inversions at all. Execution of rule *swap*, reduces the number of inversions in the sequence with at least one.

The random strategy for applying *swap* consists of choosing values *i* and *j* randomly, and establishing the postcondition if the precondition for *swap*(*i*, *j*) holds. If we assume that the random selection of data is fair, then the repeated random application of *swap* at a sequence that initially contains a finite number of inversions, must at some point yield a sequence that has no inversions left. The only sequence without any inversions is the sorted sequence. Consequently, randomly applying the rule *swap* will eventually sort any sequence of finite length.

The random strategy that is illustrated for addition and sorting can be generalized to computation components that consist of a number of rules of arbitrary (finite) arity. Using this chaotic evaluation mechanism, every computation component can be executed. It is very unlikely though, that the random execution of any program will give an optimal computational complexity. Even the average performance may not meet the required time complexity.

For the "random" method for sorting we can derive an upperbound. From a sequence of length *n* we can choose $\frac{1}{2}n(n-1)$ pairs. Swapping some of these pairs will give a better sorted sequence, while swapping others might introduce inversions. If we select a pair that is already relatively ordered, then we do not swap them. Assuming that we start with the sequence that has the most inversions — a reversely sorted sequence, we have to select $O(n^2 \lg n)$ pairs in the worst case before the sequence is sorted.

For most programs a particular course of execution can be thought of that leads to a better performance than can be expected from the random execution mechanism. In

such cases it pays off to specify a control component that explicitly plans the application of computation rules.

Most sorting algorithms use the *swap* operation as elementary computational step, but differ in the strategy that is used to apply it. Well-known sorting algorithms such as bubblesort, quicksort, mergesort, insertion sort, selection sort, odd-even sort, and Batcher's baffler can all be described as particular orderings of the *swap* computation. The different control components of these algorithms are responsible for the differences in efficiency.

A control component is an imperative statement that defines an ordering on computations that should be respected by any execution. At each step, a control component specifies which computation or group of computations is liable to be executed next, and which data is to be used. For describing the ordering on computations we use three constructs: *sequential* execution, denoted by the familiar semicolon ";", *parallel* execution, denoted by a double bar "‖", and *conditional* execution, denoted by "if .. then .. else". The keywords "for" and "forall" are used to denote repeated sequential or parallel composition.

$$\langle\ \textbf{for}\ i \leftarrow 1\ \text{to}\ n :: S_i\ \rangle \equiv (..(S_1 ; S_2) ; S_3); \ldots ; S_{n-2}) ; S_n)$$

$$\langle\ \textbf{forall}\ i : 1 \le i \le n :: S_i\ \rangle \equiv (..(S_1 \| S_2) \| S_3) \| \ldots \| S_{n-1}) \| S_n)$$

Using this notation we can write down the control strategy for Bubblesort and its parallel relative Odd-Even sort.

Bubblesort

```
⟨ for i ← 1 to n ::
    ⟨ for j ← 1 to n-i :: swap(j, j+1) ⟩  ⟩
```

The sequential Bubblesort performs $O(n^2)$ swaps. Because it computes only one swap at a time, it sorts a sequence of length n in $O(n^2)$ time. By pipelining the swaps that Bubblesort performs, we get a sorting method that still executes $O(n^2)$ swaps, but by executing $O(n)$ of them at a time, we achieve a time complexity of $O(n)$. For an account of the relation between Bubblesort and Odd-Even Sort see for example [Kn73].

A characteristic of both of Bubblesort and Odd-Even sort is that the computations that

Odd-Even Sort

```
⟨ for i ← 1 to n ::
  ⟨ forall j : 1 ≤ j < n ∧ Odd(i)=Odd(j) :: swap(j, j+1) ⟩  ⟩
```

have to be done as well as their relative ordering are known in advance. This can only be the case when these properties are independent of the input-values. A more complex control component is needed to describe Quicksort, where the number of computations and their relative ordering do depend on the input values. A control strategy for Quicksort is depicted in the box below.

Quicksort

```
split(lb, l, r, rb) =

⟨ if l<r
   then ⟨ if a l+1 < a lb
           then  ⟨  swap(l,l+1); split(lb, l+1, r, rb) ⟩
           else  ⟨  swap(l+1,r); split(lb, l, r-1, rb) ⟩
        ⟩
   else ⟨    if lb<l-1 then split(lb, lb, l-1, l-1)
           ‖ if r+1<rb then split(r+1, r+1, rb, rb)   ⟩
⟩

begin
  split(1, 1, n, n)
end.
```

In this section the problem of sorting was used as an example to show how complexity information can be separately specified from correctness information. This gives two separate texts, each of which is dedicated to one aspect of the solution method. The conjunction of these isolated representations is easier to understand than a representation that combines the correctness and complexity in one text. The importance of a separate representation of complexity related information was stressed by Hayes.

> Hayes: "*In conventional programming languages there is no separation of the logical and control aspects of the language. [...] Assigning a clear semantics [to programs expressed in such a language] is difficult, for whichever aspect one concentrates upon, the other is lost; and yet both are important.*"

pp. 110, [Ha73]
Hayes: "*Control information, in the form of detailed descriptions of how to go about deductions, is first-class information and should be treated as such: it deserves its own, carefully designed, language*."
pp. 114, [Ha73]

It is sensible to represent the control component of an algorithm using an imperative rather than a declarative formalism, because it is responsible for the efficiency of the solution. Usually a lot of work has to be done, to extract information from declarative statements. This extraction effort produces additional computing load that negatively influences the performance. If the interest in a specification is solely mathematical, then there is no theoretical obstacle that prevents us from specifying the control component declaratively.

2.3 Conclusion: Algorithm = Computation + Control

The examples in the preceding sections show that it is possible to determine which computations have to be done to solve a problem, without also knowing in what order to perform them. We claim that the computation and the control of an algorithm can be specified separately. Interpreting algorithms as the composition of a computation and a control component is useful because it leads to a clear separation of the correctness (mathematical semantics) and complexity (operational semantics) in the design as well as in the analysis of algorithms. In section 3 we will expound the advantages of this view on algorithms.

The first step in solving a problem is building an abstract (mathematical) model. The parameters in this model generate a problem space. The information that is sufficient to solve a problem, regardless of efficiency, is called the *computation component* of an algorithm. The computation component specifies the elementary computational knowledge that leads to a solution of the problem. This knowledge is formulated in terms of transformations of the problem space.

The *control component* specifies in what order actions (state transformations) must be performed. A computation component alone contains enough information to solve the problem. The control component is added in order to obtain *efficient* solutions by imposing a smart order on the applications of the rules of computation. In doing so, the control component determines the *O*-order of the algorithm.

After having stated in detail what we mean by computation and control component, we define correctness, complexity and efficiency. A program is *correct* if the result it

computes, meets its specification. *Complexity* is a property of a problem. A problem requires a minimum number of operations before a solution is obtained. *Efficiency* is an aspect of a solution method. It denotes the degree in which the available resources are utilised (for instance with respect to the optimal solution). Usually the relevant resources are time and (memory)space. We conform to the accepted nomenclature, and use the terms *time-complexity* and *space-complexity*, when we mean to refer to the efficiency of an algorithm with respect to time and space.

3 Advantages of Separating Computation and Control

Now that we have made it plausible that computation and control are separable concerns in algorithm design, the question arises whether it is a good idea to actually make this separation. In this section we answer this question positively by listing a number of issues that benefit from this approach to algorithm design.

Software Engineering

The main benefit of separating computation from control is the *improved ease of design* of algorithms. The design is split into phases that deal with clearly defined aspects of an algorithm. Firstly, focus is on the declarative point of view. A computation component is designed that forms the basis of any correct solution. Secondly, a strategy is devised that steers the computations to obtain an efficient solution. Especially the design of a computation component becomes easier, because one does not have to worry about the operational aspects of the solution. But even without control information it is possible to prove or disprove the correctness of the computation component. The fact that in the correctness stage of the development, it does not have to be taken into account whether the programs are intended for parallel or sequential execution, has important consequences for the correctness proofs of the computation component. Where proofs for (imperative) concurrent programs are often blurred by operational details, the correctness of computation components can be demonstrated by *concise proofs*. Algorithms can therefore be expected to be correct more often and easier to understand.

Furthermore, when control is left unspecified, the computation component can still be used as a program. It can be executed using the random application mechanism. This is generally not very efficient, but the executability of the computation components allows it to be used for *rapid prototyping*. When a computation component is judged adequate, the design can be continued by adding a control strategy for reasons of efficiency.

Parallelism

The order in which the operations of an algorithm are performed is determined by the control strategy. In particular, any decision regarding sequential or parallel execution of the algorithm can be postponed until a control strategy must be selected. The absence of control implies the absence of the Von Neumann bottleneck. The computation component in principle *contains all parallelism that is inherent in the logic of the program*. Hence, if the logic of the program allows for a massively parallel execution, then so does the computation component. The control strategy, in its turn, decides whether or not to exploit the available parallelism. In the case of the sorting example, the computation component does not specify whether a single pair or multiple pairs of elements should be compared and exchanged at the same time. The control strategy resolves this in favour of sequential execution by dictating Bubblesort, or in favour of parallel execution by dictating Odd-Even sort. When a programmer cannot discover the potential parallelism of a solution method, this does not have to refrain him from designing a computation component (because this is independent of sequential or parallel execution). Examination of a computation component often naturally leads to insights about potential parallelism.

Portability versus Efficiency

In many programming languages there is a trade-off between portability and efficiency. When computation and control are designed separately, no compromises have to be made in favour of one of these. Since the computation component is independent of the underlying machine, it can remain completely unchanged when ported to another architecture. Yet it is still possible to achieve efficiency of the solution method, because the control component can be adapted to any target architecture. This possibility is expected to be especially useful for parallel and distributed systems because architectural differences among the plethora of parallel machines are much more significant than among sequential machines. The proposed design method combines a high degree of *portability* without restrictions on *efficiency* by localising implementation dependent concerns.

4 Correctness and Complexity in Computing Science

We assert that correctness and complexity are identified as the two most important aspects of an algorithm by people from different areas of computing science. This claim is supported by a number of citations. The quotations included in this paper are only a few among the many passages that can be found in which these two concerns

are juxtaposed. After these excerpts from the literature, we investigate why no one has been able to separate computation and control as nicely as is illustrated by the examples in section 2.

4.1 Excerpts from Literature

Followers of different computing paradigms find that two, more or less separable concerns, are involved in the design and analysis of algorithms. The terminology used for these concerns differs slightly, but the fundamental concepts coincide. We use the terms "correctness" and "complexity" (as defined in section 2.2).

First, some quotations from textbooks on *imperative* algorithm design.

> Gries: "*The programmer has two main concerns: correctness and efficiency [...] When faced with any large task, it is usually best to put aside some of its aspects for a moment and to concentrate on the others [...] This important principle is called* ***Separation of Concerns****.*"
> pp. 237 [Gr81]

> Baase: "*There are two aspects to an algorithm: the* ***solution method*** *and the* ***sequence of instructions*** *for carrying it out.*"
> [Ba88]

> Chandy and Misra: "***1.2.6 Separation of Concerns: Correctness and Complexity*** *[...] The correctness of a program is independent of the target architecture and the manner in which the program is executed; by contrast, the efficiency of a program execution depends on the architecture and manner of execution.*"
> pp. 7-8 [CM88]

> Kaldewaij: "*There are two factors by which algorithms may be judged: their* ***correctness*** *(do they solve the right problem?) and their* ***performance*** *(how fast do they run and how much space do they use?).*"
> pp. ix [Ka90]

Statements of similar intent can be found in literature on *logic programming*.

> Kowalski: "*[...] when* ***logic*** *is separated from* ***control****, it is possible to distinguish what the algorithm does, as determined by the logic*

component, from the manner in which it is done, as determined by the control component."
[Ko79]

Bratko: "*It [..] makes sense to distinguish between two levels of meaning of Prolog programs; namely*,

- *the declarative meaning and*,
- *the procedural meaning*.

*The declarative meaning is concerned only with the **relations** defined by the program. The declarative meaning thus determines **what** will be the output of the program. On the other hand, the procedural meaning also determines **how** this output is obtained; that is, how are the relations actually evaluated by the Prolog system.*"
pp. 24 [Br86]

Jaquet: "*Most of the interest taken in logic programming comes from the possibility to ascribe two different complementary semantics to logic programs. The **declarative** semantics refers to the semantics of first order logic [...] At the opposite, the **procedural** semantics is machine-oriented and refers to a model of execution.*"
[Ja91]

The article by Kowalksi was the first widely disseminated article that acknowledges the benefits of separating what he called "*logic*" and "*control*". In texts on *functional programming* one may find statements of the following kind.

Hudak: "*Parafunctional programming isolates **functional behaviour** from **other behaviour** so you can express, reason about, and ultimately debug the behaviour independently.*"
pp. 59 [Hu88]

Meertens: "*Programs can themselves be viewed as specifications, in two ways. One is the operational viewpoint: programs as specifying a process for some (abstract) machine. The notion of efficiency is intimately tied to this viewpoint: it is meaningless to discuss the efficiency of a program outside the context of a mapping to a process on a machine. [...] We can also abstract from the internal process aspects by identifying observably equivalent processes, and consider the meaning of a program as a point in the resulting abstract space. We then obtain the 'declarative' viewpoint of programs as specifications.*"

[Me89]

Also from the area of process calculi comes the analogous statement

> Milner: "*[...] thus one of the outstanding challenges in concurrency is to find the right marriage between logical and behavioural approaches. In fact it is one of the outstanding challenges in computer science generally, because the whole subject is concerned with the relationship between assertion and action - or, if you like, between the specification of systems and their performance.*"
> pp.4 [Mi89]

These quotations demonstrate that a number of people have noticed the separable aspects of computation and control. This invites investigation into what has prevented the emergence of a programming model that capitalises on the benefits of addressing these issues separately?

4.2 Requirements for Separating Computation and Control

In this section we look at summation programs expressed in different programming languages. We examine these programs to find out where problems occur when we want to separate computation from control.

If we would code a summation algorithm in some imperative language, we would use an array to represent the input and use a for-loop to repeatedly add a new number to an intermediate result.

```
sum ← 0;
for i ← 1 to n do
    sum ← sum + a[i]
```

In a functional language we represent the input by a list. A recursive traversal of the list, adding numbers as we go, would be the most evident solution.

```
sum []     = 0
sum x:xs   = x + sum xs
```

For completeness sake, we give a representation of the program in a logical formalism, although it is not essentially different from the functional program.

```
sum ([], 0).
sum ([H|T],X) :- sum(T,Y), X is Y+H.
```

We illustrate that the above programs impose an irrelevant ordering on the computations.

Both programs prescribe an ordering like **(1)** in section 2.1 — a sequential evaluation ordering. This is an overdetermination of the ordering of the computations. The associativity property of the addition function allows any pairing of elements and accordingly any evaluation ordering would suffice. The computation as described by the above programs also limits the order of the processing of data. Even though it is not essential to the (correctness of the) algorithm, the programs dictate that data be added starting from the first element (with index 1), and consecutively add the elements with the next highest index in the sequence. This restrains the order in which data is processed, because addition is commutative, and thus any ordering of the data yields the correct result.

The observation that programs are often needlessly restrictive holds in general, and has been noticed before. In Dijkstra's authoritative work on program design, *A Discipline of Programming*, he presents an algorithm that solves Hamming's problem. The algorithms contains three pairs of variables that need to be updated — each pair independent of the others. This is done by three consecutive do-loops, about which Dijkstra notes

> Dijkstra: "*The fact that the three repetitive constructs, separated by semicolons, now appear in an arbitrary order does not worry me: it is the usual form of over-specification that we always encounter in sequential programs prescribing things in succession that could take place concurrently.*"
> pp. 133 [Dij76]

The ways in which programs are restrictive is with respect to the ordering of computations (time), and with respect to the ordering of data (datastructure). A prerequisite for being able to separate computation from control, is that the specification of the computations has several degrees of freedom; namely: freedom in time, and freedom in space (datastructure). We will use the addition problem again to illustrate this categorisation.

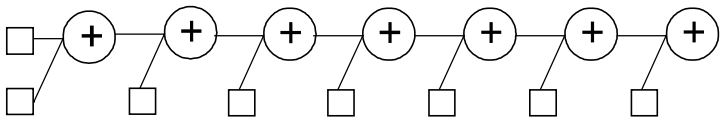

Figure 1 Sequential Addition

Figures **1** and **2** depict an ordering of computations (circles with a "+" inside) and data (squares) in time. The ordering should be read from left to right; i.e. elements left from another element, should be computed before elements to their right

The linear ordering of figure **1** requires *n* steps. The recursive doubling order of figure **2** can be computed in *lg n* steps. This illustrates that there is a degree of freedom in ordering the computations in time.

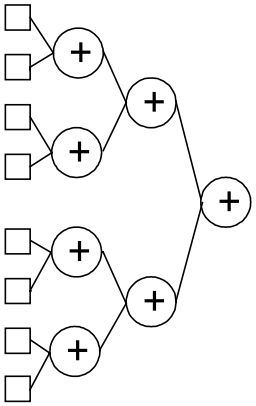

Figure 2 Parallel Recursive Doubling Addition

Squares in the figures represent elements a_i. The figures show orderings in time of the computations that produce the desired result for any substitution of data. We observe that there is a degree of freedom in choosing in what order we place the values a_i on the squares. In the above examples, the ordering of data is independent of the ordering of computations. This is not generally the case with algorithms. Some algorithms require that the distribution of data complies with the ordering of computations. In practice we often see the reverse case. Once the choice to use a certain datastructure has been made, then the ordering of the computations is partially fixed, because a datastucture allows its contents to be accessed in a particular fashion. In such a case the time-complexity is (partially) determined by the datastructure.

Assuming that the ordering in time is fixed according to figure **2**, and a hypercube topology is given as the target architecture for execution of the algorithm, then the

following figure illustrates how the control component could combine the ordering in time and an ordering in space:

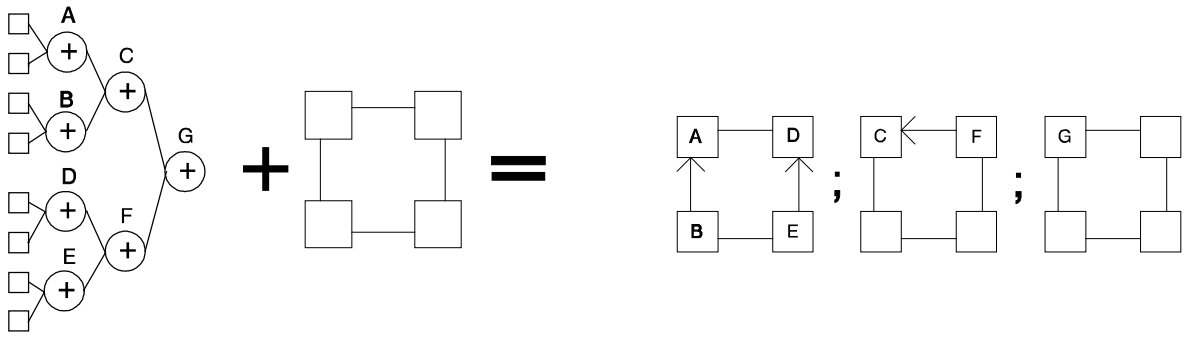

Figure 3 Mapping of an ordering in time into space

These degrees of freedom with respect to time and space that should ideally be present in the computation component, can later be eliminated by the control component. These freedoms are often lacking in programming models. We point out two of the main reasons for this:

Inhibiting datastructures

The benefits of separating the computation component from the control component have been acknowledged in the area of logic programming. The ideas could not be exploited because implementations of logic languages chose to use recursive data structure for the representation of data. The elements of a recursive structure can only be accessed one at a time. Before one finds out what the value at a certain position in a structure is, all preceding elements have to be removed one by one. Functional programming also suffers from the fact that it offers only recursive datatypes like lists and trees. This sequential mode of accessing data imposes irrelevant orderings in time.

For the highest degree of freedom in data it would be desirable to have a data structure that allows all of its constituents to be accessed directly.

The array from the imperative world might seem to be a solution, but also this representation has aspects that are not ideally suited for our purposes. One aspect

depends on the implementation of the programming language. When an array *a* is declared in the language *C*, the value of *a*[*1*] is stored next to *a*[*2*] *in memory*. In our opinion an array should be used as a logical structuring device, and not as a physical one. For efficiency reasons, it is desirable to be able to indicate where in space (could be a place in memory or a particular processor in a network) a particular variable must be stored. In many programming languages this is not possible.
Furthermore, an array requires that its elements be accessed by an index. This requires that the collection of data that is modelled is suitable for this kind of referencing. One can imagine that this clashes with the modelling of, for instance, a multiset with multiple identical elements.

Inability to explicitly determine control

Functional and logic programming stem from purely mathematical models. This makes these programming models conceptually elegant. For their execution, they rely completely on automatic control mechanisms. The same elegance that is beneficial from a mathematical point of view, becomes a weakness when programs are put into practice, because these models have no notion of execution efficiency.

Functional and logic programming have justly received much attention because they have the potential of alleviating the programmer from the operational details of programs. Unfortunately this claim is not fully fulfilled by either of the paradigms. Once a correct functional program has been written, the development of the program continues by successive rewriting for the sole reason of achieving efficiency. In logic programming on the other hand, the programmer has access to operators that intervene with the automated execution mechanism.

In both cases concessions have been made that introduce the ability to increase efficiency, at the expense of blurring the fundamental ideas of the respective paradigms.

Functional programming has an advantage over logic programming by having higher-order (or meta) functions. The essence of higher-order functions is that they steer the application of first-order functions. In this, we recognize a partial acknowledgement of the importance of an explicit control component.

There is active research, e.g. [Ma92], in extending logical and functional programming with arrays, or additional constructs such that the programming language offers more opportunities for exploiting parallelism.

5 Conclusion

Even though the opinion that the two most important aspects of an algorithm are correctness and complexity is prevalent in computer science, there exists no design methodology that exploits the benefits of separately addressing these issues. We have presented some examples that illustrated that correctness and complexity can be separated in algorithm design. The correctness component defines the basic operations to be performed in order to solve the problem at hand. The control component imposes an order on the execution of the operations with the objective to obtain an efficient solution.

The separation of correctness and complexity improves the ease of programming. The resulting computation component contains all parallelism that is inherent in the solution. The abstraction of (architectural) implementation details combines portability with efficiency.

By examining small programs expressed in existing programming languages, we identified features of programming formalisms that inhibit the separation of computation from control. These observations lead to the recommendations about the desirable properties of programming languages.

- Computation information and control information should be represented by distinct texts. This will make the correctness and complexity aspects of an algorithm easier to design, and easier to understand.

- A programming model should be able to separately address the ordering in time and the ordering in space. Ordering in time and in space both belong in the control component because they are concerned with the efficiency of the ultimate program. Consequently, the representation of the computation component must leave these issues uncovered.

Research in search of an ideal programming model is continued, and in the near future we expect to finish the design of a model that fully supports the separate design of the computation and control components of algorithms.

References

[Ba88] Baase, S.,
Computer Algorithms, Introduction to Design and Analysis,
Addison-Wesley, 1988

[Br86] Bratko, I.,
Prolog Programming for Artificial Intelligence,
Addison-Wesley, 1986

[CM88] Chandy, K. M., and Misra, J.,
Parallel Program Design: A Foundation,
Addison-Wesley, 1988

[Cl82] Clark, K. L., McKeeman, W. M., and Sickel, S.,
Logic Program Specification of Numerical Integration in
Logic Programming, eds. Clark, K. L. and Tärlund, S.-A., Academic Press, 1982

[Dij76] Dijkstra, E. W.,
A Discipline of Programming,
Prentice-Hall, 1976

[Gr81] Gries, D.,
The Science of Computer Programming,
Springer-Verlag, 1981

[Ha73] Hayes, P. J.,
Computation and Deduction,
Proceedings of the 2nd Symposium on Mathematical Foundations of Computer Science, Czechoslovak Academia of Sciences, pp. 105-118, 1973

[Hu88] Hudak, P.,
Exploring Parafunctional Programming:
Separating the What from the How,
IEEE Software, January 1988

[Ja91] Jaquet, J.-M.,
Conclog: A Methodological approach to Concurrent Logic Programming,
Lecture Notes Computer Science 556, Springer-Verlag, 1991

[Ka90] Kaldewaij, A.,
Programming: The Derivation of Algorithms,
Prentice Hall, 1990

[Kn73] Knuth, D. E.,
The Art of Computer Programming, *Vol*. 3, *Searching and Sorting*,
Addison Wesley, 1973

[Ko79] Kowalski, R.,
Algorithm = Logic + Control,
Communications of the ACM Vol. 22, No. 7, pp. 424-436, 1979

[Ma92] Maaßen, A.,
Parallel programming with data structures and higher order functions,
Science of Computer Programming, Vol. 18, 1992, pp. 1-38.

[Me89] Meertens, L. G. T. L.,
Constructing a Calculus of Programs,
in *Mathematics of Program Construction*, J. L. A. Snepscheut (ed),
Lecture Notes in Computer Science 375, Springer-Verlag, 1989

[Mi89] Milner, R.,
Communication and Concurrency,
Prentice-Hall, 1989

Further Reading

[BM93] Banâtre, J.-P., and Le Métayer, D.,
Programming by Multiset Transformation,
Communications of the ACM, Vol. 36, No. 1, pp. 98-111, 1993

[Bc78] Backus, J.,
Can Programming be Liberated from the Von Neumann-style,
Communications of the ACM, Vol. 21, No. 8, pp. 613-641, 1978

[Re81] Rem, M.,
Associons: A Program Notation with Tuples Instead of Variables,
ACM Transactions on Programming Languages and Systems, Vol. 3, No. 3,
pp. 251-262, 1981