Keywords Programming Languages, Theory of Computing
Abstract This work seeks to characterize the gap and to supply a concept that it at present missing from the science of computing. The gap that is waiting to be filled is, in a simple way, apparent when we speak informally of doing the "same" thing by means of different programs, perhaps in different languages, or perhaps modes of implementing a language, or even two "paradigmatically" different systems. It makes informal and important sense to ask, of any system of getting a computer to do things, for a piece of text or other appropriate stimulus whose performance does, in some sense, achieve a given behaviour – whether by Pascal, Haskell or Prolog, or a structured construction of a Turing machine, or by object-oriented interaction among objects. We propose to decouple the customary linkage from program to the function or relation that it realizes. What is to be interpolated is a function from data to entire calculation.

Noname manuscript No.
2(will be inserted by the editor)

Calculations
A Hole In The Heart Of The Study Of Computing

Peter J. Landin

the date of receipt and acceptance should be inserted later

Contents

1 Interlude: A Hole In The Heart Of The Study Of Computing

There is a growing divide between the theorization of computing and the practice of those whose job it is to get computers to do calculations. This can be acknowledged without judging it any more reprehensible than, say, the similar development that overtook economics a few decades ago. In the case of economics, a historical analysis of that development has still gone scarcely further than sour grapes, accusations of mathophobia and defensive claims about filling syllabuses and theses with hard science rather than discursive rhetoric. Perhaps history is written by the victors.

In the case of computing, the growing gap is unsurprising for research that follows a similar tendency in the branches of mathematics that computer scientists draw on – algebra and logic.

Programs and program-fragments are viewed as functions over some range-of-variability concerning the conditions in which they are performed. These conditions include data, textual and dynamic context, implementation, configuration and the management of time and space. These will henceforth be collectively referred to as data. Whether regarded by means of logic or algebra, and whether the semantic interest is operational or compositionally denotational, the functions that are studied are largely about outcomes, results, answers and non-answers, and about equivalences modulo these things, especially when they are elegantly and concisely characterizable.

By contrast with theorization, the practitioners of computing wade through a deepening swamp of unstable but undredgeable artifacts for which elegant and concise

Peter J. Landin
Queen Mary College, University of London, UK

characterizations are rare. Delicately balanced edifices are nudged together, propped up, and ineffectively shielded from the passing winds and hurricanes of new releases and updatings. As for anything being equivalent to anything else, there is little to be confident about.

For lack of firm equivalences, introductory learning practice becomes syntactic maze-running, performed with few semantic sign-posts or coherent nostrums about why what works, or which perturbations have what consequence. The question "Yes, but what does it *do*?" loses meaning.

Productivity in artifacts and their updatings far outstrips productivity in making sense of them. New proposals for their conceptual models become, even before they are launched, obscured by the syntax of communicating with them.

We have been living through the whithering away of the programming language. There is a diminishing importance in the concept of a language equipped with all the software, meatware, and gasware that was intended to grow around it but never really did. No passing language except a dead one ever stood still long enough to become compatibly and consistently implemented and tooled (software), consensually and reliably taught, talked about, and thought about (meatware), or precisely and accurately described and theorized (gasware). This is not new. But it is more so. In a vast and vibrant economic and social sector, each accessory, whether soft, meat or gas, breeds its own accessories. Nor is it deplorable. But it does add urgency. Individualized interactive experimentation replaces reflection and a consensual working culture. Declining stability undermines principled teaching and clear communication.

There is one exception to the divergence of theory from practice, and it is unfortunate. Both are increasingly complicit in seeing the current state-of-affairs as indivisible. Timidity and expediency, whether their own or of others, trap both in unary worlds.

As artifact instability throws users back into the arms of experimental interaction, they are forced into procedural rather than conceptual thought – A Piagetian regression to an earlier stage of childhood development – backwards from formal operational through sensory-motor. Hobbled by the preset agendas of menus that are the product of unary minds, it is risky to think in non-unary terms. There may indeed be object-oriented objects that pursue multary behaviours. But collectively their interface is to be gingerly explored without the help of that assumption. For the producers of an artifact, multariness is too long term and costly a burden. For the theorizers, unariness is a comfortable refuge.

Atavistic old hands are nostalgically rung in memory of those "real" machines that "really" "did" obey instructions. Real machines were less often updated, and so the experience of them provided an intuitive earthing for mental models that, even if never actually consulted, might, by the very assurance of their existence, provide an exemplary ground-floor for the upper storeys. For lack of this assurance, virtual machines and abstract machines are made and used by people who don't really believe in them, and who comfortably confess so when their designs show cracks and bumps. What do you expect? – it's theoretical.

The diagnosis offered here is that current computer science proceeds without looking back over its shoulder. It has not noticed, and indeed does not need to notice, that the firm ground is no longer credited or creditable. The obscure depths below which the physical machine has sunk has left a gap, a hole in the heart of the study of computing, for which it is the present purpose to propose a filling: calculations.

This work seeks to characterize the gap and to supply a concept that it at present missing from the science of computing. The gap that is waiting to be filled is, in a simple way, apparent when we speak informally of doing the "same" thing by means of different programs, perhaps in different languages, or perhaps modes of implementing a language, or even two "paradigmatically" different systems.

It makes informal and important sense to ask, of any system of getting a computer to do things, for a piece of text or other appropriate stimulus whose performance does, in some sense, achieve a given behaviour – whether by Pascal, Haskell or Prolog, or a structured construction of a Turing machine, or by object-oriented interaction among objects, or whatever.

We propose to decouple the customary linkage from program to the function or relation that it realizes. What is to be interpolated is a function from data to entire calculation, or perhaps a description of a calculation.

It is a notion often appealed to, from all levels from novice work to experienced design. Its uncertainties are perhaps disambiguated by careful dialog, but only case-by-case, not in a systematized or uniform way. It would be pleasant to approach such situations in a more sharply defined conceptual context. It would be more pleasant to forestall them with a set of concepts that express the intended aspects precisely, but without irrelevant precision or elaborate qualification.

This conceptual gap probably calls for a notion that will be as fundamental to the study of computing as the concept of mass became to the study of physics. The observations that each of them contributes to de-relativization lends support to this analogy.

Explanations of different modes of program-performance are sometimes unnecessary because they are expressed in terms of concepts whose design was specifically chosen so that such contrasts were concealed. For example the distinction between strict-lazy "if" has sometimes been poorly explained using if-notation. More generally, function notation is not a helpful tool to explain how strict and lazy functions differ. A similar situation occurs when the copying of a data-item has to be contrasted with copying its name or address – say, for stack-items, garbage-collectible items, or directory items.

For a system in which both compiled and interpreted modes are available it is a blessing to know how far the same notation "works" for both. But if the interpretive modes are explained by presenting interpreting programs, especially in the same language, then the explanation must be taken with care. To explain by particular instances is to risk not being comprehensive. But if comprehensiveness requires a program then the question may be begged.

Informatics without calculations is like physics without mass. So how did physics last so long without mass? Why was it not missed for centuries? Because physicists, or their forerunners, thought operationally about what matter did, rather than conceptually about what it was. Similarly for informaticists (and there is a trivial paradox here that arises merely from a terminological accident about the word "operational", a concern with operations in their dynamic aspect as processes leading to outcomes, has obstructed any view of behaviour as an object of study in itself). By contrast, in logic the corresponding view of proofs as objects of study in themselves is well established. However it must be admitted that at the present stage, this informatics-physics analogy is not deep. Mass underpins the way weight varies according to physical context. I seek a similar underpinning for the way in which proposed notations of behaviour vary according to the context of language, mode of execution, and implementation.

I have little to offer beyond a tentative definition. Existing theories of process have already gone beyond that, probably benefiting from their two limitations. They are restricted to state-to-state, or unary, systems, together with concurrency-configurations of these. Secondly, and at a more pragmatic level, they lack adjustable focus. They do not pay attention to the ranges-of-variability, from coarse to refined, and from concrete to abstract, that play an important part in views of program-performance.

For example, steps that achieve name-value-lookup, are from some points of view relevant – from others forgettable. Similarly with other situations requiring the selection of one from among several alternatives. Other examples of such discretionarily elidable administrative steps will be given later. The trade-off between unadministered particular cases and administered general procedures is an issue that pervades computing. To shield this from view is usually rated a merit of a programming system. To exhibit it we have usually been driven towards "the machine". But to *what* machine? and please can there be not a lot of them?

Calculations are an attempt at this affirmative. Perhaps there *was* a time when adding up apples was different from adding up oranges. To propose otherwise threatened venerated historic tradition, NIH hubris, trade barriers, Chinese walls, and competitive secrecy. Equivalently there still *is*, despite unstoppable decimalization. Napoleon didn't invent numbers but it might have seemed so to anyone who was forced to realize that lots more things were merely numbers than had previously been apparent. (It has been left to the tax-collecting departments of British local governance to decimalize the year – by the Procrustean amputation of February and March.)

There are two existing concepts whose credentials for filling the hole-in-the-heart need consideration. Data-flow-diagrams have some claim to staticizing the dynamic. And likewise for a closed nested where-term such as might result from partly reducing a functional program as far as eliminating function-variables, and without doing any of the actual evaluation. Dfd's will turn out to be groomable into rigorous calculations, once their two awkward features are recognized. There is no difficulty about streams occurring as the data-items of a calculation. They will need proper start-up operations, and the feedback, or cyclic nature of most dfd's must be faced. Also the "implementation" of a time-changing data-base by a stream of time-stamped updaters is unattractive. As for nested where-terms and beta-redexes, they are close but with some serious irrelevancies. Why be committed to particular names? And to particular syntactic nesting?

It is not necessary to claim that Newton and Napoleon rolled into one have been at work here, to hope that calculations have two things to offer. First, analytically, as a low-level explanatory tool, and synthetically, as a way of moving to a program when the starting point is some of the (ground) calculations you want it to do.

There is one benefit that becomes clearer as this work proceeds. Through the basic concepts of computing, a dividing line can be drawn separating aspects of programming that are properties of ground runs, from properties that only have meaning in terms of the program being performed. But the dividing line is not simply that of a run-time *versus* compile-time. For example, looking at the entire history of a specific execution, straticized as a composite "first-class" piece of information, we may observe a certain intermediately resulting data-item that is used more than once, or a certain sub-set of data-items that constitutes a candidate for single threadedness, or two fragments that are sufficiently congruent to suggest a procedure abstraction.

Each of these properties is innate in the calculation and independent of whether it arises from some program-plus-data (run-time attribute), or program sans data (compile-time), or program-plus-some partial determination of its data ("half-compile"-time attribute). Thus exception-raising is usually regarded as a run-time attribute of a program, and recursion as a compile-time attribute. But as attributes of a calculation neither has meaning, except perhaps as a recognizable pattern.

Pedagogically, it is proposed that teaching about calculations serves the purposes that have been in the past attributed to teaching about hardware and machine-code, and that have in recent years been undermined by the disappearance and diminished pragmatic relevance of the machine. These purposes were and are valuable. They can be resumed, indeed better served, by taking calculations as the pedagogic point of departure. Machine-code is not thereby banished. It takes a place as a low rung on a ladder of particular, approximately single-threaded, calculations whose role is usually to intimate multary calculations.

2 What a Calculation Is

2.1 A Calculation Is a Graph-Like Thing

2.1.1 Graphs and Labels

It is a vertex-labelled, edge-labelled "poly-graph", where "poly-" here indicates a generalization of the usual connections between edges and vertices. The source of an edge is **either** a vertex, **or** (disjointly) an indexed set of vertices (see fig 1). Likewise for the target of an edge (fig 2). A calculation is thus characterized as consisting of calculational steps usually plugged together. Any further features of calculations, that draw attention to multi-step parts of it, for example as nested sub-calculations, or as threads, will be introduced as mappings between calculations. The disjoint options for source and for target impose on data-items a severe dichotomy into non-records (fig 3) and records (fig 4). They permit calculations to acknowledge, for every occurrence of a record, whether it is derived intact or is thus piecewise, and whether it is used and possible shared intact or piecewise. This will be referred to as the "disjointness feature" of this definition of polygraph[1]. In a calculation each edge is one (occurrence of a) calculational step, and a calculation consists of (occurrences of) steps, usually plugged together. In the preceding sentence, the cautious phrasing reflects the sense in

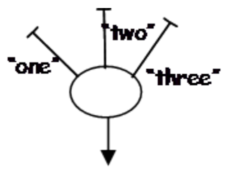

Fig. 1 A 3-In 1-Out Edge

[1] The following slight generalization of this definition of "poly-" has a usefulness that becomes apparent when we consider the compositional structure of calculations, and their (unsurprising) correspondence with sets of definitional equations. This generalization is: The source of an edge is an indexed tree of vertices (i.e. it is **either** a vertex, or (disjointly) an indexed set of indexed trees of vertices). Likewise for the target of an edge. By comparison, the definition given in the main text constitutes a restriction on the tree-depth to less than two. The term "leaf-indexed source" therefore comprises all shapes of source. In particular it fails to distinguish what in the simpler version can be referred to as the unindexed case. It would be a distraction to pursue this generalization through the various elementary remarks about calculations that follow. When it is re-introduced there will be a bit of catching up to do, especially in respect of indexed sets that are empty, and operations that collect and distribute the parts of a composite data-item, and also the pictorial display of a calculation. In the catching up no revision will be required concerning word "unindexed". By a small overload, it can mean a single point tree, and can continue to exclude the disjoint case if the empty indexed set.

which, say, the addition of 1 to 66 might occur more than once in a calculation – see 2.7.

Historically the word "graph" is still wobbly enough to need explicit definition by almost every user of it. In the terminology adopted here it means polygraph. When appropriate the special case of unary graphs will be signalled.

Every calculation is a labelled graph. But not every labelled graph is a calculation. There are two questions that immediately arise, but are largely adjourned until after the analogy that follows here.They are: What are the labels? And secondly, is it envisaged that a piece of data might be, in any sense, calculated in more than one way? Is there a requirement of, at most, single assignment? Can a vertex and its label be pointed to more than once? Two conditions must be met. The labels on edges and vertices are, respectively, operations and items of data that "fit" them, in an unsurprising sense that is spelt out as the "consistency" requirement in 2.2. Secondly, no vertex is "arrived at" (aka pointed to, in an unsurprising sense spelt out in 2.1.4) more than once.

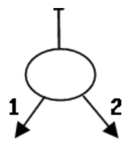

Fig. 2 A 1-In 2-Out Edge

So, anticipating slightly, we define *calculation* to mean: consistently vertex- and edge-labeled graph, in which no vertex is arrived at more than once. With this arrangement, we can, for example, envisage an instance of a number-operation for quotient-and-remaindering, that steps from the two numbers 17 and 4 to the two numbers 4 and 1: the earlier 4 being at a distinct vertex and being a result of another step. Again, we can envisage an instance of a logical inference-rule that steps from nothing to the sentence "communism is dead". Steps from nothing are often called axioms in logic and constants in algebra.

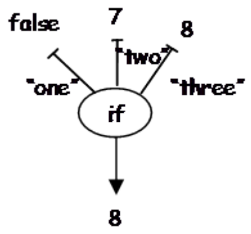

Fig. 3 Non-Records

Some rules about drawings of calculations are given later. Meanwhile it might be possible, without further explanation, to attempt some useful pictures (figs 5 6). It is the label of a calculation, rather than the vertices and edges themselves, that carry its calculational information, namely, items of data and results and the operations that link them. For the above two steps to occur in a calculation, it must contain vertices labeled 17, 4, 1 and "communism is dead". There must ordinarily be two labeled 4 (fig 7). Otherwise would compel the graph to contain a cycle – not prohibited by the nature of calculations, but not suitable for an elementary illustration. The significance of cycles in calculations is discussed in 2.1.7. Also it must contain two edges labeled by an operation of which these steps are instances, in obedience to the requirement of consistency.

2.1.2 A Hazardously Apt Analogy

Something very close to this idea of calculation is an occupied computer memory in which the elastically-sized registers is dedicated either to a singly-assigned data-item or to a once-only-executed instruction that indicates three things: an operation, the registers it consults, and the registers it assigns to. Each register is then either a vertex or an edge.

Corresponding to the consistency condition for a graph's labels, there is the requirement that, for each instruction, the registers it names are occupied consistently with its operation. Thus we are looking at the memory-state as it might be after every instruction has been executed (exactly once). This can be said with no explanation about how they came to be appropriately sequenced, nor about how it all began, nor about what mishaps might have been avoided, nor even about what magic might have hit on solutions for any cycles.

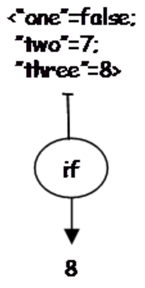

Fig. 4 Records

Various options for an interpretive mechanism will be discussed in a later section 3.4. Part of the interest of the way various interpreters differ will lie in how they suggest different relaxations of the present definition of calculations, and how they do or not respond over a wider range. (It can be noticed in passing that this arrangement precludes instructions whose parameters are indirectly addressed or addressed via address-modifiers. Arrays are not thereby vitiated. An elastic-sided register may be occupied by a single composite data-item. But a singly-executed instruction cannot index it in more than one way).

This collection of singly-executed instructions is one that might arise from the loops and calls of a more conventional program when they, and the addresses they refer to, are totally unfolded in accordance with specific data. In this set-up it is the ultimately remembered features that constitute a calculation in the present sense. Though apt, this analogy is hazardous if it encourages a dynamic rather than a static view of calculations. A calculation is a composite data-item that records calculational behaviour, presented as a staticized, "first-class citizen" of the data world. It is not a dynamic process. It is a static record of (perhaps only some aspects of) a completed dynamic process. For this reason, there cannot be a question about whether an instruction in this set-up is inapplicable or non-determinate, or whether exceptions are raised, or whether the program failed to complete.

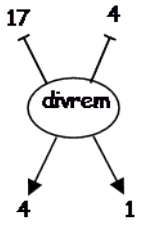

Fig. 5 A Step

There is, in this set-up, an asymmetry that is partly misleading. It is also present in the above definition of polygraphs, and indeed in one of the two customary approaches to ordinary graphs. There is a suggestion that edges "point-to" vertices, and not the other way round. The fortuitousness of this is exposed by recalling that an alternative, and for some purposes more convenient, representation of a graph in computer registers might accompany each vertex with a catalogue of its roles as a target and source, with each role recorded as: one bit for a source-or-target indicator, plus an index if any, plus (a "pointer-to") the relevant edge. There is a little task to undertake here, of designing a corresponding interpretive mechanism.

Lying in this analogy there is another risky suggestion to be resisted. The question whether a vertex or edge might be "pointed-to" by, or from within,

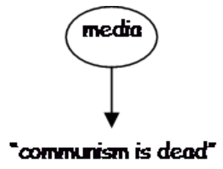

Fig. 6 A 0-in step

a vertex- or edge-label, is a straightforward confusion of abstraction-levels. For example there cannot be an operation sensibly described as: adding the list-sum of the label of the vertex 77. Nor can there be a data-item sensibly described as: the ordered pair consisting of 7 conjoined with the label of vertex 88.

2.1.3 An Edge's In-Arity and Out-Arity

Each edge has a numerical in-arity of 0 or more, namely the number of its source vertices. And likewise for out-arity. A (zero-to-zero)-ary edge is necessarily unconnected to any vertex. It is called a *detached* **edge**. Detached edges seem to be without intuitive significance, but their exclusion seems unjustified. They have no use and cause no problems. In passing, it may be observed here that an *isolated* **vertex** involved neither as source nor target, does have intuitive significance. It is certainly not excluded – see 2.8.

In particular, we do not exclude graphs with just one vertex and/or no edges. Whether or not tolerate no vertices, with or without those detached edges, is a technical issue that can be resolved by further developments. We can expect that it will be settled by technical neatness when defining the composability of calculations. Likewise, at the other extreme, for infinite collections of vertices, edges, and/or indices.

It must be noted that these are attributes of edges, not of operations. Uniformity in this respect, let alone more rigorous type-uniformity, is not here taken for granted. Nor, even, are they attributes of instances of operations. When a record occurs as the argument of an operation-instance, the question whether it appears intact at a single vertex or piece-wise spread over several, is immaterial to the identity of the operation instance.

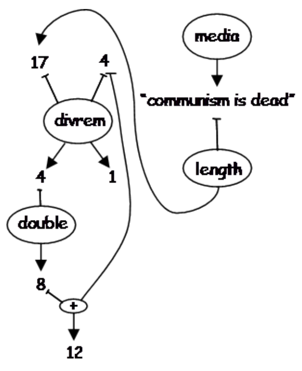

Fig. 7 Combination

2.1.4 Arrow-Tails and Arrow-Heads

The above definitions for the arities of an edge are forgetful about the great variety latent concerning index-sets, and concerning the disjointness feature of a single indexed source, let alone more refined notions of type. A (3-to-2)-ary edge includes 5 "half-edges" called here *arrow-tails* and *arrow-heads.* A path consists of alternate tails and heads and indicates calculational dependence. This notion is intuitively violated if a path is cyclic. Nevertheless, cycles will find their uses below when streams are acknowledged as unitary data-items.

It is an arid and intricate exercise to formulate a definition of polygraphs in terms of four sets rather than two, namely vertices, edges, tails and heads, with the obvious incidence functions for mapping tails and heads to vertices and edges. The rub comes with the indexing. It becomes clear that, if the purpose is to provide tails and heads with some mathematical identity, it should be relative to edges (see 2.6).

2.1.5 Special Cases: Term-Evaluations, Unary and Single-Threaded Calculations

In term-evaluation, and "tree-automata", we see examples of calculations that are special in three ways. Multi-out-arities are excluded, and so are multiply-used data-items; and, furthermore, there is just one vertex that is undeparted from (or "final", if we anticipate the terminology introduced below). Thus in calculations, there are two kinds of forward branching not present in term-evaluation.

A calculation is *unary* if each of its edges is unary, i.e. one-to-one-ary. It is necessarily a forest, in which the un-arrived at vertices are the roots, and undeparted-from vertices are the leaves. A unary calculation is *single-threaded* if, in addition, it has just one root, and no vertex has more than one edge departing from it.

A path through a state-transition system is stringy, and can be expressed as a single-threaded calculation, in which each vertex is labeled by a state, perhaps repeated, and they are linked by unary in-and-out-edges, each labeled by a member of the systems controlling alphabet. It is reasonable, but not essential, for the sources and targets of these edges to be un-indexed rather than indexed singletons.

There is another small design-choice about whether the first index appears as the target of no edge or as the target of a zero-to-one-ary edge. Dually for the last. The significance of this will be more easily discussed after the notions of initial and final vertices have been introduced.

This does not quite exhaust the shapes of calculation that are possible if multariness is excluded. The topic is taken up in 2.10.

2.1.6 Convergence At A Vertex? – Pariahs

Two arrow-tails departing from the same vertex is an indication that this particular occurrences of this data item is participating in more than one way as data for a step. Two arrow-heads arriving at the same vertex is, however, violating of our intuition. It violates our intention that each calculated data-item results from just one calculational occurrence of an operation. If, in later developments, calculations containing these "pariahs", or *multiple-produced* vertices, deserve attention they should probably be called "pre-calculations", because they seem to indicate a range-of-variability over calculations.

In the working practice of computing, redundancy or over-determinedness is sometimes desired for added confidence. It is not relegated to "pariah" status. It can be expressed in a calculation that includes either the confirmatory checking or the conflict resolution – not both. This is the first of many later references to situations concerning variations of interpretive mechanisms, where a contrast between calculations offers an alternative means of explanation of what is usually displayed as a contrast between programs.

2.1.7 Cycles In Calculations?

In a graph having no multariness, a cycle is likely to involve some vertex at which there are multiple arrivals – two edges converging. In calculations, this kind of cycles is ruled out merely on account of the ban on "pariahs". But in polygraphs, banning convergence at a vertex does not mean banning convergence. In the presence of in-multary-ness, the convergence can be in an edge instead of at a vertex. Thus the way is opened for our intuition to see significance in cycles without being violated by convergence at a vertex. If cycles are to be excluded it must be for some other fault than convergence at a vertex. Legalistically, convergence at an edge facilitates inclusion of cycles without introducing over-determined vertices. But it does better than that. It provides an escape from an intuitive objection that might appear at first sight to damn any cycle – namely: how could it, calculationally, start? We now briefly anticipate here a subsequent and precise account of "fragments" of a calculation. A *fragment* is a subset of its edges with at least enough vertices to include their targets and sources, and all with the same labels. This allows a vertex to be isolated with respect to the fragment, but not with respect to the context. Calculations will be seen to provide a range of views of the same calculational events. In particular, if boundaries can be envisaged around multi-edge fragments then these might coarsely be viewed as single composable steps. If this is reminiscent of a procedure-call standing in for in–line expansion then give pause. Our edges and fragments are not program-steps. They are non-parametric run-time steps.

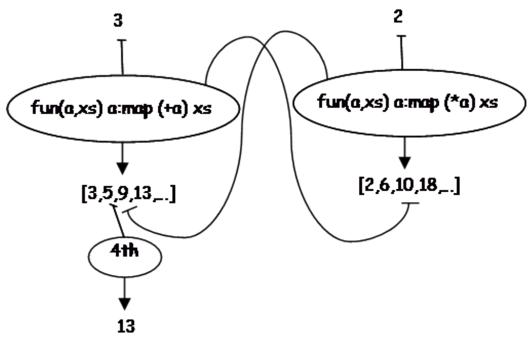

Fig. 8 Cycles

Thus a specific instance of, say, (an occurrence of) a stream-operation can be expected, when blown up, to be a multi-edge fragment whose edges deal with individual specific items of input and output streams. For example, if the source vertices of (an occurrence of) a stream-operation include a (specific value for a) start-up parameter, and a stream, then two such composite steps might be plugged into each other. The resulting cycle (fig 8) becomes reasonable in terms of our intuitions about calculation providing it can be seen to be disentangle-able by zooming into their interiors (fig 9).

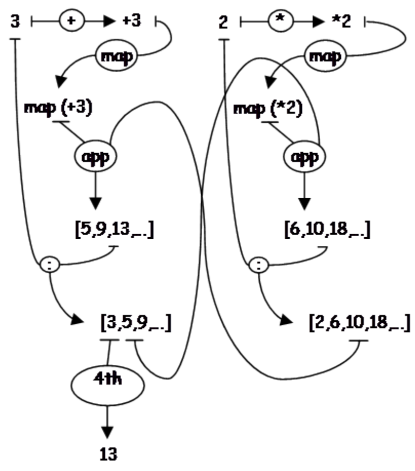

Fig. 9 Unwinding A Cycle

In a telescopic regard the boundaries, with their embarrassing mutual clutch, melt away. The finer grained configuration of edges stands revealed as cycle-free. It is tempting to conclude that the attribute of cycleness is in the eye of the beholder, not in the calculation. But the technical terminology of calculations provides a sharper utterance. A calculation *is* what the eye sees and remembers of a historical process. If, as suggested by the above stream example, one eye can see cycles where another sees none, this is not two views of a calculation. It is two calculations that themselves sharply characterize two views of a fuzzier notion – namely what is going on.

Here is the strongest claim that is to be made for the notion of calculation. It replaces fuzzy views of a fuzzier phenomenon by a unified framework for sharp views of the same fuzzier phenomenon.

The questionableness of cycles can also be assuaged by turning the telescope the other way around. If, in a non-cyclic graph, you draw some boundaries around fragments, and then take a blurred view of their interiors, are you going to ban the ones in mutual embrace? After all, they may be exactly the fragments that make good sense at a coarse-grained level. It is acceptable that by selecting boundaries around two fragments of a non-cyclic calculation we may introduce a "macro"-cycle, two composite steps that depend on each other. To the objection, namely: How could it, calculationally start?, there is an answer: Lazily!

This topic is taken up in section 3.5, where special attention is paid to an apparent paradox. The need for laziness is indeed a property that can be attributed to a calculation. Yet it is not, except for a degenerate situation, a property that can meaningfully be attributed or denied for a single edge. Anticipating here the resolution of this paradox, we can observe that, for whole calculations, the need for laziness, like cyclicness, is a global property, not local to individual edges. The uninteresting exceptions to this are single re-entrant edges.

Reverting again to the position of looking from the outside inwards, any cycle in a calculation presents a challenge: is there a plausible fine-grained view, or refinement, of it that disentangles the cycle? But such questions lead directly out of the topic in hand, and into territory familiar to students of implementation and verification. It would be rash at the present stage of development to claim that calculations have anything to offer to those studies, except a pedagogic tool, a metaphor, and an opportunity to reformulate on slightly shifted ground.

2.1.8 Two Terminological Disclaimers

Graphs have sometimes been called "indexed graphs" when the edges departing from each vertex are (distinctly) indexed, e.g., serially ordered, or named "car" and "cdr". Our graphs are not of this kind. For a vertex that is the source of several edges, any index of an arrow tail that departs from it relates to the edge itself and not the vertex. Thus they need not be distinct.

In graphs two or more edges have sometimes been called "parallel" when they agree as to both the source and target. In the present context this feature is, except in the zero-out-ary case, excluded by an embargo on the "pariah" situation described above. It is at any rate relegated to what have just be dubbed "pre-calculations". Located there, it would present a choice between two perhaps identically labeled edges that step from the same source to the same target.

Notice that the exclusion of multiply-produced vertices a fortiori excludes two parallel occurrences of the same calculational step. Anticipating a later section, the steps of a calculation constitute a bag not a set, and by no means characterize it. By contrast, the edges of a calculation constitute a set not a bag, and do.

Notice also that two edges may be identical in their (multi-used) source vertices, and their operation, and their target indices, and their target labels, but not of course their target vertices.

With multi-out-ary-ness there arises the quite different possibility of the parallel arrow-heads. These are, of course, excluded by the same "pariah" embargo.

On the other hand, and again quite differently, the presence of parallel arrow tails is merely an indication that an (instance of a) multary operation is making double use of a part of its source data – as when 7 is added to the same occurrence of 7.

2.2 Data Items

The term "data-item" (dit) here comprises all things that might occur as labels, including operations. A vertex label is permissibly an operation. An edge-label prescriptively. The labeling throughout a calculation is "consistent" in the following unsurprising sense. Each edge has an argument and a result. Its argument is **either** the label is its unindexed source, **or** (disjointly) the indexed set comprising (in the obvious way) the zero or more labels of its indexed source. Likewise for its result.

Consistency is the requirement that for each edge the argument-result pair belongs to the extension of the edge-label. Beyond being a set of pairs, or a characterizing property of a set of pairs, nothing need be said about what counts as an extension. In particular no notion of types, or of type uniformity, or totalness is supposed.

For example, a tail-less, head-less edge does not breach consistency, provided the pair (nothing,nothing) belongs to the extension of its operation. Nor is the extension of an operation required to be functional. For example: the non-functional operation that derives from an input number any number greater than it. This does not open the door to a notion of "nondeterminate" calculation. A calculation is not a variable calculation any more than 77 is a variable number. A program may be non-determinate for a host of reasons, starting with the range-of-variability of its data. But a calculation is not.

From the above definitions of argument and result there arises a significant question concerning index-sets that are empty. We return to this later in 2.10.

As a constraint on labeling, this notion of consistency is a weak requirement providing the graph is cycle-free. Hence the same is true for cycles providing there is a known justification – a non-cyclic refinement of the kind described in section 2.1.7. Consistency gets non-trivial when the justification is yet to be discovered. This is outside the present topic.

2.3 Calculations Are Semantic

As suggested above, the labels are "semantic". This leaves, as issues separate from the concept of calculations, questions about how they might be described and displayed, and about how consistency, equality, applicability, etc., might be studied. In many straightforward cases involving numbers, sentences and other familiar dits (including operations), the worst problems arise when the size or repetitiousness calls for some naming system that is external to the calculation itself. But this anticipates a later topic of what might count as descriptions of calculations.

Many language-designers have adopted a similar separation of issues when they have taken abstract syntax as a point of departure, and thus left many subtle questions about ambiguity and parsing for independent design, perhaps even going so far as to rely on disambiguation by dialogue.

There are precedents. At a time in the past when a programming system was expected to be a list of its operations – APL was currently the big novelty – there was considerable audacity about a system that consisted exclusively of connections and configurations. To the unfriendly, and probably rhetorically intended question: "yes yes but what are the operations? What kind of data and i/o? What can it **do**?", the only answer was "Oh, I see what *you* mean. That's all just a parameter It does anything you want it to. You just choose the basics." Similarly with the vertex- and edge-labels of calculations. Since the labels are abstract, calculations are necessarily at least as abstract as the dits in them.

2.4 Abstract Graphs, Not Concrete Graphs

Calculations are abstract for another more familiar reason. The information-content of a calculation rests entirely in its shape, its labels, and its indexing of sources and targets – not in the edges or vertices themselves. If the vertices are 2D points on a screen then consistent dragging does not change the calculation.

As in many other contexts involving graphs, the vertices and edges figure merely as hooks on each of which there hangs a dit. Of the various informational components, vertices and edges are to be abstracted, and of course not indices and labels. The same point is made by being blind to the difference between isomorphic graphs; or by drawing a graph with its labels, but with no acknowledgment of what constitute the vertices of edges beyond their being small regions connected by spidery lines. Again the same point is made when, in programming a graph-operation, we avoid dependence on the memory locations occupied by its components, while at the same time embracing dependence on *equality* of such locations – for shared fragments.

2.5 Displays Of Calculations

2.5.1 Pictures

A network of drawn lines with an annotation for each vertex- and edge-label is the most natural way to draw a calculation. An edge, multary-in and multary-out, can look a bit like a spider whose body is a roundish outline enclosing a display of its operation. Isolated vertices and detached edges can be incorporated into such a drawing. Note that every detached edge is 0-to-0-ary but not vice versa. Arrow-tails, annotated with their indices when such exist, arrive at this (occurrence of an) operation, from argument-dits; likewise arrow-heads depart from it, to result-dits.

If a calculation is drawn with a downwards and perhaps rightwards tendency, then source-indices 1,2,3,... can be implicit in the clockwise order of their attachment points to the spider-body. Likewise anti-clockwise for serially indexed results. There is a small but unavoidable risk of confusion concerning a source or target that (genuinely i.e. abstractly) lacks an index.

The vertices of these displays carry no other information than their dit-displays. This reflects their abstractness in the sense drawn attention to above. Such a drawing misleadingly suggests an ordinary (non-poly)graph in which two kinds of nodes might be discernible. Each of these "nodes" actually indicates either a vertex or an edge (disjointly), and any "node"-to-"node" path visits these alternately. Such drawings are somewhat reminiscent of Petri-nets, and of DFDs, and of flow-charts, and of state-transition systems. In Later sections of these notes we examine possible connections. The points of interest will be cycles, and mutables, i.e. variables.

As an explanatory tool, the problem of size is not one that arises in coarse-grained, or zoomed, or toy situations. Nor is the conceptual effort of sharply characterizing coarse-grained components of calculations any greater than is already seen as acceptable these days in modular design. A starker difficulty is that these pictures are not easy to process. They provide an application for a graphical editor that can drag and patch while maintaining consistency among captions and text. Perhaps this, rather than size, is why explanatory texts display a lot more programs than calculations.

2.5.2 Textual Display

There is another display technique, more writable (i.e. typable in the old-fashioned sense) and less readable than network pictures. It depends on vertex-identity, though not necessarily on edge-identity, and so the graph it displays is not abstract. It consists, unsurprisingly, of two tables, one for the vertices and one for the edges.

From the exclusion of parallel edges, and parallel arrow-heads it follows that the edge-table is a set not a bag. There can be no repeated item. Hence there is enough information here to characterize the concrete graph without providing edge-identities. Also, the edge-table is a set not a list. It is of course true that, viewed as a list, the order numbers would provide edge-identities, though unnecessary.

This textual display is essentially another presentation of the "occupied memory" idea discussed in section 2.1.2. The resemblance is probably close enough to affront the reader. One table shows the dit at each vertex. If vertices are viewed as names, there are synonyms but no homonyms. (That would imply some sens in which a dit at a vertex could "change"). If vertices are taken to be letters, and hence are "self-displaying", then items of this table might look like: x is 3, y is 4, z is 7, v is 4, t is 1, s is 5, k is (7,8), p is plus, and q is "communism is dead".

The other table consists of source-operation-target triples. Each triple comprises (here changing slightly their order): (a display of) the operation, and (displays of) its source and target vertices together with (displays of) any indices. Edge labels might look like: z is plus(x,y), y is greater than x, (v,t) is devrem(w,y), w is apply(p,k), and q is ideology(). These illustrations depend on some intended rule by which their punctuation indicates indices such as the numbers 1 and 2, or the strings "first" and "second".

The appearance in these illustrations of a dit (7,8) invites a question about the relationship between indexing within a label, and the indexing already introduced as a feature of graphs. This question is taken up later in 2.9.

This display is essentially the single-assigned, single-executed set-up that was mentioned earlier with helpful intent.

In the uninteresting circumstance of no synonyms, i.e. every vertex having a distinct dit, the former table is discardable, and the edge items look like: 7 is plus(3,4), 4 is greater than 3, (5,1) is divrem(21,4), 17 is apply(plus,(7,8)), and "communism is dead" is ideology().

There are overtones here of Prolog or (if non-functional operations are excluded) of Miranda. But that is wandering out of the world of (ground) calculations and into the world of their connections with programs – a later topic.

2.6 Separate "Nodes" For Edges??

In diagrams similar to calculations, e.g., in the trees that are drawn for term-evaluation and for parses and for logic derivations, there is a subtree and therefore a root-node corresponding to each nested composite part. It is common for the operation that relates whole to parts to be indicated *at* the root-node, rather then mid-way between it and (the roots of) its parts.

It is relevant here to note that the information-content of such diagrams is often curiously independent of whether the arrows point from whole to parts ("distributing") or from parts to while ("collecting"), and that this is true event when, instead of trees there is a graph-structure that indicates the shared-ness of the parts. Little or nothing is lost if linking lines are unoriented, and something gained. The operation symbol, and hence the entire tree, can be overloaded to serve several purposes – e.g. expression synthesis ("collecting"), expression analysis ("distributing"), inductive expression evaluation or translation (by synthesizing collecting), recursive expression evaluation or translation (by analyzing, distributing) etc.

So, by handing the operation on the whole rather than the parts, or midway between them, no clash arises, *with the proviso that* there is not a whole that is composed of parts in more than one way – "no confusion", as the adage goes, or "monotectonic" as Curry said. Moreover the embarrassment is avoided of having to invent a notation that hangs one operation on several parts, or in the case when several means none, on no parts.

In calculations, although the proviso is exactly guaranteed by an embargo on "pariahs", the embarrassment is unavoidable in either direction. There is no requirement to indicate a dit that occurs as an intermediate result in more than one way. But multiple targets would not neatly be accommodated. Nor would a targetless edge. Moreover, we shall later see the benefit of being able to distinguish between one occurrence of the operation divrem, and a pair of occurrences of its two projected operations div and rem.

There is also a pedagogic advantage. The customary overloading of uncles or squares, or liberated, to refer both to the relation and to the set of its target individuals risks inefficient communication when it comes to names, or last, or parts – "part is a relation not a predicate" chants the teacher, if there is one.

And, more fundamental than these, there is another ground for the design decision being defended here. The unary special case of this is familiar, and demonstrates the point. In a diagram for a state transition system, arrow-heads *can* converge, and the transition-signals attached to converging arrows must therefore be properties of the arrows, not of the nodes. The state transition graph of a machine expresses the full range of possible calculations. Each is a unary chain that is a path or orbit through the underlying machine.

Generalizing to multi-in and multi-out operations, the underlying poly-graph has dits for vertices, and expresses every instance of every operation. Each calculation is a "poly-path", or "poly-orbit" through the underlying machine. In this underlying machine, or "computational field" just one of the vertices is the number 7 and there are many ways of arriving at this dit . . . $3 + 4 = 10$ - 3 etc. In the intimate dependence that a calculation has on its graph it would be weird for the edge-label to have slid along to the target node.

To sum up: at each vertex there are zero or more arrow-tails (departing divergently from it), and zero or more arrow-heads (arriving convergently at it). By contrast, for an (instance of an) operation no such restriction applies. Zero or more tails arrive at it and zero or more heads depart from it – with a possibly embargo on *both* being zero, as already mentioned.

2.7 77 Occurrences Of 7 Is plus(3,4)

A "step" means an operation together with one instance of it. In a calculation each edge has one step, comprising its label together with its argument and result. This dit-pair belongs to the extension of the operation. There may be many vertices having the same dit, and, even more plausibly, many edges having the same operation. Even, many edges having the same step, i.e., in agreement concerning the label, source dit(s), and target dit(s). For example, the step of testing the number 7 for zero-ness might occur at many edges. And likewise of course the dit 7 and the dit false might occur at many vertices, not necessarily in this configuration.

We shall, in talking about a calculation, have occasion to distinguish between a dit and an occurrence of it – for example observing that the calculation contains 77 occurrences of numbers between 10 and 20, *and* also contains three of the numbers between 10 and 20.

The word "vertex" will be overloaded to be our abbreviation for "occurrence of a dit". Similarly, to distinguish carefully between steps and occurrences of a step we shall refer to the latter by overloading the word "edge". So there might, in a particular calculation, be, say, seventy-seven 7-vertices, and sixty-six 7-tested-for-zero-ness edges.

If this overloading were to have been avoided, then perhaps the word "action" is apt for "occurrence of a step". Also perhaps "dit" should have been dedicated to their occurrences, in which case the terminological gap thus arising might have been filled by "data-value" (pace ML, which is severer than the present study in its attributions of first-class citizenship). With that arrangement, here rejected, a data-item would have been an occurrence, in a particular calculation, of a data-value.

One final remark to hammer home this distinction. In a discarded version of this text, a *step* was defined as an edge together with its label, its source and target, and their labels. This vacuated any question about how many times some step occurs in a calculation.

2.8 The Initial And Final Vertices

.

An edge has a source and a target. A step has an argument and a result. (Also of course an operation-instance has an argument and a result.) What about an entire calculation? An *initial* of a graph is a vertex that is not the target, or part-target, of any of its edges. Dually, a *final* is a vertex that is not a source, or part-source, of any of the edges. These concepts are relative to a particular graph, and may demand careful phrasing when several graphs are simultaneously under discussion. In particular, an initial of a fragment of a graph, viewed as itself a graph, i.e. a subgraph, is usually not an initial of the including graph. In the extreme, a single vertex, viewed as a subgraph, is its own initial and final. And for a single edge, treated as a subgraph, its source- and target-vertices are, barring cycles, exactly its initials and finals.

A vertex that is neither initial nor final is an intermediate vertex, or an *intermediate* of the graph. A vertex that is both initial and final is an *isolated* vertex, or an *isolate.* Such vertices have a role. For example, when we discuss fragments it will violate intuition to exclude a fragment that describes a dit being transmitted without being operated on.

These notions carry through when we hang labels on the graph and turn it into a calculation.

To treat multi-edge calculations as composable units requires some technical design-choices about their intended incidence. If we limit their incidence to a superposition of finals onto initials then some of this has already been effected. The chosen notion of initials and finals has warded off the famous "leakage" complication of automata theory. There is another bit of the design-choice that can be contemplated now.

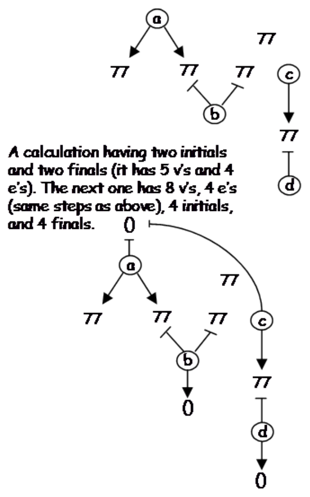
a
77
77
77
77
b
c
77
d
A calculation having two initials and two finals (it has 5 v's and 4 e's). The next one has 8 v's, 4 e's (same steps as above), 4 initials, and 4 finals.
0
a
77
77
77
77
b
0
c
77
d
0

Fig. 10 Initials and finals

When there is exactly one initial, and its participation as a source is exclusively as an unindexed source, then it might be supposed that the calculation, viewed from outside as a composable unit, has one unindexed initial. Likewise for finals, but with a slight simplification, on account of the fact that no vertex, final or otherwise, can participate more than once as a target. However, the proper setting for this is less assumptive. What is required is the kind of renaming operations customary in process algebras, but slightly generalized to accommodate the disjointness feature. This idea will be completed later.

Protocol about indexing for composition ought at this point to be tentative. It will be influenced by the introduction of parametrization in which the abnormal termination or non-termination of a program may figure as a contingent property. By contrast it should be noted that for calculations, the notion of normal/abnormal termination has no innate meaning. This would not be the case were a calculation to include some kind of flags, exogenously imposed, as indications of finality. The notion of finality defined above is exogenous, i.e. defined dependently upon the characterizing attributes of a calculation. Our design-choice has rendered it meaningless to ask concerning a calculation: Did it/is it/has it finish(ed)/terminate(d)?

For example, the same calculation might be an aborted eager performance that fell down on an account of over-eagerness, or a performance of a different program that terminated in an orderly manner.

The entire calculation of the dits that label its initial vertices and all of its edges, is called the *demands* of a calculation. This collection will sometimes be viewed as a bag, other times as a set. Also sometimes the initials and the edges will be segregated, and other times conflated.

A familiar situation will be met when we study the senses in which a calculation might be imitated by another calculation whose demands are similar or different.

Leaving aside for the moment any more precision about how a collection of dits might be viewed, we can anticipate an inclusion relation amongst collection. We shall say that a calculation is *resourced* by any dit-collection that includes its demands, We also anticipate some sense in which a calculation *supplies* a dit-collection. Whether this should be its terminals only, or the totality of its vertex-labels is another technical question here deferred.

Concerning edge labels, it seems reasonable to anticipate that they will be in no sense supplied by it. For an operation to be supplied, it must be a vertex-label. As such no special concern is needed.

2.9 One Indexed Dit At A Vertex *Versus* Some Dits At Indexed Vertices

Indexing arises in two ways – as part of the graph-structure when a source or target is indexed and as part of the dit-structure of a single, usually composite, dit. The definitions of source and argument that have been adopted here introduce a bump into the correspondence between edges and steps. A step whose argument is indexed may occur either as an edge with an indexed source (usually multi-tailed), or as an edge with an unindexed source (always uni-tailed) bearing an indexed label. Two examples follow, respectively 2-ary (fig 11) and 0-ary (fig 12).

For the purposes of this paragraph we consider 2-ples to be indexed sets of numbers, or of vertices or whatever, whose indexes are the strings "first" and "second". Then the step from the 2-ple (3,4), by means of plus, to the number 7, can occur either as a two tailed edge or as a one-tailed edge. These have different shapes but the same argument. To paraphrase this, the source of a plus-step that adds 3 and 4 may be either an indexed set of two vertices each bearing a number, or (disjointly) a single unindexed vertex bearing an indexed pair of numbers.

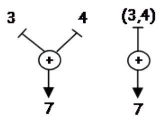

Fig. 11 2-ary Example

Again, a step that derives the dit 77 from nothing (i.e. from an empty indexed set of dits), can occur either as a tail-less edge (i.e. one whose source is the empty indexed set of vertices), or as a uni-tailed edge whose source is unindexed and is labeled by nothing. The argument nothing occurs both as the absence of arrow-tails, and as a single unindexed arrow-tail from (a vertex labeled by) the empty indexed set.

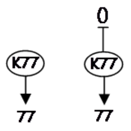

Fig. 12 0-ary Example

Thus, the 2-ple (3,4) can occur as an argument without occurring as a vertex label; or without 3 and 4 occurring as vertex-labels. Likewise the 0-ple (), i.e. nothing, can occur as an argument without occurring as a vertex-label. This possibility has been smoothly anticipated by the definitions already given for the notions of source and argument. And dually for target and result.

The in-arity is a property of an edge, not of a step. It is the number of arrow-tails attached to the spider-body, i.e., the size of the source if it is indexed and 1 if it is not. This does not correspond smoothly to the size of the argument, should there be such a size. Every edge has a numerical in-arity, greater than or equal to zero, whereas to attribute an arity to every step the disjointness feature must be acknowledged. Likewise for out-arity.

If size is to be an attribute of dits, then its range-of-variability could be the numbers disjointly unioned with a singleton that represents the property of not being indexed. This attribute of dits may be relevant if and when there is some discussion of types, i.e., of operations whose extensions have some uniformity.

In the design of the concept of calculation, we have rejected the option of keeping separate these two uses of indexing. Thus was avoided the unattractive presence either of arguments (and results) that are "second-class" in the sense of not being dits, or of two parallel identical but distinct systems of indexing, or of an awkward two-layer indexing that would arise from an insistence that every source (and target) be indexed.

The design-options are summed up (referring to sources only and omitting the repetitious facts about targets) this: **Either:** Two indexing systems and (whether or not unindexed sources are excluded) **either:** Include both in dits...then redundancy. **or:** Exclude source-indexing from dits...then second-class arguments. **Or**: Conflate the indexing systems (and hence eliminate the choice about whether source indexing is excluded from dits) and **either**: Exclude non-indexed sources (i.e. abandon the disjointness feature)...then two-layer indexing. **or**: Include the disjointness feature...then the design presented here.

We have also rejected the option of corralling all indexed arguments into special constructing (or collecting) steps, and dually of corralling indexed results into special selecting (or distributing) steps. It is true that given the appropriate operations (and it should be noted that with our cavalier disregard of types there need be no proliferation of such operations), there is a trivial "normalization", a calculation-to-calculation operation, that eliminates all indexing of sources and targets except in those special contexts.

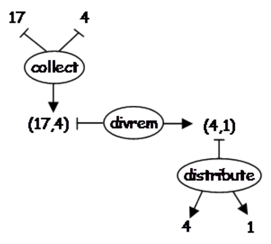

Fig. 13 Collect/Distribute

But the distinction thereby elided is one that will turn out to have been helpful when calculations are studied for possibly orderings of their steps, for single-threadedness, over-writing, re-use of names or locations, and forgetting.

A total ban on indexed targets would trade mult-out-aries like divrem for their projected operations like div and rem. But a calculation whose "resource" includes divrem but lacks div and rem suffers a constraint that can be expressed in a calculation but would otherwise be lost.

A total ban on indexed sources would eliminate most of what calculations are good for, with the exception of (distributive) abstract phrase analysis.

If both are excluded, thus abandoning collecting and distributing, the notion of calculation would collapse into the notion of a branching forest of paths, as when a state transition system is unfolded from some start-state. Special cases include a set of single-threads (not a bag, not a list), and of course one single-thread. What would also collapse would be the study of how (multary-in-and-out) calculations are imitable by single-threads, such as stacks.

What would survive would be the study of imitating a big-state system by a configuration of smaller-state systems, either non-communicating as in automata-decomposition, or communicating as in Petri-nets and process-algebras.

It is on the relative poverty of such systems that any merits of calculations are based.

Indexed sets are often called records, bindings, environments, look-up tables etc. No harm other than a jarring with custom would have resulted from such alternatives. It might have been necessary to stress the difference between a record of vertices and a single vertex labeled by a record of dits.

In the following section two aspects of empty indexed sets are described. They provide a technical trick that avoids an upwards or downwards dangling dit from being mistaken for an initial or final. They act as tourniquets to tie off danglers. But first the slightly elusive presence of nothing-dits (or rather, of occurrences of the nothing dit) is discussed. It may be relevant now to observe again that a calculation is more than a set of steps. Apart from the unusual and unimportant case when no dit occurs multiply, the connectivity cannot be inferred from merely the set, or bag, of steps.

2.10 Nothing

2.10.1 Two Kinds Of Occurrence Of Nothing

The previous section mentions a small perturbation, referred to as "corralling" the indexing into special operations that collect and distribute. This perturbation does not disturb the initials, finals, or the dependency relation (except by the insertion of new intermediates). If it merits attention, then so does another, which does not. In the zero-arity case the difference is subtle and has repercussions on dependency.

One consequence of the disjointness feature is that two calculations may agree totally as to the steps (i.e. each argument-operation-result triple) and their dependency-relation while differing slightly as to their vertices and edges (i.e. each argument-source-operation-target-result quintuple), the difference being exactly that between an already "collected" argument (resp. subsequently "distributed" result) and an "uncollected" argument (resp. "undistributed" result). This phenomenon is possible with the proviso that all these source dits are initial (resp. these target dits are final). So the small disagreements are confined to initials and finals.

The proviso is needed. If a source has non-initial parts they may be derived separately rather than already packaged as a single dit. Then there is not an alternative calculation that differs slightly in this way. There is, of course, the corralled variant that results from inserting an extra step, and does preserve initials, finals, and dependency.

The preceding paragraph has an exact dual.If a target has non-final parts, they may be used separately rather than consumed as a single packaged dit. Then there is ... etc.

In the zero case this proviso does of course hold, the difference being exactly the presence or absence of an extra nothing vertex. This vertex, if present, is either en extra initial or final (not both), or is an extra link in the dependency relation – perhaps thereby itself losing the status of initial or final.

Summing up, the replacement of an explicit nothing-source by an implicit one damages the dependency relation unless it is initial. The reverse change merely introduces a plausibly harmless initial nothing, providing it is not done by unreasonably hooking onto an already present nothing vertex.

And likewise for the replacement of an explicit nothing target, and for the reverse.

As an intermediate, a nothing is reminiscent of the fleeting shadow of an un0named transient state that is sometimes signalled by "void" in the procedure-types.

2.10.2 Why Not Just Look At The Program?

The pedantic study of a small difference between two calculations, making no mention of the programs that they might have come from, seems to turn tradition upside-down. But this is a tradition of theoretical study. It unrealistically neglects the dynamics of program-making. A program in the making is perceived as a blur of close alternatives, together with lots of clues as to which is which, including diagnostics yielded by specific but unclearly identified points in the blur (compile-time diagnostics), and diagnostics yielded by specific runs of these points which specific data (run-time diagnostics). Programmers puzzle over subtle differences between runs. So calculations offer pedagogically an abstracted model of a familiar working situation.

But they do more. If a run-time diagnostic shows the result of a flow-analysis of the dependencies between intermediate results, then it is showing something close to our calculations. A machine-free interface that lies between a programming language and its usually heavily specialized records of traces is an interface that might facilitate thinking both up and down, both analytically and synthetically. It makes a space that can incorporate both tradition and its inverse.

2.10.3 Breeders And Killers

If initials and finals are to play a role in the part-wise topography of calculations, then it must be stressed that an initial and a target of a tail-less edge, are intuitively very different. Almost dually, for a final and a "victim", i.e. a vertex that is a source, but only of head-less edges. Notice the bump in duality. Paraphrasing, there are two ways in which a vertex may have no dependent vertices, i.e. no vertices following it. There may be no departures from it, or there may be a (or more than one) departure, but only head-less.

Tail-less edges and head-less edges will be called *breeders* and *killers*, respectively. It should be noted that these are attributes of edges, not of steps – and certainly not of the operations whose instances might include them, along with other argument-result pairs. Also, they are attributes of edges, not of vertices. There is no embargo on a breeder and a non-breeder departing from the same (nothing-)vertex, although its intuitive significance is not apparent.

The word "constant" can then be dedicated to constant operations, including constant functions. For example, an operation whose extension contains nothing (sic) but (nothing-number) pairs is a (perhaps non-deterministic) constant. Each tail-less occurrence of it is a number-breeder. It is a non-determinate zero-ary operation, which is a special case of a non-determinate constant. For it to be determinate, its extension cannot be plural.

There might appear to be a trade-off between initial vertices and breeder-steps. Corresponding to a dit there are breeder-steps, each of whose results is that dit. Two calculations differing only in this respect have approximately the same demands –

trading a dit for a breeder. But, as regards composition by plugging finals into initials, their properties are very different.

Initials are a demand that might be "met" by composition with a "previous" calculation. Intuitively, operations are a demand to be met by a deeper-rooted resource, in a sense to be described in a later section. In the extreme, corresponding to any calculation there is indeed another tail-less calculation whose demands are entirely its operations. Compositionally, an initial-less fragment plays the role of a "macro"-breeder – whatever its finals, they arise entirely from an initial-less calculation. Initials are part of the demands. Finals are not. So the dual is not quite symmetric. Instead of a final vertex there is a "victim" of a head-less edge that "consumes" it. This, should it be desirable, must be a net addition to the demands, even when the final happens to be an initial.

2.10.4 Zero-Arity Ties Up Loose Ends

Calculations will later be presented as views of what happens when a program is executed. From various view-points, various execution features are visible or unseen. Zero-arity, in and/or out, enables steps to be elided without exposing vertices to initiality or finality. Across a broad spectrum of detailedness there is one view that is fairly extreme. Everything is disregarded except input and output. This view pays attention only to data and results. The calculation perceived is an empty hull that includes none of the intervening steps.

In an empty hull the only edges are zero-out-aries that depart from its initials and zero-in-aries that arrive at its finals. Even these are absent in the case of an initial that is also a final – remember that isolates are not excluded and their intuitive usefulness for the composability of fragments has been claimed to be useful.

The configurations possible with just edges that are either tail-less or head-less (or both) are almost exhausted by empty hulls, but not totally. In such a configuration, each vertex is either initial or final or both (i.e. isolated) or neither. In the last case it can play no role. It springs from nothing (not even from a nothing-dit of the calculation and it's not initial). It contributes to nothing (not even to a nothing-dit of the calculation and it's not final).

This is a single-vertex special case of a wider phenomenon – a sub-calculation wholly detached from initials and finals. It can arise unobjectionably in the course of unprogrammed exploratory dialog, or as a pathological special case of a programmed general administration, or as an oversight of careless programming.

Summing up: As an argument the nothing-dit can occur in two ways. It is the argument of a tail-less edge that cuts its target vertices off from all ancestry. They are not even initials. Or, it is the argument of one or more one-in-ary edges, and is either an initial or else it is an intermediate vertex.

Almost dually: Occurring as a result, the nothing-dit can be the result of a head-less edge that cuts off its sources from all descendants. They are not even finals. Or, it can be the result of exactly one one-out-ary edge and is either intermediate or final.

There are other less extreme circumstances in which a calculation might, for some of its intermediate vertices, leave no room for the progeny and/or the ancestry. Such vertices are an essential ingredient of the above-mentioned broad spectrum of views to be described in a later chapter.

It is strange to look back now at an earlier developmental stage of these ideas in which it seemed important to insist on some notion of connectedness in these respects, variously formulated in terms of backwards and/or forwards reach ability.

In the present formulation, the dependence-closure of the set of initials necessarily contains every vertex. The intuitively stronger requirement for the "relevance" of every initial excludes any vertex that does not depend on at least one initial, and in particular excludes initial-less calculations.

The nearest we can get is the trivially true assertion that every vertex is dependent on (i.e. is contributed to by) at least one initial or breeder. (Recall that dependence is a relation over the union of vertices and edges).

The duals of these assertions also hold but have less obvious intuitive appeal. The co-dependence-closure of the set of finals necessarily contains every vertex. Every vertex is co-dependent on (i.e. contributes to) at least one final or killer. But, to balk at requiring that every vertex is co-dependent on at least one final is much less compelling. This requirement is nevertheless rejected in order to make allowance for the kind of partial views described above.

2.10.5 Different Looks At The Aame Program Execution

We shall be introducing the term "performer" for a function that, given a program and sufficient data to fix its behaviour, produces some view of this behaviour. Hence a performer that produces the empty hulls corresponds to what is ordinarily called denotational semantics.

In the broad spectrum of possibilities, empty hulls are not the most extreme. There could be a mere indication of termination, or some other attribute such as whether or not a particular operation, say writing, is demanded. It is perhaps worth observing that these are not the program-attributes uncovered by abstract interpretation. We are concerned here with the attributes of ground runs, not programs.

Moving along the spectrum in the other direction, there are views that give a full account of the paths of dependency that lead from initials to finals, but undecorated by any account of the administrative arrangements that must have played a part. It is at this point that there is the richest possibility of identifying some shared ground on which to base the behaviour of different programs, perhaps in different languages and different styles of language. The gap that is waiting to be filled is, in a simple way, apparent when we speak informally of doing the "same" thing by means of different programs, perhaps in different languages, or different modes of implementing a language, or even two "paradigmadically" different systems. Going further, there are calculations that include administrative steps such as those involving names, look-ups, procedure-calls, and exception-raising. The more there is, the less chance of shared ground, but there are also merits.

There is one matter of perennial concern to programmers that calculations can exhibit – the trade-off between unadministered special cases and administered general regulation or regularities. For example for a programmer an if-statement brings two special cases together into a single regularity. The administrative steps are: the derivation of a boolean, the derivation of a package of two alternatives (perhaps two numbers, perhaps two programs) and the use of the boolean to select one of these. Without the if-statement there must be some other way of selecting which unadministered special case is appropriate, e.g., human judgment, or prior specialization of the program. The if puts the administration into the program. Discovering regularities is the very essence of automation.

The relationship between the if-program and the two special cases can be approached from either side. Narrowing the special cases down to specific ground instances, given two calculations we can seek to add administrative steps to each of them with the goal of "unifying" them, i.e., reaching a pair of calculations for which there is a single program, and a single performer, according to which each of them is, with differing data, a performance. At the trivial worst this means an outermost if. Conversely, given such a program plus data, its behaviour can be so viewed as to observe the selection-steps that discard the parts inappropriate to the data. By another view, it is shorn of these steps, unrevealing of what could have been done had the facts been otherwise.

This contrast will appear later as different performers for if-statements. One performer shows all the steps that derive the boolean, the package of alternatives, and the step of selection. Another elides all these, showing merely the steps relevant to the given data. The calculations hows no trace of the discarded alternative, nor even of the choice having been made.

There is also a midway viewpoint that relies on zero-out-aries. The boolean is seen to be derived, but no clue appears as to its relevance to other steps. The boolean appears with no progeny, nor is it a final. It occurs as a "victim" – the source of a killer-edge, that leaves the boolean looking like a dead-end. The calculation contains no trace of the discarded alternative, but it does acknowledge that a choice has been made.

One small design-choice has arisen here from the disjointness feature. The killing might plausibly be done with an unindexed nothing-target thus leaving a final nothing as a careless clue to the murder.

As suggested above, in an earlier stage of the development of these ideas, dead-ends were excluded. It was required of a calculation that every edge be backwards reachable from its finals. That design choice disabled the midway viewpoint just mentioned.

As with if, similarly with case-selection, records, arrays, name-lookup in environments, and address-lookup in memories. The above discussion of if foreshadows a systematic range of the views that performers can take when looking at the process of selecting a component from a composite dit.

3 Some Disclaimers And A Claim

Through the basic concepts of computing, a dividing line can be drawn separating aspects of programming that are properties of ground runs, from properties that only have meaning in terms of the program being performed. But the dividing line is not simply that of run-time *versus* compile-time. For example, failing to reach the end, is usually thought of as a run-time property. But as an attribute of calculations it makes no sense. This section analyzes aspects of the dividing line.

3.1 A Calculation Is Not The Execution Of A Program

It is misleading to think that a calculation is what arises from *the* performance of a program (relative to a sufficient fixing of its data).

In the first place there is no obvious single choice concerning what and how much detail might be conveyed by an account of program behaviour. At one extreme it might be pared down to a few hints about outcome – e.g., it didn't happen. At the other we might demand finer and finer accounts of its pre-processing, compilation, time and memory management, arithmetical micro-steps, etc.

Secondly by acknowledging certain midway postures on this spectrum, say a history shorn of names, addresses, calls and returns and other housekeeping, we are immediately led to the extremely attractive notion that the same calculation can result from different implementations of the same program, different programs, different languages, or different "paradigms".

And lastly, there are other ways in which a calculation can arise – from the search for a path through a maze, or derivation through a given logic-system, or just be serendipitous forward growth. It may be a useful idea, but currently neglected, to think of proof in logic as some performance of a parametrized proof/program.

3.2 Calculations Are Ground; A Vertex Is Not A Variable

A calculation is a particular derivation of dits from dits, composed of steps, each of which is a particular instance of a particular operation. It is ground, not parametric. A vertex is not in any straightforward sense a variable. There is not a particular range-of-variability associated with a vertex, except the various ranges that might be implied by compatibility with incident edges.

However, since everything is a dit then so are sets and arrays and lists. The range-of-variability among the elements of an aggregate dit provides a bridge between mutables and non-mutables, especially because one familiar kind of aggregate is the perhaps indefinitely long list known as a stream, or lazy list. The successive values of a variable, its time series of history, perhaps mingled with historical, chronological commentary such as time-stamping, tracing, etc., constitutes a stream.

This opens a way in which a history of a set of updatable registers, responding to instruction-steps that each assign to some of them in terms of some others, can be decomposed into a collection of the histories of each register. Each register-history is a function of some or all of the others, albeit expressed rather awkwardly on account of the need to collect them together into it all the perhaps scattered instruction-steps that contribute to it.

Viewed thus as a calculation containing streams-to-streams operations, each register is a vertex, and the edge pointing to it connected with registers that are named in its instruction-operations.

3.3 Calculations Are Ground: An Edge Is Not A Variable

The similar disclaimer holds for an edge, but less needfully. In hidebound minds, arguments may fit functions, but little thought is given to which functions fit a particular argument. Just as a particular operation implies some limited range of amenable dits, so dually a particular dit (which might be an operation) implies some limited range of applicable operations. This is the symmetry that is strikingly encountered in "currying". The same symmetry is more widely familiar as a pointless quarrel that concerns selecting a component from an aggregate dit, say an array or a record: Is the aggregate an argument to a selector function, or is it a function applied to an amenable position or index or field-name? Either way round, the result is one component of the aggregate.

The quarrel is calmly resolvable by resorting to the apply operation, which is in any case forced upon us whenever a *derived* operation is to be applied. This is exactly the pedagogic device by which, in applicative formulas, a composite operator is sugared into ordinary term-notation to avoid frightening the horses.

3.4 A Calculation Is Not For Executing: It Does Not Flow

The source-target orientation of edges suggests a rough drift of flow, concurrent and asynchronous, washing through the vertices, that cannot oppose the arrows and cannot be opposed by any historical process that the calculation reflects.

For a cycle-free calculation this rough drift might be seen as related to the obvious partial order of the vertices and/or edges. Informally, thinking of this drift as a process, its purpose might be to check consistency and/or, in the case of an edge-label that is determinate (i.e. is a determinate operation), to fill in missing results. Even with cycles, the idea need not be entirely discouraged. A rough drift of flow might wriggle through a cycle if it's in luck concerning non-contributing or irrelevant arguments, or if, following up an earlier subsection, it knows a non-cyclic refinement, or simply by plain good luck in guessing a solution.

Let us briefly resume the "Hazardously apt Analogy" of an earlier section. The unlabeled graph corresponds to these two register-sets together with source- and target-maps as occurring in the respective parts of instruction-registers (assuming some conventional detail about indexing). The labels of edges and vertices correspond respectively to (the occupants of) the operation-parts, and to the eventual occupants of the data-registers – eventual in the following sense. Suppose each data-register is either initially primed with an occupant, or (disjointly) initially vacant but destined to be singly-assigned by one of the singly-executed instructions. If this process completes with no mishap, neither from absent not unnameable data, not from attempted re-occupation, then the eventual occupied memory corresponds exactly to a cycle-free calculation.

One of the mismatches between this set-up and the labeled graphs introduced above relates to the question that was raised and temporarily adjourned. The requirement of single-assignment corresponds to a prohibition of any vertex pointed to in more than one way. This point is taken up in section 2.1.6 and after section 2.1.4.

It may also be noted that, for the instructions, their memory-positions, perhaps sequential but also perhaps arbitrarily scattered amongst the non-instructions, are irrelevant provided that the computer's interpretive mechanism is clever enough to recognize them and execute them in an order that respects their dependence. The eventual memory-state has forgotten which sequence took place. although it does record the dependency constraints among them that cannot be breached.

On the other hand, there can and should be questions about whether an analogy can be stretched to include cycles, and whether an instruction might be lazy with respect to parts of its data or result. The link between cycles and laziness is taken up in section 2.1.7.

It is tempting to think of *executing* a calculation, perhaps by choosing arbitrarily a sequence of its steps compatible with its arrows, visualizable as a conquering frontier marching jerkily onwards as each edge yields its new dit(s). But, each such choice gives rise to *another* calculation of which the successive dits correspond to the successive boundaries of the transiently completed fragment.

A closer look at how tightly this "flow" might be controlled, suggests a range of possibilities, of which the slackest extreme sees concurrent asynchronous processes, one for each injection of an initial and one for each edge, each quipped with "wait" signals and "ready" signals for its source vertices and target vertices, with arbitrary timing of their births, perhaps all simultaneous, or even contradicting edge-interdependence.

In this arrangement there are questions about synchrony or synchrony within single steps for which a calculation, to its credit, shows not answer. For example, a 2-to-2-ary edge might be "really" the parallel composition of two 1-to-1-aries, acting independently of, and perhaps asynchronously with, each other. But this distinction is properly expressed, not by two ways of "executing" a calculation, but by two *different* calculations. Similarly, a question about laziness with regard to some parts of an argument is really a question about the relationship that this particular edge bears to the operation of which it is one instance – see the next section, which explains how a single ground-instance of laziness is expressed as a *different* calculation.

A slightly less extremely slack image of flow might place the edge-interdependence in the interpretive foreground, reminiscently of the singly-assigned, single-executed registers suggested earlier as a "Hazardously apt Analogy" for a calculation. Among these various ways of viewing the "rough drift", this one is a special case. The "different calculation" happens to be not different, but exactly the same.

At the other extreme there is, at least for cycle-free cases, any totalization of the partial order. The afore-mentioned conquering army is one image of this. Or, the conquering army can be seen as consuming the graph, so that the frontier at each stage becomes the initials of the diminished calculation. Each of the successive dits of this unary calculation is itself a calculation, and the "flow" is that of a graph-rewriting system.

Such a process is state-transitional, i.e. one-to-one-ary. To express it as a calculation requires some analysis, and some design-choices about what is the dit that corresponds to each transient configuration of the conquering army, and what is the unary operation corresponding to each jerky step.

For a limited shape of tree-like calculations, stacks are a famously well-thought-out design-choice for this. A stack-history is a one-to-one-ary evaluation. A valid but less well-thought-out design-choice is to remember everything – the conquering army maintains complete archives just in case. This is the chosen method when a proof is presented as a list (as opposed to the biologically-oriented trees of natural deduction) of its successively derived items (sentences, sequents, or whatever), each annotated to indicate which bits of the archive justify it. It is a method that is difficult to avoid for a proof whose steps need non-local justification, whether it is topologically a list, a tree, or a graph.

There is, however, a proper setting for such matters in which calculations are not dynamic things that get executed. Imagine instead a programming language whose programs happen to be calculations, or rather, calculations minus the labels of their non-initial vertices. We may there bring to bear the concepts to be explained later by describing a connection between calculations and programs, or more precisely programs closed by a specific choice of data. A function by which a closed program (i.e. one that needs no more data to fix its execution), determines a calculation, will be called a "performance-rule" or "performer". So a performer is a closed program-to-calculation function. In the special case where the program *is* a calculation, a performer is a calculation-to-calculation function. In the above situation concerning two asynchronous one-to-one-aries, a performer, using knowledge about the decomposability of that particular operation, will transform the calculation into a *different* calculation.

So each image of flow introduced in this section does not provide support for thinking of calculations as dynamic entities. It is a correspondence by which a calculation, viewed as a program, yields another (or perhaps exceptionally the same) calculation.

It is difficult to think of calculations as programs without giving them a textual representation. Two styles spring to mind. Both are close to the textual display given in the previous chapter. One, Miranda-wise, treats the set of edges as an unordered set of mutually self-referential definitions. The other, ML-wise, organized them into nested where-expressions non-recursive except in so far as cycles require otherwise. This is pursued in a later section. Meantime it is instructive to go through the various images of flow mentioned above, and attempt to restate each of them for each of these styles.

There is a conjecture here, and the inevitable challenge to disconfirm it that must accompany any conjecture. Is there any distinction expressible as different ways of "interpreting" a calculation that is not expressible as a different calculation? And, less sharply, is there any distinction expressible as different programs, perhaps in different languages, that cannot be expressed as performance-rules that map programs-plus-data to calculations?

To sum up dogmatically, there are may ways of performing a calculation, and each of them usually yields a *different* calculation. In this sense, and only in this sense is it valid to think of a calculation as a dynamic entity.

3.5 Neither Strict Nor Non-Strict

It might be suspected that, in the above mention of wait- and ready-signals, there is a distinction concerning strict/non-strict that has been overlooked. That this is not so is one of the consequences of the "groundness" of calculations.

The source(s) of an edge provide those dits that contribute to that particular occurrence of that particular instance of its operation. Non-strictness is merely a discrepancy between an operation itself, as a *collection* of instances, and an individual member of this collection.

For example, among possible arguments for the usual lazy 3-ary if-operation (with obvious indexing) there are the following indexed sets: {"first":true,"second":5}, and {"first":false,"third":7}, yielding, respectively, the results 5 and 7, as well as including, say, {"first":true,"second":5,"third":666} and {"first":false,"second":888,"third":7}. For the (unusual) strict if-operation the argument domain is smaller. It excludes the "gappy" cases. This does not exhaust conceivable flavours of if. But notice that, although the "gappy" cases alone do not constitute a function, it does not qualify as a lazy function.

A connection between cycles and laziness has already been introduced. The above lazy if-operation provides an illustration, albeit trivial. Consider the 1-edge cycle whose indexed source is ("first":v,"second":vv,"third:vvv) and whose target is vvv. Suppose the vertex-labels are respectively false, 7, and 7, and the edge-label is lazy-if. Then there is a non-cyclic mapping, or refinement, of this. If the "second" tail is deleted, consistency is not damaged, and the resulting calculation has lost a cycle.

As a feature of calculations (by contrast with programs) the strict/nonstrict distinction figures only as a property of an operation itself, discernible, so to speak, only by looking at the semantic object inside the spider-body. There are lots of operations of which the step from {"first":false, "third":7} to 7 is a valid instance, Some are strict, some are nonstrict.

In programs, ranges of choice have to expressed. By contrast, for one particular run the die is cast.

3.6 Calculations Contain No Forward Branching For Selection

Among the arrows of a calculation there are two kinds of forward branching – multi-used vertices and multi-result edges. In a networkish display these show respectively as multiple arrow-tails departing divergently from the same vertex, and multiple arrow-heads departing divergently from the same (occurrence of an) operation.

This is perhaps worth paraphrasing.

Multiple arrow-tails diverging from an occurrence of a dit indicate several uses of it. Were we planning time and space we would plan to delay the dit's death until all uses had been made.

Multiple arrow heads diverging from an (occurrence of an) operation indicate that its result comprises several separately usable dits.

Forward branching in the sense of selection is not meaningful as a feature of a calculation. For a ground run, all choice has been made. This does not exclude from calculations the fragments that prepare an aggregate of alternatives, and select one of them. The items of such an aggregate are likely to be the same complicated kind of dit as arises when a calculation exhibits a programmer defined function in the process of being communicated as an argument.

Why should a calculation containing such complicated dits be if interest? For two reasons, corresponding to analysis and the synthesis of programs. A program (with enough fixing of its data) may yield, under fine-grained analysis such as is achieved by a fine-grained "performing function", a calculation that exhibits such mechanisms. This can be a halfway stage on the way to a less committed one that does not – less committed in the sense that it might have arisen from differently structured, but in some sense equivalent, programs.

Dually, the finer grained calculation might arise from seeking to "unify" it with others as part of a search for a program of which they are (with varying data) all performances.

3.6.1 Digression: Is Your Will Free?

The confusion dissolved above corresponds to one resolution of an apparent anomaly made famous by its vast philosophical literature. From a divinely cosmic vantage-point my entire life, past-present-future, is a reified, unitary, staticized, composite dit – what programmers call a first-class citizen. So whence arises my delusion of free-will? Proposed resolution: Me living is (like) me performing a program. My consciousness includes the interpretive steps, including choice-steps. My consciousness of my free-will is merely the inclusion of these choice-steps in my consciousness.

When we come to explain below the plurality of relationships that exist between programs and calculations, we shall encounter various "views" of program-performance. From some views, interpretive steps are deleted. There remain only the bare bones of what actually happened. My consciousness includes my memory of past consciousness. The difference between past and present lies in that for the past there is a greater range of views, including those from which interpretive choices are deleted. The forward movement of the present is the expansion of these abbreviated views.

Of course, the notion of godly omniscience is no more an essential feature of the apparent anomaly, or of this resolution of it, than the less extravagant notion of my own present perspective of my own past.

3.7 Calculations Contain No Forward Branching For Non-Determinateness

Multiple arrow-heads diverging from an (occurrence of an) operation do not indicate alternative results. In particular, the question whether an operation is delivering an exceptional result or a normal result does *not* show. Either one or the other, but not both, can appear in a (ground) calculation. The die is cast.

By the headings of this and other sections they are proclaimed as disclaimers. A particular case of the present disclaimer can be spelt out.

Multiple arrow-tails diverging from (and occurrence of) a dit do not indicate alternatives of behaviour. For example, for a nondeterminate state-transition system there is a unary graph. Its one-to-one-ary edges diverge from each vertex that offers choice. If all downward convergence has been unfolded the resulting forest is indeed a calculation (in our sense), but one whose finals (i.e. of those branches that terminate) comprise the bag of all possible end-states, each replicated in accordance with the number of ways of getting there. The unfolded graph is a calculation, but it does not have the merit of being a run of the system.

If on the other hand there are edges that do converge, i.e., there is a vertex to whose label there is more than one path, then the graph is not even a calculation. It is a "pre-calculation", in the sense introduced earlier, and to be discussed in a later chapter as a particular kind of program.

In either case, viewed as a program its data is the successive nudges that choose among its branches. Each path from initial to final of this (pre-)calculation is one calculation. An example of state-transition systems appeared above when non-determinate graph-rewriting was put forward as a possible image for the somewhat misleading notion of calculation "flow".

The conjecture was made there that instead of seeing such an image as lending support for a dynamic view of calculations, there is, for each such image, another calculation. Further, that the proper relation this other calculation bears to the original is that it the performance of the original in accordance when some appropriate performance rule.

Each of the various disclaimers, including the present one, is a consequence of the ground-ness of calculations. Each is obvious. The best excuse for nevertheless airing them arose during the development of this work. It did seem that the conjecture was beaten by the single-threaded nondeterminate graph-rewriting image of "flow". How might asynchronous localized steps, transforming different parts of a state, be captured by the notion of a calculation?

But of course what were then being overlooked were the inappropriateness of expecting a calculation to indicate single-threaded nondeterminateness and the correctness of viewing nudges as data.

A forward-branch in a calculation, whether it be at a vertex or an edge, does not indicate nondeterminateness, except for a very limited sense which is inherent in a programmed fork – this is the topic of a later section. It can arise from diverging arrow-tails, i.e., with a multiply-used dit, or from diverging arrow-heads, i.e., with a multi-target step. An example is the non-observable nondeterminatess concerning the relative speeds of concurrent non-memory-sharing processes during the intervals between communications, formulated as mutually dependent histories.

By contrast, for graph-rewriting, Petri-nets, and other local but asynchronous memory-sharing steps, each calculation is a unary, single-threaded possible history. A calculation does not include the rules about toe-treading. It merely exhibits individual consequences of such rules.

3.8 Calculations Contain No Forward Branching For Search

A unary calculation is necessarily forest-shaped. So there are obviously search-processes, for example looking for a particular leaf. But to express this search as a calculation means a *different* calculation.

In the natural generalization of this to multary-steps, a root-to-leaf path is generalized to the backwards, i.e., co-dependency, closure of one final.

3.9 Calculations Do Contain Forward Branching For Forking

Forward-branching, i.e. arrows that diverge, is frequently used to indicate selection among alternatives. But never in calculations.

On the other hand, forward branching in the sense of forking is indeed a structural part of calculations. Opportunities for concurrency are presented both by diverging arrow-tails and by diverging arrow-heads. Dually joins correspond to converging arrow-tails. But not of course (to complete the four combinatorial possibilities) to converging arrow-heads. This feature could only mean that the dit converged on is a result of more than one step – the afore-mentioned "pariah" situation.

A forward-branch in a calculation, whether it be at a vertex, or an edge, does not indicate nondeterminateness, except for a very limited sense which is inherent in a programmed fork. It can arise from diverging arrow-tails, i.e., with a multiply used dit, or from diverging arrow-heads, i.e. with a multi-target step. An example is the non-observable nondeterminatess concerning the relative speeds of concurrent non-memory-sharing processes during the intervals between communications, formulated as mutually dependent histories.

4 Programs And Performers

We propose to decouple the customary linkage from program to the function or relation that it realizes. What is to be interpolated is a function from data to entire calculation, or perhaps a description of a calculation. The new link connects a program-plus-data (or closed program, or program-plus-environment) to a calculation. Such a link is called a *performer*.

4.1 Different Views Of The Same Behaviour

This section spells out several views that can be taken of program-behaviour even for a low-level language that leaves little choice about how to interpret it, and about what actually happens.

The following conclusion will be drawn. If, with even this down-to-earth notion of program, there is such a wide range of views of its behaviour, then it is not surprising that, for languages designed to rely on interpreters and compilers, there is not a single obvious choice of "behavioural semantics".

Once upon a time, when programs were more prescriptive of behaviour, there might have been a set of states, called memory-states, each one an indexed set of dits (perhaps each of, say, 48 bits), always including a special one called the control-cursor whose occupant in every state is a memory index (i.e., address) and a rule that associates with each of these dits, a state-to-state operation called its effect-operation, or effect.

For this set-up, behaviour is wholly described as the indefinite repetition of a single operation called its "interpretive cycle". The immediate successor of a state is one of the results of applying the operation (in accordance with the above mentioned rule) of the memory-item pointed to by the current control-cursor. This the control-cursor would run through a sequence of (not necessarily consecutive) memory indices (i.e., addresses).

In this never-never land each instance of each of these operations might have consulted and updated only a small subset of the memory items, in accordance with a multary dit-operation called its registers-to-registers operation. (By contrast with the earlier "Hazardously apt Analogy", address modification and indirect addressing are not excluded. It is assumed merely that the registers-to-registers operation factors through the occupants of the indexed set of addresses that, for each current state, the effect operation choices for its sources. This is weak. But it does exclude, say, an operation that either adds or multiplies its arguments, depending on whether they come from odd- or even- numbered addresses).

This behaviour is a single-threaded and hence unary calculation, stepping from memory-ful to memory-ful (including the cursor) by repeated application of the interpretive cycle. Each dit is some memory-state, and there is just one operation, the interpretive cycle. By being non-determinate this operation can accommodate exogenous influences from outside the memory. For example, an input stream, or the consultation of other external information.

Each thread is a root-to-leaf path through a forest of possibilities, that is itself a non-single-threaded calculation. But this offers little support for seeing divergent arrow-tails as an expression of non-determinateness. Diverging arrow-tails here indicate that a state is being updated in multiple ways that breach single-threadedness. They either lead to multiple results, or by subsequent join, indicate that a plurality of alternative states contribute to the final state.

The inside/outside distinction is as much a matter of formulation as of reality. A change of formulation that incorporates an input stream or an interacting program into the state might transform a non-determinate system into a determinate one. Conversely, as we shall see directly, a change of formulation that excludes a control-cursor from the state, might have the reverse effect.

For the same program-behaviour there is a second and more familiar view. It is also single threaded, but each dit is a memory-state minus the control-cursor. (The correspondence is one-many on account of omitting the control-cursor). It has a plethora of operations corresponding (one-many) to the plethora of effect-operations indicated by the control-cursor. The control-cursor's familiar steady incrementation does not show in these steps. Nor will any other resetting of it. For example a pure jump-instruction will appear as the identity-operation. The control-cursor itself now figures as an exogenous influence (like an input stream) in any step that copies it into an included part of memory, and such an operation is non-determinate.

As yet, these views have not invoked the registers-to-registers feature. They depended only on the control-cursor feature. For the same program-behaviour there is a third view, one in which the multariness of calculations comes into its own. It is a calculation expressing flow-analysis of the program-execution, i.e., a ground flow-analysis. Concretely, each vertex is an address, time-stamped to characterize its most recent re-occupation. Its label is that new occupant. And each m-to-n-ary edge is likewise an address time-stamped to characterize one execution of its occupant, and its label is the semantic, registers-to-registers operation.

Recall that the graph underlying a calculation is abstract not concrete. In a calculation there is no essential role for the concrete identities of vertices and edges. It follows that in a ground flow-analysis inessential instruction-ordering has been forgotten.

It is perhaps worth acknowledging that there is no assumption here about dedicating memory areas to read-only program-space. And punning is not even noticed, let alone excluded. If prime numbers are executed or instructions are squared, its just another calculation.

We add some more candidates to this list of different views of the same behaviour, each of them perhaps of occasional interest. An elaboration of the second view above might explicitize the contributions of instruction-segments instead of drawing on an unsystematized plethora of monolithic views of whole instructions. And, for each of these there are finer and coarser variants, showing, at one extreme, every bit-operation, and at the other extreme, just a single step from start to finish.

There is a conclusion to be drawn from the preceding rather laborious account of an uncomplicated programmed machine. If, with even this down-to-earth notion of a program, there is such a wide range of views of its behaviour, then it is not surprising that, for languages designed to rely on interpreters and compilers, there is not a single obvious choice of "behavioural semantics".

4.2 Performers *Versus* Interpreters

The calculation yielded by a performer, given an interpreter-plus-data, i.e., a program-plus-*its*-data that is being interpreted, is likely to include lots of interpretive steps. Of course, if the performer is not one that deals comprehensively with the whole language in which the program is expressed, but is specialized to just this one program, then much elision is possible. Indeed, the options are exactly those for a performer of the whole *interpreted* language.

4.3 Performers Versus Behavioural Semantics

When a language is designed with a range of implementations in mind, intended to differ from one another only transparently, little wonder if what language-definers choose to count as visible, modestly shrinks from entire behaviour to mere outcome. So we must wonder why it is that denotational semantics is nevertheless often so behavioural. There are three reasons, pressing both from the pragmatic and theoretical sides.

If practice is to be theorized honestly, it must acknowledge that histories can include breakdowns, abortions, traumas that strike in what would otherwise be mid-behaviour. So an account of outcome cannot stray too far from the path that should get there, for fear that it doesn't, that it misses the point at which breakdown occurred.

For example, parsing can be elegantly expressed as the inverse of the mapping from parse-tree to string, but only over wholly parsable strings. For error-action to be seen as a possible outcome, a closer hold on the step-by-step behaviour is necessary.

Hence in practice, an account of final outcome is likely to be an account of its derivation.

Again, if theory is to be practiced economically, it must exploit nested program-structure. This leads to compositional descriptions, and in particular to compositional semantics, whether or not allowances are to be made for mishaps. So outcomes must be attributed to nested fragments as well as to whole programs. Some account of stage-by-stage behaviour is inescapable.

And there's a third reason for doubting that a great chasm separates behavioural description from output description. In so far as execution involves the meanings of a repertoire of operation-names, it is plausible to envisage a systematic adjustment of these meanings by which each derived dit is accompanied by a trace, i.e., its entire derivation from the overall demands of the program, namely from the initial dits by means of the operations. Thereby, a description of final outcome becomes a description of how it was derived.

Arising out of this third reason there is a small technicality in the design of such a systematic adjustment, that does not have to be faced in the unary case. It concerns the choice between folded and unfolded derivations. The correct backwards folding is required unless we are willing to tolerate derivations that have forgotten about how dits are multiply used.

If shared fragments of derivations are to be acknowledged, then derivations must distinguish and characterize separate occurrences of a dit. This is of course a problem that does not arise in single-threaded derivations.

The study of behavioural semantics has not come out of a wish to specify behaviour. Descriptions of the journey have been given grudgingly because there was no other way of saying where it ended. So it is not discreditable if the journeys have benefited from less analysis that might otherwise have been the case. If you can't do a perfect job without first perfecting your tool then depth-first despair is your dangerous destiny.

Neither implementers nor semanticists need feel shame about being uncritical of their unremitting unariness, or incurious about the details of calculational steps. Their own goals compel them to patch together some notion that serves the current purpose and lacks generality.

4.4 77 Different Calculations For (3+4)(5+6) – Administrative Steps

Let us focus on one straightforward three-edge, seven-vertex calculation. It has four initials, which are (occurrences of) the numbers 3, 4, 5 and 6 an one final, 77. Its other demands are two operations, addition, occurring twice, and multiplication occurring once, of (unlimited) integers. Its two intermediates, are 7 and 11, and they are the results of the two addition-steps.

This is the calculation that we might expect a performer to yield from a program-fragment "(3+4)(5+6)", and also from a fragment "(a+b)(c+d)" performed relative to an appropriate binding.

For the latter case there is another, more detailed calculation that includes four lookup-steps that derive numbers 3, 4 etc., from names "a", "b", etc. Further details that might or might not appear, depending on the choice of performer, are a binding, with or without names for the operations, and some manifestation of the program itself, perhaps with steps for phrase-breakdown.

In each variant, there are some aspects taken for granted and other spelt out. For example the steps of phrase-analysis might disregard the lookups for "+" and "*". Another variant might include them, this forcing the operations + and * to occur as vertex labels, rather than edge labels, and hence also be compelled to include apply-steps that marry each (intermediate) operation to its arguments.

Lookup is a selection operation and in the above examples, the index determining the selection is a name, such as "a". It is either an initial (i.e. no derivation acknowledged), or is calculated by steps of phase-analysis.

For selections by if, the index is a boolean, and its derivation is usually a more main-stream part of the calculation. But, as with name-lookup, there is a range-of-variability about how explicitly this shows up in a calculation. A previous section showed how, by means of nothing and breeders and killers, such chunks of derivation might at choice be included or omitted without being forced to include embarrassing danglers.

Likewise for case-selection, records, arrays, name-lookup in environments, and address-lookup in memories.

In this respect the administration of calls for in-program definitions of functions and procedures offers considerable choice. For example, a performer that forgets these steps yields calculations that do not distinguish between a call and its unfolding.