Generic Graph Semantics

Nachum Dershowitz
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
nachum@tau.ac.il

A proposed graph semantics for generic programs is elucidated. It incorporates edges indicating control flow alongside edges for data flow.

1 Motivation

Algorithms, divested of any particular linguistic attire, are state-transition systems, as argued by Knuth [10]. The *states* of an algorithm are logical structures, as suggested by Post [12] for the effective case and by Gurevich [6] for algorithms in general. The notion that computations can be described as one big loop will play a prominent rôle here and is one of the "folk theorems" of computer science [8]. These and other considerations have led to the design of abstract state machines (ASMs) as a generic model of computation able to describe arbitrary sequential algorithms [5, 6, 2]. We build loosely upon those insights and that framework in what follows.

ASMs are composed of conditional parallel assignments, expressing a single step of an algorithm. For example, **if** $x \neq y$ **then** $[x := y \parallel y := x]$ requires that $x \neq y$ be tested before the two assignments are executed in parallel to swap their values. Qua ASM, this will be repeated ad infinitum, unless x and y are initially equal, in which case nothing happens.

The ultimate goal of the ongoing investigation described herein is to devise a more precise, more flexible, and more general generic language for describing algorithms than ordinary ASMs. More precise, on account of the exact control over the order in which values are accessed and over the number of times they each are. More flexible, because it allows sequences of assignments within a single step. More general, since it is designed to incorporate additional modes of choice and nondeterminism.

To begin with, an algorithm might need to first check some precondition before proceeding with other operations. In the above ASM snippet, the purport of the test $x \neq y$ is to obviate the swap when it's ineffectual. Likewise, a program may test whether some expression is equal to 0, and only use it as a divisor when its value is determined to be nonzero, because division by zero yields no result. Thus, it behooves us to ensure that the check actually takes place before invoking the potentially problematic division operation. The issue of axiomatizing and modeling, in the ASM framework, contingent memory accesses and truly undefined instances of operations, such as division by zero, which – whenever accessed – never respond (corresponding perhaps to nonterminating function calls), has been previously explored [1]. Despite their importance, we will ignore the possibility of failure for the time being and concentrate on precision.

At the same time, one may not want to evaluate r more than once – if that might incur a large overhead. Accordingly, we need the ability to avoid unnecessary accesses even when they do terminate. Temporary variables and **let** expressions are used for such purposes in programming languages. We would like our semantics to go further than prior efforts for ASMs [1], and to allow this degree of precision of execution as well. In the same vein, we would like to incorporate operations that have multiple outputs, again in the interests of efficiency – and also reversibility.

Consider next a program involving four actions, α, α', β, and β', and suppose that proper execution of α' depends on the outcome of α, whereas β' requires the completion of both α and β. We would like to be able to specify those ordering constraints, at the same time not imposing an execution order between α and β or between α' and β'. We need to be able to specify α before α' together with α and β before β'.

It may be the case that there are alternative ways, α and α', of achieving one and the same goal. When resources allow, one might want to try both in parallel, until one of them succeeds, say α, before proceeding to the next stage β. Or one may have multiple processors at one's disposal and wish to allocate resources for speculative lookahead. We want to express these and other kinds of choice in programs with an operational semantics that allow for partial executions of alternatives. ASMs with bounded nondeterminism have already been studied [7].

In the sections that follow, we suggest a graph semantics with read and write vertices and with control-flow and data-flow edges.

2 Flow of Control

A computation transforms a given initial state. But what exactly transpires during such a computation? And how can we describe that?

We begin by considering computations, whether or not they are algorithmic – in the sense of being finitely describable. Later, we will explore how one might specify algorithmic computations by means of graphical programs.

We contend that a *computation* involves a quasi-ordering of *actions* on a given initial state, expressing the *control* order in which those actions transpire. Actions are said to be *ordered* if one strictly precedes the other in the quasi-ordering. Actions are *unordered* if they are either equivalent in the quasi-ordering or incomparable.

For control (quasi-) ordering $\precsim$, action α *precedes* action β and β *succeeds* α, written $\alpha \prec \beta$ or $\beta \succ \alpha$, if $\alpha \precsim \beta$ but $\alpha \not\succsim \beta$. They are equivalent ($\alpha \simeq \beta$) when both $\alpha \precsim \beta$ and $\alpha \succsim \beta$, and incomparable ($\alpha \# \beta$) when neither $\alpha \precsim \beta$ nor $\alpha \succsim \beta$.

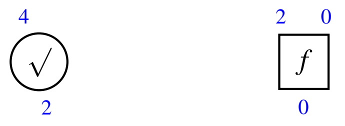
Equivalent actions occur at the same time – understood literally or figuratively; the execution order of incomparable actions is unknown or unknowable.

A step in a computation may also be thought of as a finite or infinite directed graph of action vertices – the *control graph*. We draw an edge $\alpha \dashrightarrow \beta$ from action α to action β if β is an *immediate* successor of α. The associated quasi-ordering $\succsim$ is reachability, the transitive closure of $\dashrightarrow$. There are no loops in the graph, other than to indicate simultaneous actions.

As Alan Turing observed in the pencil and paper domain, there are only two kinds of basic actions of importance in computations: reads and writes.

- *Read*: Query and retrieve a value from the state, or – put another way – apply some operation known to the state to some arguments.
- *Write*: Request and store a value within the state, or – viewed alternatively – update the way future queries are to be answered.

We will draw round read vertices and square writes, labeled with the relevant operation, and indicate the operands by means of additional side annotations. For example, here is a read of the square-root of 4 and a write of 0 to $f(2)$:



Think of f as an array, if you like.

Actions act on states. Reads access values known to the state. Writes modify them. An action only proceeds after it receives control from all immediate predecessors, much like simple Petri nets [13].

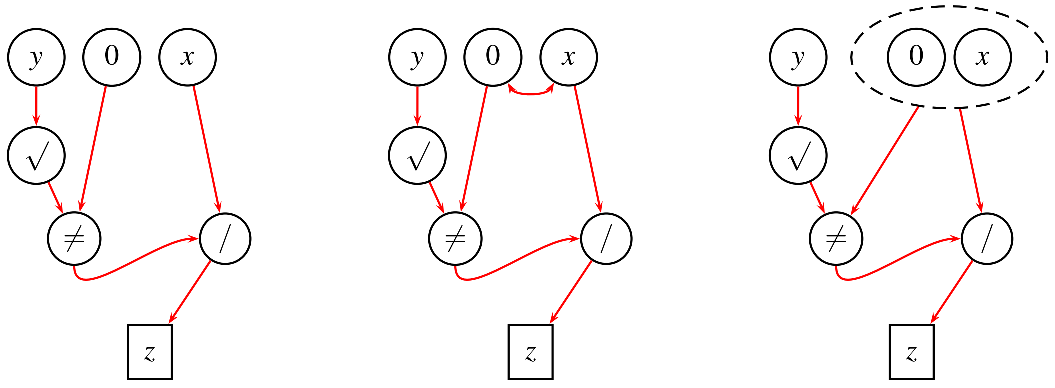
The salient aspects of computational states, in the ASM framework, are captured by (first-order) logical structures. Let Σ be the vocabulary of states and U, their domain. States interpret each of the symbols f in Σ as an operation $[\![f]\!]$ over U. The value $[\![f]\!](\bar{a}) = [\![f(\bar{a})]\!] \in U$ of $f \in \Sigma$ at $\bar{a} \in U^*$, as interpreted by a state σ, resides at *location* $f(\bar{a})$.

Reads are labeled by a function symbol f, take values $\bar{a}$ for the coördinates of the location being explored, and produce a value $b = [\![f(\bar{a})]\!]$. We may annotate the vertex fully by labeling it $f(\bar{a}) : b$. It may happen that no value is ever produced despite the query, in which case we could write $f(\bar{a}) : \bot$ with the understanding that the "undefined value" $\bot$ is never actually obtained or transmitted. The action is a dead end, successorless. We defer analysis of this happenstance.

Writes are also labeled by a function symbol f, take values $\bar{a}$ for the coördinates of the location being explored, plus a value $b \in U$ to be assigned, and change the value of f at $\bar{a}$ in the current state to be b, whenceforth $[\![f(\bar{a})]\!] = b$. We label the vertex $f(\bar{a}) := b$. It may also happen that this requested action never completes, leaving $[\![f(\bar{a})]\!] = \bot$, so to speak. We could indicate this by labeling the write likewise $f(\bar{a}) := \bot$.

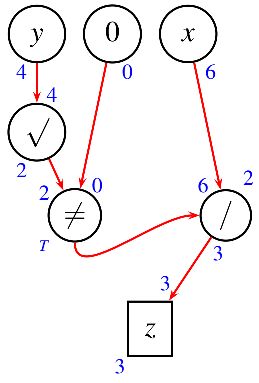
The action vertex on the left in the previous illustration would be labeled $\sqrt{4} : 2$, and the one on the right, $f(2) := 0$.

Flow of control is indicated in the graph via (red) directed edges (arrows) between action vertices:



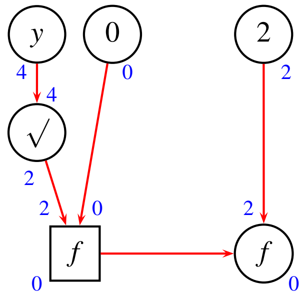
In the left case, the order in which the scalar x and constant 0 are accessed is unspecified; in the middle, the two actions happen simultaneously. On the right, equivalent actions are encircled, making the order easier to comprehend. In all, the computation of square root precedes the check for zeroness, which precedes the reading of the outcome of division, which – in turn – precedes the write to z.

Adorning the vertices of the previous graph (on the left) with a collection of appropriate values, assuming that initially $[\![x]\!] = 6$ and $[\![y]\!] = 4$ and that arithmetic constants and operations are interpreted as usual, we get this more complete picture of the goings on:



The numbers above a vertex are its inputs; what's below is the result of applying the indicated operation or (redundantly) of making the indicated assignment. For example, the division $6/2$ produces 3, which is stored in z for future use.

Writes in the ASM framework can be to arbitrary locations, not just to scalars. For example, the left half of the following is the graph for $f(\sqrt{y}) := 0$:



The graph of the function f at the point $\sqrt{4} = 2$ is assigned 0, as indicated by the 0 placed below the f, which henceforth is the value of $f(2)$, as portrayed in the right half.

3 Axioms of Action

Some axioms regarding computations are in order. First of all, reading and writing should behave as intended:

Axiom I (No Forgetting)

a. *Ordered reads of the same location must produce the same value, unless there is a write to that location between them.*

b. *After a write, all reads of that location must result in the value that was written, unless another write intervenes between them.*

For this axiom to be viable, we need another to avoid ambiguities and impose consistency in the presence of concurrency:

Axiom II (No Confusion)

a. *Unordered reads of the same location must retrieve the same value.*

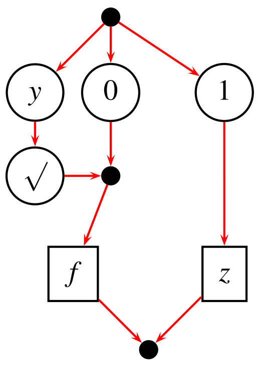
b. *Unordered writes to the same location must store the same value.*

c. *If a read and write for the same location are unordered, the value retrieved by the read must be the same as the value stored by the write.*

Every control graph has *root* vertices with no incoming edges and *sinks* with no outgoing ones. Roots reading from the same location, in particular, must agree on the values stored in the starting state. Likewise, all sinks should agree on the ending state of affairs.

It is convenient to add the possibility of a *junction* action • that does nothing more than pass control to all outgoing edges after receiving control from all incoming edges. In essence, the junction is shorthand for the full bipartite graph from all incoming neighbors to all outgoing one. With this feature, we can make life easy by insisting on a single junction for the root whence all action starts, with control edges going from it to all the vertices that would otherwise have been roots. Similarly, we can focus on a single sink signaling the end of the computation.

An example of the use of junctions would be this:



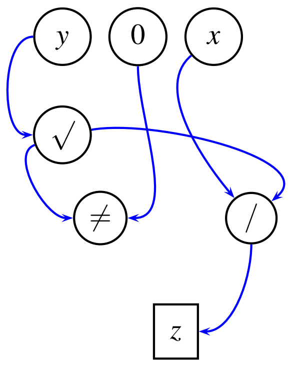
The assignment to f awaits completion of both reads, 0 and $\surd$, at which point control passes onward through the junction to the write.

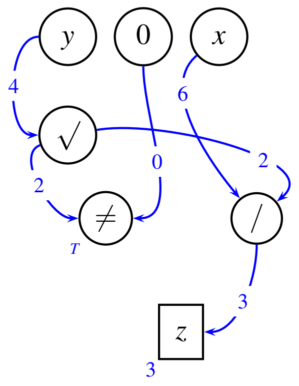
4 Flow of Values

The arrows of control graphs only indicate the order in which read and write actions are performed. But each of the values used by those reads and writes must come from somewhere – not pulled magically out of a hat. To cope with this requirement, we need to add another layer of edges to the graph, drawn from the producer of a value to each of its uses. If action α produces a value and action β uses it, then we draw an edge $\alpha \rightsquigarrow \beta$ (in blue). These edges form a dag, as they must be acyclic. We will refer to these data-edges as *channels*; they represent conduits for the transmission of produced values for subsequent use. The resultant structure is what we have – in a related context – called a *drag* [3], a labeled directed multi-root multiple-edge graph. A mix of control and value edges is found in the operational semantics of the Vienna Design Language (VDL) [11].

Reads $f(\bar{a}) : b$ have incoming channels for each of the components of $\bar{a}$ and optional outgoing channels carrying b. Scalar (zeroary) reads have no inputs. Writes $f(\bar{a}) := b$ have incoming channels for the $\bar{a}$ and also for b, but no outgoing channels. Incoming channels to reads and writes are fixed in number and ordered; outgoing channels can be manifold and are not ordered (as we are dealing, for now, with the single-output case). When a read produces a value that is not used, it would make matters uniform to send it to a terminal junction (with no outlets).

Here are the same vertices as before, embellished with blue channels showing the flow of data from vertex to vertex:

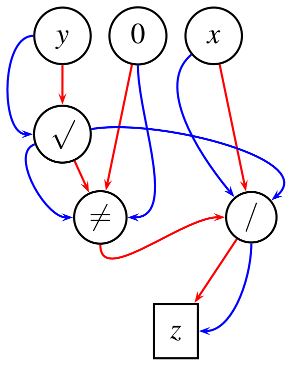


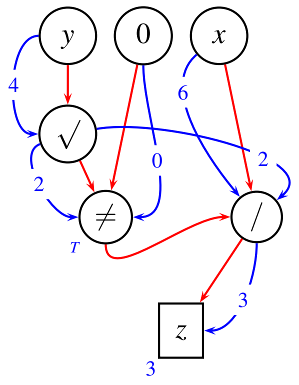


The square root is produced once but used twice. The graph on the right incorporates the annotations showing the traversing values, in a particular case, too. The channel label is the value at its two ends, which must agree for the edge to be sensible. A graph containing the combination of control edges and data channels between vertices, but without values, is what we will refer to as a *scheme*.

When a schema is annotated consistently with incoming and outgoing values at all vertices, we obtain a graph detailing a specific *step*. As we will see, the algorithm determines the scheme. Combined with a starting state, the schema determines the step.

In our running example, the schema and step look like this:





The value 2 from the square-root is shared by the comparison test and the division; the division awaits the outcome of the comparison, so has an incoming red edge from the disequality operation. The result, T, of the test is not transmitted anywhere.

Channels must obey a natural flow condition, precluding magic.

Axiom III (No Magic)

a. *A channel must have matching values at its two ends: the value produced by the read vertex at the tail end of the channel must equal the value used by the vertex at its head (which can be a read or write).*

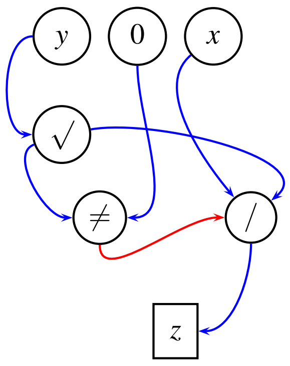
b. *The two ends of a channel must be (strictly) ordered: the producing vertex preceding the using side in the control order.*

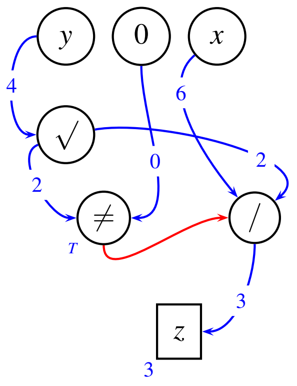
c. *All outgoing channels of a vertex must carry the same value.*

In the previous example, the square root produces the value 2, which is used twice, once by its direct successor and again by its grandchild, the division.

If a vertex of a graph has no incoming control edges, then, by Axiom IIIb, it also cannot have incoming values. Moreover, since there must be a (red) control path (consisting of one or more edges) wherever there is a (blue) channel, it will be frugal to combine data and control and omit the control edge when both are present.

In both the schema (below left) and step (right) we use solid blue for double-duty control-cum-data edges and red for control-only edges:

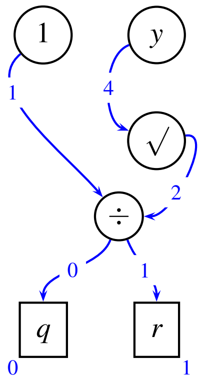




In the graph of a step, values that are produced but not used are also indicated as an integral aspect of the behavior of actions. For convenience, values set by assignments are also indicated.

For some applications, it is imperative to incorporate actions that produce multiple values, rather than perform two separate – perhaps costly – operations.

We illustrate this with an example of integer division with remainder:



Now outgoing channels are also ordered, one for the quotient and another for the remainder. They may each sprout more than one arrowhead for use by more than one subsequent action.

The structures used for states with such operations will also need to be designed accordingly. The preceding axiom also needs minor adjustments to allow for multiple output values. In the absence of a convenient syntax for operations with multiple outputs (cf., however, [9]), one can make do with a tuple, as in $q,r := 1 \div \sqrt{y}$.

5 Graph Semantics

A schema π is *applied* to a state σ by annotating each read of a location ℓ with the value stored at ℓ in σ or with the value assigned by the latest write to ℓ that precedes the read. One begins by annotating the reads of scalars that are roots of the schema with their values in σ, and then just works one's way along the channels to annotate all subsequent reads and writes in a straightforward manner.

The behavior described by π when applied to σ is just the (partially) ordered sequence of reads and writes contained in the resultant graph, each location accessed exactly as often as it appears in the graph. This annotated graph π^σ is what we referred to as a step, because it represents the behavior of a step of computation, starting in state σ and acting in accordance with schema π.

The net effect of applying a finite schema to a state σ is to change the values of its locations as per the write actions appearing in the graph, leading to a *next* state, σ'.

Let the vocabulary of the state be Σ and its domain U. We will assume a set $D \subseteq U$ of *designated* domain elements, such as a truth value T (standing for "true"). These will be needed for conditionals.

It is convenient to view a state σ as a set of location-value pairs $f(\bar{a}) \mapsto b$, for each $f \in \Sigma$ and $\bar{a} \in U^*$ (the length of the vector matching the arity of f) and $b \in U$. The value $[\![f(\bar{a})]\!]$ of location $f(\bar{a})$ in σ is that $b \in U$ such that $f(\bar{a}) \mapsto b \in \sigma$.

Let ρ be any set of location-value pairs, which we will refer to as a *region*. States are regions, of course. We define an *update* operator $\boxplus$ on states or regions that replaces the current values of selected locations in region ρ with new ones υ, as follows:

$$\rho \boxplus \upsilon = \rho \setminus \{f(\bar{a}) \mapsto d \in \rho : \exists d' \in U.\, f(\bar{a}) \mapsto d' \in \gamma\} \cup \upsilon$$

Let υ be the annotations of all the writes in schema π, excluding writes to a location for which there is also a subsequent write. Then the state $\sigma' = \pi(\sigma)$ obtained by applying π to σ is $\sigma \boxplus \upsilon$, as just described.

For lack of space, we omit the details of how precisely to gather υ, but point out that the No Confusion axiom then allows one to combine updates from parallel paths sans worry.

If, on the other hand, the instantiated schema shows unordered actions that violate the No Confusion axiom, then for the computation to obey the axiom one can add control edges (whether according to some policy or not) to force an ordering between the actions and resolve any confusion.

A finite or infinite sequence of schemata $\pi_1, \pi_2, \ldots$ applied to a starting state σ_0 induces a sequence of states $\sigma_0, \sigma_1, \sigma_2, \ldots$, where $\sigma_i = \pi_i(\sigma_{i-1})$ for $i = 1, 2, \ldots$. It also defines a sequence of steps $\gamma_1, \gamma_2, \ldots$ by instantiating the schemata $\gamma_i = \pi_{i-1}^{\sigma_i}$ for $i = 1, 2, \ldots$. We refer to the sequence of states as a *history* and to the sequence of steps as a *run*. The latter is much more informative.

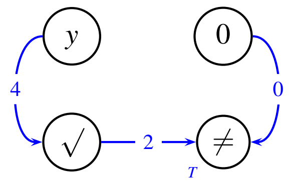
6 Simple Statements

Given the description of an algorithm or a non-algorithmic procedure plus a starting state, we would want to be able to construct the corresponding computation. So, how shall we express algorithms and procedures? What better way than graphs that reflect all possible computations?

The simplest algorithmic case to consider is the evaluation of a term t in some state σ. This is a universal aspect of programming. Evaluating, for instance, $t = f(g(h(a), c), c)$ in a call-by-value semantics, involves reading (accessing) the value of a before that of $h(a)$; $h(a)$ and c before $g(h(a), c)$; and $g(h(a), c)$, as well as c again, before applying f to obtain the value of the whole term t.

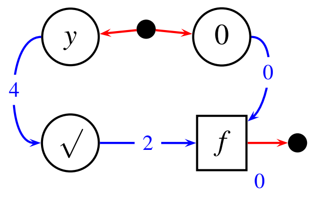
The control graph for such an evaluation has the shape of the tree representation of the term; all its nodes are reads. The node corresponding to a subterm $f(\bar{s})$ is annotated $f([\![s_1]\!],\ldots,[\![s_*]\!]) : [\![f(\bar{s})]\!]$. Every control edge is also a channel carrying the value that is read at the tail end.

The graph of the step that evaluates $\sqrt{y} \neq 0$, with $[\![y]\!] = 4$ in σ, looks like this:



The graph for an assignment $f(\bar{s}) := t$ has a similar form, with subgraphs for the evaluations of the s_i and of t, plus one write vertex at the root on the bottom, annotated $f([\![s_1]\!],\ldots,[\![s_*]\!]) := [\![t]\!]$. This affects the state for all subsequent actions.

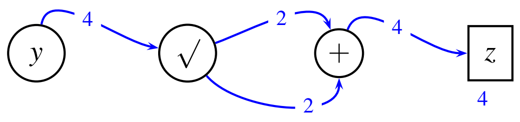
For example, the following is the graph for $f(\sqrt{y}) := 0$ in the same state as before:



The write is annotated $f(2) := 0$, as indicated by the 0 placed below the f, which henceforth is the value of f at 2. We have also added start and end junctions for convenience.

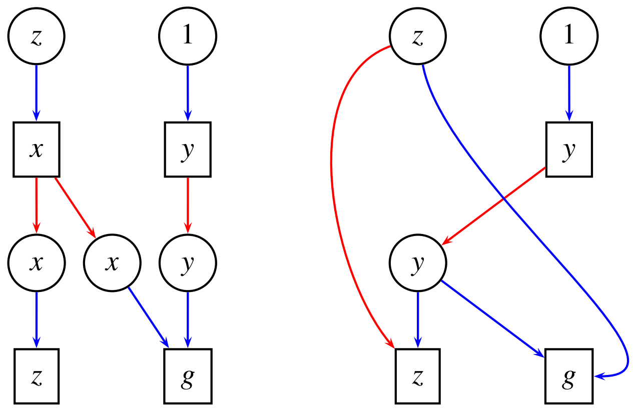
One can allow channels to join at a junction, but there can only be outgoing channels if there is exactly one incoming channel, so as not to mess up the fixed and ordered incoming channels at action vertices.

Significantly, a subcomputation may be shared by using the value it produces in more than one place. For example, the textual program **let** $t = \sqrt{y}$ **in** $z := t + t$ corresponds to the following algorithmic step:



Viewed this way, t is not a temporary value held somewhere in the current state; rather, it is a means of referring to connections between operations and their arguments.

Moreover, one can use channels to bypass intervening or potentially conflicting assignments. Sometimes dependencies are hard to express in usual program syntax, but are captured nicely by graphs, as in the following:



On the left, we have two sequences of assignments that may take place in parallel, $x := z\ ;\ z := x$ and $y := 1\ ;\ g(x) := y$, except that the last must also follow the first. The assignment $z := y$ in the right example can proceed before $g(y) := z$ because the value of z is already on its way to the latter via the indicated channel.

In a textual setting, provision needs be made for expressing such unstructured control-order dependencies.

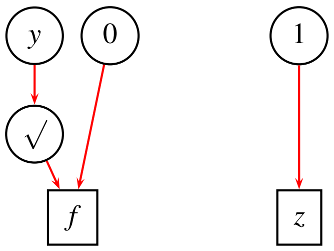
7 Composing Computations

There are two structured forms of composing computation steps: in parallel or serially.

Consider the parallel version first. Parallel composition $\gamma \parallel \gamma'$ of steps γ and γ' is simply their disjoint union $\gamma \uplus \gamma'$, with the onerous caveat that no conflict is created, so that the No Confusion axiom is maintained. In particular, all reads and all writes to each particular location ℓ must agree on that location's value.

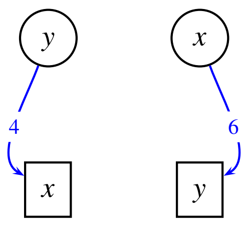
Likewise, if subprograms operate independently and in parallel, then their combined graph is just the (disjoint) union of the graphs of the subprograms.

The parallel composition $f(\sqrt{y}) := 0 \parallel z := 1$ is simply

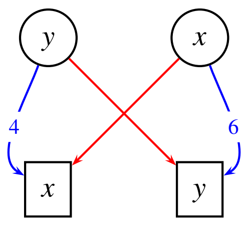


The constructs we have considered so far give us more than is needed to handle the steps of an ordinary sequential ASMs, except that their parallel assignments are meant to behave differently than disjoint union.

Consider the parallel assignment $x := y \parallel y := x$, intended to swap the values of x and y held in the state, and suppose that initially $[\![y]\!] = 4$ and $[\![x]\!] = 6$. The graph



does not impose an order on the reads and writes of each of the two variables, so it violates No Confusion (IIc). Were the read of x on the top right to occur after the write to x at the lower left, one would expect the former to produce 4, not 6. On the other hand, the following imposes order and satisfies the axioms:



Both reads, of the two locations x and y, complete before either write, to x or y, transpires.

In general, then, after evaluating all the conditions of an ASM and determining which assignments need to be executed, the evaluation of all the terms in those assignments should precede the actual updates.

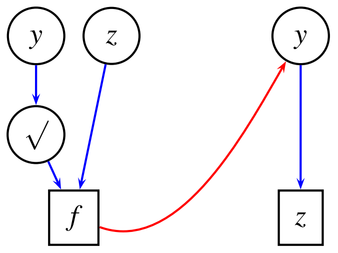
We have disallowed any possibility of parallel components with conflicting writes. So, the behavioral graph of the step needs to reflect the intended semantics, whether by forcing an ordering of the writes one after the other, or by making both set the value of the location to some specific "failure" value, or by hanging and never completing the action.

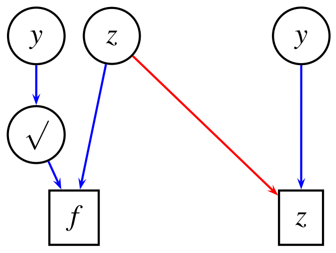
We are also interested in incorporating serial composition, as mentioned at the outset of this section. The serial composition $\gamma\,;\gamma'$ of γ and γ' requires some sewing. Complete sequentiality is easily obtained by appending start and end junctions to the graphs and having the start of γ' follow the end of γ.

Suppose p and q are programs, and $p\,;q$ is meant to indicate that execution of p should precede that of q. Then the graph of $p\,;q$ should include (red) control edges from each of the control sinks of p to all of the roots of q. Extra (blue) channels are not called for.

Or, we can take a minimalist approach and only impose as much order as is necessary. Every read or write action in γ to a particular location ℓ that is not itself followed by a subsequent read or write to ℓ is made to precede – in the control order – the earliest reads and write to ℓ in γ'.

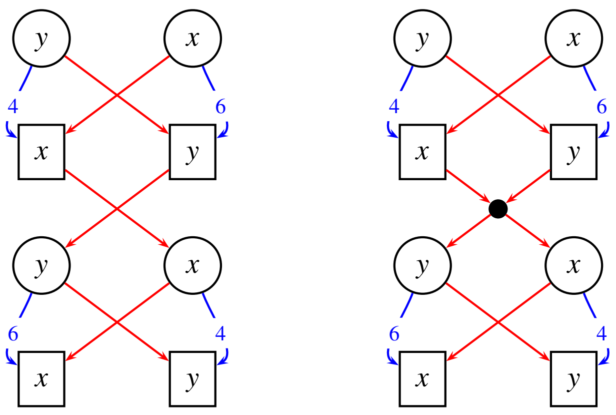
For example, the sequence $f(\sqrt{y}) := z\,;z := y$ induces the schema on the left in the complete sequencing approach and the one on the right in the more lax alternative:





Here we could not compose in parallel, since there could be a conflict over z.

Swapping x and y twice, that is $[x := y \parallel y := x]$; $[x := y \parallel y := x]$, corresponds to the following graphs, full sequentiality on the left and the weaker ordering constraints on the right:



In a structured language setting, control dependencies will form a series-parallel graph, the result of serial and parallel composition of components (junctions come in handy to maintain order), but in general arbitrary connections are plausible, such as those illustrated at the end of Section 6.

8 Making Choices

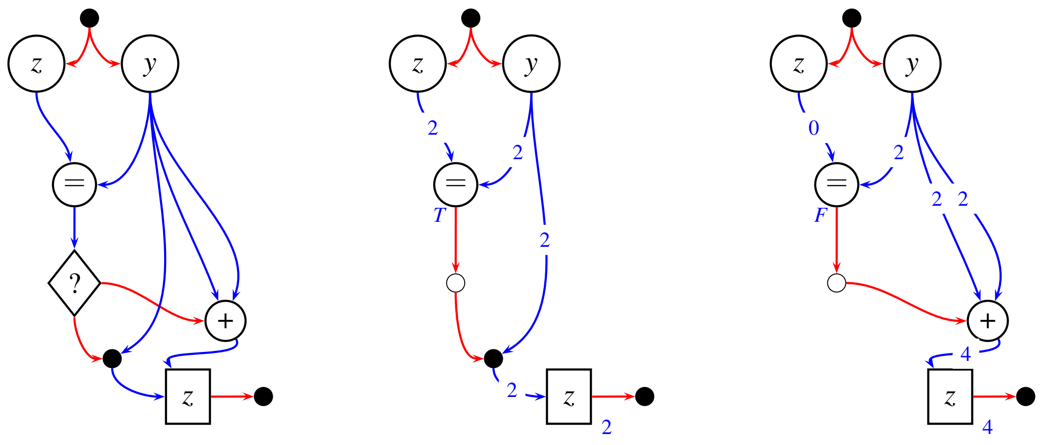
Real programs involve choices. The program schemas we have seen are choiceless, corresponding to straightline programs. More general programs or procedures involve tests leading to alternatives. We are also interested in modeling nondeterministic choice.

Consider first the case where a program segment is prefaced by a conditional. Then we must first of all include the graph of the evaluation of the conditional. Depending on the resultant value, the execution of a particular branch is included in the resultant computation. In any case, control passes from the conditional to the chosen branch.

The algorithm itself needs to include all alternatives, only one of which is taken at each stage of a computation. Let's use a diamond with a question mark for conditional choice. There should be control exits for each designated value, plus one for all other domain values.

Suppose T is the only designated value, and consider **let** $t = y$ **in if** $z = t$ **then** $z := t$ **else** $z := t + t$. There are two cases to consider, according as the test yields T or not. In each case, the relevant part of the algorithm forms the actual scheme. We need to instantiate the values, replace the conditional vertex with a junction and an arrow for the correct outcome.

The graph of this conditional statement – with a choice vertex – and the two possible schemata that can result (T and otherwise) are these:



The bottom exit from the test is for T. The vertices pointed to by the incorrect outcomes and all vertices that become inaccessible from the start vertex are erased, as they will never obtain control. The open circle junctions in the schemata replace the instantiated test. A junction node with an incoming channel will just pass a value on. We omit the details of what exactly gets erased or replaced; suffice it to say that decay spreads in the expected natural manner.

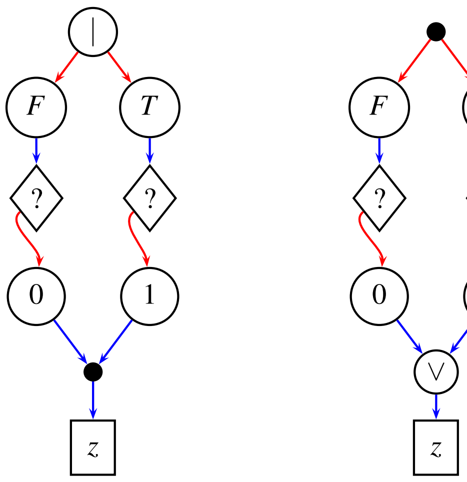
A multi-outcome **case** statement is handled in the same way; it is more general than the binary **if** and depends on more designated values than just T [1], but is otherwise the same. One could also make allowance for more than one type of conditional, each with its own set of designated values.

There are two additional, unconditional ways in which a choice of values r and s or choice of programs p and q can be made:

- In the description of algorithms, we can allow free choice vertices $\obar$, with the understanding that one branch or the other is chosen for execution, a priori, oblivious of the possible outcomes of either choice. Only the chosen branch contributes to the actual computation graph. This is *demonic* (in the mild sense of "don't know") choice. If either r or s can go on forever, then the composite computation might, too. As a textual program, one can allow for expressions with choice in the form of $r \mid s$, for terms r, s, as well as alternative programs $p \mid q$ for programs p, q involving assignments.
- In the alternate version of choice, using a vertex $\ovee$, the right choice – if any – is always taken. To implement such a choice, one can try both until at least one completes. In that case, the control graph includes one complete computation and another (perhaps partial) one – in parallel. If either terminates, then the combined computation does, too. Control, in this case, passes from the successful alternative to the continuation of the computation. If any interim results do not agree along the alternate paths, then provision must be made for the chosen path's effects to win out. This is *angelic* (don't care, erratic) behavior. It can be denoted $r \vee s$, for terms, boolean or not, as well as $p \vee q$ for programs.

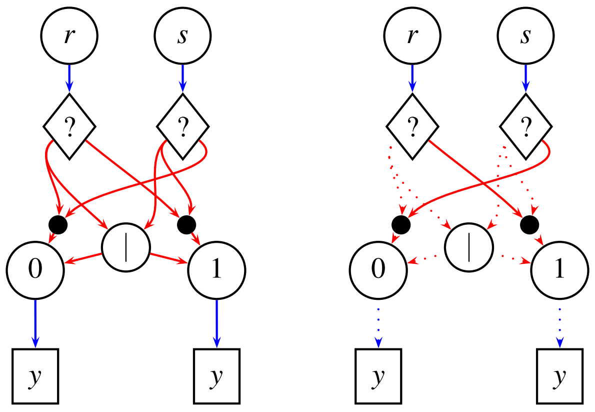
Both varieties of choice can be expressed in our framework. In the demonic case, noted by means of an $\obar$ vertex in the procedure description, one outgoing branch is chosen for the computation, regardless of what may happen as a consequence. In the angelic case, it may be that some branches are sterile, or it may be that different branches yield different outcomes. We model this by means of an $\ovee$ vertex with multiple incoming edges and/or channels. If control arrives via at least one, then control will pass onward. If multiple values arrive via channels, then exactly one is chosen to be passed on. (If part of the computation languishes, the exact form of the computation graph may depend on the vagaries of the execution mechanism, an eventuality we ignore here.)

As an example, consider an instruction $z := ([\textbf{if } F \textbf{ then } 0] \textit{ or } [\textbf{if } T \textbf{ then } 1])$. The demonic (on the left) and angelic (right) interpretations of *or* are as follows:

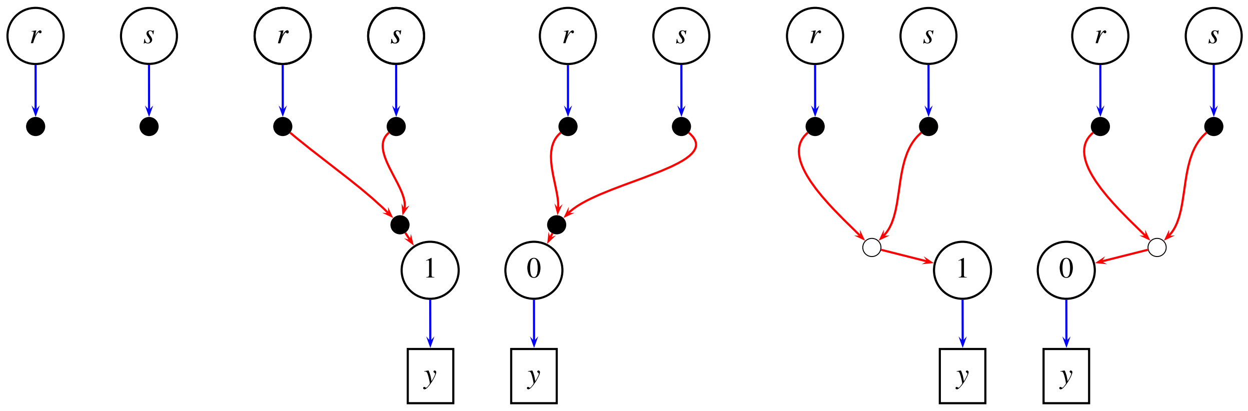


In the former, an execution may fail to proceed or may assign 1; in the latter, it will necessarily assign 1.

Dijkstra's guarded commands [4] require a mix of conditional, sequential, parallel, and demonic composition: first all the conditions are evaluated, then an oblivious choice is made among the enabled branches. The statement **if** $r \to y := 0 \,\square\, s \to y := 1$ **fi** translates into the procedure on the left:



No choice is needed when only one branch is enabled; a junction, acting as a conjunction, suffices. If neither r nor s is true, then only the solid control lines on the right are taken. The resulting schema for that case and the other four possible outcomes (two when both conditions are true and a demonic choice is made) are depicted below.



The open circle junction indicates where a choice was made.

9 Program Semantics

As we have seen, algorithms may be expressed in terms of similar diagrams to those embodying computations, with the addition of vertices for conditional choice, as well as for the various forms of nondeterministic choice. Whereas an algorithm describes the choices to be made, the behavioral graph describes the consequences of those choices.

For some given algorithm or procedure π, let Ξ be the set of its states (for some vocabulary and domain) and Γ the set of its computations (for the same). Both single-step behavior of an algorithm (we think of the algorithm as proceeding step by step, perhaps forever) and full-scale behavior of a potentially infinite procedure are embodied by graphical programs of the kind we have described.

We obtain intensional (operational) semantic functions and extensional (denotational) ones. Each comes in three flavors: local (single-step), global (all steps), and final (last step).

- The *next step* is the behavior $\pi^\sigma \in \Gamma$ of the step described by algorithm π given current state $\sigma \in \Xi$. The *next state* is the state $\pi(\sigma) \in \Xi$ obtained by taking that next step from the current state.
- A computational *run* is a finite or infinite sequence of computations (in Γ^∞), describing one step after the other, beginning with a start state. (If the sequence is finite, the last graph might be infinite.) A *history* is a finite or infinite sequence of states (in Ξ^∞), each being a next state of the prior one.
- The *computation* as a whole is the sequential composition of the computation steps in a run (in Γ). The *result* is the last state (in Ξ) of a run – if there is a last state.

If a program π is repeated over and over, finitely many times or infinitely many – as in the ASM model, then the full computation is just that of the sequential repetition $\pi\,;\pi\,;\pi\,;\cdots$.

Since our programs are nondeterministic, all the above semantic functions are (multivalued) mappings. As step semantics, an algorithm π defines a mapping from states to computation graphs, representing the behavior of the upcoming step, $\pi : \Xi \rightrightarrows \Gamma$. All the other semantic functions may be derived from this. Iterating, for example, gives the algorithm's semantics as a mapping from states to runs, $[\![\pi]\!] : \Xi \rightrightarrows \Gamma^\infty$.

10 Summary

The control order of a computation is dictated by two considerations:

a. Dataflow of values: values must be retrieved before they can be used.

b. Consistency of values: references to the same unchanged location should yield the same values.

The circuit-like diagrams we are promoting have the advantage of allowing maximal potential for sharing and parallelism. They provide a very precise operational semantics for a wide range of paradigms.

References

[1] Andreas Blass, Nachum Dershowitz & Yuri Gurevich (2010): *Exact Exploration and Hanging Algorithms*. In: *Proceedings of the 19th EACSL Annual Conferences on Computer Science Logic (Brno, Czech Republic)*, *Lecture Notes in Computer Science* 6247, Springer, Berlin, Germany, pp. 140–154, doi:`10.1007/978-3-642-15205-4_14`. Longer version at `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Partial.pdf`.

[2] Nachum Dershowitz (2012): *The Generic Model of Computation*. In: *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zürich, Switzerland)*, Electronic Proceedings in Theoretical Computer Science, pp. 59–71, doi:`10.4204/EPTCS.88.5`.

[3] Nachum Dershowitz & Jean-Pierre Jouannaud (2018): *Drags: A Simple Algebraic Framework For Graph Rewriting*. In: *Proceedings of the 10th International Workshop on Computing with Terms and Graphs (TERMGRAPH)*, Oxford, UK.

[4] Edsger W. Dijkstra (1975): *Guarded Commands, Non-determinacy and a Calculus for the Derivation of Programs*. *SIGPLAN Notices* 10(6), pp. 2–2.13, doi:`10.1145/800027.808417`. Available at `https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD418.PDF`.

[5] Yuri Gurevich (1995): *Evolving Algebras 1993: Lipari Guide*. In Egon Börger, editor: *Specification and Validation Methods*, Oxford University Press, Oxford, pp. 9–36. Available at `https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/103.pdf`.

[6] Yuri Gurevich (2000): *Sequential Abstract State Machines Capture Sequential Algorithms*. *ACM Transactions on Computational Logic* 1(1), pp. 77–111, doi:`10.1145/343369.343384`. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.3017&rep=rep1&type=pdf`.

[7] Yuri Gurevich & Tatiana Yavorskaya (2006): *On Bounded Exploration and Bounded Nondeterminism*. Technical Report MSR-TR-2006-07, Microsoft Research. Available at `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2006-07.pdf`.

[8] David Harel (1980): *On Folk Theorems*. *Communications of the ACM* 23(7), pp. 379–389, doi:`10.1145/358886.358892`.

[9] James Kajiya (2010): *Ramified Expressions – Expanding the Syntax of Expressions*. Unpublished report, Microsoft Research.

[10] Donald E. Knuth (1966): *Algorithm and Program: Information and Data (Letter to the Editor)*. *Communications of the ACM* 9(9), p. 654, doi:`10.1145/365813.858374`.

[11] Peter Lucas & Kurt Walk (1969): *On the Formal Description of PL/I*. *Annual Review in Automatic Programming* 6, pp. 105–182, doi:`https://doi.org/10.1016/0066-4138(69)90005-6`.

[12] Emil L. Post (1994): *Absolutely Unsolvable Problems and Relatively Undecidable Propositions: Account of an Anticipation*. In M. Davis, editor: *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, Birkhaüser, Boston, MA, pp. 375–441. Unpublished paper, 1941.

[13] Wolfgang Reisig (1985): *Petri Nets: An Introduction*. *EATCS Monographs on Theoretical Computer Science* 4, Springer-Verlag, Berlin, doi:`10.1007/978-3-642-69968-9`.