TStreams: A Model of Parallel Computation

(Preliminary Report)

Kathleen Knobe
Carl D. Offner

HP Cambridge Research Lab
{kath.knobe,carl.offner}@hp.com

July 25, 2005

Note

This is two chapters of what is really a much larger project. It is quite incomplete—for instance it completely lacks any references and comparisons with other work. As the title indicates, this is preliminary and will be superseded by a longer and more complete work.

We wanted nonetheless to make it available because it gives a cogent introduction to our work on TStreams.

Abstract

TStreams is a new language for describing parallel computations. It distinguishes the expression of the potential parallelism of the program from both

- the lower-level serial details of the algorithm, and
- the actual parallelism for a given target.

TStreams describes what to compute and when it can be computed. It represents all styles of parallelism (e.g., task parallelism, data parallelism, loop parallelism, pipeline parallelism) in a uniform way.

On the other hand it is not concerned with details of the actual computations or the data types. So TStreams is not a general-purpose language. You cannot add 2 and 2 in TStreams. TStreams has a single data structure, a single control structure and a single operation.

TStreams makes no assumptions about the target architecture (shared memory/distributed memory/grid/web service/streaming). On the other hand, TStreams maximizes the effectiveness of optimizing the program for a specific target by eliminating all inessential constraints.

Contents

Chapter 1

An Introduction to TStreams

1.1 Introduction

Writing parallel programs is difficult, and one of the reasons is certainly some limitations of the tools we are provided with. Looking at parallel languages in particular, we see that they have the following characteristics:

- In general, parallel languages specify parallelism by marking particular parts of a program as parallel. Any part not so marked is assumed to be not parallel. For example,
  - The OpenMP languages (OpenMP C and OpenMP Fortran) use constructs such as parallel do loops to identify parts of the program capable of being executed in parallel.
  - Fortran 2003 indicates parallelism through forall constructs and array assignments.
- The kind of parallelism expressed is specific to the language:
  - The OpenMP languages basically specify task parallelism and control parallelism.
  - High Performance Fortran leans heavily toward data parallelism.

  ... and no common language gives any support for expressing pipelined parallelism.
- Expressing parallelism often involves low level details:
  - MPI, which is the most popular form in which parallelism is expressed today, is a message-passing language, in which all sends and receives have to be written explicitly by the programmer.
  - The Open MP languages implicitly insert barriers, which often may be unnecessary. The programmer must be conscious of the situations in which these barriers may be explicitly inhibited.

TStreams is a new model of parallel computation. In contrast to the difficulties just raised, TStreams has the following properties:

- TStreams is optimistic, in the sense that everything in a TStreams program is assumed able to be executed in parallel unless it is explicitly constrained. And the only constraints are those imposed by the algorithm of the program itself—that is, they represent producer and consumer relations in the program.
- TStreams expresses all forms of parallelism equally well—it is not biased in favor of any particular form.
- Low-level details are explicitly not part of a TStreams description; they are handled elsewhere as needed.

We can view TStreams form in two different ways:

- TStreams is a form which can be specified directly.
- Alternately, TStreams form can be generated from ordinary serial code in any conventional language.

While we have some good ideas as to how TStreams form can be generated automatically, in this paper we will regard TStreams as a form which is specified directly.

A program in TStreams form can then be processed in two ways:

- It can be interpreted directly, or
- It can be compiled into code targeted to any parallel architecture.

This last point is quite important: TStreams carries no information and makes no assumptions about the architecture of the target machine. Its role is limited to that of specifying the maximum amount of parallelism in a program. This may seem like a very restrictive goal, but as we will see, it has some important advantages.

We will start out by showing how a simple computational problem can be looked at from the TStreams perspective. Along the way we will introduce the basic TStreams concepts and show how the TStreams model is useful.

1.2 A simple problem: finding patterns in chromosomes

1.2.1 Biological motivation for the problem

A chromosome is a single huge DNA molecule composed of two complementary linear sequences of nucleotides. One such nucleotide differs from another only in what is called its *base*. There are four bases: Adenine, Cytosine, Guanine, and Thymine. Because of this, we simply refer to the nucleotides themselves as bases and denote them by the letters A, C, G, and T.

The chromosomes carry the genetic information in a cell, in the form of genes. Each gene consists of a contiguous subsequence of nucleotides in a chromosome. Each gene determines the sequence of amino acids in a specific protein via the genetic code: Each of 61 of the 64 possible triples of consecutive bases codes for a particular amino acid, and the sequence of amino acids thus coded

for is assembled into a protein. (The remaining 3 possible triples indicate where the code for the protein stops.)

The two complementary strands of nucleotides that make up a chromosome match up, base by base, to form a sequence of base pairs. The bases A and T always pair together, and C and G always pair together. When a cell reproduces, these two sequences come apart and each one is used as a template to construct a new one which pairs with it. While this is of crucial importance for reproduction and inheritance, for our purpose here we are only concerned with the genetic information contained in the chromosome, and so we can simply consider it as a single strand of bases.

Human cells have 46 chromosomes, containing a total of about 6 billion base pairs.

Now actually, only 3% of the bases in a chromosome code for amino acids. The remaining 97% either serve other structural functions, such as determining under what conditions genes get expressed (i.e., get used to generate proteins), or are simply part of long sequences, often composed of repeating patterns, that so far as we know have no function at all and are referred to as junk. And it's not at all obvious, just given the sequence of bases in a chromosome, which of them constitute genes, or anything else for that matter. Even within a gene, there are stretches of base pairs, called introns, that don't code for anything and are ignored.

However, even if we could identify all the genes, we would still be far from understanding how they work: Each cell in an organism contains the same genetic information. Yet liver cells and nerve cells are quite different, and the reason is that different genes are expressed in those two cells. For a gene to be expressed, a special protein needs to bind to a nearby site on the chromosome. Such a site is called a promoter site. And there are also other binding sites, called repressor sites, that other proteins can bind to that can inhibit a gene from getting expressed. A lot of work is going into identifying these binding sites on chromosomes and understanding how they function.

Typically the sequence of bases (the "recognition sequence") that constitutes a binding site for a promoter or repressor is between 8 and 20 bases long. These binding sites often occur in clusters, and a good indication that such a sequence in a chromosome actually functions as a binding site is that it occurs in such a cluster. So in looking for binding sites, we look for clusters of these sequences that are close together.

1.2.2 The problem, and a program to solve it

Here then is an example of the kind of problem we will consider[1]: each chromosome is for our purposes just a string over the alphabet of the four bases $\{A, C, T, G\}$. A particular recognition sequence is simply a pattern such as

GGGwdwwwCCm

where

w	is	A or T
d	is	A, T, or G
m	is	C or A

[1]This is taken from Markstein et al., *Genome-wide analysis of clustered Dorsal binding sites identifies putative target genes in the Drosophila embryo*, Proceedings of the National Academy of Sciences 99 (763–768), 2002.

We want to find clusters of at least 4 sequences of this form within a stretch of 400 bases. We surmise that this indicates the presence of binding sites.

Our program will take as input the following data:

- The genome of the organism; that is, the sequence of base pairs in each of its chromosomes.
- The pattern (describing a recognition sequence) being sought. In the example above, the pattern is GGGwdwwwCCm.
- The maximum size of a contiguous subsequence of the genome in which we expect to find a cluster of matches. In the example above, this size is 400.
- The minimum number of matches needed to define a cluster. In the example above, this minimum number is 4.

The program performs its processing in two stages:

Stage 1. It finds all matches of the pattern anywhere on the genome and indexes them.

Stage 2. It identifies clusters of these matches.

We will cut up the representation of each chromosome into a family of contiguous blocks of base pairs. We do this to increase the opportunity for parallel processing. We look for matches in each block. The pattern length is quite small, as we have seen, and so by arranging matters so that the blocks overlap slightly we can be sure that each pattern match is contained within a block.

Using the information from the first stage, we look for clusters of these matches. A cluster may span two blocks, but in any case the maximum length of a cluster will be shorter than a block, so that it cannot span more than two blocks.

The output of the program is the set of clusters.

1.2.3 The TStreams way of specifying the program

TStreams models this program as a set of *steps* that constitute the computations and a set of *items* that constitute the data. Figure 1.1 indicates this high-level view.

1.2.3.1 Items

There are many blocks of chromosomes that are input to the program. Each such block is encapsulated in its own item. We manage this by tagging each item: Each item, in addition to its contents, has a *tag* that, together with the name of the item, serves to identify it uniquely. A `blockOfChromo` item, for instance, could have a tag that specifies which chromosome it comes from and which block on the chromosome it represents.

Tags are in some sense arbitrary, but since they are the way items are accessed, they will naturally reflect something about the data structures of the program. For instance, we might have a family of items representing the columns of an array. The tag of each of those items would then typically be the column number.

The set of tags that parametrize an item in this way is called a *tag space*.

In our program, there are two more items:

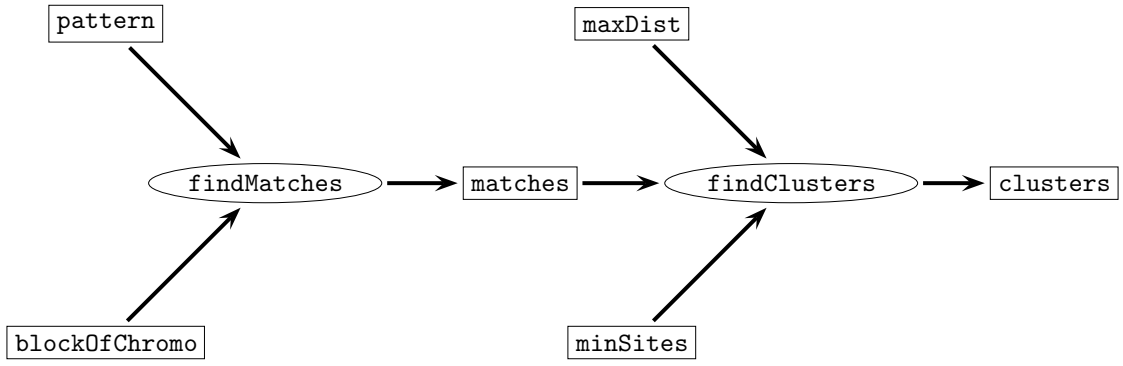

Figure 1.1: A TStreams program that finds clusters of matches of a given pattern in the genome of a given organism. Items (representing data) are in rectangles; steps (representing computations) are in ovals.

`matches` The contents of each such item is a list of the pattern matches found in the corresponding `blockOfChromo` item. There is one of these items for each `blockOfChromo` item. The `matches` items are parametrized by the same tag space as the `blockOfChromo` items.

`clusters` The contents of each such item is a list of the clusters found starting in the corresponding `blockOfChromo` item. There is one of these items for each `blockOfChromo` item. The `clusters` items also are parametrized by the same tag space as the `blockOfChromo` items.

Note that a `blockOfChromo` item, a `matches` item, and a `clusters` item may all share the same tag, but they are different items nonetheless, because their names (`blockOfChromo`, `matches`, and `clusters`, respectively) are different.

There are also some items that only have one instance. In our program, we have several of this form:

`pattern` This item contains the pattern for which matches are being sought.

`maxDist` This item contains the maximum size of a contiguous subsequence of the genome in which we expect to find a cluster of matches. In the example above, this size is 400.

`minSites` This item contains the minimum number of matches needed to define a cluster. In the example above, this minimum number is 4.

While in TStreams every item has a tag, a tag space of any item such as this just contains one tag.

1.2.3.2 Steps

From the TStreams point of view, a step is an atomic computation. Of course it may actually be quite complex internally, but TStreams views it as a black box. So for instance, the `findMatches`

step takes as input the `pattern` being matched and one `blockOfChromo` item and finds all matches of that pattern in that block. Of course a separate step of this form needs to execute for each `blockOfChromo` item. Therefore steps, just like items, also have tags. In this case the `findMatches` step can be parametrized by the same tag space as the `blockOfChromo` items.

The `findClusters` step may similarly be parametrized by the same tag space. However, there is a significant difference between the way the `findMatches` and `findClusters` steps execute. `findMatches` takes as input one `blockOfChromo` item. `findClusters`, however, takes as input *two* `matches` items: one corresponding to a `blockOfChromo` (the one with the same tag as the executing step), and one corresponding to the next `blockOfChromo` item on the same chromosome (assuming there is one). The reason for this is that while a pattern may not span two blocks, a cluster of matches might.

In this simple program, the tags of the steps were really the same as the tags of the items. This is not always the case. Since steps are computations, step tags naturally represent something about the computational structure of the program. For instance, if each step represents the computations in what would be one iteration of a loop in a serial program, the step tag would naturally be the value of the loop variable.

1.2.3.3 The program with tags

Figure 1.2 shows the first stage of the computation, with the tags of the items and the step made explicit. Of course the tag $\langle 7, 3\rangle$ is just illustrative; there is one `blockOfChromo` for each tag, and a corresponding step `findMatches` and item `matches` for each tag as well.

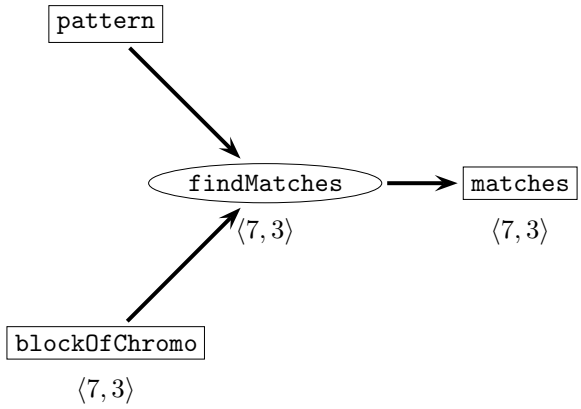

Figure 1.2: The first stage of the TStreams program that finds clusters of matches of a given pattern in the genome of a given organism. The tag $\langle 7, 3\rangle$ could be written out more fully as

tag: ⟨chromosome: 7, block: 3⟩

Figure 1.3 shows the second stage of the computation. Here it is clear that each instance of the step `findClusters` takes as input two separate `matches` items.

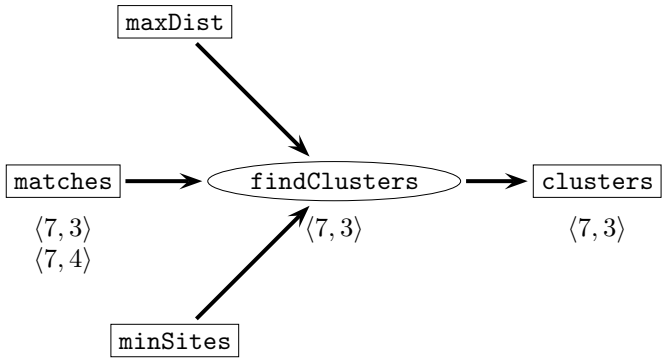

Figure 1.3: The second stage of the TStreams program that finds clusters of matches of a given pattern in the genome of a given organism. Note that each instance of the step `findClusters` takes as input two `matches` items.

Although in these figures the items `pattern`, `maxDist`, and `minSites` are not shown as having tags, in TStreams every item has a tag. Since there is only one of each of these items, the tag space corresponding to each of them contains only one tag, whose specific representation is irrelevant.

1.2.4 What does this buy us?

1.2.4.1 Minimal constraints on execution

Suppose we had just jumped right into writing our program in some standard programming language. We would probably have written a doubly-nested loop around the `findMatches` code (the outer loop running over the chromosomes and the inner loop running over the blocks in each chromosome). Then we would have similarly written a doubly-nested loop around the `findClusters` code.

These loops would specify an ordering of the execution of the loop iterations, and thereby would specify an order in which the data is produced.

Now in the TStreams representation, we see no loops at all. And in fact, there is no implied order of execution of the different `findMatches` steps. Further, the step `findClusters<7,3>` can execute as soon as the two items `matches<7,3>` and `matches<7,4>` have been produced, and this could be at any time. There is nothing in the TStreams representation that precludes these being the first two `matches` items produced, or the last two, for that matter.

In fact, the only constraints on execution ordering in the TStreams representation as shown in Figures 1.2 and 1.3 are those indicated explicitly by the arrows in those figures. Those arrows represent producer and consumer relations and are an essential part of the algorithm itself. But other than that, there are no constraints at all.

1.2.4.2 More parallelism: running the algorithm more than once

Now suppose we want to run the algorithm several times, each time looking for clusters of different patterns. In a conventional programming language, we would write a loop around all the code we have written so far, each iteration of the loop handling a separate pattern. So one might think that in the TStreams representation there would be some sort of loop back from the end of the computation at the right to the beginning on the left.

Actually, however, no such loop would appear, just as there is no loop in the TStreams representation around either of the two steps. In fact, there is very little change to the TStreams representation, and at the level of Figure 1.1 there is no change at all. The items, steps, and connections between them are all unchanged. The only change is to the tags:

- The `pattern` item is no longer a constant item. It now has a tag, whose value is perhaps an integer, identifying which of several patterns it is.
- The `blockOfChromo` item tags are unchanged, as are the `maxDist` and `minSites` item tags.
- The remaining items and steps—i.e., the `findMatches` and `findClusters` steps and the `matches` and `clusters` items—have an additional field in their tags which is the value of the corresponding `pattern` tag.

  So for instance, a tag for a `matches` item might look like this:

  $$\text{tag: } \langle \text{pattern: } 2, \text{ chromosome: } 7, \text{ block: } 3 \rangle$$

We now have a further opportunity for parallelism: there is no inherent reason why all the clusters for one pattern have to be found before the processing for the next pattern can start. Nevertheless, conventional coding would imply this, for two reasons:

1. The outer loop (around the whole computation, running over the different patterns) would usually be written as a serial loop. Perhaps this could be recognized as not necessarily serial, either by a programmer-written directive, or by careful compiler analysis. However, this could well run up against another problem:
2. Depending on how the program is written, the iterations of the outer loop might not in fact be independent. This is because the data structures that encode matches, for instance, on an iteration of the outer loop might be reused, and therefore overwritten, on a subsequent iteration of the loop. This would mean that, while the iterations of the outer loop might occur in any order, no two could execute simultaneously.

This problem cannot occur in the TStreams representation. Since `matches` items corresponding to different patterns (which thus correspond to different iterations of the outer loop) have different tags, they are different items, and cannot interfere with each other.

TStreams in fact has a pronounced functional flavor:

- Items are immutable: an item is conceptually produced only once, and once produced it cannot be changed. So if in an actual implementation an item is produced twice for some reason, the two versions of that item will be indistinguishable—they will share the same name, the same tag, and the same contents.

- Similarly, steps are conceptually executed only once. If in a particular implementation a step actually executes twice, the second invocation of the step will nevertheless take as input the same items as the first (since items are immutable) and produce the same output items. Steps are thus analogous to pure functions.

1.3 Tags as first-class objects; a second application

In the simple application presented in the previous section, the use of tags is very straightforward. In general, however, the use of tags can be much more complex. To support this, tags themselves need to have a more explicit representation. In this section we consider an example where this explicit representation becomes important. In the previous example, the use of tags was completely statically known. In this next example, however, the tags, items, and steps are dynamically determined. Such dynamic applications are very common and important.

1.3.1 n-body problems

The first great success of Newtonian mechanics was the derivation of Kepler's laws. In particular, given an inverse square law for gravitational attraction, the earth must travel around the sun in an ellipse with the sun at one focus. This was a formidable problem at the time. Now it is taught to first-year physics students. It is an example of a 2-body problem. In contrast, the 3-body problem is much more difficult, and only a few special cases have explicit solutions.

Astronomers, of course, are interested in modeling assemblages of many bodies—for instance, entire galaxies—and seeing how they may be expected to evolve in time. Such problems are called n-body problems. The assumption in all such problems is that n is a very large number.

Computational chemists are also interested in n-body problems. The atoms in a large molecule, such as a protein, exert forces on each other, and the shape of the protein is determined by those forces. So algorithms for solving n-body problems are quite significant.

In a typical n-body problem there are a large number of bodies. Each body acts on every other body by some computable force. The force, however, falls off rapidly with distance, and so a group of bodies at a distance from a particular body can be considered to act as a single larger body with little loss of precision but much savings in computational cost. In fact, this kind of simplification changes the cost from $O(n^2)$ to $O(n \log n)$ or even $O(n)$.

The way this kind of computation is typically structured is that first a hierarchical tiling of the bodies is produced so that each body can interact with different size tiles at different distances.

To show how this works, let us consider a 2-dimensional variant of the problem. Suppose we have a certain number of bodies distributed in some fashion in a large square, as in Figure 1.4.

We subdivide the large square recursively as follows: If a square contains more than 1 body, we divide it into four equal squares. Eventually we arrive at the situation in Figure 1.5. The squares can be regarded as nodes in a tree with up to 4-way branching, the root of the tree being the original large square containing all the bodies. By convention, the empty squares in Figure 1.5 are not actually in the tree of nodes, so each node in the tree has from 0 to 4 children.

Suppose we consider the body that is uppermost to the right. (See Figure 1.6.) Call this body A. The force on A from the two bodies near it to the left may be approximated by considering those

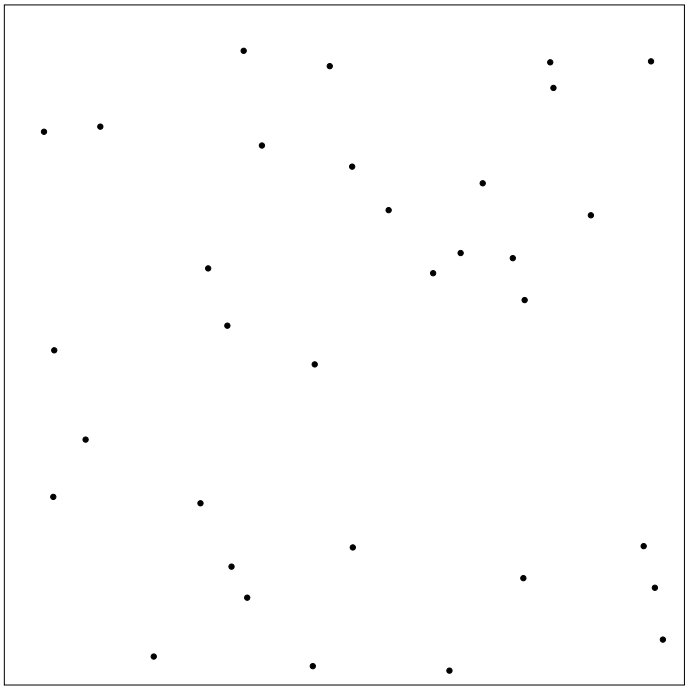

Figure 1.4: n-body problem

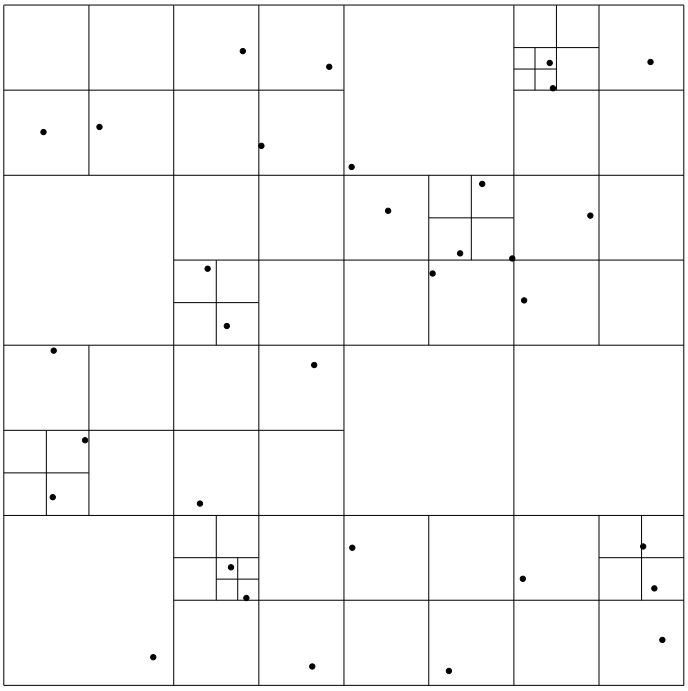

Figure 1.5: n-body problem, subdivided

two bodies to be a single body in the square containing both of them. The mass of that new body is the sum of the masses of the two bodies in the square, and the position of that new body is the center of mass of those two bodies. This works because the separation of those two bodies is small compared to their distances from A.

Similarly, the total force on A from the 5 bodies in the medium-sized square "south-west" of A may be approximated by amalgamating those bodies into one body in that larger square. And the total force on A from the 9 bodies in the south-western quadrant may similarly be approximated by the force due to a single body. We can continue in this way, replacing all the bodies in an entire square by a single body. The size of the squares becomes larger as the square is farther from A.

Of course when we compute the force on a body different from A, the particular squares we use will also differ, depending on the distance from that body. But as we build the tree of all squares, we can precompute a "summary body" which is associated with each square: the mass of that body is the total mass of the bodies in the square, and its position is the center of mass of those bodies. Then we use those summary bodies that are associated with the squares that are appropriate for each body in turn.

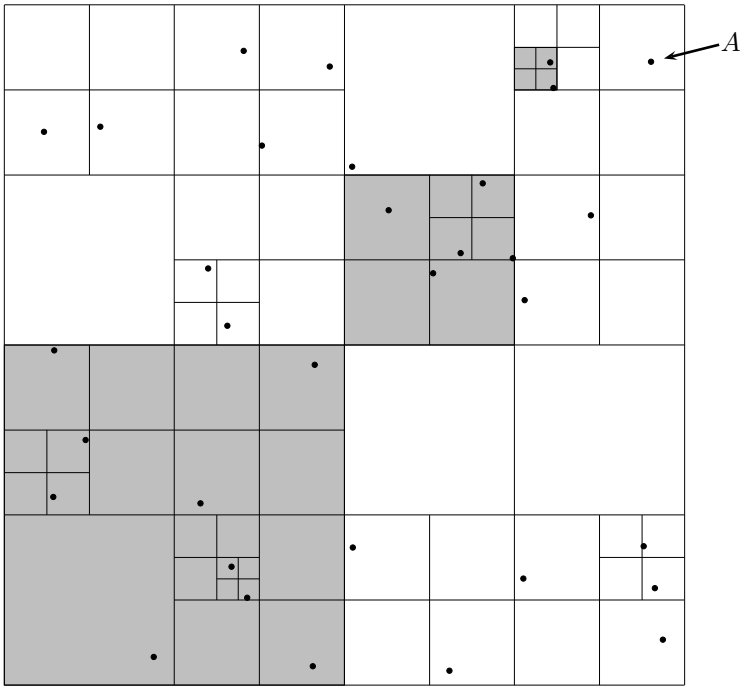

Figure 1.6: n-body problem: approximating distant groups of bodies

The approximation technique we have been describing is known as the Barnes-Hut algorithm. There are many variations that are possible. For instance, we may not subdivide squares containing only a few bodies. There are also alternative algorithms such as the Fast Multipole Algorithm, which do rather better. But for our purposes, they are all quite similar in that they recursively decompose the original set of n bodies and use this decomposition as the basis for approximating the forces on each body.

In actual practice, of course, the space, being 3-dimensional, is decomposed into cubes, not squares,

and each cube has between 0 and 8 children.

After a hierarchical tiling of the bodies is produced as above, the forces on each body are computed and the position of the body changes slightly based on these forces. After a certain number of iterations of this process, the positions of the bodies have changed sufficiently that a recomputation of the hierarchical tiling is called for.

Thus the computation of cubes happens over and over again as the algorithm proceeds. Let us look just at this part of the computation. Figure 1.7 shows how we can represent it as a TStreams program.

As before, steps are represented by ovals and items (of which there is only one in the figure) by rectangles. Now, however, we represent tag spaces explicitly, by triangles. (This is just a mnemonic: "triangle" and "tag space" both start with "t".)

The tag space of all the cubes (i.e., the root cube and those produced from it by the subdivision algorithm) is at the top of the figure. Since it parametrizes the set of cubes, there is a map, denoted by π, from that tag space to the space of cubes. This map is 1–1 and onto: every tag has a corresponding cube, and each cube has a unique and distinct tag.

Let us look at how this process evolves in time. We start out with one cube, the root. This cube of course has a tag, so there is one `cube` item and one tag in the upper tag space. Each cube has to be analyzed to see if it needs to be subdivided. That analysis is carried out by an `analyze_cube` step. Since there is one instance of that step for each cube, the `analyze_cube` steps are parametrized by the same tag space, and so we see a π map from that tag space to the `analyze_cube` step. The analysis step looks at the bodies in the cube to see if the cube contains more than 1 body. Therefore, the cube itself must be an input to the step, and that input is represented by the horizontal INPUT arrow.

If there is more than 1 body in the cube, the `analyze_cube` step returns a tag, which is the same as the tag that parametrizes it. We can think of this step as actually generating that tag and entering it in the bottom tag space, which is the tag space that specifies those cubes needing subdivision.

That tag space in turn parametrizes the `subdivide_cube` step, which takes as INPUT the same cube and generates between one and eight sub-cubes. Each cube that is generated in this way also needs to have its own tag. This tag is generated by the same step that produces the cube item.

The process continues in the same fashion.

The set of cubes is not known a priori; it needs to be computed. In fact, it needs to be computed over and over again. And the tag spaces, which we represent explicitly, make it possible for us to represent exactly how this happens.

1.3.2 What does this buy us?

As in the previous example, the TStreams representation specifies the minimal constraints on execution:

If we wrote a program in a conventional programming language to generate the tree of cubes, we would most likely decide on one of several methods of traversing the tree. For instance, we might build the tree in a depth-first manner, or in a breadth-first manner.

In contrast, the TStreams representation does not hard-code any specific way of building the tree into the program. The only constraints that it imposes are the unavoidable ones that a child cube

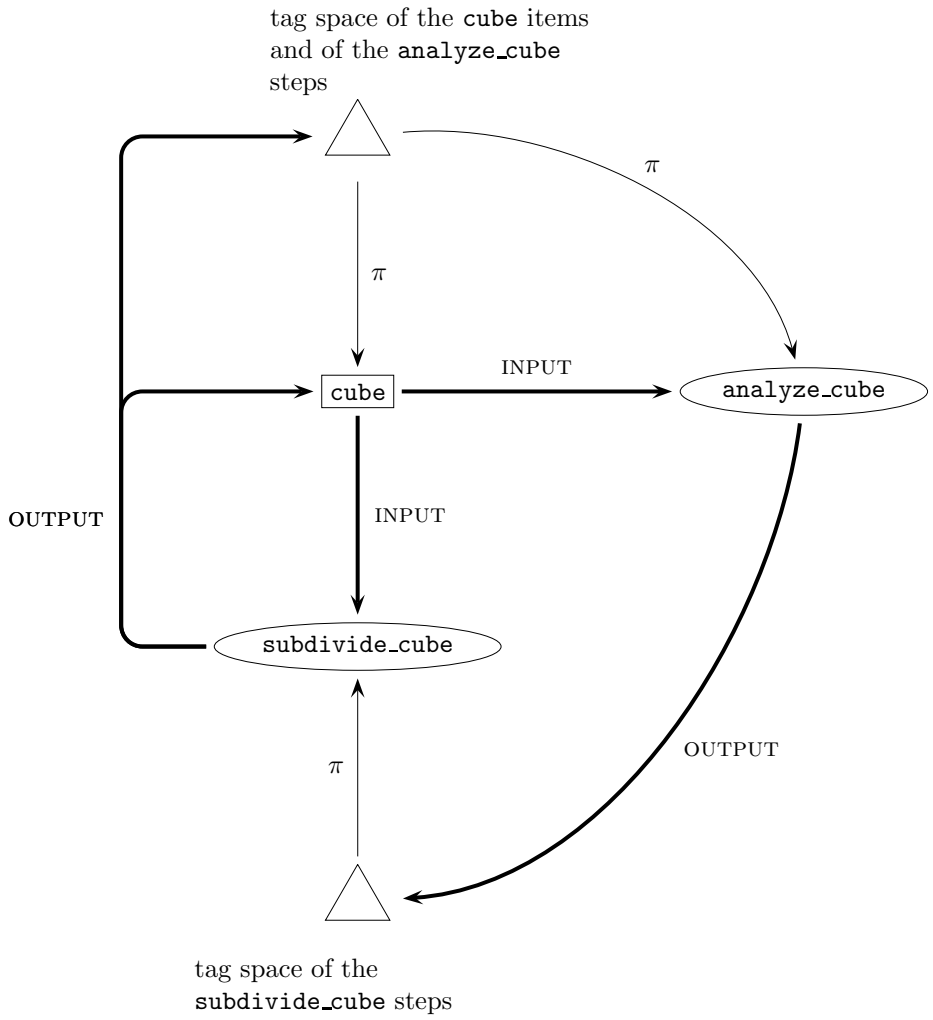

Figure 1.7: The subdivision process.

The thick arrows indicate flow of data. These arrows are labeled as INPUT to indicate that an item is input to a step or OUTPUT to indicate that an item or a tag is output from a step.

The thin arrows indicate mappings from tag spaces. Such mappings are by definition parametrizations, and are labeled as π.

can be produced only after its parent has been produced. This leaves the question of the precise order of production to be determined separately. Such a determination would take into account the nature of the target machine, the resources available, and so on.

The TStreams representation makes no assumption that the tree-building phase must complete before the next phase of the program starts. In fact, once a cube and all its descendants have been produced, that cube can be used by the next phase of the program, even if there are more cubes in the tree that have not yet been produced. So there are no implicit barriers that would inhibit a natural pipelining of program phases.

In these ways the TStreams representation allows for the possibility of more parallelism in the running program: the tree of cubes may be constructed in parallel in any of several ways, and the next phase of processing may begin before the tree is complete.

1.4 Tags and TStreams

1.4.1 The role of tags

Most languages contain control flow constructs such as linear code sequences, loops, if statements, case statements, and function invocations. These constructs handle the following two tasks:

- to describe *what* computations are executed. For instance, the execution of an *if* statement chooses between two different computations, only one of which is executed. The bounds of a loop determine how many times the body is executed, and the value of the loop index on each of those iterations.
- to describe *when* computations are executed. For instance, a loop specifies a linear ordering of the separate iterations of the code in its body.

Most languages also contain constructs to describe types of structures, to allocate new structures of a described type, and to associate a structure with a scope. These constructs handle two similar tasks:

- to describe *what* data objects exist.
- to describe *when* data objects come into existence. For instance, local variables in a procedure come into existence when the procedure starts execution. Other variables may simply be declared globally, in which case they come into existence at the start of the program. And variables may also be explicitly allocated, in which case they come into existence at the point of the allocation.

In TStreams, data is encapsulated in items and computations are encapsulated in steps. Tags are used to indicate what the steps are and what the items are. Thus tags handle the first of the three tasks (the *what exists* task) given above for both computations and data.[2]

However, tags by themselves do not address the *when* task. This task is managed by augmentations to the TStreams form. We do not go into these augmentations in this paper, but note only that they are an important part of the program. We will come back to this below.

Thus in the two examples we have seen so far, tags are used to specify what steps and items exist:

[2]In the language of parallel compiler technology, tags are thus closely related to both data and iteration spaces.

- In the chromosome pattern matching example, one tag space (containing a single tag) indicates the pattern and the values `maxDist` and `minSites`. The other tag space indicates the instances of the `findMatches` and `findClusters` steps, and the `blockOfChromo`, `matches`, and `clusters` items.
- In the n-body problem, one tag space indicates the instances of `cube` items and `analyze_cube` steps. The other indicates the instances of `subdivide_cube` steps.

However, as we have already noted, these tag spaces by themselves tell us nothing about the ordering of execution of the steps or which processors execute them, and where the items live, when they are produced, and their relations to each other.

1.4.2 The role of the producer and consumer relations

The producer and consumer relations, which are represented as arrows in the figures we have drawn, begin to address some of the questions above that are not addressed by the tags themselves: while they do not determine an ordering of computations, for instance, they do specify the minimal constraints which must be satisfied by any legal ordering.

- In the chromosome pattern matching example, a `matches` item must be produced before it can be processed by the `findClusters` step.
- In the n-body problem, a `cube` item must be produced before it can be processed by the `analyze_cube` step.

As we already pointed out, there are no loops in the TStreams graph for the first example. The multiplicity of steps and items is handled entirely by the tags. On the other hand, in the n-body problem, there is a loop in the TStreams graph. This loop represents the fact that data produced on one instance of a step is indirectly used by another instance of the same step. (That is, by another step with the same name, or equivalently, another step in the same step space.) Thus a loop in the TStreams graph represents, not multiple instances of steps or items—these are handled entirely by the tag structure—but a loop in the flow of data from one step instance to another in the same step space. There would also be a loop in a program in which the flow of data was indirect—in such a case more than one step space would be involved in the loop.

There were no loops in the flow of data in the first example, which is why there were no loops in the TStreams graph for that example.

1.4.3 TStreams and program development

In designing a program to implement an algorithm, there are three views that are important:

1. At the top level, we have knowledge of the algorithm itself, the computations that have to be performed, the data (including intermediate data) that has to be produced, and the relations between all these. At this level, we can see opportunities for parallelism. We are not concerned at this level with the computing resources that are available to us.
2. At the second level, we add to our knowledge of the first level by also taking into account the available computing resources. At this level, we make decisions regarding scheduling and mapping of computations and data.

3. At the bottom level, we need specific knowledge of the target machine and operating system in order to include in our program appropriate system calls and any other low-level machine-specific information.

The distinction made above in Section 1.4.1 between the tasks handled by tags and the tasks not handled by tags is just the distinction between levels 1 and 2 in this enumeration.

In addition, there is another important task that is handled by parallel languages at level 2: in addition to the *what* and *when* tasks, there is a *where* task, which, like the other two tasks pertains both to data and to computations:

- to describe where computations are executed. That is, we specify which processor executes each computation. Such specifications may be implied by the distribution of data over processors, or may be specified directly.
- to describe where data objects live. In parallel languages geared toward distributed-memory machines, such as High Performance Fortran, there are ways of specifying explicitly how data is laid out over the different processors.

A programmer building a parallel application today generally begins by asking questions such as

- What is the architecture of the machine the application will run on? Is it shared-memory or distributed-memory? Does it have many processors or just a few? How easy is it for processors to communicate?

The programmer will then ask questions such as

- Am I using a threads package? Will I use MPI? OpenMP? HPF?

The answers to questions such as these will have a great influence on the design of the resulting program.

These are all questions at levels 2 or 3. We believe they are the wrong questions to ask initially, because the answers freeze certain kinds of parallelism into the program and inhibit others.

We believe that the first thing a programmer needs to do is to consider the application at level 1—to represent it in a target-independent manner in which every constraint on potential parallelism is explicit. This is what TStreams enables us to do, and as we have seen, the resulting program description is quite rich. It exposes opportunities for parallelism that in many cases are just not expressible in conventional languages.

Based on this description, the programmer is then in a position to take into account all the available resources at level 2, and create a program tailored to a particular machine architecture.

Thus, in this method levels 1 and 2 are decoupled so that they can be dealt with separately. In doing this, we know that any decisions we make at the second level are constrained only by algorithmically necessary constraints and not by any peculiarities of how the program was written. Further, since the description at level 1 is independent of the target machine, any changes to the program that are necessary in retargeting to a different machine occur only at levels 2 and 3.

Chapter 2

The Semantics of TStreams

2.1 The TStreams kernel and its semantics

We are now going to specify more precisely exactly what we mean by a TStreams program. We will do this by providing a set of definitions of the fundamental TStreams objects together with a functional semantics.

The reason we can do this is that TStreams as we have described it so far is really functional. So in Figure 1.7, for example, while we have spoken of the step `analyze_cube` as producing a tag in the bottom tag space, from a functional perspective we would just regard it as mapping to a tag in that tag space. That is we look at the tag spaces, item spaces, and step spaces as being static rather than growing in time. The problem as stated then has a solution if such spaces all exist such that all the producer and consumer relations are satisfied. We explain this below in more detail in Section 2.1.2.

In this chapter we ignore questions of I/O, and more generally, of interactions with the external world.

2.1.1 The TStreams kernel

We first define what we call the *TStreams kernel*. The TStreams kernel is a language which is small enough that we can reason about it conveniently, while being rich enough to express anything that can be written in a more elaborate version of the TStreams language. The semantics of any TStreams program is defined in terms of an equivalent program written in the TStreams kernel.

A program written in the TStreams kernel consists of a specification of certain objects, relations between those objects, and boundary conditions:

objects

tag spaces There is a specified collection of tag spaces, each identified by a name.

Each tag space is a collection of tags. A tag holds a (usually simple) piece of data, called its *value*. Values of tags may be, for example, elements of $\mathbf{Z}$, or of $\mathbf{Z}^n$, or they may be keys in a database, or strings. Values of tags should be thought of as indices. They are used to parametrize steps and items.

Often we simply refer to a tag when we really mean to refer to the value of the tag. The context will always make our meaning clear.

While the values of tags may be used in computations inside step code (see below), from the point of view of TStreams, the only relevant property of tags is that their values must be able to be checked for equality.

Two different tags in the same tag space must have different values. In that sense, a tag really consists of nothing more than a name (that is, the name of its tag space) and a value.

While the tag spaces in a program must be specified, with the exception of initial tags (see below) the tags in a tag space are not specified.

item spaces There is a specified collection of item spaces, each identified by a name.

An item space is a collection of items. An item is an object holding some data, which is referred to as the *contents* of the item. In general, when we refer to an item, we are really referring to its contents.

In contrast to tags, two different items in the same item space may have the same contents. Such items are distinguishable by their tags (see below).

While the item spaces in a program must be specified, with the exception of initial items (see below), the items in an item space are not specified.

step spaces There is a collection of step spaces, each identified by a name.

A step space is a collection of steps. A step is a computation, represented by code. All steps in a step space have the identical code.

The step spaces in a program must be specified. In general, the steps in a step space, however, are not specified. The only exceptions are those steps that correspond to initial tags (see below).

relations

item space parametrization Each item space corresponds to a unique tag space. As we will see below, there is automatically a 1–1 correspondence between the items in an item space and the tags in its corresponding tag space. This correspondence is denoted by a parameter map π from the tag space to the item space.

No tag space can correspond to more than one item space.

step space parametrization Each step space corresponds to a unique tag space. Each step has a unique tag in its corresponding tag space. In the other direction, each possible tag value corresponds to a possible step. However, there is no a priori assumption that corresponding to each tag in the tag space there is a step in the step space.

Our intention, however, is that there be a 1–1 correspondence between the steps in an step space and the tags in its corresponding tag space. This correspondence, once it is proved to exist, is denoted by a parameter map π from the tag space to the step space. When we define the notion of a TStreams solution below, we will define it in such a way that this 1–1 correspondence must exist.

consumer relations Each step may consume zero or more items. The tags of the items consumed by a step are computable functions of the tag of the step. This relation between a step's tag and the tags of the items it consumes is called a *consumer* relation .

In addition to its local variables, the computation in a step has access to the following external data:

- the tag of the step.

- the data in the items which are consumed by the step.

and to no other data.

producer relations Each step may produce one or more items, tags, or both.

A step that produces an item also produces the tag of that item—that is, if the item is in the item space I and if T_I is the tag space that corresponds to I, then the step produces a tag in T_I and associates that tag with the item, in effect filling in an entry in the π map. In the other direction, a step that produces a tag in T_I also produces its corresponding item in I. Further, if a step produces more than one item in I, the corresponding tags in T_I must be distinct.

Other than producing items and tags, steps have no other visible effect. That is, steps have no side-effects.

Because of this, a step really has to produce something, otherwise it is useless. However, we do not make this a requirement.

The output of a step (that is, the tags and items produced) is determined solely by the tag of that step together with the contents of any items input to that step. We express this by saying that "steps are functional." This relation between the tag and inputs to a step and the items produced is called a *producer* relation.

products, or **derived tag spaces** A tag space may be designated as a *product* tag space. A product tag space (which we also call a *derived* tag space) is never an initial tag space. Tags in a derived tag space are not produced by any step. Rather, each derived tag space T_d is simply the Cartesian product of a finite number of other tag spaces.

We say that the tag spaces used to form the Cartesian product are the *component tag spaces*. The tags in those tag spaces are *component tags*. We say that the product tag space is derived from the component tag spaces, and that the tags in the product space are derived from the component tags.

boundary conditions

initial tags and items One or more tag spaces may be designated as composed of *initial* tags. Such tags are not produced by any step.

One or more item spaces may be designated as composed of *initial* items. Such items are not produced by any step. Each initial item is fully specified. (That is, its contents are specified.)

Every initial item space corresponds to an initial tag space. There is a 1–1 correspondence between the items in an initial item space and the tags in the corresponding initial tag space. As before, this correspondence is represented by a map π from the tag space to the item space.

There may be initial tag spaces that do not correspond to initial item spaces. However, they may not correspond to any other kind of item spaces: if an item space corresponds to an initial tag space, that item space is an initial item space. (It follows that an initial tag space not corresponding to an initial item space would appear in a program only because it corresponds to one or more step spaces.)

terminal item spaces and tag spaces One or more item spaces and/or tag spaces are specified as being *terminal* . Intuitively, finding the objects in these spaces is the goal of the entire computation. It may not be known a priori how many objects are in these terminal spaces, or indeed if there are any such objects at all.

An item space may not be both an initial item space and a terminal item space. A terminal tag space may not be either an initial tag space or a derived tag space.

Finally, the program must be *well-formed*. The precise definition of what it means to be well-formed is given below on page 28. We don't give it here because it is not syntactically specifiable, nor is it statically checkable. However, any ordinary program is virtually guaranteed to be well-formed.

Thus,

- There are three kinds of tags:

  initial tags

  derived tags

  computed tags These are the tags that are produced by steps. These include the terminal tags, which do not form a separate class in this context.

- There are two kinds of items:

  initial items

  computed items These are items produced by steps. These include the terminal items, which do not form a separate class in this context.

- There is only one kind of step.

2.1.2 The meaning of a program

We regard a program as a mapping from the initial items and the initial tags to the terminal items and terminal tags. This mapping is defined by a *solution* which intuitively consists of a family of items, tags, and steps satisfying the specified relations and boundary conditions.

Of course since we have not yet said yet exactly what the terminal items and tags are, the last boundary condition is not at this point well-specified. Nevertheless, this is the idea to keep in mind when we talk about a solution. The precise definition will appear below, after some preliminary work.

It is important to keep this terminology straight: a solution is not, as one might think, the family of terminal items and tags. It is the entire family of steps, items and tags that satisfies the relations and boundary conditions stated above. In exactly that sense, it is a solution to those relations and boundary conditions. The terminal items and tags in such a solution are what we are really interested in—they constitute the meaning of the program—but the entire solution itself is what we reason about.

We will define the notion of a solution more precisely below. In order to do this, we need to address the following questions:

- Does a solution exist?
- Can more than one solution exist?
- Can there be more than one meaning to the program?

It will turn out that there are in general several different reasonable families of items, tags, and steps that can be called solutions. Among these solutions, there is a unique maximal solution and a unique minimal one, and both of these solutions are significant. In any case, it is crucial that all solutions should agree on the terminal items and tags, in the sense that they agree on which terminal items and tags exist and on their contents (for items) and values (for tags). We will see that with our definition this is true. This makes the program an actual function from its initial items and tags to its terminal items and tags. So every program has a well-defined meaning.

There is a certain kind of pathology that can arise in a program as we have defined it above: there can be steps, items, and/or tags that might naively be thought to be part of a solution but which are *unreachable*. For instance, one might have, as in Figure 2.1 a step whose only input is an item that it itself produces, and which also produces its own tag. Intuitively, it is clear that such a step is unreachable from the initial items and tags, and so anything it produces is as well. Such unreachable objects are not inherently fatal. But it is clear that unreachable objects should not form any part of any actual solution. In our formulation, they will not.

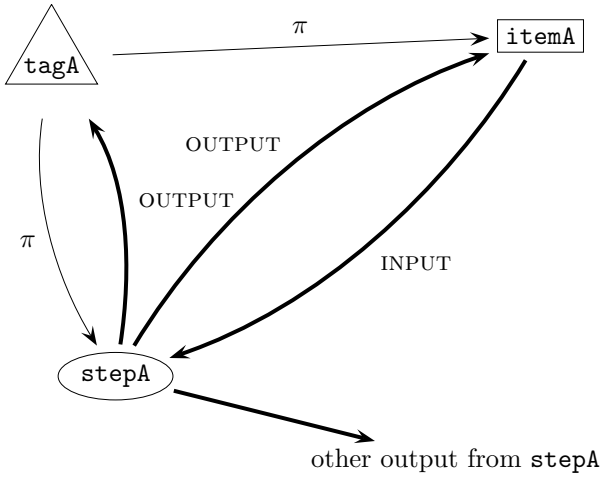

Figure 2.1: Unreachable nodes. The tag space `tagA` contains just one tag, which corresponds to the the single step in `stepA` and the single item in `itemA`. `stepA` has no other input.

Our first task then will be to characterize the reachable objects. This in turn will enable us to give a precise definition of all possible solutions.

2.1.3 Reachability and consistency: approximation from above

We first define what we will call a *pre-solution*. Intuitively, a pre-solution is a family $\langle T, I, S\rangle$ of tags, items and steps that is "closed backwards". For instance, if an item is in I then the step producing it must be in S. Before giving the precise definition, we need a preliminary definition:

Definition *A set T of tags is* closed under derivation *iff any tag derivable from tags in T is itself in T.*

Clearly there is a well-defined notion of the closure of a set of tags T in this sense; it can be constructed recursively, and since the number of derived tag spaces is finite and derivation increases dimensionality, this recursion must terminate. The closure can also be characterized as the intersection of all closed sets of tags that include T.

We use the notation $\overline{T}$ for the closure of T under derivation.

This operation of closure satisfies the following properties[1]:

1. $\overline{\varnothing} = \varnothing$
2. $T \subseteq \overline{T}$
3. $\overline{\overline{T}} = \overline{T}$
4. $T_1 \subseteq T_2 \implies \overline{T_1} \subseteq \overline{T_2}$
5. The intersection of an arbitrary family of closed sets is closed.
6. Since each derived tag space is produced from a finite number of tag spaces, the union of an increasing sequence of closed sets is closed.

Now we are ready to define what we mean by a pre-solution of a TStreams program:

Definition *A* pre-solution *consists of:*

1. *A set T of tags,*
2. *A set I of items, and*
3. *A set S of steps,*

such that

A0. 1. *All initial tags are in T.*
2. *All initial items are in I.*

A1. *Each item in I is either*

1. *an initial item, or*
2. *an item produced by a unique step in S.*
   Further, for each such item, its corresponding tag (produced by the same step) is in T.

A2. *Each tag in T is either*

[1] Kuratowski came up with four "axioms" which characterize the closure operator in topological spaces. The first three of these properties are three of the Kuratowski axioms. The fourth Kuratowski axiom

$$\overline{T_1 \cup T_2} = \overline{T_1} \cup \overline{T_2}$$

is not satisfied by our closure operator, and therefore our closure operator does not define a topology. This fact is irrelevant for our purposes here, however.

Properties 4 and 5 of our closure operator are automatically true for topological closure operators, but since ours is not one, we have to point out that these properties are true for our operator in any case.

Property 6 of our closure operator is not true for closure operators in general, but is true for this one.

1. *an initial tag, or*
2. *a tag (uniquely) derived from other tags in T, or*
3. *a tag produced by a unique step in S.*
   Further, if this tag is the tag of an item (which is necessarily produced by the same step), then that item is in I.

Further,

4. *T is closed under derivation.*
   (That is, we don't regard the act of derivation as "moving forward" in any sense. Derived tags exist automatically in a pre-solution if the tags they are derived from exist in that pre-solution.)

A3. *If a step is in S then*

1. *its tag is in T (precisely, there is a tag in T corresponding to that step), and*
2. *all its input items are in I.*

(The "**A**" prefix stands for "approximation from **A**bove".)

2.1 Theorem *If **i** is an item space and **t** is the tag space corresponding to it in a pre-solution, then there is a 1–1 correspondence (given by the parametrization map π) between the set of tags in **t** and items in **i**.*

PROOF. We may assume that neither **t** nor **i** is an initial tag or item space, since the statement of the theorem holds by definition in such a case.

We know that each item in **i** corresponds to a tag in **t**, since we have defined production relations in such a way that the step that produces an item also produces its tag.

In the other direction, we know similarly that a step producing a tag in **t** must produce an item with that tag, so there is a map in the other direction as well.

Two different items in **i** cannot correspond to the same tag, since either

- they are produced by the same step, in which case their tags must be different, or
- they are produced by two different steps, in which case those two steps would have to produce the same tag, which cannot happen in a pre-solution.

Similarly, two different tags in **t** cannot correspond to the same item, since either

- they are produced by the same step, which then must also be the step producing the item. But a step can't produce two tags for one item.
- they are produced by two distinct steps, each of which must produce the same item, which is also impossible in a pre-solution. □

Suppose that $\langle T_0, I_0, S_0\rangle$ and $\langle T_1, I_1, S_1\rangle$ are two pre-solutions. By the intersection of these two presolutions we mean the sets of tags, items, and steps that they have in common. To be precise, we say that

- A tag in T_0 is the same as a tag in T_1 iff they are in the same tag space and have the same value.
- An item in I_0 is the same as an item in I_1 iff they are in the same item space, they have the same tag, and they have the same contents.
- A step in S_0 is the same as a step in S_1 iff they are in the same step space and have the same tag.

The intersection of a family of pre-solutions is not necessarily a pre-solution. This can happen when there are unreachable objects. For example, in Figure 2.1, let one pre-solution be one in which (the contents of) `itemA` is 3, and let another be one in which (the contents of) `itemA` is 4. The intersection of those two pre-solutions contains `stepA` and `tagA` but does not contain `itemA` (since the items in the two pre-solutions are different). Since the input to `stepA` is not in the intersection, that intersection cannot be a pre-solution.

A pre-solution which is also "closed forwards" is called an *aggressive pre-solution*. To be precise,

Definition *A pre-solution $\langle T, I, S\rangle$ is* aggressive *iff it satisfies the additional two conditions:*

A4. *If a step is in S then*

1. *all the items produced by the step are in I, and*
2. *all the tags produced by the step are in T.*

*Note that this condition subsumes the two conditions (each starting with "Further..." at the end of conditions **A1**(2) and **A2**(3).*

A5. *A step is in S if both*

1. *its tag is in T, and*
2. *all its input items are in I.*

It may be that no aggressive pre-solution exists. For instance, two initial tags may be the tags of two steps (with no input items) both of which produce the same tag. This would violate condition **A2**(3). If there exists at least one aggressive pre-solution, we say the TStreams program is consistent:

Definition *A TStreams program is* consistent *iff there is at least one aggressive pre-solution.*

Whether or not a TStreams program is consistent cannot be determined statically in general. However, we consider programs that are not consistent to be pathological. Typical programs are consistent.

Now to continue: It is not necessarily true that the intersection of a family of aggressive pre-solutions is a pre-solution (aggressive or not). (For instance, the counterexample given above for pre-solutions is also a counterexample for aggressive pre-solutions.) But as we will see, the intersection of *all* aggressive pre-solutions is an aggressive pre-solution.

We can now define what it means for a tag, item, or step to be reachable:

Definition *The set $\langle RT, RI, RS\rangle$ of all* reachable *tags, items, and steps in a consistent TStreams program is the intersection of all the aggressive pre-solutions.*

We will see shortly that this exactly agrees with what we mean intuitively by the term "reachable".

For convenience, we will denote the set of all aggressive pre-solutions of a consistent TStreams program by $\mathcal{A}$ and we will denote a generic aggressive pre-solution (that is, an arbitrary element of $\mathcal{A}$) by $\langle AT, AI, AS\rangle$.

2.1.4 Reachability and consistency: approximation from below

We now show how to approximate $\langle RT, RI, RS\rangle$ from below. For each level starting from 0, we define sets of objects as follows:

Level 0:

$$\begin{aligned} BT_0 &= \overline{\{\text{initial tags}\}} \\ BI_0 &= \{\text{initial items}\} \\ BS_0 &= \varnothing \\ BE_0 &= \{\text{“enabled steps”}\} \\ &= \{\text{steps with tags in } BT_0 \text{ and all of whose input items are in } BI_0.\} \end{aligned}$$

Level n (for $n > 0$)**:**

$$\begin{aligned} BT_n &= \overline{BT_{n-1} \cup \{\text{tags produced by steps in } BE_{n-1}\}} \\ BI_n &= BI_{n-1} \cup \{\text{items produced by steps in } BE_{n-1}\} \\ BS_n &= BS_{n-1} \cup BE_{n-1} \\ BE_n &= \{\text{“enabled steps”}\} \\ &= \{\text{steps with tags in } BT_n \text{ and all of whose input items are in } BI_n\} - BS_n \end{aligned}$$

and for consistency we define $BT_{-1} = BI_{-1} = BE_{-1} = BS_{-1} = \varnothing$.

We continue by defining

$$\begin{aligned} BT &= \lim BT_n = \cup BT_n \\ BI &= \lim BI_n = \cup BI_n \\ BS &= \lim BS_n = \cup BS_n \end{aligned}$$

(and clearly $BE = \lim BE_n = \varnothing$ and $\cup BE_n = \lim BS_n = BS$.)

2.2 Theorem *The following are equivalent:*

1. *The TStreams program is* consistent*.*
2. *$\langle BT, BI, BS\rangle$ satisfies the following constraints:*

   CST-1. *No item in BI is produced by two separate steps.*

   CST-2. *No tag in BT is produced by two separate steps.*

Further, if the TStreams program is consistent, then $\langle RT, RI, RS\rangle = \langle BT, BI, BS\rangle$.

PROOF. (2) $\implies$ (1): We will show that $\langle BT, BI, BS\rangle$ is an element of $\mathcal{A}$, thus showing that $\mathcal{A}$ is not empty and so the program is consistent.

To show that **A0**, **A1** , and **A2** are satisfied**:** We will first show that

1. The items in BI_n are just
   (a) the initial items,
   (b) the items in BI_{n-1}, and
   (c) each item produced by a step in BE_{n-1}.
2. The set of tags in BT_n is the closure of the set consisting of the following tags:
   (a) the initial tags,
   (b) the tags in BT_{n-1},
   (c) each tag produced by a step in BE_{n-1}.

   These statements are true when $n = 0$ (since BI_0 and BT_0 consist entirely of the initial items and the closure of the set of initial tags). When $n > 0$, the construction shows (by induction) that the items in BI_n are precisely those items that satisfy either 1b or 1c. Similarly, when $n > 0$ the set of tags in BT_n is precisely the closure of the set of tags that satisfy either 2b or 2c.

   With the exception of the uniqueness assertions, the properties **A0**, **A1**, and **A2** then follow from this by induction.

   The uniqueness assertions in **A1**(2) and **A2**(3) follow from the constraints **CST-1** and **CST-2**.

To show that **A3** and **A4** are satisfied**:** If a step is in BS, then it is in BS_n for some n. Let n be the smallest number for which the step is in BS_n, so the step is in BE_{n-1} but not also in BS_{n-1}. Then

- its tag is in BT_{n-1} and all of its input items are in BI_{n-1}, so its tag is in BT and all its input items are in BI. This yields **A3**.
- all the items it produces are in BI_n and all the tags it produces are in BT_n. This yields **A4**.

To show that **A5** is satisfied**:** If the tag of a step is in BT and all the input items of the step are in BI, then there is a least number n such that the step's tag is in BT_n and all its input items are in BI_n. This implies that the step is in BE_n, and is hence in BS, thus yielding **A5**.

(1) $\implies$ (2): Since the program is now assumed to be consistent, there is at least one aggressive pre-solution. We will show that if $\langle AT, AI, AS\rangle$ is any aggressive pre-solution, then each level $\langle BT_n, BI_n, BS_n\rangle$ is contained in $\langle AT, AI, AS\rangle$.

We have

$BT_0 = \overline{\{\text{initial tags}\}}$, so BT_0 is contained in AT by **A0**(1) and **A2**(4).

$BI_0 = \{\text{initial items}\}$, so BI_0 is contained in AI by **A0**(2).

BE_0 is contained in AS by **A5**.

Now say $n > 0$ and assume that

BT_{n-1} is contained in AT
BI_{n-1} is contained in AI
BE_{n-1} is contained in AS

Then

tags We know that

$$BT_n = \overline{BT_{n-1} \cup \{\text{tags produced by steps in } \overline{BE_{n-1}}\}}$$

By the inductive hypothesis, BT_{n-1} is contained in AT.

By **A4**(2), all the tags produced by steps in BE_{n-1} (which is contained in AS) are in AT.

By **A2**(4), AT is closed under derivation.

Therefore BT_n is contained in AT.

items We know that

$$BI_n = BI_{n-1} \cup \{\text{items produced by steps in } BE_{n-1}\}$$

By the inductive hypothesis, BI_{n-1} is contained in AI.

By **A4**(1), all the items produced by steps in BE_{n-1} (which is contained in AS) are in AI.

Therefore BI_n is contained in AI.

enabled sets We know that

$$BE_n = \{\text{steps with tags in } BT_n \text{ and all of whose input items are in } BI_n\} - BS_n$$

By **A5** together with the results just proved for BT_n and BI_n, BE_n is contained in AS.

Thus we have

$$BI = \cup BI_n \text{ is a subset of } AI$$
$$BT = \cup BT_n \text{ is a subset of } AT$$
$$BS = \cup BE_n \text{ is a subset of } AS$$

Thus $\langle BT, BI, BS\rangle$ must be contained in each aggressive pre-solution, and so $\langle BT, BI, BS\rangle$ satisfies conditions **CST-1** and **CST-2**.

Further, in the proof of (2) $\implies$ (1), we saw that this in turn implies that $\langle BT, BI, BS\rangle$ is an aggressive pre-solution. Thus $\langle BT, BI, BS\rangle$ is an aggressive pre-solution that is contained in every other aggressive pre-solution. Hence it is the intersection of all aggressive pre-solutions. That is, $\langle BT, BI, BS\rangle = \langle RT, RI, RS\rangle$. □

2.3 Corollary *The intersection of all the aggressive pre-solutions is an aggressive pre-solution.*

Proof. It is $\langle BT, BI, BS\rangle$, which we proved is an aggressive pre-solution. □

2.4 Lemma *The contents of each reachable item in a consistent TStreams program are uniquely determined by the name and tag of that item.*

PROOF. It's true by definition for items in BI_0. We proceed by induction. If it's true for all items in BI_j for $0 \leq j < n$ and if i is an item in $BI_n - BI_{n-1}$, then i is produced by a step in BE_{n-1}. Further, that step is unique. All the inputs to that step are in BI_{n-1}, and hence their contents are uniquely determined. Hence (by functionality of steps), the contents of i are uniquely determined. □

A *descending chain* is a list

$$\cdots \to S_k \to TI_k \to S_{k-1} \to TI_{k-1} \to \cdots \to S_2 \to TI_2 \to S_1 \to TI_1$$

where each S_j is a step and each TI_j is either a tag or an item, and where TI_1 may be absent—that is, the chain may end with S_1—such that each TI_j is produced by S_j and each TI_j parametrizes S_{j-1} (if TI_j is a tag) or is an input item to S_{j-1} (if TI_j is an item). Note that a descending chain by definition terminates on the right, although it might be extendible to the right.

We say that a TStreams program satisfies the *descending chain condition* iff each descending chain of reachable objects is finite (that is, does not extend indefinitely to the left).

2.5 Theorem *A consistent TStreams program satisfies the descending chain condition.*

PROOF. Let us say that the level of a tag, item, or step is the least n such that the tag, item, or step is in BT_n, BI_n, or BS_n, respectively. It is easy to see by induction that the levels of elements in a descending chain must strictly decrease as we move to the left. Since each level is non-negative, there can be only a finite number of elements in the chain. □

2.6 Corollary *The set of reachable objects of a consistent TStreams program does not contain a* loop *of the form*

$$S_1 \to TI_1 \to S_2 \to TI_2 \to \cdots \to S_{n-1} \to TI_{n-1} \to S_1$$

where S_j and TI_j are defined as above and where the first and last elements are the same.

PROOF. Such a loop would unroll into an infinite descending chain. □

2.1.5 Complete and well-formed programs

Definition *A TStreams program is* complete *iff it satisfies the following condition:*

*If **s** is a step space and **t** is a tag space corresponding to it (i.e., **t** is the tag space that holds the tags of each step in **s**), then each reachable tag in **t** corresponds to a reachable step in **s**.*

(What this condition says is that in such a case all the input items to that step are reachable.)

Figure 2.2 illustrates the kind of behavior that can lead to a program not being complete.

Definition *A TStreams program is* well-formed *iff it is both consistent and complete.*

Intuitively, one can think of the property of consistency as stating that a program does not have too many nodes, while the property of completeness says that a program has enough nodes.

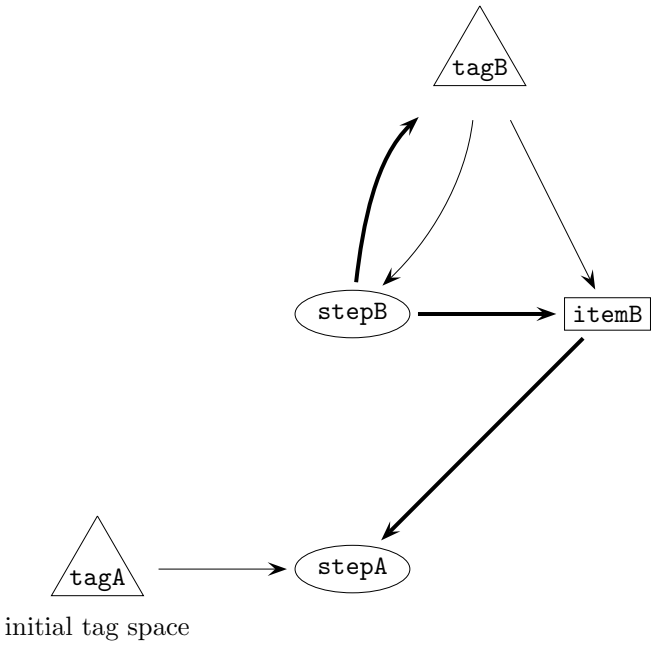

Figure 2.2: An incomplete program. Each space consists of exactly one tag, item, or step. There is one initial tag—no other objects are initial. The only reachable object is the initial tag (`tagA`), and although `stepA` is parametrized by `tagA`, it is not reachable because its input item (`itemB`) is not reachable.

2.1.6 Terminal items and solutions

We are now finally in a position to specify exactly what we mean by a solution and to characterize the set of all solutions.

Definition *If a TStreams program is well-formed, we say that* $\langle BT, BI, BS \rangle$ *(equivalently, the set of reachable tags, items, and steps) is the* maximal solution.

Definition *The* terminal items *of a well-formed program are all those reachable items which are in the terminal item spaces.*

The terminal tags *of a well-formed program are all those reachable tags which are in the terminal tag spaces.*

Definition *A* solution *of a well-formed program is a pre-solution which is a subset of the maximal solution and which contains the terminal items and terminal tags.*

2.7 Lemma *If* $\mathcal{Z}$ *is a family of pre-solutions of a well-formed program, each of which is a subset of the maximal solution, then the intersection of the elements of* $\mathcal{Z}$ *is a pre-solution.*

PROOF. We denote elements of $\mathcal{Z}$ by $Z = \langle T_Z, I_Z, S_Z \rangle$. We denote the intersection by

$$T = \cap T_Z$$
$$I = \cap I_Z$$
$$S = \cap S_Z$$

A0. Each initial tag is in each T_Z, so it is in T. Similarly for initial items.

A1. $i \in I \implies$ i is in each I_Z

$\implies$ either

- i is an initial item, or
- i is produced by a unique step in each solution Z, and this step must be the same in each solution since it is uniquely determined in the maximal solution.

A2. $t \in T \implies$ t is in each T_Z

$\implies$ either

- t is an initial tag, or
- t is a tag derived (uniquely) from tags in T_Z for each solution Z (this unique derivation must be the same in each solution), or
- t is produced by a unique step in each solution Z, and this step must be the same in each solution since it is uniquely determined in the maximal solution.

Further, T is closed, because it is the intersection of the sets T_Z, each of which is closed.

A3. $s \in S \implies$ s is in each S_Z

$\implies$ the tag of s is in each T_Z, and all its input items are in each I_Z. Further, each input item has the same contents in each I_Z, since for reachable items the content of the item is determined by its tag. Thus the input items are identical in each I_Z.

$\implies$ the tag of s is in T and all its input items are in I.

This concludes the proof. □

2.8 Theorem *The intersection of a family of solutions of a well-formed program is a solution.*

PROOF. The intersection is certainly a subset of the maximal solution, and it includes the terminal items and tags. The lemma shows that the intersection is also a pre-solution, and we are done. □

Thus the intersection of all solutions of a well-formed program is a solution, which we call the *minimal solution*.

Note that if there are no terminal items or tags, then the minimal solution just consists of the initial items and tags.

2.9 Theorem *Any reachable pre-solution of a well-formed program either*

1. *contains a step not all of whose output tags and items are in the pre-solution, or*
2. *contains a tag and items that enable a step which is not in the pre-solution, or*
3. *is the maximal solution.*

Remark Note that we do not assume that the reachable pre-solution contains the terminal items and tags, so it is not necessarily a solution.

PROOF. If 1 and 2 are not true for some reachable pre-solution, then conditions **A4**, and **A5** are both true. Therefore, the state is an aggressive pre-solution. Since it is contained in the maximal solution (which is the minimal aggressive pre-solution), it must therefore be the maximal solution. □

2.2 Executing a TStreams program

We now describe (at least conceptually) how a TStreams program actually runs. We will assume from this point on that all TStreams programs are well-formed.

An execution of a TStreams program is defined by a sequence of *states* that the program passes through. Thus in order to describe how a TStreams program is executed, we need to provide a model in which the following questions are answered:

1. What do we mean by a state of a TStreams program? What is an allowable state?
2. How does a program start? That is, what is an initial state?
3. How do we pass from one state to another?
4. How does a program terminate? Equivalently, how do we recognize a terminal state?

2.2.1 The abstract execution model

We first describe an abstract execution model, in terms of which any other execution model can be interpreted:

The objects, relations, and boundary conditions of the abstract execution model are the same as those in the model of the TStreams kernel. The only differences are these:

- Tag spaces, item spaces, and step spaces are now sets which—except for the initial tag and item spaces—are initially empty, and whose contents change as the program executes.

  The initial tag and item spaces are initialized to their specified values, and the contents of those spaces never change during execution of the program.

- Each step falls into one of three categories:

  prescribed A step is **prescribed** if its tag has been produced.

  enabled A step is **enabled** if its tag has been produced and all its input items have been produced.

executed A step is in this category if it has already been executed and all its output items and tags have been produced.

In this model, execution of a step is atomic, in the sense that either all the output items and tags have been produced or none of them have.

This choice of 3 categories above might seem to imply that a step must be **prescribed**—that is, its tag must have been produced—before its inputs are produced. This is not the case. The inputs of a step may be available without its tag having been produced. If its tag is then produced, it is immediately in the **enabled** category without ever having been in the **prescribed** category. These 3 categories are chosen because they are exactly the ones we will need below.

Now we are ready to answer the four questions stated above:

What is a state? Intuitively, a state is the collection of

- all the tags that have been produced, together with
- all the items that have been produced, together with
- all the steps whose tags have been produced

in a running program, together with the knowledge for each such step of what category it is in.

To give a precise definition, we simply say that a state is a collection of tags, items, and steps (each step with its category) which is either an initial state (as defined below) or is a state that is a successor of another state (as also defined below).

What is an initial state? There is a unique initial state. This is the state $\langle T, I, S\rangle$ consisting of

- the set I of initial items,
- the closure T of the set of initial tags, and
- the set S of **prescribed** steps (that is, the set of steps whose tags are in T).

  Some of these steps may be **enabled**. (In fact as we will note below, at least one must be if the program is not completely trivial.)

How do we pass from one state to another? In short, by *executing* a step. Let us denote an arbitrary state by $\langle T, I, S\rangle$. A step in S that is **enabled** may execute. When such a step executes, it it produces certain tags and/or items. Then the following happens:

- The new tags and items are added to the current state to produce a new family $\langle T', I', S\rangle$ of tags, items, and steps.
- The set T' of tags is then closed with respect to derivation to produce $\overline{T'}$.
- The step that has just executed changes its category to **executed**.
- Each step whose tag is a new tag in T' is thereby **prescribed**, and is added to S to form a set S'. Each step in S' that is **prescribed** then checks to see if all its input items are available. If so, it changes its category to **enabled**.

The new state is the state $\langle \overline{T'}, I', S'\rangle$.

All these actions, starting with the production of tags and items by the step being executed, happen simultaneously so far as the model is concerned.

How do we recognize a terminal state? A state is a terminal state iff it contains the terminal items and tags.

Now we show that this model of execution implements the semantics that we have defined in Section 2.1. First, if $\langle T, I, S\rangle$ is a state, let us denote by S_0 the steps in S that are enabled or executed. We will say that $\langle T, I, S_0\rangle$ is the *kernel* of the state $\langle T, I, S\rangle$.

2.10 Theorem *In an execution of the program as outlined above, the kernel of each state is a reachable pre-solution which contains every step enabled by tags and items in the kernel.*

PROOF. By induction, all objects in the kernel of each state are reachable. In fact, all objects in the kernel of the n^{th} state are in level n as defined above in Section 2.1.4 (page 25).

Further, again by induction, the kernel of each state satisfies conditions **A0**–**A3**, and is therefore a pre-solution. □

2.11 Corollary *A state is terminal iff its kernel is a solution of the TStreams program.*

In each state there may be more than one enabled step. Theorem 2.10 shows that the same set of terminal items and tags is computed regardless which enabled step executes first. In fact, if in a state there are two steps that are enabled, then executing them in either order yields the same resulting state. Hence we can even allow steps to be executed simultaneously in our model, simultaneous execution of a family of steps being equivalent to the execution in some order of those steps.

2.12 Theorem *If a state is not a terminal state, then it contains at least one enabled step.*

PROOF. This follows immediately from Theorem 2.9. □

Thus, execution can always continue until a terminal state is arrived at—there is no possibility of a well-formed program stalling.

This still leaves a lot of room for variation, since in general there may be more than one terminal state. Here are two specific kinds of execution:

demand-driven (or "backward-looking") A demand-driven execution is one in which the terminal state is the (unique) minimal solution.

aggressive (or "forward-looking") An aggressive execution is one in which the terminal state is the (unique) maximal solution. This amounts to saying that there are no enabled steps that have not been executed. Thus, execution continues until there is nothing more than can be done.

In any case, aggressive execution and demand-driven execution compute the same set of terminal items and tags, and thereby assign the same meaning to a well-formed program.

There are some questions we have not really addressed so far in this model:

- How is control of the running system managed? Where does control happen? How many loci of control are there?

  One way to deal with this is to regard each step as a *step processor*, and to regard each tag as also embodying a *tag processor*. To say that a tag is produced by a step is to say that the tag together with its tag processor is produced by the step processor executing that step.

The function of a tag processor is to produce step processors for each step parametrized by its tag.

The function of a step processor is to decide when its step is enabled and to then execute that step. To do this, each step processor has knowledge of the (tags of the) input items required by its step, and it knows if and when those input items have been produced.

The reason this is possible is that, as was already specified in Section 2.1, the tags of items consumed by a step are computable functions of the tag of the step.

We assume that each step processor sees any inputs to its step as soon as they are produced.

As we have seen, these processors can all operate completely independently.

- How does a program terminate?

  A program should terminate when it reaches a terminal state. But how is such a state recognized? We can assume that part of the run-time system for a TStreams program is a special processor that has a global view of the TStreams state, and so can recognize when a terminal state has been reached. At such a point, this processor can halt execution.

2.2.2 A concrete execution model

There are a number of practical issues we have ignored in our description of the abstract execution model. Here are a few; there are of course many more, but these are quite immediate:

- In practice, the number of processors, and therefore the number of loci of control, is limited. We cannot simply say that each step instance constitutes a separate processor.
- We have implicitly assumed that there is one notion of time that is shared by all the processors.
- We have explicitly assumed that processors see changes of state instantaneously.

We can relax these assumptions, and build a more concrete execution model, in the following way:

We assume that there are a certain number of processors. The number must be finite, and must be at least 1. Each processor has its own view of the entire executing TStreams program.

We will assume that the processors act asynchronously and independently. This changes the way we look at time. In the abstract execution model, to the extent that we need to think of time at all, we can think of it as a variable whose domain is the non-negative integers. In the concrete model, it seems more natural to think of a global "wall-clock" time which is a variable whose domain is the non-negative real numbers.

As before, we assume that all steps execute atomically. Then at any time there is a well-defined state of the system, where again the kernel of each state is a reachable pre-solution. Let us denote the state at time t by Σ_t.

We do not assume, however, that a processor sees the entire state of the system at any particular time. In fact, we do not assume that any processor has the same clock as the global clock—it may have its own clock. All we assume is this:

- Each processor has a local clock. The local time on a processor is an increasing function of the global time which tends to ∞ as the global time tends to ∞. When we use the word "time" without any qualification, it refers to the global time. A processor's local time is always referred to as such.

- At (global) time t each processor p, having local time t_p, sees a subset of the state Σ_t. There is guaranteed to be a local time $t'_p \geq t_p$ such that at local time t'_p processor p sees everything which is in Σ_t. (Of course, it may also see other objects as well, as the global state may have progressed by that later time.)

We do not assume that anything is known about the relations between t, t_p, and t'_p, other than what is stated. For instance, we make no assumption about the computability of t_p in terms of t. And in particular, $t'_p - t_p$ may be unbounded as a function of t_p.

At any time, a processor that sees an enabled step may execute that step, thus changing the state. If two processors see the same enabled step, they may both execute that step, because, since steps are functional, they will produce the same results. Just as in the abstract execution model, every state thus produced is a reachable pre-solution.

The kernel of each state is either the maximal solution or contains one or more enabled steps that have not yet been executed. While a processor may not see such an enabled step immediately, our assumptions above ensure that in a finite amount of time, the processor will see the tag of the step together with all the input items of the step, and so will be able to execute the step, if it has not already been executed. (Or even if it has by then been executed by some other processor—as already mentioned, this can do no harm.) Therefore progress in execution can always be made, although any particular processor may have to wait a certain amount of time before proceeding. Further, every step that is executed will eventually be seen by all the processors as having been executed. Therefore, no processor will spend its time continually re-executing an already executed step. So not only is it the case that progress in execution *can* be made—it *will* be made.

In this model, it is nowhere assumed that the number of processors is constant. Therefore, this model naturally accommodates systems in which processors may fail, or in which processors may be added or withdrawn at arbitrary times during the execution of the program.

Index