Tom's Turing Complete Architecture

v 0.96

by Thomas Walker Lynch

thomas.walker.lynch@gmail.com

www.thomas-walker-lynch.com

Contents

Preface

This is a work in progress, *version .95*, i.e. not yet a first edition. The current text will surely be edited, and more content will be added. The silver lining to this is that your feedback can affect the future direction of the text.

In 2005 a colleague and I shopped a proposal to build my “Turing Complete” Processor, but did not find funding. I am writing this text so the proposal will not be lost to the sands of time. This volume describes the theory behind the architecture, and while doing so it also builds a generally descriptive bridge from computation theory to real computer architectures.

This document follows RFC 2119 for the definition of the words: *may*, *should*, and *must*. See https://www.rfc-editor.org/rfc/rfc2119. Accordingly the word *may* signifies the giving of permission or the presentation of options. Consider this example. You are entering a subway station, and there is a turnstile that you must pass through to get inside. The turnstile is a state machine of sorts. With no coin inserted it is locked. With a coin inserted it allows one turn then goes back to the locked state. You look at the plaque by the coin slot and it says: "You *may* pass through the turnstile after depositing a coin." These directions are not making a prediction for the future, rather they are telling you what you are allowed to do. In this paper when I want to predict that something is probable to happen, but not certain, I will use the word *might*. This paper talks a lot about state machines, so understanding these definitions will be very helpful.

Source code will be shown in a dark red monospaced font. Headings follow the form of sentences without the final period. Technical terms that are being defined in a section and appear in the section heading are shown in italics. In the body of sections technical terms not in the heading will be shown italics when first introduced, and otherwise will have no special decoration. Citations are given within the text by DOI or ISBN number.

Formatting this book for the Kindle reader was a challenge that ironically involved some of the very issues discussed in this book. The Kindle reader allows the user to change the fonts. Hence all dimensions are relative to logical units. In contrast Microsoft word is one of the oldest WYSIWIG editors – i.e. what you see is what you get. It favors absolute measurements for distances from anchors and the sizes of text boxes. In Word when something does not fit, the author moves things around until they look right. These two philosophies of logical formatting and absolute measure formatting are generally incompatible.

It is instructive to consider the case of the text box. Microsoft Word text boxes do not grow, so if a user makes the font bigger, content will be clipped. A hedge against this is to pad the text box with spaces. However, too much space is not aesthetic because it looks wasteful. How much space is necessary? We cannot know. I say this is instructive, as we discuss this very problem of fixed length memory allocations for expanding data in this volume.

In this document the diamond, ♦, placed before a figure is a one character long paragraphs that serve as an anchor point. If the diamond is not on the upper left of the figure you know that Word and Kindle conspired together to float the figure into the anchor paragraph, but hopefully Kindle's evil deed was thwarted because the paragraph is only one character long, so the diamond just moves to the bottom of the figure.

Another thing that Kindle does, which is disastrous for a CS book: it strips out the monospaced fonts. It will keep italic and bold, but not monospacing. This messes up source code examples. The recent Kindle Create program has a work around – it turns all text boxes into images.

Introduction

Let us begin by contemplating the meaning of the term *Turing Complete* as it is applied to an *instruction set architecture*.

An instruction set architecture defines a set of instructions and registers used at the assembly language level for programming a processor. Invariably we will find among the instructions those used for computation, and those of a different nature which either directly or through side effects load and store data from a system memory. Unless I/O is memory mapped, other instructions will be for moving data to and from peripheral channels. It will almost always be the case that there is a secondary storage device that is accessed through a channel, via memory mapped I/O, or through a network interface. On modern machines there will also be some sort of interrupt architecture whereby various events may be programmed to invoke instruction sequences without having to explicitly call or branch to those instruction sequences.

Typically the instruction set architecture is not used directly by programmers. Instead there will be a collection of programs which simplify and abstract the architecture interface so that the work of programming the computer can be divided among programmers of various disciplines.

What happens at the layer below the instruction set architecture depends on the specifics of the implementation. At this lower layer every single bit of information is explicitly manipulated using transistors. These transistors will be composed to create logic gates, and transistors and gates will be combined to implement the desired functions. During the design phase a synthesis program or engineers will draw these circuits, and should they pass logical and timing tests, the transistors will be drawn to size, given places where they will appear on the substrate, and have lines drawn between them to 'wire them up'. In the current era of computing, these designs will be realized on silicon chips.

Hence the instruction set architecture is a nice vehicle for communicating just enough information to compiler writers that they may port their tools, while also not conveying a lot of extra information that would be difficult for a non-specialist of hardware design to understand, and in any case would be irrelevant to programmers.

Given two sufficiently expressive instruction set architectures for machines, say A and B, it will be possible to write an emulator program using instruction set A, where this emulator can read and run programs that were written using instruction set B. Such an emulator might be very useful if a person does not have in their possession hardware for instruction set B. Indeed it might even be the case that instruction set B was never intended to ever be implemented in hardware. For example, when instruction set B is Java byte code, which by its very design was intended to be emulated.

Alan Turing published his computing machine abstraction in 1937 as part of, "On Computable Numbers With an Application to the Entscheidungsproblem" DOI 10.1112/plms/s2-42.1.230. His objective was to embody the meaning of 'algorithm'. He used the Turing Machine as a tool to answer a question posed by Hilbert in 1928 as to whether an algorithm exists that can generally decide if a given logical statement is provable. Turing showed that in so far as algorithms are defined in terms of programming his machine model, that no such algorithm for generally deciding if statements are provable can exist. This was an earthshaking result. Although it was not the first result to show limitations for a system of logic. Gödel's Completeness theorem was published in 1929, and Russel's paradox dates to 1901, with a similar observation by Zemelo a couple of years earlier.

Turing's machine is truly simple. It consists of a tape, a read-write head, and a state machine controller. The state controller chooses its next state based on the value under the head, and then it may choose to write a new value to tape, or to take a step. The Turing Machine becomes a true computer when the state controller is designed to read a description of Turing Machine logic from tape and then to execute that. The controller description found on the tape is a program encoded as data. Such a ‘Universal Turing Machine’ can emulate any other Turing Machine, thus showing that a Universal Turing Machine is programable.

Simpler models for computation are possible, as examples: finite state machines by themselves, or a state controller combined with a stack instead of a tape. However, such simplifications come at the cost of reduced expressiveness. I.e. there are some problems that Turing Machines can compute solutions for, which these other machines cannot.

More complex models also exist, as examples: machines with multiple heads, multidimensional tapes, or even those that employ multiple co-operating Turing Machines. Yet these more complex machines are *not* computationally more expressive. Any problem that these machines can compute solutions for, so can a Turing Machine. So it appears that the Turing Machine is as simple as we can make a computing model without diminishing its functional expressiveness. When another machine model can compute the same set of problems as a Turing Machine, we say that the other model is *Turing complete*.

The infinite tape of the Turing Machine does not intimidate engineers. Some engineers have brushed off this detail to create realizations that are the same except that they have very long tapes instead of infinite ones. Engineers justify doing such a thing by pointing out that it takes a long time for computers to use a lot of memory, so we will probably never see the difference. Yet no matter how long the tape is made, i.e. how much memory is in a system, some calculations do hit the end of any tape. When this happens the engineers explain to us that the problem is a ‘memory fault’.

If we had an engineer’s Turing Machine realization, the first thing we would discover is that it is tedious to program. The instruction set architecture gives us no preconceived data types, not even integers, nor any instructions that accomplish arithmetic tasks. Another thing we would notice is that it does not do much on each step. Whereas a conventional processor might add two fixed width integers in a single step, a Turing Machine will take many steps to do this same.

Typically in order to keep our lives simple when we are faced with programming a Turing Machine we use unary representation for integers. Accordingly the numbers 1, 2, 3, would be something like ‘s’, ‘ss’, ‘sss’. The ‘s’ represents our successor function from Peano arithmetic. We then may write a program library with some useful functions. For example, to increment an integer the program looks under the read-write head, if it sees an ‘s’, it steps right. If it doesn’t see an ‘s’ then it writes an ‘s’ and halts. In this manner the string of ‘s’ symbols gets longer by one. We might then write subroutines to perform all the instructions found in a more conventional instruction set architecture, and in this manner construct an emulator.

When we try to write programs of any complexity with our new emulator, we run into a problem that is at a higher order of computation than the arithmetic itself: How can we make a memory map of variables that hold variable width data? Say we wanted to compile a program and place three integer variables next to each other on the tape. Then we would discover that if one of the variables grew in width, it would walk into the memory that was allocated for its neighbor and clobber its value.

Along comes our engineer, and he has a practical fix for this problem. He notes that if we instead use Arabic representation, numbers will grow very slowly while they are incremented. Hence, he concludes that it will be sufficient to replace the need for an infinite allocation space for each integer with a large fixed width one. We call such a large fixed width allocation for our basic integer type a *word.* In addition our engineer will ask, how high does a person need to count anyway?

But what happens in the highly unlikely case an integer does get too big to fit in one word of allocation? One possibility is that we call such an unlikely event an ‘exception’ and then stop the program. This exception is called ‘integer overflow’. Another possibility is to saturate at the maximum value. More recently it has been proposed that we instead substitute in a special bit pattern that will be interpreted to stand for *infinity* and then keep computing. Today most computers will simply wrap back to zero and keep going as though nothing happened. But computers often have important jobs to do. It is not unimaginable that such a computer would be controlling a rocket, and then the rocket loses control and blows up, or something even worse could happen.

There is one more solution to the exceptional case of integer overflow which has recently become popular in interpreted languages, and that is to use bignums. Accordingly, when the integer gets too big we either move it, or place its pieces into a data structure such as a linked list. Both solutions typically require that we have something called an ‘address’ that can be used to locate our integer or integer pieces.

In terms of a Turing Machine tape, an address is the offset from the first tape cell. We may generate addresses by counting the steps taken while a machine simply steps right repeatedly. Thus an address is an integer, and the bignum solution does not solve the integer overflow problem, it just displaces the problem from our bignum integers to our address integers. This is fine with engineers, because they have now moved from 32 bit addresses to 64 bit addresses.

If we modify our engineer’s variation on the Turing Machine by adding libraries or hardware for handling fixed length integers and use addresses for accessing the tape, then we get close to a modern instruction set architecture.

Since the beginning of computing we have lived with the simplification that integers fit in fixed width words, and consequently have also lived with otherwise perfectly good programs terminating with exceptions. The problem typically gets worse during the transition to a new technology node when performance reaches a new plateau not imagined as being possible just a short time before. We might be nearing such a technology node now by moving to quantum computing.

Is there a point where the engineer's solution will finally be sufficient? We live in a finite universe, and address spaces grow exponentially as we add more bits to addresses. There are approximately 2^{265} protons in the universe. Hence with a 265 bit address, we could give every proton an address. Surely no memory would ever be made that large. An architecture using 265 bit fixed width addresses would never signal an *address space is exhausted* fault while running real programs, because any real program would have had an *out of physical memory* fault before that could happen. However such an argument must be taken with a grain of salt, because the same argument form was used to justify using 32 bit addresses as the end all solution, and they are now being used to justify 64 bit or 96 bit addresses as the end all solution.

One possible hitch is that OSs tend to reserve blocks of address space. It is conceivable that an OS designer would create an address space management algorithm that is so inefficient, that despite memory being available, the OS still would not be able to map it even with 265 bit addresses. Also, with so many address bits the temptation is strong to instead use the addresses as associative keys or as content holders for a content addressable memory. (With content addressable memories large address widths are used for sparsely addressing relatively small memories so as to achieve searches with performance on the order of hardware decode circuit times.) As another possibility, we might soon have to deal with state explosions in quantum computers, so perhaps we really will need all those bits, and more, as conventional addresses.

Independent of those possible scenarios, there is a *here and now* problem with using lots of bits to safeguard the fixed width data assumption. Namely that large fixed width values are an inefficient way to allocate memory. Addresses might be large numbers, but they still tend to congregate around a base value, i.e. indexes are typically small. The typical integer in a program does not exceed 12 bits, but it will be allocated in 32 or 64 bit words. If we did not waste this space, caches could have higher hit rates, we could afford to run more threads, and we would get more work done with a given amount of system memory.

And we have not even touched on the issues of floating-point representation.

All this raises a question, is the fixed width assumption necessary for practical computing?

In computation theory we often use 'pumping' type proofs to show that a particular machine is not Turing Complete. The idea is that we show some problem that a Turing Machine can solve independent of the length of input, and then we prove that for any solution on the non-Turing Complete machine that there will be a given length of input that is so large that the machine breaks and cannot solve the problem.

Take for example, suppose that our input is a series of the symbol 'a' followed by a series of the symbol 'b', and we want the machine to tell us whether or not the number of 'a' symbols is equal to the number of 'b' symbols by printing a 'y' or a 'n' on the tape.

♦

ε	**y**	empty string
ab	**y**	same number of a and b
a	**n**	fewer b
b	**n**	fewer a
aaaaabbbbb	**y**	same number of a and b
aaaaabbbb	**n**	fewer b
aaaaabbbbbb	**n**	fewer a

This is an easy problem to solve on a Turing Machine. The controller moves the head to the ends and trims off one symbol from each of the strings of 'a's and 'b's. If the ends meet the controller writes 'y', otherwise it writes 'n'. If we consider that each of the strings is a unary number, then these operations can be thought of as subtracting 1 from each number and expecting both numbers to become zero on the same iteration.

Now suppose we instead of a Turing Machine, that are input is fed to a finite state machine. We can have a series of states that stand for one ‘a’, two ‘a’s, three ‘a’s, etc. Each state in the series of ‘a’ states would have a ‘b’ arc leading into the series of ‘b’ states that regress back, two ‘b’s, one ‘b’ and if the string ends with us on the initial state we know the ‘a’s and ‘b’s strings were equal in length. The problem is that we will only have a fixed number of states in a finite state machine, so there will be a bound to how many ‘a’s and ‘b’s that can be counted.

We can discover the limitation of the state machine solution by *pumping* the input. First we give the machine length one strings, then length two strings, etc. At some point the machine will attempt to go past the last ‘a’ state and go to a failure state.

Now let’s consider our engineer’s solutions for the unbounded memory problem, the address space problem, and the problem of creating a memory map for variable width variables. All those solutions will break as various aspects of computation are pumped. We will get a fault after using all the memory. If we are swapping pages back to the hard drive of sufficient size we will get a fault when we run out of address space. If we have integers and the program makes them grow as it runs longer, as the input gets longer, there will eventually be a numeric overflow.

When modern instruction set architectures are used in the manner they were designed to be used, we cannot arbitrarily pump input lengths without things breaking. I.e. using the add instruction to perform addition will lead to overflows, load instructions accept address operands of fixed width, and the address translation system for the VM will only be able to handle a fixed number of page translations. It is for this reason that I would say that today’s instruction set architectures are not Turing Complete.

But can we use our instruction set architecture to write a Turing Machine emulator? If so, then the instruction set architecture is Turing Complete, as the emulator can do anything that a Turing Machine can do. Theorists put no weight on a metric such as ‘what it was designed to do’.

Any Turing Machine emulator approach must address how we are going to deal with the infinite tape. A simple approach is to put the processor in the position of being the state controller, and for external storage to be the tape.

Say we use an actual magnetic tape transport as the secondary storage, and then say it is not the processor’s fault if we hit the end of a very long tape. Real tape storage we will run into faults when pumping that are not related to hitting the end of the tape. Tape storage mechanism use formatted tapes, and tape controllers support seek commands that accept address operands. At this point we can wave our hands and imagine a special tape drive and controller. One that has been designed for our purposes. This simplified device only accepts step right, step left, and read and write operations. It is OK for us to wave our hands and do this because it would be possible to build such a tape drive.

This setup is nice in that the only fault will be that of physically running out of tape. We can wave our hands and imagine that we always add more tape. However, this is also a lame proof. It is hard to imagine an instruction set architecture that cannot emulate a finite state machine. In addition most all of the features of the given instruction set architecture will not be used. The message that an instruction set architecture is Turing Complete as determined by using this set of assumptions carries little or no interesting information.

The remainder of this volume is about an instruction set architecture where, even while using the instructions in the instruction set, the only faults will be due to program design, the properties of mathematics, or due to real resource limitations. This instruction set architecture is Turing Complete due to the features of the instruction set itself, rather than being in spite of it.

The TTCA Turing Machine variation

The material in this chapter owes a great deal to, *Elements of the Theory of Computation* by Harry Louis and Christos Papdimitriou, ISBN 978-0132624787.

However in this volume we are not exploring the nature of computation, rather we are going in the other direction. We are trying to work our way up from computation theory to arrive at a practical computer architecture. Towards this end we start with a definition for the Turing Machine that itself resembles a computer architecture specification.

♦

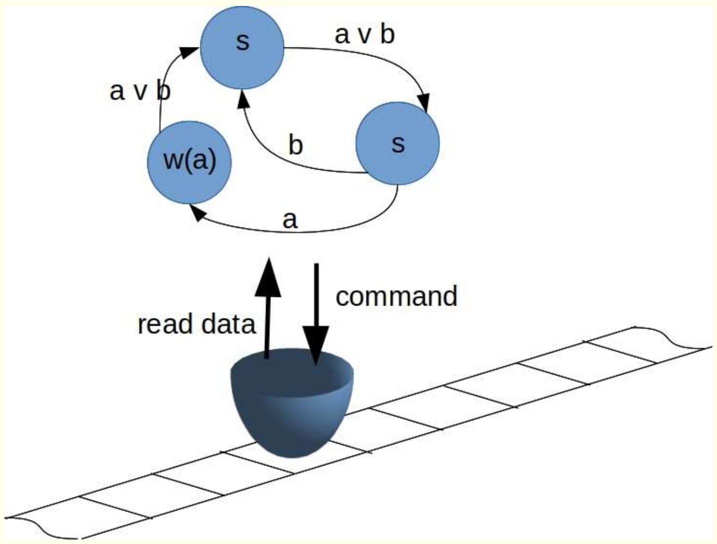


Figure 1 A Turing Machine

Accordingly, our Turing machine consists of a:

1. fixed finite *alphabet*

2. fixed distinct *empty-symbol*
3. variable single ended *tape*
4. variable read/write *head*
5. variable read value buffer
6. fixed state machine *controller*
7. fixed left from leftmost error
8. fixed start-state
9. fixed halt-state
10. variable current-state
11. fixed *procedure* for using these.

The values assigned to the variables are not part of the definition.

A *symbol* may be any mathematical object for which there may be multiple instances, where within a given context, instances compare to be equal among other instances of the same symbol, but compare not equal to instances of other symbols. As a matter of convenience in this document, I use sequences of letters and/or digits for symbol instances.

The *alphabet* is a finite set of *symbols*.

The distinct *empty-symbol* may be any symbol that is not found in the alphabet. Like alphabet symbols, this symbol may appear in the tape sequence.

Only instances of alphabet symbols or the empty-symbol may be written to the tape. Intuitively we might consider that the alphabet symbols are useful while the empty symbol is just taking up space while waiting to be displaced, in the same manner that we consider a bookshelf to be empty rather than being full of air. (And if we put a bookshelf underwater is it still empty, or is it full of water?).

The tape is an infinite sequence. We say that the first element in the tape sequence is the *leftmost* element. The second element in the sequence is then the *right neighbor* of the leftmost element. Etc. this nomenclature is drawn from a left to right writing convention for sequences.[1] The leftmost element has no left neighbor, but it does have a right neighbor. All other sequence elements have both a left neighbor and a right neighbor. A Turing Machine tape has no rightmost element because it is an infinite sequence.

We notice three special subsequences:

1. a first, leftmost, element which is either an alphabet symbol or the empty-symbol
2. optionally this is followed by a sequence extending to a rightmost alphabet symbol
3. if the second subsequence is finite, it is followed by an infinite sequence of empty-symbols.

So as to distinguish this tape partitioning from others found in this volume, we give it the name, the 'Leftmost Rightmost Partitioning', because it emphasizes the location of the leftmost element (alphabet or empty) and the rightmost alphabet element.

It is because of this third subsequence, the infinite tail, that the empty-symbol became distinct from the alphabet. There is no way for a Turing Machine to compute an infinite empty tail, so such a tail must come from somewhere else. Here, we provide it by definition.

Putting together the first two subsequences noted above gives us the *active area*. If there is no alphabet character on the tape, then there is no active area (i.e. the

[1] Later we will add addresses, and the tape then will have a number line under it, where the leftmost cell has a zero under it, the right neighbor has a one under it, etc. For writing systems that progress from right to left, or even up and down, we will want to maintain a convention that keeps the addresses increasing in the positive direction for a number line.

active area has length 0). The infinite right going tail of cells holding the empty-symbol is then the *inactive* area.

The initial active area provided to a Turing Machine is called the *input*. The final active area, obtained after the Turing Machine halts, is called the *output*. If the input has been computed solely from another Turing Machine, then that input, will necessarily be finite in length. If the input is provided as a general mathematical object, perhaps even one abstracted from 'what a Turing Machine computation would produce if it ran to an infinite limit number of steps', then the input may be either finite or infinite.

The read/write head indicates[2] one of the elements in the tape sequence. The head is not a Natural Number as you are probably accustomed to for sequence element indexes. Such a definition would be circular because later we use a Turing Machine to define Natural Numbers and the concept of an address. Instead, we can think of the head as partitioning the tape into two subsequences. We will call this the 'Head Centric' partitioning. The first subsequence extends from the leftmost element up to the indicated element, inclusively. During computation this first subsequence is necessarily finite. We call it the *left-hand side*. The second subsequence holds the rest of the tape. We call it the *right-hand side*.

To step the head to the right we remove the leftmost element from the right-hand side, and then append it to, i.e. make it the new rightmost element of, the left-hand side. To step the head left, we remove the rightmost element of the left-hand side, and prepend it, i.e. make it the leftmost element of, the right-hand side. In the section, **Addresses and cells** , we will justify using the term *cell* in place of *element*, and then we will be able to say such things as *the cell that the head is on*, or the *indicated cell*.

[2] Computer people have a habit of using the word 'indexes' in place of the word 'indicates'. Accordingly they might say that an index variable *indexes* a value in an array, instead of saying that the index variable *indicates* a value in an array.

The controlling state machine is a graph. It may be either a Mealy or Moore style state machine graph, as these are equally expressive. Only the fixed procedure would change. We describe a Moore style machine and corresponding procedure here. The graph is a set containing three elements:

1. set of states
2. set of <state, command> pairings
3. set of arcs

A state is a symbol. Instances of state symbols appear in a different context than those of alphabet symbols or the empty-symbol, and thus they do not need to be distinct from them.

Each state has a command attached to it, <state, command>. Each state appears in exactly one such pairing. We have a fixed set of commands: { **`do-nothing`**, **`step-left`**, **`step-right`**, and **`write(value)`** }, where the **`value`** parameter is set when the controller is defined. **`value`** must be an instance of an alphabet symbol or an instance of the empty-symbol. In addition, we should not forget the implied **`read`** command that returns a value from the tape, and is used by the fixed procedure.

The arcs are triples of symbols, <*from-state*, *to-state*, *value*>. The *from-state*s, *to-state*s, and *value*s are set when the controller is defined. The *value* is an instance of either an alphabet symbol or the empty-symbol. We require that for each *from-state* that there exists exactly one arc for each member of the alphabet, one for the empty-symbol, and for states with a **`step-left`** command, that there be one arc for the left from leftmost error. Arcs can be dropped from diagrams if we can prove that they cannot be used. As a notational convenience we might also reduce the work of listing so many arcs by adopting the convention of explicitly providing only unique arcs, and then saying the rest of the arcs are identical to a *default* arc or an *error* arc.

Here we present the fixed procedure in the form of a main procedure and two subprocedures, 'Initialize', and 'Take a Step'. To perform a computation follow the steps in *Main*, and its subprocedures, until being told to halt:

♦

```
Deterministic (Uniplex) Procedure

Initialize:

1. place the input on the tape

2. place the head on the leftmost element

3. set the current state to the start state.


Take a Step:

1. read the symbol instance indicated by the head

2. execute the command corresponding to the current state

3. select the arc that has a from-state that matches the current-state, and a
value that matches the instance just read. Take that arc's next state and make
it the machine's next state.

4. make the next state the current state.
```

We initialize the machine by instantiating the input on the tape, putting the head on the leftmost element of the tape, and setting the current state to the start state.

We take a step by first reading the symbol under the head and storing that value in our read value buffer, executing the command associated with the state, and then following the arc selected by the element we read. We must read the element before executing the command, because the command might perform a write to the cell. Consequently, as part of the Turing Machine definition we picked up the buffer for holding the read value.

Walking through this procedure is variously known as running the machine, performing a computation, evaluating the input, evaluating the function, or simply as computing.

Note that while walking through this procedure we are always in exactly one state. This is because we required that there be one arc for each letter in the alphabet, empty-symbol, and possibly the left from leftmost error. We say that this property means that the machine is *deterministic*.

To create a *non-deterministic* machine we allow multiple arcs to have the same read value as a second member, and we modify the second step of the procedure to follow all arcs that have matches with the symbol returned by **`read`**. When we do this we might find that zero, one, or many of the arcs are followed in one step, and thus that we may have zero or more next states. Conceptually each of these states corresponds to a separate machine head with its own read buffer. At the end of the step we rename our next state set to be the current state set. Should the halt state ever be placed in the current state set, then we halt the machine. Should the current state set become empty without every having reached a halt state, we say the machine has failed.

♦

Non-Deterministic (Multiplex) Procedure

```
Initialize:
1. place the input on the tape
2. for each start state place a <start state, head location on leftmost> pair in the
   current state set

Take a Step:
1. For each <state, head-location> member of the current state set:
   1.1. read the symbol instance from the corresponding head location
   1.2. execute the command corresponding to the state
   1.3. for each arc match, add a <next-state, head-location> pair into the next
        state set
2. for each arc match, add a <next-state, head-location> pair into the next state
   set
```

As reviewed by Papadimitriou, such non-deterministic machines can be converted to deterministic ones. This is done by calling each unique 'state set' a state. You might recall from our definition of symbol, that we can use any mathematical object for which instances are comparable. That includes sets. When two current state sets have the same members, we say they are instances of the same state. When they have different members we say they are different states. Hence existence proofs on non-deterministic machines will turn out the same as for deterministic ones, but the non-deterministic machines might require fewer steps to evaluate.

If we keep a trace of execution through a non-deterministic machine, and then walk backwards from each halt state, we find paths through the state machine, that, had we taken one of these paths, the machine would have reached the halt state. Had we known such a path in advance we could have saved ourselves from doing the work of following the other paths. This is why such machines are called 'non-deterministic' – we don't know which of the paths we are following while running forward will be the path that leads to the halt state first. We might arbitrarily pick a path, then we would have some probability of eventually hitting the halt state.

But notice, that the conversion algorithm going from a non-deterministic machine to a deterministic machine is forward moving. It starts by giving the initial state set a state name, and moves forward by giving each unique set of states a state name, until hitting the halt state. No stochastic variables, or "guesses" are involved. Note also, when we run a non-deterministic machine we do so in a deterministic manner. There are no stochastic variables involved.

Hence, what is non-deterministic is an interpretation rather than the machine definition or anything we are asked to do while walking through the fixed procedure. So as to avoid any confusion due to this nuance, I am going to introduce a different term for non-deterministic machines. I am going to say they are *multiplexed*. Should the need arise to emphasize that a machine is not multiplexed, I will say such a machine is *unixplex*. This new nomenclature is especially helpful when discussing the TM Library code, where the multiplexing of heads is readily apparent.

Orders of analysis

Turing Machines that halt in finite number of steps *for any finite input* are *computational.* I.e. for any given non-computational Turing Machine there will exist at least one input for which computation will never complete. By analyzing a machine we might learn that some machine, say **M**, is conditionally computational on a given set of inputs, say **I**. Then in case of the strongest delineation, computation will not halt for members of **I**, or it might be the case that we don't know about the inputs in **I**, or that **I** analysis shows that **I** can be broken into two sets, inputs for which computation will not stop for, and those for which analysis does not provide an answer.

To *analyze* a given First Turing Machine we do not execute it, but instead we place its definition on the tape as input for a second Turing Machine to analyze. We might also add on the input tape to the first Turing Machine another Turing Machine that defines the language input to the first Turing Machine, otherwise we have to assume that any input is possible. We then run this analysis Turing Machine so as to learn about the first one. (It is often the case that this second Turing Machine which performs the analysis is in a person's brain. Perhaps the person is writing a proof about the Turing Machine that is being studied.) Analysis can be applied to both computational and non-computational machines. In many cases it is possible to learn something useful about a non-computational machine, one which we cannot evaluate, by analyzing it.

The term 'analysis' can be applied more generally. Accordingly, 'first order' analysis is the same as computation. ‘Second order’ analysis is what was described in the prior paragraph. When the term 'analysis' is used without further qualification, the order is implied by context, but typically we are referring to second order analysis. It is possible to perform higher orders of analysis. Here is a question that belongs to the next higher order: does there exist a Turing Machine A that can analyze a Turing Machine B to decide if B will always halt? We are standing above looking down and asking a question about machines of the second order.

Addresses and cells

Natural Number, *address*

We can define a Turing Machine that is identical to the recursive definition of Natural Numbers as given by Peano. As the Natural Numbers never end, we cannot run this machine to a halting point, but we can analyze it. As such we can assign a Natural Number to each member of the tape sequence. We will call the machine that does this the *address machine*.

Since multiple tape Turing Machines are equivalent to a single tape Turing Machines, we begin with the simplification and say that our address machine has two tapes. One we will call the main tape. We will assign addresses to the elements on the *main tape*. The second tape we will call the *address tape*. It is initially empty.

The controller makes use of our increment routine for unary numbers which we described earlier. We begin with the main tape head on the leftmost element. The address tape reads zero, this is the address for the leftmost element. The controller then steps the main tape head right one element, and it runs the increment routine on the address tape. The address tape then reads 's', i.e. one. This is the address for the right neighbor of the leftmost element. We then repeat this procedure while assigning addresses to each successive element, two, three, four, etc. We call the set of addresses created in this manner the 'Natural Numbers'.

Though we can never run this machine to the point where it halts, we may analyze the Natural Number machine and reason about it. As one example, we can take a first given Turing Machine with it's head somewhere on a tape, encode that machine and its state, and also encode an address machine, and provide these two machines to a third machine, one called *address-of.* The address-of machine can then run the address machine while stepping the head left on the first given machine. At the point where the next left step would cause a left of leftmost error, the address will appear on the address tape of the address machine.

Distance

The distance between two elements is defined to be the difference in their corresponding addresses.

Area and length

An *area* is a subsequence of cells on a tape. An area may be defined using two addresses. The first address being that of the leftmost cell in the area, and the second address being that of the rightmost cell in the area.

The *length* of an area is one greater than the distance between the address of the leftmost cell of the area, and that of the rightmost cell of the area. Hence the smallest length that an area may have is 1. This occurs when the address to the leftmost cell is the same as the address to the rightmost cell, i.e. when there is only one cell.

The concept of area may be abstracted to include a set of cells that share a property. This might be a property of the values in the cells, or a property of the addresses of cells. An abstract area can be *colored* by a property detecting machine. Such a machine would start at the leftmost cell, check the value in that cell for the given property and mark it accordingly. It would then step right and repeat. For such a marker machine to be computational there will have to be a leftmost cell in the area, and upon finding a leftmost cell in the area, a rightmost cell will have to exist. Without either of these a marker machine will never halt. Even if a marker machine might not halt, it still might be useful for analysis. Such marker machines that color based only the value found in each cell are context free. Abstract areas may also be defined based on functions of cell values that include context.

When the cells in an abstract area are contiguous, we say that the area is *compact.* An example of an area that is not compact is that of the odd addressed cells. The odd and even addressed cells of a tape form two non-overlapping abstract areas. Given a cell in either of these areas, its direct neighbors will not be in the same area.

Zero length **is a second order concept**

Suppose we have a Turing Machine that is designed to mark an area based on some property of the symbols. Suppose further that starting with the head on the first cell, our Turing Machine will step right zero or more times until it finds a cell that holds a symbol that has the special property. Once it finds such a cell it will write an area marker symbol to that cell, step right, and repeat writing area marker cells until it finds a cell that holds a symbol that does not have the area property. At which point the machine halts.

Once an area is marked, we can go back and run a length measuring machine that counts the sequence of marks. We will call this the length of the area.

Now suppose we employ a second order analysis. Instead of running our area marker Turing Machine, we examine its definition and the definition of the machine that generated the tape data, to learn if such a machine will ever halt. Although we know that it is not possible in general to analyze machines to know if they will halt, or not, it is certainly possible to do so in some cases, and this is one of those cases. Upon analysis of our area marking machine we make a startling discovery – inputs exist for which the area marker will never halt. In one case, if there is no leftmost symbol with the property that defines the area we are looking for, the marker machine will search forever. In the second case, once the area is found, it is open on the right and it never ends.

We might say that inputs that have no leftmost symbol have an area of *zero length*. This an abstract concept, because our area marking and area measuring machines will never be run then halt and return a length measure of zero. Instead we might arrive at this conclusion that a non-existent area has zero length through inductive reasoning: Say we have an area of length i, and then remove one element from the area, then it has length i-1. As we repeat this, then eventually we will have an area of length 1 as discovered by our length assigning machine. Now we remove 1 more element. Each time we removed an element before, it made the length smaller by 1, so we reason that 1-1 = 0. The area is now length zero. We cannot go any further because there are no more elements to be removed.

In this case we arbitrarily imagined a longer area. No such longer area was stated in the problem formulation. In real world programming, this is the difference between having a container that by implication will hold elements, and not even having the container. By applying the limit logic, we have implied that an area exists. If running the area maker machine is asserting the existence of such an area, then it makes sense to speak of a zero length area. However, if we are not making this assertion, then there is no area so the issue of length is irrelevant. This is a nuanced problem.

Note, then, that zero length is a second order concept. We cannot mark nor measure it, and given our first order definition for an area, nor can we even represent such an area at a first order. There must be some external structure present that implies the existence of the area for zero length to even make sense.

This insight explains a lot of the pain related to the processing of end cases in computing. It will come up again in this volume when we examine the question of the emptiness of containers, see the section . It also explains why loops so often need to be primed or given special case guards – which is the same as adding a layer of analysis. We will discuss this further later when introducing the first-rest pattern see, .

It is interesting that non-existence has collided with zero length. This seems to be a contradiction, as something that does not exist should not have any length at all. All of this happens at the second order, at the order of analysis. At this order we can make a distinction between an area that we have given a name to, and perhaps a location, as compared to an area for which we have done neither of these things. Thus for purposes of second order analysis we will say that an area exists if it has a name or a location, even if it has zero length. We will say an area does not exist if it has neither a name or a location. Again, execution of our first order area marking and length machines cannot provide us with any such information.

Need for the concept of *cell*

Let us ask a question, what is it that an address is actually locating? Let us consider this question in the light of an example. Suppose we have the tape sequence of:

a, b , c, ε, ε, ε …

Now consider that we have an address of '2'. If we read tape address 2 we get back the letter 'c'. So the address is locating the 'c'. Now suppose we write at address 2. Say we write 'γ'. Now our sequence is:

a, b , γ, ε, ε, ε …

Now we write 'Γ', resulting in the sequence:

a, b , Γ, ε, ε, ε …

It would appear that the answer to our question is that address 2 is locating different things at different times. First it was, c, then γ, then Γ. Yet the address did not change. It feels a little unsatisfactory to suggest that our concept of location depends on the value addressed. Also, notice, that when we created the Natural Number Turing Machine, that the values on the tape that was placed into correspondence with the Natural Numbers were never mentioned. Yet, we can't seem to answer the question of 'what is being addressed' without giving a value.

Addresses speak to the structure of the tape rather than the values held on the tape. So as to facilitate this interpretation, we note that a sequence consists of a sequence of *cells* holding elements, rather than being just a sequence of elements. Given the concept of a *cell* we can say that an address always locates the same cell, though the contents of that cell can change.

This is only a small extension to the already existing concept of a variable in mathematics. In mathematics we allow that a variable can take on different values, though its name never changes. Now we are going to say that a cell can take on different values, though its address never changes. Furthermore, as the cell is part of the sequence, we are going to say the cell itself has a left neighbor or right neighbor – not the value in the cell.

Address of an *area*

The address of a cell is the number of steps required to reach the cell when starting from the leftmost cell on the tape. The leftmost cell has an address of zero. It might seem intuitive to set the address of an area on the tape to be that of the first cell in the area. If we require that an area have at least one cell to exist, this approach works even for machines with a cell delete command.

Suppose that we are deleting the cells in an area.[3] The delete command affects the cell to the right of the cell the head is on.[4] Hence to delete all the cells of an area, the head is placed on the left neighbor cell to the area. Say that we delete an area of three cells. We will call delete three times. It seems unsatisfactory to say the area no longer exists after the three deletes because the head locating the

3 For finite tape up to the rightmost alphabet symbol we can emulate the deletion of cells on the conventional Turing Machine by moving all the instances in cells on the right-hand side to their left neighbor cells.

4 The reason for this is that we can't delete the cell that the head resides on as that would break the machine. I chose this convention over other possibilities that required moving the head, as those potentially invoked an error for moving the head off of the end of the tape.

area never moved. I.e. we still have a location marker for the area, so we could, for example, call append and put a cell back into the area. For the area to truly not exist, it would not have a location. Thus it seems more satisfactory to locate an area by its left neighbor cell, than to locate it by its leftmost cell.

The inverse case also suggests that the cell to the left of an area defines its address. I.e. if we call append to grow an area, it grows to the right of the cell the head is on. According to this definition of area location, the rightmost cell locates a currently zero length right tail of the tape that will potentially be grown.

In a sense what we are doing while exploring the meaning of area, existence, and zero length with these delete and append examples is performing a discrete limit in analogy to a limiting operation in calculous. To support having such a limit operation, the location of an area is that of the left neighbor of the leftmost cell in the area.

Not all is satisfactory. When using the left neighbor of an area as the location for an area, we cannot locate an area that starts on the leftmost cell of the tape. If we make this a special case, then we have failed to create a first order definition for area, because the special case information will have to be stored in a higher level structure that describes attributes of the tape.

Nor can this use the left cell approach work in the case of multiple areas. Suppose we have two adjacent areas.

♦

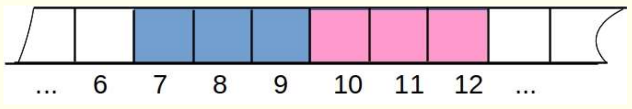


Figure 2 Tape with two areas

Here we show a first area, say a0, that includes cells 7, 8 and 9. And a second area, say a1, that includes cells 10, 11, and 12. Thus, by the convention of using the address of the cell to the left of an area as the area's address, a0 has an address of 6, while area a1 has an address of 9. Because the areas are adjacent, the address of area a1 is the same as the address of the rightmost cell in area a0.

We know that area a0 is located to the left of area a1 because a0's address is less than a1's. i.e. 6 < 9.

Suppose we delete cell 10. Though cell 10 is gone, our addresses remain consecutive, so what was cell 11, is now called cell 10, etc. and the diagram appears much as before, though area a1 is now only 2 in length:

♦

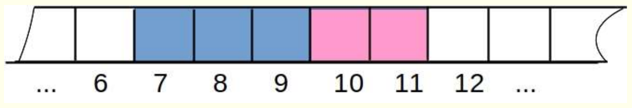


Figure 3 Step by step area two becomes shorter

Now we delete the cell at address 10 two more times, and all the area a1 is gone. During the deletion, and just after, the head will be on the rightmost cell of area a0, i.e. on cell 9. We can now say we have an area of zero length located at cell 9.

♦

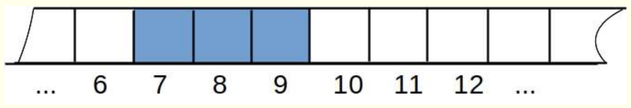


Figure 4 Second area goes to zero length

Now suppose after deleting the area a1, as just described, we continue on to delete area a0. Each time we delete the leftmost cell of a1 the rest of area a1 moves left by one. Hence, after the first deletion the address for empty area a1 becomes 8. Area a0 is still located to the left of a1, because 6 < 8:

♦

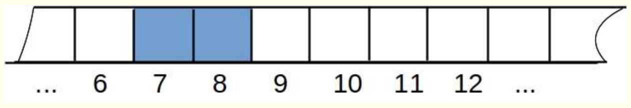


Figure 5 Limit towards zero length being applied to the first area

Finally when all cells in area a0 have been deleted, a1 has collapsed into a0. Both have the address of 6, so the order between them can no longer be derived from looking at the base addresses. Should we attempt to reverse the steps above, and only be given the machine at its final state we would have to begin by guessing what the order was between the two areas, a0 and a1.

So again, there is either missing information, or some information is not stored on the tape we are discussing. As yet another problem case that is either not possible or requires external information, if an area includes the leftmost cell of the tape, then its location cannot be described with an address.

Hence this approach of using a discrete analogy to limits has led to some unsatisfactory end cases. We will visit this subject of areas on tape in the section, **Area as a mapped tape** and present a higher order approach for defining area location that does not have such end cases. (As a small hint, we use inclusive bound address intervals placed in a symbol table stored on another tape.)

Multiple machines and sharing tapes

Suppose we unmount a tape from a halted Turing Machine, say machine T0, and then mount the tape on another Turing Machine as input, say machine T1. Suppose we do this so that machine T may calculate the length of the output created by the first machine. When we do this, we run into some problems.

Firstly, for a conventional Turing Machine, the tape that is mounted on T1 will be infinite, so no computational T1 machine will be able to process it unless there is a message on the tape telling T1 where the end of the input is. For our TTCA machines, if T0 starts with a null tape, and then expands it, and T0 is computational, then at the time T0 halts, the tape will be finite. T1 can then use the continuation of stepping beyond rightmost to know that it has processed all of its input.

If a Turing Machine does nothing then halts, it will implement an identity relationship between input and output. If we do not want the input given to a machine to 'bleed through', we will have to erase it. A computational TTCA Machine starting from a null tape can only produce finite tapes, so T1 can erase whatever T0 writes to the tape. However, if we are analyzing, instead of running the TTCA machine, we might discover that the tape length would be infinite if the machine could be run. We know that if we pass an infinite tape to a computational TTCA, it would not be able to erase the tape. Fortunately we can derive this fact through analysis and proof without having to run the computational machine to see if it erased the tape. Hence if we allow for infinite inputs, we should add an 'erase-to-end-of-tape' command to our machines, so that the computational machines may leave a tape with only their outputs on them. Our erase-to-end-of-tape command will be executed in a single step.

Once an input tape is mounted, the machine must

1. T0 is a computational TTCA machine given a finite tape, its output is a finite tape, and that is taken to be the input for T1.
2. T0 is required to provide meta information about the length of the active area. This approach is known as *in-band signaling*. Because length information is mixed with the data.
3. T0 updates a dedicated table where it keeps track of the location of data objects and their lengths. Such a table is called a *symbol table*, and such a system is called a *type system*. This is written to a separate tape, or it is inserted at a known location on the main tape. In the case it is written to the front of the tape, the data following will have to be moved when the table grows. In the case it is written at the end of the tape, the table will have to be moved when the data grows. In either case the symbol table entry lengths themselves must also be managed, typically they are either coded into the controller or make use of in band signaling. It is conceivable that a symbol table would describe itself.
4. We adopt a convention of maintaining a *compact* tape. As such we have no embedded empty-symbols in the active area. Then the empty-symbol marks the end of the active area.
5. Each machine has two tapes, an input tape, and an output tape.

A *compact* area is one that has only alphabet symbols (no empty-symbols). We can extend this concept to say that the *density* of an area is the ratio of alphabet symbols to empty-symbols.

The approach we use in modern computing is often the 3nd one listed above, that of the type system. We carefully account for the length of each *instance* of data. Then we build up each larger instance from smaller ones, and while doing so, we add the lengths of the smaller instances to calculate the length of the larger instance. All programs then specify when they create or compose instances and their types.

A system for creating the appearance of many tapes over a single tape is called a memory manager or a memory allocation system. It is typically better to use one of these and multiple tape models rather than solving the problem of moving objects around in memory in an ad hoc manner.

Here is an important question for the TTCA model: is it possible to create a memory manager that creates the appearance of multiple expanding tapes over the top of a single expandable tape? This problem is solved for managing files on a hard disk. However, all current file systems break at some point when pumped. So the question being asked here is equivalent to asking if a files system can be made without such architectural limits. The answer to this is yes, it is possible, as we will see in a later chapter of this book.

Hence there are multiple conventions we may use for implementing the abstraction of multiple tapes on a tape machine. Multiple tape machine are equivalent in power to one tape machines, but sometimes they are easier to think about.

Now as we have multiple tapes it is fairly easy to show that having multiple tape machines is equivalent to having one machine. We first view the multiple tapes worked on by the multiple machines as the multiple tapes of one machine. We then compose the state controllers in series, where the halt state of one is the initial state of the next one. Thus we may also conclude that having multiple tape machines is not more expressive than having one tape machine. We may also look at our multiple composed state controllers as one multiplexed controller, and then conclude that having multiple machines will not have an order of speed or space usage advantage.

The advantage of having multiple tape machines each perhaps having multiple tapes is that sometimes it is easier to think about. It is a method of partitioning the problem.

Suppose our machines have two tapes where one is called the *input tape*, and the other tape the *output tape*. Our procedure for passing tapes between machines will consist of *umounting* the tape from the source machine, then taking that tape over to the destination machine and *mounting* it as the input tape.

Some properties of Turing Machines

For a given Turing Machine, the input is the sole determiner of the output. I.e. each time the same input is given, we get the same output. The input and output are mathematical objects, hence Turing Machines are functions. However, it is common in computer science to speak of Turing Machines as 'solving problems' rather than saying they are functions. This is because we often think of the inputs and outputs of Turing Machines as being something other than mathematical objects. For example, when a Turing Machine sorts sequences found on its input, we might say that it solves a sorting problem.

Turing Machines will differ due to differing alphabets, state controller graphs, associated commands, start, and halt states. The choice of empty-symbol is inconsequential as long as it is distinct from the alphabet. As we will see in the later discussion on variations, , the choice of alphabet is not very important. For two alphabets of the same cardinality we can create a one to one mapping, and for those of differing cardinalities we can use sequences of alphabet symbols that map to alphabet symbols. For example, given an alphabet of ‘T’ and ‘F’, and a second alphabet of ’t’, ‘f’, ‘x’, ‘z’ we may make the following map:

♦

f	FF
t	FT
x	TF
z	FF

Then given this mapping, we may use two cells for each one cell for any {‘f’, ‘t’, ‘x’, ‘z’} alphabet machine, and then use the only the {‘T’, ‘F’} alphabet.

There are a countably infinite number of permutations for alphabets, state controller graphs, associated commands, start and halt states, hence there are a countably infinite number of Turing Machines that fit our definition. However, there are an uncountably infinite number of mathematical functions. Consequently, we must expect that some functions cannot be computed with Turing Machines.

There might be multiple Turing Machines that perform the same function. A set of such machines forms a functional equivalence class. Within a functional class there will be a class of members related in that they all use the smallest number of steps when considered against the limit of input length. We discuss this further in the section on complexity, .

Of special importance to computation theory is the existence among all these infinite Turing Machines of a class of machines that read their state controller definition from the tape as an input. This is the *Universal Turing Machine* class.

Performance Analysis

An interesting aspect of the Turing Machine procedure is that it introduces the concept of *stepping the machine*. With the addition of some simple constraints it becomes possible to map the parts of the Turing Machine abstraction to the parts of some real machines. These constraints may take the form of such things as bounds on the length of the inputs, or the addition of out-of-resource errors. Because such constraints do not affect the 'normal' workings of the machine, the derived relationship between a Turing Machine step, and that of a unit of real time might not be that complicated. Indeed, except for some enumerable cases this relationship might even be so simple as to assign to a step an approximately constant amount of time. Because of the existence of a relationship between steps and time, particularly when it is a simple one, it is very interesting for us to know how many steps a Turing Machine will take.

There are many ways to measure the complexity of a Turing Machine. Among these is something called the *time complexity,* which is a function that relates the length of the input to the number of steps required to reach the halt state. To derive time complexity we typically start with a step count formula which maps the length of Turing Machine input to the worst case largest number of steps. We then consider the behavior of this formula as input length goes towards infinity. To get this, we take the highest order term from the step count formula. Conventional results are constant time, polynomial time, or exponential time.

We can derive the 'worst case length of the area written or read by the machine during computation' function in an analogous manner as for the step count function. This function is known as the *space complexity*. We may also consider the limiting behavior of this function to derive an order of space complexity.

The order of time or space complexity will remain the same against certain variations of our Turing Machine definition. For example, if we double all the states by adding a second state that we always visit, where this second state does nothing, the number of steps would double but the functionality would not change. Order of complexity also would not change. A fixed time machine before doubling up on the states would still be a fixed time machine afterward. It is just that the number of steps would be twice as large, but still a fixed number relative to the size of the input. A polynomial time machine would still be polynomial time, just with double size constants. We say that changes which do not change order of complexity, nor existence proofs, are *inconsequential*.

Suppose we have a complete Turing Machine functionality class. We say that it is *complete* because all possible machines for implementing the function are in this class. Some machines in this class will have a different order of time complexity than others. Now we consider the set of minimum order of time complexity machines from this class. As the larger set was complete, the set built against this constraint will also be complete relative to the constraint. We then say that this minimum order of time complexity is a property of the problem being solved, rather than being a property of a particular machine.

Conventional Turing Machine variations

In the first section of this chapter we gave a rather conventional definition for a Turing Machine. In the prior section, Complexity, we noted that we can analyze Turing Machines to find their time and space complexities. In this section we will discuss some variations that one finds in the literature.

A variation on the conventional Turing Machine definition is allowed when it can be proven that the variation never causes existence, order of time complexity, nor order of space complexity results to change, and in this respect is *inconsequential*. Earlier we gave the example of doubling up the states as being such a variation, though that is not a conventional variation.

Open in both directions tapes

Some Turing Machine descriptions describe a tape with no end in either the left or right directions, i.e. a tape that is that is *open* in both directions.

This feature adds no richness of expression, because we can get the same behavior from a Turing Machine with a single ended tape. To do this we partition the single ended tape into odd addressed cells and even addressed cells. The odd cells are said to be the right side of the tape, and the even ones the left side. We then rewrite any tape controller based on a bidirectional tape to instead use the 'odd' and 'even' channels instead of the left and right sides of the tape.

This same approach form can be used to show that multiple tapes, or even multi-dimensional tapes, add no expressive power. The good news is that such variations can be used whenever convenient, and we will get the same results.

Going in the other direction, the open in both directions tape is not a simplification. There is still a start cell, being the cell that the head is initially placed on. And as noted above, the topology around this start cell is no different, it is just a question of the adjectives we use for describing it.

Alphabet replacement

Without loss of generality, we may replace the alphabet with a single symbol, say 's' (short for successor). This is because symbols in any alphabet can be placed into correspondence with a sequences of 's' symbols. For example, the symbols of the alphabet of {w, x, y, z} can be placed into one to one correspondence with the sequences in the manner of {<w, s>, <x, ss>, <y, sss>, <z, ssss>}. We will need to put the empty-symbol between any such sequences on the tape, so that two sequences can be distinguished from one longer sequence. Alternatively we can adopt a two symbol alphabet instead of a single symbol, where the second alphabet symbol is an end of sequence marker.

In contemporary computing we use an alphabet of two symbols, {0, 1}, and fixed length sequences. No end marker is needed when the sequences to be placed into correspondence are of fixed length. So for example, the symbols in the alphabet {dog, cat, mouse, fish} may have the correspondence of {<dog, 00>, <cat, 01>, <mouse, 10>, <fish 11>}. Conventional fixed sequence lengths are 8, 16, 32, and 64. So for example, when the sequence length is 8, any alphabet of 256 symbols or less may be placed into correspondence. A conventional correspondence table is that of the ASCII code. The fixed length to be used can depend on computational context. (In contrast, UTF8 does not use fixed length sequences, so there must exist at least one end of sequence marker.)

Another alternative to end of sequence markers for variable length sequences is to externally account for sequences lengths. We call such an accounting system a *type* system.

Leaving out the empty-symbol

When we use a fixed sequence of {1, 0} to stand for symbols, it is expensive to reserve a sequence for the empty-symbol. This expense is due both to losing the use of a symbol in the alphabet, and in the complexity of control circuitry when keeping track of it.

The empty-symbol is a property of the machine rather than just another alphabet member, because the tape initially has an infinite tail of empty-symbols. A computational Turing Machine is limited to taking a finite number of steps. Thus it cannot compute a tape initialized with an empty-symbol (or any other value).

However we can add a constraint on all Turing Machine controllers that requires of controllers that they never write the empty-symbol, and always write an alphabet symbol to a cell before reading it. Then, because a cell is never read before being written, it does not matter what we write into it for initialization. We may even use an alphabet symbol. Consequently this constraint allows us to eliminate the empty-symbol. This gives us the following Turing Machine variation:

1. fixed finite alphabet
2. fixed write before read constraint
3. variable single ended tape
4. variable read/write head
5. variable read value buffer
6. fixed controlling state machine
7. fixed left from leftmost error
8. fixed start state
9. fixed halt state
10. variable current state

11. fixed procedure for using these.

To validate that this is an inconsequential Turing Machine variation, rather than a description of new abstraction that is not a Turing Machine, we must show two things: Firstly, that any of the now disallowed controllers never compute something that cannot be computed in the presence of the constraint. And secondly, that there are same complexity class alternatives for any disallowed controller.

No need to step into the inactive area

1. When attempting to step into the inactive area, instead keep a counter for the number of steps the machine would take. Only allow reads or writes or head movement when the counter is no longer needed due to the head having moved back into the active area. 2. Just write an alphabet character and change the inactive area traversed into an active area.

The TTCA Turing Machine variation

The active area on the tape can grow at most by one unit for each machine step. This largest growth occurs when the Turing Machine steps right and writes an alphabet character in every visited state. This means that for computational machines that start with a tape that has a finite input (active area), the output (active area) will be finite. This also means that space complexity can never be larger than time complexity.

A *fixed value* is one that is provided with a Turing Machine definition, and does not change while the machine runs. Suppose we chose a fixed length Turing Machine tape. The tape would then have a rightmost cell. That cell would have no right neighbor, but would have a left neighbor. We would also add another error, that of *right from rightmost*. This error would be invoked when the controller attempted to step right from the rightmost cell.

Consider a machine that does not step out of the active area, has constant space complexity, and where this space complexity is less than the fixed length for the finite tape – such a machine would never trip the right from rightmost error, and thus there would be no difference between a finite tape and an infinite one.

Now suppose that we bound the length of the input, and that the maximum space required for such inputs or shorter ones is less than or equal to the length of the tape. Then again, the right from rightmost error would never be taken, and thus the tape would be indistinguishable from an infinite one. (Today we typically pad programs with lots of memory and long address words in hopes this will be the case.) An analogous argument can be made if we bound the number of steps that may be taken.

Now consider the case where we do not fix the length of the input, nor the number of steps allowed, and that space complexity is such that space usage grows with growing input length, at least for very long inputs. For such machines we can always find an input of sufficient length to trip the right from rightmost error.

As another approach to finite computing we can run computations twice. For a given input we first run the Turing Machine variation that does not step out of active area but still has an infinite tape. We watch this machine closely while it is running and find the bound on the active area. Now we can create a second machine that has a fixed length tape at least as long as our active area measurement but is otherwise the same. Now with this second machine we can run the same input and there will be no right of rightmost error, and thus there will be no difference between having the finite tape or an infinite one.

Unlike for the constant space complexity proposal, and the bounded input length proposal, which only work for small subsets of potential inputs, this 'run twice' proposal derives a finite machine that works for any given input which a Turing Machine works for. Though, unfortunately, in all cases the second run will be moot, as we could have simply taken the output from the first run.

In a variation on the run it twice approach, instead of running the first machine, we might instead analyze it and should we be able to surmise a maximum tape length, we could use that.

In yet another approach we can extend the finite tape as needed. We create a control layer over the finite tape. When a step right command from the Turing controller invokes the right from rightmost error, the lower layer allocates memory, lengthens the tape, and then performs the requested step right. As long as this occurs in fixed time, (or of sufficient lesser order time than the dominate order of the time complexity), and as long as there is indeed more memory to allocate - this Turing Machine variation will yield the same order of computational complexity as one with an an infinite tape.

I propose the following Turing Machine variation:

1. fixed finite alphabet
2. write before read constraint
3. variable and extendable finite tape
4. variable read/write head
5. variable read value buffer
6. fixed controlling state machine
7. fixed left from leftmost error
8. fixed right from rightmost error
9. fixed start state
10. fixed halt state
11. variable current state
12. fixed procedure for using these

In our original Turing Machine model, the controlling state machine commands were limited to, **`do-nothing`**, **`step-left`**, **`step-right`**, **`write`**, with reading as an implied command. To this list we add **`append`**. The **`append`** command may only be called when the head is on the rightmost tape cell. This is not limiting because the command may be called from a state that is at the end of an arc triggered by the *right from rightmost* error. When we have no empty-symbol, **`append`** accepts an alphabet symbol and performs a write into the new cell. This is not limiting because if need be, one can always perform an extraneous write of an alphabet symbol.

With this extendable tape model all Turing Machine components remain finite during computation, though some may be arbitrarily large. This variation is more suited for creating a mapping between a Turing Machine and a real program running on a real machine.

More about commands

The Turing Machine state controller has a command symbol tied to each state. The Turing Machine procedure then has us take action based on this symbol. This is our current command set:

1. `step-right`, causes the head to move to the right neighbor cell.
2. `step-left`, causes the head to move to the left neighbor cell.
3. `read`, returns the symbol under the head. The returned symbol is then used to chose the transition arc.
4. `write(x)`, causes the symbol *x* to be written to the cell that the head is currently on. The symbol *x* is any symbol from the alphabet.
5. `append(x)` may only be called when on the rightmost cell. Extends the tape by one cell, and writes the symbol *x* into that cell, where *x* is any symbol from the alphabet.

We are going to relieve the constraint that append can only occur from rightmost. Our new append is functionally identical to adding a cell to the rightmost extremity, and then shifting all the symbols over by one cell starting at the new cell and ending when the new rightmost has been written - and then doing the requested write of *x* on the right neighbor cell.

We will also include the inverse function for append. delete(append(x)) reads x while deleting the cell that x was in. The current Turing Machine model can emulate this function by shifting all the symbols in cells the right of the head left by one, and then just not using the rightmost cell.

We are also going to support multiplexed state controllers. Our multiplexed Turing Machine will have multiple heads. One for each separate thread of execution through the state controller. As explained in the following chapters, supporting multiplexing makes our machine more complex, especially in the presence of the delete command. However, we defer that discussion to the relevant chapters.

In cases where successive states are visited in a fixed order it is convenient to combine the commands. We have developed the concept of a command statement to support this:

♦
```
statement::[direction]command+[modifier][&contract]*[arg]*
direction::- | ε
command::r | w | s | a | d | m | e | ♀
modifier::◧ ◨ n
command
      r  read cell under the head
      w write cell under the head
      s  step
      a allocate/add/append a new cell
      d deallocate/drop/delete cell
      m move, no allocation or deallocation of cells, requires fill
      e entangled copy
      ♀  entangled copy on a new thread
modifier
      ◧ operate on leftmost
      ◨ operate on rightmost
      n  repeats n times, n provided through an argument
contracts
      h◧ head is at leftmost
      h◨ head is at rightmost
examples
      -s   ; step left
      a◧  ; creates a new leftmost cell
      a◨  ; append to rightmost
      sn  ; step n times
to derive a longer command, combine them:
      as   ; append then step
      -a-s  ; append to the left, step to the left
```

The left direction is specified with a minus sign, otherwise the direction is taken as right going. So the letter `s` is the step-right command, and `-s` is the step-left command. The command `s3` steps right three times.

The command `a` appends and writes a new cell to the right of the head. We use two special characters from the UTF character set to signify the rightmost and leftmost of the tape. This one looks like a little tape with its left cell inked in, ◧, so we use it to stand for leftmost. We use this one, ◨ ,to mean rightmost. Hence a◨ creates a new rightmost cell, and a◧ creates a new leftmost cell.

In some cases it is possible to implement higher performance implementations for commands when the programmer tells us some additional information. For example `a◨&h◨` has identical functionality as `a◨`, while the programmer also guarantees that we are on the rightmost cell. This saves the function from having to scan to the end of the tape.

We can concatenate the command letters into a string to summarize what would happen sequentially in adjacent state transitions. If these compound commands need arguments, then they are pulled from the argument list in order as they are needed. For example, as means to append, with the parameter for the append taken from the argument list, and then to step.

We support multiplexing with the command `e`, which is short for *entangled*. This operator returns what appears to be a second independent machine, but this apparently independent second machine actually shares the same tape with the first machine. It is functionally identical to giving one machine two heads, and thus the ability to have state sets.

The command `esr` is a compound command referring to sequentially applying three other commands. The `e` says to make an entangled copy of the head. The `s` says to step this copy, and the `r` says to do the read. The analogous `esw` does a write as the last step. This sort of combination of letters to make more complex commands was inspired by Lisp's `car` and `cdr` compositions.[5] Though this is functionally what the command does, its actual implementation might be completely different.

The combination of multiplexing and cell deletion creates the hazard where one thread can delete a cell the head is on in another thread. We add a *collision* error continuation to our multiplexed Turing Machine interface just because of this situation.

[5] Lynch, Thomas W. “Towards a Better Understanding of CAR, CDR, CADR and the Others” arXiv:1507.05956 2015-07-25

Chapter conclusion

The modifications made to the Turing Machine to create our TTCA machine were inconsequential, in that for order of complexity and existence proofs we may swap one machine for the other and the results will be the same.

The original Turing Machine had an infinite tape. In contrast the TTCA machine has a surprising property: for computational problems all of its components remain finite. This follows from the fact that during computation a machine makes a finite number of steps, so the tape can only be expanded to be a finite size.

TTCA Turing Machine in Lisp

Because our TTCA Turing Machine has finite sized components, we may create a software model for the TTCA Turing Machine without having to make assumptions of the sort that 'very large approximates the infinite'. Rather we can show a one-to-one mapping of code and data in the software model and the TTCA variation of the Turing Machine. Consequently, the software model and theoretic model are isomorphic.

It follows that we can use our TTCA Turing Machine software to make theoretical statements about computation in the form of programs. By following this path we will learn some interesting things in this chapter about *analysis*, the meaning of *emptiness*, *data type*, the properties of *non-destructive* vs *destructive* programming styles, and *multi-threaded* programming among other things.

The Turing Machine, and our variation of it, may be partitioned into two parts. One part of the part consists of the tape head and the tape. We call this the Tape Transport Unit, as that is the name used for the mechanism that does this work on real tape storage units. The other part of the part is the Controller.

The Tape Transport Unit accepts commands for reading, writing, and moving the head. (In real Tape Transport Units the head is in a fixed position and we move the tape, but the relative affect is the same.) Our modified model adds commands for extending the tape. During normal operation these commands only come from the controller.

We have two types of controllers. One type of controller is a state machine. Its design is an integral part of the Turing Machine. To step the Turing Machine means to step this state machine to its next state. The state machine definition exists before the Turing Machine takes its first step, and its definition remains intact for as long as said Turing Machine exists. When we speak of a Turing Machine without adding further words to the term as qualifiers, we mean that it uses this type of controller. For clarity we can call this a Directly Controlled Turing Machine.

The second type of controller is the Universal Controller, and a Turing Machine that uses this this type of controller is called a Universal Turing Machine. The Universal Controller reads the definition of a state machine controller off of the tape. Hence the Universal Turing Machine may emulate any Directly Controlled Turing Machine. We can also call this an Indirectly Controlled Turing Machine. Here the qualifier 'Indirectly' is intended in the sense it is used in assembly languages, meaning the data is not present in the controller, but rather it must be fetched from memory.

With our Tape Machine library, your program is the controller, and the library implements the Tape Transport Unit. (In the future I will rename this library to 'TTU' from the current 'TM'. Also the use of the term 'function' will be changed to 'routine'.)

Your program that uses the TTU library is in a sense direct control, because the program is already defined before the first step of the machine, and with the possible exception of self-modifying code, it does not change until the program exits, and thus the process no longer exists. On the other hand, your program is loaded from memory by the processor, thus demonstrating that the processor is an indirect, i.e. universal, controller. Either interpretation works depending on how broad of a view we want to take.

Installing the library

TM is presented as an iteration library on the de facto package manager for Lisp `quicklisp`. Alternatively, a person may clone the repository `www.github.com/Thomas-Walker-Lynch/tm` and then checkout the latest release tag, which as of this writing is `v0.7-alpha`. After installing the code `cd` into the `tm` directory and run your lisp interpreter. Inside your lisp interpreter type the commands `(load “load”)` and `(test-all)`. `test-all` should return with a message that all of the tests passed. It is possible that the threading tests, '`ts1-`' might fail if your machine is heavily loaded or very slow, as they have timing built into them, but this is unlikely. Then type either `(use-package :tm)` or `(in-package :tm)` depending on what your objectives are.

The examples in this chapter either come from the tm/test directories, or from the tm/docs/examples directory. At the time they were placed in the book, they executed correctly, and I have endeavored to keep the examples up to date.

This is what it looks like when I follow install using the `git clone` method:

♦
```
> git clone http://www.github.com/Thomas-Walker-Lynch/tm
Cloning into 'tm'...
remote:Counting objects: 3052, done.
remote:Total 3052(delta 0),reused 0(delta 0),pack-reused 3052
Receiving objects: 100% (3052/3052),2.41 MiB|1.52 MiB/s,done.
Resolving deltas: 100% (2366/2366),done.
> cd tm
> git tag
v0.1-alpha
v0.7-alpha
> git checkout v0.7-alpha
Note: checking out 'v0.7-alpha'.
You are in 'detached HEAD' state ...
> sbcl
This is SBCL 1.3.14.debian ...
* (load "load")
; compiling file "/home/tm/package-def/conditions.lisp"
; compiling (IN-PACKAGE #:TM)
  ... about 10 pages of these
hooking test: TEST-TS1-5
T
* (test-all)
     + TEST-UNWRAP-0
... about a page of these tests
     + TEST-TS1-5
all 78 passed
T
* (use-package :tm)
T
* (≠ 1 0)
T
*
```

Notice I used `git tag` to see the releases. At this time, `v0.7-alpha` is the latest, so I checked that out. If you want the unstable latest code rather than the stable latest release, leave out the `git checkout` command.

The TM Library makes use of Unicode. There is no getting around it. This is discussed further in the next section. In addition TM defines synonyms for commands such as 'not equal', which is the one command shown at the end of the transcript given above.

Unicode usage

For your convenience there is a file "emacs-keys" in the docs directory of the distribution. It sets the

```
C-x g name SPC
```

command to enter one of the Unicode characters that are used in the library. Here 'name' is a nickname.[6]

So to type the character *capital delta* after emacs-keys has been loaded, type `C-x g D SPC`. Actually `Δ` occurs twice in Unicode, once as capital delta, and once as a symbol for 'increment' in mathematics. We consider the increment version to only be there for typography purposes. We only use capital delta, even when it is for an increment variable.

I've limited the use of Unicode mostly to things that 'probably would have been this way had Unicode been around before'. This includes conventional notation and a couple of symbol extensions that were needed to facilitate the TM access language.

In the file `src-0/fundamental.lisp` find synonym bindings for the usual operators and common symbols such as `∧, ∨, ≥, ≤, λ, ∅,` etc.

[6] I would have preferred to have modified the existing C-x 8 so that such a nickname followed by a space would enter a character, but when delving into the emacs library I found it to be rather 'Unicode technical'. Perhaps someone can help with this.

Specific to the library we use the character '◧' as shorthand for 'leftmost'. This is because it looks like a little tape with the inked over cell being the leftmost cell. In the same manner the character '◨' is shorthand for rightmost. We use '➜' in continuation function names, and '⟳' as a loop operator.

The ◧ and ◨ symbols are used in compound command names and for access language statements to indicate operation on rightmost or leftmost rather than the cell the head is on, or, to specify contracts with the programmer of the sort: "this function is only called when the cell is on rightmost."

Since I had symbols for leftmost and rightmost, I started using them generally to mean leftmost or rightmost wherever it was convenient. For example, to shorten up the names of continuations so that parameter lists would fit on a line.

Synonyms

♦

```
(defmacro defsynonym (old-name new-name)
 "Define OLD-NAME to be equivalent to NEW-NAME."
 `(defmacro ,new-name (&rest args) `(,',old-name ,@args))
 )

 (defconstant ∅ nil)

 (defsynonym /= ≠)
 (defsynonym <= ≤)
 (defsynonym >= ≥)

 (defsynonym not ¬)
 (defsynonym and ∧)
 (defsynonym or ∨)

 (defsynonym string/= string≠)
 (defsynonym string<= string≤)
 (defsynonym string>= string≥)

 (defsynonym lambda λ)
```

Some reader macros

q – a non literal quote

In Lisp a quoted list is taken as being literal. However the result of modifying a literal is undefined, and often leads to bad results. Hence we provide the macro `q` which returns a quoted list which is not a literal.

```
* (q a b c)
  (A B C)
```

{…} - unevaluated list

When a form enclosed in parentheses, (...), is evaluated the head is taken as the name of a function, looked up and called. The list members are also evaluated, and then passed as arguments to said function. If we don't want the head treated specially, but rather just want to define a list, we can use a front item of `#'list`, which is the function to create a list.

```
* (list 1 2 (+ 1 2))
  (1 2 3)
```

We have defined a macro called `L` that like `#'list`, creates a list, but which also has some extra functionality.

```
* (L 1 2 (+ 1 2))
  (1 2 3)
```

We also provide a reader macro for `L` as braces.

```
* {1 2 (+ 1 2)}
  (1 2 3)
```

If the apparent function open, `#'o`, appears inside of a call to `L`, then the arguments of the `#'o` function are included directly in the resulting list:

```
* (defvar a {1 2 3})
  A

  * (defvar b {4 5 6})
  B

  * {a (o b)}
  ((1 2 3) 4 5 6)
```

Quoted non-literals can also occur within such an `L` list:

```
    * {a (o b) (q a b)}
       ((1 2 3) 4 5 6 (A B))

       * {a (o b) (o (q a b))}
       ((1 2 3) 4 5 6 A B)
```

`L` is a little bit like quasiquote turned inside out. Whereas the default in quasiquote is to quote items, and a comma operator turns that off, the default in `L` is to evaluate items, and a `q` operator turns that off. Quasiquote has an `@` marker to open up lists, while `L` has an `o` operator to open up lists. Inside of a quasiquote we could get in trouble if the name of a variable starts with an `@` character; if such a variable appears after a comma, quasi quote will consider the variable name without the `@` sign is to be opened and included. There is no analogous problem with the `o` operator because it only appears in the function channel. (The problem with quasiquote is that it uses 'in-band signaling' which mixes control with data in one channel.).

[...] - head is variable holding a function name

In Lisp the head of an evaluated list is taken as a function name. Consider this example that curries a two parameter function into a unary function by replacing one parameter with a constant argument of `3`:[7]

```
(defun curry-three (f n) (f n 3)) ; has errors
```

When we compile this function we get two errors:

```
The variable F is defined but never used.
```

and

```
undefined function: F
```

The first `f` is in the parameter list of the function definition, so it is taken as a variable name. In contrast the `f` in the body is at the head of an evaluated list,

[7] curry example is expanded from that given in "The Common Lisp Cookbook" see http://cl-cookbook.sourceforge.net/functions.html

so it is taken as a function name. Hence there is a disconnect, and we get error messages describing this disconnect.

The Lisp operator `#'` indicates that the symbol that follows is a function name to be taken literally, and not a variable name. This gets our function name into the data space for use as an argument. The Lisp function `funcall` accepts as a first argument the name of a function to be called, while the remaining arguments are passed through to said function as its arguments:

♦

```
(defun curry-three (f n) (funcall f n 3))

(defun plus (x y) (+ x y))

(curry-three #'plus 2)

5
```

In the definition for `curry-three` we do not get interpreter/compiler errors. This is because `f` is consistently used as a variable name. `funcall` will use the value of the variable `f` as the name of a function to call.

In the second line we define a function to pass into curry-three. I just put something simple here for sake of discussion. `plus` is defined to be a function that takes two arguments and sums them.

In the third line we use the `#'` operator to tell Lisp that `plus` is a function name to be used literally as a value.[8] This will be a value passed into `curry-three` no differently than had we put a number or string instance as an argument. Inside of `curry-three` the `#'plus` becomes the value of the variable f. Then the `funcall` function will use this value as a function name, and then call it.

This is how function pointers are handled in Lisp as it does not have an explicit pointer type.

[8] An implementation might use a different type of reference to the function than the functions name, such as a pointer to an entry in a symbol table, or perhaps a pointer to the function. All that is required is that #' produces a value that can be used by funcall.

Actually we didn't need to define the function `plus`, because '+' is already a function. We don't have reserved operator symbols in Lisp, instead we just have loose rules on what can be used for function names.

```
(curry-three #'+ 2)

5
```

We introduce a shortcut with the TM Library. Normally a list to be evaluated is in parenthesis, and its head is taken literally as a function name. With the TM Library loaded, when a list in square brackets is evaluated, the head is taken as a variable name, and the value of this variables is the function to be loaded. It is a nice coincidence that square brackets mean indirect addressing in many assembly languages.

We implemented this feature with a reader macro which simply turns the square bracket list into a regular list and inserts the `funcall` as its head. This occurs before the Lisp evaluate phase sees the syntax.

Using the square brackets we may define `curry-three` as:

```
* (defun curry-three (f n) [f n 3])

CURRY-THREE

* (curry-three #'plus 2)

5
```

Here is another example. Suppose that instead of passing `#'plus` in as an argument, that we first assigned it to a variable and then pass the variable value as an argument:

```
* (defvar our-fun #'plus)

* [our-fun 4 7]

11

* (curry-three our-fun 2)

5
```

Here the value assigned to the variable **`our-fun`** is a function name. The variable is then used as any other, and its value is passed in as an argument to **`curry-three`**. Note, all arguments are evaluated before the function is called, so variables are replaced with their values. And as we know, inside **`curry-three`**, **`funcall`** will take the function name value from the corresponding parameter and call it as a function.

Things can become a little confusing when the variable name has the same name as the function.

```
* (defvar plus #'plus)
* (curry-three plus 2)
5
```

Here **`#'plus`** is the function name as data. **`plus`** when it appears as the head of an evaluated list is a function name, otherwise it is a variable name. It is little wonder that the Lisp dialect Scheme put function names and variables name in the same space. However by doing so they had to provide some automatic conversions between variable names and function names.

Summary

When evaluated:

`(q f g h)` – non-literal list of symbol literals, **`f`**, **`g`**, and **`h`**. Unlike for quote, it is safe to modify the resulting list.

`(o f g h)` – not a list, rather **`f`**, **`g`**, and **`h`** are to be included in a **`{}`** form that contains this form. E.g. **`{ (o 1 2 3) }`** is the same as **`{1 2 3}`**.

`{f g h}` – list. **`f`**, **`g`**, and **`h`** are variables that are replaced by their values.

`(f g h)` - function call. **`f`** is literally the function name. **`g`** and **`h`** are variables that are replaced by their values, and then passed as arguments. The result of the function replaces the entire form.

`[f g h]` – indirect function call. `f` is a variable replaced by its value, and that value is then looked up and called as a function. `g` and `h` are variables replaced by their values and passed as arguments. The result of the function replaces the entire form.

Continuations

In the early 1960s many of the programming concepts we take for granted today were not yet solidified. Many programmers wrote what we call 'spaghetti code'. When programmers wanted to branch to some code shared by multiple parts of the program, they might add code to branch there directly. It only made sense to keep the address for continuing the computation afterward in a register. With the advent of Algol the conventions for calling such reused segments of code, *subroutines*, were formalized. The method call that begins by pushing a stack frame with arguments, an allocation for local variables, and the address to return to after the subroutine completes has become ubiquitous to computing.

An alternative method for calling functions is to pass the continuation points into the function as arguments. The function itself then decides which continuation to follow. For this to work the continuation will also need to carry along with it some memory context.

In the 1993 paper, "The Discoveries of Continuations", DOI 10.1007/bf01019459, John C Reynolds explores the origins of the continuation method. In our computing epoch Andrew W. Appel has written extensively on compiling with continuations. He and his colleagues developed compilation techniques and wrote a compiler. See "Compiling with Continuations", ISBN 978-0521033114.

The TTCA supports the continuations call model. This approach is introduced in this chapter and further developed in later chapters.

Concept

Conventionally programmers think of programs as linear sequences of instructions, and any disruptions being temporary. As examples, an if block is considered to be a temporary election between two linear sequences that both lead to a common point at the end of the block. Calls to subroutines are also temporary diversions, as execution will continue at the instruction after the call.

In the TM Library we move to a non-linear model of function calling by passing as arguments multiple possible continuations. When a given function reaches a completion point, instead of returning, it calls one of the passed in continuation functions. A function may also call continuation functions in a non-blocking manner to launch threads.

Here is a silly read function gasket that either returns a result or a status code. The method of returning a status code is a fairly common approach to error processing. Sometimes the status code is a null pointer flat, and sometimes it is an integer value.

♦

```
(defun mock-read (n)
 (if (= n 0)  ; zero marks end of stream
  'eos
  (if (oddp n)  ; say the lsb is a parity bit
   'parity
   (write-to-string n);convert n to a string and return it
   )))
```

After calling `mock-read` the caller will check the returned status code. If all is OK, the program will use the result.

♦
```
(defun use-mock-read (n)
 (let(
    (result (mock-read n))
    )
  (case result ;unwind the status code
   (eos (print "end of stream"))
   (parity (print "parity error"))
   (otherwise (print result))
   )
  t
  ))
```
Notice that we are checking for the same error condition twice. The program does this first inside the function to decide to return an error code, and then later the caller checks the status code.

We might have instead used throw and catch. Typically we would still have a redundant structure for pairing up the catch. Generally the overhead of throw and catch is too high for them to be used as part of the main computation thread. They are intended to be used only for handling exceptional conditions. Not all non-zero status codes are so exceptional.

If we could have multiple exits, status codes and redundant tests would not be necessary and the code would be easier to read. Here is a new version of **`mock-read`** where a list of *continuation functions* are passed in as a list:

♦
```
(defun use-mock-read-with-continuations (n)
 (mock-read-with-continuations n
  {
   :➜ok     (λ(result) (print result))
   :➜eos    (λ() (print "end of stream") t)
   :➜parity (λ() (print "parity error") t)
   }))
```

I have given this list that holds the continuation functions the one character long name of ➜. When speaking in the context of parameter passing, I will refer to this as the 'continuation' character. This is not an operator, rather it is just a one character variable name.

I have used destructuring-bind to parse the continuations list. `destructuring-bind` is the same function that is used by eval to bind arguments to parameters. Consequently we use the same syntax here as for specifying function parameters, and `&key` gives us keyword arguments, etc. Keywords are useful for function arguments because lambda function arguments can get quite long.

In this example, the continuations list is used to pass three arguments, which are bound to the parameters `➜ok`, `➜eos`, and `➜parity`. In this example is not necessary to provide continuations because `#'echo`, `(be 'eos)` and `(be 'parity)` have been set as defaults.

By adopting the convention of prefixing a '➜' onto continuation function names we can easily recognize that they are function continuations when we see them inside the function body. Note the lonely '➜' is the name of the continuation list as a whole. It isn't a separator. As a convention we always provide the continuations on the tail of the argument list. We often tip our hat to the old concept that one exit is somehow *normal*, by calling it `➜ok`.

The very last thing that a function that is passed continuations must do is to call at least one of the continuation functions, or to terminate execution of the thread. The continuation result cannot be used in the function, and indeed no processing can occur after the continuation is called. Continuations are not function calls, because they do not return. These rules are currently enforced by contract with the programmer.

Here is mock-read with continuations being used:

♦

```
(defun mock-read-with-continuations (n &optional ➜)
  (destructuring-bind
   (&key
    (➜ok #'echo) ; if all goes 'ok'
    (➜eos (be 'eos)) ; we don't like zeros!
    (➜parity (be 'parity))
    &allow-other-keys
    )
   ➜
   (if (= n 0) ; zero byte marks end of stream
    [➜eos]
    (if (oddp n) ; parity check
     [➜parity]
     [➜ok (write-to-string n)]
     ))))
```

Regular &optional and &rest arguments

Our use of `&optional` just before the continuations list parameter constrains the programmer from using `&rest` for regular parameters. Should the programmer need to provide a variable number of regular arguments, then he or she should pack them into a list. The '{..}' macro makes this simple to do.

♦

```
(defun f (a b &rest c &optional ➜) ;bad parameter syntax
(defun f ( a b more-args &optional ➜) ;good parameter syntax
```

When control arrives in the function the `more-args` list can be unpacked using `destructuring-bind` where `&rest` and `&optional`, etc. are available. Indeed this is the approach we have used for the list of continuation functions, and we have already provided an example. A call with variable arguments would then look like this:

♦

```
(f a-val b-val {c0 c1 c2} {cont-ok cont-bad0 cont-bad1})
```

Some useful continuations

All the functions in the library use continuations for end cases rather than throwing errors, because what is one person's error, is another person's opportunity to run some more code. In particular, this is how we add code for handling such things as continuing from left from leftmost, and right from rightmost. Continuations are important for achieving Turing Complete computing.

Here are some useful stub and error reporting functions defined in the distribution file **`functions.lisp`**. These are often convenient when setting continuation defaults.

♦

```
(defun do-nothing (&rest x) (declare (ignore x))(values))

(defun echo (&rest x) (apply #'values x))

(defun be (&rest it)
  (λ (&rest args)(declare (ignore args))(apply #'values it))
  )

(defun cant-happen () (error 'impossible-to-get-here))

(defun alloc-fail () (error 'alloc-fail))
```

The function **`do-nothing`** lives up to its name. It accepts any number of arguments, performs no operation and does not return anything. **`echo`** returns its arguments. Actually it is just a synonym for **`values`**. **`be`** returns a constant independent of what is passed in for arguments. **`cant-happen`** throws an error saying it is impossible to get to this function. This is useful for finding mistaken assumptions. **`alloc-fail`** throws an error saying that allocation failed.[9]

Functional programming

Some Terminology

The communication between the Turing Machine controller and the Turing machine tape transport unit consists of a limited number of instructions, such as step right, step left, read symbol, and write symbol. With a Universal Turing Machine, the controller for a specific Turing Machine is encoded on the tape.

[9] However, in the version of the library as of this writing you will see the Lisp allocation condition rather than calls to the allocation fail continuations.

While the Universal Turing Machine interprets this description it acts like the specific Turing Machine described by it.

The activity of creating the specific Turing Machine controller encoding can be called *coding*. The encoded controller description sitting on the tape can be called *code*. These terms come from communication theory. Note that not all codes are secret codes.

Similarly a microprocessor reads its encoded control instructions from system memory. In analogy to the Universal Turing Machine, this specific machine description causes the general purpose processor to take actions on its interfaces.

Processor instructions occur in groups we call *routines*. A routine is invoked either through an interrupt, a jump instruction, or if the processor has a dedicated call instruction, by using that.

In turn routines are grouped into entities called *programs*. Programs are managed by an operating system. A running program is called a *process*. The operating system assigns to each process a unique unsigned integer ID and a virtual memory space. The same program may be run more than once simultaneously. Each time the same or a different program is loaded, then run, it is given a separate process, and will have a different process ID and different virtual memory space.

Each program has one *main* routine. After the program is loaded, the OS will call the main routine. The main routine then calls one or other routines, which we may call *subroutines* to distinguish them from the main routine. Subroutines in turn may call other subroutines. Some subroutines might come from dynamic shared subroutine libraries that exist outside of the program. Others might come from precompiled libraries and linked into the code to create an executable image of the program. Still other subroutines will be custom written by the programmer.

In programming, a *function* is a routine that designates a return value. This return value is inserted into the program at the place the function was called from. In this manner functions may be used in expressions.

With some small changes, this same nomenclature works for interpreted languages. Accordingly, references to the OS and to the processor are replaced with references to the interpreter. The interpreter loads the program and interprets its instructions. Hence, the interpreter and the accompanying program model is a kind of virtual computer, also called a virtual machine.

Side effects

In this section we will discuss some properties of functions in programming and how functions in programming are related to functions in mathematics. To facilitate this discussion in the text of this section I will write function related terms used in programming while using source code style. In this section I do not mean that these are keywords in a language, though they might be, rather I am using this styling convention solely as a device to distinguish between the programming context and the mathematics context.

The term `function` is used by Fortran to describe routines that have a designated return value. Such `functions` can be invoked within expressions. In Fortran `function` routines are distinct from `procedure` routines, because `procedures` do not designate a return value, and thus cannot be called from within expressions. Examples of `functions` include `sqrt`, `sin`, and `cos`. An example `procedure` would include such things as a routine that writes input values as records into a database and does not return anything. Procedures may also write results back through arguments that are passed by reference. In the C language there are no `procedures`. If the programmer does not want a designated return value the `function` can is defined to return any value, perhaps a zero, and the calling code ignores it. Relatively recently in the evolution of C the `void` semantic has been added to the language. We can now say that a `procedure` is a `function` that returns `void`.

The term `function` was borrowed from mathematics for a couple of reasons. Firstly because in mathematics functions return a single value so they can be used in expressions, and Fortran functions share this attribute. Secondly, because programmers typically give their routines that perform approximations the same name as the functions in mathematics that they shadow. People who read the source code and see a `function` name that matches a function name in mathematics and will reasonably assume that the mathematical function is what the programmer wanted to have, but was forced to use an approximation because, well, after all, we can't always have what we want.

A `function` causes a *side effect* at run time if it writes a passed by reference input, and later that modified value is used by the program in a consequential manner; if it shares variables with another function and changes the value of one or more of those variables in a manner that is consequential; if it reads a shared variable as an input that did not come through the argument list and this read has consequences; and/or if it maintains state variables, which when modified change the future behavior of the function in a consequential manner. Another type of side-effect occurs if a function writes its designated result while also returning an error saying that it could not write the result, and that write has some consequence. As an example, this could happen if the memory associated with the designated result is also used as a temporary variable during computation, or if for some reason the function continues to run after setting the result and incurs an error.

Conventionally, a `function` is said to *have side effects* as an intrinsic property if there is any way that the function may be used at run time in a manner that causes a side effect. For the opposite case, where a function can never cause side effects no matter how it is used, it is said that the function does *not have side effects*. I prefer to not mix the *have* and *cause* forms of these statements, and the *cause* form works grammatically when we talk about run time events, so I will use 'cause' rather than 'have'.

By definition, functions in mathematics do not cause side effects.

The proposition that a function cannot cause side effects at run time, or alternatively the inverse proposition, that a function potentially will cause side effects, is a property of the function itself. A function can be known to never cause side effects either through proofs based on analysis of its code, or through knowledge that the method of construction can only lead to definitions of functions that cannot cause side effects. For example, if a language does not have syntax for expressing code that potentially causes side effects, then we can know that functions created in that language cannot cause them.

Although Fortran and C **`functions`** accept inputs and return a value, as do functions in mathematics – they can differ from functions in mathematics due to the potential for side effects. Fortran has common blocks and output arguments, and C has global variable scoping and syntax for passing inputs by reference.

When computer scientists wanted a term for a **`function`** that cannot cause side effects, so it would be more like the function of mathematics, they found themselves in a pickle sauce of their own making, as the term *function* in programming was already commonly used to mean a C type **`function`**. Instead of trying to rectify this, they patched the problem by using new term for this better behaving **`function`**, namely, **`pure function`**.

The use of the term **`function`** in programming will always be questionable in the absence of a formalism that shows it fits the mathematical definition.

Hence in this text I will refer to **`functions`** as *routines*. This is precise and correct. If it is otherwise unclear from context if a routine has a designated return value; I will state this explicitly. Of course all C-like routines have a return value. If it is important to note that a routine cannot cause side effects, I will say so.

But why limit ourselves to just talking about routines? Perhaps we want to analyze arbitrary sequences of instructions. The term *referential transparency* says that replacing any instruction sequence, expression, or function, with its result will not cause consequential changes to the program. We might imagine being able to replace referentially transparent instruction sequences with big lookup tables. This is the same as saying that said sequence of instructions cannot cause side effects. Hence, in this text, when I want to say that a sequence of instructions has referential transparency, I will say, "this sequence of instructions cannot cause a side effect."

It is possible to write routines without side effects in Lisp, Fortran, C, and for that matter in all popular languages. It is just a matter of contracting with the programmer to make a commitment to use this programming style, and for the programmer to have the discipline to pull it off. Sometimes programing teams who want to use this programming style will not want to rely on such a contract and programmer discipline, so some specialized languages have been designed with a routine definition semantic that simply does not allow the coding of side effects, notably Haskell. (Haskell goes beyond this by also supporting routines as data.)

Our TM library routines might make changes to their input arguments. For example, by moving a tape head of a tape machine argument. Actually most of the routines in this library have been written for the very reason of causing side effects on tape machines.

Routines as data

The Universal Turing machine stores an encoding of its state controller on its tape. In a similar manner it was a great advance in electronic computing when programs began to be treated as data. And though we have all memorized the phrase "programs are data", we have also been taught that "self-modifying code is bad."

Programs in contemporary computers are loaded onto virtual memory pages before they are run. Virtual memory page accesses are constrained by permission settings. After the routines are loaded, the pages are set to allow execution and to be read-only. Data to be used by a program are placed on separate pages that are readable and writable but are not executable. Hence, in some contexts programs are like data, for example when being loaded or stored; but in other contexts they are not, for example when being executed.

The advantage of read-only code pages is that they may be shared between processes that are running the same program without danger of this becoming a pathway for communication between the processes. However, having read-only code pages means that without making special arrangements with the operating system, programs cannot modify their own instruction sequence.

Nothing prevents a programmer from writing in advance all variations of a routine that could come about from self-modification, and then choosing among them at run time. It might be inefficient, but in theory it can be done, and it implements the same functionality as self-modifying code without forcing the virtual memory system to put write permissions on code pages during execution. A miniature example of writing variations in advance is a case statement or a jump table. Even an `if` statement chooses between two code variations.

In the spirit of code being data, some languages support assigning a routine to a variable, passing it around as data, and then calling the contents of said variable. Typically what is really happening is that all the functions to be used are loaded in advance on their read-only executable pages, and it is the addresses of the functions, and not the definitions of the functions, that are assigned to variables. This is the explicit case in C.

Suppose for example, we want to write a first routine that when passed a second routine as an argument, operates on it, and then returns a third routine. More specifically suppose we want to curry away an argument of the passed-in routine by replacing it with a constant. We can do this without self-modifying code by returning the address of a prebuilt gasket function that accepts a reduced number of parameters and then calls the second routine which was passed in.

In general it is impractical to expect all possible self-modification results to be compiled in advance. This trick of using a gasket function works for prepending prefix instruction sequences, and appending postfix instruction sequences, but not for more profound modifications. Also it comes at a performance cost when programmer might have intended a performance increase.

Lisp program definitions that were typed in by the programmer remain available at run time. This is because Lisp is an interpreted language. In contrast in a compiled language the function definitions typed in by the programmer will have been compiled away before run time. Thus in Lisp it is possible to write a first routine that accepts a second routine as an argument, and then for the first routine to literally modify the routine definition that was passed in and to return this modified definition as a result.

Lisp also has a facility for compiling routines on the fly, so the new definition may be compiled if the point was to gain performance. The on the fly compile takes care of the permissions on the virtual pages. However, this comes at the cost of not following the OS's built-in understanding of the relationship between programs, processes, and virtual memory pages.

Transactional behavior

In data science a *transaction* is a group of database operations that either all succeed or none of them are applied.

Routines without side-effects are transactional. If an error is thrown while executing such a routine, no data will be changed. Without side effects there is simply no way to change the program state apart from returning a value, but if the routine terminates early, no value is returned.

The routines in the TM library do not return, rather each follows a continuation, and in multithreaded programming, potentially many continuations.   However we can still maintain a property similar in sprit to being transactional. Namely, we handle data in such a way that continuations can perform their jobs.

For example, if a routine attempts to advance a tape head beyond rightmost and then write a value, the `right-of-rightmost` continuation will be taken. The routine must preserve the ability of `right-of-rightmost` to fix the problem and to complete the write. It does this by leaving the tape head on the rightmost cell, and passes to both the machine being operated and the value to be written to the continuation. Then `right-of-rightmost` may expand the tape and complete the write. It might complete the write itself, or it might re-enter the origin routine. In the latter case there will not be an infinite loop because the tape was expanded.

Descriptive and procedural

An example of descriptive programming language is Prolog. For example, in Prolog we may describe the data we would like to have, while using blanks for the missing parts. Due to the semantics of Prolog this will cause a unification engine to fill in the blanks. This work might trigger a database query. A couple of other descriptive languages are the compiler tools Lex and Yacc. With Lex the programmer provides regular expressions for tokens, and then the tokenizer is pointed at a file and tokenizes it automatically. In Yacc grammar rules are given in terms of tokens, and the built in parser examines the token stream and builds a parse tree. Another example of descriptive programming comes from my QST radio node processor, where the programmer first configures the arithmetic logic units for streaming solutions, and then from the assembly level the program may begin the streaming.

It is sometimes the holy grail of computer scientists to find generally useful engines that reduce the burden on the programmer to merely describing the problem. We see this in library designs, and in the school of thought that one of the jobs of computer scientists is to design languages that ‘fit problems’.

In contrast to *descriptive programming*, there is *procedural programming* where we provide step by step instructions on how to solve a problem. Descriptive programing requires being able to formally describe the problem while being in possession of an engine that solves a general class of problems based on their descriptions. While writing a procedural program requires first having a solid understanding of a problem, and then writing a program that provides step by step instructions for solving it.

Lisp can be programmed in a descriptive style for recursively defined problems, and this style is usually taught in programming courses for Lisp. Theoretically, all computable problems will have a recursive form. Given an appropriate library the C language can be used in this manner, but it is usually nowhere near as elegant. C came of age as a procedural language, and almost all C programs are procedural.

The TM library is descriptive of a tape machine transport unit. Turing Machines are all about stepping, and sequences of instructions for solving problems. They are the essence of procedural programming. I sometimes refer to the TM library as an iteration library for Lisp. This is somewhat of a conceptual contradiction, as the Lisp programming style is usually based on recursion rather than iteration. The point of using Lisp was the ability to make formal statements in the language. Still I occasionally see on the programming boards someone looking for an iteration library to be used with Lisp.

Summary

The chapter title suggests that we are going to define a term called 'procedural programming', but what we have discovered is this term does not mean one thing. It can mean that we have pure functions, i.e. routines that cannot cause side effects. It can mean that the language supports treating routines as data. We found that this could mean passing around addresses to precompiled routines, and perhaps composing existing routines using a gasket. Or it could mean that the language can modify routine definitions directly. We also might have routines that describe our problem, or those who explain how to solve it step by step.

Lisp is usually called a *functional programming language*, while C is not often called this. In both C and Lisp it is possible to write routines that cause side effects, or to write them in a manner that they can not cause side effects. It is possible in both languages to pass routines around as values held in variables. Lisp allows routines to be anonymous, C has a long tradition of letting the programmer ignore information he or she does not want to have, which is pretty much the same thing. The one difference is that in Lisp, the programmer read routine definitions at run time, and created new modified versions. In C program definitions are compiled away, so at run time they cannot be read, let alone operated on.

Hence is not entirely clear what a functional language is. We might be better off explicitly giving the properties of the routines relative to their ability to cause side effects, to be modified at run time, or to be used in a descriptive manner, and to be transactional.

The TM library does make use of passing routines as arguments, so in this sense it is an example of being functional. But the arguments are often tape machines that will be modified through side effects, so in this sense the library is not being functional. The library routines are transactional in that they leave data in a state that is appropriate for each continuation that might be called. This is similar to the way routines that do not cause side effects are transactional.

Classes

After the great fashion craze in computer science of *structured* programming came the mantra of object oriented programming. This chapter discusses the terminology we will use for object oriented programming in the TM Library. Lisp's object oriented programming extension is called CLOS.[10] We explain in

[10] *C*ommon *L*isp *O*bject *S*ystem

this chapter how the CLOS approach is a dual concept to that used in most other languages (such as in C++).

Format, instance, type, and *abstract data type*

In the section **Multiple machines and sharing tapes** we introduced *data type* and examined it from the point of view of its role in setting the format of memory.

We may apply operators to compose data format descriptions to create the format description of a compound data type. Such operators are provided in language type facilities. As examples, in the C language, we have `struct` which compounds types of differing length components, and `[·]` which compounds formats of the same length components. If the program respects the type formats, and the symbol table locates variables into memory so there are no overlaps, then writing a value to one variable, will not corrupt the value of another variable - unless those variables are in an explicit union.

Instantiating a compound format creates a composite instance made of the smaller instances. We may even consider all of memory as being one grand composite instance. In high level languages we give names to data types. As data type is bound to format, this facilitates the consistent use of format in assignment, in copies across routine boundaries, among other situations.

But data type is more than formatting. By using names for data types a compiler tool, or a human programmer, can ask the 'is-a' question about an area of memory, and the answer will be a symbol, i.e. the name given to the type.

Typically programmers create the type names, and these names often mean something more to the programmer than they do to the compiler. Take for example, a `struct` called ‘employee’ that has fields for a ‘name’ and ‘ID number’. To the compiler this might correspond to a block of 8 bytes, which may only be copied to other areas of the same data type. In contrast, the human programmer might have in mind the company employees, and he or she might have plans to add more descriptive fields later. The compiler and the programmer may have very different goals when asking 'is-a' questions, or when stating 'is-a' facts about a data type.

Now here is an interesting development. Suppose we describe two types that have different names, but they are otherwise identical. In real programming this happens often. Take for example a `struct` with two numbers that in one case is a vector on a real plane, but in another case is a complex number. Perhaps we determined their formats were identical by having analyzed the lengths of their components and any operators used when compounding those components. As a result of having defined these data types with different names our compiler may refuse to copy data even though there is no danger of corrupting memory. In this situation the format name carries with it a distinction to the compiler that is independent of that of the details of the format.

We can explicitly acknowledge that these two types of the same format. by separating format out as being a separate entity from the data type. Then the vector type and the complex type provide alternative interpretations of the two number format.

♦

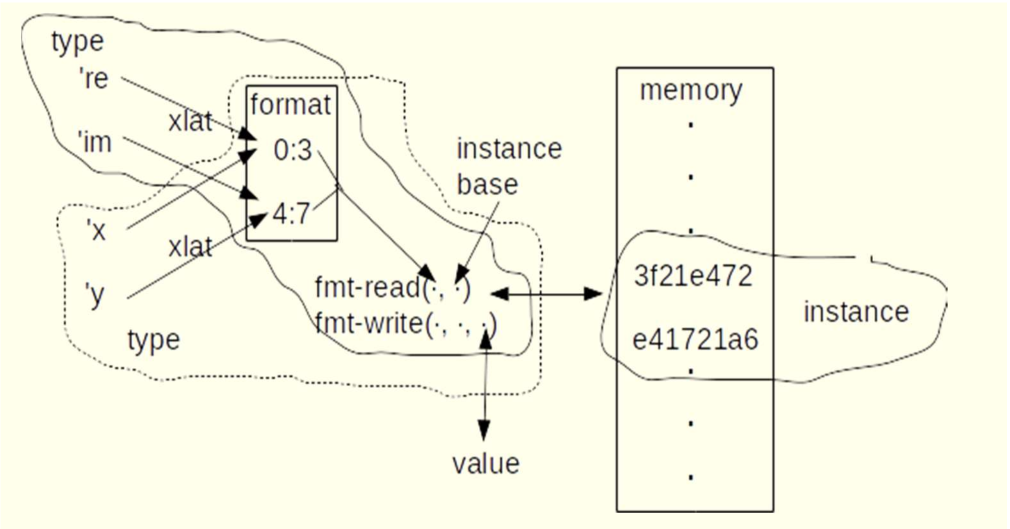


Figure 6 Type format, instance

This sharing of format is similar to the object oriented programming inheritance concept variously called *specialization*, *extension*, or *derived type*, but the inheritance concepts more generally allows for base format block to be extended. Hence derived types can have different formats, so the concept of shared format is different, though related. Part of the job of modern compilers and languages is to take over the formatting work for us. Programmers typically see very little of this problem, and when they do deal with format they often find it annoying. Examples of format choices made at the programming level include choosing among floating-point single or double, or choosing 32 bit or 64 bit integers.

We must have memory access routines to be able to make use of formatted data. In single threaded programming we will only need *read* and *write*. In multithreaded environments we will also need some sort of locking or atomic read modify write operation. In this example `fmt-read` accepts an instance base, an offset value gotten from the format description, and then returns an instance from memory. `fmt-write` accepts an instance as a third argument and then copies that instance into memory.

Conflating data type with data format holds us back from writing generic code. Suppose we pull out the format from the definition of type. What is left is an abstract notion of type. We are then freed to use that notion of type when writing generic code. This style of programming is called *generic programing*. Of course before our program can run, we will need to bind definitions of format to the abstract types. We might then have different generic programs that have the exact same control flow in the abstract, that compile into different instruction sequences.

As an example of abstracting out format, suppose we have an instance of our complex number type, and it was instantiated from format declared with a `struct` in C. Our access routine for reading from this formatted memory accepts a given field name, say either `re` or `im`, which are symbols, and then returns the contents of the corresponding field. Write does the inverse action. Now compare this to using C++ STL map, where the access routines accept field names as keys, and then traverses a red black tree to access the instance. In both cases the call to the read routine accepts the same symbol and reads or writes the same instance. Either way the two programs will return the same results. Clearly there is something going on with type which is independent of implementation.

In languages such as C we bind format to data type to help us write correct code. In fact, it appears the concept of type was invented for this. We can add another option for type checking by specify in advance the operations allowed on particular type. This would be very helpful in preventing programming errors in either concrete code or generic code. C++ programmers will recognize that class declarations can be used as such a routine list specification.

We can abstract the routine list by using routine names instead of the actual routines. We may then have a list of generic routines that constrain the use of abstract type. And of course, by runtime these abstractions would have to be bound to format and defined routines. With this approach we may write generic programs with type checking.

Earlier we discovered that there is a many to one relationship between type and format. Consequently we cannot work backwards from format to define the type it is bound to. But what about the relationships between type and the routine list that constrains its use, can we derive the type from the routine list? In the case where we have a complete list of all routines to be allowed with a type, and there is a one to one correspondence between the type and its routine list, we can do this. Python programmers call this ‘duck typing’. This comes from the phrase, ‘if it quacks like a duck it must be a duck’. This technique comes from the 1974 paper where Liskov and Zilles developed the concept that, “An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects”, DOI 10.1145/942572.807045. However, it is possible to have many types sharing the same routine list. These bound types might mean different things to the human programmer. In such a case the type cannot be derived from the routine list.

A C **`struct`** and an STL map are decidedly different formats. Hence, they cannot be the same data type. However, either can be used on the lower layer for returning values in response to being given keys (or field names), i.e. the can be used for implementing the same abstract data type.

♦

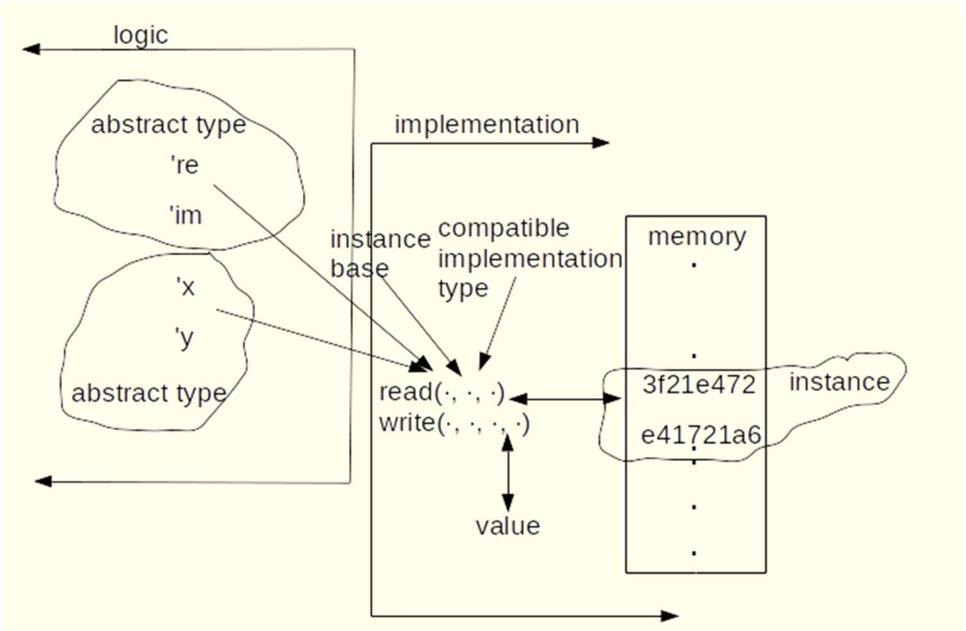


Figure 7 Abstract data type

In the figure above the programmer sees only a symbolic form of the type and a read/write interface. Different implementations may be created by providing different compatible implementation types. Though the programmer creating the logic might not see it, we still need a data type for the implementation.

With CLOS, a new type is defined by giving a type name and field names in the `def-type` form.[11] CLOS also provides an implementation, but we emulate abstract data type by trying not to think too much about it, and instead

[11] `def-type` is an alias we provide for CLOS's `defclass`. I would like to avoid conflating the concepts of type and class, so have provided this synonym.

concentrating on the program logic.[12] This is Lisp so there will be many parenthesis. Here is a type called `complex-number:`

♦

```
(def-type complex-number () ((real-part) (imaginary-part)))
```

You will see more examples of this in the remainder of the book. Note that in CLOS the 'field names' are called 'slots'. I guess they didn't want to conflate what looks like an abstract type declaration using `def-type` with the combined type and format building of C's `struct` or perhaps with fields from database records.

The definition of type built on formatting is the result of the composition of more primitive types, with the base case of a bit. The definition of type from routines is based on analysis. Hence we can refer to these in a descriptive manner as *composition type* and *analytical type*. Composition type is a bottom up definition, where as analytical type is a top down definition.

Inheritance

The second parameter of the `def-type` macro form provides a place to put a list of other types, and then the slots from those other types will be automatically included in the new type. This process of including slots from other types is called *inheritance*. We say the new type is more *specialized* than the types it inherited slots from. If we do not want to inherit slots from other types, as was the case in the complex-number example, give an empty list as the value to bind to the second parameter, i.e. give an empty list as an argument.

Typically instances of a specialized type can be used by routines that were designed for instances of the more general type. This is because all the required slots will be present in the more specialized type, while code written for the more generalized type ignores slots it does not use. In fact, CLOS will assume

[12] Alas, perhaps some day the compiler will be smart enough that such an approach does have such a performance impact. Though sometimes flexibility does translate into performance. It certainly translates into faster development time.

we want to do this, and during dispatch it will fall back to type matching against more general types when it does not find a match on the specialized type.

This explanation of extra slots being ignored works well when arguments are passed by reference. However, when arguments are passed by value, we can't ignore each instance's actual size. In Lisp all instances created from user defined types are passed by reference.

There can be multiple specializations for a given type, each adding different additional slots. This happens in the TM Library.

Routine classes and generic routines

When programming in a language like C we are forced to write our routines with format correct data usage embedded in them. Routines are then linked by name. However, sometimes we would prefer a more abstract, or more generic behavior. Say for example, we want a print routine that takes an argument of any type, and then outputs a Unicode image of the argument.

In such cases the logic of the program is defined in the abstract, but how can we implement this? One solution is to implement many print routines, and then have the tools choose which print routine to link based not just on the routine's name, but also on the type of its argument. Accordingly we place a simple call of the form *print(x)* in our source code. In a compiled and linked language, the linker will select the *print* routine to be used based on the type of *x*. Thus *print(5)* will call a routine that converts integers to strings and copies them to the output, while *print("Tom")* will call a different print routine that copies a string to the output.

The set of routines used are equivalent in the sense that they all provide the same abstract functionality. When we put routines that are equivalent according to some property together in a set, we have something called an equivalence class. In this case it is an *abstract routine equivalence class*. We will shorten this term to *routine class*. In our print example, *print* is the name for the class rather than for a particular member of the class.

In our version of CLOS a routine class is declared using a `def-routine-class` form.[13] If we were trying to create the behavior as described for our print example, then we would define the print class as:

♦

```
(def-function-class print(x))
```

In C++ we can implement routine classes, but for some unknown reason the language does not give us a facility for declaring them.

Type classes

There is dual concept to the routine class where we place all of the routines that accept the same type of argument into a set. This set would define a *type of argument equivalence class,* or *type class* for short. We name a type class after the type that is used to define it. For example, if we defined routines that operate on a type called, 'complex-number', and put them all in a set, we would have a *complex-number type class*.

The set of routines that operate on formatted data gives that data its personality. This is to say, that a programmer can analyze the functional use of data and recover a definition of abstract data type. The definition of abstract data type recovered from any particular program will might not include routines that might be used on the same data type in the future or in another program. Hence we define abstract type against a specification that describes all the routines that may be used, rather than doing so from the routines that are used. (If there are some routines that are never used, we might want to consider changing our specified routines list.)

Theoretically we know that there is a second order infinity number of mathematical routines, but only a first order infinity of data that can be

[13] def-function-class is an alias we provide for CLOS's defgeneric

expressed as computer variables. Yet computer routines are coded as data. Perhaps this explains why routines tend to be longer and less regular (more complex) than data type declarations.

CLOS does not require us to define type classes, but other languages, such as C++, place a great deal of emphasis on it. In C++ a programmer includes the literal `class`, followed by an open brace, a list of all the member routines, and then a closing brace. This works well when the defining type applies to a single argument, but, when there are two or more typed arguments the routines will belong to multiple type classes - so where should the programmer declare them? Surely we don't want the programmer to have to declare the same routine more than once! In such cases C++ drops into a 'friend function' approach. We would probably need this, for example, to define binary operators on our complex-numbers. When calling a friend function there is no prefix argument preceding a dot, instead all arguments appear in the argument list.

Perhaps it is because of this multiple argument issue that CLOS does not have a construct for declaring type classes.

Dispatch

Though CLOS does not facilitate the explicit definition of type classes it does select each routine based on types of its arguments, in a manner similar to selecting friend functions in C++, though because it is Lisp, this is done dynamically. The process of matching of the routine name, matching of argument types with possible generalization, and calling a routine is called *dispatching* a routine. Routine dispatch incurs considerable runtime cost.

Though CLOS does not have built in support to help the programmer declare the *type* classes, it will throw warnings if the programmer does not declare the *routine* classes.

Lisp does not normally perform argument type matching, so we need a new form for declaring routines with typed arguments.[14] One defines routines with type signatures by using the **defun-typed** form.[15]

Here are a couple of examples:

♦
```
(defun-typed print ((x integer) …)
(defun-typed print ((x string)) …)
```
The ellipses indicate that code was left out. They are not part of the routines. The parameter list consists of pairs. The first in the pair is the parameter name, and the second is its type. Hence the first print routine accepts a first argument of type integer. If an argument to be bound to the routine parameter may be of any type, it is listed in the old fashioned manner without a pair.

The Tape Machine data type

Introduction

The TM library interface is described in this section. This interface is abstract, being constructed from a combination of generic routines and declarations without definitions. The interface provides routines for iteration and continuation over containers.

Routines in the **tape-machine** class may be used to traverse through a container, access instances held in it, and to handle end cases. Loop constructs follow naturally as quantification operations on containers. Unlike the case of the standard loop structure in Lisp the unprocessed parts of lists are accessible after exiting a loop, and may be used to continue the operation after a fix. Unlike the case of stream libraries, the tape machine has random access capabilities between continuation limits.

The iteration model is constructed from formal concepts, most that have been well studied in computation theory. Tape machine based computation is a hybrid

[14] the term 'form' in Lisp is analogous to 'statement' in other languages

[15] defun-typed is an alias we provide for CLOS's defmethod

between that Turing Machine and a Lambda Calculus, with the Turing Machine used for iteration within a region, and then continuation routines, lambdas, invoked for extending the boundaries of the region. I make the claim that there are no library architectural limitations for working with arbitrary long containers or computations.

If for no other reason, this formal derivation is useful because it guides the terminology. As it turns out, there is an explosion in the number of end cases when dealing with containers and generalized iteration. This formalism provides a language for talking about them.

The TM Library may be used as a tool when writing Turing Complete programs. The current Common Lisp implementation of the library will be subject to the limitations of the Common Lisp interpreter or compiler. One will notice limitations for such things as address width and available virtual memory. The general Turing Complete architecture lifts the limitation on address width, and facilitates a graceful approach to the limits of the underlying physical memory. Even after depleting physical memory, one might continue again later, as the Turing Complete architecture puts no limitations on a physical implementation that supports the hot-swap addition of memory.

Nomenclature

Many implementations of the **`tape-machine`** type have a *tape*, and *indicator head* (or just *head*) fields - where these are placed into analogy with their namesakes in the Turing Machine model.

Conceptually, or actually, a tape consists of an array of one or more *cell*(s). Each cell holds exactly one *instance*, where such an *instance* is either a) a *subspace* or b) data for which the tape machine user knows, or can recover, the type for.

For a *projective* tape machine that has two or more cells, all cells, except two, have a *left neighbor* and a *right neighbor*. The two potentially special cells are called *leftmost*, which has only a right neighbor, and *rightmost*, which has only a left neighbor. When such a machine has only one cell, well, that cell has no neighbors.

For the TM Library we allow the inclusion of other topologies. For example, in an *affine* machine of three cells or more, all cells have both distinct left and right neighbors. This is accomplished by hooking the cells in a ring. When there are only two cells in such a machine, then for each cell, the other cell is both its left and right neighbor. For a one cell affine tape, the one cell is its own neighbors.

A tape machine might not be implemented over a container. Instead, for example, it can be an abstraction that is maintained through routine calls, in which case even a projective tape might appear to be open and not have a leftmost or rightmost.

Instances are the things we put into cells of the tape. Instances are not part of the container topology, i.e. they do not have connectedness. Hence they do not have neighbors or location. However, they may be placed into cells, and cells are connected. So, for example, rather than saying something like "the leftmost instance", it is better to say, "the instance in the leftmost cell."

The terminology of *left* and *right* stems from the convention in mathematics for number lines. Here the cell addresses fall on a number line. Going right on the number line leads to higher addresses, and going left to lower addresses.

Putting the head on a specific cell is to *cue* the head. A number of cue routines are defined in the library: *park*, *cue-leftmost*, *cue-rightmost, cue-to, and sn. cue-to* takes an address, where as s*n* steps the machine (takes a first difference in addresses). The distance argument is passed 'in a box' i.e. by reference so that it can be modified to show the steps remaining should we attempt to step beyond the rightmost or leftmost cell.

Heads are either *stepped*, *cued*, or *parked*. We do not apply other motion verbs to heads. In contrast, instances are said to be *moved* or *copied.* "Stepping a tape machine*"* means we follow the tape machine procedure, which is different than “stepping the head” which refers to moving the tape head one cell over on the tape. Only instances can be moved, so saying that "a tape machine *moved*", would mean that said tape machine was an instance held in one cell, but then it was moved to another cell.

Primitive routines are declared in the *decl-only* source files along with routines that have generic implementations. It is sufficient when defining a new tape machine specialization to provide an implementation for each of the routine class declarations found in the *decl-only* files. Routines defined in other files are built up from these primitives. Routines may be defined with *parameters*. At runtime *arguments* may be provided with the routine call. When the `&optional` and `&rest` keywords appear in the parameter list it is possible that a routine is called with fewer arguments than the number of defined parameters. In such a case the specified default arguments are used.

Tape machines may have *status*. Status is typically whether the machine is *abandoned, empty*, *parked*, or *active*. In contrast, Tape Machine *state* consists of the variable parts of the machine including the head location and the instances on the tape. Though status is a kind of state, it is a state about an underlying machine, which is a higher order concept.

Modifications to head location, or to the tape contents are said to be *stateful* changes. Modifications that add or remove cells from the tape are said to be *structural*. (Adding cells to the tape is called *allocation*. Removing cells from the tape is called *deallocation*.) We know from experience that programs that only modify state tend to be more straightforward and have fewer bugs than those that modify structure. Yet there always seem to be cases, such as gathering results, where structural changes are more convenient. Even Scheme allows one to use a cons cell to lengthen a list, albeit only from one end.

When we need to specify boundaries, we try to use inclusive bounds so that we will not need to represent addresses that are outside of our address space. Our use of quantifiers in place of loop structures facilitates this approach.

Variables that hold tape machines often have a suffix of '-tm'. We think of this as being analogous to putting an 's' on the end of a world to make it plural. Take for example, suppose a tape machine holds instances that describe trucks. Perhaps the truck descriptor holds an id for a truck and miles since its last maintenance. The specifics of the descriptor are immaterial here. We might call said tape machine 'truck-tm'. This implies that that one or more truck descriptor instances, or for a status machine, zero or more truck descriptor instances are referenced by said variable. We do not say 'trucks-tm' because that is redundant. Marking the variable with a '-tm' already implies plurality. Likewise we do not say 'trucks', firstly because a simple 's' is easy to glance over,, and secondly because it doesn't imply what interface can be used for accessing the referenced container.

The typical coding form is to do some work, and then to step. If the machine steps right of rightmost, or left of leftmost, then we do something else that is appropriate for this end case.

Stepping through the cells of a tape (iteration)

Each tape machine tape is typically bounded. Exceptions exist for abstract machines implemented through routines. Such machines may have tapes that appear to be infinite or looped. Also, *affine* machines have circular tapes, and thus have no bounds.

To step a tape machine, say one called `tm`, one calls the routine `s` and passes it the `tm` instance and optionally two continuations. Upon striking a bound a continuation routine is executed, and that continuation routine may overcome the bound, or perhaps take some other action. Here the two continuations are called `cont-ok` and `cont-rightmost`:

♦

```
(s tm {:➜ok cont-ok :➜rightmost cont-rightmost})
```

The routine name, `s`, stands for 'step'. The tape machine, `tm`, is a combination of iterator and container and s causes the internal iterator to step. If we put the container into correspondence with a Turing Machine tape, the internal iterator with the tape head, and put the program, which is external to our container, in correspondence with the Turing Machine controller; then `s` can be thought of as stepping the head of a Turing Machine.

`➜ok` stands for 'continue OK', while **`➜rightmost`** stands for 'continue after stepping right of rightmost'. At any point in time our container will typically have a bounded size. When this bound is struck, we call **`➜rightmost`** so that the program has an opportunity to overcome this bound and implement Turing Complete behavior.

♦
```
(s tm
 {
  :➜ok     (λ()(print "everything went well"))
  :➜rightmost (λ()(print "go get more data!"))
  })
```
For convenience we provide some general purpose continuation routines. One of these is the routine **`be`**, which returns a routine that returns a given value. Hence,

♦
```
(s tm {:➜ok (be t) :➜ok (be ∅)})
```
will return true, `t`, should the step succeed, and return false, `∅`, should it not. These are in fact already the defaults. When used in this mode, `s` will almost surely be followed by a test of the return value, so when used in this mode we will have to do tests twice to decide just one thing; once inside the step routine, and once outside.

It is a common mistake for beginning users of the library to put a form that evaluates to a value as an argument, rather than a form that evaluates to a routine. The following, for example, gets errors:

♦
```
(s tm {:➜ok t :➜rightmost ∅}) ; bad semantics !!
```

Tape Machines can also be implemented over routines rather than containers. So for example, we can have a tape machine that represents the Natural Numbers. Each time it is stepped it just goes to the next Natural Number. There will be no rightmost cell, so a step call might appear as:

♦
```
(s Natural
 {
  :➜ok     #'do-nothing
  :➜rightmost (λ() (error 'cant-happen))
  })
```
Here `#'do-nothing` is a stub routine that simply returns. Though the continuation does nothing, the step routine has done something. It stepped the `natural-number-generator`, so the next time we read the machine we will see a one larger number.

There is no end to the set of Natural Numbers. so should we ever call the continuation for stepping from rightmost, there is a bug in `Natural`, and we throw an error. This sort of error is of a different nature than that of an end case. Unlike an end case, such a control flow change would be unexpected and of unknown origin. We have no obvious recourse that would be more meaningful than the thrown error. In general it would is unusual code that knows what to do in a presence of program bugs. Such code would be 'fault tolerant'.

Note that continuations are routines, and the `➜rightmost` continuation takes no arguments, so the error call must be wrapped in a lambda so as to give it an argument.

There is already a `cant-happen` routine in the library, so instead of the lambda expression that throws an error we could have passed in `#'cant-happen`.

Like `s` above, many of the tape machine routines are single letters. This facilitates stringing routine names together to create compound access routines, in analogy to lisp's car and cdr access language (cadr, caddr, etc.). The access language is then extended into a general out of band signaling pattern matching language that works on arbitrarily long data types. See the chapter .

Here is a simple program using a tape machine:

♦
```
(let (
      (a-tape-machine (mk 'list-tm {1 2 3}))
      )
 (print (r a-tape-machine))
 (s a-tape-machine)
 (print (r a-tape-machine))
 (s a-tape-machine)
 (print (r a-tape-machine))
 (s a-tape-machine)
 )
```
When run it produces:

♦
```
  1
  2
  3
  NIL
```
`mk` accepts a tape machine type and an initializer then returns an initialized instance. In this case the leftmost cell will have a '1' in it. Its right neighbor will have '2' in it. The rightmost cell will have '3' in it.

`r` is a routine that accepts a tape machine and returns the instance indicated by the head. **`w`** does the inverse operation. It accepts a tape machine and an object, and then writes the object into the indicated cell. Hence, the first `(print (r a-tape-machine))` will print the number 1.

`s` moves the tape head so that it lands on the right neighbor cell. The second `(print (r a-tape-machine))` will print the number 2. Etc. Note we do not provide any continuation routine arguments to `s`. They are optional, and the defaults are `(be t)` and `(be ∅)`. That is why at the final step, when stepping fails, the program returns `NIL`. (The printer is not aware of our alias `∅`, so it prints `NIL`.)

Tape-Machine primitive interface

By *interface* I mean the set of routine class declarations and the routines written in terms of the declarations. These latter routines are said to be 'generic'.

tm-type.lisp

`tape-machine` is the most general type in the TM Library.

♦
```
(def-type tape-machine () ())
```

No instance will be made of **`tape-machine.`** Rather **`tape-machine`** has been defined only as a basis for specialization. Routines that have parameters with the type **`tape-machine`** will instead be passed instances of specialized types. Once such an instance has been passed into a routine, the only thing the routine can do with it is to pass it to other routines without accessing any slots. This is because tape-machine does not have any slots. This turns out to be useful for recognizing generic routines.

We discuss some implementations in more detail in a later chapter, . Implementations are created by specializing the **`tape-machine`** through the addition of slots that hold data. Here is the type definition for the singly linked list implementation. **`tape`** is a reference to the list, and head indicates a node in the list. Taking a step will move the head to the next node.

♦
```
(def-type list-tm (tape-machine)
 (
  (head ; locates a cell on the tape
   :initarg :head
   :accessor head
   )
  (tape ; a sequence of cells, each that may hold an instance
   :initarg :tape
   :accessor tape
   )
  ))
```

Programmers will notice that such a tape machine resembles an iterator. The **`head`** is state for the iterator, such as a pointer to a member in the container (to a cell holding an instance), and **`tape`** is a pointer to the container. Such an iterator need not be given a parameter naming the container, as it already has a reference to it.

Linked lists in Lisp refer to their data by reference, so any instance, independent of its type, may be found on the tape for this machine.

In our morphism between the mathematical abstraction of a Turing Machine, and our TM Library, *type* is placed into correspondence with the Turing Machine *symbol*, and instances of type are placed into correspondence with instances of said symbols. Turing Machine *symbols* should not be confused with the Lisp language *symbols*. Typically it is clear from the context which symbol definition is intended.

The calling program plays the role of the Turing Machine state controller. The calling program sends commands, or statements, by invoking **`tape-machine`** type class member routines. Hence our interface routines will correspond to the commands.

tm-mk.lisp

For tape-machine specializations with implementations we can create new instances using the routine **`mk`**:

♦

```
(defun mk (tm-class keyed-parms &optional ➜)
 (let(
    (instance (make-instance tm-class))
    )
  (init instance keyed-parms ➜)
  ))
```

`mk` has three parameters. The first parameter, **`tm-class`**, may be bound to a Lisp symbol, where this symbol is a type name, and this type is a tape machine specialization with an implementation. Later we will see that **`list-solo-tm`** is such a specialization with an implementation. Hence, an example legal argument to pass in is **`'list-solo-tm`**.

The second parameter, `keyed-parms`, may be bound to initialization data for the new machine. We might, for example give this argument a value of `:tape (list 1 2 3)`, or equivalently, `:tape {1 2 3}`. For the solo-tape machine, this initializes the first three tape cells to the numerical instances 1, 2 and 3.

There are three common continuations, though specialized versions of tape machines can define others:

♦
```
(destructuring-bind
 (&key
  (➜ok #'echo)
  (➜fail (λ()(error 'bad-init-value)))
  (➜no-alloc #'alloc-fail)
  &allow-other-keys
  )
```
Exactly one of the continuations will be called as the last thing that `mk` does. `➜ok` is called and passed the newly made instance if all goes well. `➜fail` is called if there is a semantic problem initializing the instance. And `➜no-alloc` is called if there is not enough memory for creating a new instance. Note each of these continuations is set to a default routine. Hence, when no continuations are specified we get old fashioned behavior, where we make a call to `mk` and it returns a value, or instead throws an error.

You will notice that `mk` has been declared with `defun`, and not `defun-typed`. As such, in Common Lisp, we cannot have multiple versions and then select among them depending on the type of the argument. However, `mk` creates an instance and then calls `init`. and `init` *is* defined with `defun-typed`, so we may have a different versions of `init` for the various `tape-machine` implementations.

The `def-routine-class` does little more than provide an argument count. We can't even declare defaults for optional arguments. So here in the first line we have a declaration for a class of routines called `init` where each routine in the class will take two arguments followed by two optional arguments, perhaps followed by more arguments. The list that will capture the continuation arguments is a parameter called '➜'.

♦

```
(def-function-class init (instance init-value &optional ➜))
(defun-typed init
 (
  (tm tape-machine)
  (keyed-parms cons)
  &optional ➜
  )
 (destructuring-bind
  (&key
   (➜ok #'echo)
   (➜fail (λ()(error 'bad-init-value)))
   (➜no-alloc #'alloc-fail)
   &allow-other-keys
   )
  ➜
  (destructuring-bind
   (&key tape &allow-other-keys) keyed-parms
   (cond
    ((∧ tape (typep tape 'sequence))
     (cue-leftmost tm)
     (w tm (elt tape 0))
     (loop
      for item in (subseq tape 1) do
      (as tm item {:➜ok #'do-nothing :➜no-alloc ➜no-alloc})
      )
     [➜ok tm]
     )
    (tape [➜fail])
    (t [➜ok tm])
    ))))
```

Just below the routine class declaration is a definition for an `init` routine. This `init` routine will be called if no other `init` routine has a more specialized `tape-machine` class specifier. `tape-machine` has no implementation, so this `init` version is generic.

One can also find a shallow copy routine in the `tm-mk.lisp` file. When given a `tm-machine`, it returns another `tm` machine that has a new head and a new tape. However, the tape references for both machines will point at the same instances.

Chances are good that the entanglement copies are a better choice when one desires to have copies that share references.

tm-decl-only.lisp

The **`tm-decl-only.lisp`** file contains routine class declarations. Specializations must provide:

♦

```
(def-function-class r (tm &optional →))
(def-function-class esr (tm &optional →))

(def-function-class w (tm instance &optional →))
(def-function-class esw (tm instance &optional →))

(def-function-class r◧ (tm &optional →))
(def-function-class esr◧ (tm &optional →))

(def-function-class w◧ (tm instance &optional →))
(def-function-class esw◧ (tm instance &optional →))

(def-function-class cue-leftmost (tm &optional →))

(def-function-class s (tm &optional →))
(def-function-class a (tm instance &optional →))
(def-function-class on-leftmost (tm &optional →))
(def-function-class on-rightmost (tm &optional →))

(def-function-class tape-length-is-one (tm &optional →))
(def-function-class tape-length-is-two (tm &optional →))
```

These are pretty much what one would expect given the lead up to this point. **`r`**, **`w`**, and **`s`**, are read, write, and step. **`esr`** and **`esw`** perform a read or write on the right neighbor. TM commands either operate on the leftmost of the tape, the rightmost of the tape, the cell under the head, or on an area to the right of the head. Commands that operate on the leftmost cell have a ◧ suffix on their names. So for example, **`r◧`** , reads the leftmost cell. The 'operate on the area to the right of the head' type of operators came into being when we supported an expanding tape. The cascading implications suggested it was best for **`esr`** and **`esw`** to be primitive.

`r` and `w` typically do not require continuations, as the head is always on some cell, and in the presence of our contract with the programmer that he or she always do a write on a given cell before a read on that same cell, there can be no error condition associated with these operations. In contrast, `s` has a `➜rightmost` routine that is invoked when one attempts to step right from rightmost. If `➜rightmost` is not called, then `➜ok` is called. Note it always works this way, that continuations are exhaustive of all cases and that at least one of the continuations will always be called as the last thing that a routine does. Though continuation lists are not fixed, routines that operate on specializations might have additional continuations than the those that operate on the corresponding more general types.

We cannot see which continuations are implemented by looking at the TM interface routine declarations. The only thing we will see is the continuation list parameter, '`➜`'. To see which continuations are implemented we must look at the definitions. Typically there will be a base list of legal continuations available for the most general type, with possibly more continuations available for routines defined against more specific types.

Here are the routines in the `'tm-list` class:

♦

```
(defun-typed r ((tm list-tm) &optional ➔)
 (destructuring-bind
  (&key
   (➔ok #'echo)
   )
(defun-typed esr ((tm list-tm) &optional ➔)
 (destructuring-bind
  (&key
   (➔ok #'echo)
   (➔rightmost (λ()(error 'step-from-rightmost)))
   &allow-other-keys
   )
(defun-typed w ((tm list-tm) instance &optional ➔)
 (destructuring-bind
  (&key
   (➔ok (be t))
   &allow-other-keys
   )
(defun-typed esw ((tm list-tm) instance &optional ➔)
 (destructuring-bind
  (&key
   (➔ok (be t))
   (➔rightmost (be ∅))
   &allow-other-keys
   )
```

♦

```
(defun-typed cue-leftmost ((tm list-tm) &optional ➔)
 (destructuring-bind
  (&key
   (➔ok (be t))
   &allow-other-keys
   )
(defun-typed s ((tm list-tm) &optional ➔)
 (destructuring-bind
  (&key
   (➔ok (be t))
   (➔rightmost (be ∅))
   &allow-other-keys
   )
(defun-typed a ((tm list-tm) instance &optional ➔)
 (destructuring-bind
  (&key
   (➔ok (be t))
   &allow-other-keys
   )
```

♦
```
(defun-typed r◧ ((tm list-tm) &optional →)
 (destructuring-bind
  (&key
   (→ok #'echo)
   )
(defun-typed esr◧ ((tm list-tm) &optional →)
 (destructuring-bind
  (&key
   (→ok #'echo)
   (→rightmost (λ()(error 'step-from-rightmost)))
   &allow-other-keys
   )
(defun-typed w◧ ((tm list-tm) instance &optional →)
 (destructuring-bind
  (&key
   (→ok (be t))
   &allow-other-keys
   )
(defun-typed esw◧ ((tm list-tm) instance &optional →)
 (destructuring-bind
  (&key
   (→ok (be t))
   (→rightmost (be ∅))
   &allow-other-keys
   )
```

♦

```
(defun-typed on-leftmost ((tm list-tm) &optional ➜)
 (destructuring-bind
  (&key
   (➜t (be t))
   (➜∅ (be ∅))
   &allow-other-keys
   )
(defun-typed on-rightmost ((tm list-tm) &optional ➜)
 (destructuring-bind
  (&key
   (➜t (be t))
   (➜∅ (be ∅))
   &allow-other-keys
   )
(defun-typed tape-length-is-one ((tm list-tm) &optional ➜)
 (destructuring-bind
  (&key
   (➜t (be t))
   (➜∅ (be ∅))
   &allow-other-keys
   )
(defun-typed tape-length-is-two ((tm list-tm) &optional ➜)
 (destructuring-bind
  (&key
   (➜t (be t))
   (➜∅ (be ∅))
   &allow-other-keys
   )
  ➜
  (if
   (∧ (cdr (tape tm)) (cddr (tape tm)))
   [➜t] [➜∅]
   )))
```

`on-leftmost` and `on-rightmost` are Boolean routines of convenience. `on-leftmost` takes the true continuation when the head is on the leftmost cell, otherwise it takes the false continuation. `on-rightmost` is follows the true continuation when the head is on the rightmost cell, otherwise it takes the false continuation.

♦
```
(on-leftmost tm
 {
  :➜t (λ() code to be executed when on-leftmost is true)
  :➜∅ (λ() code to be executed when on-leftmost is false)
  })
```
If one wants conventional behavior from these tests, then use the **`be`** routines:

♦
```
(on-leftmost tm {:➜t (be t) :➜∅ (be ϕ)})
```
This call will return `t` or `ϕ`. Actually, these are already the defaults.

♦
```
(on-leftmost tm)
```

tm-generic.lisp

This file contains more declarations for routine classes on the **`tm-machine`** interface. However, unlike for **`tm-decl-only.lisp`**, the file also has generic routine definitions. The generic routines exist to give implementations something to optimize. Otherwise compound operations can be specified.

♦
```
(def-function-class cue-rightmost (tm &optional ➜)
 (:documentation
  "Cue tm's head to the rightmost cell."
  ))
(defun-typed cue-rightmost ((tm tape-machine) &optional ➜)
 (declare (ignore ➜))
 (labels(
      (work() (s tm {:➜ok #'work :➜rightmost (be t)}))
      )
  (work)
  ))
```
This default implementation for **`cue-rightmost`** might not be the fastest one, but it will always work. A programmer is free to implement a faster version for specialized argument types. In general we need not implement routines in these classes, but are free to do so if we see some advantage in it. Here are more of the **`tm-generic.lisp`** routine classes and routine definitions:

♦

```
(def-function-class as (tm instance &optional ➜)
 (:documentation
  "Like #'a, but tm is stepped to the new cell
  "))
(defun-typed as
 (
  (tm tape-machine)
  instance
  &optional ➜
  )
 (destructuring-bind
  (&key
   (➜ok (be t))
   (➜no-alloc #'alloc-fail)
   &allow-other-keys
   )
  ➜
  (a tm instance
   {
    :➜ok (λ()(s tm {:➜ok ➜ok :➜rightmost #'cant-happen}))
    :➜no-alloc ➜no-alloc
    })
  ))
```

♦

```
(def-function-class a&h◨ (tm instance &optional ➜)
 (:documentation
  "#'a with a contract that the head is on rightmost.
  "))
(defun-typed a&h◨
 (
  (tm tape-machine)
  instance
  &optional ➜
  )
  (a tm instance ➜)
  )
(def-function-class as&h◨ (tm instance &optional ➜)
 (:documentation
  "#'as with a contract that the head is on rightmost.
  "))
(defun-typed as&h◨
 (
  (tm tape-machine)
  instance
  &optional ➜
  )
 (as tm instance ➜)
 )
```

Quantifiers

I build the quantifiers on top of a self recursion (looping) routine, ⟳, called “do”.

♦

```
(defun ⟳ (work)
  (labels(
      (again() (funcall work #'again))
      )
  (again)
  ))
```

The first and only argument passed to **work** is a routine, which when called, will cause work to be called again. The return value from ⟳ will be the return value from the last call to **work**.

Actually the library does not implement the definition given above, because I found that the compiler was not optimizing out the recursion. As of version 0.7 the implementation is:

♦
```
(defun ⟳ (work)
 (let(return-value)
  (tagbody
   again
   (setf return-value (funcall work (λ()(go again))))
   )
  return-value
  ))
```
Existential quantification is based on ⟳:

♦
```
(defun ∃ (tm pred &optional (➜t (be t)) (➜∅ (be ∅)))
  (⟳
    (λ(again)
      [pred tm ➜t (λ()(s tm {:➜ok again :➜rightmost ➜∅}))]
    )))
```
When `pred` is true, the quantifier exits with `➜t`. Otherwise `pred` continues on to step the machine and to call `pred` again. Should the step go right of rightmost, the quantifier exits with `➜∅`.

∃ stands for 'there exists' and it has multiple intuitive interpretations. One can think of it as a short circuiting logical OR. It can be thought of as 'loop until'. It is also a linear search for an instance with a particular property. For example, Diognese wanders Greece looking for a good person. He stops his search upon finding such a person.

We build universal quantification from existential quantification:

♦
```
(defun ∀ (tm pred &optional (➜t (be t)) (➜∅ (be ∅)))
 (∃ tm (λ(tm ct c∅)[pred tm c∅ ct]) ➜∅ ➜t)
 )
```
∀ stands for 'for all'. This can be thought of as a short circuiting logical AND, as loop while, or as a check that all is good.

Here are two trivial predicates, presented so you can see the general form:

♦
```
(defun always-false (tm ➜t ➜∅)
 (declare (ignore tm ➜t))
  [➜∅]
  )

(defun always-true (tm ➜t ➜∅)
 (declare (ignore tm ➜∅))
  [➜t]
  )
```
The star suffixed version ∃* traverses until the end of the tape and returns a pair, the first of the pair being a count of the tests done, and the second being the number of tests that were true.

The star suffix version ∀* of also traverses to the end of the list. It does not return anything.

The quantifiers begin with the cell the head is on for the tape machine that is passed in. Versions with a prefix of '`c◧`', which is the cue-leftmost command, will first cue to the leftmost of the tape, i.e. `c◧∃`, `c◧∀`, `c◧∃*`, `c◧∀*`.

If one wants to collect results, do so on a tape machine that is available in the closure. One would typically use the `as` command to do so.

The quantifiers may be used to implement the same functionality as `map` and as `while` loops. In a later chapter we discuss machines that create ranges of numbers. When used with these machines the quantifiers can be used to create `for` loops. Note also, the `ensemble` machine which is used to bind multiple machines together so they step as one machine.

This routine will return true:

♦
```
(defun test-∃-0 ()
 (let*(
     (y {1 2 {3 4} 5})
     (ytm (mk 'list-tm {:tape y}))
     )
  (∧
   (∃ ytm
    (λ(tm →t →∅)
     (if
      (∧ (typep (r tm) 'cons) (eql 3 (car (r tm))))
      [→t]
      [→∅]
      )))
   (equal (r ytm) '(3 4))
   )))
```
So will this one:

♦
```
(defun test-∃-1 ()
 (let*(
     (y {1 2 {3 4} 5})
     (ytm (mk 'list-tm {:tape y}))
     )
  (∃ ytm
   (λ(tm →t →∅)
    (if
     (∧ (typep (r tm) 'cons) (eql 3 (car (r tm))))
     [→t]
     [→∅]
     ))
   (λ()(equal (r ytm) '(3 4)))
   #'cant-happen
   )))
```
Generators

A generator is an abstract tape machine that is typically used to generate data, such as a sequence of numbers or letters. Generators may be stepped and read like conventional machines, but the value read is that created by an internal routine, rather than something that was written.

Recursive

The recursive machine is given an initial value and a routine. After a cue leftmost command, reading the machine returns the initial value. Upon each step, the provided routine is applied to the prior value. The result can be read by reading the machine.

When invoked, the routine is provided with a value and two continuation routines. Should the routine successfully compute a result, it calls the first continuation and passes it the result. If the routine cannot compute a result, then it calls the second continuation. This is useful for creating finite sequences.

For example, this program returns true:

♦
```
(defun test-recursive-0 ()
 (labels(
      (inc-to-10 (i0 c-success c-fail)
       (let(
          (i1 (+ i0 1))
          )
        (if (> i1 10)
         [c-fail]
         [c-success i1]
         )))
      )
  (let(
     (tm (mk 'recursive {:initial 1 :f #'inc-to-10}))
     (result ∅)
     )
   (∀* tm (λ(tm)
        (setf result (cons (r tm) result))
        ))
   (equal result {10 9 8 7 6 5 4 3 2 1})
   )))
```

Here we use `labels` to locally define the routine `inc-to-10`. When passed a value less than or equal to 10, it adds one to it, and calls `c-success`. Otherwise it calls `c-fail`. Though we are using numbers in this example, we may work with instances of any type for which we can define a routine for.

In Lisp a node in a linked list is called a `cons` cell. Such a cell is pair of values, where the first in the pair is called the `car`, and the second in the pair is called the **cdr**. The **car** holds an integer that might be interpreted as a number or as a reference to a value, while the **cdr** holds either **∅** or a reference to the next **cons** cell. These terms have no special meaning outside of this context. Hence the routine **(cons (r tm) result))** creates a list node with the value read from **tm**, and the reference to the next node in the list being the value of **result**. Or in short, it prepends a value to the result list.

Inside the **let**, we create a recursive machine while passing an initial value of 1, and our **inc-to-10** routine. Then we use a quantifier to run through all the instances in the recursive machine. We use cons to repeatedly prepend the instances to a list. Finally we check the list.

When working with vectors it is common to generate short sequences of integers. Hence, we have packaged the work of making an increment routine with a bound and a recursive machine into a single call, **mk-interval**.

♦
```
(defun test-recursive-3 ()
 (let(
    (tm (mk-interval 1 9 2))
    (v #(8 1 6 3 4 5 2 7 0 9))
    )
  (∀ tm (λ(tm ct c∅)
      (if
       (= (elt v (r tm)) (r tm))
       [ct]
       [c∅]
       )))
  ))
```
Here we use mk-interval to create a recursive machine that will enumerate the integers in an interval extending from 1 to 9. It will count by twos while doing so, thus generating the values 1, 3, 5, 7, and 9. We then use these values to index an array. **elt** is the routine Lisp uses to index into an array. I initialized the array so that the values at 1, 3, 5, 7, and 9, are the same as their index. The quantifier then checks that indeed this is true in all cases for tm.

With TM we use generators and quantifiers rather than **for** loops, or **dotimes** loops.

We have also defined a routine for making the Natural Numbers, as this was used as an example earlier in the book, **mk-Natural**.

Here is an interesting, though silly, little program that uses mk-Natural.

♦
```
(defun interesting-0 ()
 (let(
    (i (mk-interval 0 9))
    (N (mk-Natural))
    (count-N 0)
    (count-odd 0)
    (count-even 0)
    )
  (∀* i (λ(i)
      (declare (ignore i))
      (incf count-N)
      (if (oddp (r N)) (incf count-odd) (incf count-even))
      (s N)
      ))
  (∧
   (= count-odd count-even)
   (≠ count-N count-odd)
   )))
```
The program is interesting in that the interval is protecting the loop so that there will be a return value. It returns true because there are five even and five odd numbers encountered. The total numbers traversed is 10, which is not the same as the count of odds. The program is a little bit silly, because iterating through the interval also produces Natural Numbers, so we didn't need N. Here is a more 'pure' use of Natural:

♦
```
(defun interesting-1 ()
 (let(
    (N (mk-Natural))
    (count-N 0)
    (count-odd 0)
    (count-even 0)
    )
  (∀* N (λ(N)
      (incf count-N)
      (if (oddp (r N)) (incf count-odd) (incf count-even))
      ))
  (= count-N count-odd count-even)
  ))
```

This might be a good time to mention that hitting control C twice in Emacs interrupts a process. Though this program is not computable, it is analyzable. When we analyze it, we are surprised to find that there are just as many odd, or even, Natural Numbers as there are Natural Numbers in total, i.e. that the routine returns true. Apparently, somehow, on the way to infinity, the odd and the even counters catch up to the N counter.

Specializations

As we mentioned in the prior chapter, by simultaneously supporting multiplexing and a cell delete command, we have created the possibility of colliding operations, i.e. a *collision hazard*. We can avoid this hazard by leaving out the delete command, or alternatively, by not allowing multiplexing. These choices lead to a specialization hierarchy for the interface:

♦

```
                 tape-machine
                /            \
    nd-tape-machine     solo-tape-machine
                \              /
                haz-tape-machine
```

The **nd-tape-machine** type class does not have a delete command, but it does support multiplexing. When we do not provide a delete cell command on the interface, the programmer cannot create an event where the cell under the head gets deleted. Such an interface can have multiplexing without a collision hazard.

It is instructive to go through the code base for the TM Library and take note of how much simpler the routines belonging to the **nd-tape-machine** routine class are compared to other machine types. Although non-destructive programming is simpler, it leads to inefficiencies due to unnecessary recopying of data structures.

Although an **nd-tape-machine** does not have a delete cell command, a delete cell command may be emulated a by moving all the data to the right over by one cell.

The `solo-tape-machine` includes commands for cell deletion, but not for multiplexing. If we never have more than one head (i.e. never more than one iterator), then one head cannot be used to delete the cell to the right and wipe out a cell that a second head is on. Consequently there can be no collision hazard.

Without multiplexing we cannot make copies of the head state (i.e. cannot have an additional machine head), and this prevents us from writing some library routines that transparently operate on their tape machine arguments.

Many of the routines present in the `nd-tm-decl-only` and `nd-tm-decl-generic` source files are not available in the `solo-tape-machine` routine class. These missing routines include a simple print routine. The print routine is missing because it wants to make an entangled copy so it may transparently cue the head to leftmost.

So at the first level of specialization there is an either-or situation, the programmer can have multiplexing, but not deletion, with `nd-tape-machine`; or have deletion but not multiplexing with `solo-tape-machine`. But, what if we do an evil thing, and diamond inherit both of these types into a third type? Then we get `tm-haz` where 'haz' is short for hazardous. With `tm-haz` it is possible to break the machine by calling `entangle` to get a second head, and then to use that second head to delete the cell the first head is on. Hence, `tm-haz` machines may only be used without potential for errors when either a) there is a proof that such a collision will never happen b) they are controlled from a second order machine such as the `ea-tm` or its specializations.

nd – multiplexing but non-destructive

The non-destructive programming model interface is defined in the 'nd-tm' files. As mentioned above, there are no commands on the interface to delete (deallocate) a cell from the tape.

♦
```
(def-type nd-tape-machine (tape-machine)())
```
Non-destructive machines support having multiple heads on the same tape without tape structural hazards, because there will never be a time when one head attempts to delete the cell that another head is accessing. To place a second head on a tape we make a new machine that shares the first machine's tape. When two machines share the same tape, we say they are *entangled*. To create an entangled machine call the routine `entangle`.

♦
```
(def-function-class entangle (tm-orig &optional ➔))
```
We pick up a couple of new predicates, one to test if two machines are entangled, and another to test if two entangled machines have their heads on the same cell.

♦
```
(def-function-class entangled (tm0 tm1 &optional ➔))
```

♦
```
(def-function-class
  heads-on-same-cell
  (tm0 tm1 &optional ➔)
)
```
Given the ability to make entangled copies we can implement some new routines.

♦
```
(defun-typed s≠
 (
  (tm0 nd-tape-machine)
  (tm1 nd-tape-machine)
  &optional ➜
  )
 (destructuring-bind
  (&key
   (➜ok (be t))
   (➜rightmost (be ∅))
   (➜bound (be ∅))
   &allow-other-keys
   )
  ➜
  (heads-on-same-cell tm0 tm1
   {
    :➜t ➜bound
    :➜∅ (λ()(s tm0 {:➜ok ➜ok :➜rightmost ➜rightmost}))
    })))
```
s≠ is similar to **s**, but will take a bound continuation when one attempts to step beyond the bound set by **tm1**.

The routine **a◨** adds a new cell to the rightmost of the tape. Inside this routine an entangled copy is made, the entangled copy is then cued to the rightmost, a cell is appended, and then the entangled copy is abandoned. The caller's state is not modified.

♦
```
(defun-typed a◨
  (
   (tm nd-tape-machine)
   instance
   &optional ➜
   )
  (destructuring-bind
   (&key
    (➜ok (be t))
    (➜no-alloc #'alloc-fail)
    &allow-other-keys
    )
   ➜
   (let(
      (tm1 (entangle tm))
      )
    (c◨ tm1)
    (a tm1 instance {:➜ok ➜ok :➜no-alloc ➜no-alloc})
    )))
```
We also pick up some quantified routines, such as `esnr`:

♦
```
(defun-typed esnr
 (
  (tm nd-tape-machine)
  index
  &optional ➜
  )
 (destructuring-bind
  (&key
   (➜ok #'echo)
   (➜rightmost
    (λ(index)
     (declare (ignore index))
     (error 'step-from-rightmost)
     ))
   &allow-other-keys
   )
  ➜
  (let(
     (tm1 (entangle tm))
     )
   (sn tm1 index
    {
     :➜ok (λ()[➜ok (r tm1)])
     :➜rightmost (λ(n)[➜rightmost n])
     }
    ))))
```

Here `esnr` stands for 'entangled copy, step n times, and read'. This routine allows a program to read an instance n cells away from the current head position, without moving the head of the machine passed in. There are also analogous write and append versions of this routine.

♦
```
(def-type solo-tape-machine (tape-machine)())
```

solo – destructive operations allowed but uniplex

The 'solo' machine does not have an entangle command on its interface, so there will be exactly one head on a given tape. Consequently its controller will be a uniplex machine. Without multiple heads there can be no hazardous situations where deleting a cell would break the machine.

However, without the entangled copy routine it is not possible to create temporary machines that go off and do some work without perturbing the head position of the original machine.[16] Consequently we can't have routines like `a◨` which internally does an entanglement copy.

We gain these primitives:

♦
```
(def-function-class a◧ (tm instance &optional →))
(def-function-class d (tm &optional spill →))
(def-function-class d◧ (tm &optional spill →))
```

`a◧` appends a new cell to the leftmost of the tape. The new cell is initialized to `instance`. `d` deletes (deallocates) a cell, and `d◧` deletes the cell at the leftmost of the tape. The `spill` machine is optional, if it is present, the instance spilled from the deallocated cell is appended to it. Upon success the delete commands will call the `→ok` continuation with a single parameter of the spilled cell. The `→ok` continuation defaults to `#'echo`. There is a bit of an awkward situation in that to provide continuations we must also provide a `spill` machine, as optional

16 On an implementation that supports left stepping, it would be possible to step the head and count the steps, to perform an operation, and then step the head the opposite direction using the same count. However, the problem doesn't go away, as if the cell the head was originally on gets deleted, we cannot return the head to that cell. .. and what of d leftmost when the head is on leftmost ..

parameters occur in order. In such a situation, if there is not a **`spill`** machine, then set **`spill`** to ∅.

Implementations

Implementation specialization tree

Our first implementation uses a singly linked list for a tape. A singly linked list has the disadvantage that stepping left is not possible. However, lisp programmers have a lot of experience in working around this limitation. The primitive routines without implementations appear in a specialization hierarchy, so we add our specialization as leaves in that hierarchy. We will have an analogous hierarchy for each implementation. It ends up looking like this:

♦

```
                  [tape-machine]
                              \
                           list-tm
                           /      \
                          /        \
[nd-tape-machine]        /          \      [solo-tape-machine]
              \         /            \         /
               list-nd-tm            list-solo-tm
                         \           /
                          \         /
      [haz-tape-machine]   \       /
                        \   \     /
                        list-haz-tm
```

The machines shown in square brackets are the interface definitions we have already talked about. The other machines on this diagram are specialized to provide a single direction, right going only, linked list implementations.

File Name and Routine Name Convention

The code discussed in this chapter is organized into directories. The src-list directory holds both the interface definition code and the single direction and bi-direction list implementations.

[src- | test-]implementation

implementation:: list …

Each source directory starts literally with 'src-' and then the implementation name follows. Each source directory is paired with a test directory. The test directory holds files with the same names as those in the source directory, and the contents of those files are tests for the routines found in the corresponding source files. Additional test files will also be in the directory.

The files are pulled together in the tm.asd file. The tm package uses asdf for build management.

TM source file names have four parts. If a part is empty, then it does not appear in the name.

[implementation][hedge]tm-category.lisp

implementation:: ε | list- | array- | ….

hedge:: ε | nd- | solo- | ea- | ts-

category:: describes types of routines held in the file

For example, *tm-def.lisp* holds *def* type files. This file holds general tape-machine source code, that which has no implementation, and does not have either delete cell, nor multiplexing. As another example, *list-solo-tm-primitives.lisp*. is a file that holds routines for a list implementation that uses the solo hedge to avoid the collision hazard. The file holds primitive routines.

Types without implementations have tape machine spelled out, **`tape-machine`**, **`nd-tape-machine`**, etc. In implementations and for file names, tape machine is always abbreviated as `tm`.

TM test file names are of the form:

test-name-series

name:: name of the routine being tested or a descriptive name

series:: a number, typically sequential starting from zero.

Test routines return t when they pass. If there is an exception or the test returns something else, it has failed. To add a test to the regression suit, use the following command:

♦
```
(test-hook routine-name)
```

Where *routine-name* is the name of the routine being added to the regression. To run the regression type:

♦
```
> (test-all)
```

list-tm types

We may make instances of these types by calling **mk**.

♦
```
(def-type list-tm (tape-machine)()))
(def-type list-nd-tm (nd-tape-machine list-tm)()))
(def-type list-solo-tm (solo-tape-machine list-tm)())
(def-type
  list-ea-tm
  (ea-tape-machine list-nd-tm list-solo-tm)
  ())
```

Here are some examples. All of our test routines return true, **t**. The tests are loaded with the library, and can be run with the command **(test-all)**.

♦
```
(defun test-heads-on-same-cell-0 ()
 (let*(
      (tm0 (mk 'list-nd-tm {7 2 -3}))
      (tm1 (make-instance 'list-nd-tm))
      )
  (init tm1 tm0) ; entangles tm1 with
tm0
  (∧
   (s tm0)
   (¬ (heads-on-same-cell tm0 tm1))
   (s tm1)
   (heads-on-same-cell tm0 tm1)
   (s tm1)
   (¬ (heads-on-same-cell tm0 tm1))
   )))
(test-hook test-heads-on-same-cell-0)
```

♦
```
(defun test-dn-0 ()
 (let*(
     (tm0 (mk 'list-solo-tm {1 2 3 4 5}))
     (tm1 (mk 'list-solo-tm {-100}))
     )
  (∧
   (s tm0)
   (dn tm0 2 tm1)
   (equal (tape tm0) {1 2 5})
   (equal (tape tm1) {-100 3 4})
   (on-rightmost tm1)
   (= (dn tm0 10 ∅ (be -1) #'echo (be -2)) 9)
   )))
(test-hook test-dn-0)
```
♦
```
(defun test-ea-a◧-0 ()
 (let*(
     (tm0 (mk 'list-ea-tm {1 2 3}))
     )
  (with-mk-entangled tm0
   (λ(tm1)
    (s tm0)
    (w tm0 22)
    (a◧ tm0 7)
    (∧
     (equal (tape tm0) {7 1 22 3})
     (equal (tape tm1) {7 1 22 3})
     (¬ (on-leftmost tm0))
     (¬ (on-leftmost tm1))
     (eql (r tm0) 22)
     (eql (r tm1) 1)
     )))))
(test-hook test-ea-a◧-0)
```

Initialization and the first-rest pattern

A tape machine can only be created after there is a first item to place in it. Consequently code that produces data that is placed in a tape machine will often have separate first case and recursive case.

♦
```
(get-first
 {:➜ok (λ(tm)
     (get-rest tm)
     )})
```

In this example, get-first gets a first data item. If that item is not the last item, and there was not an error, get-first creates a tape machine and then calls a continuation with a tape machine initialized to this data item. get-rest then gets the rest of the data items but does not have to create a machine. We call this the 'get-first/get-rest' pattern.

When programming with loops this pattern manifests as 'loop priming'. Accordingly work is done before entering a while or for loop, where this priming work is a variation of the work done inside the loop. Here get-first is the priming part, and get-rest is the loop and its contents part.

Suppose we want to avoid having two routines, but instead just want a single get routine that plays either or both roles. Consider the case where get takes a tm parameter. We can pass in ∅, so that get knows it has to create a new machine.

♦
```
(get tm
 (if (not tm)
  (get-first)
  (progn
   (get-rest tm)
      tm
   )))
```
When we look inside such a get routine, we will find a test on tm to see if it is ∅. The true path will execute code which we may as well call get-first, and the false branch will execute code which we may as well call get-rest. We didn't avoid the get-first/get-rest pattern, rather we just encapsulated it.

However, we have a problem with get. We cannot create a tape machine in a routine and pass it back out though the parameter list, because the tm in the routine is a local variable, and we do not have a reference to the tm in the caller. Consequently we cannot initialize the caller's machine. We can add to the tape when a tm is passed in, but we can't create the first cell. Consequently get we will always have to return tm and we end up with this pattern for using get:

♦
```
(let ((tm ∅))
 (setf tm (get tm))
 )
```
So we find that the `get` with a flag approach adds a redundant test each time it is called, unnecessarily returns `tm`, and must redundantly assign to the outer `tm` on each call. It appears to be better to just use the `get-first/get-rest` pattern directly.

Emptiness

If a machine could hold any number of elements, including being empty, then the `get` routine discussed in the prior section would not require the return value nor outer assignment.

When defining a conventional Turing Machine in the chapter we were lead to make the empty-symbol a distinct property of the Turing Machine, rather than including it in the alphabet. We made the empty-symbol distinct in our definition because it was special from a computation point of view; namely, all the cells of the initial infinite tape held the empty symbol, but a Turing Machine cannot write all the cells on an infinite tape, hence those empty symbols could not have been put there by a Turing Machine computation. Also we discussed the possibility of not having a distinct empty-symbol.

When we discussed addresses we defined the length of an area to be one greater than the differences in between the smallest and greatest addresses. This definition has the nice property that an area consisting of a single cell has a length of 1. We then used analysis to derive a higher order construct where an area might have 0 length. This created a nuanced difference between zero length where we were talking about some area, and that of no area existing at all. We found that a zero length area can still have a location. One might think of it as a locator for growing the area in the future should we choose to do so, or as a marker as to where the area used to be.

Just as there is no way to compute and make an unbounded empty tape, there is also no way to compute and decide if a tape is empty. In analogy to defining an empty-symbol, and an area of zero length, it appears that we will need a higher order construct (i.e. belonging to a higher order analysis) so that we can keep track of an emptiness attribute for the tape as a whole.

Now consider our Turing Machine variation where our tape grows. Is this variation helpful here? What is the minimum sized tape? We never really said how many cells we start with. It should have been apparent that the initial tape must have at least one cell as the head was said to always be over a cell. Perhaps we should have modified our 'always write before read rule', and instead made it something like 'always append a first cell before the first write, and always write before read'. We would have had to have added a flag to signal that the tape had not been created yet, so as to know to append the first cell. Thus, again, we have discovered we need additional information.

The Turing Machine is a mathematical object, whereas the **`tape-machine`** is a computer programming type, and we deal with instances of this type. However, properties of the Turing Machine have implications for our implementation. And accordingly we know that we will need additional state information to create a tape machine which can be empty.

Consider our append routine. It is passed in a tape machine instance, say `tm-0`, and an instance of something to be appended, say *x*. If `tm-0` has never been written, then the append routine should create a leftmost cell and write `x`. This is appending to an empty active area. I.e. an active area of zero length. However, in contrast, if we pass in a machine with the head on the leftmost cell, and that cell has already been written, then our append routine should create a new right neighbor for the leftmost cell, and write `x` into that. Here it is appending to an active area of length 1. Creating a new leftmost cell, and appending to a leftmost cell are fundamentally different operations, and they are being distinguished by the length of the active area. However, to keep track of the length of an active area, while including the possibility for zero length, we again find that we need an external construct.

Our experience with emptiness thus far leads to the conclusion that such a machine that can appear to be empty will simply be repackaging the flag contemplated in the prior section during the discussion of the get-first / get-rest pattern. That is to say we will just be moving the logic around, and the pattern remains in tact.

Signaling emptiness with a ∅ tape slot

A machine in the empty state would have no cells on its tape. In the specific case of the list-tm machine, which employs a Lisp list to implement the tape, we are tempted to simply use a null pointer in the tape slot to signal that the machine is empty. The cost is pretty low, in the case of a null pointer we remove one possible encoding for a pointer value that otherwise might be used to access memory. Actually, in Lisp, because we have typed pointers, we could use a reference to any instance that is not of cons cell type to signal emptiness.

Signaling emptiness by using a special instance in Lisp, or the use of a null pointer, is practical. However, we have another issue with the special tape slot (tape field) value approach - we do not require that all machines have a tape slot, and when a machine has a tape slot, we do not dictate how it is used. In other words, we are specifying an architectural interface, not an implementation. Consequently the best we can do is to define a new routine class as part of the interface and leave it to those who implement the functionality behind the interface to find a way to express emptiness for any tape machine type they happen to make. However, it would be much better to have an elegant solution at the architectural level instead of leaving it as an open question and relying upon ad hoc solutions.

Signaling emptiness with a ∅ machine reference

Instead of using `∅` as a tape reference when the machine is empty, suppose instead we use a `∅` reference to a machine to mean the machine is empty. This would overload `∅` to mean both that there is no tape machine, or that the tape machine is empty. We might generalize and say that `∅` means no data is available, no matter what the reason is.

If we were to do this, argument passing would be a challenge. We discussed this issue in the prior section about the **`get-first/get-rest`** pattern. Lets look at this again a little deeper.

♦
```
(defun c7-0 (x) (setf x (cons 7 x)))
(defun c7-1 (x) (cons 7 x))
(defun c7-2 (x) (append x {7}))
(defun c7-3 (x) (nconc x {7}))

(let(
   (a0-0 ∅)
   (a0-1 ∅)
   (a0-2 ∅)
   (a0-3 ∅)
   (a1-0 {'a})
   (a1-1 {'a})
   (a1-2 {'a})
   (a1-3 {'a})
   )

 (c7-0 a0-0)
 (c7-1 a0-1)
 (c7-2 a0-2)
 (c7-3 a0-3)

 (c7-0 a1-0)
 (c7-1 a1-1)
 (c7-2 a1-2)
 (c7-3 a1-3)

 (print "a0-0:")(princ a0-0)
 (print "a0-1:")(princ a0-1)
 (print "a0-2:")(princ a0-2)
 (print "a0-3:")(princ a0-3)

 (print "a1-0:")(princ a1-0)
 (print "a1-1:")(princ a1-1)
 (print "a1-2:")(princ a1-2)
 (print "a1-3:")(princ a1-3)
 )
```

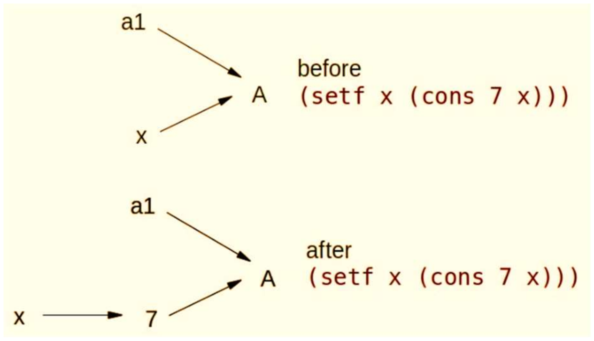
Here our intention is that routine `c7` will add the numerical instance 7 to a list. We have created four versions of `c7`. In the first attempt we try to prepend a value by creating a cons cell and assigning that to the passed in argument. The second uses the cons without the setf. The third call attempts to append the 7 to the list using Lisp's `append` routine. The fourth one appends the 7 using Lisp's `nconc` routine. All four accept `x`, the list to be modified. We then call our four routines with `∅` so `c7` will be initializing the list, then again with a symbol, `'a`, so that `c7` will be modifying the list. We then print the results:

♦
```
"a0-0:" NIL
"a0-1:" NIL
"a0-2:" NIL
"a0-3:" NIL
"a1-0:" (A)
"a1-1:" (A)
"a1-2:" (A)
"a1-3:" (A 7)
```

None of the attempts at initialization worked. Only the last attempt at modification worked. The reason the routine cannot initialize the list passed in is that it has no reference to the list. **x** only holds the value **∅**. There is no location information in **∅**, and it is impossible to initialize something when you don't know where it is.

Now consider the case where we try to modify the list, initially we have a reference to a list in one of the '**a1** variables', **a1-0**, **a1-1**, … etc. When we pass this reference into **c7**, the reference is copied. Then **x** holds a reference to the same instance that the **a1** variable referenced.

In the first routine, **c7-0**, when we **setf** on **x** it modifies **x**, and so **x** no longer points to the original instance. However the **a1** variable is still pointing to the original instance.

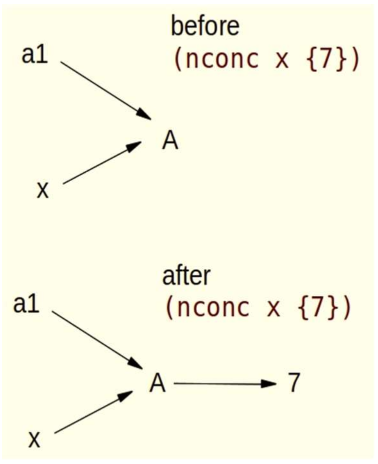
♦


From the caller's point of view, nothing has changed. In the second routine, **c7-1**, **cons** returns a value, but nothing is done with it.

In the third attempt to modify the list, **c7-2**, **append** is called with **x** and 7. However, **append**, like **cons**, is non-destructive. It simply returns a new list that is a shallow copy with the new value, 7, appended. This is returned, but we don't use it, so nothing happens.

In the fourth and successful attempt, `c7-3`, `x` points to the same instance as the `a1` variable in the caller. This is the first element in the list. We do not try to change this. Instead we go to that first element, and we modify *it*. `nconc` is destructive (i.e. it does not copy the list like `append` does). Afterward both the `a1` variable and `x` are pointing to the same modified first element, and that element points to the new value.

♦



Thus using `∅` to indicate emptiness requires either 1) that we add some sort of level of indirection so that callers and routines are referencing the same instance, and then the common instance is modified, or 2) the caller somehow gets a reference to the original variable in the caller.

Programmers using the C language often implement option 2 by passing in a pointer to the `a1` variable (or whatever variable is being used) instead of passing in the variable value directly. In this way the routine has a reference to the callers version of `a1` and can use that reference to set `a1` to `∅`. Inside of such a routine one must dereference the passed in value to see `a1`. As `a1` is also a reference, one must dereference twice to see the value `a1` is pointing to. We can do something similar in Lisp by passing the argument 'in a box'. Such a box is defined in the **`src-0/functions.lisp`** file.[17]

Of course, macros actually do have access to the caller's variables.

♦
```
(defmacro c7 (x) `(setf ,x (cons 7 ,x)))

* (let(
   (a0 ∅)
   (a1 {'a})
   )
 (c7 a0)
 (c7 a1)
 (print "a0:")(princ a0)
 (print "a1:")(princ a1)
 )


"a0:" (7)
"a1:" (7 A)
```
As another issue with using a ∅ reference to mean empty is that we are forced to pay the initialization costs for building the container at the point the first item is added, rather than incurring that cost at a place of our choosing. For a Lisp list, the container overhead correlates directly to the number of cells, and a zero cell list has zero overhead, so distinguishing between an empty list and having no list is at best moot. In contrast, our tape machines have top level structure that exists even when the tape is empty. Depending on the tape machine implementation, this top level structure might be complex, and might take some time to build.

Also when we use a ∅ reference to mean empty, we cannot hold the overhead state from a prior emptied machine to be reused when a new instance is added. We are forced to redo the initialization each time the machine transitions from empty to not empty.

As a third issue, if we want to set the reference to a container to ∅ to mean the container is empty, then we must set *all* the references to the container to ∅, not

[17] The box implementation is taken from dmitry_vk's comment on Stack Exchange.

just the one we would normally be passed as an argument. The reference to a reference approach, or the padding cell approach, works well for solving this problem, as all the secondary references will point to a single container reference, and only that single container reference need be modified.

Signaling emptiness using a padding cell

We used the 'option 1' mentioned in the prior section, that of using a padding cell, to facilitate signaling emptiness in the version 0.1 of the Lisp TM Library. This technique can still be used of course. The programmer need only ignore the leftmost cell in machines. If a machine has only a leftmost cell, then by this technique the machine is interpreted to be empty. The need for this appears most commonly when a machine is passed into a routine so as to capture a result, because we must first create an empty machine (no result yet) and then pass it into the routine. The padding cell approach works nicely with our order 1 machines because they already have a requirement of having at least one cell.

Here is a version of our c7 program that uses a padding cell.

♦
```
(defun c7 (x) (setf (cdr x) (cons 7 (cdr x))))
```

This sets the reference to the cell after the padding cell, to a node created with cons. That node has the value 7 and then points to the rest of the list. The only difference from our prior code is that `x` has been replaced with `(cdr x)`. This is done to skip the padding cell. This produces the desired behavior.

♦
```
(let(
   (a0 {'padding})
   (a1 {'padding 'a})
   )
 (c7 a0)
 (c7 a1)
 (print "a0:")(princ (cdr a0))
 (print "a1:")(princ (cdr a1))
 )


"a0:" (7)
"a1:" (7 A)
```
Note we also use `cdr` in the print to skip the padding cell. Had we not done this, the lists would have printed with a first entry of `'padding`.

Should we delete all the cells in the list, except the padding cell, the list will be considered empty. All references to the list are pointing to the padding cell, so they don't need to be updated.

The padding cell approach is related to the reference to a reference. The second reference being embedded in the cell. Though we also gain a data value which can be used to hold a property list or something similar.

The padding cell approach is uniform with the structure of the tape when the underlying data structure is a list, but will not be so when it is a different structure, such as an array or hash table. In those cases we might end up with a cons cell used for padding that then points to an array or hash table.

Using a second order machine

In the next chapter we will discuss second order tape machines which carry status, such as being empty, while wrapping a base machine. The base machine may be of any type. The second order machine has its own tape machine interface, and watches the operations that goes through it while passing most of them to the first order base machine. We use subtypes to represent state (status), so the cost is partly hidden by CLOS, which has to do the dispatch work anyway.

However, we still have an extra layer of dereferencing when the base machine is accessed.

Second Order Machines

Status - status-tm

Rather than having state bits, we let CLOS keep track of the state by keeping multiple specialized types for a **`status-tm`**:[18]

1. abandoned
2. empty
3. parked
4. active

An abandoned machine is one that has been especially marked as being left for garbage collection. An `empty` machine acts like there are no cells on the base machine tape. A `parked` machine acts like the head is not on the tape. Though the head is conceptually not on the tape, thus not on any cell, it does have a right neighbor, and that neighbor is the leftmost cell of the tape of the base machine. This definition for a parked head gives us the ability to use the area definition of an area being to the right of the head, and to still have an area that includes the leftmost cell. Note this additional structure is maintained by the higher order construct, i.e. the status machine, and not by the base machine. The base machine definition does not change.

[18] 'Status' means the same thing as 'state', but the word state is already overloaded due to the state machine in the Turing Machine definition. I.e. we are calling state in the second level 'status' so as to distinguish it from state in the first level.

♦
```
(def-type status-tm (identity-tr)
 (
  (base ; the machine being managed
   :initarg :base
   :accessor base
   )
  (address ; an integer address locating the head
   :initarg :address
   :accessor address
   )
  (address-rightmost ; address of the rightmost cell
   :initarg :address-rightmost
   :accessor address-rightmost
   )
  ))

(def-type abandoned (status-tm)())
(def-type active   (status-tm)())
(def-type empty    (status-tm)())
(def-type parked   (status-tm)())
```

Because we are analyzing the operations performed on the base machine, we can easily see the steps and keep an address integer, so we take the opportunity to do that. This makes it possible to compare locations between status machines independent of the base type, and it speeds up simple locations tests and distance calculations.

The programmer only makes `status-tm` machines. Though internally a machine is always one of the specialized types, `abandoned`, `empty`, `parked`, or `active`. These specialized types have no new slots, rather the type tags are just being used to tell CLOS which interface to use.

♦
```
(defun test-status-1 ()
 (let*(
     (t0 (mk 'list-solo-tm {:tape {1 2 3}}))
     (t1 (mk 'status-tm {:base t0}))
     )
  (∧
   (park t1)
   (= (d◧ t1) 1)
   (= (d t1) 2)
   (s t1)
   (= (r t1) 3)
   (park t1)
   (= (d t1) 3)
   (typep t1 'empty)
   )))
```
In this program we create and initialize a solo machine, `t0`. As always the first order machines must have at least one value. Then we create a status machine, `t1`. We park the head of the status machine, and then delete the leftmost instance on the tape, i.e. the 1.

According to our convention though a parked head is not on any cell, the right neighbor of the parked head is the leftmost cell of the tape, so the second delete also deletes the leftmost cell, which now holds a 2.

In the next line we step the machine. Stepping the machine will move the parked head to it's right neighbor, i.e. the leftmost cell of the tape. We read that value as a check, and find that it is 3. Then we park the head again, and then delete the last cell. That causes the machine to become empty.

The initial status of a machine can be created set to empty using `:status 'empty`, or to parked by using `:status 'parked` in the initialization line. First order machines must have at least one cell, so when we want an empty status machine we first create a base machine with a single cell holding an instance of `∅`. We could have used any value. When a status machine becomes empty, including in the initializer, the instance for the last cell is set to `∅` to facilitate garbage collection of the last instance. Adding a cell to an empty machine causes the machine to transition to the parked status.

♦
```
(defun test-status-6 ()
 (let*(
     (tm10 (mk 'list-solo-tm {:tape {∅}}))
     (tm11 (mk 'status-tm {:base tm10 :status 'empty}))
     (tm20 (mk 'list-solo-tm {:tape {∅}}))
     (tm21 (mk 'status-tm {:base tm20 :status 'empty}))
     )
  (∧
   (a tm11 101)
   (a tm11 102)
   (a tm11 103)
   (typep tm11 'parked)
   (d◧ tm11 tm21)
   (d◧ tm11 tm21)
   (d◧ tm11 tm21)
   (= (r tm21) 101)
   (c◧ tm21)
   (= (r tm21) 103)
   (s tm21)
   (= (r tm21) 102)
   (s tm21)
   (= (r tm21) 101)
   (¬ (s tm21))
   )
  ))
```

Entanglement accounting - ea-tm

`ea-tm` is a specialization of `status-tm`.

When multiple machines share a tape, we say they are *entangled*. There is a synchronization problem when a set of machines are entangled. If any member of that set becomes empty, all members of the set must become empty. Typically machines keep track of their tape through a reference to the leftmost cell, so if any has a new leftmost cell, or the leftmost cell is deallocated, all must update their leftmost cell reference. Before deleting a cell from the tape while using one machine, we must check that no other machine has a head on that cell. `ea-tm` is a specialized status machine that adds an entanglement accounting system to keep track of all of the above.

Entanglement accounting makes it safe to use a `tm-haz` machine as a base machine. This is good news, as both the `solo-tm` and `nd-tm` interfaces become available simultaneously without collision hazards.

♦
```
(def-type ea-tm (status-tm)
 (
  (entanglements
   :initarg :entanglements
   :accessor entanglements
   )
  (entanglements-pt ; our location in the entanglements list
   :initarg :entanglements-pt
   :accessor entanglements-pt
   )
  ))

(def-type ea-parked-active (ea-tm)())

(def-type ea-abandoned (abandoned ea-tm)())
(def-type ea-active   (ea-parked-active active ea-tm)())
(def-type ea-empty    (empty ea-tm)())
(def-type ea-parked   (ea-parked-active parked ea-tm)())
```

The entanglements slots (fields) are only used internally by the `ea-tm` machine. Each instance on the `entanglements` list is an `ea-tm` machine that shares the tape. `entanglements` is a sort of tape listener list. `entanglements` is a `bilist-haz-tm` machine and is shared by all tape users. `entanglements-pt` indicates the location in the `blist-haz-tm` for the given machine. We don't actually need both of these fields, because `entanglements` can be recovered from `entanglements-pt`. (We simply make a copy of it, and cue it to leftmost.)

The `abandon` routine changes a status machine's status to `abandoned`, and for an `ea-tm` machine it also removes it from the listener list. An entangled machine created using `with-entanglements` will be automatically abandoned when it goes out of scope.

♦
```
(abandon tm)

(with-entangled tm-orig (λ(tm-entangled) body*))
```

I tried weak pointers instead of having an explicit abandon routine, but the weak pointers did not change to `∅` when an entangled machine went out of static scope, say after a let form, but rather became `∅` only after they were garbage collected. I probably should have expected this, as most instances in Lisp have dynamic extent, they stick around until they are no longer used. We only know they are no longer used when the garbage collector runs. This caused machines to remain entangled in places the programmer did not expect them to be. In the case of status machines, one could park the machines when done with them, so this latent existence wouldn't cause a problem with collisions, but I had to ask, if I'm going through the effort to park machines when I'm done with them, why not just abandon them?

An additional continuation is added to the interface, that of **`➜collision`**. This is invoked if one entangled **`ea-tm`** machine attempts to delete the cell that another entangled **`ea-tm`** machine has its head on. It also makes a nice proof target, in some situations, if one can prove that the there can be no collisions, then one can use simpler machines. The delete routine ends up looking like this:

♦
```
(defun-typed d ((tm ea-active) &optional spill ➜)
 (destructuring-bind
  (&key
   (➜ok #'echo)
   (➜collision (λ()(error 'dealloc-collision)))
   &allow-other-keys
   )
  ➜
  (labels(
       (dec-rightside-addresses (tm)
        (c◧∀* (entanglements tm)
         (λ(es)
          (let(
              (etm (r es))
              )
           (when etm
            (decf (address-rightmost etm))
            (when (> (address etm) (address tm))
             (decf (address etm))
             ))))))
       )
   (entangled-on-right-neighbor-cell tm
    {
     :➜t ➜collision
     :➜∅ (λ()
        (d (base tm) spill
         {
          :➜ok (λ(instance)
              (dec-rightside-addresses tm)
              [➜ok instance]
              )
          :➜collision #'cant-happen
          (o (remove-key-pairs ➜ {:➜ok :➜collision}))
          }))
     }))))
```
First we call the predicate `entangled-on-right-neighbor-cell` to check if any machine in the entanglement set has a head on the soon to be deleted right neighbor cell. If so, we take the `➜collision` continuation. If not then we know it is safe to delete the cell, even though the base machines might be of type `tm-haz`. In this implementation of status we also maintain addresses, so after the cell is deleted, we decrement the addresses fields for all the entangled machines that have a head to the right of the cell that was just deleted. We can easily check which they are as we know the address the head is on for our own machine.

The **entangled-on-right-neighbor** routine is just a question of existence for a machine with an address one greater than ours. We are pretty careful here not to create an intermediate address that can't be represented in the same word size as the intermediate bits. Though that is not really necessary in Lisp because the addresses are bignums. Still there is nothing wrong with using good form.

♦
```
(defun-typed
  entangled-on-right-neighbor-cell
  ((tm ea-active) &optional →)
  (destructuring-bind
   (&key
    (→t (be t))
    (→∅ (be ∅))
    &allow-other-keys
    )
   →
  (c◧∃ (entanglements tm)
   (λ(es ct c∅)
    (let(
       (etm (r es))
       )
     (if
      (∧
       etm
       (typep etm 'active)
       (≠ (address etm) 0)
       (= (address tm) (- (address etm) 1))
       )
      [ct]
      [c∅]
      )))
   →t
   →∅
   )))
```
Using **d** to delete cells from an active machine can never make the machine empty.

This is because **d** deletes the right neighbor cell, so we can never delete leftmost. Delete leftmost, **d◧** cannot make an active machine empty either, as the head will be on some cell, and trying to delete that cell will result in a collision. However, status machines can be parked, and a parked machine has no head on the tape. If any of the machines in the entanglement set are empty, then all the machines are, after all they share the same tape. It follows that to make a machine empty, it, and all of its entanglements, must be parked.

♦
```
(entangled-on-leftmost (entanglements tm)
 ➔collision
 (λ()
  (let(
     (spill-instance (ec◧r (base tm)))
      )
   (labels(
        (delete-0 ()
         (if (= (address-rightmost tm) 0)
          (make-empty)
          (progn
           (step-parked-machines)
           (d◧ (base tm) ∅
            {:➔ok (λ(instance)
                 (declare (ignore instance))
                 (fix-tapes-dec-addresses)
                 )
             :➔collision #'cant-happen
             :➔no-alloc #'cant-happen
             })))
         [➔ok spill-instance]
         )
        )
    (if spill
     (as spill spill-instance
       {
       :➔ok #'delete-0
       :➔no-alloc ➔no-alloc
       })
     (delete-0)
     )
    ))))
```
Here is the code for `d◧` on a parked machine. This occurs after the definition of three helper routines, which I have omitted for brevity's sake.

We start with an existence questions similar to that for `d`, does there exist a machine in the entanglement set that has its head on leftmost? If not we begin by recovering the instance that is to be spilled. When then look at the address of rightmost. If it is address zero, then we make all the entangled machines empty. Otherwise we delete the leftmost cell of the machines. We run into a small problem though. The base machines are treated like order one machines, so for the parked machines we have left the base machine heads on the left most cells while ignoring them. To compensate for this we scan though the entanglements set looking for such cases, and if we find one, we bump the head to the right. This will always be possible because at this point we know the machines will not be empty after the delete.

Here is the code for **make-empty**.

♦
```
(make-empty ()
 (w (base tm) ∅)
 (c◧∀* (entanglements tm)
  (λ(es)
   (let(
      (etm (r es))
      )
    (when etm (to-empty etm))
    ))))
```

It is defined in the **labels** section, so there is no **defun** keyword. The first thing that it does is to write ∅ to the leftmost cell of the base machine. We do this so that the instance held in the last cell will be garbage collected. We don't actually delete the last cell because that would violate the constraints for order one machines, which must always have at least one cell. Keeping the last cell around also makes it easy to restart an empty machine by just writing a new instance to this last cell and changing the status to active.

Once the last cell's instance is freed, we walk through the entanglement list and call **to-empty** on each machine. **to-empty** is a routine class, and it will be matched with an implementation at dispatch time. For ea-tm machines it will call this one routine:

♦
```
(defun-typed to-empty
   ((tm ea-tm))
   (change-class tm 'ea-empty)
)
```

Both **ea-active** and **ea-empty** are inherited from **ea-tm** and neither have any additional slots, so we expect the **change-class** to be safe and not to require much work. All off the machine states (statuses) have analogous routines. These are primitive instructions that are used internally. There are higher order versions for abandoning and parking machines available to the programmer. Other status changes occur as side effects of operations.

Machines often reference the tape via a pointer to the first cell in the tape. If we orphan the first cell from the tape list, the other machines will no longer be able to read the tape. It would be sufficient to call `d◧` on each of the entangled machine, but instead we chose to add a routine to the first order machine's interface to tell each machine that there was a delete, and to hand it a copy of the machine with the delete, `(update-tape-after-d◧ (base etm) (base tm))`. This is found inside of `fix-tapes-dec-addresses`.

Analogously when we have multiple machines sharing a tape, and we prepend a new cell to the front of the tape on one machine, the other machines would not know about this change, and thus would have a different, erroneous, view of the tape. This is why prepend is not a command for tape-machine or nd-machine. Hence we have an analogous routine, `(update-tape-after-a◧ (base etm) (base tm))` that is run after a cell is prepended.

The existence of the update commands causes us to fall a little short of the goal whereby the second order machine uses the programmer interface of the first order machines without modification. Though, this may be indicative of the fact that we need to add some cell sharing routines to the first order. Currently the only routine that explicitly specified to move cells is `d` with its spilling feature, though the cell moving part of that specification is optional, as it must be, because implementations such as arrays cannot transfer cells.

Thread Safe - ts1-tm

`ts1-tm` is a thread safe specialization of `ea-tm`.

As wonderful as the ea-tm is, it still does not guarantee thread safety for status machines. By 'thread safety' we mean that the interface will continue to perform to specification for any order and timing of interface routine calls. Though thread safety is a great attribute for the library, this feature does not help the programmer to synchronize his or her own data. The programmer must still work that out, rather here we just want to make sure that the library doesn't break.

We can already have thread safety with the `nd-tm`. Though it is hard to imagine implementing such a thing as a production code pipe between processes non-destructively. Perhaps if we sent multiple machines as a bucket brigade, as after all, nd machines can be destroyed as whole units by deallocating them.

Any of the machines may be used in a thread safe manner if we guarantee that only one thread at a time has control. Emulated threads are not good enough in this respect, as they time multiplex at too low of a level, so an interface routine might not complete before the next thread runs. Rather we need something like a mutual exclusion lock on the machine, where the owner of the lock may use the machine, and the lock is only exchanged at the same level of execution as the routine calls.

When one thread is deleting a cell from a tape, other threads with entangled copies of the machine cannot move their heads over said cell because they might run off the tracks, so to speak. With arrays that have deleted cell markers it is dangerous to read a cell that is being deleted as the deleted cell will have its instance set to null. When two entangled machine on different threads delete different cells, there can be problems if the cells are neighbors. We have analogous problems on the entanglement list, and this reflects up through the hierarchy, so that it is not safe to simultaneously abandon two machines (because this corresponds to entanglements list deletions), nor to simultaneously abandon a machine and delete a cell (because the cell delete will read the entanglements list while the abandon may be deleting a cell). We also cannot allow head motion on any entangled machine during collision checks, or the collision check may turn out incorrect. The entanglements list is also traversed when prepending a new leftmost cell, and when setting the entangled machines to the empty state, or moving them from empty to parked.

With the **ts-tm** two entangled machines may safely run simultaneously on separate threads. An example would be one process putting data into a pipe, while the other pulls it out. Perhaps they start stepping on each other's work in the middle. We don't want anything to break should that happen.

The simplest algorithm for creating thread safety, which we call 'algorithm 1', is to acquire a common lock for each interface call. Thus each interface routine would run exclusively of the others. This simple approach has low overhead. Overhead is important, because in general, tape machine interface routines don't do much. I explored a couple of more complex algorithms that provided for more parallelism, but it wasn't apparent that under typical use they would be faster.

Hence, ts1-tm simply wraps each call in a '**with-recursive-lock-held**' environment. For example:

♦
```
(defun-typed s ((tm ts1-tm) &optional ➜)
 (bt:with-recursive-lock-held ((deed tm))`
  (call-next-method tm ➜)
  ))
```

We use recursive locks because some routines call other routines in the library. As examples, generic routines are composed entirely of calls to other routines. We would like to have called the less specialized version of these other routines that didn't have the lock wrappers, and thus avoid the redundant lock setting, but CLOS does not provide a facility for doing this.

Streaming

Here is a little worker program used for streaming computation, when called, it squares the number on tm-source, and puts the result on tm-sink.

♦
```
(defun square-worker (tm-source tm-sink)
 (as tm-sink (expt (r tm-source) 2))
 (s tm-source)
 )
```
Suppose we have an infinite input stream, and want to process, say, the first five instances. Here we set the input stream to the Natural Numbers.

♦
```
(let*(
   (tm-source (mk-Natural))
   (tm10 (mk 'list-nd-tm {:tape {∅}}))
   (tm-sink (mk 'status-tm {:base tm10 :empty t}))
   )
 (∀* (mk-interval 0 4)
  (λ(tm)(declare (ignore tm))
   (square-worker tm-source tm-sink)
   ))
 (tm-print tm-sink)
 )
```
We use universal quantification over an interval of five values to effect our count. Upon each count square-worker will do its job. For `status-tm` machines the routine `tm-print` is currently defined to print the status followed by a pair separated by a colon, the first in the pair being the head address, the second being the address of rightmost. This is followed by the contents of the tape. Square brackets appear around the instance representation corresponding to the cell the head is on. This is ad hoc, and likely to change in a later version. In the current version our output appears as:

♦
```
ACTIVE (4:4) 0 1 4 9 [16]
T
```
Notice that we had to add `(declare (ignore tm))` after the lambda on the line with the quantifier. We have no choice, as the quantifier always passes us a copy of the machine, but Lisp thinks there is something funny if a variable is passed into a routine, and is not used, so we have to tell it we are ignoring the variable. We can turn this off, but in other contexts it is useful information. I've thought

about getting the quantified tape machine from the closure, but we often use this feature for giving a name to an evaluated form.

Also notice that square-worker is called with the same input arguments repeatedly. We can make this code more clear by currying those variables out of the loop:

♦
```
(let*(
   (tm-source (mk-Natural))
   (tm10 (mk 'list-nd-tm {:tape {∅}}))
   (tm-sink (mk 'status-tm {:base tm10 :empty t}))
   (worker (λ()(square-worker tm-source tm-sink)))
   )
 (∀* (mk-interval 0 4)
  (λ(tm)(declare (ignore tm))
   [worker]
   ))
 (tm-print tm-sink)
 )
```
Now it is clear that no loop variables are being passed into `worker`. This is the preferred form. In the previous version of the library we supported this form with macros, but it can be done equally well with good programming style, so the macros are unnecessary.

We can think of `worker` as a circuit that has been wired in. `tm-source` is the input over time, `tm-sink` is the output, and the quantifier is providing the clock signal. This is the high order programming model I developed for my reconfigurable data path streaming processor at Quicksilver Technologies. In that case, each configuration command corresponded to a different worker routine.

Transforms

A transform is a tape machine that, like the second order machines, has a base machine. Its job is to make that base machine behave a little differently.

Identity

The most boring transform is the `identity-tr`, which simply passes every operation to the base machine. This machine has a fully specified interface, which makes it useful as a fallback to other more specialized transforms that might not have a full interface.

Affine

The `affine-tr` transform is a specialization of the identity transform that makes the base machine tape appear to be circular. Affine implements routines that have a leftmost or rightmost continuation, or affect their left or right neighbors, and then hooks into those. So, for example, when the base machine is stepped from rightmost, the affine machine cues the base machine to leftmost and takes the ok continuation.

As of version 0.7 the old affine code has been deprecated, and we haven't updated it yet.

Ensemble

The ensemble transform makes a group of machines appear to be a single machine. The individuals to be grouped can be provided either as a list or as instances in the cells of another machine. In the example below we use the `:list` initializer. So, for example, stepping an `ensemble` steps all of the member machines. The prior version implemented transactional behavior, i.e. if any of the machines in the group would take rightmost, then none of them would step, and the continuation would be taken. In contrast, this version walks down the list of member machines, stepping each, and if any takes a rightmost continuation, then the ensemble takes the rightmost continuation. When the rightmost continuation is taken, the internal members machine will have its head on the cell holding the instance of the machine that took the rightmost continuation. Here is an example of using ensemble, this test, like all our tests, is intended to return `t`.

♦
```
(defun test-ensemble-0 ()
 (let*(
    (tm0 (mk 'list-tm {:tape {1 2 3 31}}))
    (tm1 (mk 'list-tm {:tape {4 5 6}}))
    (tm2 (mk 'list-tm {:tape {7 8 9 91}}))
    (tm10 (mk 'ensemble {:list {tm0 tm1 tm2}}))
    )
  (∧
   (= (r tm0) 1)
   (= (r tm1) 4)
   (= (r tm2) 7)
   (s tm10)
   (= (r tm0) 2)
   (= (r tm1) 5)
   (= (r tm2) 8)
   (s tm10)
   (= (r tm0) 3)
   (= (r tm1) 6)
   (= (r tm2) 9)
   (¬ (s tm10))
   (eq (r (members tm10)) tm1)
   (on-rightmost (r (members tm10)))
   (c◧ (members tm10))
   (on-rightmost (r (members tm10)))
   (= (r (r (members tm10))) 31) ;this machine stepped
   (s (members tm10))
   (on-rightmost (r (members tm10)))
   (= (r (r (members tm10))) 6) ;this machine failed to step
   (s (members tm10))
   (¬ (on-rightmost (r (members tm10))))
   (= (r (r (members tm10))) 9) ;haven't stepped this one yet
   (¬ (s (members tm10)))
   (c◧∃ (members tm10)
    (λ(tm ct c∅)
     (if
      (¬ (on-rightmost (r tm)))
      [ct]
      [c∅]
      )))
   )))
```

TM access language

The TM Library consists of many routines with single letter names, or a single letter and a symbol. This is so they can be easily composed to make longer commands. It is already the case that when we had a routine that performed the same thing that could be done from multiple more primitive routines, we gave it a compound name, for example, `esr`. However, routines like `esr` potentially have their own implementation, the compounding is simply used to provide a descriptive name. Here with the access language stringing the letters together is writing a little program that instructs Lisp to run each command one after the other.

♦
```
(δ sn-snr tm 3 4)
(Δ sn-snr tm 3 4)
```
`δ` is the routine (interpreted) form, and `Δ` is the macro (compiled) form. We read from left to right, and parameters are removed from the parameters list at the right in order as needed. Continuation options are added as appropriate. No continuations are shown in the example just above. This access program says to step the machine right 3 places, and then to step it left four places, and then to read the instance. This program is not identical to just stepping left by one, because right stepping might hit a right bound.

As of version 0.7 we have not implemented the TM Access Language interpreter or compiler.

TM Library summary

Types

♦

```
list-tm
list-solo-tm
list-nd-tm
list-haz-tm

bilist-tm
bilist-solo-tm
bilist-nd-tm
bilist-haz-tm

recursive

identity-tr
ensemble

status-tm
ea-tm
ts1-tm
```

Interface

tm

♦

```
(defun mk (tm-class &optional init-parms ➜)
(def-routine-class ec◧r (tm &optional ➜))
(def-routine-class ec◧sr (tm &optional ➜))
(def-routine-class ec◧sw (tm instance &optional ➜))
(def-routine-class ec◧w (tm instance &optional ➜))
(def-routine-class esr (tm &optional ➜))
(def-routine-class esw (tm instance &optional ➜))
(def-routine-class init (instance &optional init-value ➜))
(def-routine-class r (tm &optional ➜))
(def-routine-class w (tm instance &optional ➜))
```

tm-generic

♦
```
(def-routine-class a&h◨ (tm instance &optional ➜)
(def-routine-class as (tm instance &optional ➜)
(def-routine-class as&h◨ (tm instance &optional ➜)
(def-routine-class c◨ (tm &optional ➜)
```

solo-tm

♦
```
(def-routine-class a◧ (tm instance &optional ➜)
(def-routine-class d (tm &optional spill ➜)
(def-routine-class d◧ (tm &optional spill ➜)
(def-routine-class update-tape-after-a◧ (tm tm-ref))
(def-routine-class update-tape-after-d◧ (tm tm-ref))
```

solo-tm-generic

currently there are none.

nd-tm

♦
```
(def-routine-class a◨ (tm instance &optional ➜)
(def-routine-class entangle (tm-orig &optional ➜))
(def-routine-class entangled (tm0 tm1 &optional ➜))
(def-routine-class heads-on-same-cell (tm0 tm1 &optional ➜))
(def-routine-class s≠ (tm0 tm1 &optional ➜)
(def-routine-class with-entangled (tm continuation))
```

nd-tm-generic

♦
```
(def-routine-class tm-print (tm))
```

haz

♦

```
(def-routine-class d. (tm &optional spill ➜)
```

bi-tm

♦

```
(def-routine-class -s (tm &optional ➜)
(def-routine-class -a (tm instance &optional ➜)
(def-routine-class -d (tm &optional spill ➜)
```

quantifiers

♦

```
(defun always-false (tm ➜t ➜∅)
(defun always-true (tm ➜t ➜∅)
(defun c◧∀ (tm pred &optional (➜t (be t)) (➜∅ (be ∅)))
(defun c◧∀* (tm function)
(defun c◧∃ (tm pred &optional (➜t (be t)) (➜∅ (be ∅)))
(defun c◧∃* (tm pred)
(defun ∀ (tm pred &optional (➜t (be t)) (➜∅ (be ∅)))
(defun ∀* (tm function)
(defun ∃ (tm pred &optional (➜t (be t)) (➜∅ (be ∅)))
(defun ∃* (tm pred)
(defun ⟳ (work)
```

quantified

♦

```
(def-routine-class -s* (tm)
(def-routine-class a* (tm fill &optional ➜)
(def-routine-class as* (tm fill &optional ➜)
(def-routine-class asn (tm n &optional fill ➜)
(def-routine-class s* (tm)
(def-routine-class sn (tm n &optional ➜)
(def-routine-class w* (tm fill &optional ➜)
(defun-typed -s*((tm tape-machine))(c◧ tm))
(defun-typed a*
(defun-typed as*
(defun-typed asn
(defun-typed s* ((tm tape-machine)) (c◨ tm))
(defun-typed sn
(defun-typed w*
```

recursive

♦
```
(defun increment-to (b &optional (stride 1))
(defun decrement-to (a &optional (stride 1))
(defun mk-interval (a b &optional (stride 1))
(defun mk-Natural ()
```

status

♦
```
(def-routine-class abandon (tm))
(def-routine-class park (tm &optional ➜))
```

Review of the C memory model

This chapter is a review of the C memory model and the nomenclature used for talking about it. The original, and still classic, reference on the C language is “C Programing Language”, by Brian W. Kernighan and Dennis Ritchie; 2nd edition ISBN 978-0131103627.

The TTCA memory model differs from the C memory model in some subtle, and some not so subtle ways. If we are to discuss these differences we will first need to have a firm grasp of the C memory model.

Persistence

The adjective *static* applies to things resolved by the compiler, linker, or perhaps loader; or by an interpreter without taking into account runtime state.

The adjective *dynamic* means something is resolved by executing the program, or at least will potentially be resolved that way.

In the C language lexical scope is a static concept. We can see it when we read our program source, but it will be gone by runtime. At runtime we will instead have dynamic scope for such things as stack frames and variables.

The term *initialization* is a synonym for the first write after data has been allocated. When data is exposed after allocation and before the first write, there is a hazard that uninitialized will be read, i.e. a *read before initialization* hazard. Generally code is malformed if it reads data before initializing it. The term ‘static initialization’ means that data was given a value at compile time.

The C language provides syntax for the initialization of data, though what can be done with these C ‘initializers’ is quite limited. When an initializer is applied at compile time to statically allocated data, we say that the data has been statically initialized.

If data is never written again after being initialized, we say that the data is ‘constant’. Using statically initialized constants (i.e. constants created at compile time) often provides the compiler with an opportunity for optimizing code.

The offsets for the fields of a `struct` are statically initialized constants, and may be accessed through the `offsetof` macro. The compiler may place these values directly into the program text, i.e. into the instructions that constitute the program.

If we have a ‘mutability’ service, we might be able to set data to be read-only for some duration of time, and not read-only during other times. It is fairly common for people to refer to read-only data as being constant; however, when mutability can be changed, the compiler typically cannot take advantage of read-only status, so it is not really useful to say that read-only data of this type is ‘constant’.

The state machine controller of a Turing Machine is static and constant. This is noteworthy because our program is the controller definition for our processor. Hence, we would not expect that requiring a program to be a statically initialized constant would limit its computational power.

Allocation and data

A memory allocation is a block of memory set aside by a memory manager such as `malloc`, which may be safely be used by the program. The memory manager maintains a contract with its users that it will never allocate the same memory to any two such active allocations. A memory allocation becomes inactive after being a call to deallocate it, which in C is done with the routine `free`.

An allocated block of memory is analogous to a physical memory device. Like for physical memory, an allocation has a fixed size, and data may be written to it and then later read back. On some computing systems physical memory may be plugged in and removed, just as a memory manager may be called to allocate and deallocate memory.

Data refers to the values of the bits found on a memory device, and by extension, those found within a memory allocation.

Data may be moved using a *copy* operation. A copy operation sets the data at some destination allocation so that it matches the data at a source allocation. The data at the source is not affected, while the data formerly at the destination will be clobbered. Because the source data remains in tact, it is possible to copy source data to multiple destinations. An exception to this is the case where the source and destination are the same. The behavior in such a case is generally undefined, but might be defined for specific systems.

Read and *write* are not distinct operations from *copy*, but rather these terms describe what happens on one end or the other of a copy operation.

The most common types of allocation are: static, program stack base, program stack, and heap.

All allocators have two main operations on their interface, *allocate* and *deallocate*. We say that an allocation is *alive* after it has been allocated and before it has been deallocated.

The allocation and deallocation operators are commonly shortened to *alloc*, and *dealloc*. It is also common that allocators of different types use different names for the allocation and deallocation operators.

Register

Most all processors have a small internal memory called a register file. Each register is typically given a name, and in the compiled code this name reduces to an integer, i.e. to an address. The general use registers are typically allocated and deallocated by the compiler. Some special purpose registers, such as a stack pointer, will typically not participate in the allocation scheme. This happens invisibly to the programmer.

However, even the assignments made by the compiler might not be where data is actually put. It is common on modern processors to attempt to reconstruct a more parallel execution friendly data flow graph on the fly. False dependencies are removed by using what amounts to a larger register file and then renaming registers, i.e. mapping them different addresses in the larger register file. If the processor is friendly, it will retire results in the same order as they would have been retired in the original program without the renaming, so that visible state remains the same.

Hence, programmers really do not have much control over register allocation.

Static

The compiler will create a symbol table that associates identifiers with the locations of data in memory. Each location will be either an absolute address or a relative offset, but in either case it will be a constant. In the C language paradigm the goal of the compiler, linker, and loader, is to embed these constant location values into the instructions of the program. Unless we have told the compiler to keep symbols as debugging information, they will be gone at runtime.

Data that has an absolute address determined at compile time is said to have been statically allocated. Such data is only deallocated when the process terminates and releases its virtual memory space. For statically allocated data all threads will see the same data located in the same place.

Program stack

The program stack is an instance of a stack data structure that is automatically managed though code inserted by the compiler and actions taken by the operating system. Each thread of execution is given its own program stack, which is a copy of the stack that the thread was forked from when it was created.

Conventionally this stack copy is done in an efficient manner by using the virtual memory system and copy on write pages.

When the program stack is used for memory allocation the allocation operator is called *push* and the deallocation operator is *pop*, occasionally someone will instead call these operations *push* and *pull*. For the stack used by C, data is pushed in blocks known as a stack frames. Each stack frame will hold the arguments and local variables to be used by the routine plus some overhead information such as the return address to continue from after the call.

Static data and program stack base data live for the duration of the program. In contrast, stack allocated data comes to life at runtime when a routine is called and the corresponding stack frame is pushed, and its life ends when the routine returns and the corresponding stack frame is popped. (This concept of life times becomes important when we consider memory usage, including register usage by programs when compiling. We find that there are both static and dynamic concepts of allocation life times.)

In C a lexical scope starts with an opening brace, and ends with a matching closing brace. At least in some compilers, although scope may be nested within a routine, all declared local variables will be placed in the stack frame of the containing routine. Thus, stack frames are specific to routines, and not specific to lexical scope. The enforcement of the nesting of lexical scope within the routine is then due to the compiler itself refusing to allow a programmer to make use of out of scope variables. Thus there is a difference in C between static scoping and dynamic scoping of data. This approach is one method for assuring the clean deallocation of local variables even when such things as a `goto` can cause a program to leave a scope level. It also causes allocation to be done only once even though a braced lexical scope with variable declarations occurs inside of a loop.

The compiled code will address arguments and local variables at constant offsets relative to the base of the routine's stack frame. Routines use stack relative addressing to find their variables, hence the same code will be usable without modification against different stack frame instances. I.e. arguments and local variables are accessed indirectly through the stack frame pointer.

However, when we use the ampersand operator, '&', on a local variable we will get its absolute address. No matter how far down in the call nesting we go, this absolute address will still locate the same value. Placing such an address in a generally accessible data structure creates the hazard that the address might be locating data that has been deallocated (due to the relevant stack frame having been popped), or even that it references allocated data from a new allocation (a new unrelated stack frame has been pushed).

Most programs are written by composing a large number of small routines. Routines need to have their arguments in the same place relative to the stack frame each time they are called. Consequently a great deal of effort will be spent moving data into stack frames before routine calls. This problem inspired AMD to provide a 'fast call' mode where arguments are passed through registers. These moves must be correctly timed before routine calls. Generally all this causes the stack to be a bottleneck for both optimization and hardware performance enhancements.

Program stack base

With 'program stack base' allocation, our data is placed in the first stack frame. This is the one that will be allocated to the \`main()\` routine in C. The first stack frame is special because it will never be popped off the stack while the program is running. This fact causes program stack base data to resemble static allocation because that also remains for the duration of program execution. However, there is an important difference between these two methods of allocation in that stack base allocated data is copied when a new thread is forked, where as static allocated data is not.

The astute reader will note that 'program stack base' and 'program stack' allocation are a first-rest pattern applied to a single program stack.

Heap

With ‘heap’ allocation we have a set of library routines for allocating blocks of memory from a free list, then later deallocating them by putting them back on the free list. In C it is conventional to use the `stdlib` routines which define only one heap to be shared by all routines. Heap overhead such as the free list base pointer will be statically allocated. The one `stdlib` heap will be shared by all threads. Modern versions of the heap have mutex locks already built in so as to assure thread safety.

Like static data, heap data is not copies when a new thread is forked. Heap allocation in C is accomplished by calling `malloc`, which then returns a pointer to the allocation. Programmers must be careful and remember that when copying such pointers, the heap data is not copied. Heap data can only be accessed from one place at a time within a single thread. However, in a multithreaded environment, if a heap pointer copy is given to multiple threads, it is conceivable that more than one threads will access the same heap memory simultaneously. This gets back to our discussion on shared tapes between multiple machines.

Garbage collection (not part of the C memory model)

Garbage collection is not part of the C memory model. Nor is it part of the TTCA memory model. However the technique has become so common that a treatise on memory allocation would be remiss for not at least mentioning it.

With garbage collection, allocation of memory is done explicitly as for heap allocation. However, unlike for the heap management of memory, with garbage collection the deallocation of data occurs implicitly in the background. One method of accomplishing this is for the overhead of a heap block to include a reference counter. When no references are left pointing to the allocation, the allocation is then put on a 'ready to be cleaned up' list. Periodically the garbage collector routine runs, typically on another thread in the background, and goes through the 'ready to be cleaned up' list and deallocates all blocks listed.

The advantage of this approach over the heap allocation approach is that there cannot be bugs of the 'woops forgot to deallocate it' nature. Such 'memory leaks' are all too common in code that makes use of heap memory management.

The C language and its standard libraries have no special support for garbage collection. Making use of a custom library for this would be tedious because the allocator would have to be told each time a pointer went out of scope, for example upon reaching a closing brace after being temporarily allocated. The calls would occur in the same places that, if we were using a heap, the deallocation calls would occur. Hence garbage collection with C would be no less error prone than heap allocation.

The proposal to modify the C compiler to by adding calls upon hitting the close brace of a pointers scope is not sufficient because copies of pointers can be made, and in C, they can be made in obviousness ways. If a person works a little harder it is possible to limit syntax, find copies, and to track scopes so that it all works. However; after do all of that, the language is no longer C.

I mentioned earlier that conventional processors are typically C machines, and you would be correct to surmise that they do not have hardware support for garbage collection. In contrast, most processors have hardware support for manipulating the program stack.

Storage Allocation Class Keywords in C

The C programming language specification speaks of ‘storage allocation classes’ rather than allocation types. The language has the storage allocation class keywords: `register`, `static`, and `auto`. In the early version of C these keywords corresponded to their namesake allocation types, with `auto` being program stack allocation. However over time `register` has become a hint, `static` has been overloaded with new meaning, and `auto` has become meaningless as in contexts where it may be used it is already the default.

Specifying the `register` storage allocation class is a request to the compiler to place the given variable in a hardware register. It is rare to see the `register` storage class keyword used because it is almost always better to let the compiler decide what should go into a register, and besides that, the compiler is free to ignore the request. Perhaps this becomes important for special purpose compilers for embedded systems.

Historically specifying a variable allocation with the `static` keyword would cause the variable to be allocated statically in memory. This was and still is the default for variables declared at global scope even without the static keyword. The names of such variables will be listed in the object code, so they can be the targets of variables declared with the `extern` keyword. In old versions of the C language this used to be the end of the story. However today the `static` keyword found on variables at global scope now means that said variable is not visible outside the source code file it is found in. It is opposite of the old meaning as trying to link to it from another object file compiled with an `extern` declaration will not work.

Routines in standard C are always declared at global scope. Declaring a routine is identical to declaring a routine pointer, and a routine pointer is data. Hence the static keyword works the same on routines as it does on other data.

If an inline routine is declared `static`, and the compiler decides to ignore the request to inline the routine, then the inline routine instead becomes a regular `static` routine. Such routines are declared in header files, so to say it is `static` is to mean it is a regular routine linkable within the scope of the file the include occurs in.

In C `auto` class variables have no external linkage. This typically means they are allocated on the program stack. This is the default for variables declared inside routines, including inside the `main()` routine. It is rare to see the `auto` storage class keyword used because this is already the default behavior inside routines, and at global scope it is illegal to use it.

Values and References

In C a *pointer* is an address bound to the type for the data being addressed. There is no address type in C, nor any `pointer` keyword. Instead when declaring a pointer C has us make a type declaration with an asterisk proceeding the identifier.[19]

♦

```
int i;
int k;
int *j = &k;
```

On line 1 of this example data is allocated with the size of an `int`. C leaves the size of an `int` to be determined by the compiler. Had we wanted to specify the size for the `int` we could have instead included the appropriate header and used a type such as `int32_t` instead of `int`.

If these declarations are placed at global scope the data will be allocated statically. If they are in the `main` routine this data will be in the base frame of the stack, and if they are made inside another routine, they will be in that routine's stack frame.

The allocation on line 1 will be associated with the symbol `i` which will be viewable within the lexical scope of the declaration, whatever that happens to be. It is not shown here. On the second line we again allocate an `int`.

Things get a bit more interesting on line 3 where we have another allocation, though this time we are allocating an *address* rather than an `int`. On my machine `int`s are 4 bytes and addresses are 8, so this line would create a larger allocation than each of the prior two. The compiler will take note that at this address in memory we find `int` type data.

[19] The actual case is more complex. Because C allows the programmer to force casting between types, programmers can pretend that a pointer is an address by casting it before each use. Historically when C programmers wanted to play this game they would use a pointer to `char`. More recently C added the void keyword. A pointer is an address bound to a type, so it follows that a pointer to a `void` type is the same as an address. The compiler then *requires* that pointers to void type be caste to a non-`void` type before being used.

Line 3 also has initialization code. We initialize `j` to be the address of `k`. `k` is indeed an integer, so the type contract check will pass without an error and the compiler will generate the code to perform this initialization.

However, all of this is a bit misleading, because at runtime when the processor needs to load data from memory, it must know the address of that data. Take for example, in a load/store architecture a processor loads data by executing a load where the operand of the instruction is the address of the data we desire to load. Hence, all data in memory will be accessed through addresses, even when no pointer has been declared. When we allocate an address we are often just making the address visible to our program rather than creating something new. Once the address is visible it may be passed as a parameter to a routine etc.

After data is loaded it will be in a processor register. If the data is part of the observable state of the program, and it is modified by the computation, it must be written back to memory. A store instruction performs the inverse operation to the load instruction.

It is possible that a variable will not appear in memory at all and thus not have an address. As an example, this can happen when the compiler decides that the variable may be allocated in a processor register. As another example, it can happen when the optimizer eliminates `i` altogether. Thus if we take the address of `i` in our source code, we might be doing something that was going to be done anyway, or we might be forcing the compiler to not use a register for `i`, or even forcing it to not eliminate `i`.

Given all of this, we can see that the C model for data and pointers is only an abstraction of what happens at runtime. It is not necessarily inefficient to read a value by dereferencing a pointer rather than using it directly, as the processor might have to use an address for the load instruction either way. We can examine the assembly output of the compiler to get a better idea of how our variables were really handled, but even that might not be the full picture, because modern processors will prefetch values, make use of caches, and rename registers so as to increase performance.

Introducing the Wave language

My objective in this volume is to describe a full path from theory to hardware definition. My intention is that this hardware is high performance. I'm not planning to implement something like a Lisp machine. Rather we are going to work towards compiled code, with an efficient assembly language that resembles what we see on contemporary processors of our computing epoch. Hence, our next step will be to convert our Tape Machine Unit code from Lisp to C.

In the prior chapter we used the Common Lisp language because of the language's ability to embody formal descriptions. We then used that model to explore properties of software. This chapter has a fundamentally different objective.

In this chapter we lay the groundwork for a new language that I am calling Wave. This will bring us closer to the programming model for the TTCA as it also mirrors what the TTCA assembly language will look like.

As of the time of this writing conventional processors are pretty much C language machines. Consequently, a good place to start when implementing our emulator is to develop a set of macros and techniques for expressing TTCA native coding styles in C. The C preprocessor macros and techniques used here will also morph into a library used by any future emulators, interpreters, or compilers.

The difficulties we stumble across while developing these macros will serve to highlight the differences between conventional processor architectures and TTCA. Similarly the manner of speaking we develop for talking about the emulator code will help us better discuss the TTCA. And who knows, perhaps this code will run especially well on conventional processors also. It does look to provide some opportunities for deep optimization by compilers.

Ending our single program stack addiction

Using a single program stack full of routine specific frames is a horribly inefficient way to encapsulate runtime state. Take for example coding the state of a loop in C versus that of a recursive call in LISP. State in C programming style is typically just a loop control variable, while in conventional Lisp programming style there will be a stack frame pushed corresponding to each iteration, and these stack frames will typically include routine call overhead, local variables and other things that were needed to lead up to the recursive call, but not be needed after it. Yet, we push all that stuff, repeatedly. What is a single integer in C becomes many kilobytes of data in Lisp.

Furthermore the existence of a stack causes optimization bottle necks, even for C programs. This affects hardware acceleration, compiler optimization, and execution models with multiple threads. The conventional compiler cannot see beyond the current compilation unit, i.e. source file, so the stack simply must be supported. What is required context on the stack, versus what the stack actually becomes, is irrelevant because the stack cannot be thrown away. Also a new thread gets a whole copy of the stack whether it needs it or not.

The inline routine feature of C has helped with compiler optimization. Inline routines are essentially text macros, and are included from header files, thus they are available to the compiler to analyze. However, this also leads to code duplication.

Prior chapters provided examples of multiple return path computation. Those examples were given in the Lisp language, where continuations were passed in as arguments. When a routine is executed it completes by taking one of the continuations.

Although the implication of those Lisp examples was that we had efficient forward moving computation, the reality was that each of those continuation routine calls pushed another stack frame, and we would eventually have to pop them all off. This situation was acceptable for out Lisp library because the point of developing that library was to study the method and explore its feasibility. It was a nice coincidence that the library was also useful for bringing iteration to Lisp.

In this chapter we will untie the knot of dependency on a single program stack by identifying alternative techniques for carrying program context forward. We will endeavor to make our program into the fixed controller, and to handle any growing data by using appropriate containers specifically for that data. We surmise this is possible, because a Turing Machine has a fixed controller, and all growth of data occurs on the tape.

Before building a dedicated compiler for Wave we will see what sort of assembly code such a compiler must output, and we will do this by coding backend code in C. Stack frame push and pop on routine calls and returns are intrinsic to the C language, so if we do not want to muck with the compiler itself we will have to call our continued routines using `goto` statements, so that is what we will do.

We will give our new unit of execution a descriptive name so as to break the association of our routines with the mental image we all have of a model stack push of input arguments and a return pop. We will call our routines *instruction sequences*. In the context of this language this term replaces the more usual terms *function*, *routine*, *subroutine*, or *procedure*. We will have two types of instruction sequences, *specific use* and *general use.*

Specific use instruction sequences make use of data within lexical scope, which in C includes the `goto` target labels. This type of instruction sequence is natural for a programmer to write when sitting in front of a terminal while coding something specifically for his or her own use.

General use instruction sequences are reusable. They are the sort we would find in a library. They cannot make use of the context they are called from because that context varies depending where the general use instruction sequence is called from. For an instruction sequence to be of general use we must define an interface which consists of a list of *parameters*. At run time these parameters will be assigned values called *arguments*, and they will carry the required context forward.

The code for this chapter may be found at:

http://www.github.com/Thomas-Walker-Lynch/TM2x

In the code examples, global variables are all caps with between word underscores. Type names and name space names are initial character capitalized in each word with words run together. Other identifiers are lower case with words separated by underscores. C does not have built in namespace syntax, so we will use a center dot character to separate namespace names from the more specific part of identifiers.

Specific use instruction sequences

An instruction sequence starts with an `SQ·def` macro call. It takes the sequence name as an argument. It ends with a matching `SQ·end` where the name of the sequence must again be provided as an argument. We call an instruction sequence by using the `SQ·continue(name)` macro. Hence, an instruction sequence will have this form:

♦

```
SQ·def(name){
 <place instructions here>
} SQ·end(name);
```

The `def` macros gives the instruction sequence a `goto` label. An instruction sequence must be reached using a `continue` call. We should never sequentially flow into it. I considered calling `abort()` in the `def` macro just above the label, but instead elected to add a `goto` that jumps to the `end` macro. Hence a sequence may be defined anywhere and it will effectively be ignored during sequential execution.

Instruction sequences do not return. All instruction sequences must execute another `SQ·continue` statement or terminate the thread before reaching the corresponding `SQ·end`.

The specific use instruction sequences defined in this manner have no explicit arguments passed in, but they may use any variable that is visible in the outer lexical scope.

Each instruction sequence name must be declared as type `SQ·Sequence`, twice, once just as `name`, and once as `SQ·name`. This is of course a hack. Due to compiler limitations, instruction sequences must be declared within a routine. We will put ours inside of `main`.

Implementation notes

Our instruction sequences are similar in some aspects to gcc's nested routines; however they do not make use of gcc nested routines. Our macros do make use of gcc extensions related to labels, so gcc must be used with this code.

Due to some C language restrictions on `goto` and labels, our instruction sequences may not be defined at global scope. This is probably due to C trying to make `goto` play well with stack based routine calls, and the compiler wanting to see both the `goto` and the label in a single compilation unit. Hence for now our sequences must be encapsulated in a routine. We will put them in `main`. Hence everything is going to be surrounded by a big `main` routine definition. All this should make the optimizer very happy.

We may call C routines from within a sequence. Currently this is a practical necessity because we do have instruction sequence libraries for things such as i/o.

It is possible to pass instruction sequence names into routines by preceding the name with two ampersands, `&&name`. A routine given such arguments might then make use of a trampoline by returning one of the sequence pointers. Such a return value may be dereferenced with a *, and then continued from it. Actually I implemented the first version of these examples by doing this. It had the nice effect of that not everything had to go into `main`; however, I promised to show the reader code that actually does not use the stack during calls, rather than just *looking* like it doesn't, so I have removed this from the examples.

Example

The following is a simple example using two specific use instruction sequences. It is located in the examples directory of the git distribution, and is called `special_use_instruction_sequences.c.`

♦
```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include "misc.h"
#include "Sequence.h"

// prints Collatz Conjecture sequences
int main(int argc ,char **argv){
 SQ·Sequence reduce ,SQ·reduce ,expand ,SQ·expand;

 uint32_t i;
 if( argc != 2 || !sscanf(argv[1], "%" SCNu32, &i) ){
  fprintf(stderr ,"expected argument that fits in uint32_t");
  return 1;
 }

 SQ·continue(reduce);

 SQ·def(reduce){
  printf("%" PRIu32 "\n", i);
  if( i == 1 ) return 0;
  if( i & 0x1 ) SQ·continue(expand);
  i = i >> 1;
  SQ·continue(reduce);
 }SQ·end(reduce);

 SQ·def(expand){
  uint64_t result = (uint64_t) i * 3 + 1;
  if( result > UINT32_MAX ){
   printf("exploded!");
   return 2;
  }
  i = result;
  SQ·continue(reduce);
 }SQ·end(expand);

}
```

Here we have two instruction sequences, `reduce` and `expand`. The `reduce` sequence divides an even valued `i` by two, while the expand sequence multiples `i` by 3 and adds 1. Note on line 29 we declare a local variable called `result`. This variable will be allocated once in `main`'s stack frame, and only deallocated when `main` returns, i.e. when the program ends. Due to our use of program stack base allocation, had this code made use of multiple threads, each thread would have its own version of `result`. However, we do not have multiple threads in this example. Here is an invocation:

♦

```
> ./specific_use_sequences 12
12
6
3
10
5
16
8
4
2
1
```

Notice on line 17 main continues into **`reduce`**. Because **`reduce`** is an instruction sequence it will never return to line 17. We will only move forward through the instruction sequence from this point. Processing proceeds forward as a propagating wave.

`i` has the value 12, and on line 22 **`reduce`** discovers it is even, so it divides it by two using a right shift. Then on line 24 **`reduce`** continues into itself. Here we see something interesting. There is functionality no difference between a recursive call and a loop form. This is because nothing is placed on the stack in the recursive call, and we will never return from it. It is not just that these two forms are functionally the same and one can be transformed to the other, rather it is that they are actually identical.

On the second pass through **`reduce`**, **`i`** initially has the value six, so this will again be divided by two, and we will again re-enter **`reduce`**. On our third time through **`reduce`**, **`i`** will have the value 3, which is odd, so then we continue with **`expand`**. **`expand`** multiplies **`i`** by 3, adds 1, and then continues into **`reduce`** again. And thus we continue in a forward manner until at some point on line 22 we discover that **`i`** is one. At that point we finish the program by returning from **`main`**.

In this example, pumping data caused integers to get larger, but did not require that the data structures expand. Had we needed an expanding data structure we would have used the heap.

Here all the only variable shared among the instruction sequences was `i`, and we allocated it on the program stack base. The program stack base allocation is thread safe. Each thread gets its own stack. Collectively we call the variables used by the instruction sequences a *tableau*. It is a sort of chalk board where we can write things then read them off later. We should put all shared variables inside of a `struct` and given this collection a name, something like `tableau_0`. Doing this would then lead to a discussion of managing tableaux, e.g. whether a tableau should be shared among threads, etc. Actually, we will have such a discussion in the next section.

General use instruction sequences

Specific use instruction sequences such as those shown in the prior section do not accept arguments. Instead they have hard coded explicit use variables found within one or more tableaux. In our prior section we had one implicit tableau which was identical to the lexical scope.

Hard coding all of the input variable locations has a serious limitation. Say for example, if we were to write a specific use instruction sequence that prints the value of a variable. Perhaps one that variously prints the values of, say, `i`, `j`, and `k`. Because our specific use instruction sequence would have to have the variable name hard coded into it, we would need three separate specific use instruction sequences, one for each of `i`, `j`, and `k`. This kind of thing is workable, it is roughly what a C inline routine does; however in some situations code duplication leads to unnecessarily long programs.

A conventional routine call is usually some variation of the following. The processor first pushes the address of the instruction after the call on to the stack. A stack frame is allocated on the stack with space for arguments and local variables. Then the arguments are copied into it. The processor then jumps to the specified routine entry point. The stack pointer is of central importance because the routine itself uses it to find its arguments and temporary variable allocations. When the routine finishes it pops the stack frame and pushes its result. The caller then pops the result, then pops the continuation address that it had pushed earlier, and then jumps to it.

This conventional approach makes it so that each routine has its own memory context and its inputs are found in this context. However this comes at the price of having to copy input arguments in, being limited to one continuation after the routine completes, and having one return value. The limitation of only one continuation and one return value causes programmers to generate a lot of ad hoc code for dealing with situations where multiple continuations are needed. Sometimes a programmer will be pressed for time and chose to not implement some of the continuation paths that seem unlikely. This leads to bugs when these cases happen to turn out to be needed. Throw and catch are an out of paradigm approach for dealing with this problem, and because it is out of paradigm there are complications in dealing with the cross stack frame jumps. Typically programmers are told to use it only for unusual cases.

Latent linking

A C program is first compiled one compilation unit at a time. Such a compilation unit is typically a source file along with its included header files. This results in an object file. Each object file will have embedded in it a symbol table that lists the unresolved symbols and where their values belong in the code. For example, if a user calls an external routine, the symbol table will have an entry that contains the routine name and the location in the the object files where the routine address is to be written when it becomes known. The object code is binary, and we would expect such a place to be in the immediate field of a call instruction.

There might be many compilation units, and thus many object files. Hence the second step is to call a program called a linker which reads the embedded symbol tables in the object files and matches them up and writes in the missing addresses into the code.

Now suppose that for an instruction sequence we wait until the very last moment, just after the call to the instruction sequence, to do a final linking step. This combination of an instruction sequence with latent linking would make the instruction sequence generally usable.

Going back to our print example, we would have one instruction sequence that prints an integer. However, the linkage of the integer to the instruction sequence would occur at time of call, and could be different upon each call. When called for printing the variable i, the blank address for the print variable will be filled with the address of i. When called for j, the address blank will be filled with the address of j. etc.

The latent linking occurs so late that we no longer have symbols, but rather everything has been reduced to addresses. Hence the symbol table used for latent linking will be a list of address pairs, the first of the pair being a constant pointing at the allocation for the variable being linked in, and the second will be the location in the code where this constant is to be written. Each call has its own symbol table for performing the latent link step.

The following is what a symbol table for a latent link might look like.

♦
```
1. typedef struct{
2.  SQ·Ptr sequence;
3.  SQ·Pairs *args;
4.  SQ·Pairs *results;
5.  SQ·Pairs *continuations
6. } SQ·CallTable;
```
Line 2 `sequence` is a pointer to the general use instruction sequence. Line 3 is a list of pairs. The first element of each pair is the address on the tableau for an input argument. The second element of the pair is the location in the general purpose sequence where this address should be written. There are analogous pair lists for the results and continuations.

Actually we only need one list, because all the fields here have the same form. The form is that of two constant addresses, where the first address is to be written at the location indicated by the second.

When we say *list* here, we are not saying how that list is being implemented. In conventional C code a person might expect to see a null terminated array of pairs. If the list is of fixed length, we might use a `struct` to destructure all or part of it.

To make latent linking work on a conventional machine, we would need to copy a code template to a thread owned buffer so that we do not affect other thread's view of the code. Then we would perform the latent link by filling in the missing addresses for input arguments, results, and continuations. Then we would jump to the buffered code. Chances are we would have to change the permissions on the virtual page the buffered code is on before we would be allowed to execute the code.

My goal here is to make it so that we can write some realistic TTCA code examples just based on the macros; so copying a block of code as data, manipulating it, then changing a data pages to a text page while requiring special permissions, is more involved than I had hoped for. But never fear pointer indirection is here! In the next section we will show how to use pointer indirection so as to avoid modifying the instruction sequence code.

Interpreted symbol table

With the latent link method the symbol table is used once so as to fix up the code immediately before we jump to it. Accordingly only the code that does the call reads the symbol table. The instruction sequence itself has no reference to the symbol table.

In contrast, with the interpreted symbol table method the instruction sequence refers to the symbol table at runtime. Instead of having blanks that are filled in with addresses, the instruction sequence contains compiled in offsets into the symbol table, and these are dereferenced to find the necessary data.

The following is such an interpreted symbol table. We also call it a *link*. The link terminology comes about from a data flow view of our program, where the instruction sequences are nodes, and these symbol tables describe the connections between the nodes.

♦

```
1. typedef struct{
2.  SQ·Ptr sequence;
3.  SQ·Args *args;
4.  SQ·Ress *ress;
5.  SQ·Lnks *lnks;
6. } SQ·Lnk;
```

Here, line 3 `args` is a pointer to a block of pointers to the inputs, line 4 `ress` is a pointer to a block of results, and line 5 `lnks` is a pointer to a block of continuation links. Actually we could have just put all of these pointers into one big null terminated array. However, for purposes of illustration it is nice to see the structure and to refer to the fields in the structures instead of indexing into an array.

For purposes of illustration in this book I should have added another field that points to a tableau allocation for temporary variables. Here temporary variables are the same as result variables where these special results will not be used as inputs to other sequences.

Here our latent linker program is called `SQ·continue_indirect`. We pass in our interpreted symbol table. Equivalently, we can use a data flow perspective and say that we are following a link.

♦

```
SQ·continue_indirect(lnk)
```

An instruction sequence will need to reference the link to find its inputs, outputs, and continuations. Only one instruction sequence may run at a time in a given thread, so for these examples I have opted to keep a thread local variable that points to the currently active interpreted symbol table, i.e. to the currently active link.

♦

```
SQ·Lnk *SQ·lnk;
```

In the example github code this declaration is found in the file `Sequence·Text.h` and you will see this file included toward the top of the `main` routine so as to avoid any forward referencing problems. Note that the type name is capitalized, while the instance name is lower case.

The `continue_indirect` macro sets the active link pointer before calling the target instruction sequence. The address of the active pointer is compiled into each general use instruction sequence.

♦

```
#define SQ·continue_indirect(lnk) \
  SQ·lnk = (SQ·Lnk *)&(lnk);    \
  SQ·continue(*SQ·lnk->sequence);
```

If our processor has speculative execution, the reuse of a thread local variable will be a bottle neck that the processor runs into when it speculates past instruction sequence call boundaries. This looks a lot like the conventional bottleneck of running over modifications of the stack pointer; however there is an important difference. There is no stack behind the link pointer. It can be safely renamed just like any other variable held in a register.

There are many ways to make a link pointer available to general use sequences. Among these alternatives, each general use instruction sequence can be given its own allocation for a link pointer.

Example

The following code is for a general use special instruction sequence that does a kind of multiply that is useful when dealing with inclusive bounds. For example, it may be used to find the byte offset to the last element in an array, more specifically consider an array of int32_t type cells, with a maximum index of 7, 8 * 4 - 4 = 28 which can be computed without risk of overflow as 7 * 3 + 7 = 28. The inputs shown here are `a_0`, the extent of an array, and `a_1`, the extent of an element. The result is `r`.

♦

```
1.  SQ·def(Inclusive·mul_ext){
2.
3.   Inclusive·3opLL·Lnk *lnk = (Inclusive·3opLL·Lnk *)SQ·lnk;
4.   uint128_t t =
5.     *lnk->args->a_0 * *lnk->args->a_1 + *lnk->args->a_0;
6.   if( t > address_t_n ){
7.    SQ·continue_indirect(lnk->lnks->gt_address_t_n);
8.   }
9.   *lnk->ress->r = t;
10.   SQ·continue_indirect(lnk->lnks->nominal);
11.
12.  } SQ·end(Inclusive·mul_ei_bi);
```

On line 3 we cast the global `SQ·lnk` to the more specific type `Inclusive·3opLL·Lnk`. All of our example general use sequences start with such a cast. Here are the type definitions:

♦

```
typedef struct{
 SQ·Ptr sequence;
 Inclusive·3opLL·Args *args;
 Inclusive·3opLL·Ress *ress;
 Inclusive·3opLL·Lnks *lnks;
}Inclusive·3opLL·Lnk;

typedef struct {
 address_t *a_0;
 address_t *a_1;
} Inclusive·3opLL·Args;

typedef struct {
 address_t *r;
} Inclusive·3opLL·Ress;

typedef struct {
 SQ·Lnk nominal;
 SQ·Lnk gt_address_t_n;
} Inclusive·3opLL·Lnks;
// retypes SQ·Lnk
```

The `Args` and `Ress` structs will hold pointers into the tableau. The `Lnks` struct will hold links to continuation instruction sequences.

The `Inclusive·mul_ext` sequence then accesses its arguments and results indirectly through the `lnk`. Hence, every link may specify different places for arguments to be gathered from, and different places for results to be written to. Note that `Inclusive·mul_ext` has two continuations, one for when all goes well, and one in case the multiply overflows. We call the all goes well continuation the `nominal` continuation. We call the overflow continuation `gt_address_t_n` which stands for ‘greater than the extent of an address type’. Note that our multiply only writes a result if the `nominal` continuation is to be taken.

Here is an example using the general instruction sequence `Inclusive·mul_ext`:

♦
```
#include <stdio.h>
#include "misc.h"
#include "Sequence.h"
#include "Inclusive.h"
#include "Inclusive·DataTypes.h"

int main(){
 SQ·Sequence nominal
   ,SQ·nominal
   ,gt_address_t_n
   ,SQ·gt_address_t_n;
 #include "Sequence·Text.h"
 #include "Inclusive·Text.h"

 // result type Tableau
  address_t a_0 = 541;
  address_t a_1 = 727;
  address_t r;

 // make a link
  SQ·make_Lnk(mul ,Inclusive·3opLL ,&&Inclusive·mul_ext)
  mul_args.a_0 = &a_0;
  mul_args.a_1 = &a_1;
  mul_ress.r = &r;
  mul_lnks.nominal.sequence = &&nominal;
  mul_lnks.gt_address_t_n.sequence = &&gt_address_t_n;

 SQ·continue_indirect( mul_lnk );

 SQ·def(nominal){
  if( r == 394575 ){
   printf("glorious\n");
   return 0;
  }
  printf("wrong answer\n");
  return 1;
 }SQ·end(nominal);

 SQ·def(gt_address_t_n){
  printf("product unexpectedly overflowed\n");
  return 1;
 }SQ·end(gt_address_t_n);
```
The include file `Sequence·Text.h` only holds the global `SQ·lnk`. The include file `Inclusive·Text.h` holds the definition for `mul_ext`, as shown above. These are included in main so that their data will be stack base allocated.

The macro `SQ·make_Lnk` makes instances of the arguments, results, and links structures. After those are made we assign the values to their fields.

Tableau

Multiple instruction sequences sharing a tableaux is analogous to object oriented programming. In this analogy each instruction sequence corresponds to a method, and the tableau corresponds to the encapsulated data. Relative to our routine call discussion a tableaux was analogous to a stack frame. These two analogies both work because they are both examples of the more general concept of memory context.

A Tableau is a finite object. Programmers can create recursive structures, and if they do this, and it leads to expanding state, then that state should be captured in a heap managed data structure, such as a list, tree, or queue. The header to that data structure will be a finite object that can be placed on a tableau.

Result centric

All reads and writes are actually data copies. Similarly, each instruction sequence input value is another instruction sequence result value. Yet there is an asymmetry. Each instruction sequence reads each input from exactly *one* upstream result. In contrast, each result might need to be distributed to *multiple* downstream inputs.

Due to this asymmetry a good heuristic for organizing a tableau might be to keep results from each given instruction sequence together as a block on the tableau, and then use lists of pointers to gather inputs from all over a tableau. We call this a result centric tableau. A given instruction sequence then needs one pointer per input to find its inputs, and one pointer to find its result block. The total number of pointers needed is determined only by the properties of the given instruction sequence.

Our instructions sequences will need some initial inputs. These can be placed in a block in our result centric tableau as though they are results from some unseen instruction sequence. Alternatively, the input pointers might reach into other tableau to locate input data.

There can be a hazard with the result centric approach. This hazard occurs when a given instruction sequence is called a second time through the same link, and clobbers a prior result, but another, yet to be called, instruction sequence still needs that prior result. The simplest fix is to use a different link for the second call, this different link would point to the same sequence, and to a different result block. Another possible fix is to use a FIFO to hold results, rather than a single register.

Input centric

The dual of the result centric approach is to have a block of inputs and result pointers. Most people find this more intuitive. In the least it is convenient when sitting in a debugger to be able to see all the inputs together just before stepping into a sequence.

With an input centric tableau, each given instruction sequence needs one pointer to find its input block, and a list of pointers for each result, so as to know where to distribute the result. Here we see the aforementioned asymmetry as the lengths of the result pointer lists are each a routine of how many other instruction sequences will use the each result. In contrast to the result centric tableau approach, this information is not a property of the given instruction sequence, rather it is a property of the data flow graph the sequence occurs in.

We have an analogous hazard to that of the result centric approach. Suppose that a given instruction sequence just ran and its output is to be used by a second downstream instruction sequence, but not on its next invocation, but rather the one after that. Again among the potential fixes we could use a different link for the second invocation, or make one of the inputs to the downstream instruction stream a FIFO.

State variables and messages

We may allocate on a tableaux variables that are interpreted as the state of the program rather than the input or output of instruction sequences. Similarly we can keep messages. The FIFOs suggested above are examples of one possible messaging scheme.

Hardware emulation

Suppose we have a special situation where we have a collection of instruction sequences where no member makes a direct execution call to any other, but rather each instruction sequence runs when its inputs change. Now suppose also, that we have a rule that an instruction sequence should run if one of its inputs changes.

We modify the link to look something like this:

♦
```
1. typedef struct{
2.  SQ·Ptr sequence;
3.  SQ·Args *arg_froms;
4.  SQ·Args *arg_block;
5.  SQ·Ress *res_block;
6. } SQ·Lnk;
```

As usual we have a pointer to the instruction sequence being called at the top of the structure. As per the result centric approach, line 3 `arg_froms,` is a list of pointers telling us where to get the input arguments from. As per the input centric approach, Line 4 `arg_block,` is a pointer to a block on the tableaux for the inputs. Again like for the result centric approach, Line 5 `res_block,` is a pointer to a block of results. There is no list of continuations, as control flow is determined by inputs changing rather than direct invocation.

To run our program we repeat the following steps until no inputs change:

1. For each and every link, use the `args_from` to write the `arg_block` with input values. For each link, if any of the arg_block value is changed by the write, put a pointer to it on the execution queue.

2. If the execution queue is empty, then take the `finished` continuation. Otherwise, pull all the links off the instruction queue and run their

instruction sequences. This will populate each `res_block`; continue to step 1.

Notice that in step 2 that all of the instruction sequences are independent of each other. Hence if our tableau is shared between threads, then we can spawn many threads, up to the number of instruction sequences, so as to get the work done as soon as possible. Indeed, some of our instruction sequences can actually be hardware coprocessors, that run to process their input data.

We may organize any feed forward data flow graph in levels, with inputs going into level 0, and results provided from level n. We add identity routine instruction sequences into the levelized graph so that each instruction sequence on level *n* receives inputs only from instruction sequences on level *n-1*. We have an instruction queue for each level.

With the approach where instruction sequences are scheduled to run whenever an input changes, execution time can be wasted on incomplete input sets, where one input has arrived but another has not. This is a typical data flow issue. If we do not want the spurious firings we must somehow recognize matched inputs sets. The data flow solution is to add tags to data and then to match up the tags. In superscalars we rename registers and instructions call out registers as operands. With a levelizing the data flow graph data remains in sync as it passes from level to level.

The levelized approach also facilitates pipelined execution, by having different threads on each level. After the initial pipeline flush, all the level threads launch, then rendezvous. After they all rendezvous, the cycle is repeated. Within each level multiple threads may also be used to shoulder the work of clearing the execution queue. However, depending on the overhead of thread creation, for small programs it will higher performance to have a only level specific threads, or perhaps even only one thread. The one thread then runs each level in order sequentially starting from level 0.

This approach works for emulating hardware. For example, our instruction sequences could be used to emulating logic gates in a logic circuit. However, when this hardware emulation approach is applied to conventional programming we have difficulty with instruction sequences that embed conditionals that chose one continuation among multiple choices. According to the conventional execution model only the chosen continuation would run, but with the hardware emulation execution model, all instruction sequences behind links that see input changes will run.

Suppose that we require that a link be called in order to turn it on. In which case, monitoring input changes would only affect performance, as the control flow of the program would be determined by the calls. Performance might be negatively impacted due to all the input comparisons, threads and queuing overhead; or it might be positively impacted due to the use of multiple threads.

Execution queue

We can throw away the input change checks of the prior example, and keep instruction sequence calling and the execution queue. Accordingly an calling an instruction sequence puts a pointer to its link on the execution queue. Instruction sequences are then pulled from the queue for execution. The queue can be shared among real threads and green threads, thus facilitating execution management. If we cannot pre-empt instruction sequences, then we would need to assure that by design they do not run for too long.

Latent linking example

It will often be the case that the implementation of a general use sequences will make use of further general use sequences. For sake of discussion, suppose that we have five sequences, s0, s1, s2, s3, and s4. Further suppose that s0 is written by an application programmer. The application programmer makes use of the general use library sequence s1. The application programmer has given s1 two continuations, say for sake of discussion they are called **`nominal`** and **`fail`**. The application programmer probably does not really care how s1 is implemented, and in fact that implementation might change with new versions of the library.

However, s1 does have an implementation. Say for sake of example that s1 will declare its own result tableau to be used for sequences inside of it, and thus that it will wire up the links between the sequences much like our top level code does. Hence, s1 might wire together s2, s3, and s4 and then call s4.

At runtime s0 will call s1. s1 will then call, say, s2, which calls s3, then s4. However, the application programmer only sees s1, and he or she expects that s1 will take one of the provided continuations, either `nominal` or `fail`. But after s1 calls s2, control flow will never go back to s1. Suppose for example the wave of execution reaches s4 when it is time to call `nominal` or `fail`.

If s4 is a special use instruction sequence, then we can write s4 to use the continuation links originally passed to s1. If s4 is a general use instruction sequence, then it will have its own link, but that link will not know anything about `nominal` and `fail`. Because those were not available until s1 was called.

What we need to have happen is for s1 to run a latent linker to connect s4 to continuations to the `nominal` or `fail` continuations. Accordingly upon entry to s1, before it continues to s2, s1 copies continuations passed to it to where they belong in the circuit of s2, s3, and s4.

Hence, part of the job of a general instruction sequence that itself makes use of general instruction sequences is to finish building the data flow graph, i.e. to perform a latent link, before it continues into it first constituent sub-sequence. Thus we have a sort of fractal pattern where every invocation of a general use instruction sequence has some differential work to be done of the same sort that was done when wiring the outer instruction sequences together.

Here is an example from the TM2x tape implementation. This routine copies elements from one tape machine to another. It does this by scaling the passed in element extent to byte extents, and then calling `copy_bytes`.

♦

```
SQ·def(TM2x·copy_elements){
  TM2x·CopyElements·Lnk *lnk =
   (TM2x·CopyElements·Lnk *)SQ·lnk;

  // tableau
  address_t src_byte_0;
  address_t dst_byte_0;
  address_t ext_byte_n;

SQ·make_Lnk(scale_src,Inclusive·3opLL,&&Inclusive·mul_idx);
SQ·make_Lnk(scale_dst,Inclusive·3opLL,&&Inclusive·mul_idx);
SQ·make_Lnk(scale_ext,Inclusive·3opLL,&&Inclusive·mul_ext);
SQ·make_Lnk(copy_bytes ,TM2x·CopyBytes ,&&TM2x·copy_bytes);

  scale_src_lnks = (Inclusive·3opLL·Lnks)
   { .nominal = AS(scale_dst_lnk ,SQ·Lnk)
    ,.gt_address_t_n = lnk->lnks->src_index_gt_n
   };
  scale_dst_lnks = (Inclusive·3opLL·Lnks)
   { .nominal = AS(scale_ext_lnk ,SQ·Lnk)
    ,.gt_address_t_n = lnk->lnks->dst_index_gt_n
   };

…
```

♦
```
  scale_ext_lnks = (Inclusive·3opLL·Lnks)
   { .nominal = AS(copy_bytes_lnk ,SQ·Lnk)
    ,.gt_address_t_n = lnk->lnks->src_index_gt_n
   };
  copy_bytes_lnks = (TM2x·CopyBytes·Lnks)
   { .nominal = lnk->lnks->nominal
    ,.src_index_gt_n = lnk->lnks->src_index_gt_n
    ,.dst_index_gt_n = lnk->lnks->dst_index_gt_n
   };

  scale_src_ress.r = &src_byte_0;
  scale_dst_ress.r = &dst_byte_0;
  scale_ext_ress.r = &ext_byte_n;

  scale_src_args = (Inclusive·3opLL·Args)
   { .a_0 = lnk->args->src_element_0
    ,.a_1 = lnk->args->element_byte_n
   };
  scale_dst_args = (Inclusive·3opLL·Args)
   { .a_0 = lnk->args->dst_element_0
    ,.a_1 = lnk->args->element_byte_n
   };
  scale_ext_args = (Inclusive·3opLL·Args)
   { .a_0 = lnk->args->element_n
    ,.a_1 = lnk->args->element_byte_n
   };
  copy_bytes_args = (TM2x·CopyBytes·Args)
   { .src    = lnk->args->src
    ,.src_byte_0 = &src_byte_0
    ,.dst    = lnk->args->dst
    ,.dst_byte_0 = &dst_byte_0
    ,.byte_n   = &ext_byte_n
   };

 SQ·continue_indirect(scale_src_lnk);

} SQ·end(TM2x·copy_elements);
```

The only macro that has not yet been discussed thus far is `AS`. `AS` is a strong, no warnings given, cast. The above code uses the same format as we have been following up to this point. First it declares a result centric tableau. Then it declares the topology. After creating the links it wires up the connections between the sequences. Notice that during this wiring up the sequence step, it copies-down links from `SQ·lnk` (which has been cast to a more specific type and is now just called `lnk`) It then ties the arguments and results of the local sequences into the local result centric tableau.

Tableau optimizations

Allocations on the tableau have lifetimes. For sake of example, say the given lifetime of an allocation spans from t0 to t7 inclusively, and the lifetime of another allocation spans from t11 to t21. Then the second allocation can use the same memory as that used by the first allocation. A couple of examples follow.

Consider single threaded execution with a result centric tableau. Only one link can be active at any given time. Suppose the active link is l_0. Now examine the input pointers for all downstream links from l_0. These can be found by traversing the continuations. If none of these downstream links have inputs that point to an upstream link's result block, then the upstream result block can be deallocated. Given a data flow diagram we can perform this analysis on a per node basis at compile time.

As another example of optimization, consider the case of single threaded execution over a levelized data flow diagram. When we are at any given level, say level n, then we know that allocations associated with all prior levels can be released.

Streaming

One option for streaming is to use stack allocated tableaux and then to spawn a new thread for each new input data set. If we want to keep results in order, we will have to keep track of the order that the threads were spawned in, and to collect the output data in that order.

Another option for streaming was described earlier, and that is to create a a levelized data flow graph of our code, and then to run data through as pipeline.

Scheduling

Because continuations do return, they are convenient to schedule ..

Implementing a tape machine in C

Mathematical notation for discussing tapes

Programs reference a cell using an address. In turn a cell holds a value. In the section, Allocation and data, we noted that reads and writes occur in pairs, i.e. the actual operation performed when reading a memory is a *copy*. Read data must be held somewhere, and written data must come from somewhere. An exception to this is in the Turing Machine, where a value read from the tape is acted on by the state controller and is not copied anywhere. Analogously in a processor, the value in a register might be read by an ALU or affect branching instead of being written to another register. Generator routines can produce data that did not come from a memory, yet the data is written to a memory.

When reading from a cell with a given address there are two steps. First the address is used to find the cell. Secondly the value of the cell is copied somewhere or acted upon. The need to have the cell as separate entity from the contained value was discussed in the section, **Need for the concept of cell**. In brief, discussions about connectivity refer to cells, and discussions about computation refer to the values in those cells.

In this chapter we wish to introduce a mathematical notation for talking about tapes, addresses, cells, and the values held in cells. As usual with mathematical discussions we will need to name things. We will refer to tapes using the letter 't' followed by something that makes the name unique to a specific tape, often a number. Here are some example tape names: $t0, t1, t2$ ...

To give a name to a particular cell on a tape we will use the tape name as a function, where the argument to said function is the name of the cell, or an address. For example $t0(leftmost)$ refers to the leftmost cell on tape $t0$. An address might also be used, so $t0(0)$ also refers to the leftmost cell of t0.

In order to access a cell, we use the access operator, $*$ as a superscript. Hence, $t0(0)^*$ is the contents of the leftmost cell on tape $t0$. (This use of the asterisk is distinct from that of the Kleene star.)

The name of an address will start with an a. If it is not clear from the context which tape's address space the address occurs in, then the tape name is given as a subscript e.g., a_{t0} being an address from the address space of tape $t0$. When we are describing a function that returns an address, the function name will start with a. The address space for a tape is signified using a quantifier. Hence $\forall a_{t0}$ is the address space for tape $t0$.

Analogously, cell names start with the letter c. If we need to make clear which tape the cell belongs to we put the tape name as a subscript. The space of all cells on tape $t0$ is $\forall c_{t0}$. In order to name the address that corresponds to a given cell, we place the cell name in parenthesis after the letter a. Hence, $a(c)$ is the address of cell c. Going the other way, if we want the cell that corresponds to a given address, that is $c(a)$.

Thus the letters a and c are used both for variable names and function names. Typically it is clear when they are functions due to the parenthesis that follow. If this is not the case, then the distinction must be made clear by the context.

Some examples:

$$c\big(a(c)\big) = c$$

$$a\big(t0(0)\big) = 0$$

$$a\big(rightneighbor(c)\big) = a(c) + 1$$

An address space for a given tape can be specified with an interval, or as noted earlier, with a qualifier:

$$[a(leftmost), a(rightmost)]$$

$\forall a$

Area as a mapped tape abstraction

In the earlier section, Addresses and cells, we defined an area of a tape based on values in consecutive cells having a given property. An area could then be marked by a machine that traverses the tape cell by cell while checking each value for that property. In the case that there is no such value with said property on the tape, our marking machine would never halt. Also in the case that all cells to the right have the property, our marker machine would never halt.

We might avoid running the marker machine in the first place by analyzing the generator machine that created the tape data. We would look to see if this generator ever produces a value with said property, and having done that, ask the second question if there is a cell to the right that does not have the property. Without putting constraints on the property we are looking for and without knowing the initial values given to the generator tape, analysis might not be sufficient for determining these things.

If we know that the tape was produced by a computational machine, our data will be finite in length. We can require that the computational machine add an end marker. If the data is compact (no embedded empty cells), and the property we are looking for is not that of an empty cell, then the end marker could be the empty symbol.

A marked area will have a leftmost marked cell, consecutive right neighbor marked cells, and then optionally end at a rightmost marked cell. Hence, it fits the definition of a tape. An area is thus a sort of sub-tape.

We can constrain our property marking machine look at addresses and see certain addresses as the marked property. It could then mark all the cells in a given address interval. A machine that is constrained in this manner will always halt. We can then consider an area to be an interval of addresses.

As per convention each tape has an address space that starts with the leftmost cell at address zero. Thus we may express the correspondence between $t0$ and $t1$ cells as functions on the addresses:

$$a_{t0,\mathrm{t1}}(a_{t1}) = a_{t0,t1}(0) + a_{t1}$$

$$a^{-1}_{t0,\mathrm{t1}}(a_{t1}) = a_{t1,\mathrm{t0}}(a_{t0}) = \mathrm{a}_{\mathrm{t0}} - a_{t0,t1}(0)$$

Here $a_{t0,t1}$ is the address map going from $t1$to $t0$. The reverse map is $a_{t1,t0}$. For this to work we must be given the *base* address of the area, $a_{t0,t1}(0)$.

This generates the set of matched address pairs:

$$\forall \mathrm{a}_{\mathrm{t1}}: \{< a_{t0}(a_{t1}), a_{t1} >\}$$

♦

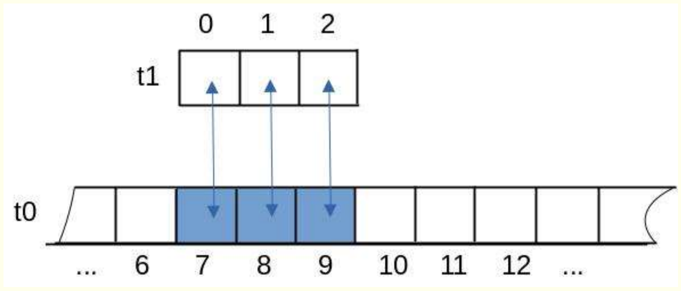


Figure 8 Area as a tape abstraction

This mapping cannot be derived through analysis even when including the definition of $t1$, because the information $a_{t0,t1}(0)$ cannot be found on either tape.

We have a special name for $a(rightmost)$ of a tape. We call it the tape's *extent* and we signify it with the symbol n. When we need to make clear which tape's extent we are signifying we add a subscript with the tape name, e.g. n_{t1}. The length of a tape will always be one greater than the extent, $length(t1) = n_{t1} + 1$. Because an extent is an address, it will always be representable in the same number representation that is being used for address. We cannot say the same for a tape's length. Programmers assuming that length fits in the same number representation as an address has been the source of bugs since the beginning of computing. Hence with our Wave language we only use the extent for bounding address space even if it is the address space for an abstract tape.

A *symbol table* is a real world example of an external structure used to hold the otherwise missing information about the mapping of an abstract tape to the concrete tape. Typically each line in such a table will define $t1$ by providing its extent, the value for $a_{t0,t1}(0)$, and perhaps a flag to signifying if $t1$ is empty. The $< a_{t0,t1}(0), n_{t1} >$ pair describes the address interval $[a_{t0,t1}(0), a_{t0,t1}(0) + n_{t1}]$.

The cells referenced by this interval of addresses form our more constrained definition for the *location* of an area. If we want to account for overlapping areas, we may place the address intervals into an interval tree. Real world symbol tables will also give each area a name. This name is used as a key for identifying the area.

If order must be maintained among multiple areas, where some areas might either have the same base address, or not have a base address at all, then we must add some accounting for the missing order information. One possible implementation is to add link columns to the symbol table. For each line in the table that must be ordered, links would be written into these fields. One link would reference the table line for the left neighbor area, and the other for the right neighbor area.

Area mappings and destructive operations

System memory cannot be destroyed, and for a given process this property is maintained to a great extent by the virtual memory system. Consequently we typically don't see the problems associated with destructive programming techniques until we implement data containers, or deal with other types of memory such as mass storage.

Containers that have many destructive operations typically identify members by using a *key* rather than an *address*. When using the approach of keys, when a program writes to a container, the container will return a key. Later when the program desires to read the previously written contained value, the program provides the key to the container. Unlike addresses, associative keys have no enforced relationship between each other. When given key k, which is associated with cell c, we cannot conclude that the right neighbor of c will have a key of $k + 1$, etc. In contrast, with addresses, provided that the right neighbor exists, we can conclude this.

When we delete a cell from a tape, the neighbor relationships for the surrounding cells change. Take for example three neighbor cells on a tape …ABC.... After deleting cell B, the tape will have …AC…. The cell A becomes the left neighbor of C, and C becomes the right neighbor of A. Now suppose that initially the addresses of cells A, B, and C were 7, 8 and 9 respectively. After deleting B are we to change our increment operator such that 7+1 results in 9? As another example, again consider a tape that is initially …ABC…, suppose we add a new cell, say cell D, resulting in a tape of …ABDC… Again the neighbor relationships have changed. The right neighbor of B was C, but now it is D, etc. What address should be bound to new cell D? Perhaps 8½? That is not an unsigned integer, so to do something like this would cause us to have to use a different data type for addresses, and still it would be unclear as to how to generate addresses in sequence.

If each address is to be uniquely bound to a corresponding cell for all of time, then address spaces will have holes where cells were deleted, and hiccups for where cells were added – as we saw in the examples given in the prior paragraph. If we are to support iteration we will be forced to add an accounting system for these holes and hiccups. After many various destructive operations on a tape the information about the topology of the tape will no longer be reflected in arithmetic on the integers; rather it will be found in the accounting system. In essence our addresses will turn into mere associative keys.

Consulting an accounting system while iterating will be less efficient than simply incrementing an integer.

Another approach is to say that each address does not remain bound to the same corresponding cell for all of time, but rather the invariant is that addresses are always step counts into the tape. After a cell is deleted, cells to the right of the deleted cell have their addresses decremented. After a cell is inserted, cells to the right have their addresses incremented, making an address available for the newly inserted cell. The difficulty with this approach is that addresses might be held as values in external variables, so the values in those variables might have to be adjusted. To make such an adjustment we will have to know where all such variables are located.

Yet another approach is to not use addresses, but instead to provide an interface for traversing the underlying tape. We see such an interface in Lisp where programs are written in terms of `first` and `rest` operations. However, because system memory is accessed through addresses, the connections between cells will likely be expressed by using embedded pointers, i.e. addresses. Indeed in Lisp each cons cell contains a value and a next cons cell pointer. Hence the approach of using an interface might hide the problem behind the interface, it does not solve it. It least not in the cases where the contain is implemented over conventional memory.

Among these three approaches, only the one that keeps addresses as step counts as the invariant is faithful to our original definition of addresses being step counts. Hence, this is the model we use in this book.

Arrays

In the prior section we discussed mapping the cells on an abstract tape $t1$ in a one-to-one to manner to cells on tape $t0$ and we embodied the map using address functions. In this section we will broaden this definition and allow that each cell on a tape $t2$ to potentially be mapped to many $t1$ cells, and as tape $t2$ has many cells, we will potentially end up with many $t1$ tapes, say $t10$, $t11$,... Where each $t1$ tape abstracts an area on a $t0$ tap.

For the discussion of this section we add some constraints. When a $t2$ cell is mapped to many $t1$ cells, those cells must start at leftmost and be contiguous. Also each of the potentially many $t1$ tapes must have the same length and map to contiguous areas of $t0$. In conventional terminology such a tape $t2$ is said to be an *array*, and the $t1$ tapes are said to be *elements* of the array. The compiler or interpreter maintains the fiction that these abstractions are concrete.

In the following figure shows the t2, multiple t1s, t0 hierarchy that defines an array. In prior tape figures I have shown the addresses next to the tapes so as not to confuse them for tape contents. Here I put the addresses directly in the boxes that represent the cells. I did this to keep the diagram compact. Of course, addresses are not cell contents.

♦

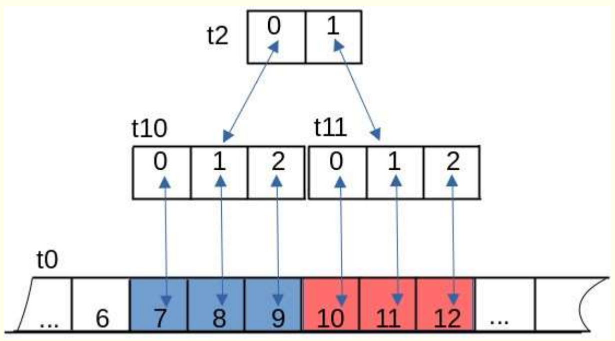


Figure 9 Array as a tape abstraction

Our address mapping functions for an array are:

$$a_{t0,t2}(a_{t2}) = a_{t0,t10}(0) + (n_{t1} + 1) \cdot a_{t2}$$

$$a^{-1}_{t0,t2}(a_{t2}) = a_{t2,t0}(a_{t0}) = \left(a_{t0} - \mathrm{a}_{t0,t10}(0)\right)\ div\ (n_{t1} + 1)$$

In conventional terminology for arrays address a_{t2} is said to be the *index*., and the value $(n_{t1} + 1) \cdot a_{t2}$ is called the *offset*. The value $a_{t0,t10}(0)$ Is the base of the array.

As an example, suppose that $t0$ is system memory and each cell is a byte, while $t2$ is an array where each element is a word. With 32 bit words the extent of each element, n_{t1}, will be 3. For 64 bit words it will be 7. Conventionally system memory is in bytes, but we consider each byte to be 8 bits. If we wanted we could create abstract tapes for the bytes, where addresses on those tapes go from 0 to 7.

We may remove the potentially overflowing 'plus one' by introducing an intrinsic multiply operator, $\triangle$:

$$a_{t0,\mathrm{t}2}(a_{t2}) = a_{t0,t10}(0) + (n_{t1} + 1) \cdot a_{t2}$$

$$a_{t0,\mathrm{t}2}(a_{t2}) = a_{t0,t10}(0) + n_{t1} \cdot a_{t1} + a_{t2}$$

$$a_{t0,\mathrm{t}2}(a_{t2}) = a_{t0,t10}(0) + n_{t1} \triangle a_{t2}$$

Let us continue the prior example where the elements of tape $t2$ are words. We know that the maximum address in each the $t1$ address spaces is n_{t1} . (This is the address of the rightmost cell for each $t1$ tape.) However suppose we wanted to instead know the address for the leftmost byte on $t0$ for the last element of the array $t2$. This address is the points to the rightmost element in the array on $t2$. This address will be,

$$a_{t0,t2}(n_{t2}) = a_{t0,t10}(0) + (n_{\mathrm{t}1} + 1) * n_{t2}$$

$$a_{t0,t2}(n_{t2}) = a_{t0,t10}(0) + n_{\mathrm{t}1} * n_{t2} + n_{t2}$$

$$a_{t0,t2}(n_{t2}) = a_{t0,t10}(0) + n_{\mathrm{t}1} \triangle n_{t2}$$

This result only makes sense because n_{t2} is an address, so we are just plugging it into the prior formula for translating addresses.

Now suppose we want the $t0$ address of the rightmost byte in the rightmost $t1$ word. For this we need to add in the extent of the word to the prior equation:

$$a_{t0,t2}(n_{t2}) + n_{t1} = a_{t0,t10}(0) + n_{\mathrm{t}1} * n_{t2} + n_{t2} + n_{t1}$$

$$a_{t0,t2}(n_{t2}) = a_{t0,t10}(0) + n_{\mathrm{t}1} \boxdot n_{t2}$$

We call $\boxdot$ an extrinsic multiply. It also will not overflow an address number representation. The value $n_{t1} \boxdot n_{t2}$ is the extent of the array on t_0.

We now have three concepts for speaking of the size of things. *Distance* is the difference between two addresses that belong to the same address space. The concept of distance is independent of any property of the cells being addressed. It is purely an address space concept. *Extent* is the largest address in the tape abstraction of an area. *Length* is a count of cells in an area, and will be one larger than the extent. It is a memory allocation concept. Yet another related term is that of *size*. Size is the number of cells used for holding an object. Data alignment issues might cause us to need a larger length in memory to hold an object than its size.

Clean array iteration

Suppose we desire to iterate in a manner that there is no danger that our expression of bounds will require a larger number representation than the one we are using for indexes we. There is a problem in the conventional approach because the bound is one greater than the largest index, which is the same as the length of the array.

♦

`0`		index of the leftmost cell
`n`		Greatest index. Extent of the array
`n+1`		length of the array

For sake of discussion, suppose that we have some sort of simple controller, and our fundamental type is a byte, even for addressing. Or similarly that we desire for our memory access to remain on a single page of virtual memory, where pages are 256 bytes, and thus the page offset is a single byte.

Now suppose we desire to iterate through this area of memory. An index into the array, i.e. an address to a cell on our tape, will fit in 8 bits, i..e. one byte. With such an index we can address any cell, including the one with the maximum index, cell index 255. Yet, it is conventional in computing to use the length of an array as an exclusive bound in loops. For example we might have something like:

♦
```
i=0;
n=255;
while( i < n+1 ){
 printf("i: %u\n" ,i);
 ...
 i++;
}
```
When using a one byte integer the bounding value `n+1` will overflow, and typically wrap back to zero. In this case the loop would not be entered.

If we expand the representation of the bounding value to `n+1` to two bytes, then we can represent 256, and thus we will enter the loop. This is wasteful as about half the bits are not used, but at least we can represent this outer exclusive bound.

Instead we could try to fix the problem by changing the loop test so that the bounding value would be `n`:

♦
```
i=0;
n=255;
while( i <= n ){
 printf("i: %u\n" ,i);
  ...
 i++;
}
```
However now there is a different problem. On what is to be the last pass through the loop we will increment `i` from 255 to 256, but `i` is one byte, and this will typically wrap `i` back to zero. Zero is less than 256 so, far from fixing the problem, we have made it worse. This loop will run forever.

The only reason that we have these end cases is because we are thinking in terms of exclusive bounds. Instead of saying "tell me if you cannot stay on the tape", we are saying "tell me if you go off the tape".

In addition to potentially causing program errors, we have also potentially create some hazards for enhanced performance hardware. When a processor tries to increase performance by watching a pointer and pre-fetching data when an address value changes, it might walk off the current page resulting in an unnecessary data fetch and possibly a page miss. If the next page does not exist we will have to distinguish between page faults caused by prefetches from real ones caused by program behavior. If we return some data anyway, we might have security issues.

If we instead use inclusive bounds we do not have these end cases. Because the bound is in the area that we are iterating over, it will be representable in the same data type as an index.

♦
```
i=0;
n=255;
loop:{
 printf("i: %u\n" ,i);
  ...
 if( i < 255 ){
  i++;
  goto loop;
 }
}
```
C does not have a built in inclusive bound loop structure, so here we construct it using a label and a goto. Note the entry into the loop is not guarded. This goes back to our discussion on orders of computation and emptiness. Here we are working with first order areas. The smallest array occurs when the leftmost cell index is zero, and the extent is also zero, i.e. when the area has one cell. We must add a second order structure to support the concept of the array being empty.

♦
```
if( empty ){
 printf("This empty array has no index values.\n");
    ...
}else{
 i=0;
 n=255;
 loop:{
  printf("i: %u\n" ,i);
  ...
  if( i < 255 ){
   i++;
   goto loop;
  }
 }
}
```
For the TTCA we will use extent values and inclusive bounding instead of length values and exclusive bounding.

Sometimes we desire to change the units the extent is measured in. Say for example we have an array with an extent of 31 as measured against 32 bit integers. Now we would like to know the extent of the array in bytes. The extent of a 32 bit integer is 3 bytes. We may use this information along with a special multiple instruction to find the answer:

mul_ext(3 , 31) = 3 * 31 + 3 + 31 = 127

Here mul_ext is intended to be a single instruction in our new instruction set. It computes `a0 * a1 + a0 + a1` as a single instruction.

Another useful multiply variation is used for converting an index in one unit of measure into an index in another unit of measure. It computes `a0 * a1 + a0` as a single instruction. Take for example that we have an array of 32 bit integers. The cell that we are indexing is found at index 4. Now we know that we have little endian packing and desire to have a byte index to the least significant byte of that same integer.

mul_idx(4 , 3) = 4 * 3 + 4 = 16

Thus given an array of integers, **`array[4]`** produces an integer, say **`i`**. For that same array now indexed in bytes, **`array[16]`** will be the least significant byte of **`i`**.

Unsigned integer representation

A scholarly recounting of the arrival of Arabic Numbers in European, and how the Europeans adapted them may be found in Charles Burnett's book "Numerals and Arithmetic in Middle Ages", DOI 10.33137/aestimatio.v9i0.25990. See the chapter, called "Why we read Arabic numerals backwards." It is a fascinating book that also explains such things as the origin of the word *algorithm* as being the name of the author who wrote a paper on how to use Arabic Numbers, and the Arabs referring to this system of representation as Indian Numbers.

When Arabic culture spilled into Europe translators changed the writing order of written documents, but they left digit sequences unaltered. Perhaps because they were afraid to disturb the math, or perhaps because account books and invoices were just too common and mundane to translate. In any case this led to some confusion where instructions describing how to use the numbers did not match the numbers shown in the same instructions. This confusion remains with us today in computer hardware. The exasperation among programmers is evident in Danny Cohen's paper, "On Holy Wars and a Plea for Peace" DOI 10.1109/C-M.1981.220208, where he coined the terms, *little-,endian* and *big-endian*.

The information about ENIAC in this chapter may be found in an old IRE article of 1947, "Electronic Computing Circuits of the ENIAC" by Arthur W. Burks, DOI 10.1109/ JRPROC.1947.234265. I found it interesting that some of the design issues for flip-flops made of tubes, aka valves, resembled those of designing static ram cells in CMOS. ENIAC represents a transitional machine from mechanical computer to electronic computing. Relative to modern computers it resembles its mechanical predecessors. ENIAC used ten digit numbers for computation.

Arabic Numbers

There is little doubt that the clever readers of this book are already familiar with Arabic Numbers. Rather the purpose of this chapter is to establish the language and terminology we will use for talking about them.

An Arabic Number consists of a sequence of digits, $d_0 d_1 d_2 \cdots$ where each digit may have a value ranging from 0 to 9. In this section we will represent such sequences using vector notation. For example, $\boldsymbol{x} = [7, 8, 9]$ is a vector with three components. Its zero index component, x_0, is 7, etc. Note that $\boldsymbol{x}$ is written in bold to denote that it is a vector.

We may interpret such vectors as numbers by using a weighted sum.

$$a = \sum_{i=0}^{n} a_i \cdot ten^i$$

Here a is a numeric value so it is not written in bold face. Each $a_i$is the ith component of the vector $\boldsymbol{a}$. The value a_i is also known as the 'ith digit' of the number. The value *ten* is called the *base* or the *radix* of the number. It is fortunate that *ten* is so well known that it has a name, because otherwise we might have been tempted to do what many other authors have done, and to write the base while using the very same representation that we are trying to define.

Although this function gives numeric meaning to our digit vectors - performing the suggested computation is pointless. The result would be a number, and we would have to represent that number, and that representation would be the very same vector that was input into the function in the first place!

In many contexts we may drop the decoration we have been using to signify a vector without causing ambiguity. It is conventional that when dropping the decoration that we also reverse the order of the components. So the vector from the prior example, $\boldsymbol{x} = [7, 8, 9]$ becomes $\boldsymbol{x} = 987$. Just to be clear, note that this number is 13 away from a thousand.

Independent if the number is represented with, or without, decoration, its *most significant digit* is the one with the greatest weight. In this example it is the digit with the value 9, i.e. x_2. Analogously, the least significant digit is 7, i.e. x_0. Using a non-standard term from the previous chapter on arrays, the *extent* of our example number is 2. The *digit count*, or *length*, of this number is 3.

Meaning of the word *digital*

The first computing machines which performed discrete state computation made use of ten symbols for a digit's potential values, just as we humans do when we perform manual arithmetic. Examples include Pascal's calculator, Babbage's machine, the many mechanical calculators that came after, Aiken's Mark computers, and ENIAC.

Mechanical machines used actual gears with ten positions each being 36 degrees apart. An index, such as a metal pointer, pointed at one of the marks thus indicating which digit value had been registered on the gear. To change which digit value was registered, the gear would be turned until the index pointed at the mark corresponding to the desired digit value. Instead of gears, the electronic computer ENIAC used circular shift registers of ten vacuum tube flip flops. These were called ring counters or decade counters. ENIAC operated on ten digit signed numbers, so there would be ten such ring counters plus a flip flop for each number. For the sake of this chapter I wish that ENIAC had operated on any other number of digits, say 20 digits, because I would not want the reader to conflate the number of *digit values* with the number of *digits*.

Mechanical machines, such as the Howard Aiken's Mark machines, gated rotational momentum with control linkages and clutches. ENIAC used an electronic analogy to the main rotating shaft, that of a central pulse clock. Pulses were sent to an electronic gate, and if the control to that gate allowed it, the pulses would pass through to the corresponding ring counters to cause each flip flop in the ring to flip in succession depending on the number of pulses.

To add to digit values on a mechanical computer, the two gears in question were mechanically connected. When one rotated back to zero, the other rotated up to the sum, possibly tipping a carry bar along the way. On ENIAC one ring counter would gate pulses to its partner while counted down. While pulses were gated to it, the second ring counter to count up, while possibly setting a carry flip-flop.

Hence the term *digital* initially meant both being discrete, and of making use of ten state logic. If we look at the control levers of the mechanical machines, or the control signals of ENIAC, we find two state switch logic, but computation units processed numbers consisting of ten state digits, just as human computers do. Early computing work gave us more than just techniques which we take for granted today, but also words such as gate, register, and index.

Earlier Claude Shannon and others wrote extensively on methods for simplifying switch logic, and it became apparent to that direct binary computation could be performed. The first electronic computer to do so was the Atanasoff-Berry machine developed in the late 1930s. Because there are two states in switch logic it is maximally efficient to use a base two number system for arithmetic. In such an arithmetic system each binary digit will have either the value of either zero or one. It is conventional to shorten the term *binary digit* to *bit*. John Von Neuman assigned Atanasoff to audio work during World War II, so presumably during this time the Atanasoff-Berry machine sat unused in a basement at Iowa State University.

Binary computation prevailed. By the 1950s almost all discrete state computers used two state switch logic for computation. Still the term *digital* continued to be used to describe them. This has led to some curious naming conventions. For example, the company *Digital Equipment Corporation*, which began shipping computers in 1959, only built binary computation computers. As another example a person who studies modern *digital electronics* will probably never see any wheels or decade counters used to implement operations in an ALUs.

Though the term *digital* continues to refer to computing with discrete state digits, the term has lost the connotation that there must be *ten* of those states.

Today computing based on ten state digits, such as we see on handheld calculators, is known as *decimal* computing. If we had a machine that made use of 256 state digits, it would be neither decimal nor binary. Because we do not have a special name for the number 256, we would say that such a computer uses *base 256 digits* and that it is a base 256 computer. Decimal computing, binary computing, and even base 256 computing are now all examples of *digital* computing. Though notice, independent of the base for computation, control signals are almost always on or off. We say these are Boolean or binary valued, sometimes referring to their values as one or zero, but note they are not numbers.

As computer implementations moved from ten state logic to two state logic it was hard to completely move away from the base ten number systems. This was particularly true in business applications where users want to see dollars and cents results that match computations done by hand, even when fractions and rounding are involved. Hence IBM came up with a scheme whereby groups of 4 bits could be used to represent the decimal digits, this code is called *binary coded decimal* or *BCD.* Even today IBM machines may be found where the machine instructions themselves expect their operands to be encoded according to BCD, and thus by definition, at least in one aspect, such machines have a decimal architecture in the old sense of the term.

Most computer users will never see a memory dump. Instead they will see the output of print functions. By default print functions render numbers in a decimal format familiar to the user. This happens independent of the internal representation used for numbers. In contemporary computing, the time it takes to convert a binary number to a decimal number is typically negligible compared to the time it takes to do a long computation.

Scanning-Order and Digit-Order

Had I written this book in Arabic, the text would have been written right-to-left. Lets explore what that looks like by using an example where we start with an English sentence and then reverse it. Notice that in this right to left string the vector $[9, 8, 7]$without decoration is 987. Both strings match, and this is a good thing.

♦

.987 si noitaroced tuohtiw [9 ,8, 7] rotcev eht thgir ot tfel morf gnitirW

When reversed this becomes:

♦

Writing from right to left the vector $[7, 8, 9]$ without decoration is 987.

Following the 1000 year old convention, I did not reverse the order of the digits of the number. For the reversed sentence to make sense I had to change 'tfel ot thgir' (right to left) to 'left to right' – this is the sort of thing that Middle Ages translators did not always do. Also I had to change the brackets so that they still enclose the vector components. Otherwise they would be pointing outward. A bad thing has happened here. When reading the number our eyes first land on a digit of unknown weight. We must scan further right to find the one's place, then count scan back while counting the digits to make sense of the number. It is noteworthy that reversing the string is not enough, we must also understand the sentence and change the phrase "right to left", to "left to right".

Now imagine, we start with the same right to left string, as the early translators did, but instead of making an exception for numbers as they did, we literally reverse the entire string:

♦

Writing from right to left the vector $[7, 8, 9]$ without decoration is 789.

Now both the scan order of the scan order of the vector and the digits are the same. This maintains the nice property that was part of the original design of the Indian Numbers. Note, that 789 here is still thirteen away from one thousand.

If per chance we desired to make things uniform again, we European culture writers would either have to start writing from right-to-left like the Arabs do, or we would have to reverse the order of digits in our numbers. This latter option has been implemented by computer architects. For example Intel processors do this.

In summary, in this section we have defined two concepts. One being that of *scanning-order* when reading and the other being *digit-order* of numbers. We discovered that if we change the scanning order without changing the digit order, then in effect the number will be read in reverse.

All conventional computers use low-to-high address scanning order for writing and reading text. Apparently, all humans feel that low-to-high address scanning order for text feels natural. Relative to the low-to-high address scanning order, *Big Endian* computers follow the current Western ideal that the big digit come first. In contrast Little Endian computers are more faithful to the design of Arabic numbers, and they store numbers in a consistent manner, with scanning order matching the order of the digit weights.

Because a page of text is two dimensional there are additional writing directions conventions. However, computer memory is one dimensional, so only the two we discussed here are relevant to our architecture discussion. If you happen to have the task of writing display drivers, then you might be exposed to some of the others.

Binary, Octal, Hexadecimal, BCD

We may interpret a vector of *n* bits as a number by using this function:

$$\sum_{i=0}^{n} a_i \cdot two^i$$

Just as for vectors of decimal digits, with a vector of binary digits we may drop the vector decorations to give us strings of bits. And, just as for decimal digit strings, we have two options for writing the string: either most-significant-digit-first, or least-significant-digit-first, i.e. the digit with the largest weight on the left, or the digit with the smallest weight on the left.

Although both strings of binary digits and strings of decimal digits grow in length logarithmic when counting - binary strings will grow more than twice as fast. The expansion to a length two string will occur immediately at the count of two. Length expands to three at a count of four, and then to string of length four at a count of eight. Hence, while incrementing to eight, the binary digit string has already expanded to length four, yet for the same count a decimal digit string remains one digit long.

It is not fun to write such long strings, so we often place bits in groups. When we group bits in threes, we will be working in base eight instead of base 2. This is called octal notation. We use one of the symbols 0, 1, 2, 3, 4, 5, 6 or 7 for the octal digit values.

♦

octal	binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

In the 1960s it was common to find computer panels with switches and lights organized in threes, and for coding forms to be filled out in octal. However, today almost all documents that must show bit strings will instead use groups of fours. Such a digit will have 16 values. We use the usual 0 through 9, and then continue with the letters, a, b, c, d, e, and f. This is known as the *hexadecimal* system, which is often shorted to *hex*.

♦

hex	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
a	1010
b	1011
c	1100
d	1101
e	1110
f	1111

It is not a coincidence that the table of hexadecimal digits is twice as long as the table of octal digits. Each time we add another bit to the grouping, the table will double in size.

Another common grouping is that of a *byte*. This name is word play on the term *bit*. Today a byte is universally a group of eight bits; however, computers of the past have used other numbers of bits, ranging from six to twelve bits. Vestiges of that past remain. K&R's "C Programming Language" leaves the length of a byte unspecified, The designers of UTF-8 wanted to make clear that they considered groups of eight, so they named such a group an *octet* rather than a *byte*.

When a group of eight bits, i.e. an octet, is considered as a digit of a number, we will be working in base 256 (here the number 256 is given in decimal representation. In hexadecimal representation the base would be written as 100.). Note *octet* and *octal* are different concepts. An octet is a group of 8 bits, whereas octal is a number system based on digits that have values running from 0 to 7.

Groups of bits can get larger than this. The organization of early RISC microprocessors was such that memory was always moved in groups of 32 bits, called *words*. Today it is common that address variables will be 64 bit words while integer variables will be either 32 or 64 bit words. Groups of bits found on internal buses might be larger yet.

We can also group bits to create numbers that have bases that are not powers of two. For example, in the BCD code we group bits in fours to create decimal digits.

♦

BCD	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

In BCD some possible bit value combinations are simply not used. This will always be the case for emulating a number base in binary switch logic when that base is not a power of 2.

Octal and hexadecimal are typically used merely as writing. Internally the computer will be computing in binary, i.e. base 2. In contrast, BCD computation is not merely a writing aid, because of the ignored encodings. When two BCD digits are summed, the carry must occur when the sum of two digits reaches ten or more, which is not on a power of two boundary.

Some processors do not support BCD computation but instead support BCD to binary conversion operations. So numbers that appear as BCD in memory are converted to binary before computation. However, other computers, most notably many IBM machines, will implement BCD architectures, i.e. have instructions for directly operating on BCD encoded numbers.

Although BCD encoding is less efficient, BCD encoded binary place values numbers still grow in length logarithmically while counting. Because numbers will be represented in base ten no number conversion has to be done when printing, but today this is of little significance. Perhaps more importantly BCD numbers divide by ten without creating infinite fractions.

Allocation

In most text documents a number is written down once and that is it. In contrast, while computing a program will often come back to the same memory location many times and change its value. This would be equivalent to going to a paper document and repeatedly erasing the old value, and then writing a new value in the same space.

Consider the case of recording a count in real time, where at any moment we might have to erase the current count value, and then write a new one. Suppose we begin at zero. Then when something arrives to be counted we increment to 1. We still have a single digit. Our count will grow to two digits in length upon reaching the count of *ten*. As we continue the count will grow by another digit in length when we reach a *hundred*. It grows again when we reach a *thousand*, etc. It is probably not a coincidence that we have names for the numbers at these boundaries. Thus a digit vector that represents the count grows in number of components, i.e. in length, against the log of the count value. The log function grows without bound, but it does so quite slowly, so the net effect is that relatively short digit vectors can be used to represent rather large numbers. This observation turns out to be key to understanding contemporary computer architecture.

If we had a paper document and only left space for a single digit count, we would run into a problem when the count grows to ten, and an even a bigger problem when it eventually grows to a thousand. If we knew in advance that we would want work our way up to counts in the thousands, we could simply *allocate* enough space for four digits to start with. Such an allocation by itself is not much of a waste of space because the digit vectors for very large numbers are not much longer than those for small numbers.

Yet, for any choice we make for amount of space to allocate, the possibility remains that a number will come along that is so large that its representation doesn't fit in the allocation, i.e. that the number will *overflow* the allocation. Basically to make reasonable allocations we must know how much numbers grow with the operations applied to them, and how many of those operations will be applied. This latter question could well be related to how long we plan to keep working on a given problem.

Just as numbers require space to be written on a page of print, they also require space to be written into computer memory. The conventional solution is to assume that all numbers will fit into fixed length allocations called *words*. With static allocation a map will be made of the locations of all the words to be used by the program even before the program begins. These addresses will be placed directly into the instructions of the program by the compiler, so there will be no option to decide on a larger allocation for a number at runtime.

If we assume that a program starts with initialized state, then when said program begins running, all numbers will fit in the words they have been allocated into. However, as the program runs, it is possible that larger and larger values will be computed then stored. This might not happen, but depending on the details of the program, it can happen.

In general it is not possible to write a program that can analyze another program and determine how many steps it will take before stopping, or if it would ever stop at all. Even in specific cases where it is possible to succeed at such an analysis, the analysis itself might be so difficult and time consuming that the programmer does not judge it to be a worthwhile exercise. IN practice such analysis is not done unless code is being placed into a life critical application, and sometimes, not even then. Because data might grow in length as a computation runs, computer users live under the Sword of Damocles of not knowing if an allocation will overflow and cause an error. If the sword does fall, the program might stop and emit an incomprehensible error, or it might even just continue blithefully on while providing garbage results. This does not surprise today's users. If bothered too much they might file a bug report. The unlucky real time computer user might fall out of the sky and not say anything, and then the error might be found in the accident analysis.

Conventional system memory is an array of allocation units called *bytes*. We call the indexes into this array *system memory addresses*. Being the minimum addressable allocation unit is the proper definition for the term *byte*. Hence the number of bits in a byte is a parameter of the computer architecture, not the compiler.

It follows that any larger allocation will consist of multiple bytes. For a given allocation unit, the smallest address among the contained bytes is taken to be the address of the allocation unit. Each allocation unit is characterized by two parameters, its address, and its extent (or length).

Let us put aside the question of scanning order for a moment and say that 'a number starts with' its least-significant-digit. This statement is justified based on the observation that independent of the writing scheme, a reader will have to read a number starting with its least-significant-digit to know the weights of the other digits. It then follows that the allocation scheme where the address of the allocation is the same as the address of the least-significant-digit is called *little-endian*. The definition of *big-endian* is then more complex. A big-endian number will have 0 padding to fill out the allocation. With big-endian, relative to starting at the least-significant digit, the allocation address will be either that of the most-significant digit, or that of the last zero in the padding.

In the terminology used in this book where we view memory as a horizontal tape with bytes in the cells, and addresses starting at zero and getting larger when moving to the right - little-endian numbers have the least-significant digit on the left, and big-endian numbers have the least significant digit on the right of an allocation. We can also say that little-endian numbers are 0 padded on the right, and big-endian numbers are zero padded on the left.

In Figure 10 we see a word that has bytes addressed represented in hexadecimal and running from *c0* to *c3*. (In decimal these addresses would be written as 192, 193, 194, 195; however, we almost always use hexadecimal notation for addresses.) The address of the byte before c*0* is the address *bf*. The address after *c3* is *c4,* etc. The address for the word itself is taken to be *c0*, because that is the smallest byte address. This word holds a little-endian number. If we consider that a digit of the number is a byte, and bytes are octets, the binary encoding for the least-significant-digit of this number is 0001 1000. The most-significant-digit is 1010 1110.

♦

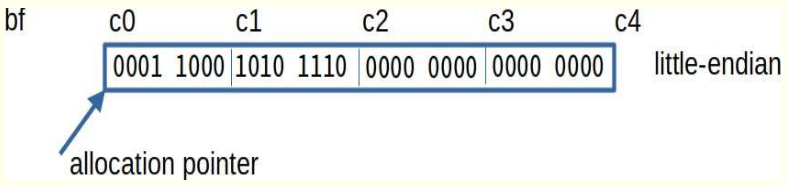


Figure 10 Left justified, least-significant-digit-first

In Figure 11 the same number is placed into the word using big-endian. For all but very large numbers, the digit being pointed at by the allocation pointer will be zero. We will continue to scan zeros until either reaching the end of the allocation or upon reaching the most-significant-digit. If we reach the end of the allocation, the contained number is taken to be zero. Because this is the same number as shown in the prior figure, it also has the same least-significant-digit and the same most-significant-digit.

♦

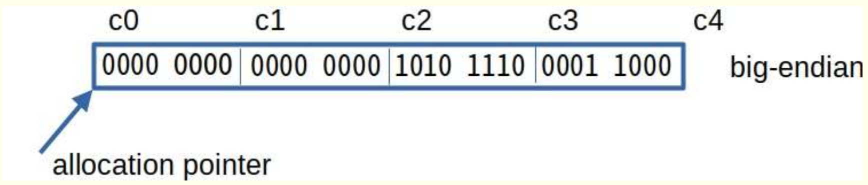


Figure 11 Same number using big-endian digit order

Now suppose our word is holding a count. When counting with the little-endian convention, a number will grow into larger addresses as the count carries into new digits. In contrast, with big-endian, counting will carry into ever smaller memory addresses.

Typically processor registers and busses are one word in length, and a processor will load the entire contents of a word of memory into a register as a single operation. Consequently the contemporary processors can be designed equally well for little-endian or big-endian number representation; however once designed for little-endian the processor will be incompatible with big-endian, and vice-versa.

Suppose we had an unconventional processor, or perhaps a future processor, which loaded numbers as digit streams. The load instruction would have to have a means of detecting the end of a number being loaded, or it would somehow have to know a-priori how long the number is. Such a problem is nearly identical to the already existing problem of loading strings of characters. For strings both length counts and end terminators have been used for solving the problem of knowing how many characters to load.

For serial computation, if little-endian notation is used, the processor may produce the first digit of the sum after receiving the first digit of the operand, etc. However, if big endian is used, the least significant digit will be the last to be loaded, so the entire number will have to be loaded before the first digit of the sum can be produced. If we try to load from the far ends of the operands this will not help, as we will not know where the far end of the sum will be located until we know if there will be a carry. A solution that would work for addition with serial big-endian is to perform signed digit arithmetic.

Bit Order Within Bytes

Data is transported between different points within a processor or computer system over bundles of wires called buses. Buses have specifications that include the order of bits in bytes, and for all contemporary machines bytes are octets. All compute processors, channel processors, and other devices plugged into a bus must conform to the bus’s specifications.

Channel processors are often used to bridge between a computer's system bus and a standard storage bus that connects to one or more storage devices. In which case the storage device designers only must know about the standard storage bus specification, and do not need to know anything about the computer at the other end of the bus. The compatibility problem for that computer on the end is limited to using a channel processor that respects the given standard.

Hence. unless a programmer is involved in the design of busses standards or conforming hardware interfaces, he or she will not even be aware of the order of bits within bytes.

This is not to say there can’t be differences. In fact some processors do store bits into memory bytes in different order than others. However, the values that are read and written into the respective memories must be placed on a bus, and at that point the bit order is described by the relevant bus specification.

On all contemporary computers it is possible to perform arithmetic on bytes. Carries always travel from the lower significant bits to more significant bits, consequently ALUs also impose a bit order within bytes. However, that order will conform to the order documented for the processor’s internal bus. In every document I have ever seen for a processor and buses, the ALU treats documented bit 0 as the least significant bit.

I challenge any programmer to try and write a C program that detects the physical order of bits in bytes of memory, or to identify a difference in bit order within bytes between computers. Because this is impossible in all but special cases, we typically do not say that a pointer to a byte of memory is pointing at either the most significant bit, nor to the least significant bit. Rather, an address points at the whole byte.

Byte Order Within Words

A given processor architecture might simultaneously have native support for byte data and various lengths of words. The most common word lengths being 16, 32, and 64 bits. Stated equivalently as 4, 8, and 16 bytes.

Most all communication channels and storage devices have payloads organized as octets, *but they have no built in support for words*. Hence when we desire to store a word or to communicate a word, we will be forced to first serialize the word into a series of bytes, then transfer or store the data, then to read the data back while deserializing the byte stream back into words.

Our Indian Number derived representations consist of vectors of digits. Words of allocation consist of consecutively addressed bytes. Bytes are handled atomically by all modern machines. Hence, any bit encoding for the digits of a number must pack cleanly into bytes, or the digits themselves would be split apart. Such a clean packing might require padding with zeros to make the length even by 8 bits. For example, a hexadecimal string of digits might be 12 bits long. Another 4 zeros must be appended to make it a multiple of 8 bits. When this criterion is met, we suffer no loss of generalization by considering that a number is a series of bytes as digits. It is because of this fact that little-endian and big-endian are often referred to as *byte orders*.

The following figure shows a stream of bytes arriving as data, and then that data being copied into a word. In this case the digits of the word, the bytes, arrive in little-endian order and they are received on a little-endian machine, so they are just copied in the order they are scanned off the channel.

♦

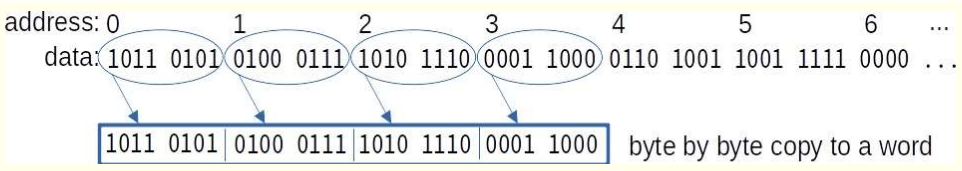

address: 0 1 2 3 4 5 6 ...
data: 1011 0101 0100 0111 1010 1110 0001 1000 0110 1001 1001 1111 0000 ...
1011 0101 | 0100 0111 | 1010 1110 | 0001 1000
byte by byte copy to a word

Figure 12 In-address-order byte by byte copy

In this second case the same stream of data arrives with words serialized as bytes in little-endian order, but this time the receiving machine is big-endian. The bytes will have to be reversed on a word by word basis.

♦

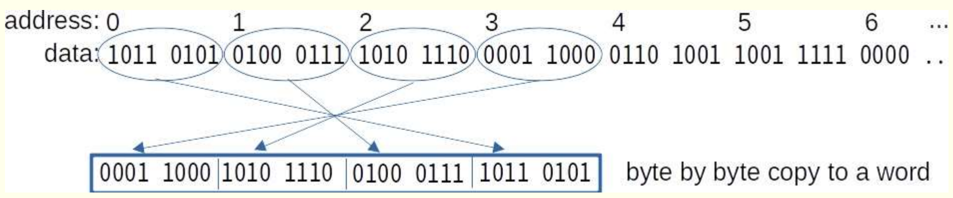

address: 0 1 2 3 4 5 6 ...
data: 1011 0101 0100 0111 1010 1110 0001 1000 0110 1001 1001 1111 0000 ..
0001 1000 | 1010 1110 | 0100 0111 | 1011 0101
byte by byte copy to a word

Figure 13 Reverse order byte copy for words

Here is the problem, when the data arrives there is no way to know where the words boundaries are. That information was lost when we serialized. If we do not know where the words are we cannot know when to do the byte order reversal. Hence this problem cannot be solved using the same approach that was used for solving the problem of bit order in bytes. The problem will spill over into the software.

At some level of abstraction all applications must be designed to be able to make sense of their data. It is just too bad that byte order within words has to be an application level design consideration, because it has nothing to do with applications.

In the case of something like the Internet Protocol, IP, the specification dictates the offset as to where data is to be placed into the packet header. Coding is done in a stable and efficient manner. However IP does not know where the words are located in the data payload it is carrying, so IP just passes the payload, byte order problems and all, up the abstraction stack.

JSON is a standard for expressing tagged structured data built from primitive types while using character only data. It comes with a specification saying how numeric character strings are to be interpreted. This makes it so that both little-endian and big-endian machines can share JSON character encoded numeric data. However, converting all numbers to strings and back is wasteful of computation time, and sending them over a channel is wasteful of bandwidth. In both cases more energy is used.

There are many file formats and data communications standards that serve various classes of applications by making it clear where words that need to be reversed are located when data is transferred between machines.

TTCA

The native integer data type for TTCA is based on high radix online number system. This number system is an extension of online arithmetic. Like online arithmetic it uses serial most-significant-digit first signed digit arithmetic. Unlike signed arithmetic the radix is much higher, typically causing a digit to be at least a byte long, numbers may be scaled, and at compile time an analysis step is recommended for setting estimated precision requirements and range bounds. This is the subject of the next chapter. Conversion instructions are provided for producing other number formats.