ROUGH MATERIAL FOR STICKY SITUATIONS

J. ALEX STARK

Contents

Date: 2016.

Code Examples

1. Rough material

1.1. Topics.

- Kinds of structures: config (Builder, add, build paradigm), data bundle, object
- POD.
- Enums and FSMs
- Graphs
- Impl
- Execution context vs injections

Labelling situations. Wildcard situation names. Continuing to the next handler.

1.2. **Mumble.** If For Return Break Continue While Until

Also distinguish *semi-pure* functions. These can have limited side effects, eg for logging or counting how often called. Basically we don't care how often and when it gets called.

Sometimes we want to merge situations, and encapsulate the typestate in data values. We will need some mechanism for this. It could be used to store the actual typestate in a container, or to complete the construction of an*any* object.

1.3. **Utopianism itemized.**

- Utopia27 tries to keep code lightweight. By this, I mean that the it does not place much extra burden on programmers compared with C++. It can do this because, for the most part, standard cases apply. Only the exceptions need be annotated explicitly. For example, customized typestate is only needed in a minority of cases: *const* and *non-const* forms suffice for the great majority of cases.
- Utopia27 makes a strong distinction between mutability and immutability. It reduces the reliance on the distinction between values and references. In line with C++, it permits control between stack and heap allocation.
- Utopia27 provides improved handling of reachability and exclusivity of ownership, and of transfer of ownership. Situations aid in keeping this simple and lightweight. For instance, if a function cannot fulfill its obligations in some circumstances, it can return with a situation under which the calling code path diverges.
- Utopia27 uses richer typestate to control the handling of member fields, eliminating the need for such things as null pointers used as sentinel references.

2. Typestate

c:multiLevel

2.1. **Multi-level.** How do multiple levels of typestate of compound data relate to the typestate of its members? Simple case: initialized structure's const members are in at least initialized state when it is promoted. It is moot whether they are immutable in of themselves: accessor makes them const.

sec:unwind

2.2. **Unwinding: Setup and tear down.** Also discuss this as an example where design decisions are adversely influenced.

Often in a function, you may allocate resources and need to exit in multiple places. Function, or construction, clean-up. (We discuss function clean-up, but often mean construction, and construction need not necessarily be done in a constructor function.) Unwinding versus not unwinding. Programmers can simplify their code by putting the resource cleanup code at the end of the function all "exit points" of the function would goto the cleanup label. This way, you don't have to write cleanup code at every "exit point" of the function.

In my opinion one can learn a lot about a computer language from style guides. Books and articles concerned with effective usage also say much about how programming concepts work out in reality. We look forward to the publication of *Effective Utopia27*, and *57 Ways to Improve Your Utopia27.*

A.1

Should probably only allow chained or unchained situations, and not a mixture thereof.

Unwinding in stages is not trivial, and this is reflected in the pseudo-code. Note that the Utopia27 compiler checks the structure, since the setup and unwind processes change the typestate of the various data fields and variables. For example, the section for unwinding allocations reverses the typestate changes made under resource allocation. All execution paths that complete the allocations must pass through the section, and those that do not must skip it.

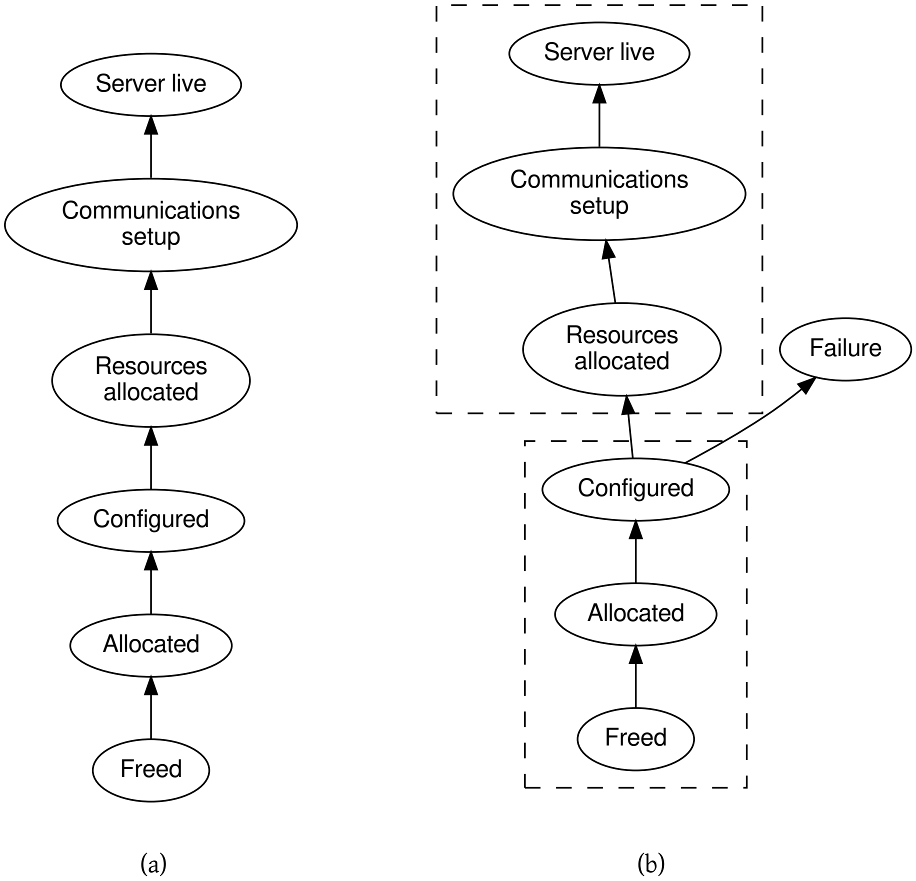

FIGURE 1. Multi-level typestate. (a) A generalization of the simple typestates of figrefSimpleTypestate is to have a customized main trunk, and (b) with a side branch. UTOPIA27 supports this kind of tree in a standardized form. The states on the main trunk are most often considered to be levels with increasing readiness. The concept of levels makes it easier to specify constraints on the states. For example, we can specify that the state is at least as high as the allocation of resources, or no higher than configured. Side branches, such as a problem in resource allocation, complicate the picture a only a little.

fig:TypestateLevels

3. INTRODUCTION

3.1. **Initial thoughts.** After decades of computer programming it is hard to argue that there are any *silver bullets* to be found in software development: Fred Brooks' assertion stands firm. This is not for want of looking. Many languages and software development practices have been explored, many of them in small-scale experiments, some widely applied. Overall, one has to say that most genuine improvements have been incremental. Nonetheless, I do not think that we should stop trying. Rather, significant improvements are within easy reach, and these may be found by looking hard at the most basic aspects of programming language design. Moreover, while many good ideas may have emerged recently, we may need to revisit some

Code example 1: Unwinding

```
function RunService(options):
   when (ProcessOptions(options)):
      log.Error("Configuration problem.")
      -> OptionsProblem
   /
   when (AllocateResources(options)):
      log.Error("Allocation problem.")
      -> AllocationProblem
   /
   when (SetupCommunications(options)):
      log.Error("Problem opening communications channels.")
      -> CommunicationsProblem
   /

   <The real work...>

   <Unwind communications setup...>
   continue
|== CommunicationsProblem:
   <Unwind allocations...>
   continue
|== AllocationProblem:
   <Probably little unwinding needed for 1st stage...>
   continue
|== OptionsProblem:
/
```

Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it!

that were proposed a long time ago, yet did not make it into the mainstream. I also think it important that, as we do this, we try to avoid falling into familiar patterns of thought and argument. (I will try to present some pertinent examples.)

As I have thought about the challenges of programming, considered ideas and possibilities, and struggled to write this article, I have become convinced that the challenge that lies before us will not be met by finding a neat and coherent solution. Our reality is messy. Might it be possible to take, say, C++ and add a little "complexity" here and there such that the result is actually not more complex and yet more reliable, secure and maintainable? I think we should try. (Aside: some of the complexity in existing programs is implicit, and in new languages this will be explicit. As a result there will be a superficial increase in complexity.)

Situations, which we will define in detail later, recur as a theme throughout our discussion. We will touch on many topics, but situations will be the glue holding everything together. The name *situation* was used by Knuth over forty years ago in describing a restricted branching control structure. Knuth was not the only one to propose it, and there were others that were very similar. However, I think that this word is very helpful. Knuth realized that it was important to distinguish between events that occur at a point in time, and conditions that occur at a particular location in a program execution. He noted that *situation* is apt for the second case because it has the dual connotation of *state of affairs* and *place or locality*.

While situations bring benefits on their own, the purpose of this article is to examine wider possibilities. Actually, a strong motivation for writing is that I am frustrated that we still have code like the following (in pseudo-code).

```
function(*inputStructure input_data):
   if (input_data == null):
       <Handle null-input situation...>       ; This should never happen.
   /
   <Code that is always executed...>
/
```

In other words, our code is typically riddled with checks for situations that should never happen and often even cannot. The problem is not solved by removing pointers from a language, or managing memory automatically. The function above is only ever called with parameters in a particular state or set of states, and so `input_data` can never be invalid. But this is not codified in a contract. The undesirable consequences of superfluous checks are many. One is that testing is awkward. How does one test something that cannot occur? A further consequence is that decisions about the organization of code, such as splitting into sub-routines, is strongly influenced by the awkwardness of testing.

Another language concept that we will explore is *typestate*, which we define as state information that augments data type information and that is associated with data at points in program execution. Typestate helps us (and compilers, etc) understand how the state of data unfolds as the program executes. In the above example, the function *expects* that `input_data` is valid when it is called, and the caller is required to *assure* that it has that typestate. When requirements are placed across boundaries such as calling interfaces, we will refer to them as *contracts*. However, we want to go further, and consider how it might be helpful to examine assurances and expectations across other boundaries. For instance, one could draw a line around a code block such as an iteration, or draw a line between two sequential lines of code. If code that follows a boundary uses a variable, it expects the variable to be defined, and the code that precedes the boundary must assure that it has been defined.

Not only do situations provide a way of describing (some) branches in programs, they also provide a clean way for describing alternative movements through code boundaries. For example, if the execution of a block of code encounters an unusual situation, it may exit early and with different typestate. The execution with the unusual exit moves to a different point in the code. I want to suggest an extension to Knuth's distinction between events and situations. I think that situations should in most uses differentiate *infrequent* code paths from *frequent* paths. They should not be used to differentiate the *exceptional* from the *normal*. If a request to open a file fails due to lack of permission, this failure is not exceptional. The program should deal with it as a normal, though infrequent, unfolding of execution. A normal infrequent situation like this should not be handled with expensive handling, but with an inexpensive diversion. Situations and typestate enable us to deal cleanly with the different data states that arise when execution unfolds like this.

3.2. **Utopianism.** There are many different sub-disciplines in computer programming, and even the best of us do not always appreciate the challenges faced by another programmer working on a different kind of problem. We can only speak from our own experience, direct or indirect. Furthermore, no languages is even nearly optimal for every task. What we will do as we proceed is to imagine what a language in the future might be like, a language that in broad terms replaces C++. This narrows the field somewhat, at least eliminating the kinds of tasks to which, say, Python, shell scripting, SQL, and so on, are better suited. It should work quite well for a web server, for image processing, for interfacing with drivers in a C library. In his 1974 paper, Knuth imagined a language he called UTOPIA84 . Since his follow-up letter to that paper is pivotal to our discussion, let us honour him by calling our imaginary language UTOPIA27 . I hope that the irony and humour in this name will help us avoid taking ourselves too seriously.

- UTOPIA27 tries to keep code lightweight. By this, I mean that the it does not place much extra burden on programmers compared with C++. It can do this because, for the most part, standard cases apply. Only the exceptions need be annotated explicitly. For example, customized typestate is only needed in a minority of cases: *const* and *non-const* forms suffice for the great majority of cases.
- UTOPIA27 performs simple deductions about types but does not try to be especially clever, and certainly does not try to prove anything much about a program. There are many reasons for this. (a) Anything but simple deductions will slow down compilation. (b) If the rules are simple, a programmer can anticipate what UTOPIA27 can figure out and what needs to be specified explicitly. (c) A lot can be done without being clever.
- UTOPIA27 makes a strong distinction between mutability and immutability. It reduces the reliance on the distinction between values and references. In line with C++, it permits control between stack and heap allocation.
- UTOPIA27 provides improved handling of reachability and exclusivity of ownership, and of transfer of ownership. Situations aid in keeping this simple and lightweight. For instance, if a function cannot fulfill its obligations in some circumstances, it can return with a situation under which the calling code path diverges.
- UTOPIA27 uses richer typestate to control the handling of member fields, eliminating the need for such things as null pointers used as sentinel references.

3.3. **Additional notes.** UTOPIA27 is designed to provide improvements in productivity and security. As mentioned above, I think it impossible to make general claims in computer programming, since there are far too many areas of specialization. Nonetheless, we will try to avoid obvious pitfalls. Also, UTOPIA27 has large collaborative projects in mind. In particular, this means that it tries to encourage good code organization and supports code exploration tools (such as Doxygen) in assisting developers who are unfamiliar with a code base.

One should always be wary of clever and fine-sounding words. I think this especially true of software development. Experience counts. Ideas have to be tried out in real and substantial projects across the whole range of target specializations. Thus, any thing put forward here should be seen as suggestive. Moreover, nothing here is really new. I hope that the combination of ideas, incomplete and imperfect though it is, will help to further the movement toward practical and effective language design.

We use a pseudo-syntax in this article, with '`:`' indicating the beginning of a block, '`/`' ending it. Also, '`|`' indicates a division at the block level, such as an else sub-block within a conditional block. Line comments begin with '`;`', in the style of Lisp. I realize that UTOPIA27 is likely to adopt brace syntax, because that is the most familiar. However, for this article I wanted to use something a little more like an 'algorithm' style.

4. SITUATIONS

4.1. **Break and continue.** Situations are quite simple. They are labelled exit paths that may only move outwards in scope and downwards in the written code. They can do the job of the `break` and `continue` keywords in familiar programming languages; let us begin with these uses. Code example refcodex:continue iterates through a set of elements. The code skips processing of any of the elements for which a condition is met. We could also say that, if a specified situation is encountered it is dealt with by transferring outwards to special handling code. Situations require the programmer to provide their names and arrival points. In contrast with `continue` control flow, this means that a piece of documentation is associated with each situation, that there can be multiple situations, and that the iteration being interrupted is specified. For instance, one can jump out of nested iterations.

Uses of the `continue` keyword are relatively few, whereas uses of `break` are more common. An illustration is given in code example refcodex:break, in which an array of elements is traversed, and processing of some elements is terminated early for some reason. In order that situation code be clear and readable, it is required that the handler be at the end of an enclosing scope. In the example this is slightly awkward, because we have to place the code in an artificial block structure. In reality the block structure would be an enclosing function, or something like that. In its purest form the `break` is simpler than an implementation using situations. However, having looked at a lot of examples, I think that in practice situations would allow for more elegant code. This is because the reasons for leaving an iterator are typically not simple, and it is an advantage to be able to provide particular handling code.

It is perhaps clear why we chose a symbol (‘`|`’) to indicate a division point in our pseudo-code. In traditional languages a small set of keywords (`else`, `case`, and so on) indicate such a break. In UTOPIA27 these are more common. I think that, at least in pseudo-code, this symbol helps to show the code structure. Furthermore, the *only* unexceptional way that a control structure can be exited is via a situation. Since they are only allowed to move downwards and outwards, when one sees an arrival point, one can look back within the scope to find the departure point and how the situation was established.

4.2. **Limitations of traditional control flow.** The control flow structures of traditional programming languages serve most needs of programming quite effectively. However, there are quite a few cases that cannot be expressed well. We will consider some examples briefly now, and expand on some of them later.

Exiting nested loops: It is awkward to exit more than one level from a nested loop.

Code duplication: When multiple conditions need to be considered separately and there is code to be executed that is common to them, it is often easiest to duplicate the code. A typical way to avoid this is to break the common code out into a function.

Invented Boolean variables: One means of working around deficiencies in control structures is to use a Boolean variable. This might be set to false before a section of code, and is set to true only when a particular situation arises. An example of a comment on this practice is WilliamsD084 [12, section 2.2, *Deeply Nested IF-THEN-ELSE Considered Harmful*]: “My personal summing-up of the theme of these papers is the following: at the point where the programmer has to ‘invent’ a boolean variable merely in order to map the problem onto the available control structures, then this ‘invented’ variable is as dangerous and confusing as the poorly regarded GOTO statement.”

Invented functions: Another means of working around deficiencies in control structures is to wrap functions in functions. This is even less desirable than inventing Boolean variables.

Setup and tear-down: The main work of some functions needs to be preceded by a succession of setup steps and followed by a reversed succession of tear-down steps. Perhaps resources need to be allocated and freed, or perhaps something needs to be configured. Some of the setup steps might fail, in which case we typically return prematurely from the function. With traditional control structures unwinding incomplete work is not straightforward. We will describe this kind of scenario in more detail in section 2.2

Handling infrequent but normal situations: Traditional control structures do not provide a way to handle infrequent situations. They have to be treated as exceptional.

Deficiencies such as these lead to code that is hard to understand. Moreover, programmers’ decisions about code organization is too often influenced or even determined by the limitations of a language instead of its provisions. In discussions of languages I have often read denigrating comments along the lines of “you may encounter difficulties if you have painted yourself into a corner”. This is nonsense. Also, techniques for circumventing language deficiencies are sometimes described as programming idioms. Plenty of idioms are good, but just because we have become accustomed to doing something a particular way does not mean that it is the best way. about this.

4.3. **Unexceptional.** Situations in UTOPIA27 can be placed in most locations. A situation has one or more departure points and one arrival point. They all have the same label and the arrival point must come later in the code and further out in scope than all departure points.

When situations were discussed forty years ago the formulations typically required that all situations be declared in a list at the beginning of the block structure, at the same level as the arrival points. I think it better that this not be a requirement, since it would impede acceptance of situations. Brevity is much preferred, and code exploration tools (and automatic documentation tools) could annotate control structures with a list of their situations.

Situations, with minor variations, were proposed by quite a few writers in the early 1970s. Among them is Knuth, who made various interesting suggestions in his 1974 article on structured programming KnuthDE74 [7]. Even more enlightening is the note KnuthDE75 [6] that Knuth and Zahn wrote in clarification. We quote this in entirety here. (We have changed the line breaking in the syntax example to be more like that in the original article. The columns were quite narrow in *ACM Forum*.)

> **Ill-Chosen Use of "Event"**
>
> The recently proposed event-driven case statement ZahnCT74 [13] has been described ZahnCT74,KnuthDE74 [13, 7] in terms of "event- indicators" and "event-statements." A so-called event-statement asserts that an indicated event has occurred and the appropriate context is thereby immediately terminated.
>
> Further reflection has suggested that our word "event" was ill-chosen. An event, in common English usage, is a happening or occurrence associated with a passage of time (as the performance of an assignment statement). As such, "event" is essentially synonymous with "action" as used by Dijkstra DijkstraEW72 [4]. The discovery at a particular place in a program execution that certain variables are in a specified state is better described by the word *situation* which has the dual connotation ACDict [1] of "state of affairs" and "place or locality."
>
> We would, therefore, like to propose that this control statement be called the "situation case statement" with situation-indicators denoted by identifiers ⟨situation$_k$⟩, and perhaps with the following syntax.
>
> ```
> until ⟨situation$_1$⟩ or ⟨situation$_2$⟩ … or ⟨situation$_n$⟩:
>     ⟨statement$_0$⟩
> then case
>     begin ⟨situation$_1$⟩:⟨statement$_1$⟩;
>               ⋮
>           ⟨situation$_n$⟩:⟨statement$_n$⟩;
>     end
> ```
>
> Within ⟨statement$_0$⟩ a situation statement denoted by the appropriate situation-indicator causes ⟨statement$_0$⟩ to be immediately terminated and the appropriate case selection to be performed. More generally, parameters can usefully be introduced into the situation statements and into the corresponding parts of the case clause; see KnuthDE74 [7].

> It is interesting to note that the word "event" seems to also be improperly used in many discussions of concurrent processes and their mutual synchronization. Processes must usually defer their activity until particular conditions or situations arise. Some clarity might be gained by consistently using the words condition, situation, and event, as follows:
>
> condition ≡ a state of affairs involving program variables
> situation ≡ a condition occurring at a particular location in a program execution
> event ≡ the occurrence of some action usually altering the state of program variables.
>
> Donald E. Knuth
> Charles T. Zahn Jr.
> Stanford University
> Stanford, CA 94305

We embrace this almost in entirety. As noted above, we think that situations should be implemented without the need to declare all new situations at the beginning of a block. Let us define situations, as entities rather than as a control structure mechanism.

Definition 1 (Situation)**.** A circumstance occurring at a particular location in a program execution. A circumstance is typically a condition, which is a state of affairs involving program variables, reflected either in their values or in their state.

This does not say how situations should be used, and I want to propose one extra criterion. It came out of reading debates and comment threads concerning exceptions, and how return values can inform callers of abnormal outcomes. (These are the sorts of threads that grow out of discussions about the Go Language, for instance.) Naturally, I have wondered how situations might be used in the examples described in such debates. I have come to think that we need to make a clearer distinction between what is infrequent and what is exceptional. For example, some criticisms of exceptions may be incorrectly framed. Exceptions may (or may not) be good in handling exceptional cases, but they are not the best way to handle normal cases, and moreover situations may do so better.

I think that the ways in which situations are used should vary with the distance of the throw (how far the arrival point is from the departure points). Some are short, such as when acting like a `break` in an iteration. I think it not necessary that such execution paths be infrequent: the probability of the situation being triggered with each condition can be even. However, when the throw is longer I think that good practice is that the situation be infrequent. In either case, it may help the compiler if situations are infrequent since the imbalance in diversion frequency can be used in optimization.

5. Typestate

5.1. **Typestate.**

Readers who were programming twenty years ago may remember when compilers did not analyse the chains of definitions and uses. Now we assume a compiler will tell us if we fail to set a variable before we use it. There was a period in which style guides would instruct programmers to initialize all variables explicitly when they were declared, just as a precaution. Even today, the formal definitions of many languages say that default constructors will be applied. (While often allowing compilers to skip unneeded construction.) It is this kind of analysis that is made easier and improved by typestate in combination with situations.

Typestate augments the familiar *type information*. Type information controls how data is stored, how it is manipulated, and how it is interpreted. Typing can be hierarchical; it can be *static* and *dynamic*. And so on. Typestate, on the other hand, expresses how the state of data *changes* through *execution* paths. It facilitates communication between the programmer and compiler as to whether data is initialized, accessible or owned at a point in program code. It might track constraints such as the range of an integer variable. Let us define it formally.

Definition 2 (Typestate)**.** Typestate is state information that augments data type information. It is associated with variables, being instances of a particular type, at points in program execution. It is used to determine at compile time what operations can be applied, and to eliminate validity checks and range checks.

FigrefSimpleTypestate illustrates the typestate of variables with traditional classification. Mutable value data only needs two states: *allocated* and *initialized*. After initialization it may be modified, but that does not change its state. A reference variable at first does not point to allocated data. Constant data has an extra *immutable* state. Compound data structures operate recursively. They are initialized if and only if all members are initialized. If treated as constant during a part of a program, then accessed members are also in the constant typestate.

One might well wonder if typestate is rather complicated. Furthermore, many programmers might say that "C++ serves the great majority of cases quite well". I think that this claim can help us in harnessing the benefits of typestate without creating undue burdens. The trick for UTOPIA27 is to simplify the basic cases of variables, whether value or reference, mutable or immutable ("const"). Only when it is advantageous to go beyond the basic cases should one have to qualify the code more fully and explicitly.

Side branches, such as failure branches figrefSimpleTypestated add a little complexity. This is a good example of how complexity can be introduced gradually, on necessity, and with verbosity of code minimized for the most common uses. In UTOPIA27 member fields are assumed to be initialized only in the structure's initialized state. Fields marked for the error state are assumed to be initialized only in the error state. (This is not only convenient but means that assumptions are handled uniformly across branches.) Any members that are to be initialized in both branches require more verbose annotation.

5.2. **In between.** What happens when one part of a compound structure has been initialized and another not? While a structure is in the process of being initialized, UTOPIA27 allows it to be between two states. (Defining explicit in-between states would add overall complexity.) A potential difficulty is that code might be interrupted at such a point. Correct behaviour is ensured because initialization of compound data requires sufficient ownership. An interrupting routine should not typically be accessing the same structure. In contrast, if it is *not* catastrophic to leave a structure partially initialized, then extra states should be defined. We discuss this in section 2.1

In order for UTOPIA27 to handle between-states, the edges are directed. In the direction of the edge constraints may only be added. There are often multiple constraints. For example, when a compound structure moves from allocated to initialized, its members can be initialized in any order. The typestate changes of the member fields are represented as a set of independent additional constraints on the parent structure. Apart from clarity, simplicity and speed, this rule of direction and addition of constraints greatly helps with error messages. If a structure ends up in an incomplete state the compiler can say something like: "your data structure was allocated at line x and needs to be initialized at line y, but member field z is not yet initialized". The restriction on the edges in the typestate graph ensure that the set of constraints that can be violated is reasonably simple and can be explained in words.

There is a slight subtlety in here. In the allocated state, a compound's members are typically all required to be in an allocated state. Thus, when the structure is in transition from allocated to initialized, it is not so much a case of additional constraints being added to the members as their constraints being changed. Nonetheless, the natural interpretation of these states is that one is higher, or more complete, than another. Therefore UTOPIA27 imposes direction on every typestate transition. To maintain a lightweight language, we need to be able to assign constrain-ed-ness automatically to all relevant entities. The main rule that UTOPIA27 uses for this is to assume that any enumeration has an implicit ordering according to the order in which the enumerations are first defined. (This can, of course, be customized.)

5.3. **Multi-level.** How do multiple levels of typestate of compound data relate to the typestate of its members? Simple case: initialized structure's const members are in at least initialized state when it is promoted. It is moot whether they are immutable in of themselves: accessor makes them const.

5.4. **Unwinding: Setup and tear down.** Also discuss this as an example where design decisions are adversely influenced.

Often in a function, you may allocate resources and need to exit in multiple places. Function, or construction, clean-up. (We discuss function clean-up, but often mean construction, and construction need not necessarily be done in a constructor function.) Unwinding versus not unwinding. Programmers can simplify their code by putting the resource cleanup code at the end of the function all "exit points" of the function would goto the cleanup label. This way, you don't have to write cleanup code at every "exit point" of the function.

In my opinion one can learn a lot about a computer language from style guides. Books and articles concerned with effective usage also say much about how programming concepts work out in reality. We look forward to the publication of *Effective UTOPIA27*, and *57 Ways to Improve Your UTOPIA27*.

A.1

Should probably only allow chained or unchained situations, and not a mixture thereof.

Unwinding in stages is not trivial, and this is reflected in the pseudo-code. Note that the UTOPIA27 compiler checks the structure, since the setup and unwind processes change the typestate of the various data fields and variables. For example, the section for unwinding allocations reverses the typestate changes made under resource allocation. All execution paths that complete the allocations must pass through the section, and those that do not must skip it.

6. Revision marker

7. Contracts and Constraints

7.1. **Constraints.**

- Initialized completely, or not at all.
- Initialized in an intermediate state, and whether a member reference leads to data of a specific typestate.
- Range data, especially of integers.
- Exclusivity of ownership.
- Disjointness.

Main reason for more than basic levels is with compound data whose members may have different allocation and initialization patterns.

Such states are not for FSM emulation. Rather, they reflect data validity

Levels of state.

7.2. **Lightweight contracts.** One contract may be a subset / subcontract of another, with additional constraints. Also allow intersections and subcontracts thereof. Maybe also provide a *trust-me* declaration for overlaps that cannot be trivially expressed.

7.3. **No returns.** Our choice in presentation of UTOPIA27.

8. Not Yet Discussed

RAII.

9. Reach and Ownership

9.1. **Richer object state: Member state.** Multilevel. Branches in state, eg file open or unable to open. Large objects initialized in stages. Objects representing systems that are initialized through multiple stages.

9.2. **Richer object state: References.** Ownership. Exclusivity. Uniqueness. Sharing, passing.

10. Discussion

- Infrequency
- Typestates, levels. Initialization via levels and alternatives via branching.
- Get rid of RAII. Rethink initialization, ownership.
- Consider getting rid of break and continue and return.
- Situations, or something else. Bifurcation of typestate paths. Initializer function alternative contracts therefore alternative return.
- Realistic contracts. Fast arrays and safety. No more “this cannot happen”.
- Need. Lifetime productivity. e.g. including code exploration.
- Need. Much of existing languages are fine. There are significant weaknesses such as checking for the impossible.
- Need. Some aspects are semi-formal, such as style guide or effectiveness guides. e.g. Passing ownership across function call. Bring in to language.
- Need. Differences in public interfaces, such as APIs and wire protocols. Bring annotations in to main language.
- Need. Lightweight. Common cases. Slightly different approach. Instead of assuming common case and permitting deviations, design constructs for general range and elide in common case.

This paper does not present answers, just suggests avenues to explore. Impact of current offerings not good.

- Security is weakened. Why? Awkwardness distracts. Poor organization confuses. Testing is more difficult.
- Productivity is reduced. Mental effort when expression is poor. Full range of activities not supported well.

Examples of problems with existing languages.

- Awkwardness of existing control structures.
- Invented Boolean variables.
- Splitting off into functions when that is not best, but control structure limitations demand.
- Unnecessary checks, eg null pointers. Or language checks anyway, and exceptions can be raised. Weaker compile-time checking.
- Not splitting off into functions because spurious checks need awkward or impossible testing.

- Severely limited states for types. Loss of expression of intent, clarity when changes are staged.
- Not organizing properly into functions when, say, initialization is staged.

10.1. Principles.

(1) *You cannot make a virtue out of a vice in programming by calling it an idiom.*
(2) *In discussions about computing, it is easy to conflate.* Example: exceptions and the infrequent. Example: configuration structures are different. Furthermore, *computing encompasses very different application areas and each of us experience a small minority.*
(3) *Style guides and advice on effective use often tell us as much as a language manual.* Furthermore, *good ideas in computing should be rejected.* The reason is that all programming features, however beneficial, get used where they should not and in ways they should not.
(4) *We often fail to strive to deliver on basic needs.* An example is that of code organization. We are taught that functions are useful because they enable us to divide up code appropriately. We have seen that this is does not work out in practice.

11. Not Revised

Principle: It is not good to make decisions on code organization, such as whether or not to place code in a separate function, based on awkwardness of the result. Example: putting in a function to avoid duplication, or not putting in a function to avoid difficulties testing in the absence of contracts.

11.1. Style guides.

11.2. Things about Utop to cover.

- In a small percentage of code the programmer has to provide guidance, principally by assuring that some constraint is met. These assurances are the things that in C++ one would want to document and test.
- Large immutable objects never should be, and in many cases never were, copied even if passed by value. Utopia27's improved concept of ownership and transfer of ownership eliminates copying except when data is clearly forked.
- Tracing code. Simple measures can improve local-scale tracing.
- Good code organization. Remove perverse disincentives, such as testing disincentives, not checking contracts, not supporting initialization in stages.
- Bad stuff, like switch statement.

11.3. A basic example.

11.4. More variations. The typestate is different in the two situation handlers. In one the `neededItem` is defined (fully initialized), whereas in the other is not valid.

In order to be lightweight, no upfront declarations. However, could list all situations. This alternative loop considers finding an item to be a positive outcome.

11.5. Sequential conditions. These are tasks where there are a set of conditions to be checked, but where the data of one depends on results that can only be calculated when another is false. The classic example in pointer languages is where a null pointer is checked, then the value pulled, and then a decision made on the value. Reasons for breaking out rather than one big conditional:

(1) Trace outcomes specifically via counters;
(2) Test of `header` ("suitable format" in example) might require more involved processing, and so require commands to proceed it; and

Code example 2: Alternative styles. Leaving an iteration.

Original in pseudo-code:

```
block:
   loop item through list_of_items:
      if (isWhatWeNeed(item)):
          neededItem := item
          -> ItemFound
      /
   /
   <Handle not-found situation...>

|== ItemFound:
   <Handle situation: (neededItem defined)...>
/
```

Perhaps heavy-handed

```
block:
|---$ (ItemFound, ItemNotFound)
   loop item through list_of_items:
      if (isWhatWeNeed(item)):
         neededItem := item
         -> ItemFound
   /  /
   -> ItemNotFound

|== ItemFound:
   <Handle situation: (neededItem defined)...>
|== ItemNotFound:
   <Handle situation...>
/
```

When procrastinating

```
bool found
item neededItem
block:
    loop item through list_of_items:
        if (isWhatWeNeed(item)):
            neededItem := item
            -> ItemFound
        /
    /
    found := False

|== ItemFound:
    found := True
/
<Subsequent use of found...>
```

A situation that is traditionally implemented using the break statement. There are many cases in which we need to leave a loop when a condition is met. Perhaps the termination situation cannot be encapsulated in a single expression. Perhaps there is a reason to leave in addition to the usual termination condition. One advantage of situations is that we can easily use common code between multiple situations. (Note that the block structure used here could be any block control structure, such as a function or conditional.)

Code example 3: Explicit situation list

```
function multiExitFunction():
|$$$(ItemFound, ItemNotFound)
    loop item through list_of_items:
        if (isWhatWeNeed(item)):
            neededItem := item
            -> ItemFound
        /
    /
    -> ItemNotFound

|== ItemFound:
    <Handle situation: (neededItem defined)...>
|== ItemNotFound:
    <Handle situation...>
/
```

Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it!

(3) The `header` variable might be used for subsequent tests or processing, and so we would not want to parse it more than once.

Idea: Have slightly different syntax for situation handlers that land their from multiple departure points. *Likely style guide:* Either have many all-single situations, or two (success / failure) merged situations.

11.6. **Nested conditionals.** Could be balanced, not frequent-infrequent.

11.7. **Iterator.**

11.8. **Worked example.**

12. Definitions

12.1. **Typestate.** static-dynamic state.

For every typestate (set), an enumeration is automatically defined.

All these object could have versions with the state as a member or meta-object member field. Perhaps some form of boxing. Any boxing mechanism would need to be able to specify the typestate that would encapsulated within, since we permit multiple typestates. Maybe should say *reified*.

Not good enough to erase type information for generics. Not all lists are alike.

Aside: Compile-time constants in Dart with equal constructor arguments result in same canonical instance. This implies keeping copies at compile time.

Typestates as arranged in levels. Unalloc = 0/1, alloc = 1/1, init = 2/1, final = 3/1, const (compile-time) = 4/1. Error states run as branches with parallel levels. In staged initialization, some members may be final on entry, others alloc, but the level of the item is determined by how much is initialized (2 or above). Similarly, partial allocation would be designated by proportion that is allocated or above, even though possibly some (dimension?) fields might be initialized in the allocated state.

Code example 4: Sequential conditions

```
block:
    message := GetNextMessage()

    if (!CheckHeader(message)):
        -> MissingHeader
    /
    header := message.header.Parse()
    if (header.format != SuitableProtocol):
        -> FormatUnsuitable
    /
    encodingConvertor := message.encoding.GetConverter()
    if (!encodingConvertor.Compatible()):
        -> FormatUnsuitable
    /

    messageLength := header.size          ; Reuse header variable.

    <More checks and processing...>
    -> MessageProcessed

|== MissingHeader:
    outcomeCounter(outcomes.MissingHeader).Increment()
|== *.FormatUnsuitable:
    outcomeCounter(outcomes.FormatUnsuitable).Increment()
|== MessageProcessed:
    outcomeCounter(outcomes.MessageProcessed).Increment()
/
```

Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it!

Code example 5: Nested conditionals

```
block:
==(ItemFound, ItemNotFound)
    loop item through list_of_items:
    ...
```

Example: `GraphSubgraph::setElementSelected`. Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it!

Code example 6: Iterator with complications

```
block:
    iterator := ...
    loop:
        if (skipThis(item)):
            -> ItemToSkip
        /
        <Process item...>
    |== ItemToSkip:                       ; Empty handler.
    /
/
```

Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it!

13. A Language of the Future?

Utopia27!

Our choice of syntax.

We do not promise simplicity, or even brevity. Rather, we strive for clarity. And to avoid:

- Trivial contract checks;
- Contrived indications of intent (e.g. C++ pointers vs references);

We do not even hope for:

- Near optimal balance of brevity with program checking;
- Comprehensive contracts;
- Comprehensive safety (at least without penalties);
- Magic, namely promises of optimal compilation for simple code.

Returns: We take a “fill-in” design in our examples. This is not the most familiar. It is likely that Utopia27 would employ the more established syntax. However, we have so far found it easier to trace type state with the fill-in method.

Most of the early variations on situations require that the situations in a block be listed at the beginning. Each situation appears at least three times: in the list at the beginning, where the situation arises in the body and then at the end where the situation is dealt with. It seems however, that there is a strong utility placed on minimizing code and avoiding repetition. Thus we have dropped the list of situations at the beginning of the block. The main consequence of this is that the set of situations must be extracted by reading the labels, most easily as they are dealt with. This implies a stylistic preference for keeping the handlers short so that they can be viewed together. This is probably good style anyway

14. In More Depth

14.1. **Slicing and dicing.** There may be some benefit from making the distinction between “one way of doing things” versus “lots of ways of doing things”, sometimes with a middle ground of “one idiomatic way of doing something”. However, I think this potentially unhelpful. Furthermore, while in mathematics or physics *Grand Unification* might be desirable and helpful (that is, finding unifying patterns or theories), it is often not in programming. While it may be true that one iterator might be able generate the results of all

Code example 7: Worked example

```
bool KGVSimplePrintingCommand::print()
{
  <Initial setup...>
  <declare painter, collatedLoops, loopsPerPage,
           pagesToPrint, fromPage>

  if (dlg->exec() != QDialog::Accepted) {
    return true;
  }

  if (!painter.begin(&printer))
  {
    return false;
  }

  <initialize collatedLoops, loopsPerPage, pagesToPrint,
              fromPage>

  bool firstPage = true;
  for (uint copy = 0; copy < collatedLoops; copy++)
  {
    uint pageNumber = fromPage;
    QList<int>::ConstIterator pagesIt = pagesToPrint.constBegin();
    for(;(int)pageNumber == fromPage || !m_previewEngine->eof(); ++pageNumber)
    {
      if (pagesIt == pagesToPrint.constEnd()) //no more pages to print
        break;
      if ((int)pageNumber < *pagesIt)
      { //skip pages without printing (needed for computation)
        m_previewEngine->paintPage(pageNumber, painter, false);
        continue;
      }
      if (*pagesIt < (int)pageNumber) { //sanity
        ++pagesIt;
        continue;
      }
      for (uint onePageCounter = 0; onePageCounter < loopsPerPage; onePageCounter++) {
        if (!firstPage)
          printer.newPage();
        else
          firstPage = false;
        m_previewEngine->paintPage(pageNumber, painter);
      }
      ++pagesIt;
    }
  }

  if (!painter.end()) {
    return false;
  }

  return true;
}
```

Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it! Describe it!

others, there is a difference in intention (mechanistic), communication (design) and execution between "do this 5 times", "loop i through 0 to 9", "loop through the elements of array x", and "apply this iterator that is not done until it is done".

Asynchronous nodes in a graph: send off and wait for responses, genuinely parallel work (needing threads, or whatever), a for-each.

14.2. **Slicing and dicing.** At this point we will consider some questions of code organization. This might seem like a diversion, but we think that situations, along with richer states and contracts, can improve how we divide, arrange and present code. As we approach this topic, our main focus will be on the use of functions to divide code up into manageable pieces. We will also pay some attention to the presentation of code, which involves not only the content but also the ordering of code. There are many other issues, such as the design of APIs or of classes, but we will only touch upon these.

The best presentation of code depends on its intended use and how it is to be maintained.

XXXXXXXXXX

The problem is that, to add a new field to the structure it proved necessary to create a new constructor just for my small edge case. The usual solution is "have named parameters with defaults", but I think this unsatisfactory. I had just been using Java with builder idioms, and my thought was that we are conflating different kinds of data structures. There is a separate genre of structure for configuration. Some fields may be defined, others not. Fields may be added. What is considered valid may be changed, normally by adding new valid fields and combinations. Each field may be initialized or not, and the builder needs to know this, rather than set defaults. A builder constructs a structure of a different type from this type. (Although there could be an almost 1-1 match.)

In UTOPIA27 a configuration data structure has only pre-defined constructors and setters. Valid combinations of field definitions and values are established via typestate.

Insight: An enum automatically has value typestates defined. Thus an enum in a configuration structure can be coupled in the typestate definition with requirements such as which fields must be explicitly defined.

YYYYYYYYYY

Notes: Slicing and dicing: moving code. Current state: limited control flows mean that use of functions is forced to avoid duplication, whereas in other cases use of functions is made awkward through lack of contracts and their enforcement.

We will now argue that

The task of organizing a program is often framed in terms of the division of code into functions. The main criterion is the size of functions, and it is said that splitting longer functions into smaller ones makes the code more readable and maintainable. While this can be applied too simplistically, such as by limiting the number of lines of code, it can also be used effectively

This has not always been the case. One reason Knuth developed the concept of *literate programming* was to present code clearly when functions were not being used extensively.

The extent to which division into functions can be used to present code in the most desirable order is often limited by the language.

14.3. **Local control flow.** Exceptions as exceptional. The following is not novel. Perusing comments in online forums makes it clear that some if not many perceive there to be a problem with using exceptions for routine situations. This is not a question of use of words. It is sometimes argued that it is hypocritical to assess (judge?) some uses of programming languages as mistaken when the alternative is just as bad but hidden in cleverness.

Finally and defer? Is Java's `finally` really a good mechanism for this?

Uses not considered, because orthogonal, or whatever. Low-level improvements. While it might be dreamed that a high-level language should also be able to express low-level optimizations, we consider that to be the realm of extended language features. It should not determine the core of the language.

Layered object states. Ability to initialize in stages, eg for complex objects. Or different initialization. Static-dynamic typing using situations. Ability to split up functions appropriately: not bound by inability to interrupt initialization in the middle. Lightweight contracts facilitate free code organization.

Appendix A. In Style

c:linuxstyle

A.1. **Linux style guide.** The following is an extended quotation from the long-standing Linux kernel coding style guide [11]. LinuxStyle42

```
int fun(int a)
{
        int result = 0;
        char *buffer;

        buffer = kmalloc(SIZE, GFP_KERNEL);
        if (!buffer)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                        ...
                }
                result = 1;
                goto out_buffer;
        }
        ...
out_buffer:
        kfree(buffer);
        return result;
}
```

A common type of bug to be aware of is “one err bugs” which look like this:

```
err:
        kfree(foo->bar);
        kfree(foo);
        return ret;
```

The bug in this code is that on some exit paths `foo` is NULL. Normally the fix for this is to split it up into two error labels “*err_bar:*” and “*err_foo:*”.

Appendix B. Implementation

B.1. **Enumerations.**

B.2. **Boxing clever.** Boxing. Sentinel values. Optional types.

B.3. **Situations and functions.**

References

ACDict 1. *American College Dictionary*, Random House, cited in KnuthDE75 [6].

ClintM72 2. M. Clint and C. A. Hoare, *Program proving: Jumps and functions*, Acta Informatica **1** (1972), no. 3, 214–224.

CoblenzM16 3. Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull, *Exploring language support for immutability*, Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, 2016, pp. 736–747.

DijkstraEW72 4. E.W. Dijkstra, *Notes on structured programming*, In Structured Programming (Dalai, Dijkstra, and Hoare, eds.), Academic Press, 1972, cited in KnuthDE75 [6].

GordonCS12 5. Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy, *Uniqueness and reference immutability for safe parallelism*, Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, ACM, 2012, pp. 21–40.

KnuthDE75 6. D.E. Knuth, *Ill-chosen use of "event"*, Communications of the ACM (ACM Forum) **18** (1975), no. 6, 360.

KnuthDE74 7. Donald E. Knuth, *Structured programming with go to statements*, ACM Comput. Surv. **6** (1974), no. 4, 261–301, also cited in KnuthDE75 [6].

KnuthDE92Book 8. ______, *Literate programming*, CSLI lecture notes, no. 27, Center for the Study of Language and Information, Stanford, CA, USA, 1992.

KnuthDE92 9. ______, *Structured programming with go to statements*, Literate Programming, CSLI lecture notes, no. 27, Center for the Study of Language and Information, Stanford, CA, USA, 1992.

LandinPJ65 10. P. J. Landin, *Correspondence between ALGOL 60 and Church's Lambda-notation: Part I*, Commun. ACM **8** (1965), no. 2, 89–101.

LinuxStyle42 11. Linus Torvalds, Dan Carpenter, et al., *Linux kernel coding style*, ch. 7: Centralized exiting of functions, Linux Kernel, December 2014, https://www.kernel.org/doc/Documentation/CodingStyle.

WilliamsDO84 12. David O. Williams, *Structured transfer of control*, SIGPLAN Notices **19** (1984), no. 10, 46–51.

ZahnCT74 13. C.T. Zahn, *A control statement for natural top-down structured programming*, Proc. of a Programming Syrup (Paris), Lecture Notes in Computer Scienceq, vol. 19, Springer-Verlag, April 1974, cited in KnuthDE75 [6].