IS IT TIME THAT SITUATIONS STUCK?
AN EXPLORATION OF SITUATIONS AND TYPESTATE

J. ALEX STARK

ABSTRACT. We bring together some ideas, old and new, and discuss how the combination might be useful for future production programming languages. Of special interest are constructs that have been given the name "situations", which were developed over forty years ago, but were not adopted into the mainstream. The newer ideas come from lessons learned with popular production languages, along with lessons learned in pragmatic research. Most of these concern the state, or "typestate" of an object and how it can be traced through the execution of a program. We pay particular attention to states of initialization and validity, and to ownership and reachability.

CONTENTS

Date: 2016.

Code Examples

1. A good time

While there have been interesting developments in computer languages over the whole history of computing, I think that we are in a particularly promising period. I realize that it might be argued that popular languages such as C++, Java and Python have not (at least lately) been bringing significant improvements. Nonetheless, at the same time pragmatic research, combined with an established body of experience, has provided the basis for language projects such as *Go* and *Rust*. Personally, I like the approach taken by these projects: they show pragmatism, along with a willingness to rethink established practice and to reconsider older ideas, even those that were never fashionable.

1.1. **General aims.** The purpose of this article is to consider some ideas that might be beneficially incorporated into these endeavours. The ideas are not new, and in fact a few are rather old. We will touch on various concepts, but will focus much of our discussion around three broader themes. Let us define these briefly now.

First, we will be especially concerned with *typestate*. This was included as part of Rust in its earlier versions, but later dropped. We will look at how we might make it work. We define typestate 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. For example, it helps to determine, at compile time, whether or not some piece of data is valid. It helps in tracing ownership. I think it might also be used to check that array indices are within range their ranges.

Second, we will consider a really old idea (from over forty years ago). This, the idea of *situations*, which we will define in detail later, will recur as a theme throughout our discussion. The name *situation* was used by

Code example 1: Continuing to the next iteration.

```
loop item through list_of_items:
   if (skipThis(item)):
      -> ItemNotRelevant
   /
   <Process item...>
|== ItemNotRelevant:                    ; Empty handler.
/
```

In traditional languages the `continue` keyword is used to skip remaining processing of the current iteration and proceed to the next. Situations support labelled skips instead. In this case the execution moves to the `ItemNotRelevant` label if the item has a particular property. No handling code is executed, and so the code continues directly to the next item in the list. The code could have multiple situations with multiple different labels.

Knuth in describing a restricted branching control structure. Knuth was not the only one to propose it, and there were other ideas 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*. We will refine this idea for our own purposes. We will propose that situations should be used to distinguish infrequent states of affairs from those that occur frequently. This is not the distinction between the exceptional and the normal, since an infrequent situation need not be exceptional. We will also couple it, for all but short-range branching and merging, with typestate. That is to say we prefer to use situations when different typestates result. This probably sounds cryptic, but we will take time to explore situations in depth.

The third theme is a natural complement to the first two. Consider the second: situations concern patterns of program execution, and in particular paths of execution with branching and merging. Consider the first: typestate concerns the ways that state information about data changes as such execution unfolds. All this is only meaningful if typestate can be tracked across boundaries such as functions calls. Furthermore, if modules are separately compiled then it is a requirement that there be mechanisms for such tracking. Thus the third theme concerns *contracts*, which for our present purpose comprise typestate *expectations* and *assurances* as execution unfolds. This includes either side of a function call, but one could draw a line through any point in the code and ask, for all possible paths: what does the code on one side set up that is used by the code on the other?

1.2. **Situations.** Situations are quite simple. They are labelled branch 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 1 (in pseudo-code) 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. The programmer has to provide the situation names and arrival points. In contrast with `continue` control flow, this means that the iteration being interrupted is specified. For instance, one can jump out of nested iterations. There can be multiple situations. Also, a piece of documentation is associated with each situation, albeit just a name, though an annotation could be also associated via that label.

Code example 2: Breaking an iteration.

```
block:                                                          1
   loop i through [0:k]:                                        2
      if (finishNow(itemArray[i])):                             3
         -> ProcessingDoneEarly                                 4
      /                                                         5
      <Process item...>                                         6
   /                                                            7
|== ProcessingDoneEarly:                                       8
   <Handle situation...>                                        9
/                                                               10
```

In traditional languages the `break` statement is used quite often to leave an iteration from a place other than the loop termination check. Implemented using situations, the code would be placed within a block structure, which might be a function or a conditional. I believe that, in the great majority of cases, situations could be used to write more elegant code than using the `break` statement. For example, one advantage of situations is that we can easily use common code between multiple departure paths.

1.3. **Breakage.** Uses of the `continue` keyword are relatively few, whereas uses of `break` are more common. (At least according to an informal check I conducted.) An illustration is given in code example 2, in which an array of elements is traversed, and processing is terminated early if a specific reason is found for one of them. 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 typically be an enclosing function, or something like that. (Again according to my informal code survey.) 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 block division in our pseudo-code. In traditional languages a small set of keywords (`else`, `case`, and so on) indicate such a break. When situations are used, these are more common. I think that, at least in pseudo-code, this symbol helps to show the code structure. Furthermore, the *only* not-usual way that execution comes to the end of control structure is via a situation. (Exceptional events could cause an early exit.) 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.

1.4. **Imagining Utopia.** To what sort of language should we target in these discussions? What kinds of applications? When considering such questions, we need to be realistic about our own knowledge and understanding. 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. Given this, and the fact that no language is even nearly optimal for every task, we should hesitate to make authoritative statements about what makes a good language. Nonetheless, it is important to aim for a fairly specific target.

As we proceed we will 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,

Code example 3: Definition and use.

```
block:                                  ; Just to establish scope.
   int x := 0
   <...>
   if (y > 0):
      x := y
   | else
      x := -y
   /
   <Code that, somewhere, uses x...>
/
```

A simple example illustrating definition and use of a variable. The variable `x` is defined in two places, guaranteeing that it is defined for all execution paths. Therefore the initialization to zero where `x` is declared is unnecessary. (The whole of the example is enclosed in a code block to clarify the maximum lifetime of `x`.)

Python, shell scripting, SQL, and so on, are better suited. It should work quite well for a web server, for image processing, for a numerical simulation toolkit. 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 humour and obvious lack of realism in this name will help us avoid taking ourselves too seriously.

This article employs a pseudo-syntax, 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.

1.5. **Rambling along.** For my previous attempt at writing about situations and typestate I tried to organize my thoughts into coherent themes and arguments. I found it too big a big task to explore all the issues. Even if it were a modest task, the result would be boring. So instead, I will try to address some real issues that resonate with me, in the hope that the result will resonate with some readers. While doing so, I will try to address these issues thoughtfully and in some depth.

2. Basics

2.1. **Dealing with invalid data.** We still have a lot of 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 scenarios that should never happen and often even cannot. I confess that this frustrates me. 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. It is worth, I think, looking into the basics of this in detail.

2.2. **Avoiding the invalid.** It was not that long ago that many compilers did not (thoroughly) consider def-use (definition-use) patterns. Specifically, the compiler would not tell you if you used a variable without having previously defined it. Many style guides would recommend defining the variable at declaration. Thus code would look like that in example 3. There is a completely unnecessary initialization, and when `x` is set in the conditional statement, what should be a defining assignment is now a mutation. Now that compilers have improved, this (pseudo) code looks a little strange. In fact, initializing `x` at declaration is actually a bad idea, because the compiler will no longer tell you if you forget to define it properly.

If we think that the languages and tools of today are fundamentally better, we need to look harder. The reality is that C++ basically does the equivalent of initializing `x` to zero for any local object. This is an integral part of RAII: initializing all objects when locally allocated. Example 3 illustrates one of the problems with RAII, namely that we might want to initialize an object in different ways. Another is that we may not be able to initialize it all at once. Programmers experience these problems clearly when a variable should be `const` but cannot, because it cannot be initialized when declared. It might be countered that these are the minority of cases. However, they are not a tiny minority. (We will consider later in this article how we can deal effectively with minority scenarios.) If data is initialized differently along different execution paths, including not initialized at all, then it may be better to use situations to manage the differing typestate rather than to merge the execution paths and create ambiguities.

Let us return to the null pointer example. Many languages alleviate the problems of invalid pointers through RAII, automatic memory management or runtime checks. While containing the immediate damage, these do not solve the problem. My experience is that exceptions are all too common, and the handling of those exceptions is all too often unhelpful and generic. We still need better handling of the validity of objects, of references to and between objects, and of indexed data access. In all mainstream languages using an out-of-bounds array access is possible and the consequences are serious.

In this 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 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. While in this case the variable is defined along every conditional branch, some branches might not. The resulting typestate at the end of each of the branches will differ. In UTOPIA27 the preferred way of dealing with this is to have some of branches use situations to diverge the subsequent execution.

2.3. **Initial definitions.**

Definition 1 (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 information available during compilation that helps in the analysis of how the state of data unfolds as a program executes.

Definition 2 (Situations)**.** Situations are labelled branches that move downward in written code and outward in scope. They are associated with circumstances occurring at a particular location in a program execution. A circumstance is a state of affairs involving program variables, reflected either in their values or in their state.

Definition 3 (Contracts)**.** Contracts, or more specifically typestate contracts, are implicit or explicit agreements about typestate across boundaries in code. The boundaries can be around a function, around a block

of code, or between two lines of code. The contract specifies what the preceding code *assures* and what the following code *expects*. Boundaries around blocks or functions have entry and exit requirements. When multiple execution paths pass through a boundary, such as with situations, the contracts can be different for each.

2.4. **Neutrality in code organization.** There is some danger of this article reading like a complaint with a long list of challenges faced by today's programmers. Please stick around: we are moving in the direction of proposing solutions. Actually, I hope that situations are straightforward enough that one can read descriptions of the problems and at the same time be thinking how situations might help. Situations help to clarify alternative execution paths. For example, many procedures begin by reading data, configuring, allocating and initializing. Situations provide a simple way of separating the various outcomes of such tasks. Later we will look specifically at questions such as how we should unwind partially-allocated data structures. We will also consider how some uses of the Go language's `defer` can be implemented as situations. For the present, let us consider an embarrassingly basic task, namely that of splitting a procedure into smaller parts. This may appear too basic, but simple examples can help to clarify fundamental issues. Moreover, if we cannot get the simple things right, there is not much hope for solving complex things. Later we will consider other ways in which situations can bring major improvements to code organization.

Splitting up a procedure is important task as it underpins the freedom to use the best design. Moreover, organizations and projects have their own styles and rules as to how and when a procedure should be broken up. However, the real criteria behind decisions on code organization are often different from these and far from ideal. Two obvious problems are testing and documentation. One doesn't need much programming experience to know the feeling. "I'd prefer to break this out into a separate function, but then I would have to document (and maintain documentation) in two places, and I would have to write tests, …" Testing can become really problematic. If we pass a pointer (or any data that may be invalid), we have probably already checked that it is valid. When broken out into a function there is no need to check this. But what happens if that function is reused elsewhere?

The decision to split a section of code into a sub-procedure should be *neutral*. That is to say a language like UTOPIA27 should not bias a programmer's choices. Of course, we want other code organization decisions to be neutral as well, but we can more easily judge success on this narrower scenario. When code is split off into sub-procedures they are used once, and all the code remains in the same compilation unit (and probably private to it as well). As mentioned above, two obvious potential biases are documentation and testing. Documentation should be handled relatively easily. The names of the parameters of a sub-procedure will be the same as the variables used in passing arguments. Thus variable annotations in the caller could be used as parameter annotations in the callee function, and *vice versa*. Or some other scheme could be provided, appropriate to the expected standard for documentation. Thus the amount of initial work and the ongoing maintenance should be much the same whether a section of code is directly in a procedure or put in a sub-procedure.

Contracts can either support or conflict with neutrality of code division. That is to say, while contracts should deal with the problem of validity, thus simplifying testing, their design could make it harder to separate a section of code into a sub-procedure. If the programmer has to state expectations and assurances in detail, the process will be burdensome. To minimize this burden, we can borrow from the treatment of documentation. At the point at which the sub-procedure is called, we can gather all the constraints that we know are satisfied about the variables used as arguments. We can then examine each term and see if it is required by the sub-procedure. It might, of course, be possible to gather the expectations of the sub-procedure from the code within, though I suspect there may be some difficulty in merging them cleanly. This will require some practical exploration.

3. Situations

Let's look at situations in more depth. What meaning should typically be attached to them? How should they be used? Where and when should they be used?

3.1. **A state of affairs not an event.** 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 (or at least not further in) 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 tends to lead to clarity, 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 [7]. Even more enlightening is the note [6] that Knuth and Zahn wrote as a follow-up to clarify a point. 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 [13] has been described [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 [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 [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 [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

I suggest we embrace this almost in entirety. As noted above, I think that situations should be implemented without the need to declare all new situations at the beginning of a block. We have already tried to codify this in definition 2 above.

3.2. **Returning situations.** Until now we have largely assumed that situations must be handled within a function. However, in section 2.4 we suggested that the decision to break out a block or section of code into a separate sub-procedure should be as neutral as possible. What if such a section raises a situation that is handled in an enclosing scope? If we separate the function out, it will have two return paths: the common exit and the exit initiated by the situation. Thus we propose that functions in UTOPIA27 may have alternative return paths

A familiar and simple example is that of opening a file. When we call a library function to do this, it may be successful, but for a variety of reasons it may not. In the C programming world (with `fopen()`), the possibility of not succeeding is handled first by checking the return value. A null `FILE` pointer is returned if the file is not opened, and the global variable `errno` is set to the reason. Other languages are have neater arrangements than this, using either exceptions or return values. In UTOPIA27 the approach is for the function interface to return a situation. The caller requesting opening of the file handles the situation that the opening was unsuccessful.

Example 4 shows how this might work in practice. We assume that there is an existing library function that uses return values to indicate success or failure when parsing a string into a pair of integers. (Of course it may be best to have a library routine that returns a situation.) Then `ParsePoint()` looks at these return values and either returns a failure or constructs a new point.

3.3. **Unexceptional but mostly infrequent.** When I have read debates about return values versus exceptions, the sort of threads that sometimes emerge from discussions of the Go language, I have had situations in mind. In light of the contrast between situations and events made by Knuth and Zahn, I have come to think that such debates often mistakenly conflate events and situations. Moreover, I have come to think that we, more widely in computing, confuse what is exceptional with what is infrequent. The example of opening a file illustrates this very clearly. We use the word *exceptional* so often we will readily describe the failure to open a file as exceptional. But really it is a rather ordinary scenario. If a user gives an incorrect name

Code example 4: Returning a situation from a function.

Original in pseudo-code:

```
bool parsePoint(String input_string, QPoint& point_ref)
   int x, y
   res := GenericParse(input_string, &x, &y)
   if (!res):
      return false
   /
   point_ref := QPoint(x, y)
   return true
/
```

In UTOPIA27 pseudo-code:

```
function ParsePoint(input_string String):
|--> output_point QPoint
|++> Fail assures output_point @ uninit

   ;; Reuse an existing library routine, or a generic routine.
   (x, y, success) := ParseIntegerPair(input_string)
   if (!success):
      -> Fail
   /
   output_point.InitializeFromPair(x, y)
/
```

A sketch of a reworking of the function `parse_point` from `dotgrammar.cpp`. This scans an input string for two integers and constructs a new point from them. On the frequent return path it is implicit that the output point, the return value, is initialized. In the original version, the result is written into existing data `point_ref`. The way that we have written it, the UTOPIA27 pseudo-code assumes that we have a library function that parses a string into a pair of integers and uses return values rather than situations. In reality the code would use whatever function is available: this is just an example.

for a configuration file, we should deal with the situation as a matter of ordinary process. The meaning of this English word is broad enough that one can in common usage say that such any infrequent situation is exceptional, it is merely "an exception to the rule". However, in computing the word has become associated with the need for extra special handling.

Having considered many examples, I think that situations in UTOPIA27 should in most uses differentiate *infrequent* code paths from *frequent* paths. They should not be used to differentiate the *exceptional*, or *abnormal*, 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 require expensive handling, but by an inexpensive diversion. Situations and typestate enable us to deal cleanly with the different data states that arise when execution unfolds like this.

With a little hesitation, here are some suggestions on the use of situations. I think the uses should vary according to the *reach* of the situations, that is the distance (loosely interpreted) between the departure and arrival points.

(1) Situations relate to states of affairs and their handling along associated execution paths rather than to events and occurrences.
(2) Situations are not well-suited to (very) long-reach execution changes, such as are often handled by last-resort exceptions with stack traces.
(3) Situations are well-suited to medium-reach uses.
    (a) Medium-reach situations should be used for infrequent execution paths, that is those expected to be executed much less than 50% of the time. They allow for alternative execution paths that can be quite ordinary yet infrequent.
    (b) Medium-reach situations are best used when there are typestate differences. For example, the primary return values may not be initialized along an alternative execution path. Likewise, explanatory data such as a reason code may not be defined along the frequent path.
    (c) Situations in UTOPIA27 do not have a data payload (parameters). Return values can be used to pass data in medium-reach uses, via return tuples if appropriate.
(4) Situations are useful for short-reach control flow, such as in examples 1 and 2.
    (a) In short-reach usage, a situation would not necessarily be associated with different typestate or an infrequent execution path.
    (b) Nevertheless, data may well have different typestate where there is a situation.
    (c) Also, while an execution path might not be *infrequent*, it is preferable that it is *less frequent*. This gives a hint to the reader and the compiler about branching probabilities.
    (d) Situations in UTOPIA27 do not have a data payload (parameters). Local variables can be used to pass data in short-reach uses.

A notable benefit of situations is that default and sentinel values are not needed so often. A function can return its primary results and the data that explains any failures as separate parts of the return tuple. The compiler knows from the typestate which parts are initialized on each of the execution paths. There is no need to set aside special values to indicate what has been intentionally initialized.

3.4. **Technical details of situations.** Technically functions with situations could be implemented in (at least) two different ways. One is to use return values, and the compiler would insert extra return values and test the results to see if an infrequent situation had occurred. Another is to pass an alternative return address along with a pointer to a situation enum. If a situation occurs in the function, it would substitute the alternative return address for the usual return address. It would also change the situation enum. The function knows nothing about the context in which it is called. However, in the calling code this may not be the only function that is called that returns situations. Each of these functions would be called with different base enum values.

Situations generally span between points in code.

(1) A condition that leads to the alternative execution path, either an `if` condition or an opposite `else` condition.
(2) The departure point, where the code says to go to the handler of the situation.
(3) The arrival point. Any handling code begins here.
(4) The end of the handling code that is specific to that situation. As we will discuss later, this can continue on and merge into another handler.
(5) The final end of the handling code.

We consider the whole of that execution path to be associated with the situation. Most importantly, the situation named at the departure point effectively labels the code that precedes it. All code from the condition to the departure point, the whole of that path, along with the condition that establishes the circumstances, is associated with the situation.

4. Control flow and return values

4.1. **Some context.** *(This section will need to be revised.)* As I mentioned at the beginning of this article, I think that a lot of interesting work has been done over the last few years, embodied both in pragmatic research articles and in new languages such as Go and Rust. Quite likely we will see the next generation replacement for C++ emerge from this work. Research articles such as [5] and [3] consider questions such as how we can communicate knowledge about data and data references without placing burdens on programmers. Questions about ownership, mutability and reachability are being explored. Particular attention is being paid to what a program needs to specify, and how those specifications can be minimized.

The general concept of Utopia27 is that it targets many of the kinds of tasks for which C++ and Java are considered well-suited. While hypothetical, I prefer to imagine Utopia27 in concrete terms. It is intended to be simpler. While arguments can be made for rich template support and rich injection frameworks, the final results of incrementally good intentions can be large execution size and degradation of performance.

Perhaps more importantly, there is a great danger in programming language design in making programming easier for the initial programmers at the expense of guest programmers coming in to add modifications later. That is why we emphasize code exploration tools so heavily. Large corporate projects and most substantial open source projects have core teams with ownership who review the contributions of many other programmers. How does a project "welcome contributions from other teams", or encourage new members to join? They need to be able to explore and understand just enough of the code to get general context and specific details. And so on. This is surely obvious? However, my experience is that languages along with programming "paradigms" and "frameworks" often encourage design choices that make guest programming difficult. Guest programmers take more time, cannot readily write contextually appropriate code, core developers take more time reviewing, and potential new core developers are discouraged.

Another feature of Utopia27 that we will note now and discuss at greater length later is that it does not intend to prove much. Checking of typestate rules and contracts is simple. Compilation needs to be fast. Programmers need to know what the compiler can check. With this in mind, Utopia27 has what might be called *managed unsafety*. If the compiler cannot check something, then it has to be explicitly asserted via an assurance statement. My claim is that these explicit assurances will be relatively few. They would be the bits of code given extra attention in documentation in tests. If we are not doing this already in unsafe languages, we should be.

While much of this article discusses how basic language features might improve productivity, we should always have in mind improvements to security. I believe that simple aspects of computing can have significant impact. It is easy to dismiss a syntax proposal as *bike-shedding*, but I think this misguided. If we take a careful and sensible approach, we can achieve better reliability and security even in large collaborative projects.

4.2. **The return value initialization concept.** Situations can replace the `break` and `continue` keywords. I would like to propose that Utopia27 eliminate the `return` statement as well, even when a value or tuple of values are returned. There are two aspects of this: (a) specifying the points from which the function can return from the perspective of the program counter and stack, and (b) specification of return values. Looking at the examples of early returns, I think that the majority of uses would be clearer if they were associated with named situations, thereby adding self-documentation. Overall, the benefit of labelling and annotating outweighs the modest effort. Also, requiring the use of situations means that functions have explicit lists of exit reasons. One can work back from the arrival points, exploring the code more readily.

In example 4 we employed an up-front declaration scheme. That is we declare the return values at the beginning of the function. This is not at all novel. Languages have done this, and had bare `return` statements. Some of the advantages are as follows.

(1) It simplifies typestate and contracts. The assurances that a function makes about output data it returns can depend on the expectations it makes about input data. Even if this were not the case, it would be awkward to specify contractual assurances anywhere but the beginning of a function.
(2) It unifies some function types. Constructors, initializers and named constructors are essentially much the same. The differences lie in where the allocation is done, and whether or not the *self* data object is being initialized.

4.3. **Traditional control flow: limitations.** 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. (These points are not original.)

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 [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 7.2

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 one.

5. Typestate in more detail

5.1. **Code communication.** Code development is in many ways an exercise in communication. Therefore clarity is important. However, brevity is not necessarily helpful. In this section we will explore how Utopia27 can be concise in common cases and also how we can ensure that verbosity increases only gradually with infrequency. Utopia27 tries to be a little like a Huffman code. We want pieces of code to convey meaning clearly and simply, and we want code in context to avoid clutter and awkwardness.

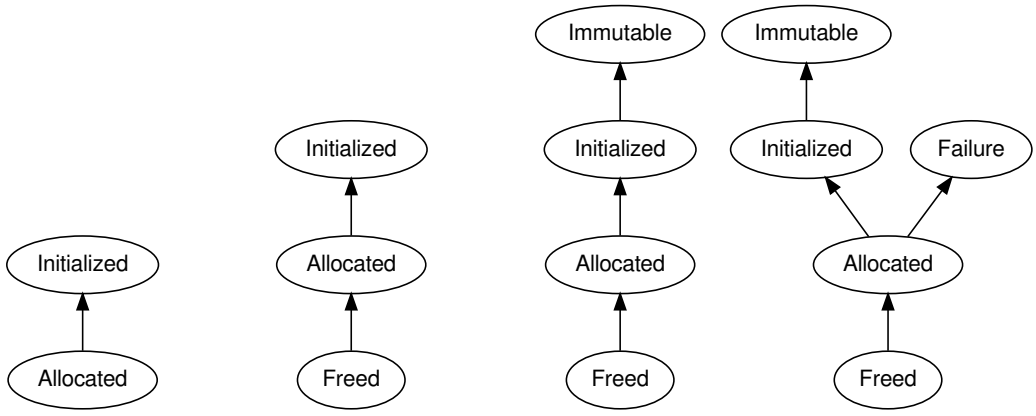

FIGURE 1. Simple typestate. (a) Variables, whether simple or compound, are generally allocated within their scope and so only need to be initialized. (b) Basic mutable reference variables have three states: freed (or unallocated), allocated and initialized (or set). Languages such as C++ try to hide the allocated state so that data is apparently either freed or initialized. Value members do not have the freed state. (c) Constant data has an additional immutable state. Data is in that typestate between initialization and the beginning of destruction. (d) Most modifications of this would be in the form of branching; UTOPIA27 makes it very easy to add these.

Communication involves more than this. The arrangement and organization of code is important I am mindful of the experience described by one programmer. Code that was implemented as a long switch statement (in C) was later implemented using dynamic dispatch in C++. The original switch statement was not elegant, and one might expect the new code to have been an unqualified improvement. However, what was in one place in the original code was scattered all over the place in the new. This is not to say that abstract classes are wrong, rather to point out that there can be disadvantages that are not obvious.

5.2. **Straightforward typestate.** Typestate, as we defined initially in definition 1, 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.

Fig. 1 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 has an extra state, since at first does not point to allocated data. Constant data also has an extra *immutable* state. Languages that emphasize RAII, such as C++, tend to hide or suppress some of these states, but they are nonetheless there as data is allocated and initialized.

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. From the point of view of these simple states, the main duty of the typestate of a compound is to specify patterns of typestates of its members. We shall explore some more elaborate patterns in section 7.

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.

5.3. **Branches and ordering.** Side branches, such as failure branches (fig. 1(d)) 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. Therefore we need only mark fields that are to be initialized in the failure 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 need to be marked for both.

Edges in the graphs of typestate have direction, indicating a sense of completeness. These were shown in fig. 1. While most graphs are in practice trees, more general directed acyclic graphs are permissible, albeit with more verbose definitions. For every node in a graph one can find all the nodes that are backwards or forwards in terms of the edges. That is, there are nodes that can be reach by only going backwards, those that can be reached only going forwards, and some that cannot be reached with one type of move. For any graph one can order the nodes in a sequence that can test these relationships. Pick any two nodes. If B can be reached from A by only forward moves, then we can say that $B > A$. This is known as a *topological ordering* of the graph, and it can be constructed in linear time.

The importance of this ordering comes in the fact that we can write statements about typestate using inequalities. When we say that an object is initialized, we normally mean that it can be in any state "greater than or equal to initialized", such as the immutable state. The above analysis also highlights the fact that relationships between states on different branches are undefined. At least for trees we could resolve this by having two orderings. Construct one sequence one by assigning higher sequence indices to left branches, and another by assigning higher indices to right branches. Then we can say that $B > A$ only if that is true in both orderings.

(Aside: I think that two sequences are insufficient for DAGs in general. However the main reason for allowing typestate graphs to be more general than trees is really only to allow the possibility of merging two branches. For example, one might initialize sub-units of a data structure in more than one order. It may be reasonable for UTOPIA27 to restrict typestate graphs to those that can be checked in this way. I think that I have a straightforward algorithm for creating two suitable topological sequences for a graph that can be drawn without crossings. Graphs for which this can be done are known as *planar* graphs. There are $O(n)$ algorithms for *planarity testing*. One of them, the *Boyer-Myrvold algorithm* can be also used to create a *planar embedding*. See, for example, the Boost planar graphs library. There are non-planar graphs for which two sequences exist. Consider the utility graph with nodes (1, 2, 3) connected to nodes (4, 5, 6) and direct from the first to the second. Suitable sequences in this case are (1, 2, 3, 4, 5, 6) and (3, 2, 1, 6, 5, 4).)

5.4. **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 with this 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 7.1

Code example 5: A simple array function.

```
function SumElements(input_array []int32):
|--> total int64
   ;; The default contract is that input_array is initialized on entry and
   ;; total is initialized on exit.
   total = 0
   for i through [0..len(input_array)]:
      total += input_array[i]
/  /
```

A simple example of a loop in which the safety of an array access is readily guaranteed. UTOPIA27 does not need to perform checks on array access. Observe that no contract terms need be explicitly stated, since the defaults apply. (The accumulator variable `total` is not initialized by default. In some cases the compiler can elide zero initialization.)

The need for UTOPIA27 to handle between-states is another reason why edges must be 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 subtlety 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.

6. Rules for typestate and contracts

6.1. **No proofs.** *This section is an initial sketch, and will be developed further.*

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 good reasons for this.

(1) Anything but simple deductions will slow down compilation.
(2) If the rules are simple, a programmer can anticipate what UTOPIA27 can figure out and what needs to be specified explicitly.
(3) A lot can be done without being clever.
(4) Writing any kind of real proof engine is difficult, will not cover important cases, and will be prone to bugs, including security bugs.

With this in mind, UTOPIA27 adopts a kind of *managed unsafety*. If it cannot figure out that something is safe, explicit assurance must be given in the code by the programmer. These are not going to be needed most of

the time. After all, what frustrates programmers is the fact that in most cases the safety is obvious. It would be good to focus thought and testing.

- Null pointer checks, or equivalents in languages without null pointers, are generally not required. We typically know that a reference is to an allocated and initialized data structure. UTOPIA27 handles this by knowing simple things about the typestate of data, and by having contracts with sensible defaults.
- Loops with induction variables used in indexed access, and iterator loops, are almost always obviously safe. UTOPIA27 handles this by expressing simple typestate facts that are assured within such control blocks.

Some of these features are illustrated in example 5. There are nonetheless facts that are harder to establish and so UTOPIA27 requires them to be assured explicitly by the program. By their nature these are things that we should be covering more heavily in tests and documenting more carefully.

6.2. **Basic ILP solution.** *This section is an initial sketch, and will be developed further.*

The main engine used by UTOPIA27 to handle constraints is an integer linear programming (ILP) solver. The ILP problems are of modest size, and so confirming the existence of solutions is quite fast. In example 5 the ILP for the contract expected by the array access is

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \texttt{i} \\ \texttt{len(input\_array)} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1}$$

(We are assuming an element-wise inequality.) This is also the contract assured by the `for` loop.

However, what is obvious written out for a human might not be obvious to UTOPIA27. All constraints handled by UTOPIA27 are expressed as logical sums of products of terms. The terms might be themselves expressed as sums of products, and so every set of constraints can be expanded into single sum of products. At the point in the code of example 5 where the array is accessed a set of assurances have been set up. For example, the input array and accumulator variable have been initialized. The induction variable `i` has been initialized and its value constrained as described above. Let us focus just on the value for the moment, and disregard the states of the data. The array access expects that `i` be within bounds. All UTOPIA27 need do is check for solutions in violation of this.

In general, we want to know if there is any solution, in terms of integer variables, enumerations, or functions of data objects, that lies within the assured constraints and without the required expectations. Such a solution would violate the expectations of the contract. I think it worth working through this with a little formality. Consider the expression

$$\begin{aligned}(A_1A_2 + B_1B_2 + C_1C_2)&\overline{(M_DD_1D_2 + M_EE_1E_2 + M_FF_1F_2)} \\ &= (A_1A_2 + B_1B_2 + C_1C_2)(\overline{M_D} + \overline{D_1} + \overline{D_2})(\overline{M_E} + \overline{E_1} + \overline{E_2})(\overline{M_F} + \overline{F_1} + \overline{F_2}),\end{aligned} \tag{2}$$

which must be true for any violating solution. The first half of the left-hand side is a sum of products of assurance constraint terms (such as $i > 0$), which must be true, and the second half Boolean negation of the sum of products of required constraint terms. The number of terms will, of course, depend on the contracts. To look for violations we would expand and examine each product, such as

$$A_1A_2\overline{M_D},\ A_1A_2\overline{D_1},\ A_1A_2\overline{D_2},\ B_1B_2\overline{M_D}, \ldots$$

in turn.

In the case of our simple example above, the negated expectations are

$$\begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \texttt{i} \\ \texttt{len(input\_array)} \end{bmatrix} \geq \begin{bmatrix} 0+1 \\ -1+1 \end{bmatrix}. \tag{3}$$

Therefore we look for solutions to

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \texttt{i} \\ \texttt{len(input\_array)} \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \tag{4}$$

and

$$\begin{bmatrix} 1 & -1 \\ 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \texttt{i} \\ \texttt{len(input\_array)} \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{5}$$

Since neither can be solved, there are no violations. These are readily analysed, since there are no solutions to the corresponding real-valued linear programming problems. Indeed, UTOPIA27 can rule out most constraint violation checks using ordinary linear programming.

The general contract checking task looks to be rather a burden. However, in practice it is not. In many cases there would only be one product in the assurances and one product in the expectations. Let us assume, however, that we have more terms in the assurances established by the various execution paths that lead up to the code point where we are now checking a contract. In other words, we might have more than one product in the assurances summation. Consider the expectations. This is almost always simpler.

(1) Most functions only have a single product in the expectations sum of products. To put it informally, "functions expect a bunch of things to be satisfied". Data has to be sufficiently initialized, has to be reachable, and perhaps has to be mutable. Values are constrained.
(2) It is rather rare for the general case to apply.
(3) There is a middle-ground case that sometimes is required, and this opens up an interesting way for UTOPIA27 to analyse and present contracts. This is where each product has a term that excludes the other products. For example, a function may take an object of some sort of *general* or *any* type. The expectations may then be something like: "If the object is of type X then this and that constraint must apply, or if it is of type Y then this other constraint must apply, or if it is of type Z…".

Let us explore the middle-ground case just a little more. We chose to name some of the terms M_D, M_E, and so on, in the example above because UTOPIA27 can identify such terms quite readily. For example, M_D excludes the other products if

$$M_D(M_E E_1 E_2) \qquad \text{and} \qquad M_D(M_F F_1 F_2)$$

have no solutions. In other words, if M_D is true then $M_E E_1 E_2$ and $M_F F_1 F_2$ must always be false. The practical ramification is that, when checking for violations we can eliminate many possibilities. If we are checking terms involving D_1 we can safely assume that M_D is true, otherwise the truth of D_1 is irrelevant. Hence, we do not need to incorporate any other of the expectation terms into the violation checks.

Of course, searching for exclusive terms is not trivial. However it has the side benefit that UTOPIA27 can present to the programmer some indication of the structure of contractual conditions. Once UTOPIA27 has checked that there are no solutions for $M_D(M_E E_1 E_2)$, it can then check for solutions to $M_D M_E$, $M_D E_1$ and $M_D E_2$, and thereby identify M_D and M_E as mutually exclusive. And this will be the possible in cases like those with a general type as described above, since the object will be either of type X or Y or Z.

Code example 6: An example of a hierarchical enumeration.

```
;; Orange varieties.
enum Oranges:
   WASHINGTON,        ; Washington navel, good for eating.
   VALENCIA,          ; Valencia, good for juicing.
   CARA_CARA,         ; Cara Cara, pink flesh.
   SEVILLE,           ; Seville bitter orange, good for marmalade.
/

;; Mandarin varieties.
enum Mandarins:
   PAGE,              ; Page mandarin, a Minneola tangelo x Clementine cross.
   MINNEOLA,          ; Minneola tangelo, a tangerine x grapefruit cross.
   PIXIE,             ; Pixie mandarin, late maturing.
/

;; Citrus varieties.
enum Citrus:
   ORANGES   Oranges,    ; Orange varieties.
   MANDARINS Mandarins,  ; Mandarin orange varieties.
   CITRON,               ; Not a lemon.
/
```

A hierarchical enum, with `ORANGES.SEVILE`, `MANDARINS.PAGE` and `CITRON` being values within the `Citrus` enumeration. This illustrates how leaf members and sub-enums can be combined. (While we treated Citron in this way, from the taxonomy point of view, it might well be better to place Citron in its own enumeration that contains specific varieties, even if that enumeration only contains one value.

In conclusion, UTOPIA27 needs to do a little extra work to check constraints but, considering the benefits, the work is quite modest. The general cases are not hard to handle, and would rarely impact compilation speed significantly. Moreover, handling contractual expectations well actually could open up means for documenting or otherwise presenting contracts in a way that is helpful.

6.3. **Enumerations.** *This section is an initial sketch, and will be developed further.*

UTOPIA27 has specific support for producer graphs, for finite state machines and for fully featured enumerations. Let us look at enumerations in more detail. UTOPIA27 considers basic entity documentation as almost part of the code, and blurs the line especially in the case of enumerations. Entity documentation as in, say, Doxygen, is documentation associated with functions methods, parameters and so on. Typically a short and a long documentation string can be associated with any named entity. UTOPIA27 treats short documentation strings for members of enumerations as explanatory strings that can be used at run time and for diagnostics during compilation.

Perhaps most importantly, UTOPIA27 makes use of hierarchical enumerations in compact form. Example 6 is an illustration of this. Within the `Citrus` enumeration, UTOPIA27 might reserve values 0 through 3 for `Oranges`, then compactly values 4 through 6 for `Mandarins`, and finally it might assign 7 to `CITRON`. One usage of these hierarchies in UTOPIA27 is for situations. These are identified by enumerations that are generated automatically. If, within a block of code, a sequence of function calls are made to functions that

return situations, they can be handled together. The situations for each function are identified by their own enumerations. The calling code combines these in a new hierarchy. Each of the called functions take a reference to a situation identifier, along with a base value. If they return via their frequent path, they do not mutate the value. If they return a situation, they set it to the sum of the base value and their own situation enum value.

Compact hierarchical enumerations are efficient, but not good for communications and storage. This is because inserting a value into a sub-enum affects many values in the parent. In UTOPIA27 they are used in APIs, which are versioned. A change in an enumeration results is a material change to an API. UTOPIA27 allows the programmer to reserve a range of values for future expansion.

6.4. **Straightforward functions.** *This section is an initial sketch, and will be developed further.*

As mentioned above, UTOPIA27 has specific support for producer graphs. That is, graph-like code can be defined. Functions that are DAGs can be defined as such. The code involved is really not much different from imperative code that does the same kind of thing.

UTOPIA27 also has a concept of *straightforward* functions. (Actually, I want to find the correct name for these.) Such functions are not exactly pure. The main feature is that they do not mutate their arguments and are repeatable. They can have side effect, such as sending a message to write to a log. What matters is that we do not care how many times they are called, if at all. In addition to all this, UTOPIA27 has a strong concept of expressions that it can process when compiling and when generating documentation. This feature is in no way unique to UTOPIA27, and even C++11 took steps in this direction with constant expressions.

There are two ways that UTOPIA27 uses this analysis, in addition to other benefits such as compilation optimization and richness of documentation. The first is that UTOPIA27 can track DAG expressions and functions, and specifically integer arithmetic expressions. This enables it to apply ILP solving widely. As a simple example, it understands when what happens when an variable is incremented. It understands when, say, an induction variable loops through the first half of an array. The second use that UTOPIA27 makes of this analysis is that it knows when something changes. Suppose that example 5 were a little different, that it set array values instead of getting them. How would UTOPIA27 know that the mutation has not changed the length of the array? Of course, we could construct a special case for arrays, but doing so would be undesirable. Nor do we want some clever pseudo-generality.

The resolution, or at least the general approach to it, is actually quite clear. UTOPIA27 tracks the typestate of an array, and we only need to consider what happens to an array object. An array has a kind of management object, evidenced by the fact that we can ask its size. Once the management object is allocated, which is one state, we set its size, moving it to a new state. Then we allocate the array data, and move to the full allocated state. We initialize the array data and we are in the "mutable" initialized state. Note that in, say, C++, we typically jump from right to this last state. However, by articulating this sequence we notice that the array object as a size getter that is returns constant results. One might say this is obvious, but in UTOPIA27 we need a conceptually clean and general way to say that when the array is in this state the value returned by a sizing function is constant, even when the array content is mutated.

There is a potential danger in all this. If UTOPIA27 is too clever and too invasive, it will analyse and make use of internal implementation details. In effect, encapsulation will be broken. To deal with this the writer needs to tell UTOPIA27 how far it should go. That said, we might need occasionally to be realistic about encapsulation and accept that some implementations cannot be conveniently hidden from view.

7. Adding more states

7.1. **Multi-level typestate.** *To come.*

7.2. **Winding and unwinding.** *To come.*

7.3. **In style.** *To come.* (Include the Linux style-guide example.)

7.4. **More notes.** *To come.* (Include Go's `defer`.)

References

1. *American College Dictionary*, Random House, cited in [6].
2. M. Clint and C. A. Hoare, *Program proving: Jumps and functions*, Acta Informatica **1** (1972), no. 3, 214–224.
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.
4. E.W. Dijkstra, *Notes on structured programming*, In Structured Programming (Dalai, Dijkstra, and Hoare, eds.), Academic Press, 1972, cited in [6].
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.
6. D.E. Knuth, *Ill-chosen use of "event"*, Communications of the ACM (ACM Forum) **18** (1975), no. 6, 360.
7. Donald E. Knuth, *Structured programming with go to statements*, ACM Comput. Surv. **6** (1974), no. 4, 261–301, also cited in [6].
8. ———, *Literate programming*, CSLI lecture notes, no. 27, Center for the Study of Language and Information, Stanford, CA, USA, 1992.
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.
10. P. J. Landin, *Correspondence between ALGOL 60 and Church's Lambda-notation: Part I*, Commun. ACM **8** (1965), no. 2, 89–101.
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`.
12. David O. Williams, *Structured transfer of control*, SIGPLAN Notices **19** (1984), no. 10, 46–51.
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 [6].