Glitch: A Live Programming Model

Sean McDirmid
Microsoft Research Asia
Beijing China
smcdirm@microsoft.com

Abstract

Input changes are often handled by reactive and incremental constructs that are tedious to use or inexpressive, while changes to program code are typically not handled at all during execution, complicating support for "live programming." We propose that change in code and input should be managed automatically, similar to how garbage collection eliminates memory management as an explicit programmer concern. Our programming model, Glitch, realizes such *managed time* by progressively re-executing nodes of program execution when they become inconsistent due input/code state changes. Unlike many reactive models, Glitch supports expressive shared-state procedural programming, but with one caveat: operations on shared state must be undoable and commutative to ensure re-execution efficiency and eventual consistency. Still, complex programs like compilers can be written in Glitch using mundane programming styles.

Overview

Programs react continuously to changing inputs by repairing their execution state. Beyond tediously using callbacks and observers, declarative approaches based on functions [6], constraints [7], and data binding [15] automate state repair by encoding intermediate state as data-flow; e.g. for `Z = X + Y`, `Z`'s value will be recomputed whenever `X` or `Y` change. These approaches are, however, often too inflexible in defining complex programs like incremental compilers.

Change in program code during execution is handled even more poorly, which is essential to providing programmers with responsive execution feedback during *live programming* [10] or in Bret Victor's demoes [19, 20]. We showed in [12] how input and code changes can be supported in similar ways in SuperGlue [14], but this language was declarative and not very expressive. Today, code changes are at best handled with Smalltalk-style "fix-and-continue" [9] that does not repair program state.

The goal of our work is to (a) manage input and code changes automatically, removing change as an explicit programmer concern, while (b) also improving expressiveness to expand what programs can take advantage of such **managed time**, a concept that was introduced by Edwards [5]. Incremental programming models [16] exploit the fact that small changes in input often only cause small changes in output by updating, rather than redoing, computations. For example, self-adjusting computation [1] uses dynamic dependency graphs to record reads on state so state changes can invalidate computations. However, this model involves significant programmer effort, provides little support for imperative operations, and relies on code immutability.

To do better, we are inspired by Time Warps in virtual time [11] where distributed simulation computations are executed optimistically without concern for dependency—if state is read too early, computations are "rolled back" and re-executed. Something similar could also help make incremental computations more expressive: on a change, "undo" and then redo effected computations. But we must be careful: Time Warps are prone to inefficient cascading rollback waves [18], while preparing for rollbacks can be expensive.

We meld elements of incremental computation and Time Warps together to form an expressive managed time model that can support input and code changes efficiently. Our solution improves on rollback by logging imperative operations on execution, undoing them only when they are not performed on re-execution. Because re-execution can occur in arbitrary orders, imperative operations must also be commutative. We have found that these requirements are not overly restrictive; e.g. they are good enough to write a compiler.

Glitch

A program execution in our model, called Glitch, is decomposed into a tree of *nodes* that can be re-executed independently on a code or input change. Node decompositions are specified by programmers based on their understanding of the program's run-time modularity; e.g. execution nodes for a compiler can be chosen to correspond to nodes of the syntax tree (AST) for the code being compiled. Consider:

```
node def ParseExpr(ref lexer, symtab): return match lexer.Peek:
  case ID: CallExpr(ref lexer, symtab)
  case INT: IntExpr(ref lexer)

def CallExpr(ref lexer, symtab):
  var id = Consume(ref lexer, ID)
  var args = new is List
  for ConsumeSeq(ref lexer, OPAR, COMMA, CPAR):
    args.Add(ParseExpr(ref lexer, symtab))
  var sym = symtab.Get(id.Text)
  if sym == null: SemanticError("Not found")
  else: return sym.Apply(args)
```

The language used in this paper is mostly derived from Python, but augmented with Scala pattern matching and C# reference parameters; e.g. `lexer` is both an input to and output of the `ParseExpr` and `CallExpr` methods. In this code, `ParseExpr` is defined as a node method (`node def`), meaning every call to `ParseExpr` will form a new node in the program's execution. The `CallExpr` method does not form a node itself but nonetheless is called by `ParseExpr` nodes and can create new `ParseExpr` nodes through recursive `ParseExpr` calls.

Nodes have direct data-flow dependencies with their parents for arguments and their children for return values, where they are re-executed if these change; e.g. if the code lexer stream that a `ParseExpr` node is called on changes, the node is re-executed to repair the parse tree accordingly. Likewise, nodes are re-executed when the shared state they read, which is traced dynamically, changes. For example, the `CallExpr` method reads a symbol from the `symtab` dictionary based on a parsed ID; the calling `ParseExpr` node is re-executed whenever the binding in the `symtab` changes for this ID's text. Such changes are propagated transitively: a `ParseExpr` node is re-executed when a `symtab` change causes a called `CallExpr` method to return different symbols. Return values are memoized so that a re-executed `CallExpr` method reuses constituent `ParseExpr` nodes without re-executing them.

Code in Glitch can do imperative operations; consider:

```
node def VarDecl(ref lexer, symtab):
  var kw = Consume(ref lexer, VAR)
  var id = Consume(ref lexer, ID)
  var asnOp = Consume(ref lexer, ASSIGN)
  var initial = ParseExpr(ref lexer, symtab)
  var sym = new is Symbol
  if !symtab.Put(id.Text, Sym):
    SemanticError("Duplicate definition")
```

This code parses a variable declaration, allocating a symbol that is put into the `symtab` dictionary based on a parsed ID. The imperative operations of a node are placed in a log as the node executes so they can later be undone during a rollback; in our example, the `Put` operation on a key, an id's text, and the allocated symbol, is placed in the log of a `VarDecl` node execution. When re-execution completes, the newly created log is compared with the old log of the last execution to undo operations that are no longer performed (they exist in the old but not new log). If the ID read in the first execution of a `VarDecl` node is `"fo"`, and on re-execution is `"foo"`, then the `Put("fo", ...)` is undone as it has been replaced by a newly installed `Put("foo", ...)` operation. Operation undo eliminates the need for checkpoints, and because operations are undone selectively by comparing old and new logs, indiscriminate full rollbacks are avoided when change occurs.

Nodes must conserve the objects they allocate on re-execution, otherwise shared state would be lost between re-executions while operations would be constantly undone as new object identities were created; e.g. the `Put` operation would be prevented from being reused in our example if a new `Symbol` object was allocated whenever a `VarDecl` node re-executed. Glitch models allocations as imperative operations identified by *execution addresses* that can be reproduced during node re-executions. Allocations are then re-used by identifying them by their execution addresses in old logs; e.g. the new `Symbol` assigned to the `sym` variable in our example is always the same object when a `VarDecl` node re-executes. Likewise, node invocations are also expressed as allocations identified by execution addresses to conserve their results and log state; e.g. a `ParseExpr` node conserves on re-execution whatever `ParseExpr` nodes were called under it. When a log comparison indicates that a node is no longer invoked by a parent, it is undone by undoing its logged operations, which can involve recursively undoing child node invocations.

Glitch makes no guarantees about node re-execution order as changes can affect arbitrary nodes in a program execution. Accordingly, all imperative operations must be **commutative** as they can be executed in any order, which is similar to how retroactive data structures [3] are restricted. Some operations, like aggregation or set insertion, are naturally commutative and require little modification to be used in Glitch. However, many necessary operations, like putting an entry into a dictionary as in the above example or just assigning to a cell, are not commutative as they can interfere with each other. We deal with this in Glitch by using execution addresses to restrict interfering operations to dynamic single assignment semantics; e.g. notice that the `Put` operation in the above example can fail, meaning a symbol was already placed into the symbol table at the same key. Once an operation assigns a cell or dictionary entry, all other operations with different execution addresses will fail dynamically, ensuring commutativity; e.g. failure of the above `Put` operation results in a "duplicate def" compiler error.

Due to commutativity, execution order is irrelevant in Glitch; consider:

```
var A = B; var B = A
```

Two `VarDecl` nodes are created when this code is parsed; initially, parsing of the `A` var is unable to find `B`, so an error is raised, but after the `B` var is parsed, `A`'s initializer is re-executed and `B` is found. As dependencies go, the `A` initializer depends on `B` var, and the `B` var initializer depends on `A` var. Such cycles are valid and meaningful in Glitch; while nodes can even depend on themselves!

Glitch provides primitive state structures like sets, sorted trees, dictionaries, cells, and simple aggregators, which can

be composed into user-defined structures like tables, syntax trees, and so on. Still, some data structures cannot be expressed efficiently because their operations are not undoable and/or commutative; e.g. a maximum value aggregator must be encoded as the last element of a sorted set, while even simple lists must leverage sorted execution addressees for ordering. Beyond this, Glitch is expressive in how stateful structures can be organized: they can be nested inside other structures, aliased freely, and accessed indirectly; dynamic tracing ensures that dependencies are always recorded precisely based on actual accesses. Even dependencies are encoded as logged set insertions that are undone as soon as a node no longer depends on certain shared state, eliminating the weak references needed for more conservative tracing.

Glitch supports code changes through logs that only observe actual node executions; no analysis of code is necessary. When code changes are considered, reproduced execution addresses must remain stable as code is inserted and deleted around it, which we accomplish through custom incremental lexing that conserves token identities through an edit, and then using these tokens and the call stack to form execution addresses. Loops present an additional challenge: we could append an integer index to an execution address to represent each loop iteration, but this would break down as elements were added to whatever structure was being iterated over; consider code to parse call expression arguments:

```
for ConsumeSeq(ref lexer, OPAR, COMMA, CPAR):
  args.Add(ParseExpr(ref lexer, symtab))
```

This code iterates over a parenthesis delimited, comma separated list, but uses tokens, rather than integer indices, to form execution addresses that identify each iteration, allowing the state of each ParseExpr call to be conserved even as arguments are added/removed around it via code editing.

Live Programming

Glitch as presented in this paper is used to implement a live programming experience that augments debugging with responsive execution feedback [13]. Our implementation involves very-incremental compiler and editor code written in C# that leverages Glitch as a library; as a result, execution addresses must be provided manually while code changes are not supported. The C# library approach is very flexible; e.g. parse nodes are scheduled for re-execution directly by the incremental lexer, which otherwise cannot use Glitch due to a lack of reasonable node boundaries. Currently, 3,000 lines of parsing code, 2,000 lines of type checking code, 1,000 lines of code generation code, and 2,000 lines of UI code (all C#) use Glitch to implement live programming—most of this code is oblivious to incremental capabilities. Glitch itself consists of 1,500 lines of C# code.

Glitch also forms the basis behind the programming model of our live programming language, called YinYang, that is supported with transparent execution addresses, using conserved lexical token identities, and repairs program executions after every keystroke-based code edit. Code changes bubble through the incremental compiler, triggering code generation, which then invalidates (schedules for re-execution) nodes of the executing program whose code has changed. Execution of a YinYang program then produces a list of trace entries, ordered by execution addresses, and individual probe entries that are displayed in the editor to help the programmer debug code.

Looking Forward

We believe that Glitch is efficient, performing well enough for our current live programming demos. However, while computations are conserved effectively, the logs for some nodes can grow to have many entries, e.g. 50+ operations for one AST parse node, raising concerns about memory efficiency. Additionally, we must consider the cost of logging during execution as well as unnecessary redundant re-executions. Other systems, such as those based on topological sorts [2] or Edwards' Coherent Reactions [4], attempt to find a "correct" order of re-evaluation that avoids redundant computations. Such orders are impossible to find in Glitch, and so we just deal with the temporary inconsistencies of imperfect executions orders—hence the name "Glitch."

Nodes in Glitch conceptually execute concurrently and only interact through data dependencies, which are simply re-executed until these data-dependencies stabilize. As a result, Glitch is currently suitable only for programs that can tolerate eventual consistency; i.e. programs that do not perform irrevocable real world actions. While programs are re-executed until consistent, it is possible to construct programs that will never be consistent, e.g. A = B and B = !A, and result in infinite re-execution. Future work is needed to determine how such hazards can be detected or avoided.

Without a notion of time, Glitch is also inappropriate for interactive or event-based non-batch programs, where time, not just input or code changes, can drive state and output changes. To support such "event" time, we are looking at Time Warp-related [11] *space-time memory* [8] that adds a time dimension to state. State in a Glitch program would then be allowed to change according to an external event, while nodes could be "split" on re-execution if their behavior changed across an event. Such support would then allow us to implement demos where programmers can easily "scrub" through time, and provide responsive feedback as code changes propagate through a time line.

Finally, the Time Warps that Glitch is inspired by deal with distributed and parallel computation, not incremental computation. The fact that optimistic execution is useful for concurrency is also not surprising, which is exploited by software transactional memory [17]. We should explore if the re-execution mechanism that allows Glitch to manage change for incremental computation, could also be useful for eliminating inconsistency that arise in concurrent, parallel, and distributed computing contexts.

References

[1] U. A. Acar. Self-adjusting computation: (an overview). In *Proc. of PEPM*, pages 1–6, 2009.

[2] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proc. of ESOP*, pages 294–308, 2006.

[3] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures. In *Proc. of SODA*, pages 281–290, 2004.

[4] J. Edwards. Coherent reaction. In *Proc. of Onward!*, pages 925–932, 2009.

[5] J. Edwards. Time is of the essence. `alarmingdevelopment.org/?p=739`, Feb. 2013.

[6] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of ICFP*, pages 263–273, 1997.

[7] B. N. Freeman-Benson. Kaleidoscope: mixing objects, constraints, and imperative programming. In *Proc. of OOPSLA/ECOOP*, pages 77–88, 1990.

[8] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proc. of ICPP*, pages 201–208, 1991.

[9] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[10] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[11] D. R. Jefferson. Virtual time. *ACM TOPLAS*, 7(3):404–425, July 1985.

[12] S. McDirmid. Living it up with a live programming language. In *Proc. of OOPSLA Onward!*, pages 623–638, October 2007.

[13] S. McDirmid. Usable live programming. In *Proc. of SPLASH Onward!*, 2013. To Appear.

[14] S. McDirmid and W. C. Hsieh. Superglue: Component programming with object-oriented signals. In *Proc. of ECOOP*, pages 206–229, 2006.

[15] A. Nathan. *Windows Presentation Foundation Unleashed (WPF) (Unleashed)*. Sams, 2006.

[16] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Proc. of POPL*, pages 502–510, 1993.

[17] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of PODC*, pages 204–213, 1995.

[18] S. C. Tay, Y. M. Teo, and R. Ayani. Performance analysis of time warp simulation with cascading rollbacks. In *Proc. of PADS*, pages 30–37, 1998.

[19] B. Victor. Inventing on principle. Invited talk at CUSEC, Jan. 2012.

[20] B. Victor. Learnable programming. `worrydream.com/LearnableProgramming`, Sept. 2012.