Glitch: a programming model for Live Programming

Sean McDirmid
Microsoft Research Asia
Live Feedback: Archer Analogy

[Hancock, 2003]

Archer:

- aim ➔ shoot ➔ see ➔ repeat
- edit ➔ run ➔ debug ➔ repeat

Hose:

- aim ↔ see
- edit ↔ debug
Can we make this real? (beyond small demos)
Challenges

**Infrastructure** *(lots of change!)*

- Incremental compiler
- Editor
- Execution engine
Living it up with a **Live Programming Language**
[McDirmid, 2007]

Superglue (my thesis)
FRP-like language with signals
Handling code changes “fit” into its programming model
Cool toy demos

What about **real** code!?  
All infrastructure for this demo was written with something else!
Living it up with a Live Programming Language
[McDirmid, 2007]

Incremental Compilation Framework
Damage/repair of memoized tree nodes
Originally for Scala in Scala
Incremental Compilation

foo(10, x)
Incremental Compilation

foo(10, xanadu)
Incremental Compilation

Type checkers need symbol tables
Indirection, key names depend on input, non-local

Go beyond plain memoization
Trace non-local dependencies
Log, undo symbol table entries per tree
Incremental Compilation

```
foo(10, x)  
x = 20
```

def Assign(ref cursor):
    var id = consume(ref cursor, ID)
    consume(ref cursor, ASSIGN)
    var to = Expr(ref cursor)
    symtab.Add(id, to.Type)

def IdE(ref cursor):
    var id = consume(ref cursor, ID)
    return symtab[id]
**Incremental Compilation**

foo(10, x)

x = “hello”

def Assign(ref cursor):
    var id = consume(ref cursor, ID)
    consume(ref cursor, ASSIGN)
    var to = Expr(ref cursor)
    symtab.Add(id, to.Type)

def IdE(ref cursor):
    var id = consume(ref cursor, ID)
    return symtab[id]
foo(10, x)

x = “hello”

def Assign(ref cursor):
    var id = consume(ref cursor, ID)
    consume(ref cursor, ASSIGN)
    var to = Expr(ref cursor)
    symtab.Add(id, to.Type)

def IdE(ref cursor):
    var id = consume(ref cursor, ID)
    return symtab[id]
Incremental Compilation

Able to weave this into scalac
Could mostly handle the way scalac was implemented
Actually this was a requirement...

No need to lose imperative programming
But then scalac also was not very imperative...

Adapted quickly for SuperGlue
Live programming in 2013 is cool again!
Time to dust off my old tricks

This time...generalize

Infrastructure/language based on same programming model

“symbol table add” → any imperative operation?
Glitch as the simplest model that could possibly work
Glitch

Divide program execution up into nodes
Trace dependencies
Log side-effecting operations (imperatives)
Re-execute node on dependency change
Reap after re-exec: undo dead imperatives
  Compare old and new log; recursively undo dead nodes
Imperatives must be undoable

Nodes can re-execute in any order:
Imperatives must be commutative w.r.t. order!

Supported imperatives:
- Set Add, Aggregation (trivial)
- Assignment (dynamic single assignment restriction)
- Dictionary Set (like assignment)
- List.Append (provide ordered execution address)
```python
def Assign(ref cursor):
    var id = consume(ref cursor, ID)
    consume(ref cursor, ASSIGN)
    var to = Expr(ref cursor)
    symtab.Add(id, to.Type)

def IdE(ref cursor):
    var id = consume(ref cursor, ID)
    return symtab[id]
```
foo(10, x)
x = “hello”

```python
def Assign(ref cursor):
    var id = consume(ref cursor, ID)
    consume(ref cursor, ASSIGN)
    var to = Expr(ref cursor)
    console.Append(“hello “ + to.Tp)
    symtab.Add(id, to.Type)

def IdE(ref cursor):
    var id = consume(ref cursor, ID)
    return symtab[id]
```
Why “Glitch”? 

Consistency is **eventual**

- No attempt is made to find an “optimal” re-execution order
- Sandbox during development, commit when nothing is damaged in production

No **fancy analyses; just logging**

- Kind of **boring** from a technical perspective, but it works!
Glitch in the stack

Programmer

2000 loc

C#

Editor

3500 loc

C#

Incremental Compiler

8000 loc

Compiled Code

Runtime

Compiled Code

Program Execution

YinYang
Glitch payoffs

Expressive semi-imperative code

Automatic repair management

Simple implementation
  Completely dynamic (also a cost...
*But wait, there's more!*

<table>
<thead>
<tr>
<th>Iterative computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No restriction on what writes can be seen</td>
</tr>
<tr>
<td>Parsing and type checking together in same pass</td>
</tr>
</tbody>
</table>

| Optimistic speculative parallelism |

| Distributed computing |

...
Asynchronous Execution

// synchronous - old/blocking/deterministic
var dataA = fileA.read()
process(dataA)
var dataB = fileB.read()
process(dataB)

// asynchronous - new/non-blocking/non-deterministic
fileA.readAsync(dataA => process(dataA))
fileB.readAsync(dataB => process(dataB))

// Glitch - retro/non-blocking/deterministic
var dataA = fileA.read()
process(dataA)
var dataB = fileB.read()
process(dataB)
**Time** as major future work

Cannot abstract over time, events, and **interactivity**

...so no debugging of time-stepped computations (yet)
Time as major future work

Cannot abstract over time, events, and interactivity
...so no debugging of time-stepped computations (yet)
Drawbacks

- Tracing logging have costs *(performance)*
- Only semi-imperative *(expressiveness)*
- Eventual consistency makes pulling triggers harder
- Not complete until *time* is included
Conclusion

Why not manage change like garbage collection manages memory?