Correct-by-Construction Interactive Software: From Declarative Specifications to Efficient Implementations

Kyle Headley

Matthew Hammer

OBT ‘16
Interactive System
Interactive System

User Input

Behavior

System Output

State
Programming interactive experiences is complex

Existing interactive experiences are underspecified
Complex examples

Obvious
- Extending emacs/eclipse

Subtle
- What is the semantics of UNDO?
- Undo with multiple buffers?
- Undo over higher order commands?
Interactive System

System input

Carefully Calculated State Modifications

System output

• data cache
• global vars
• display objects
• animation progress
Using PL Features, Algorithms, Techniques
for interactive behavior

Programming interactive experiences can be simple

Existing interactive experiences can be unambiguous

Code can be independent of the framework used
Correct-by-construction Interactive Program

Logical Spec
- Deterministic functions
- Functional Programming

Executable Spec
- Library Integration
- HCI Considerations

Correct Implementation
- Incremental Computation
- Optimisations

Responsive Implementation

Semantics preserving transformation
Correct-by-construction Interactive Program

Logical Spec
- Deterministic functions
- Functional Programming

Executable Spec
- Library Integration
- HCI Considerations

Correct Implementation
- Incremental Computation
- Optimisations

Responsive Implementation
A computation is incremental if repeating it with a changed input is faster than from-scratch recomputation.
Incremental Computation

Incremental computation can use:

• Memoization / Caching
• Dependency graphs
• Internal efficient data structures
Incremental Computation

An incremental computation library may be simple or complex

```rust
def fn cmdz_of_actions
  <A:Adapton
   ,Acts:TreeT<A,Action>
   ,Cmds:TreeT<A,Command>
   ,Edit:ListEdit<A,Command,Cmds>
 >
 (st: &mut A, acts:Acts::Tree)
 -> (Edit::State, Option<A::Name>) { 
  ...
  let z = Edit::insert(st, z, Dir2::Left, c) ;
  let z = Edit::ins_cell(st, z, Dir2::Left, nm.clone()) ;
  let z = Edit::ins_name(st, z, Dir2::Left, nm) ;
  ...
}
def fn foo(a: u32) -> u32 { 
  ...
}
def memo!(ic_tables, foo, a:a)```
```
IC_Edit

Features:

• Text creation/removal
• Undo/Redo
• Multiple ‘cursors’
Proof of Concept: Text Editor

Logical Spec
- Deterministic functions
- Functional Programming

Executable Spec
- Library Integration
- HCI Considerations

Correct Implementation
- Incremental Computation
- Optimisations

Responsive Implementation
IC_Edit

‘Action’

A* ⇓ Z

Content list

Complete action List
IC_Edit

Primitive operations

Edits
Insert, Overwrite, Delete, Undo, Redo

Cursor Management
Move, Make, Jump, Switch, Join
Primitive operations

Edits

Insert, Overwrite, Delete, Undo, Redo

Cursor Management

Move, Make, Jump, Switch, Join

Demo — Correct Implementation
IC>Edit Spec

Build content from action list

Initial cursor

Handle Undo/Redo

Build content

α fresh

$A^* \downarrow \langle C_1^* || C_2^* \rangle$

$\langle \varepsilon \mid \alpha \mid \varepsilon \rangle \vdash \text{reverse}(C_1^*) \downarrow Z$

$A^* \downarrow Z$
Zipper

Linked list of earlier data  Optional Cursor  Linked list of later data

First  Mid  Last

Links progress away from cursor
IC_Edit Spec

A* ↓ {C₁* || C₂*}

A ::= UNDO | REDO | Command

A*: Action List

C₁*: Commands to process

C₂*: Commands that have been undone
$Z_1 \vdash \text{reverse}(C_1^*) \downarrow Z_2$

Command ::= Insert | Delete | Overwrite | CursorCommands

$Z_1$: Initial content zipper

$C_1^*$: Active commands

$Z_2$: Final content zipper
Which data structure to choose?

**Array**
- **Easy**: Random access
- **Hard**: Insert, Search

**Zipper**
- **Easy**: Insert, local move
- **Hard**: Search

**Tree**
- **Easy**: Search, global move
- **Hard**: Insert (balancing issues)
Key Insight

- Use data structure that is easiest for each operation
- Transform to other structures and keep them in cache
- Use incremental computation for optimizations
Preliminary Results

Update Time After Many Interactions

Time (milliseconds)

Interactions

Spec
w/ IC
IC_Edit Future

Optimise Searching

Reduce overhead

Implement additional features
- Search for words
- Better navigation
Correct-by-construction next steps

Spreadsheet  IDE

Improved Adapton interface

Better composition of techniques

Servo:
Web browser in rust
Take-aways

Use IC to cache useful data structures

Spec as program

Correct-by-construction Interactive Program

Simple state

Design

Logical Spec
Executable Spec
Correct Implementation
Responsive Implementation

Semantics preserving transformation

Specialized input

Specialized output

Kyle Headley, Matthew Hammer, University of Colorado at Boulder, OBT’16