Rediscovering Ousterhout’s Dichotomy in the 21st Century while Developing and Deploying Software for Set-Theoretic Empirical Analysis:

*From R to Python/Qt to OCaml and Tcl/Tk*

Claude Rubinson  
University of Houston—Downtown  
rubinsonc@uhd.edu

26th Annual Tcl/Tk Conference  
Houston, TX  
November 7, 2019

Funding support:  
UHD Organized & Creative Activities Award, 2019
[I propose that] you should use two languages for a large software system: one, such as C or C++, for manipulating the complex internal data structures where performance is key and another, such as Tcl, for writing small-ish scripts that tie together the C pieces and are used for extensions. For the Tcl scripts, ease of learning, ease of performance and ease of glue-ing are more important than performance or facilities for complex data structures and algorithms. I think these two programming environments are so different that it will be hard for a single language to work well in both.

— Ousterhout, 1994
...Prickly theoreticians seek to understand the world through the abstractions of thought, whereas gooey empiricists return ceaselessly to the real world for the ever-more-refined data that methodical experimentation can yield. The complementarity of the two approaches is widely recognized by scientists themselves. A constant dialectic between empiricism and theory is generally seen to promote the integrity and health of scientific inquiry.

More recently, the advent of computer programming has brought with it a new round in the prickly-gooey dialectic. By temperament, there seem to be two types of programmers, which I will call the planners and the doers.

— Flynt, 2012
Inductive/Deductive Knowledge Creation

- *a posteriori* knowledge, via observation
  - inductive research
  - Flynt’s “gooey doers”

- *a priori* knowledge, via logical reasoning
  - deductive research
  - Flynt’s “prickly planners”
**Programming Languages, Domain Characteristics, and Development Strategies**

<table>
<thead>
<tr>
<th>Domain Familiarity</th>
<th>Application (Inductive exploration)</th>
<th>Analytic (Deductive reasoning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Known</td>
<td>Scripting (e.g., DDD)</td>
<td>DSL or System Programming (e.g., TDD)</td>
</tr>
<tr>
<td>Unknown</td>
<td>Scripting (e.g., Agile)</td>
<td>System Programming</td>
</tr>
</tbody>
</table>
Programming as Knowledge Creation

• Scripting complements inductive reasoning
  - goal: model an application domain to solve an existing problem
  - encourages: rapid acquisition of needed knowledge; quick and regular feedback from stakeholders
  - prioritizes: ease and speed of writing and deploying code

• Systems programming complements deductive reasoning
  - goal: develop a coherent model (algebra) of an analytic domain
  - encourages: deep understanding of abstract entities and their relationships with one another; casing practices
  - prioritizes: developing semantically meaningful abstractions; avoiding logical (syntactic) errors
Qualitative Comparative Analysis (QCA)

- A configurational-comparative methodology that uses set theory and Boolean algebra to investigate multiple conjunctural causation

- Software design goals:
  - Explore/develop QCA methodology (analytic domain)
  - Identify effective ways of conducting QCA (application domain)
The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

— Knuth, 1974

...the only downside to Python I’ve found is that, as currently implemented, its execution speed may not always be as fast as that of compiled languages...

As a general-purpose programming language, Python’s roles are virtually unlimited: you can use it for everything from website development and gaming to robotics and spacecraft control.

— Lutz, 2009
Prior Implementations of QCA Software

R (ca. 2006)
- Poor performance; R programming “considered harmful”
- But: helped me realize that UI should be task-oriented

Python (ca. 2009–17)
• Two versions: acq (Unix CLI) & Kirq (crossplatform GUI)
• Advantages:
  - Compared to R, lower SLOC with comparable functionality and better performance
  - Core language is relatively compact, with large standard library
  - Strong, well-developed environment of GUI toolkits, installers, etc
• Disadvantages:
  - Out-of-box performance often still too slow; optimization can be difficult
  - Low SNR online; insular community overly concerned with “idiomatic Python”
  - Bit rot and package churn hurts maintenance and distribution (especially Win10; see also: Python3)
OCaml & Tcl/Tk (2017–present)

Analytic domain (OCaml)

- Libraries of QCA data structures and algorithms
- Basic CLI interfaces
- Easy distribution by building platform-specific executables

Application domain (Tcl/Tk)

- User-interface(s)
- Data management and transformation
- Session history/management
- Crossplatform persistence layer (SQLite)
- Easy distribution by building crossplatform executables
Algebraic Data Types:
Easily express complex and recursive data structures

- **Product types (tuples and records)**
  - `type fzvar = string * Fuzzy.t list`
  - `type qcadata = {obs: string list;
    vars: fzvar list;
    directives: string list}`

- **Sum types (variant/discriminated union)**
  - `enum that optionally carries a payload`
  - `type bexp =
    Atom of atom
    | Not of bexp
    | And of bexp * bexp
    | Or of bexp * bexp`
Algebraic Data Types:
Process (deconstruct) using pattern matching

- The structure of your function matches the structure of your data
- Type checking prevents omitting a case
- Recursive application is straightforward

- (* apply De Morgan's laws to push negations in until they only apply to literals, so that statement consists only of literals, conjunctions, and disjunctions *)

```ml
let rec nnf = function
  Atom a -> Atom a
| Not (Atom a) -> Atom (neg_atom a)
| Not (Not a) -> nnf a
| Not (And (a,b)) -> Or (nnf (Not a), nnf (Not b))
| Not (Or (a,b)) -> And (nnf (Not a), nnf (Not b))
| And (a,b) -> And (nnf a, nnf b)
| Or (a,b) -> Or (nnf a, nnf b)
val nnf : bexp -> bexp
```
Hindley-Milner Type Inference

- Consistent & Complete
  - Type checks entire program during compilation
- Supports parametric polymorphism
  - Type checking of abstract functions and types
  - Eliminates run-time type errors for pure functions
- Extensible
  - Haskell’s type classes and Ocaml/SML module system
- Infers most general (principle) type

\[ (* p \text{ is a subset of } q \text{ when } p \lor \neg q = 0 *) \]

let is_subset p q =
    is_contradiction (And(p, Not q))

val is_subset : bexp -> bexp -> bool
Analytic Domain:
Symbolic Boolean Algebra

- Strengthens and makes explicit the set-theoretic foundation of QCA
  - Boolean expressions may be arbitrarily complex
  - Encourages analysis of complex sets, rather than individual conditions and outcomes

- Boolean expressions may be associated with particular constraints
  - Impossible conjunctions
  - Theoretical/empirical expectations
type atom =  
  Yes of string  
  | No of string  
  | Dc of string  
  | Imp of string  
  | One  
  | Zero  

type bexp =  
  Atom of atom  
  | Not of bexp  
  | And of bexp * bexp  
  | Or of bexp * bexp  

let neg_atom = function  
  Yes a -> No a  
  | No a -> Yes a  
  | One -> Zero  
  | Zero -> One  
  | a -> a  

let neg_atom : atom 
-> atom  

let rec bnot = function  
  Atom a -> Atom (neg_atom a)  
  | Not a -> a  
  | And (a,b) -> Or (bnot a, bnot b)  
  | Or (a,b) -> And (bnot a, bnot b)  

val bnot : bexp 
-> bexp  

(* boolean negation *)  

let rec band p q =  
  match (p,q) with  
    a, Or (b,c) | Or (b,c), a ->  
      Or(band a b, band a c)  
    | a,b -> And (a,b)  

val band : bexp 
-> bexp 
-> bexp  

let rec bor p q =  
  match (p,q) with  
    a, And (b,c) | And (b,c), a ->  
      And(bor a b, bor a c)  
    | a,b -> Or (a,b)  

val bor : bexp 
-> bexp 
-> bexp  

(* distributive laws for boolean multiplication/addition *)
Analytic Domain: Modeling Missing Data

- QCA currently accommodates missing data only via listwise deletion

- Can extend QCA’s conventional 2-valued Boolean logic to a 4-valued logic (cf., Codd’s RM/V2)
  - Two forms of missing data: unknown values and inapplicable values
  - 4VL allows calculation of “maybe” and “impossible” set relationships; provides foundation for supervaluation
  - Because 4VL complexity taints the entire program (McGoveran 1994), type inference becomes crucial for avoiding logical errors (cf., truthy/falsey values)
module Fznum = struct
  type t = 
    | Unk
    | Iap
    | Fz of float

  let fznot = function
    | Fz a -> Fz (1. -. a)
    | Unk -> Unk
    | Iap -> Iap
  val fznot : t -> t

  let fzand p q = match (p,q) with
    | Fz a, Fz b -> Fz (min a b)
    | Fz 0.0, _ -> Fz 0.0
    | _, Fz 0.0 -> Fz 0.0
    | _, Iap -> Iap
    | Iap, _ -> Iap
    | _, Unk -> Unk
    | Unk, _ -> Unk
  val fzand : t -> t -> t
end

module Fzset = struct
  type t = Fznum.t list

  let fsnot p = List.map Fznum.fznot p
  val fsnot : Fznum.t list -> Fznum.t list

  let fsand p q = List.map2 Fznum.fzand p q
  val fsand : Fznum.t list -> Fznum.t list -> Fznum.t list

  let fsor p q = List.map2 Fznum.fzor p q
  val fsor : Fznum.t list -> Fznum.t list -> Fznum.t list
end

let fzor p q = match (p,q) with
  | Fz a, Fz b -> Fz (max a b)
  | Fz 1., _ -> Fz 1.
  | _, Fz 1. -> Fz 1.
  | _, Unk -> Unk
  | Unk, _ -> Unk
  | Iap, Iap -> Iap
  | Fz a, Iap -> Fz a
  | Iap, Fz a -> Fz a
val fzor : t -> t -> t
Application Domain:
UIs that encourage retroductive, interrogative analysis

• Extended Tcl console
  – Tcl provides general computing environment for
  • data management/transformation, math & statistics, etc.
  • calling out to external programs
  – Adds commands for conducting QCA
  – Also provides venue for procedures that are
    • infrequently used,
    • exploratory/under development, or
    • don’t (yet) fit into the GUI’s model

• Tk GUI
  – Spawned from Tcl console (permits bi-directional communication)
  – User-friendly but opinionated
  – Easy to maintain and modify
Application Domain:
Incorporation of Multivalued Sets

- Multivalued sets permit >2 states per condition
- Tcl for representing/manipulating data sets
- Convert multivalued set to series of (disjoint) crisp sets and defining derived conjunctions as impossible:
  - party\{dem, rep, ind\} →
    \[ d\{0,1\}; r\{0,1\}; i\{0,1\} + \text{Imp}\{d\&r, d\&i, r\&i\} \]
- Convert between multivalued and conventional notation(s)
Implications:  
The Continuing Relevance of Ousterhout’s Dichotomy

● Scripting and system programming languages have distinct capabilities, not just for gluing versus writing components as Ousterhout argued but also for conducting inductive versus deductive research.

● The contemporary prioritization of scripting languages above system programming languages is therefore problematic.
  − A mismatch between the type of research and one’s choice of language can hinder knowledge acquisition.
  − Especially pernicious is the contention that a single scripting (or DSL) language such as Python or R is sufficiently general to meet all needs.

● And yet, scripting languages are indeed more accessible, especially for researchers, who are often casual programmers. How to address?
  − Maintain conventional split between researcher and programmer.
  − Continued work on both DSLs and “high level” system programming languages.
  − Distinguish between “gooey doing” and “prickly planning” activities.
I hope that programmers will consider the differences between scripting and system programming when starting new projects and choose the most powerful tool for each task.

— Ousterhout, 1998

Program close to the problem domain.

— Hunt and Thomas, 2000