Correctness-by-Construction in Stringology

Bruce W. Watson

FASTAR Research Group, Stellenbosch University, South Africa

bruce@fastar.org

Institute of Cybernetics at TUT, Tallinn, Estonia, 3 June 2013

Aim of this talk

- Motivate for *correctness-by-construction* (CbC)

  ...especially in stringology
- Introduce CbC as a way of explaining algorithms
- Show how CbC can be used in *inventing* new ones
- Give some new notational tools

Contents

fastar
RESEARCH GROUP

What *is* CbC?

Methodology sketch:

1. Start with a *specification*

   . . . and a simple programming language

   . . . and a logic
2. *Refine* the specification

   . . . in tiny steps

   . . . each of which is *correctness-preserving*
3. Stop when it's executable enough

What do we have at the end?

- An algorithm we can implement
- A *derivation* showing how we got there
- An interwoven correctness proof

fastar RESEARCH GROUP

Why is correctness critical in stringology?

- Many stringology problems in *infrastructure soft-/hardware*
- Devil is in the details, cf. repeated corrections of articles
- Stringology is curriculum-core stuff
- The field is very rich — overviews, taxonomies, etc. are needed to see interrelations

What are the alternatives?

Testing

- Only shows the *presence* of bugs, not *absence*
- Most popular

A postiori proof

- Think up a clever algorithm, then set about proving it
- Leads to a decoupling which can be problematic, potential gaps, etc.
- Most popular proof type

Automated proof

- Requires a *model* of the algorithm
- Potential discrepancy between algorithm and model
- Tedious

Bonus?

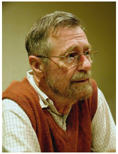
We get a few things for free.
The 'tiny' derivation steps often have choices which can lead to other algorithms, giving:

- Deriving a *family* of algorithms

  . . . e.g. the Boyer-Moore type 'sliding window' algorithms
- *Taxonomizing* a group of algorithms with a *tree* of derivations
- *Explorative algorithmics* — at each opportunity, try something new

Short history

We stick to a CbC for imperative/procedural programs[1]:

- In the late 1960's
- Largely by these guys:  

  with Floyd, Knuth, Kruseman Aretz, . . .
- Followed in the 80's by more work due to Gries, Broy, Morgan, Bird, . . .
- Taught in algorithmics at various uni's

[1] Other paradigms exist of course: functional, logical

Key components

We're going to need

- A simple pseudo-code: *guarded command* language (GCL)

  5 statement types
- A simple *predicate* language (first order predicate logic)
- A calculus and some strategies on these things

Hoare triples, frames, . . .

Hoare triples, e.g. $\{P\}S\{Q\}$

- P and Q are *predicates* (assertions), saying something about variables

  P is called the precondition

  Q is the postcondition
- S is some program statement (perhaps compound)
- For reasoning about *total correctness*: this triple asserts that

  if P is true just before S executes, then S will terminate and Q will be true
- E.g. $\{x = 1\}x := x + 1\{x = 2\}$
- Invented by Tony Hoare[2] and Robert Floyd
- Was used for (relatively ad hoc) reasoning on flow-charts

[2] He didn't just do Quicksort

Useful things you can do with Hoare triples

Dijkstra et al invented a *calculus* of Hoare triples

- Start with $\{P\}S\{Q\}$ where S is to be invented/constructed

  This triple is a *algorithm skeleton*
- We can *elaborate* S as a compound GCL statement

  Using rules based on the syntactic structure of GCL
- Work *backwards*

  Our post-condition is our only goal

What can we legally do?

- *Strengthen* the postcondition: achieve more than demanded
- *Weaken* the precondition: expect less than guaranteed

Morgan and Back invented refinement calculi

Sequences of statements

Given skeleton $\{P\}S\{Q\}$, split S into two (still abstract) statements

$$\{P\}S_0; S_1\{Q\}$$

What now?

- We would like the two new statements to each do part of the work towards Q
- 'Part of the work' can be some predicate/assertion R, giving

$$\{P\}S_0; \{R\}S_1\{Q\}$$

- Now we can proceed with $\{P\}S_0\{R\}$ and $\{R\}S_1\{Q\}$ more or less in isolation

Note that ';' is a *sequence operator*

Example: sequence

{ **pre** m and n are integers }
S
{ **post** $x = m \max n \wedge y = m \min n$ }

can be made into

{ **pre** m and n are integers }
S_0;
{ $x = m \max n$ }
S_1
{ **post** $x = m \max n \wedge y = m \min n$ }

which can be further refined (next slides)

Assigning to a variable

Sometimes it's as simple as an *assignment* to a variable:

Refine $\{P\}S\{Q\}$

to $\{P\}x := E\{Q\}$ (for expression E) if we can show that

$$P \implies Q[x := E] \qquad \text{i.e. } Q \text{ with all } x\text{'s replaced with } E\text{'s}$$

For example

$\{$ **pre** m and n are integers $\}$
S_0;
$\{\ x = m \max n\ \}$
$y := m \min n$
$\{$ **post** $x = m \max n \wedge y = m \min n\ \}$

because clearly

$$(x = m \max n \wedge m \min n = m \min n) \equiv (x = m \max n)$$

fastar RESEARCH GROUP

IF statement

Refine $\{P\}S\{Q\}$ to

$\{\ P\ \}$
if $G_0 \rightarrow \{\ P \wedge G_0\ \}\ S_0\{\ Q\ \}$
$[\!]\ G_1 \rightarrow \{\ P \wedge G_1\ \}\ S_1\{\ Q\ \}$
fi
$\{\ Q\ \}$

if $P \implies G_0 \vee G_1$
For example

$\{$ **pre** m and n are integers $\}$
if $m \geq n \rightarrow x := m;\ \ y := n$
$[\!]\ m \leq n \rightarrow x := n;\ \ y := m$
fi
$\{$ **post** $x = m \max n \wedge y = m \min n\ \}$

Note nondeterminism!

DO loops

What do we need to refine to a loop?

Invariant:
- Predicate/assertion
- True before and after the loop
- True at the top and bottom of each iteration

Variant:
- Integer expression
- Often based on the loop control variable
- Decreasing each iteration, bounded below
- Gives us confidence it's not an infinite loop

DO loops

For invariant I and variant expression V we get

$$
\begin{array}{l}
\{ \; P \; \} \\
S_0; \\
\{ \; I \; \} \\
\textbf{do} \;\; G \rightarrow \{ \; I \wedge G \; \} \\
\qquad\qquad\quad S_1 \\
\qquad\qquad\quad \{ \; I \wedge (V \text{ decreased}) \; \} \\
\textbf{od} \\
\{ \; I \wedge \neg G \; \} \\
\{ \; Q \; \}
\end{array}
$$

Remember to check $P \implies I$ and $I \wedge \neg G \implies Q$

Example: DO loop

Given

$\{\ x, i$ are integers and A is an array of integers and $x \in A\ \}$
S
$\{$ **post** i is minimal such that $A_i = x\ \}$

we can choose

Invariant $x \notin A_{[0\ldots i)}$
Variant $|A| - i$

in

$\{\ x, i$ are integers and A is an array of integers and $x \in A\ \}$
$\{$ **invariant** $x \notin A_{[0\ldots i)}$ and **variant** $|A| - i\ \}$
do $A_i \neq x \rightarrow$
$\quad i := i + 1$
od
$\{$ **post** i is minimal such that $A_i = x\ \}$

Example derivation: the Boyer-Moore family

Specification and starting point

{ **pre** p, S are strings }
T
{ **post** $M = \{x : p \text{ appears at } S_x\}$ }

Output variable M is used to accumulate the matches
We'll introduce auxiliary variables as needed, starting with j left-to-right in S
The 'collection' M indicates we need a loop

fastar RESEARCH GROUP

Introducing the outer loop

Invariant $I : M = \{x : x < j \wedge p \text{ appears at } S_x\}$
Intuitively, this says we have accumulated the matches left of j
Variant V: $|S| - j$

$\{$ **pre** p, S are strings $\}$
T_0;
$\{ \ I \ \}$
do $j \leq |S| - |p| \rightarrow \{ \ I \wedge (j \leq |S| - |p|) \ \}$
T_1
$\{ \ I \wedge (V \text{ has decreased}) \ \}$
od
$\{ \ I \wedge \neg(j \leq |S| - |p|) \ \}$
$\{$ **post** $M = \{x : p \text{ appears at } S_x\}$ $\}$

Clearly, T_0 must set j, M and T_1 must

- Update M if there's a match at j
- Increase j to move right and decrease V
- Ensure that I is true again

Updating M

Update M using a straightforward test

$\{$ **pre** p, S are strings $\}$
$j := 0;\ \ M := \emptyset;$
$\{\ I\ \}$
do $j \leq |S| - |p| \rightarrow \{\ I \wedge (j \leq |S| - |p|)\ \}$
  if p appears at $S_j \rightarrow M := M \cup \{j\}$
  $[\!]$ otherwise $\rightarrow$ **skip**
  fi;
  $\{\ \ldots\ \}$
  T_2
  $\{\ I \wedge (V$ has decreased$)\ \}$
od
$\{\ I \wedge \neg(j \leq |S| - |p|)\ \}$
$\{$ **post** $M = \{x : p$ appears at $S_x\}\ \}$

fastar RESEARCH GROUP

More ideas on updating M

What does "p appears at S_j" actually mean?
We can expand this to

$$\forall_{0 \leq x < |p|} : p_x = S_{j+x}$$

We can implement such a characterwise check from left-to-right or vice-versa or in arbitrary orders
Can also be done in hardware, ...

fastar RESEARCH GROUP

Still more ideas on updating M

Consider doing it left-to-right

Invariant J :

$$\forall_{0 \leq x < i} : p_x = S_{j+x}$$

Variant W : $|p| - i$ in

$i := 0;$
$\{ \ J \ \}$
do $i < |p| \wedge p_i = S_{j+i} \rightarrow$
  $\{ \ J \wedge i < |p| \wedge p_i = S_{j+i} \ \}$
  $i := i + 1$
  $\{ \ J \wedge (W$ has decreased$) \ \}$
od;
$\{ \ J \wedge \neg(i < |p| \wedge p_i = S_{j+i}) \ \}$
if $j \geq |p| \rightarrow M := M \cup \{j\}$
$[\!]$ otherwise $\rightarrow$ **skip**
fi

Updating j in the outer loop

Recall we can use $J \wedge \neg(i < |p| \wedge p_i = S_{j+i})$ in updating j

$$\forall_{0 \leq x < i} : p_x = S_{j+x} \wedge \neg(i < |p| \wedge p_i = S_{j+i})$$

We would ideally like to move to the *next match* using

$$j := j + (\min_{1 \leq k} : p \text{ appears at } S_{j+k})$$

This really is the magic of 'shifting windows'
How do we make this shift distance realistic?
Look at the predicate in the min

Realistic shift distances

Consider two predicates $A \implies B$ (B is a *weakening* of A)
We have

$$\min_k : B \leq \min_k : A$$

Additionally, for two predicates C, D

$$\min_k : (C \vee D) = (\min_k : C) \min (\min_k : D)$$

and

$$\min_k : (C \wedge D) \geq (\min_k : C) \max (\min_k : D)$$

So we can also split con-/disjuncts

Realistic shift distances

If we can 'weaken' predicate

p appears at S_{j+k}

we have a usable shift
What do weakenings look like?

- Boyer-Moore d_1, d_2 shift predicate
- Mismatching character predicate
- Right-lookahead (Horspool) predicate
- ...

Calculus of shift distances exploring all possible shifters

Final version of the algorithm

$\{$ **pre** p, S are strings $\}$
$j := 0;\quad M := \emptyset;$
do $j \leq |S| - |p| \rightarrow i := 0;$
  do $i < |p| \wedge p_i = S_{j+i} \rightarrow$
    $i := i + 1$
  od;
  if $j \geq |p| \rightarrow M := M \cup \{j\}$
  $[\!]$ otherwise $\rightarrow$ **skip**
  fi;
  $j := j + (\min_{1 \leq k}$: weakening of "p appears at S_{j+k}")
od
$\{$ **post** $M = \{x : p \text{ appears at } S_x\}$ $\}$

A totally *new algorithm skeleton*

{ **pre** p, S are strings }
{ Todo is a stack }
Todo $:= \emptyset$; M $:= \emptyset$;
Todo $:= \{[0, |S| - |p| + 1)\}$;
do Todo $\neq \emptyset \rightarrow$ pop $[l, h)$ from Todo;
    if $[l, h)$ is not empty $\rightarrow$
        probe $:= \lfloor \frac{l+h}{2} \rfloor$;
        if p appears at $S_{\text{probe}} \rightarrow$
        [] otherwise $\rightarrow M := M \cup \{\text{probe}\}$
        fi;
        push $[m +$ window shift to right, $h)$ onto Todo;
        push $[l, m -$ window shift to left$)$ onto Todo
    [] otherwise $\rightarrow$ **skip**
    fi
od
{ **post** $M = \{x : p$ appears at $S_x\}$ }

Redundant push/pop can be removed

Conclusions & ongoing work

- Simple/interwoven logic + language are sufficient
- CbC is relatively idiot-proof
- Notation is important
- Creativity is not hampered: new algorithms can be invented
- Useful methodology for bringing coherence to a field

  . . . and detecting unexplored parts
- Parallel programming is exponentially more difficult than sequential
  - Testing exhaustively is difficult due to all possible interleavings
  - A postiori proof is similarly difficult
  - Automated proofs are possible

fastar
RESEARCH GROUP

References

1. Dijkstra. *A Discipline of Programming*, P-H, 1976
2. Gries. *The Science of Computer Programming*, Springer, 1980
3. Cohen. *Programming in the 1990's*, Springer, 1990
4. Kaldewaij. *Programming: The Derivation of Algorithms*, P-H, 1990
5. Morgan. *Programming from Specifications*, P-H, 1998, available as PDF
6. Feijen & van Gasteren. *On a Method of Multiprogramming*, Springer, 1999
7. Misra. *A Discipline of Multiprogramming*, Springer, 2001
8. Kourie & Watson. *The Correctness-by-Construction Approach to Programming*, Springer, 2012