Using Correctness-by-Construction to Derive Dead-zone Algorithms

Bruce Watson    Loek Cleophas    Derrick Kourie

FASTAR Research Group
Stellenbosch University & Pretoria University
South Africa

{bruce, loek, derrick}@fastar.org

Prague Stringology Conference, 1 September 2014
The journey is the reward

- Derive an iterative version of the dead-zone algorithm
  Give correctness proof
- Motivate for correctness-by-construction (CbC)
- Introduce CbC as a way of explaining algorithms
- Show how CbC can be used in inventing new one

Often in Science of Computer Programming, Elsevier Journal
Contents

1. What is CbC?
2. Problem statement
3. Intuitive solution ideas & related work
4. From positions to ranges-of-positions
5. Greater shifts
6. Representing the set of live-zones
7. Concurrency
8. Conclusions & ongoing work
What *is* CbC?

1. Start with a *specification*
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?

- Algorithm we can run
- *Derivation* showing how we got there
- Interwoven correctness proof
- ‘Tiny’ derivation steps give choices
  
  *Family* of algorithms
Problem statement

Single keyword exact pattern matching:

Given two strings $x, y \in \Sigma^*$ over an alphabet $\Sigma$ ($x$ is the pattern, $y$ is the input text) find all occurrences of $x$ as a contiguous substring of $y$.

For convenience:

$$\text{Match}(x, y, j) \equiv (x = y_{[j, j+|x|]})$$

Now we have our postcondition:

$$\text{MS} = \bigcup_{j \in [0,|y|): \text{Match}(x, y, j)} \{j\}$$

For example, $y = \text{abbaba}$ and $x = \text{ba}$ gives

$$\text{MS} = \{2, 4\}$$
Intuitive solution

Partition the indices in $y$ — i.e. set $[0, |y|)$

1. MS — a match has already been found
2. Live_Todo — we know nothing still live.
3. $\neg(MS \cup \text{Live}_\text{Todo})$ — we know no match occurs

1 and 3 together are the dead-zone
Intuitive solution (cont.)

Start with \( \text{Live}_{\text{Todo}} = [0, |y|) \) (all are live) and \( \text{MS} = \emptyset \)
... reduce to \( \text{Live}_{\text{Todo}} = \emptyset \) (all dead), i.e.
DO loops

What do we need to derive 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

Bertrand Meyer 2011 (rephrasing Edsger Dijkstra 1970)

“Publish no loop without its invariant”

DO loops

For invariant $I$ and variant expression $V$ we get

\[
\begin{align*}
\{ & P \} \\
\{ & I \} \\
\textbf{do} & \quad G \rightarrow \quad \\
& \{ \ I \land G \land \text{expression } V \ \text{has a particular value} \ \} \\
&S_0 \quad \\
& \{ \ I \land \text{expression } V \ \text{has decreased} \ \} \\
\textbf{od} \\
& \{ \ I \land \neg G \ \} \\
& \{ \ Q \ \} 
\end{align*}
\]
First algorithm

Live\_Todo := [0, |y|);
MS := ∅;
{ invariant: (∀ j : j ∈ MS : Match(x, y, j)) }
{ (∀ j : j ∉ (MS ∪ Live\_Todo) : ¬Match(x, y, j)) }
{ variant: |Live\_Todo| }
S : Some kind of loop
{ invariant ∧ |Live\_Todo| = 0 }
{ post }
Ranges of positions

Be cheap:
change Live_Todo to be a pairwise disjoint set of live ranges \([l, h]\)

Live_Todo := \([0, |y|]\);
MS := \(\emptyset\);
\{ invariant: (\(\forall j: j \in MS\) : \text{Match}(x, y, j)) \} \\
\{ \land (\(\forall j: j \not\in (MS \cup \text{Live}_\text{Todo})\) : \neg \text{Match}(x, y, j)) \} \\
\{ variant: |\text{Live}_\text{Todo}| \} \\
do Live_Todo \not= \emptyset \rightarrow 
  Extract some \([l, h]\) from Live_Todo;

  \(S_1\) : do some stuff to check matches in \([l, h]\) and update Live_Todo
od
\{ invariant \land |\text{Live}_\text{Todo}| = 0 \} \\
\{ post \}
Ranges of positions (stripped of invariant stuff)

Live\_Todo := \{[0, |y|]\};
MS := \emptyset;
do Live\_Todo \neq \emptyset \rightarrow
    Extract some \([l, h)\) from Live\_Todo;
    \[
    S_1 : \text{do some stuff to check matches in } [l, h) \text{ and update Live\_Todo}
    \]
od
\{ post \}
Ranges of positions (details)

Choose middle of a live range \( \left\lfloor \frac{l + h}{2} \right\rfloor \)
and check there (also exclude end):

Live\_Todo := \{[0, |y| − |x|)\};
MS := ∅;
\( \text{do } \) Live\_Todo \( \neq \) ∅ \( \rightarrow \)
   Extract \([l, h)\) from Live\_Todo;
   \( m := \left\lfloor \frac{l + h}{2} \right\rfloor \);
   if \ Match(x, y, m) \( \rightarrow \)
      MS := MS \cup \{m\}
   fi;
   Live\_Todo := Live\_Todo \cup [l, m) \cup [m + 1, h)
\( \text{od} \)
{ post  }

What if we insert an empty range into Live\_Todo??
Ranges of positions (details)

Live_Todo := \{[0, |y| - |x|]\};
MS := \emptyset;
do Live_Todo \neq \emptyset \rightarrow
   Extract [l, h) from Live_Todo;
   if l \geq h \rightarrow \{ empty range \} skip
   \mid l < h \rightarrow
      m := \lfloor \frac{l+h}{2} \rfloor;
      if Match(x, y, m) \rightarrow
         MS := MS \cup \{m\}
      fi;
      Live_Todo := Live_Todo \cup [l, m) \cup [m + 1, h)
   fi
od
\{ post \}
Greater shifts

We can of course use Match (or other) information to make larger window shifts

\[ l', h' := m - shl, m + shr; \]
\[ \text{Live\_Todo} := \text{Live\_Todo} \cup [l, l') \cup [h', h); \]
Representing the ‘set’ of live-zones

- Live_Todo are pairwise disjoint. . . can be done in parallel
  Simone & Thierry have presented an algorithm with similar characteristics

- Live_Todo is a set
  Extracting \([l, h]\) gives an arbitrary pair
  Very poor performance with cache misses in \(y\)

- Live_Todo can easily be represented using a queue or stack
  Breadth- or depth-wise traversals of the ranges in \(y\)
  Queue: worst case size \(|y|\), best case \(\left\lceil \frac{|y|}{|x|} \right\rceil\)
  Stack: worst case size \(\log_2|y|\)
Live_Todo as a stack

Live_Todo := ⟨[0, |y| − |x|)⟩;
MS := ∅;

\textbf{do} Live_Todo \neq ∅ \rightarrow \\
\hspace{1em} \text{Pop } [l, h) \text{ from Live_Todo;} \\
\hspace{1em} \textbf{if } l \geq h \rightarrow \{ \text{ empty range } \} \textbf{ skip} \\
\hspace{2em} | \text{ if } l < h \rightarrow \\
\hspace{3em} m := \left\lfloor \frac{l+h}{2} \right\rfloor; \\
\hspace{3em} \textbf{if } \text{ Match}(x, y, m) \rightarrow \\
\hspace{4em} \text{MS } := \text{MS } \cup \{ m \} \\
\hspace{4em} \textbf{fi}; \\
\hspace{3em} l', h' := m - \text{shl}, m + \text{shr}; \\
\hspace{3em} \text{Push } [h', h) \text{ onto Live_Todo;} \\
\hspace{3em} \text{Push } [l, l') \text{ onto Live_Todo} \\
\hspace{2em} \textbf{fi} \\
\textbf{od} \\
\{ \text{ post } \}
Optimization: L-R deadness sharing

maintain integer $z$ with invariant (such that)

$$(\forall i : 0 \leq i < z : i \text{ is dead})$$

and keep $z$ maximal, giving:

```plaintext
):
z := 0;
):
engan Endo \neq \emptyset \rightarrow
Pop [l, h) from Live_Todo;
l := l \max z;
z := l;
if \ l \geq h \rightarrow \{ \text{ empty range } \} \ skip
):
```
Concurrency: decouple match verification from shifting

\begin{verbatim}
Live_Todo := \langle[0, |y| - |x|]\rangle;
MS := \emptyset;
\textbf{do} Live_Todo \neq \emptyset \rightarrow
    \text{Pop } [l, h) \text{ from Live_Todo;}
    \textbf{if} l \geq h \rightarrow \{ \text{ empty range } \} \textbf{skip}
    \textbf{endif}
    l < h \rightarrow
        m := \lfloor \frac{l + h}{2} \rfloor;
        \text{Add } m \text{ to queue Attempt}_t \text{ for some thread } t;
        l', h' := m - shl, m + shr;
        \text{Push } [h', h) \text{ to Live_Todo;}
        \text{Push } [l, l') \text{ to Live_Todo}
\textbf{fi}
\textbf{od}
\{ \textbf{post} \}
\end{verbatim}
Conclusions & ongoing work

- Interesting new algorithm skeleton
- Performance is similar to comparable algorithms
  Not yet clear how to integrate advances in other algorithms
- CbC is robust and relatively easy
  Creativity is not hampered: new algorithms can be invented
- Useful methodology for bringing coherence to a field
  ...and detecting unexplored parts
Performance

\[(x - \text{nhh}) / \text{nhh} \times 100\]

Data Sources: i7 / Wall plug / Sequential / * / * / Bible / Machine time