Faster Searching by Elimination

Theodore S. Norvell
Electrical and Computer Engineering
Memorial University

December 26, 2010

Abstract

The SIMPLE system, under development at Memorial University, allows abstract problem descriptions to be refined by abstract algorithms. By data refining both the problem and its solution, we can reuse verified algorithms. We use binary search as an example of this method.

1 Introduction

The SIMPLE system [1], currently under development by the author, is a programming environment for interactively deriving programs from their specifications.

One of the goals is to allow the programmer to leverage abstraction to the greatest extend possible. Abstraction allows one to concentrate on some aspects of a problem while ignoring others. This paper illustrates how SIMPLE allows one to solve some aspects of algorithmic problems without solving others. We will show how the SIMPLE language allows one to separate the description and proof of an algorithmic technique from its application to particular problems and data representations. This will allow a library of abstract problems and algorithmic techniques to solve them to be built, stored, and later applied to concrete problems.

As an example, we will look at the algorithmic problem of search and the algorithmic technique of binary search. We will then apply the technique to two concrete problems.

Although SIMPLE is still under development, this paper is intended to be a realistic exposition of the capabilities that the SIMPLE language and system are expected to have, as well as an exploration of the capabilities it will be required to have in order to meet its aspirations.

2 Search and Binary Search

The search problem we will look at is that of finding a value in a set that satisfies a given description

Definition $Search$ ::=
    require $S \cap G \neq \emptyset$·
    ensure $x' \in G$

This specification consist of a precondition, $S \cap G \neq \emptyset$, that indicates that it is an assumption that a solution exists within a search space S, and a postcondition that indicates that a solution is to be placed in variable x. This specification is generic over an anonymous type for the variable x. If the type of x is α, the type of S and G is $set(\alpha)$.

As in Hehner's predicative programming approach [2, 3], each specification can be interpreted as a boolean expression relating initial and final states, specifying which behaviours are acceptable.

Of course not all search problems fit this pattern; in particular it is often not known *a priori* whether a solution exists.

The binary search algorithm can be given by

Definition *BinarySearch* ::=
  while $|S| > 1$ inv $S \cap G \neq \emptyset$
    let $S_0, S_1 \mid S = S_0 \cup S_1 \cdot$
    ($S_0 \cap G \neq \emptyset \rightarrow S := S_0$
    $[\!]$ $S_1 \cap G \neq \emptyset \rightarrow S := S_1$)
  require $|S| = 1$ ensure $S = \{x'\}$

The algorithm uses nondeterministic choice $[\!]$ and guarded commands. A guarded command, $A \rightarrow P$, ensures that its guard, A, is true before executing its body, P.

That this algorithm refines the search problem is expressed in SIMPLE as a theorem

Theorem *Search* $\sqsubseteq$ *BinarySearch*

The proof of this theorem will also be expressible in SIMPLE and the SIMPLE environment will be able to check the proof. Note that $\sqsubseteq$ expresses a partial correctness relation, so that the theorem means that any behaviour acceptable to *BinarySearch* is acceptable to *Search*.

3 Data Refinement

A data refinement [4, 5] of a specification P under an abstraction invariant I with variables v to be removed is a predicate

$$dr_{v;I}\ P = \forall v \cdot I \Rightarrow \exists v' \cdot I' \wedge P$$

Data refinement is a means of representing certain variables that occur in P with other variables. For example, if we have a statement $x := \neg x$ but we represent the boolean variable x with an integer i under the abstraction invariant $(x \wedge i = 0) \vee (\neg x \wedge i = 1)$ then the data refinement is

$$dr_{x;(x \wedge i=0) \vee (\neg x \wedge i=1)}\ \ (x := \neg x) = (i := 1 - i)$$

Now let us look at two examples of applications of the binary search problem.

4 Application

4.1 Searching a sorted array

The classic example of binary search is that of searching a sorted array.

We can define what it is for an array to be sorted

Definition *Sorted* $A ::= \forall i, j \in index(A) \cdot i \leq j \Rightarrow A\ i \leq A\ j$

Now the problem we wish to solve is searching a sorted array when we know the thing we are looking for, p, is in the array.

Definition *ArraySearch* $A\ p$ ::=

require $Sorted\ A \wedge (\exists x \in index(A) \cdot A\ i = p)$
ensure $A[x'] = p$

In order to understand the array search problem as a concretization of the abstract search problem, we need to relate their variables with an abstraction invariant. We use:

$$I = (S = \{a, ..b\} \wedge G = \{i \in index(A) \mid A\ i = p\})$$

where $\{a, ..b\}$ is the set of all integers greater or equal to a and less than b. Now data transforming Search, we get

$$ArraySearch \sqsubseteq \mathsf{require}\ Sorted\ A \cdot \mathsf{var}\ a, b := 0, length(A) \cdot\ dr_{S,G;I}(Search)$$

Where $\sqsubseteq$ represents refinement of specifications. That is $P \sqsubseteq Q$ means that any behaviour accepted by Q is accepted by P.

As data refinement is monotonic with respect to refinement we have

$$\begin{array}{rcl} ArraySearch & \sqsubseteq & \mathsf{require}\ Sorted\ A \cdot \mathsf{var}\ a, b := 0, length(A) \cdot\ dr_{S,G;I}(Search) \\ & \sqsubseteq & \mathsf{require}\ Sorted\ A \cdot \mathsf{var}\ a, b := 0, length(A) \cdot\ dr_{S,G;I}(BinarySearch) \end{array}$$

The binary search under this data refinement is further refined by

while $a + 1 < b$ inv $\langle \exists i \in \{a, ...b\} \cdot A\, i = p \rangle$
  let $S_0, S_1 \mid \{a, ..b\} = S_0 \cup S_1 \cdot$
  $(\langle \exists i \in S_0 \cdot A\, i = p \rangle \rightarrow$
    $\mathsf{change}\ a, b \cdot \mathsf{ensure}\ \{a', ..b'\} = S_0$
  $[\!]\ \langle \exists i \in S_1 \cdot A\, i = p \rangle \rightarrow$
    $\mathsf{change}\ a, b \cdot \mathsf{ensure}\ \{a', ..b'\} = S_1)$
require $a + 1 = b$ ensure $\{a, ..b\} = \{x'\}$

Now sets S_0 and S_1 can be chosen such that $S_0 = \{a, .. \lfloor \frac{a+b}{2} \rfloor\}$ and $S_1 = \{\lfloor \frac{a+b}{2} \rfloor, ..b\}$ giving algorithm

while $b - a > 1$ inv $\langle \exists i \in \{a, ...b\} \cdot A\, i = p \rangle$
  $(\langle \exists i \in \{a, .. \lfloor \frac{a+b}{2} \rfloor\} \cdot A\, i = p \rangle \rightarrow$
    $b := \lfloor \frac{a+b}{2} \rfloor$
  $[\!]\ \langle \exists i \in \{\lfloor \frac{a+b}{2} \rfloor, ..b\} \cdot A\, i = p \rangle \rightarrow$
    $a := \lfloor \frac{a+b}{2} \rfloor)$
$x := a$

From the fact that A is sorted we can obtain that if $A \lfloor \frac{a+b}{2} \rfloor > x$

$$\begin{array}{rcl} & & \langle \exists i \in \{a, ..b\} \cdot A\, i = p \rangle \\ & = & \left\langle \exists i \in \{a, .. \left\lfloor \frac{a+b}{2} \right\rfloor\} \cdot A\, i = p \right\rangle \vee \left\langle \exists i \in \{\left\lfloor \frac{a+b}{2} \right\rfloor, ..b\} \cdot A\, i = p \right\rangle \\ & = & \left\langle \exists i \in \{a, .. \left\lfloor \frac{a+b}{2} \right\rfloor\} \cdot A\, i = p \right\rangle \end{array}$$

and that if $A \lfloor \frac{a+b}{2} \rfloor \leq x$

$$
\begin{array}{ll}
 & \langle \exists i \in \{a, ..b\} \cdot A\, i = p \rangle \\
= & \left\langle \exists i \in \{a, .. \left\lfloor \frac{a+b}{2} \right\rfloor\} \cdot A\, i = p \right\rangle \vee \left\langle \exists i \in \{ \left\lfloor \frac{a+b}{2} \right\rfloor, ..b\} \cdot A\, i = p \right\rangle \\
= & \left\langle \exists i \in \{ \left\lfloor \frac{a+b}{2} \right\rfloor, ..b\} \cdot A\, i = p \right\rangle
\end{array}
$$

Hence we can strengthen the guards to $A \left\lfloor \frac{a+b}{2} \right\rfloor > x$ and $A \left\lfloor \frac{a+b}{2} \right\rfloor \leq x$ respectively, after which the body of the loop simplifies to

$$
\mathsf{if}\ A \left\lfloor \frac{a+b}{2} \right\rfloor > x\ \mathsf{then}\ b := \left\lfloor \frac{a+b}{2} \right\rfloor\ \mathsf{else}\ a := \left\lfloor \frac{a+b}{2} \right\rfloor
$$

4.2 Calculating a square root

As a second example we will derive an algorithm for finding the integer part of the square root of a natural number. The specification is

Definition *Root* y ::=
  require $y \in \mathbb{N}$
  ensure $x' = \lfloor \sqrt{y} \rfloor$

We will determine the square root bit-by-bit, starting with the left-most bit. We represent the set S using two natural numbers: x represents the bits calculated so far, while i represents the number of bits yet to be calculated. Using an abstraction invariant

$$
I = \big(S = \{x2^i, ..(x+1)2^i\} \wedge G = \{\lfloor \sqrt{y} \rfloor\}\big)
$$

We get

$$
\mathit{Root}\, y \sqsubseteq \mathsf{var}\ i := N \cdot x := 0; dr_{S,G;I}\ \mathit{Search} \sqsubseteq \mathsf{var}\ i := N \cdot x := 0; dr_{S,G;I}\ \mathit{BinarySearch}
$$

where $N \geq \lceil \log_2 \lfloor \sqrt{y} \rfloor \rceil$, i.e. N is enough bits to hold the answer.

The data refinement of the binary search algorithm gives

  while $i > 0$ inv $x2^i \leq \lfloor \sqrt{y} \rfloor < (x+1)\, 2^i$
    let $S_0, S_1 \mid \{x2^i, ..(x+1)2^i\} = S_0 \cup S_1 \cdot$
    ($\lfloor \sqrt{y} \rfloor \in S_0 \rightarrow \mathsf{change}\ x, i \cdot \mathsf{ensure}\ \{x'2^{i'}, ..(x'+1)2^{i'}\} = S_0$
    [] $\lfloor \sqrt{y} \rfloor \in S_1 \rightarrow \mathsf{change}\ x, i \cdot \mathsf{ensure}\ \{x'2^{i'}, ..(x'+1)2^{i'}\} = S_1$)

We can pick S_0 and S_1 as $\{x2^i, ..(2x+1)2^{i-1}\}$ and $\{(2x+1)2^{i-1}, ..(x+1)2^i\}$. This gives

  while $i > 0$ inv $x2^i \leq \lfloor \sqrt{y} \rfloor < (x+1)\, 2^i$
    ($\lfloor \sqrt{y} \rfloor < (2x+1)2^{i-1} \rightarrow x, i := 2x, i-1$
    [] $\lfloor \sqrt{y} \rfloor \geq (2x+1)2^{i-1} \rightarrow x, i := 2x+1, i-1$)

Now we need to refine the guards

$$\begin{array}{ll} & \lfloor\sqrt{y}\rfloor < (2x+1)2^{i-1} \\ = & \text{Property of floor.} \\ & \sqrt{y} < (2x+1)2^{i-1} \\ = & \text{Property of square root} \\ & y < \left((2x+1)2^{i-1}\right)^2 \\ = & \text{Algebra} \\ & y < 2^{2i}x^2 + 2^{2i}x + 2^{2i-2} \end{array}$$

This gives a loop body of

$$\mathsf{if}\ y < 2^{2i}x^2 + 2^{2i}x + 2^{2i-2}\ \mathsf{then}\ x, i := 2x, i-1\ \mathsf{else}\ x, i := 2x+1, i-1$$

Now this involves a number of multiplications, which we would like to get rid of. We can do this by means of another data refinement, this time to introduce variables to track the three terms in the guard.

$$J = \left(p = 2^{2i}x^2 \wedge q = 2^{2i}x \wedge r = 2^{2i-2}\right)$$

The initialization for these variables is

$$\mathsf{var}\ p, q, r := 0, 0, 2^{2N-2}$$

The test becomes $y < p + q + r$. The assignment $x, i := 2x, i-1$ is transformed to

$$\begin{bmatrix} x \\ i \\ p \\ q \\ r \end{bmatrix} := \begin{bmatrix} 2x \\ i-1 \\ 2^{2(i-1)}\,(2x)^2 \\ 2^{2(i-1)}2x \\ 2^{2(i-1)-2} \end{bmatrix}$$

which simplifies, by the abstraction invariant to

$$\begin{bmatrix} x \\ i \\ q \\ r \end{bmatrix} := \begin{bmatrix} 2x \\ i-1 \\ q/2 \\ r/4 \end{bmatrix}$$

The assignment $x, i := 2x+1, i-1$ is transformed to

$$\begin{bmatrix} x \\ i \\ p \\ q \\ r \end{bmatrix} := \begin{bmatrix} 2x \\ i-1 \\ 2^{2i}x^2 \\ 2^{2(i-1)}x \\ 2^{2(i-1)-2} \end{bmatrix}$$

which simplifies to

$$\begin{bmatrix} x \\ i \\ p \\ q \\ r \end{bmatrix} := \begin{bmatrix} 2x \\ i-1 \\ p+q+r \\ q/2+r \\ r/4 \end{bmatrix}$$

In summary we have

$$
\begin{array}{l}
\mathsf{var}\ i := N \cdot x := 0; \\
\text{while } i > 0 \text{ inv } x2^i \leq \left\lfloor \sqrt{y} \right\rfloor < (x+1)\,2^i \\
\quad \text{if } y < p + q + r \\
\quad\quad x, i, q, r := 2x, i-1, q/2, r/4 \\
\quad \text{else} \\
\quad\quad x, i, p, q, r := 2x, i-1, p+q+r, q/2+r, r/4
\end{array}
$$

All the divisions can be implemented as shifts. This algorithm is suitable for implementation in either hardware or software.

5 Conclusion and related work

The methods presented here can be extended to other problems. For example, division of integers, with the goal of a hardware suitable implementation.

The origins of this paper came with the realization that the principled strength reduction method in [6] applied to a square root algorithm could be seen as a data refinement and then the realization that the original algorithm could be seen as a data refinement of a more abstract binary-search algorithm. The resulting abstract binary search algorithm turns out to be an abstraction also of the Searching by Elimination algorithm presented in [7]. Their algorithm only eliminates a single member of the search space in each loop, yielding algorithms with time complexities linear in the search space size at best; my algorithm allows any subset of the search space to be eliminated, which is faster; hence the title of this paper.

References

[1] Theodore Norvell and Zhikai Ding, "An environment for proving and programming," in *Newfoundland Electrical and Computer Engineering Conference*, October 1999.

[2] Eric C. R. Hehner, "Predicative programming," *Communications of the ACM*, vol. 27, no. 2, pp. 134–151, 1984.

[3] Eric C. R. Hehner, "Abstractions of time," in *A Classical Mind*, A. W. Roscoe, Ed., chapter 12, pp. 195–214. Prentice-Hall International, 1994.

[4] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, pp. 271–281, 1972.

[5] David Gries and Jan Prins, "A new notion of encapsulation," in *SIGPLAN '85 Symposium on Language Issues in Programming*, 1985, pp. 131–139.

[6] Yanhong A. Liu, "Principled strength reduction," in *Algorithmic Languages and Calculi*, Richard Bird and Lambert Meertens, Eds. IFIP, February 1997, pp. 357–381, Chapman and Hall.

[7] Anne Kaldewaij and Berry Schoenmakers, "Searching by elimination," *Science of Computer Programming*, vol. 14, pp. 243–254, 1990.