Correct Programming Guided By Inductive Refinement

Kwangkeun Yi

December 2, 2009

1 Introduction

Inductive refinement is useful, even essential, in developing bug-free programs. We demonstrate this powerful programming methodology by using the regular expression match problem.

2 Background

Let Σ be an *alphabet*, that is, a finite set of *letters*. We use c to denote a letter. Σ^* is the set of finite strings over the alphabet Σ. We use s to denote a string. The null string is written ϵ. The set Σ^* is inductively defined as

$$\begin{array}{rcll} s & \rightarrow & \epsilon & \\ & \mid & c{\cdot}s & (c \in \Sigma) \end{array}$$

String concatenation of s and s' is written $s{\cdot}s'$. The empty string is the identity element for the concatenation operator, that is, $\epsilon{\cdot}s = s = s{\cdot}\epsilon$.

A *language* $\mathcal{L}$ is a subset of Σ^*. The size $|s|$ of string s is defined as $|\epsilon| = 0$ and $|c{\cdot}s| = 1 + |s|$. We use the following operations on languages:

$$\begin{array}{lcl} \mathcal{L}\,\mathcal{L}' & = & \{s{\cdot}s' \mid s \in \mathcal{L}, s' \in \mathcal{L}'\} \\ \mathcal{L}^0 & = & \{\epsilon\} \\ \mathcal{L}^{i+1} & = & \mathcal{L}\,\mathcal{L}^i \\ \mathcal{L}^* & = & \cup_{i\geq 0}\mathcal{L}^i \end{array}$$

Regular expressions are notation for languages. The set of regular expressions is inductively defined as

$$\begin{array}{rcl} r & \rightarrow & \epsilon \\ & \mid & c \\ & \mid & r\,r \\ & \mid & r+r \\ & \mid & r^* \end{array}$$

Each regular expression r denotes language $L(r)$ inductively as follows:

$$\begin{array}{rcl}
L(\epsilon) & = & \{\epsilon\} \\
L(c) & = & \{c\} \\
L(rr') & = & L(r)L(r') \\
L(r+r') & = & L(r) \cup L(r') \\
L(r^*) & = & L(r)^*
\end{array}$$

We use $L(R)$ also for a set R of regular expressions to denote $\cup_{r \in R} L(r)$. $L(\emptyset)$ is defined as $\emptyset$. The size $|r|$ of regular expression r is defined as: $|\epsilon| = |c| = 1$, $|rr'| = |r + r'| = |r| + |r'| + 1$, and $|r^*| = |r| + 1$.

3 A Regular Expression Matching Program

3.1 Problem and Specification

The regular expression matching problem is, for a string $s \in \Sigma^*$ and a regular expression r, to determine whether $s \in L(r)$.

Our goal is to program "$r\,!\,s$" that returns true iff string s matches regular expression r. The inductive specification for "$r\,!\,s$" consists of five cases, following the definition of the regular-expression grammar:

$$\begin{array}{rcll}
\epsilon\,!\,s & = & s = \epsilon & \\
c\,!\,s & = & s = c & \\
r_1 + r_2\,!\,s & = & r_1\,!\,s \ \vee \ r_2\,!\,s & (1) \\
r_1 r_2\,!\,s & = & \exists s_1 \exists s_2 : s = s_1 \cdot s_2 \ \wedge \ r_1\,!\,s_1 \ \wedge \ r_2\,!\,s_2 & (2) \\
r^*\,!\,s & = & s = \epsilon \ \vee \ (\exists s_1 \exists s_2 : s = s_1 \cdot s_2 \ \wedge \ r\,!\,s_1 \ \wedge \ r^*\,!\,s_2) & (3)
\end{array}$$

The problem is to find s's substrings s_1 and s_2 that satisfy the conditions for the last two cases.

3.2 Inductive Refinement

Our first step toward the implementation of (2) and (3) uses an inductive analysis of the string argument $s \ \rightarrow \ \epsilon \mid c \cdot s$ (for $c \in \Sigma$):

$$\begin{array}{rcll}
r^*\,!\,\epsilon & = & \mathit{True} & \\
r^*\,!\,c \cdot s & = & r' r^*\,!\,s \quad \text{for some } r' \in r\,\dagger_c & \\
 & = & \mathit{False} \ \vee \ \bigvee \{ r' r^*\,!\,s \mid r' \in r\dagger_c \} & (4) \\
 & & \text{where } L(r\dagger_c) = \{ s \mid c \cdot s \in L(r) \}. &
\end{array}$$

That is, $r\dagger_c$ denotes the set of regular expressions for the strings in r whose leading letter c has been removed.

Analyzing $r_1 r_2 \,!\, s$ proceeds along the same lines:

$$\begin{aligned} r_1 r_2 \,!\, \epsilon &= r_1 \,!\, \epsilon \;\wedge\; r_2 \,!\, \epsilon && (5) \\ r_1 r_2 \,!\, c{\cdot}s &= r_1' r_2 \,!\, s \quad \text{for some } r_1' \in r_1 \dagger_c \\ &= \mathit{False} \;\vee\; \bigvee \{ r_1' r_2 \,!\, s \mid r_1' \in r_1 \dagger_c \} && (6) \end{aligned}$$

The definition of the function $r\dagger_c$ again follows the definition of the regular-expression grammar:

$$\begin{aligned} \epsilon\dagger_c &= \emptyset \\ c'\dagger_c &= \emptyset \quad (c \neq c') \\ c\dagger_c &= \{\epsilon\} \\ r_1 + r_2 \dagger_c &= r_1 \dagger_c \;\cup\; r_2 \dagger_c && (7) \\ r_1 r_2 \dagger_c &= \{ r_1' r_2 \mid r_1' \in r_1 \dagger_c \} && (8) \\ r^* \dagger_c &= \{ r' r^* \mid r' \in r \dagger_c \} && (9) \end{aligned}$$

Are the definitions of $r \,!\, s$ and $r\dagger_c$ correct?

3.2.1 Checking $r\dagger_c$ Correct

First, the termination of $r\dagger_c$ is clear, because arguments to the recursive calls follow a well-founded order: every regular expression argument to recursive callees is smaller than that of the caller. From a finite regular expression there is no infinitely decreasing chain.

We have to check that our definition of $r\dagger_c$ satisfies the specification:

$$L(r\dagger_c) = \{ s \mid c{\cdot}s \in L(r) \}.$$

Alah, the case $r_1 r_2 \dagger_c$ has an error. By its definition (Eq. (8)), $L(r_1 r_2 \dagger_c)$ is $L(r_1 \dagger_c) L(r_2)$. Is this set equal to $\{ s \mid c{\cdot}s \in L(r_1) L(r_2) \}$? Unfortunately it is not; for example, in the case $L(r_1) = \{\epsilon\}$ and $c{\cdot}s \in L(r_2)$, the latter set is nonempty, while the former set is empty.

What to do? To fix the problem is to inductively refine the definition one step further. Because the check naturally suggests the consideration of the cases for r_1, we refine the definition of $r_1 r_2 \dagger_c$ by the inductive sub-cases for r_1:

$$\begin{aligned} c r_2 \dagger_c &= \{ r_2 \} \\ c' r_2 \dagger_c &= \emptyset \quad (c \neq c') \\ \epsilon r_2 \dagger_c &= r_2 \dagger_c && (10) \\ (r_{1_1} r_{1_2}) r_2 \dagger_c &= r_{1_1} (r_{1_2} r_2) \dagger_c && (11) \\ (r_{1_1} + r_{1_2}) r_2 \dagger_c &= r_{1_1} r_2 \dagger_c \;\cup\; r_{1_2} r_2 \dagger_c && (12) \\ r_1^* r_2 \dagger_c &= r_2 \dagger_c \;\cup \{ r' r_1^* r_2 \mid r' \in r_1 \dagger_c \} && (13) \end{aligned}$$

The cases where $L(r_1)$ can have ϵ are handled either by case analysis or by recursive calls.

Now about the termination of the new definition. Because of Eq. (11) we have to find a different well-founded order for the recursive calls. Because, for the recursive call $r_{1_1}(r_{1_2}r_2)\dagger_c$ from $(r_{1_1}r_{1_2})r_2\dagger_c$, the regular expression's left-hand side ($\mathit{Left}(rr') \overset{\text{let}}{=} r$) is decreasing, the arguments to recursive calls follow the order

$$
\begin{array}{lll}
(r,c) > (r',c) & & \\
\quad \text{iff} & |r| > |r'| & \text{(for Eq. (7),(9),(10),(12),(13))} \\
\quad \text{or} & (|r| = |r'| \ \wedge\ |\mathit{Left}(r)| > |\mathit{Left}(r')|) & \text{(for Eq. (11))}
\end{array}
$$

The order is well-founded; there is no infinitely decreasing chain from finite (r, c).

3.2.2 Checking $r\,!\,s$ Correct

First, the termination of $r\,!\,s$ is clear, because the arguments to recursive calls follow a well-founded order:

$$
\begin{array}{lll}
(r,s) > (r',s') & & \\
\quad \text{iff} & |s| > |s'| & \text{(for Eq. (4),(6))} \\
\quad \text{or} & (|s| = |s'| \ \wedge\ |r| > |r'|) & \text{(for Eq. (1),(5))}
\end{array}
$$

There is no infinitely decreasing chain from finite (r, s).

Unfortunately, the case of $r_1 r_2\,!\,c{\cdot}s$ has an error (Eq. (6)). When it returns false, it means that, by its definition, $r\dagger_c = \emptyset$ or $\forall r' \in r\dagger_c : r'_1 r_2\,!\,s = \mathit{False}$. Consider the case $r\dagger_c = \emptyset$, which means that $c{\cdot}s \notin L(r_1)$. Can we conclude, from this, that $c{\cdot}s \notin L(r_1 r_2)$? No, because if $\epsilon \in L(r_1)$ and $c{\cdot}s \in Lr_2$, it is possible that $c{\cdot}s \in L(r_1 r_2)$.

To fix this problem we refine the inductive definition by one more step. We refine the definition of $r_1 r_2\,!\,c{\cdot}s$ by analyzing the five inductive sub-cases for r_1:

$$
\begin{array}{rcll}
\epsilon r_2\,!\,c{\cdot}s & = & r_2\,!\,c{\cdot}s & (14) \\
c r_2\,!\,c{\cdot}s & = & r_2\,!\,s & (15) \\
(r_{1_1} r_{1_2}) r_2\,!\,c{\cdot}s & = & r_{1_1}(r_{1_2} r_2)\,!\,c{\cdot}s & (16) \\
(r_{1_1} + r_{1_2}) r_2\,!\,c{\cdot}s & = & r_{1_1} r_2\,!\,c{\cdot}s \vee r_{1_2} r_2\,!\,c{\cdot}s & (17) \\
r_1^* r_2\,!\,c{\cdot}s & = & r_2\,!\,c{\cdot}s \vee \bigvee\{r'(r_1^* r_2)\,!\,s \mid r' \in r_1\dagger_c\} & (18)
\end{array}
$$

The termination is easy to see, because arguments to recursive calls follow the well-founded order :

$$
\begin{array}{lll}
(r,s) > (r',s') & & \\
\quad \text{iff} & |s| > |s'| & \text{(for Eq. (4),(15),(18))} \\
\quad \text{or} & (|s| = |s'| \ \wedge\ |r| > |r'|) & \text{(for Eq. (1),(5),(14),(17),(18))} \\
\quad \text{or} & (|s| = |s'| \ \wedge\ |r| = |r'| \ \wedge\ |\mathit{Left}(r)| > |\mathit{Left}(r')|) & \text{(for Eq. (16))}
\end{array}
$$

Figure 1 displays the complete and provably correct definition of our pattern matching program.

$$\begin{array}{rcl}
\epsilon\,!\,s & = & s = \epsilon \\
c\,!\,s & = & s = c \\
r_1 + r_2\,!\,s & = & r_1\,!\,s \;\vee\; r_2\,!\,s \\
r^*\,!\,\epsilon & = & \mathit{True} \\
r^*\,!\,c{\cdot}s & = & \mathit{False} \;\vee\; \bigvee\{r'r^*\,!\,s \mid r' \in r\dagger_c\} \\
r_1 r_2\,!\,\epsilon & = & r_1\,!\,\epsilon \;\wedge\; r_2\,!\,\epsilon \\
\epsilon r_2\,!\,c{\cdot}s & = & r_2\,!\,c{\cdot}s \\
c r_2\,!\,c{\cdot}s & = & r_2\,!\,s \\
(r_{1_1} r_{1_2}) r_2\,!\,c{\cdot}s & = & r_{1_1}(r_{1_2} r_2)\,!\,c{\cdot}s \\
(r_{1_1} + r_{1_2}) r_2\,!\,c{\cdot}s & = & r_{1_1} r_2\,!\,c{\cdot}s \;\vee\; r_{1_2} r_2\,!\,c{\cdot}s \\
r_1^* r_2\,!\,c{\cdot}s & = & r_2\,!\,c{\cdot}s \;\vee\; \bigvee\{r'(r_1^* r_2)\,!\,s \mid r' \in r_1\dagger_c\}
\end{array}$$

$$\begin{array}{rcl}
\epsilon\dagger_c & = & \emptyset \\
c'\dagger_c & = & \emptyset \quad (c \neq c') \\
c\dagger_c & = & \{\epsilon\} \\
r_1 + r_2\dagger_c & = & r_1\dagger_c \;\cup\; r_2\dagger_c \\
r^*\dagger_c & = & \{r'r^* \mid r' \in r\dagger_c\} \\
c r_2\dagger_c & = & \{r_2\} \\
c' r_2\dagger_c & = & \emptyset \quad (c \neq c') \\
\epsilon r_2\dagger_c & = & r_2\dagger_c \\
(r_{1_1} r_{1_2}) r_2\dagger_c & = & r_{1_1}(r_{1_2} r_2)\dagger_c \\
(r_{1_1} + r_{1_2}) r_2\dagger_c & = & r_{1_1} r_2\dagger_c \;\cup\; r_{1_2} r_2\dagger_c \\
r_1^* r_2\dagger_c & = & r_2\dagger_c \cup \{r' r_1^* r_2 \mid r' \in r_1\dagger_c\}
\end{array}$$

Figure 1: Correct implementation of $r\,!\,s$ and $r\dagger_c$

4 Conclusion

Inductive refinement from the sketch of algorithms to their completion is useful, even essential, in developing correct programs. Errors are fixed by refining the algorithms by repeatedly applying inductive analysis to a function's input structure.

This approach is powerful enough to guide us to arrive at practical as well as correct programs. See [**?**].