JFP **16** (6): 663–670, 2006.
doi:10.1017/S0956796806006149 First published online 14 September 2006 Printed in the United Kingdom

EDUCATIONAL PEARL

'Proof-directed debugging' revisited for a first-order version

KWANGKEUN YI

School of Computer Science and Engineering, Seoul National University, Seoul, Korea
(*e-mail:* kwang@ropas.snu.ac.kr)

Abstract

Some 10 years ago, Harper illustrated the powerful method of proof-directed debugging for developing programs with an article in this journal. Unfortunately, his example uses both higher-order functions and continuation-passing style, which is too difficult for students in an introductory programming course. In this pearl, we present a first-order version of Harper's example and demonstrate that it is easy to transform the final version into an efficient state machine. Our new version convinces students that the approach is useful, even essential, in developing both correct and efficient programs.

1 Introduction

Harper (Harper, 1999) demonstrated the power of inductive reasoning in developing correct programs. To illustrate the principle, he used the example of regular-expression matching. Unfortunately, his example used functions in higher-order continuation-passing style (CPS), making it inaccessible to beginning students.

If we wish to teach the principles and usefulness of inductive reasoning to a wide audience early in the curriculum, we must develop a simple version of the example. Ideally, the new version should rely only on first-order functions. Even more importantly, we must also address the efficiency of the functions. For beginning students, the gap between Harper's example with higher-order functions and the conventional regular expression matcher based on finite-state machines (Aho *et al.*, 1986) is just too large.

In this education pearl, we show how to overcome both problems. The resulting functions are first-order and the development uses nothing but inductive reasoning. The students in our first-year undergraduate programming course can readily follow the program and its development via proof-directed debugging. Furthermore, we also show how to reformulate the resulting functions as finite-state machines whose time complexity is linear in the input string size. In short, our new version can

convince students that the idea is useful, even essential, in developing both correct and efficient programs.[1]

The rest of the pearl proceeds as follows. Section 2 introduces the necessary background, section 3 the problem statement and a first solution. Following Harper, the correctness proof fails and exposes errors. We fix the errors with a program refinement that repeatedly applies inductive reasoning to the inputs. Finally, section 4 shows that it is straightforward to transform the proven first-order program into a finite-state machine that matches the string input in time linear in the string size.

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

$$\begin{array}{rcl} s & \rightarrow & \epsilon \\ & | & c{\cdot}s \quad (c \in \Sigma) \end{array}$$

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* $\mathscr{L}$ 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:

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

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

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

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 $|r^*| = |r| + 1$.

[1] This pearl is not about teaching the transformation (Wand, 1980) of CPS into accumulator-style programs. Our point is to illustrate how to directly arrive at "conventional" first-order programs by means of only inductive reasoning.

3 A regular expression matching algorithm

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 $s \in L(r)$.

The inductive specification for "$r\,!\,s$", which is true iff string s matches regular expression r, consists of five cases, following the definition of the regular-expression grammar:

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

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 $c \in \Sigma$):

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

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? The natural way to approach the proofs is to proceed by induction. We first prove and debug the definition of $r\,\dagger_c$ and then that of $r\,!\,s$.

3.2.1 Inductive proof for $r\dagger_c$

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.

Our proof proceeds by induction on the same order, that is on the size $|r|$ of r. In proving the correctness of $r\dagger_c$, the inductive hypothesis is that $r'\dagger_c$ is correct for every r' such that $|r'| < |r|$. Our proof goal is to show that our definition of $r\dagger_c$ satisfy the specification:

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

Base cases: $\epsilon\dagger_c$ and $c'\dagger_c$ when $c' \neq c$ must be $\emptyset$ because $L(\epsilon)$ or $L(c')$ has no string starting with c. Case $c\dagger_c$ must be $\{\epsilon\}$ because $L(c) = \{c\}$.

Inductive cases: By $r_1 + r_2\dagger_c$'s definition (Eq. (7)), $L(r_1 + r_2\dagger_c)$ is equal to $L(r_1\dagger_c) \cup L(r_2\dagger_c)$. It follows from the hypothesis (applied to r_1 and r_2) that

$$\begin{aligned} L(r_1\dagger_c) \cup L(r_2\dagger_c) &= \{s \mid c{\cdot}s \in L(r_1)\} \cup \{s \mid c{\cdot}s \in L(r_2)\} \\ &= \{s' \mid c{\cdot}s' \in L(r_1 + r_2)\}. \end{aligned}$$

Applying the $r^*\dagger_c$'s definition (Eq. (9)), $L(r^*\dagger_c)$ is $L(r\dagger_c)L(r^*)$. By inductive hypothesis to r yields:

$$\begin{aligned} L(r\dagger_c)L(r^*) &= \{s \mid c{\cdot}s \in L(r)\}L(r^*) \\ &= \{s' \mid c{\cdot}s' \in L(r)L(r^*)\} \\ &= \{s'' \mid c{\cdot}s'' \in L(r^*)\}. \end{aligned}$$

What about $r_1r_2\dagger_c$? By its definition (Eq. (8)), $L(r_1r_2\dagger_c)$ is $L(r_1\dagger_c)L(r_2)$. By inductive hypothesis applied to r_1, it is $\{s \mid c{\cdot}s \in L(r_1)\}L(r_2)$. Is this set equal to $\{s \mid c{\cdot}s \in L(r_1)L(r_2)\}$ so that we can conclude our proof? 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? Following Harper (Harper, 1999), we may leave the definition as it is and assume a pre-processed input r, such that for every r_1r_2 occurring in r, $\epsilon \notin L(r_1)$.

Another way to fix the problem is to inductively refine the definition one step further. Because the proof failure naturally suggests the consideration of the cases for r_1, we refine the definition of $r_1r_2\dagger_c$ by the inductive sub-cases for r_1:

$$\begin{aligned} cr_2\dagger_c &= \{r_2\} \\ c'r_2\dagger_c &= \emptyset \quad (c \neq c') \\ \epsilon r_2\dagger_c &= r_2\,\dagger_c && (10) \\ (r_{1_1}r_{1_2})r_2\dagger_c &= r_{1_1}(r_{1_2}r_2)\,\dagger_c && (11) \\ (r_{1_1} + r_{1_2})r_2\dagger_c &= r_{1_1}r_2\,\dagger_c \cup r_{1_2}r_2\,\dagger_c && (12) \\ r_1^*r_2\dagger_c &= r_2\,\dagger_c \cup \{r'r_1^*r_2 \mid r' \in r_1\dagger_c\} && (13) \end{aligned}$$

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

Now a new problem arises with Eq. (11) because we can no longer induct solely on the size of r. 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 ($Left(rr') \stackrel{\text{let}}{=} r$) is decreasing, the arguments to recursive calls follow the order

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

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

Hence, our correctness proof proceeds by induction on the new order: the induction hypothesis in proving $r\dagger_c$ is that $r'\dagger_c$ is correct for every $(r',c) < (r,c)$. The proof proceeds smoothly, following the same pattern of reasoning as before.

3.2.2 *Inductive proof for* $r\,!\,s$

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).

Our proof proceeds by induction on this order. In proving $r\,!\,s$, the induction hypothesis is that $r'\,!\,s'$ is correct for every $(r',s') < (r,s)$. Our proof goal is to show two things:

$$r\,!\,s = True \implies s \in L(r) \qquad \text{and} \qquad r\,!\,s = False \implies s \notin L(r).$$

The base cases are trivial. Consider the inductive cases. If $r^*\,!\,c\cdot s$ returns true, $\exists r' \in r\dagger_c : r'r^*\,!\,s = True$ (by Eq. (4)). By inductive hypothesis applied to $(r'r^*, s)$, the assertion is equal to $\exists r' \in r\dagger_c : s \in L(r'r^*) = True$. Thus,

$$\begin{array}{rcll}c\cdot s & \in & c\cdot L(r'r^*) & \\ & \in & c\cdot L(r')L(r^*) & \\ & \subseteq & L(r)L(r^*) & \text{since } r' \in r\dagger_c \\ & \subseteq & L(r^*). & \end{array}$$

If $r^*\,!\,c{\cdot}s$ is false, then by its definition (Eq. (4)), $r\dagger_c = \emptyset$ or $\forall r' \in r\dagger_c : r'r^*\,!\,s = False$. If $r\dagger_c = \emptyset$, then by its correctness, $\emptyset = L(r\dagger_c) = \{s \mid c\cdot s \in L(r)\}$, i.e., $c\cdot s \notin L(r)$ hence $c\cdot s \notin L(r^*)$. If $\forall r' \in r\dagger_c : r'r^*\,!\,s = False$, then by inductive hypothesis applied to s, $\forall r' \in r\dagger_c : s \notin L(r'r^*)$. This means by the correctness of $r\dagger_c$ that $s \notin L(r\dagger_c)L(r^*)$. Hence $c\cdot s \notin \{c\}L(r\dagger_c)L(r^*)$. This means that $c\cdot s$ has no common prefix with $L(r)$'s strings with leading c. That is, $c\cdot s \notin L(r)L(r^*)$. Thus $c\cdot s \notin L(r^*)$.

What about $r_1 r_2 ! c \cdot s$ (Eq. (6))? When it returns true, the proof proceeds by a similar pattern of reasoning without a problem. When it returns false, it means that, by its definition, $r_1 \dagger_c = \emptyset$ or $\forall r_1' \in r_1 \dagger_c : r_1' r_2 ! s = \mathit{False}$. Consider the case $r_1 \dagger_c = \emptyset$, 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 $c \cdot s \in L(r_2)$, it is possible that $c \cdot s \in L(r_1 r_2)$.

We also fix this problem with a refinement of the inductive definition with one more step, instead of transforming the given regular expression into its (equivalent) standard form. We refine the definition of $r_1 r_2 ! c \cdot s$ by analyzing the five inductive sub-cases for r_1:

$$\epsilon r_2 ! c \cdot s = r_2 ! c \cdot s \tag{14}$$

$$c r_2 ! c \cdot s = r_2 ! s \tag{15}$$

$$(r_{1_1} r_{1_2}) r_2 ! c \cdot s = r_{1_1} (r_{1_2} r_2) ! c \cdot s \tag{16}$$

$$(r_{1_1} + r_{1_2}) r_2 ! c \cdot s = r_{1_1} r_2 ! c \cdot s \vee r_{1_2} r_2 ! c \cdot s \tag{17}$$

$$r_1^* r_2 ! c \cdot s = r_2 ! c \cdot s \vee \bigvee \{ r' (r_1^* r_2) ! s \mid r' \in r_1 \dagger_c \} \tag{18}$$

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

$(r, s) > (r', s')$

iff $|s| > |s'|$ (for Eq. (4),(15),(18))

or $(|s| = |s'| \wedge |r| > |r'|)$ (for Eq. (1),(5),(14),(17),(18))

or $(|s| = |s'| \wedge |r| = |r'| \wedge |\mathit{Left}(r)| > |\mathit{Left}(r')|)$ (for Eq. (16))

Our correctness proof proceeds by induction on the order. The induction hypothesis in proving $r ! s$ is that $r' ! s'$ is correct for every $(r', s') < (r, s)$. The induction proof proceeds without a problem, following the same pattern of reasoning as before.

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

4 Performance improvement by automaton construction

Note that the worst-case number of recursive calls during $r ! s$ is exponential. Its recursive calls span a tree, because it sometimes invokes two or more recursive calls. The depth of the call tree is the length of the decreasing chain from the initial input (r, s). By the definition of the order $(r', s') < (r, s)$ of recursive call arguments, the length of the chain is proportional to the sum of $|s|$ and a quantity proportional to $|r|$. Hence the number of recursive calls (i.e., nodes in the call tree) is exponential to this length.

From the proven code, how can we achieve an efficient version like the automaton-based matcher (Aho *et al.*, 1986)? By using the $r \dagger_c$ function. Given r, we can easily build a finite state machine that decides on $s \in r$ in time linear in $|s|$. The automaton's states are regular expressions, one expression per state, denoting that the machine at state r expects strings in $L(r)$. The machine's starting state is thus the input regular expression r, and its accepting states are those regular expressions whose languages contain ϵ. From each state r, we compute $r \dagger_c$ for each $c \in \Sigma$ and draw an edge

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

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

Fig. 1. Correct implementation of $r ! s$ and $r \dagger_c$.

labeled c to r' whenever $r' \in r \dagger_c$. The rationale behind this edge construction is obvious from the definition of $r \dagger_c$: each regular expression in $r \dagger_c$ expects c-prefixed strings of $L(r)$, but with the c being removed. We apply this procedure until no longer possible, transitively closing the states and edges. (This construction in a more general setting that covers the intersection and the complement operators is also presented in (Brzozowski, 1964) and later in (Berry & Sethi, 1986) with an efficiency improvement.)

This automaton construction is finite. There are only finitely many states (regular expressions). A formal proof in a more general setting is in (Brzozowski, 1964), yet from our definition it is easy to see this finiteness. By $r \dagger_c$'s definition, the generated regular expression is always either ϵ, a sub-part of r, or an expanded one from r. But, because the expansion occurs only when r has a leading r_1^* and the expansion is to prefix the input $r_1^* r_2$ by $r' \in r_1 \dagger_c$, the number of the transitive expansions is bounded by the number of nested stars in r_1^*. The newly expanded one $r' r_1^* r_2$ cannot keep having a leading $\bullet^*$ forever, because the prefix r' is from r_1, one with the outermost star of r_1^* removed.

For example, for regular expression $a^* ab$ with alphabet $\{a, b\}$, the following states and edges are constructed (Figure 2):

- $a^* ab$ with a goes to $\epsilon a^* ab$ and b, because

$$a^* ab \dagger_a = ab \dagger_a \cup \{ r a^* ab \mid r \in a \dagger_a \} = \{ b, \epsilon a^* ab \}.$$

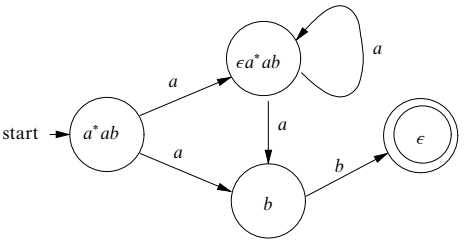

Fig. 2. Non-deterministic automaton constructed from a^*ab by the daggar ($r\dagger_c$) operator.

- ϵa^*ab with a goes to b and itself, because

$$\epsilon a^*ab\dagger_a = a^*ab\dagger_a = \{b, \epsilon a^*ab\}.$$

- b with b goes to ϵ, because $b\dagger_b = \{\epsilon\}$.

A non-deterministic automaton can always be transformed into a deterministic one by the standard subset construction (Aho *et al.*, 1986).

5 Conclusion

Our educational pearl reformulates Harper's example of proof-directed debugging so that it becomes accessible to first-year students:

- We introduce a first-order version of the regular-expression matcher.
- We use inductive reasoning throughout program development: from the sketch of the algorithms to their correct completion. That is, refining the algorithms by repeatedly applying inductive analysis to a function's input structure fixes the errors.
- We are left with a small gap between theory and practice because it is easy to derive an efficient finite-state machine from the proven code. Based on our experience, this close connection to practice helps convince students of the practicality of the approach.

Acknowledgements

We thank especially Matthias Felleisen and anonymous referees for their very kind and detailed comments on our submission, and also Bob McKay, who gave helpful suggestions on an earlier draft.

References

Aho, A. V., Sethi, R. & Ullman, J. D. (1986) *Compilers: Principles, techniques, and tools.* Addison-Wesley.

Berry, G. & Sethi, R. (1986) From regular expressions to deterministic automata. *Theor. Comput. Sci.* **48**(1), 117–126.

Brzozowski, J. A. (1964) Derivatives of regular expressions. *J. ACM*, **11**(4), 481–494.

Harper, R. (1999) Proof-directed debugging. *J. Funct. Program.* **9**(4), 463–469.

Wand, M. (1980) Continuation-based program transformation strategies. *J. ACM*, **27**(1), 164–180.