Reasoning about string-based runtime code generation

Nathaniel Charlton

School of Informatics, University of Sussex
n.a.charlton@sussex.ac.uk

Abstract. Many programming languages, such as JavaScript and Perl, support string-based runtime code generation. This facility allows programs to build up commands, represented as strings, at runtime. To parse and execute such commands a special *eval* statement is used. Though *eval* is often regarded with suspicion, it is widely used in practise, especially for web programming. We present a Hoare logic for reasoning about non-trivial uses of *eval* in a simple programming language. This is to our knowledge the first such logic. Our main idea is to use a set of annotated grammar productions to represent or overapproximate the set of strings which a program can generate as it runs. We then provide proof rules for induction on the grammar productions, allowing us to prove behavioural specifications about sets of strings representing commands. We demonstrate our approach with example proofs.

1 Introduction

Many programming languages, such as JavaScript and Perl, provide a special *eval* statement, whose function is to parse an input string into a program of that language, and then execute it. This *eval* facility allows runtime code generation: at runtime programs can build up commands, represented as strings, and then execute them. Though *eval* is often regarded with suspicion, its use is widespread in practise, especially for web programming [16, 15]. Recent research has found that "nearly every site that uses JavaScript also uses *eval*" [16], and that some uses of *eval* "are industry best practises, such as JSON, and asynchronous content and library loading" [15].

However, *eval* must be treated with care: if user-provided strings are not appropriately checked before inclusion in an argument to *eval*, attackers may be able to execute arbitrary code remotely (an *eval injection* attack, e.g. [1]). Also, *eval* can be used to hide harmful code from systems which attempt static detection of malicious behaviour [7].

Given the widespread nature of string-based runtime code generation and *eval*, and its security implications, the matter warrants a proper treatment within Hoare logic. This paper presents a Hoare logic for reasoning about non-trivial uses of *eval* for a simple programming language, to our knowledge the first such logic.

Program P1:

$s :=$ “$a := 1$”;
$m := n$;
(while $m \neq 0$ do
  (if $s =$ “$a := 1$” then
    $s :=$ “$a := x$”
  else
    $s :=$ “(” $+\!\!+$ s $+\!\!+$ “; $a := a \times x$)”) ;
  $m := m - 1$);
eval s

Program P2:

$s := S_0$; $n := 10$; eval s

where S_0 is the following string:

“if $(n = 0)$ then $r := 1$
else
  ((((($n := (n - 1)$;
  eval s);
  $n := (n + 1)$);
  $r := (r \times n)$);
  $s :=$ (((((((“if $(n =$” $+\!\!+$ IntToStr(n)) $+\!\!+$ “)”) $+\!\!+$ “then”) $+\!\!+$ “$r :=$”)
    $+\!\!+$ IntToStr(r)) $+\!\!+$ “else”) $+\!\!+$ s))”

Fig. 1. Motivating examples: an exponential calculation program, and a self-modifying factorial program. Quotes appearing inside quotes are shaded.

Our main idea is to use a set of annotated grammar productions to represent (or overapproximate) the set of strings which a program can generate as it runs. We then provide proof rules for induction on the grammar productions, allowing us to prove behavioural specifications about sets of strings representing commands.

The rest of this paper is structured as follows. In **Section 2** we motivate our problem with two example programs and we explain, in informal terms, our main idea. **Section 3** presents the simple programming language we work with, supporting *eval*. **Section 4** gives the judgements and proof rules for our Hoare logic, including rules for reasoning about *eval*. In **Section 5** we demonstrate the use of our logic, showing how to prove our two motivating example programs. **Section 6** gives the semantics of our judgements and **Section 7** establishes the soundness of our proof rules. **Section 8** discusses related and future work.

2 Main idea and motivating example

Consider the program P1 in Fig. 1, which, given inputs $x \in \mathbb{Z}$ and $n \in \mathbb{N}$, calculates x^n in a. Instead of performing the multiplications directly, the program constructs code, in its string representation, which performs the instruction $a :=$

$a \times x$ repeatedly, n times. This string is run by means of a statement $\mathsf{eval}\ e$, which parses the string expression e into a program statement and then executes it.[1]

How can we reason statically about this program? The immediate difficulty is that the contents of s when the $\mathsf{eval}\ s$ statement is reached is not a fixed string, but depends on what execution path has been taken through the program — which in this case is determined by the input n. In fact, the set of strings which s might contain when the $\mathsf{eval}\ s$ statement is reached is countably infinite.

A natural idea is to use symbolic values to represent (or conservatively over-approximate) infinite sets of concrete values such as strings. There are two problems we need to solve if we are to use this approach:

1. We need a way to symbolically represent or over-approximate infinite sets of strings
2. We need a way to reason about the behaviour of using eval to run a string from such a symbolically represented set of strings

We will represent sets of strings using inductively defined predicates, which we shall specify using a grammar production notation. For our example program, the following grammar over-approximates[2] the strings that can be in s when the $\mathsf{eval}\ s$ statement is reached.

$$\begin{array}{lcll} \mathrm{P(n)} & \leftarrow & \text{“}a := 1\text{”} & \mathrm{n} = 0 \\ \mathrm{P(n)} & \leftarrow & \text{“}a := x\text{”} & \mathrm{n} = 1 \\ \mathrm{P(n)} & \leftarrow & \text{“(”}\ \mathrm{P(n{-}1)}\ \text{“; }a := a \times x\text{)”} & \end{array} \tag{1}$$

In general, a symbol P appearing in the grammar with k arguments corresponds to a $k+1$-ary predicate, with the extra initial argument being the string under consideration. Thus an equivalent definition in conventional logical notation is:

$$\begin{array}{lll} \mathrm{P(s,n)} & := & (\mathrm{s} = \text{“}a := 1\text{”} \wedge \mathrm{n} = 0) \ \vee\ (\mathrm{s} = \text{“}a := x\text{”} \wedge \mathrm{n} = 1) \\ & & \vee\ \exists \mathrm{s}_1.\ \mathrm{s} = \text{“(”} +\!\!+ \mathrm{s}_1 +\!\!+ \text{“; }a := a \times x\text{)”} \wedge \mathrm{P}(\mathrm{s}_1, \mathrm{n}{-}1) \end{array}$$

(Because our grammars contain non-terminals with arguments, and equality constraints, they are not strictly context-free, and are more like the *data-dependent* grammars considered in [10].)

Having used a grammar to represent the set of strings, how can we reason about the effect of using eval to run a string from the set? We would like to show

$$\text{for all strings } l \text{ and all } \mathrm{n} \in \mathbb{N}, \text{ if } \mathrm{P}(l, \mathrm{n}) \text{ then } \{\mathit{true}\}\, l\, \{a = x^{\mathrm{n}}\} \tag{2}$$

Our idea is to use induction on the grammar. We will develop a proof rule which breaks down the proof of (2) into the following three obligations, corresponding to the three grammar productions defining P:

$$\mathrm{n} = 0 \vdash \{\mathit{true}\}\ \text{“}a := 1\text{”}\ \{a = x^{\mathrm{n}}\}$$

[1] This example is based on a similar one from [8].

[2] For instance, the grammar generates “$(a := 1; a := a \times x)$” which is never produced by the program.

$$\mathrm{n} = 1 \vdash \{\mathit{true}\}\ \text{“}a := x\text{”}\ \{a = x^{\mathrm{n}}\}$$

$$\{\mathit{true}\}\ \mathrm{p}\ \{a = x^{\mathrm{n}-1}\}\ \vdash\ \{\mathit{true}\}\ \text{“(”} +\!\!+\ \mathrm{p} +\!\!+\ \text{“; }a := a \times x\text{)”}\ \{a = x^{\mathrm{n}}\}$$

The first two proof obligations form base cases for the induction; the assumption on the left of the turnstile reflects the fact that we may make use of the constraint $\mathrm{n} = 0$ (resp. $\mathrm{n} = 1$) when proving the triple on the right. The third proof obligation forms an inductive case for the proof: informally the assumption on the left of $\vdash$ allows one to assume that (programs represented by) strings obtained from the grammar in derivations of depth k “work correctly”, while proving that strings obtained in derivations of depth $k + 1$ “work correctly”.

Note that because we are reasoning about parsing, in Fig. 1 we have been strict about including all necessary parentheses inside quoted strings; when working with ordinary (non-quoted) programs we can omit parentheses where this causes no trouble.

The second program in Fig. 1 uses a self-modifying factorial program to calculate 10!. As the program runs, s contains an increasingly large cascade of if-then-else statements, testing the input against all inputs seen to so far. The final else clause, which is reached for previously unseen inputs, calculates the output value recursively and adds another if-then-else to the cascade.

Two features of the factorial program P2 make its proof more difficult than that for the exponential program P1. Firstly, whereas P1 only assembled code from constant strings, P2 uses string expressions which depend on variables, in particular the expression $\mathrm{IntToStr}(n)$ which evaluates to the string representation of the current value of variable n. Secondly, P2 uses a kind of implicit recursion, which has been called *recursion through the store*, to calculate factorials: when the final eval s statement executes, s contains a string in which “eval s” appears, leading to a recursive invocation of the code in s.

In the rest of this paper, we develop and demonstrate a Hoare logic which allows one to carry out the kind of grammar-based induction argument described here, including reasoning about $\mathrm{IntToStr}(e)$ and implicit recursion.

3 The programming language and its semantics

In this section we define a simple imperative programming language featuring string manipulation operations and, crucially, the eval command needed to parse and execute string representations of program statements. We then give the language's semantics using a small-step execution relation.

3.1 Syntax

Variables. We introduce two countably infinite disjoint sets of names: program variables $x, y, \ldots \in \mathsf{Var}$, and auxiliary variables $\mathrm{p}, \mathrm{q}, \ldots \in \mathsf{AuxVar}$.

Strings and tokens. For simplicity we work with an idealised notion of strings: our strings are finite sequences of *tokens* rather than characters, as defined in Fig. 2. Note that a quoted string is treated as a single token (making the

$$
\begin{array}{lll}
\mathit{token} & ::= & \mathsf{NopTk} \mid \mathsf{AssignTk} \\
& & \mid \mathsf{LParenTk} \mid \mathsf{RParenTk} \mid \mathsf{SemiTk} \\
& & \mid \mathsf{IfTk} \mid \mathsf{ThenTk} \mid \mathsf{ElseTk} \mid \mathsf{EvalTk} \mid \mathsf{WhileTk} \mid \mathsf{DoTk} \\
& & \mid \mathsf{IntLitTk}\ n \mid \mathsf{StrLitTk}\ l \mid \mathsf{VarTk}\ v \\
& & \mid \mathsf{PlusTk} \mid \mathsf{MinusTk} \mid \mathsf{TimesTk} \mid \mathsf{EqualsTk} \mid \mathsf{NotEqTk} \mid \mathsf{PlusPlusTk} \mid \mathsf{IntToStrTk}
\end{array}
$$

where l is a string literal, $n \in \mathbb{Z}$ and $v \in \mathsf{Var}$

$$\mathsf{Str} := \mathit{token}\ \mathrm{list}$$

Fig. 2. Definitions of tokens and strings.

$$
\begin{array}{llll}
\text{expressions} & e & ::= & 0 \mid 1 \mid \ \ldots\ \mid e_1 + e_2 \mid e_1 = e_2 \mid \ \ldots\ \mid x \mid \mathrm{x} \mid l \mid e_1 +\!\!+ e_2 \mid \mathrm{IntToStr}(e) \\
\text{commands} & C & ::= & \mathsf{nop} \mid x := e \mid C_1; C_2 \mid \mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ e\ \mathsf{do}\ C \mid \mathsf{eval}\ e
\end{array}
$$

Fig. 3. Syntax of expressions and program statements.

notions of token and string mutually dependent), as are integer literals and variable names. For convenience we shall write tokens and strings using an obvious textual representation, for example writing "$x := 0$" for $[\mathsf{VarTk}\ x, \mathsf{AssignTk}, \mathsf{IntLitTk}\ 0]$.

Expressions and program statements. Fig. 3 gives the syntax of expressions and program statements. Expressions e occurring in program statements must not contain auxiliary variables x. Let Cmd and Exp be the sets of program statements and expressions respectively.

3.2 Semantics

We first fix a bijection $\mathscr{S} : \mathsf{Str} \to \mathbb{Z}$ from the set of strings to the integers; thus, strings can be encoded as integers and it will suffice to have a single datatype in our language. Let $\mathsf{Store} := \mathsf{Var} \to \mathbb{Z}$ be the set of program stores. Let $\mathsf{Env} := \mathsf{AuxVar} \to \mathbb{Z}$ be the set of environments for auxiliary variables, typically ρ.

Expression evaluation. We write $[\![e]\!]^{\mathrm{ex}}_{s,\rho}$ for the value of expression e in store s and environment ρ. Where e contains no program (resp. auxiliary) variables we omit s (resp. ρ). Expression evaluation is standard apart from the cases given in Fig. 4. String literals evaluate to their integer representation under the encoding $\mathscr{S}$. To evaluate concatenations $e_1 +\!\!+ e_2$ we first evaluate e_1 and e_2, which gives integer encodings of strings; we use $\mathscr{S}^{-1}$ to recover the two strings represented, concatenate these and finally apply the encoding again. The expression $\mathrm{IntToStr}(e)$ returns (an encoding of) the string representation of the integer value of e; for example $[\![\mathrm{IntToStr}(3)]\!]^{\mathrm{ex}} = \mathscr{S}(\text{“3”})$.

Small-step execution relation. Fig. 5 gives a small-step operational semantics. A *configuration* is a pair (C, s) of a command and a store, and is

$$[\![l]\!]^{\text{ex}} := \mathscr{S}(l) \qquad [\![e_1 +\!\!+ e_2]\!]^{\text{ex}}_{s,\rho} := \mathscr{S}(\mathscr{S}^{-1}([\![e_1]\!]^{\text{ex}}_{s,\rho}) +\!\!+ \mathscr{S}^{-1}([\![e_2]\!]^{\text{ex}}_{s,\rho}))$$

$$[\![\text{IntToStr}(e)]\!]^{\text{ex}}_{s,\rho} := \mathscr{S}([\mathsf{IntLitTk}\ [\![e]\!]^{\text{ex}}_{s,\rho}])$$

Fig. 4. Semantics of expression evaluation for non-standard cases.

$$\frac{}{(x := e, s) \to (\mathsf{nop},\ \lambda v.\ \text{if } v = x \text{ then } [\![e]\!]^{\text{ex}}_s \text{ else } s(v))} \qquad \frac{(C_1, s) \to (C_1', s')}{(C_1; C_2, s) \to (C_1'; C_2, s')}$$

$$\frac{[\![e]\!]^{\text{ex}}_s = 1}{(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s) \to (C_1, s)} \qquad \frac{[\![e]\!]^{\text{ex}}_s \neq 1}{(\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s) \to (C_2, s)}$$

$$\frac{[\![e]\!]^{\text{ex}}_s = 1}{(\mathsf{while}\ e\ \mathsf{do}\ C, s) \to (C; \mathsf{while}\ e\ \mathsf{do}\ C, s)} \qquad \frac{[\![e]\!]^{\text{ex}}_s \neq 1}{(\mathsf{while}\ e\ \mathsf{do}\ C, s) \to (\mathsf{nop}, s)}$$

$$\frac{}{(\mathsf{nop}; C, s) \to (C, s)} \qquad \frac{\mathit{parseCmd}(\mathscr{S}^{-1}([\![e]\!]^{\text{ex}}_s)) = \text{Just } (C, \text{“”})}{(\mathsf{eval}\ e, s) \to (C, s)}$$

Fig. 5. Small-step operational semantics of programs.

terminal if C is nop. One execution step is written $(C, s) \to (C', s')$ and if there is an execution sequence of zero or more steps from (C, s) to (C', s') we write $(C, s) \xrightarrow{*} (C', s')$. Note the semantics of the eval e command: we use a function *parseCmd*, to be defined later, to parse the argument string e into a command, and then run that command. (For convenience the eval e command gets "stuck" if e is not a syntactically correct string, but nowhere will we exploit this; our proof rules will enforce that e is well-formed.)

3.3 Parsing

Appendix A gives a reference implementation of a parser for our programming language, consisting of two main functions:

$$\mathit{parseExp} : \mathsf{Str} \to \text{maybe } (\mathsf{Exp} \times \mathsf{Str}) \qquad \mathit{parseCmd} : \mathsf{Str} \to \text{maybe } (\mathsf{Cmd} \times \mathsf{Str})$$

If *parseExp* is able to parse an expression from its input string, it returns Just (e, r) where e is the parsed expression and string r is any remaining input. If the input cannot be parsed, Nothing is returned. The details of the implementation do not matter; only two things are significant. Firstly, the parser functions are executable so that we can run them to conclude facts such as

$$\mathit{parseCmd}(\text{“nop”}) = \text{Just } (\mathsf{nop}, \text{“”})$$

Secondly, we eliminate ambiguity by demanding that bracketing is used with every binary operator. Thus for example “nop ; nop ; nop” is not allowed but

assertion	P, Q	::=	$P_1 \wedge P_2 \mid \neg P \mid \forall v.P \mid e_1 \leq e_2 \mid \mathrm{P}(e_1, \ldots, e_k)$
middle part of triple	$\mathcal{C}$	::=	$C \mid e \mid \mathrm{P}(\cdot, e_1, \ldots, e_k)$ where $e, e_1, \ldots, e_k$ contain no prog. vars
specification	Σ	::=	$\forall \vec{v}.\, \{P_1\}\, \mathcal{C}\, \{P_2\} \mid e_1 = e_2$ where e_1, e_2 contain no prog. vars
specification in context	Π	::=	$\Sigma_1, \ldots, \Sigma_k \vdash \Sigma \qquad (k \geq 0)$
representation judgement	$rep(e, e')$		

Fig. 6. Syntax of assertions, specifications and representation judgements.

"((nop ; nop) ; nop)" is. This means that the parser functions obey a locality property; for *parseCmd* this is

$$\frac{parseCmd(s) = \text{Just } (C, r)}{parseCmd(s +\!\!+ t) = \text{Just}(C, r +\!\!+ t)} \tag{3}$$

4 Our Hoare logic and its proof rules

Let PredN be a countably infinite set of predicate names, whose elements are typically written $\mathrm{P}, \mathrm{P}_1, \mathrm{P}_2, \ldots$. Fig. 6 gives the syntax of our assertions. Using the available assertion language one can express the "missing" connectives *true*, $\vee$, $\exists$, $=$ and so on in the usual way. Fig. 6 also introduces our logic's two judgements. A specification Σ is either an equality $e_1 = e_2$ between expressions, or a Hoare triple $\{P\}\, \mathcal{C}\, \{Q\}$ describing the behaviour of some set $\mathcal{C}$ of commands. $\mathcal{C}$ can be given in three ways: as a single concrete command C, as a string expression e which is parsed to obtain a command, or by a predicate application with a hole, $\mathrm{P}(\cdot, e_1, \ldots, e_k)$. The latter means the set of commands which results from taking all strings l satisfying $\mathrm{P}(l, e_1, \ldots, e_k)$ and parsing them; this lets us state the behaviour of all commands of a particular grammatical category P.

Our logic uses two judgements. The first, termed *specification in context*, has a familiar shape: $\Sigma_1, \ldots, \Sigma_k \vdash \Sigma$ means intuitively that specification Σ must hold in any context where specifications $\Sigma_1, \ldots, \Sigma_k$ also hold. For example,

$$\forall \mathrm{n}.\, \{x = \mathrm{n}\}\, \mathrm{P}(\cdot)\, \{x = \mathrm{n} + 1\} \;\vdash\; \left\{\, \mathrm{P}(s) \wedge x = 0 \,\right\}\ \mathsf{eval}\ s\ ;\ x := x + 1\ \left\{\, x = 2 \,\right\}$$

says that, in any context where all the strings in grammatical category P (i.e. satisfying the predicate P) parse into commands which increment x, the triple $\{\mathrm{P}(s) \wedge x = 0\}\, \mathsf{eval}\ s\ ;\ x := x + 1\, \{x = 2\}$ holds. (A similar kind of judgement is used in proof systems for recursive procedures, overviewed in [2] and given a modern treatment in [13], where the assumptions are about the procedures which might be called, rather than the strings which might be parsed and executed.

$$\frac{}{\{P\}\,\mathsf{nop}\,\{P\}}\ \epsilon \qquad \frac{}{\{P[x\backslash e]\}\,x := e\,\{P\}}\ \mathrm{A} \qquad \frac{\Gamma \vdash \{P\}\,C_1\,\{R\} \qquad \Gamma \vdash \{R\}\,C_2\,\{Q\}}{\Gamma \vdash \{P\}\,C_1; C_2\,\{Q\}}\ \mathrm{S}$$

$$\frac{\Gamma \vdash \{P \wedge e = 1\}\,C_1\,\{Q\} \qquad \Gamma \vdash \{P \wedge e \neq 1\}\,C_2\,\{Q\}}{\Gamma \vdash \{P\}\,\mathsf{if}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\,\{Q\}}\ \mathrm{I}$$

$$\frac{\Gamma \vdash \{I \wedge e = 1\}\,C\,\{I\}}{\Gamma \vdash \{I\}\,\mathsf{while}\ e\ \mathsf{do}\ C\,\{I \wedge e \neq 1\}}\ \mathrm{L} \qquad \frac{\Gamma \vdash \{P'\}\,\mathcal{C}\,\{Q'\}}{\Gamma \vdash \{P\}\,\mathcal{C}\,\{Q\}}\ P \Rightarrow P', Q' \Rightarrow Q\ \mathrm{W}$$

$$\frac{\Gamma \vdash \{P\}\,\mathcal{C}\,\{Q\}}{\Sigma, \Gamma \vdash \{P\}\,\mathcal{C}\,\{Q\}}\ \mathrm{C} \qquad \frac{\Gamma \vdash \Sigma_1, \ldots, \Sigma_n \qquad \Gamma, \Sigma_1, \ldots, \Sigma_n \vdash \Sigma}{\Gamma \vdash \Sigma}\ \textsc{Cut}$$

$$\frac{\Gamma, \forall \vec{\mathrm{x}}.\,\{P[\mathrm{x}\backslash e]\}\,\mathcal{C}[\mathrm{x}\backslash e]\,\{Q[\mathrm{x}\backslash e]\} \vdash \Sigma}{\Gamma, \forall \vec{\mathrm{x}}, \mathrm{x}.\,\{P\}\,\mathcal{C}\,\{Q\} \vdash \Sigma}\ \text{no program variables appear in } e\ \forall\textsc{InstL}$$

Fig. 7. Some standard Hoare logic rules, supported by our logic.

$$\frac{}{\mathit{rep}(\text{“}x\text{”}, x)} \qquad \frac{}{\mathit{rep}(\mathrm{IntToStr}(e), e)}$$

$$\frac{\mathit{rep}(e_1, e_1') \qquad \mathit{rep}(e_2, e_2')}{\mathit{rep}(\text{“(”} +\!\!+ e_1 +\!\!+ \text{“}\oplus\text{”} +\!\!+ e_2 +\!\!+ \text{“)”}, e_1' \oplus e_2')}\ \oplus \text{ is any binary op.}$$

Fig. 8. Some rules for *rep*.

Logics for assembly language given in [12, 4] also have similar assumptions, this time describing the behaviour of the code stored at particular memory locations.) We will often refer to a list $\Sigma_1, \ldots, \Sigma_n$ as a *context* and call it Γ.

Our logic features the standard Hoare logic rules for assignment, sequential composition, consequence etc. given in Fig. 7.

The *representation* judgement $\mathit{rep}(e, e')$ says approximately that expression e evaluates to the string representation of an expression equal to e'. For instance, $\mathit{rep}(\text{“}(x+1)\text{”}, x+1)$ because in all states “$(x+1)$” parses to the expression $x+1$ which is trivially equal to $x+1$. Fig. 8 gives proof rules for *rep*.

4.1 Proof rules for reasoning about eval

Note that Fig. 7 contains no rule for the eval statement which parses and runs strings. We now introduce such rules, which appear in Fig. 9.

The first rule (Parsed) is the simplest. This rule says that if a string literal l parses to a command C, then triples for C also apply to l. Of course, determining

$$\textsc{Parsed}\quad \frac{\mathit{parseCmd}(l) = \text{Just } (C, \text{“”}) \qquad \Gamma \vdash \{P\}\, C\, \{Q\}}{\Gamma \vdash \{P\}\, l\, \{Q\}} \qquad \textsc{ParsedA}\quad \frac{\mathit{rep}(e, e')}{\{P[x \backslash e']\}\ \text{“}x :=\text{”} +\!\!+ e\ \{P\}}$$

$$\textsc{ParsedS}\quad \begin{array}{l}\{P\}\, e_1\, \{R\}\\ \{R\}\, e_2\, \{Q\}\end{array}, \vdash \{P\}\ \text{“(”} +\!\!+ e_1 +\!\!+ \text{“;”} +\!\!+ e_2 +\!\!+ \text{“)”}\ \{Q\}$$

$$\textsc{ParsedI}\quad \frac{\mathit{rep}(e_g, e)}{\begin{array}{l}\{P \wedge e = 1\}\, e_1\, \{Q\}\\ \{P \wedge e \neq 1\}\, e_2\, \{Q\}\end{array}, \vdash \{P\}\ \text{“if”} +\!\!+ e_g +\!\!+ \text{“then”} +\!\!+ e_1 +\!\!+ \text{“else”} +\!\!+ e_2\ \{Q\}}$$

$$\textsc{ParsedL}\quad \frac{\mathit{rep}(e_g, e)}{\{I \wedge e = 1\}\, e_b\, \{I\} \vdash \{I\}\ \text{“while”} +\!\!+ e_g +\!\!+ \text{“do”} +\!\!+ e_b\ \{I \wedge e \neq 1\}}$$

$$\textsc{UseEq}\quad \frac{\Gamma, e = e' \vdash \{P'\}\, \mathcal{C}\, \{Q'\}}{\Gamma, e = e' \vdash \{P\}\, \mathcal{C}\, \{Q\}} \quad P \wedge e = e' \Rightarrow P' \text{ and } Q' \wedge e = e' \Rightarrow Q$$

Fig. 9. Rules for running strings with eval.

exactly which string an expression e evaluates to is not possible in general; for instance, we may know only that e has the form “(” $+\!\!+$ x $+\!\!+$ “;” $+\!\!+$ y $+\!\!+$ “)” where x, y are logical variables. But if we know, as assumptions, specifications for x and y, then we should nevertheless be able to conclude something about the behaviour of e. The (ParsedS) rule covers this case.

The rules (ParsedA), (ParsedI) and (ParsedL) deal with assignments, conditionals and loops in a similar fashion, using the *rep* judgement to bridge between string expressions and the expressions whose string representation they evaluate to. The (UseEq) rule is for making use of equations from the context; such assumptions are generated by the (GrInd) rule which we shall see in Section 4.4.

4.2 Proof rules for reasoning about recursion through the store

As mentioned in Section 2, although our programming language includes no special mechanism for declaring recursive procedures, one can still acheive recursion using eval. The simplest example is the program $x :=$ “eval x” ; eval x which creates a non-terminating recursion. This kind of implicit recursion has been called *recursion through the store*. Fig. 10 gives proof rules for reasoning about such recursion, inspired by those of [14]. As explained in [6] these rules have broadly the same shape as the well-known rules for reasoning about recursive procedures. For simplicity our (μ) rule is for simple recursion; the generalisation to mutual recursion is straightforward. The (seemingly trivial) first premise of

$$\mu\quad \frac{\{true\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{true\} \qquad \forall \vec{\mathrm{N}}.\,\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\} \;\vdash\; \{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}}{\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}}$$

$\vec{\mathrm{N}}$ is all auxiliary vars free in $P, Q, e_1, \ldots, e_k$

$$\mathrm{H}\quad \frac{P \;\Rightarrow\; \mathrm{P}(e, e_1, \ldots, e_k)}{\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\} \vdash \{P\}\,\mathsf{eval}\ e\,\{Q\}} \qquad \mathrm{R}\quad \frac{P \Rightarrow \mathrm{P}(e, e_1, \ldots, e_k) \qquad \{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}}{\{P\}\,\mathsf{eval}\ e\,\{Q\}}$$

Fig. 10. Rules for reasoning about recursion through the store.

(μ) makes sure that all strings in $\mathrm{P}(\cdot, e_1, \ldots, e_k)$ are syntactically well-formed commands.

4.3 Grammars

As explained in Section 2, using grammars to represent infinite sets of strings is a key part of our work. Here we explain how we treat them. A grammar is a sequence $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of *productions*, where each production $\mathcal{P}$ has the form

$$\mathrm{P}(\mathrm{v}_1, \ldots, \mathrm{v}_K) \quad \leftarrow \quad \mathcal{W}_1 \;\cdots\; \mathcal{W}_N \tag{4}$$

where

$$\mathcal{W} \quad ::= \quad \mathrm{P}(e_1, \ldots, e_k) \mid l \mid \mathrm{IntToStr}(e) \mid e = e'$$

(here expressions e may not use program variables). There can be multiple productions for a predicate, but predicate symbols must be used with consistent arities. We say a variable v occurs *privately* in (4) if it occurs in one of $\mathcal{W}_1, \ldots, \mathcal{W}_N$ but is not one of $\mathrm{v}_1, \ldots, \mathrm{v}_K$; such variables are treated as existentially quantified. Let $\mathrm{Pr(P)}$ denote the set of productions for a predicate P.

4.4 Proof rule for grammar-based induction

Our proof rule for grammar-based induction is the most involved of those we use, and requires some preliminary definitions. For ease of presentation we give a rule which works with a single inductive predicate definition; the generalisation to mutual inductive definitions is straightforward.

Let P be a predicate symbol, such that no production for P uses predicates other than P itself, and such that the left hand side all productions for P is $\mathrm{P}(\mathrm{v}_1, \ldots, \mathrm{v}_K)$. We choose a specification $\{P\}\,\mathrm{P}(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}$ which we wish to prove, inductively, about commands in P.

We now define, for each production $\mathcal{P} \in \mathrm{Pr(P)}$, an expression $E(\mathcal{P})$ and a context $\Gamma(\mathcal{P})$ based on $\mathcal{P}$. Each such $\mathcal{P}$ has the form $\mathrm{P}(\mathrm{v}_1, \ldots, \mathrm{v}_K) \leftarrow \mathcal{W}_1 \cdots \mathcal{W}_M$. We choose unused auxiliary variables $\mathrm{p}_1, \ldots, \mathrm{p}_M$ and then define

$$E(\mathcal{P}) \;:=\; E_1(\mathcal{W}_1) +\!\!+ \cdots +\!\!+ E_M(\mathcal{W}_M) \qquad \Gamma(\mathcal{P}) \;:=\; \Gamma_1(\mathcal{W}_1) \cup \cdots \cup \Gamma_M(\mathcal{W}_M)$$

where for $j = 1, \ldots, M$,

$$E_j(\mathrm{P}(e_1, \ldots, e_K)) := \mathrm{p}_j \qquad E_j(l) := l$$

$$E_j(\mathrm{IntToStr}(e)) := \mathrm{IntToStr}(e) \qquad E_j(e = e') := \text{“”}$$

$$\Gamma_j(l) := \emptyset \qquad \Gamma_j(\mathrm{IntToStr}(e)) := \emptyset \qquad \Gamma_j(e = e') := e = e'$$

$$\Gamma_j(\mathrm{P}(e_1, \ldots, e_K)) \;:=\; \{P\}\,\mathrm{p}_j\,\{Q\}\,[\mathrm{v}_1, \ldots, \mathrm{v}_K \backslash e_1, \ldots, e_K]$$

Our rule for grammar-based induction is then as follows.

$$\textsc{GrInd}\quad \frac{\bigwedge_{\mathcal{P} \in \mathrm{Pr(P)}} \Gamma, \Gamma(\mathcal{P}) \;\vdash\; \{P\}\,E(\mathcal{P})\,\{Q\}}{\Gamma \vdash \{P\}\,\mathrm{P}(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}}$$

no variable occurring privately in a production for P is free in P or Q

no variable occurring in a production for P is free in Γ

5 Example proofs

We shall now show how to use our logic to prove correctness properties of the two programs in Fig. 1 which we introduced in Section 2.

5.1 Exponential calculation example

We shall prove $\{n \geq 0\}\,\mathrm{P1}\,\{a = x^n\}$, which we do using the grammar for P given in (1). The proof has three parts:

1. Prove that strings l satisfying $\mathrm{P}(l, \mathrm{n})$ behave as we want them to, i.e. prove
$$\{\mathit{true}\}\,\mathrm{P}(\cdot, \mathrm{n})\,\{a = x^{\mathrm{n}}\} \tag{5}$$
(which expresses (2) formally).
2. Prove that for any input $n > 0$, the while loop in P1 successfully generates in s a string satisfying $\mathrm{P}(s, n)$.
3. Combine 1. and 2. to prove the whole program.

Part 1: To show (5) we use the following instance of the GrInd rule:

$$\frac{\begin{array}{c}\mathrm{n} = 0 \vdash \{\mathit{true}\}\text{ “}a := 1\text{”}\,\{a = x^{\mathrm{n}}\} \\ \mathrm{n} = 1 \vdash \{\mathit{true}\}\text{ “}a := x\text{”}\,\{a = x^{\mathrm{n}}\} \\ \{\mathit{true}\}\,\mathrm{p}\,\{a = x^{\mathrm{n}-1}\} \;\vdash\; \{\mathit{true}\}\text{ “(” } +\!\!+ \mathrm{p} +\!\!+ \text{ “;” } +\!\!+ \text{ “}a := a \times x\text{” } +\!\!+ \text{ “)”}\,\{a = x^{\mathrm{n}}\}\end{array}}{\{\mathit{true}\}\,\mathrm{P}(\cdot, \mathrm{n})\,\{a = x^{\mathrm{n}}\}}$$

The first premise will follow, by the (Parsed) rule, from

$$\mathrm{n} = 0 \vdash \{\mathit{true}\}\,a := 1\,\{a = x^{\mathrm{n}}\}$$

By the (UseEq) rule, this in turn will follow from

$$\mathrm{n} = 0 \vdash \{\mathit{true}\}\, a := 1\, \{a = 1\}$$

which trivially holds. The second premise is proved in a similar way. It remains to prove the third premise. The following is an instance of the (ParsedS) rule:

$$\begin{array}{l}\{\mathit{true}\}\ \mathrm{p}\ \left\{a = x^{\mathrm{n}-1}\right\},\\ \left\{a = x^{\mathrm{n}-1}\right\}\ \text{“}a := a \times x\text{”}\ \{a = x^{\mathrm{n}}\}\end{array}$$

$$\vdash \{\mathit{true}\}\ \text{“(”} +\!\!+\ \mathrm{p} +\!\!+\ \text{“;”} +\!\!+\ \text{“}a := a \times x\text{”} +\!\!+\ \text{“)”}\ \{a = x^{\mathrm{n}}\}$$

Using the (A), (Parsed) and (C) rules together, we derive

$$\{\mathit{true}\}\ \mathrm{p}\ \left\{a = x^{\mathrm{n}-1}\right\}\ \vdash\ \left\{a = x^{\mathrm{n}-1}\right\}\ \text{“}a := a \times x\text{”}\ \{a = x^{\mathrm{n}}\}$$

Combining these last two with the (Cut) rule gives us the third premise.

Part 2: We use $\mathrm{P}(s, n-m)$ as our invariant for the while loop. To show that the body preserves this invariant, we need to show

$$\left\{\mathrm{P}(s, n-m) \wedge m \neq 0\right\}\ \begin{array}{l}(\mathsf{if}\ s = \text{“}a := 1\text{”}\ \mathsf{then}\\ \quad s := \text{“}a := x\text{”}\\ \mathsf{else}\\ \quad s := \text{“(”} +\!\!+\ s +\!\!+\ \text{“;}a := a \times x\text{)”})\ ;\\ m := m - 1\end{array}\ \left\{\mathrm{P}(s, n-m)\right\}$$

Using standard Hoare logic reasoning, this reduces to the following subgoals:

$$\left\{\begin{array}{l}\mathrm{P}(s, n-m)\\ \wedge\ m \neq 0\\ \wedge\ s = \text{“}a := 1\text{”}\end{array}\right\}\ s := \text{“}a := x\text{”}\ \left\{\mathrm{P}(s, n-m+1)\right\}$$

$$\left\{\begin{array}{l}\mathrm{P}(s, n-m)\\ \wedge\ m \neq 0\\ \wedge\ s \neq \text{“}a := 1\text{”}\end{array}\right\}\ s := \text{“(”} +\!\!+\ s +\!\!+\ \text{“;}a := a \times x\text{)”}\ \left\{\mathrm{P}(s, n-m+1)\right\}$$

These are discharged using the consequence (W) and assignment (A) rules; the following facts are needed to prove the implications that arise:

$$\mathrm{P}(s, n-m) \wedge s = \text{“}a := 1\text{”} \Rightarrow n - m = 0$$

$$\mathrm{P}(\text{“}a := x\text{”}, 1)$$

$$\mathrm{P}(s, r)\ \Rightarrow\ \mathrm{P}(\text{“(”} +\!\!+\ s +\!\!+\ \text{“;}a := a \times x\text{)”}, r+1)$$

We shall not prove these formally here; they follow from the definition (1) of P.

Part 3: It is easy to see that our loop invariant is established. Thus upon termination of the while loop we have $\mathrm{P}(s, n-m) \wedge m = 0$, that is, $\mathrm{P}(s, n)$. Thus, in order to acheive our final goal $\{n \geq 0\}\, \mathrm{P1}\, \{a = x^n\}$, it remains to prove $\{\mathrm{P}(s, n)\}\ \mathsf{eval}\ s\, \{a = x^n\}$. But this follows from (5) by the (R) and (W) rules.

Remark 1. Note that this example program performs an equality test on commands, represented as strings; we include it to emphasise that such syntactic ("intensional") operations are supported in our approach. Approaches such as [17], based on domain theory, cannot support such operations because stored commands are represented as functions and their syntactic information is lost.

5.2 Self-modifying factorial example

We shall prove $\{true\}\,\mathrm{P2}\,\{r = 10!\}$. Here is how we define our grammar:

$$\mathrm{P} \quad \leftarrow \quad S_0$$

$$\mathrm{P} \quad \leftarrow \quad \text{“}\mathsf{if}\ (n =\text{”}\ \mathrm{IntToStr(p)}\ \text{“})\ \mathsf{then}\ r :=\text{”}\ \mathrm{IntToStr(q)}\ \text{“}\mathsf{else}\text{”}\ \mathrm{P} \quad \mathrm{q} = \mathrm{p}!$$

(Recall that S_0 is given in Fig. 1.) We define a shorthand $T(\mathcal{C})$ as follows:

$$T(\mathcal{C}) \quad := \quad \left\{ \begin{array}{c} \mathrm{P}(s) \\ \wedge\, n = \mathrm{N} \\ \wedge\, n \geq 0 \end{array} \right\}\ \mathrm{P}(\mathcal{C})\ \left\{ \begin{array}{c} \mathrm{P}(s) \\ \wedge\, n = \mathrm{N} \\ \wedge\, n \geq 0 \\ \wedge\, r = n! \end{array} \right\}$$

We will concentrate on the main part of the proof, which is to show $T(\mathrm{P}(\cdot))$, which says that commands in P correctly calculate factorials, while ensuring that s always contains a program from P. To do so, we will use the following instance of the (μ) rule:

$$\frac{\{true\}\,\mathrm{P}(\cdot)\,\{true\} \qquad \forall \mathrm{N}.T(\mathrm{P}(\cdot)) \ \vdash\ T(\mathrm{P}(\cdot))}{T(\mathrm{P}(\cdot))}$$

(μ) is the right rule to use because commands in P invoke themselves with eval i.e. they perform recursion through the store. The first premise simply states that all strings in $\mathrm{P}(\cdot)$ are syntactically well-formed commands. This is the least interesting part of the proof so we omit it. For the second premise, we use the following instance of the (GrInd) rule:

$$\frac{\begin{array}{c} \forall \mathrm{N}.T(\mathrm{P}(\cdot)) \ \vdash\ T(S_0) \\ \forall \mathrm{N}.T(\mathrm{P}(\cdot)), T(\mathrm{r}), \mathrm{q} = \mathrm{p}! \ \vdash\ T\left(\begin{array}{l} \text{“}\mathsf{if}\ (n =\text{”} +\!\!+\ \mathrm{IntToStr(p)} +\!\!+\ \text{“})\text{”} +\!\!+\ \text{“}\mathsf{then}\ r :=\text{”} \\ +\!\!+\ \mathrm{IntToStr(q)} +\!\!+\ \text{“}\mathsf{else}\text{”} +\!\!+\ \mathrm{r} \end{array} \right) \end{array}}{\forall \mathrm{N}.T(\mathrm{P}(\cdot)) \vdash T(\mathrm{P}(\cdot))} \tag{6}$$

Proving the first premise of (6) **(the base case)**: By the (Parsed) rule it will suffice to show

$$\forall \mathrm{N}.T(\mathrm{P}(\cdot)) \ \vdash\ T\left(\begin{array}{l} \mathsf{if}\ n = 0\ \mathsf{then}\ r := 1 \\ \mathsf{else} \\ \quad n := n - 1; \\ \quad \mathsf{eval}\ s; \\ \quad n := n + 1; \\ \quad r := r \times n; \\ \quad s := \text{“}\mathsf{if}\ (n =\text{”} +\!\!+\ \mathrm{IntToStr}(n) +\!\!+\ \text{“})\text{”} +\!\!+\ \text{“}\mathsf{then}\text{”} \\ \qquad +\!\!+\ \text{“}r :=\text{”} +\!\!+\ \mathrm{IntToStr}(r) +\!\!+\ \text{“}\mathsf{else}\text{”} +\!\!+\ s \end{array} \right)$$

The true branch is easily dealt with. For the false branch it will suffice to show:

$$(T(\mathrm{P}(\cdot)))[\mathrm{N}\backslash\mathrm{N}-1] \tag{7}$$

$$\vdash \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}-1\\ \wedge\, n \geq 0\end{array}\right\} \begin{array}{l}\mathsf{eval}\ s;\\ n := n+1;\\ r := r \times n;\\ s := \text{“}\mathsf{if}\ (n =\text{”} +\!\!+ \mathrm{IntToStr}(n) +\!\!+ \text{“)”}\\ \quad +\!\!+ \text{“}\mathsf{then}\ r :=\text{”} +\!\!+ \mathrm{IntToStr}(r) +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ s\end{array} \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\}$$

(here we have used ($\forall$InstL) to instantiate N on the left of $\vdash$ with N$-$1). The (H) rule gives us

$$(T(\mathrm{P}(\cdot)))[\mathrm{N}\backslash\mathrm{N}-1] \ \vdash\ (T(\mathsf{eval}\ s))[\mathrm{N}\backslash\mathrm{N}-1]$$

Using this with (S) and (C), we will have (7) (and thus be done) if we can show

$$\left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}-1\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\} \begin{array}{l}n := n+1;\\ r := r \times n;\\ s := \text{“}\mathsf{if}\ (n =\text{”} +\!\!+ \mathrm{IntToStr}(n) +\!\!+ \text{“)”}\\ \quad +\!\!+ \text{“}\mathsf{then}\ r :=\text{”} +\!\!+ \mathrm{IntToStr}(r) +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ s\end{array} \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\}$$

Standard reasoning, and using familiar properties of factorials, reduces this to:

$$\left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\} \begin{array}{l}s := \text{“}\mathsf{if}\ (n =\text{”} +\!\!+ \mathrm{IntToStr}(n) +\!\!+ \text{“)”}\\ \quad +\!\!+\ +\!\!+ \text{“}\mathsf{then}\ r :=\text{”} +\!\!+ \mathrm{IntToStr}(r) +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ s\end{array} \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\}$$

We can finish this using (A) and (W) because we have the following entailment:

$$\mathrm{P}(s) \wedge r = n! \quad \Rightarrow \quad \mathrm{P}\left(\begin{array}{l}\text{“}\mathsf{if}\ n = (\text{”} +\!\!+ \mathrm{IntToStr}(n) +\!\!+ \text{“)”} +\!\!+ \text{“}\mathsf{then}\ r :=\text{”}\\ +\!\!+ \mathrm{IntToStr}(r) +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ s\end{array}\right)$$

Proving the second premise of (6) **(the inductive case):** Here we shall prove something stronger, namely

$$T(\mathrm{r}), \mathrm{q} = \mathrm{p}! \ \vdash\ T\left(\begin{array}{l}\text{“}\mathsf{if}\ (n =\text{”} +\!\!+ \mathrm{IntToStr}(\mathrm{p}) +\!\!+ \text{“)”} +\!\!+ \text{“}\mathsf{then}\ r :=\text{”}\\ +\!\!+ \mathrm{IntToStr}(\mathrm{q}) +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ \mathrm{r}\end{array}\right)$$

By the (Cut) rule, we will be done if we can show

$$T(\mathrm{r}), \mathrm{q} = \mathrm{p}! \ \vdash\ \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, n = \mathrm{p}\end{array}\right\} \begin{array}{l}\text{“}r :=\text{”}\\ +\!\!+\mathrm{IntToStr}(\mathrm{q})\end{array} \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\} \tag{8}$$

and

$$T(\mathrm{r}), \mathrm{q} = \mathrm{p}!, \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, n = \mathrm{p}\end{array}\right\} \begin{array}{l}\text{“}r :=\text{”}\\ +\!\!+\\ \mathrm{IntToStr}(\mathrm{q})\end{array} \left\{\begin{array}{c}\mathrm{P}(s)\\ \wedge\, n = \mathrm{N}\\ \wedge\, n \geq 0\\ \wedge\, r = n!\end{array}\right\} \ \vdash\ T\left(\begin{array}{l}\text{“}\mathsf{if}\ (n =\text{”} +\!\!+ \mathrm{IntToStr}(\mathrm{p}) +\!\!+ \text{“)”}\\ +\!\!+\text{“}\mathsf{then}\ r :=\text{”} +\!\!+ \mathrm{IntToStr}(\mathrm{q})\\ +\!\!+\text{“}\mathsf{else}\text{”} +\!\!+ \mathrm{r}\end{array}\right) \tag{9}$$

$$[\![l]\!]_i \;:=\; \{(\rho, l) \mid \rho \in \mathsf{Env}\} \qquad [\![\mathrm{P}(e_1, \ldots, e_k)]\!]_0 \;:=\; \emptyset$$

$$[\![\mathrm{P}(e_1, \ldots, e_k)]\!]_{i+1} \;:=\; \{(\rho, l) \mid (l, [\![e_1]\!]^{\mathrm{ex}}_\rho, \ldots, [\![e_k]\!]^{\mathrm{ex}}_\rho) \in \chi_i(\mathrm{P})\}$$

$$[\![\mathrm{IntToStr}(e)]\!]_i \;:=\; \{(\rho, [\mathsf{IntLitTk}\ [\![e]\!]^{\mathrm{ex}}_\rho]) \mid \rho \in \mathsf{Env}\}$$

$$[\![e = e']\!]_i \;:=\; \{(\rho, \text{“”}) \mid \rho \in \mathsf{Env} \text{ is s.t. } [\![e]\!]^{\mathrm{ex}}_\rho = [\![e']\!]^{\mathrm{ex}}_\rho\}$$

$$\chi_i(\mathrm{P}(\mathrm{v}_1, \ldots, \mathrm{v}_K) \leftarrow \mathcal{W}_1 \cdots \mathcal{W}_N) \quad := \quad \left\{ (l_1 +\!\!+ \cdots +\!\!+ l_N, \sigma) \,\middle|\, \begin{array}{l} \exists \rho \in \mathsf{Env} \text{ s.t. } \rho(\mathrm{v}_1), \ldots, \rho(\mathrm{v}_K) = \sigma \\ \text{and for } j = 1..N, (\rho, l_j) \in [\![\mathcal{W}_j]\!]_i \end{array} \right\}$$

$$\chi_i(\mathrm{P}) \quad := \quad \bigcup_{\mathcal{P} \in \mathrm{Pr}(\mathrm{P})} \chi_i(\mathcal{P})$$

Fig. 11. Semantics of grammars.

The proof for (8) is straightforward, using the (ParsedA) rule and the fact *rep*(IntToStr(q), q) to deal with the assignment, then using (C) to add the assumptions, and then using the (UseEq) rule to make use of the assumption q = p!. To prove (9) one uses the (ParsedI) rule together with the fact that *rep*(“($n =$” +\!\!+ IntToStr(p) +\!\!+ ‘)’, $n =$ p).

6 Semantics of assertions and judgements

6.1 Semantics of grammars

To give meaning to a grammar, we construct a sequence of interpretations $\chi_0, \chi_1, \ldots : \mathsf{PredN} \to \mathbb{P}(\mathsf{Str} \times \mathbb{Z}^*)$ of the predicates, where χ_i corresponds to allowing grammar derivations of depth $\leq i$. Then we obtain our interpretation by $\chi := \bigcup_i \chi_i$. The details are given in Fig. 11. We overload symbol χ to interpret both predicates and productions; each “word” $\mathcal{W}$ is interpreted at depth i by $[\![\mathcal{W}]\!]_i \subseteq \mathsf{Env} \times \mathsf{Str}$. The important point is that having the depth-limited interpretations $\chi_0, \chi_1, \ldots$ allows us to argue by induction on derivation depth.

6.2 Semantics of assertions

Let $\mathcal{P}_1, \ldots, \mathcal{P}_n$ be an arbitrary, but fixed, grammar, and let χ be the associated predicate interpretation. We write $[\![P]\!]^{\mathrm{as}}_\rho \subseteq \mathsf{Store}$ for the interpretation of assertion P in environment ρ. The interpretation of assertions is completely standard apart from for predicates:

$$[\![\mathrm{P}(e, e_1, \ldots, e_k)]\!]^{\mathrm{as}}_\rho \quad := \quad \{s \mid (\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{s,\rho}), [\![e_1]\!]^{\mathrm{ex}}_{s,\rho}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{s,\rho}) \in \chi(\mathrm{P})\}$$

$$[\![\forall \vec{\mathrm{v}}.\, \{P\}\, \mathcal{C}\, \{Q\}]\!]^{\mathrm{sp}} := \left\{ (\rho, n) \in \mathsf{Env} \times \mathbb{N}_{-1} \;\middle|\; \begin{array}{l} \text{for all } \rho' \text{ agreeing with } \rho \text{ except possibly} \\ \text{at } \vec{\mathrm{v}},\ 1.\ \mathsf{parseError} \notin [\![\mathcal{C}]\!]^{\mathrm{mid}}_{\rho'}, \text{and} \\ 2.\ \text{either } n = -1 \text{ or for all } C \in [\![\mathcal{C}]\!]^{\mathrm{mid}}_{\rho'} \\ \text{we have } \rho' \vDash^n \{P\}\, C\, \{Q\} \end{array} \right\}$$

$$[\![e = e']\!]^{\mathrm{sp}} := \{\rho \mid [\![e]\!]^{\mathrm{ex}}_{\rho} = [\![e']\!]^{\mathrm{ex}}_{\rho}\} \times \mathbb{N}_{-1}$$

$$[\![\Sigma_1, \ldots, \Sigma_k \vdash \Sigma]\!]^{\mathrm{sic}} \;:=\; \left\{ \rho \in \mathsf{Env} \middle| \forall n \in \mathbb{N},\ \text{if } (\rho, n-1) \in \textstyle\bigcap_{i=1}^{k} [\![\Sigma_i]\!]^{\mathrm{sp}} \text{ then } (\rho, n) \in [\![\Sigma]\!]^{\mathrm{sp}} \right\}$$

Fig. 12. Semantics of specifications, and specifications in context.

Note that $\mathscr{S}^{-1}$ is used to decode the value of e into a string. Entailment $P \Rightarrow Q$ means that for all ρ, $[\![P]\!]^{\mathrm{as}}_{\rho} \subseteq [\![Q]\!]^{\mathrm{as}}_{\rho}$.

6.3 Semantics of Hoare triples and specifications

Before we can give semantics to specifications, we state our (partial correctness) interpretation of Hoare triples. For our soundness proofs it will be convenient to add another possibility for the middle part $\mathcal{C}$ of Hoare triples:

$$\mathcal{C} ::= \quad \ldots \mid \mathrm{P}^m(\cdot, e_1, \ldots, e_k)$$

where $m \in \mathbb{N}$ and $e, e_1, \ldots, e_k$ contain no program variables.

Definition 1. *Let the middle components $\mathcal{C}$ of Hoare triples be interpreted to subsets of $\mathsf{Cmd} \cup \{\mathsf{parseError}\}$ as follows:*

$$[\![C]\!]^{\mathrm{mid}}_{\rho} := \{C\} \qquad [\![e]\!]^{\mathrm{mid}}_{\rho} := \{F(\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{\rho}))\}$$

$$[\![\mathrm{P}(\cdot, e_1, \ldots, e_k)]\!]^{\mathrm{mid}}_{\rho} := \{F(l) \mid (l, [\![e_1]\!]^{\mathrm{ex}}_{\rho}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{\rho}) \in \chi(\mathrm{P})\}$$

$$[\![\mathrm{P}^m(\cdot, e_1, \ldots, e_k)]\!]^{\mathrm{mid}}_{\rho} := \{F(l) \mid (l, [\![e_1]\!]^{\mathrm{ex}}_{\rho}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{\rho}) \in \chi_m(\mathrm{P})\}$$

where $F(l) := \begin{cases} C \text{ if } \mathit{parseCmd}(l) \text{ is } \mathrm{Just}\ (C, \text{“”}) \\ \mathsf{parseError} \text{ otherwise} \end{cases}$ □

Definition 2. Semantics of Hoare triples. *For $\rho \in \mathsf{Env}$, $n \in \mathbb{N}$ and $C \in \mathsf{Cmd}$ we write $\rho \vDash^n \{P\}\, C\, \{Q\}$ to mean that if $s \in [\![P]\!]^{\mathrm{as}}_{\rho}$ and if $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer then $s' \in [\![Q]\!]^{\mathrm{as}}_{\rho}$.* □

From now on we use $\mathbb{N}_{-1}$ as a shorthand for $\mathbb{N} \cup \{-1\}$. Fig. 12 gives the semantics $[\![\Sigma]\!]^{\mathrm{sp}} \subseteq \mathsf{Env} \times \mathbb{N}_{-1}$ of specification Σ, and also the semantics $[\![\Sigma_1, \ldots, \Sigma_k \vdash \Sigma]\!]^{\mathrm{sic}} \subseteq \mathsf{Env}$ of a specification in context. For convenience we shall often write $[\![\Sigma_1, \ldots, \Sigma_k]\!]^{\mathrm{sp}}$ as shorthand for $\bigcap_{i=1}^{k} [\![\Sigma_i]\!]^{\mathrm{sp}}$.

The indices n appearing in Fig. 12 are used to count execution steps; $(\rho, n) \in [\![\{P\}\, C\, \{Q\}]\!]^{\mathrm{sp}}$ says that (in environment ρ) command C cannot start in a P-state and terminate in a non-Q-state within n steps. Specifications on the right of $\vdash$ are considered with one extra step relative to those on the left. Knowing $(\rho, -1) \in [\![\{P\}\, \mathrm{P}(\cdot)\, \{Q\}]\!]^{\mathrm{sp}}$ tells us nothing about the behaviour of the commands in grammar category P, but it does tell us that they are all syntactically well-formed (i.e. parsed without error). We say that a specification in context $\Sigma_1, \ldots, \Sigma_k \vdash \Sigma$ *holds* if $[\![\Sigma_1, \ldots, \Sigma_k \vdash \Sigma]\!]^{\mathrm{sic}} = \mathsf{Env}$.

Definition 3. Semantics of *rep*. *We define* $rep(e, e')$ *to hold if for all environments* ρ *and all stores* s, $parseExp(\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{s,\rho}))$ *has the form* Just $(E, \text{“”})$ *where all variables in* E *are program variables, and furthermore,* $[\![E]\!]^{\mathrm{ex}}_s = [\![e']\!]^{\mathrm{ex}}_{s,\rho}$. □

7 Soundness of proof rules

Here we prove the soundness of two interesting rules, (ParsedI) and (GrInd). Further soundness proof are given in Appendix B.

Theorem 1. *The (ParsedI) rule is sound.*

Proof. Assume $rep(e_g, e)$. Let $\rho \in \mathsf{Env}$ and $n \in \mathbb{N}$ be such that

$$(\rho, n-1) \in [\![\{P \wedge e = 1\}\, e_1\, \{Q\}\,, \{P \wedge e \neq 1\}\, e_2\, \{Q\}]\!]^{\mathrm{sp}}$$

Unpacking the definitions, we find that $[\![e_1]\!]^{\mathrm{mid}}_\rho = \{C_1\}$ for some command C_1, where for some string l_1, $[\![e_1]\!]^{\mathrm{ex}}_\rho = \mathscr{S}(l_1)$ and $parseCmd(\mathscr{S}^{-1}(l_1)) = \text{Just } (C_1, \text{“”})$. Furthermore, provided $n \geq 1$, we have $\rho \vDash^{n-1} \{P \wedge e = 1\}\, C_1\, \{Q\}$. The analogous things hold for e_2, for a command C_2 and string l_2. We are required to prove that

$$(\rho, n) \in [\![\{P\}\ \text{“}\mathsf{if}\text{”} +\!\!+ e_g +\!\!+ \text{“}\mathsf{then}\text{”} +\!\!+ e_1 +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ e_2\, \{Q\}]\!]^{\mathrm{sp}}$$

From the semantics of *rep*, we find that $parseExp(\mathscr{S}^{-1}([\![e_g]\!]^{\mathrm{ex}}_\rho))$ has the form Just $(E, \text{“”})$ where all variables in E are program variables, and furthermore, for all $s \in \mathsf{Store}$, $[\![E]\!]^{\mathrm{ex}}_s = [\![e]\!]^{\mathrm{ex}}_{s,\rho}$. Let C be the command $\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$.

Using the definition of our parser, the parser's locality property (3), the above equations for l_1, l_2, and $parseExp(\mathscr{S}^{-1}([\![e_g]\!]^{\mathrm{ex}}_\rho)) = \text{Just } (E, \text{“”})$, it follows that $parseCmd(\mathscr{S}^{-1}([\![\text{“}\mathsf{if}\text{”} +\!\!+ e_g +\!\!+ \text{“}\mathsf{then}\text{”} +\!\!+ e_1 +\!\!+ \text{“}\mathsf{else}\text{”} +\!\!+ e_2]\!]^{\mathrm{ex}}_\rho)) = \text{Just } (C, \text{“”})$. Hence all that remains to prove is that $\rho \vDash^n \{P\}\, C\, \{Q\}$.

So let $s \in [\![P]\!]^{\mathrm{as}}_\rho$ and suppose that $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer. We need to show $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$. There are two cases: when $[\![E]\!]^{\mathrm{ex}}_s = 1$ and when $[\![E]\!]^{\mathrm{ex}}_s \neq 1$. These cases are very similar so we prove only the first. By the structure of the transition relation, we must have $(C_1, s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n-1$ steps or fewer, and we must have $n \geq 1$.

It follows from $[\![E]\!]^{\mathrm{ex}}_s = [\![e]\!]^{\mathrm{ex}}_{s,\rho}$ that $[\![e]\!]^{\mathrm{ex}}_s = 1$ and therefore $s \in [\![P \wedge e = 1]\!]^{\mathrm{as}}_\rho$. Thus the thing we require, $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$, follows from $\rho \vDash^{n-1} \{P \wedge e = 1\}\, C_1\, \{Q\}$. □

Theorem 2. *The (GrInd) rule is sound.*

Let P, P, Q, Γ be as described in Section 4.4. For $m \in \mathbb{N}$ we define $\Phi(m)$ to be the following property:

$$[\![\Gamma \vdash \{P\}\,\mathrm{P}^m(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}]\!]^{\mathrm{sic}} = \mathsf{Env}$$

Intuitively $\Phi(m)$ states that commands in P with derivations of depth at most m meet their specification. Our proof will be by induction on m. The following lemma does most of the work.

Lemma 1. *Let $m \in \mathbb{N}_{-1}$ be such that either $m = -1$, or $m \geq 0$ and $\Phi(m)$ holds. Let $l \in \mathsf{Str}, \rho \in \mathsf{Env}, n \in \mathbb{N}$ be such that $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$ and*

$$(l, \rho(\mathrm{v}_1), \ldots, \rho(\mathrm{v}_K)) \in \chi_{m+1}(\mathcal{P})$$

where $\mathcal{P} \in \mathrm{Pr}(\mathrm{P})$. Then there exists an environment $\hat{\rho} \in \mathsf{Env}$ such that:

(A) $(\hat{\rho}, n-1) \in [\![\Gamma, \Gamma(\mathcal{P})]\!]^{\mathrm{sp}}$,
(B) $\mathscr{S}^{-1}([\![E(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}}) = l$, *and*
(C) $\hat{\rho}$ *agrees with* ρ *on all variables free in* P *or* Q.

Proof (Lemma 1). We show how to produce the required $\hat{\rho} \in \mathsf{Env}$. Production $\mathcal{P}$ has the form $\mathrm{P}(\mathrm{v}_1, \ldots, \mathrm{v}_K) \leftarrow \mathcal{W}_1 \cdots \mathcal{W}_M$. Therefore l has the form $l_1 +\!\!+ \cdots +\!\!+ l_M$ where for some environment ρ' we have: ρ' agrees with ρ on $\mathrm{v}_1, \ldots, \mathrm{v}_K$, and for $j = 1..M$, $(\rho', l_j) \in [\![\mathcal{W}_j]\!]_{m+1}$. Among the premises of (GrInd) is $\Gamma, \Gamma(\mathcal{P}) \vdash \{P\}\,E(\mathcal{P})\,\{Q\}$. We define the required environment $\hat{\rho}$ as follows:

$$\hat{\rho}(\mathrm{v}) \quad := \quad \begin{cases} \mathscr{S}(l_i) & \text{if v is } \mathrm{p}_i \text{ for some } i \in \{1, \ldots, M\} \\ \rho'(\mathrm{v}) & \text{if v occurs in a production for P} \\ \rho(\mathrm{v}) & \text{otherwise} \end{cases}$$

(Note that there is no overlap between the first two cases, since $\mathrm{p}_1, \ldots, \mathrm{p}_M$ were chosen as unused auxiliary variables.) The proof that this $\hat{\rho}$ actually has the claimed properties (A), (B), (C) is given in Appendix B. □

Proof (Theorem 2). It is easily seen that the conclusion of the rule is equivalent to having $\Phi(m)$ for all $m \in \mathbb{N}$. We shall prove this by induction on m, using Lemma 1 to obtain a suitable environment $\hat{\rho}$ with which to fire the premise of the rule. For space reasons we omit the base case; it is very similar to the inductive case.

Inductive case: Let $\Phi(m)$ hold for some $m \in \mathbb{N}$, and we shall prove $\Phi(m+1)$. Let $\rho \in \mathsf{Env}$; we must prove that $\rho \in [\![\Gamma \vdash \{P\}\,\mathrm{P}^{m+1}(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}]\!]^{\mathrm{sic}}$. So let $n \in \mathbb{N}$ be s.t. $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$; we must show $(\rho, n) \in [\![\{P\}\,\mathrm{P}^{m+1}(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}]\!]^{\mathrm{sp}}$.

So let l be any string s.t. $(l, \rho(\mathrm{v}_1), \ldots, \rho(\mathrm{v}_K)) \in \chi_{m+1}(\mathrm{P})$. It follows that there exists a production $\mathcal{P} \in \mathrm{Pr}(\mathrm{P})$ such that $(l, \rho(\mathrm{v}_1), \ldots, \rho(\mathrm{v}_K)) \in \chi_{m+1}(\mathcal{P})$.

It will suffice to prove that there exists a command C such that $\mathit{parseCmd}(l) = \mathrm{Just}\,(C, \text{“”})$ and $\rho \vDash^n \{P\}\,C\,\{Q\}$.

By Lemma 1 there exists $\hat{\rho} \in \mathsf{Env}$ with properties (A), (B), (C) stated above. Among the premises of (GrInd) is $\Gamma, \Gamma(\mathcal{P}) \vdash \{P\}\,E(\mathcal{P})\,\{Q\}$. Combining this with (A) we see that $(\hat{\rho}, n) \in [\![\{P\}\,E(\mathcal{P})\,\{Q\}]\!]^{\mathrm{sp}}$. Thus $\mathit{parseCmd}(\mathscr{S}^{-1}([\![E(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}}))$ has the form $\mathrm{Just}(C, \text{“”})$ for some command C such that $\hat{\rho} \vDash^n \{P\}\,C\,\{Q\}$. Using (B) we derive $\mathit{parseCmd}(l) = \mathrm{Just}(C, \text{“”})$ as required. From (C) and $\hat{\rho} \vDash^n \{P\}\,C\,\{Q\}$ it follows that $\rho \vDash^n \{P\}\,C\,\{Q\}$ as required. □

8 Conclusions

We have presented a Hoare logic, to our knowledge the first, for reasoning about string-based runtime code generation and *eval*, for a simple language, and we have demonstrated its use through examples.

Future work. Our next goal is to integrate the ideas developed here into a logic for a real programming language such as JavaScript, which makes frequent use of string-based runtime code generation and *eval*. In particular, we envisage combining our ideas with Gardner, Maffeis and Smith's logic for JavaScript [9]. However, such an integration requires substantial work for the following reasons. Firstly, we would need to work with character sequences instead of our idealised strings, and account for a lexer as well as a parser. Secondly, as explained in Section 3.3, the parser we work with in this paper requires full use of bracketing to avoid ambiguity. Parsers for real languages such as JavaScript do not do this, and thus do not satisfy the locality property (3) (page 7) which we have used. For example, for a JavaScript parser $\mathrm{JSParse}\,(-)$ one has $\mathrm{JSParse}\,(\text{“}3 + 4\text{”}) = \mathrm{Just}\,(3{+}4, \text{“”})$ but one does *not* have $\mathrm{JSParse}\,(\text{“}3 + 4 \times 2\text{”}) = \mathrm{Just}\,(3{+}4, \text{“}{\times}2\text{”})$.

However, such parsers do satisfy certain weaker locality properties; we might call them *weakly local* after Smith [18]. Smith uses context logic [5] to reason about weakly local operations on trees; we believe we will be able to use context logic over sequences to reason about semi-local parsing operations on strings.

Related work. Cai, Shao and Vaynberg [4] give a logic for reasoning about code generation at the assembly code level. Although we work with a completely different representation of commands, there are commonalities. In both works one needs to deal with a concrete (syntactic) representation of code and with recursion through the store. Also both works provide a means to represent infinite (but structured) sets of commands which might be generated at runtime: we use grammars for this purpose whereas in [4] “parametric code blocks” are used.

Berger and Tratt [3] give a program logic for a language $\mathrm{PCF}_{\mathrm{DP}}$ which is PCF extended with metaprogramming features. $\mathrm{PCF}_{\mathrm{DP}}$ supports runtime code generation, but does not use strings and *eval* for this purpose, and does not support testing of equality between pieces of code.

Minamide [11] presents a static analysis which, given a PHP program, computes a context-free grammar which overapproximates the contents of each string variable at each program point. This is used to verify that server-side programs produce well-formed web pages. Thiemann [19] presents a similar static analysis,

formulated as a type inference algorithm, for programs in a lambda calculus with strings. These works do not consider runtime code generation or *eval*, but from the point of view of future automation of our logic it is encouraging that such techniques already exist.

Acknowledgements This work has been supported by EPSRC grant EP/G003173/1. The author thanks Bernhard Reus and Gareth Smith for helpful discussions.

References

1. Secunia Advisory SA23834 - Vote! Pro PHP 'eval()' injection vulnerability (2007), available at http://secunia.com/advisories/23834
2. Apt, K.R.: Ten years of Hoare's logic: A survey - part I. ACM Trans. Program. Lang. Syst. 3(4), 431–483 (1981)
3. Berger, M., Tratt, L.: Program logics for homogeneous meta-programming. In: LPAR (Dakar). pp. 64–81 (2010)
4. Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. In: PLDI. pp. 66–77 (2007)
5. Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. In: POPL. pp. 271–282 (2005)
6. Charlton, N.: Hoare logic for higher order store using simple semantics. In: WoLLIC. pp. 52–66 (2011)
7. Chellapilla, K., Maykov, A.: A taxonomy of JavaScript redirection spam. In: AIRWeb (2007)
8. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM 48(3), 555–604 (2001)
9. Gardner, P., Maffeis, S., Smith, G.: A program logic for JavaScript. In: POPL (2012), to appear.
10. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: POPL. pp. 417–430 (2010)
11. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW. pp. 432–441 (2005)
12. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: POPL. pp. 320–333 (2006)
13. von Oheimb, D.: Hoare logic for mutual recursion and local variables. In: FSTTCS. pp. 168–180 (1999)
14. Reus, B., Streicher, T.: About Hoare logics for higher-order store. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP. Lecture Notes in Computer Science, vol. 3580, pp. 1337–1348. Springer (2005)
15. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do - a large-scale study of the use of eval in JavaScript applications. In: ECOOP. pp. 52–78 (2011)
16. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: PLDI. pp. 1–12 (2010)
17. Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested Hoare triples and frame rules for higher-order store. In: CSL. pp. 440–454 (2009)
18. Smith, G.D.: Local reasoning about web programs. Ph.D. thesis, Dept. of Computing, Imperial College London (2010)
19. Thiemann, P.: Grammar-based analysis of string expressions. In: TLDI. pp. 59–70 (2005)

A Parser code

See Figures 13, 14 and 15.

B Further soundness proofs

Full proof of Lemma 1.

Proof. **For (A):** To show $(\hat{\rho}, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$, it is enough to show that ρ and $\hat{\rho}$ agree on all variables appearing free in Γ. So let v appear free in Γ; it is clear that v is not one of $\mathrm{p}_1, \ldots, \mathrm{p}_M$. It also follows, by the side conditions of the rule, that v does not appear in a production for P. Thus $\hat{\rho}(\mathrm{v}) = \rho(\mathrm{v})$.

It remains to show $(\hat{\rho}, n-1) \in [\![\Gamma(\mathcal{P})]\!]^{\mathrm{sp}}$. So let Σ be an arbitrary element of $\Gamma(\mathcal{P})$, and we need to prove that $(\hat{\rho}, n-1) \in [\![\Sigma]\!]^{\mathrm{sp}}$. There are two cases:

1. Σ has the form $e = e'$. So what we need to show is $[\![e]\!]^{\mathrm{ex}}_{\hat{\rho}} = [\![e']\!]^{\mathrm{ex}}_{\hat{\rho}}$. For some $j \in \{1, \ldots, M\}$ we must have that $\mathcal{W}_j$ is $e = e'$. We know that $(\rho', l_j) \in [\![e = e']\!]_{m+1}$, and from this it follows that $[\![e]\!]^{\mathrm{ex}}_{\rho'} = [\![e']\!]^{\mathrm{ex}}_{\rho'}$. Thus it will be enought to show $[\![e]\!]^{\mathrm{ex}}_{\rho'} = [\![e]\!]^{\mathrm{ex}}_{\hat{\rho}}$ and $[\![e']\!]^{\mathrm{ex}}_{\rho'} = [\![e']\!]^{\mathrm{ex}}_{\hat{\rho}}$. But these follow from the fact that any variable v which appears in e or e' obviously occurs in a production for P, so by definition of $\hat{\rho}$ we have $\hat{\rho}(\mathrm{v}) = \rho'(\mathrm{v})$.

2. Σ has the form $\{P\}\,\mathrm{p}_j\,\{Q\}\,[\mathrm{v}_1, \ldots, \mathrm{v}_K \backslash e_1, \ldots, e_K]$ for some $j \in \{1, \ldots, M\}$. Then $\mathcal{W}_j$ must be $\mathrm{P}(e_1, \ldots, e_K)$. We know that $(\rho', l_j) \in [\![\mathrm{P}(e_1, \ldots, e_K)]\!]_{m+1}$, and from this it follows that $m \geq 0$ and

$$(l_j, [\![e_1]\!]^{\mathrm{ex}}_{\rho'}, \ldots, [\![e_K]\!]^{\mathrm{ex}}_{\rho'}) \in \chi_m(\mathrm{P})$$

What we need to show is

$$(\hat{\rho}, n-1) \in [\![\{P\}\,\mathrm{p}_j\,\{Q\}\,[\mathrm{v}_1, \ldots, \mathrm{v}_K \backslash e_1, \ldots, e_K]]\!]^{\mathrm{sp}} \tag{10}$$

We will now take advantage of the fact that $\Phi(m)$ holds. We define an environment $\rho^\dagger$ as follows.

$$\rho^\dagger(\mathrm{v}) \quad := \quad \begin{cases} [\![e_i]\!]^{\mathrm{ex}}_{\hat{\rho}} & \text{if v is } \mathrm{v}_i \\ \hat{\rho}(\mathrm{v}) & \text{otherwise} \end{cases}$$

We next show that $(\rho^\dagger, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. This follows from $(\hat{\rho}, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$, which we have already proved, since $\mathrm{v}_1, \ldots, \mathrm{v}_K$ (the only places where $\rho^\dagger$ differs from $\hat{\rho}$) cannot occur free in Γ (due to the side condition of the rule we are proving). $\Phi(m)$ is $[\![\Gamma \vdash \{P\}\,\mathrm{P}^m(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}]\!]^{\mathrm{sic}} = \mathsf{Env}$ so in particular, using this with $\rho^\dagger$ and n, we get

$$(\rho^\dagger, n) \in [\![\{P\}\,\mathrm{P}^m(\cdot, \mathrm{v}_1, \ldots, \mathrm{v}_K)\,\{Q\}]\!]^{\mathrm{sp}} \tag{11}$$

We know that $(l_j, [\![e_1]\!]^{\mathrm{ex}}_{\rho'}, \ldots, [\![e_K]\!]^{\mathrm{ex}}_{\rho'}) \in \chi_m(\mathrm{P})$. All variables in $e_1, \ldots, e_K$ clearly appear in a production for P, so we have $(l_j, [\![e_1]\!]^{\mathrm{ex}}_{\hat{\rho}}, \ldots, [\![e_K]\!]^{\mathrm{ex}}_{\hat{\rho}}) \in$

```
type VarName = [Char]

data Token =
    NopTk
  | AssignTk
  | LParenTk
  | RParenTk
  | SemiTk
  | IfTk
  | ThenTk
  | ElseTk
  | EvalTk
  | WhileTk
  | DoTk
  | IntLitTk Int
  | StrLitTk Str
  | VarTk VarName
  | PlusTk
  | MinusTk
  | TimesTk
  | EqualsTk
  | NotEqTk
  | PlusPlusTk
  | IntToStrTk

type Str = [Token]

data Expr =
    IntLiteral Int
  | StrLiteral Str
  | Plus Expr Expr
  | Minus Expr Expr
  | Times Expr Expr
  | Equals Expr Expr
  | NotEq Expr Expr
  | PlusPlus Expr Expr
  | IntToStr Expr
  | Var VarName

data Cmd =
   Nop
 | Assign VarName Expr
 | Seq Cmd Cmd
 | Ite Expr Cmd Cmd
 | While Expr Cmd
 | Eval Expr
```

Fig. 13. Parser code (1/3)

```
-- A recursive descent parsers for commands, which in turn depends on a
-- parser for expressions.

parse_exp :: Str -> Maybe (Expr, Str)

parse_exp [] = Nothing

parse_exp (t : rest) =

  case t of

    LParenTk    ->  parse_binop rest
    IntLitTk n  ->  Just (IntLiteral n, rest)
    StrLitTk s  ->  Just (StrLiteral s, rest)
    IntToStrTk  ->  parse_inttostr rest
    VarTk vname ->  Just (Var vname, rest)
    _           ->  Nothing


parse_binop :: Str -> Maybe (Expr, Str)

parse_binop s =
  case parse_exp s of

    Just (e1, PlusTk:s1) ->
      (case parse_exp s1 of
          Just (e2, RParenTk:rest) -> Just (Plus e1 e2, rest)
          _ -> Nothing
      )

    Just (e1, MinusTk:s1) ->
      (case parse_exp s1 of
          Just (e2, RParenTk:rest) -> Just (Minus e1 e2, rest)
          _ -> Nothing
      )

    Just (e1, TimesTk:s1) ->
      (case parse_exp s1 of
          Just (e2, RParenTk:rest) -> Just (Times e1 e2, rest)
          _ -> Nothing
      )

    Just (e1, EqualsTk:s1) ->
      (case parse_exp s1 of
          Just (e2, RParenTk:rest) -> Just (Equals e1 e2, rest)
          _ -> Nothing
      )

    Just (e1, NotEqTk:s1) ->
      (case parse_exp s1 of
          Just (e2, RParenTk:rest) -> Just (NotEq e1 e2, rest)
          _ -> Nothing
      )

    Just (e1, PlusPlusTk:s1) ->
      (case parse_exp s1 of
          Just (e2, RParenTk:rest) -> Just (PlusPlus e1 e2, rest)
          _ -> Nothing
      )

    _ -> Nothing


parse_inttostr :: Str -> Maybe (Expr, Str)

parse_inttostr s =

  case parse_exp s of
    Just (e, rest) -> Just (IntToStr e, rest)
    _ -> Nothing
```

Fig. 14. Parser code (2/3)

```
parse_cmd :: Str -> Maybe (Cmd, Str)

parse_cmd [] = Nothing

parse_cmd (t : rest) =

  case t of

    NopTk       ->  Just (Nop, rest)
    LParenTk    ->  parse_seq_comp rest
    IfTk        ->  parse_ite rest
    WhileTk     ->  parse_while rest
    EvalTk      ->  parse_eval rest
    VarTk vname ->  parse_assign vname rest
    _           ->  Nothing


parse_seq_comp :: Str -> Maybe (Cmd, Str)

parse_seq_comp s =
  case parse_cmd s of
    Just (c1, SemiTk:s1) ->
      (case parse_cmd s1 of
          Just (c2, RParenTk:rest) -> Just (Seq c1 c2, rest)
          _ -> Nothing
      )
    _ -> Nothing


parse_ite :: Str -> Maybe (Cmd, Str)

parse_ite s =

  case parse_exp s of
    Just (e, ThenTk:s1) ->
      (case parse_cmd s1 of
        Just (c1, ElseTk:s2) ->
          (case parse_cmd s2 of
            Just (c2, rest) -> Just (Ite e c1 c2, rest)
            Nothing -> Nothing
          )
        _ -> Nothing
      )
    _ -> Nothing


parse_while :: Str -> Maybe (Cmd, Str)

parse_while s =

  case parse_exp s of
    Just (e, DoTk:s1) ->
      (case parse_cmd s1 of
        Just (c, s) ->  Just (While e c, s)
        _ -> Nothing
      )
    _ -> Nothing


parse_eval :: Str -> Maybe (Cmd, Str)

parse_eval s =

  case parse_exp s of
    Just (e, rest) -> Just (Eval e, rest)
    _ -> Nothing


parse_assign :: VarName -> Str -> Maybe (Cmd, Str)

parse_assign vname (AssignTk:rest) =

  case parse_exp rest of
    Just (e, rest') -> Just (Assign vname e, rest')
    _ -> Nothing

parse_assign _ _ = Nothing
```

Fig. 15. Parser code (3/3)

$\chi_m(\mathrm{P})$. Then by the definition of $\rho^\dagger$ we have $(l_j, [\![\mathrm{v}_1]\!]^{\mathrm{ex}}_{\rho^\dagger}, \ldots, [\![\mathrm{v}_K]\!]^{\mathrm{ex}}_{\rho^\dagger}) \in \chi_m(\mathrm{P})$. By this and (11) we find that $parseCmd(l_j)$ has the form Just $(C, \text{“”})$ where

$$\rho^\dagger \vDash^n \{P\}\, C\, \{Q\} \tag{12}$$

We are now in a position to show (10). If $n = 0$ then just note that $\hat{\rho}(\mathrm{p}_j) = \mathscr{S}(l_j)$ which parses successfully, and we are done. So suppose $n > 0$. Because $\hat{\rho}(\mathrm{p}_j) = \mathscr{S}(l_j)$ it will suffice to show that

$$\hat{\rho} \vDash^{n-1} \{P[v_1, \ldots, v_K \backslash e_1, \ldots, e_K]\}\, C\, \{Q[v_1, \ldots, v_K \backslash e_1, \ldots, e_K]\}$$

Using familiar properties of substitution, this is equivalent to $\rho^\dagger \vDash^{n-1} \{P\}\, C\, \{Q\}$ which follows from (12).

For (B): We know that $l = l_1 +\!\!+ \cdots +\!\!+ l_M$ and $E(\mathcal{P})$ has the form $E_1(\mathcal{P}) +\!\!+ \cdots +\!\!+ E_M(\mathcal{P})$. Hence it will suffice to prove that for $i = 1..M$, $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}}) = l_i$. Recall that $(\rho', l_i) \in [\![\mathcal{W}_i]\!]_{m+1}$. There are four cases, depending on the form of $\mathcal{W}_i$:

1. $\mathcal{W}_i$ has form $\mathrm{P}(e_1, \ldots, e_K)$, and $E_i(\mathcal{P})$ is p_i. Then $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}}) = \mathscr{S}^{-1}([\![\mathrm{p}_i]\!]^{\mathrm{ex}}_{\hat{\rho}}) = \mathscr{S}^{-1}(\hat{\rho}(\mathrm{p}_i))$ which by definition of $\hat{\rho}$ is l_i as required.

2. $\mathcal{W}_i$ is a string literal l', and $E_i(\mathcal{P})$ is also l'. Then $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}})$ is l'. From $(\rho', l_i) \in [\![\mathcal{W}_i]\!]_{m+1}$ we get $l_i = l'$ and hence $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}}) = l' = l_i$.

3. $\mathcal{W}_i$ has the form $\mathrm{IntToStr}(e)$, and $E_i(\mathcal{P})$ is also $\mathrm{IntToStr}(e)$. Then $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}})$ is $\mathscr{S}^{-1}(\mathscr{S}([\mathsf{IntLitTk}\ [\![e]\!]^{\mathrm{ex}}_{\hat{\rho}}]))$ which is $[\mathsf{IntLitTk}\ [\![e]\!]^{\mathrm{ex}}_{\hat{\rho}}]$. From $(\rho', l_i) \in [\![\mathcal{W}_i]\!]_{m+1}$ we get $l_i = [\mathsf{IntLitTk}\ [\![e]\!]^{\mathrm{ex}}_{\rho'}]$. We'll be done if we can show $[\![e]\!]^{\mathrm{ex}}_{\hat{\rho}} = [\![e]\!]^{\mathrm{ex}}_{\rho'}$. But this follows from the fact that any variable v which appears in e obviously occurs in a production for P, so by definition of $\hat{\rho}$ we have $\hat{\rho}(\mathrm{v}) = \rho'(\mathrm{v})$.

4. $\mathcal{W}_i$ is an equality $e = e'$, and $E_i(\mathcal{P})$ is “”. Then $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}})$ is “”. From $(\rho', l_i) \in [\![\mathcal{W}_i]\!]_{m+1}$ we get $l_i =$ “” and hence $\mathscr{S}^{-1}([\![E_i(\mathcal{P})]\!]^{\mathrm{ex}}_{\hat{\rho}}) =$ “” $= l_i$.

For (C): Let v be a variable occurring free in P or Q. By the side condition of (GrInd), v cannot appear privately in a production for P. Also v cannot be one of $\mathrm{p}_1, \ldots, \mathrm{p}_M$ as these were chosen to be unused variables. This leaves two possibilities.

1. v is one of $\mathrm{v}_1, \ldots, \mathrm{v}_K$. Then by definition of $\hat{\rho}$, $\hat{\rho}(\mathrm{v}) = \rho'(\mathrm{v})$. To finish note that ρ' agrees with ρ on $\mathrm{v}_1, \ldots, \mathrm{v}_K$.
2. v does not appear in a production for P and is not one of $\mathrm{p}_1, \ldots, \mathrm{p}_M$. Then by the definition of $\hat{\rho}$ we have $\hat{\rho}(\mathrm{v}) = \rho(\mathrm{v})$ as required. □

Theorem 3. *The (μ) rule is sound.*

Proof. Let $\Theta(n)$ (for $n \in \mathbb{N}_{-1}$) be the following statement: for all $\rho \in \mathsf{Env}$,

$$(\rho, n) \in [\![\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}]\!]^{\mathrm{sp}}$$

We shall use induction to prove $\Theta(n)$ for all $n \in \mathbb{N}_{-1}$; from this the desired conclusion follows easily.

Base case: We need to show $\Theta(-1)$, that is, we must show that for an arbitrary $\rho \in \mathsf{Env}$ we have $(\rho, -1) \in [\![\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}]\!]^{\mathrm{sp}}$. This amounts to showing that, for all strings l, $(l, [\![e_1]\!]^{\mathrm{ex}}_\rho, \ldots, [\![e_k]\!]^{\mathrm{ex}}_\rho) \in \chi(\mathrm{P})$ implies that $\mathit{parseCmd}(l)$ has the form Just $(C, \text{“”})$.

So let l be a string such that $(l, [\![e_1]\!]^{\mathrm{ex}}_\rho, \ldots, [\![e_k]\!]^{\mathrm{ex}}_\rho) \in \chi(\mathrm{P})$. From the first premise, $\{\mathit{true}\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{\mathit{true}\}$, just unpacking definitions, it follows that $\mathit{parseCmd}(l)$ has the form Just $(C, \text{“”})$.

Inductive case: Suppose $\Theta(n)$ holds for $n \in \mathbb{N}_{-1}$; we will prove $\Theta(n+1)$. So let $\rho \in \mathsf{Env}$. We shall first show

$$(\rho, n) \in [\![\forall \vec{\mathrm{N}}.\,\{P\}\,\mathrm{P}(e_1, \ldots, e_k)\,\{Q\}]\!]^{\mathrm{sp}} \tag{13}$$

Let $\rho' \in \mathsf{Env}$ agree with ρ except possibly at $\vec{\mathrm{N}}$. Let l be a string such that

$$(l, [\![e_1]\!]^{\mathrm{ex}}_{\rho'}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{\rho'}) \in \chi(\mathrm{P})$$

Instantiating the induction hypothesis $\Theta(n)$ with ρ' for ρ, we see that $\mathit{parseCmd}(l)$ has the form Just $(C, \text{“”})$ where either $n = -1$, or $n \geq 0$ and $\rho' \vDash^n \{P\}\,C\,\{Q\}$. (13) follows from this. From (13) and the second premise we obtain $(\rho, n+1) \in [\![\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}]\!]^{\mathrm{sp}}$ which is exactly what we needed to prove. □

Theorem 4. *The (H) rule is sound.*

Proof. Suppose the premise of the rule holds. Let $\rho \in \mathsf{Env}$, $n \in \mathbb{N}$ be such that $(\rho, n-1) \in [\![\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}]\!]^{\mathrm{sp}}$; we are required to prove that $(\rho, n) \in [\![\{P\}\,\mathsf{eval}\ e\,\{Q\}]\!]^{\mathrm{sp}}$. So we need to show $\rho \vDash^n \{P\}\,\mathsf{eval}\ e\,\{Q\}$.

So suppose $s \in [\![P]\!]^{\mathrm{as}}_\rho$ and suppose further that $(\mathsf{eval}\ e, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer. We are required to show $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$. From $s \in [\![P]\!]^{\mathrm{as}}_\rho$ and $P \Rightarrow \mathrm{P}(e, e_1, \ldots, e_k)$ we deduce that

$$(\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{s,\rho}), [\![e_1]\!]^{\mathrm{ex}}_{s,\rho}, \ldots, [\![e_k]\!]^{\mathrm{ex}}_{s,\rho}) \in \chi(\mathrm{P})$$

It follows from this and $(\rho, n-1) \in [\![\{P\}\,\mathrm{P}(\cdot, e_1, \ldots, e_k)\,\{Q\}]\!]^{\mathrm{sp}}$ that $\mathit{parseCmd}(\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{s,\rho}))$ has the form Just $(C, \text{“”})$, where

$$\rho \vDash^{n-1} \{P\}\,C\,\{Q\} \tag{14}$$

By the structure of the transition relation, we must have $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in $n-1$ steps or fewer. We cannot have $n = 0$ (because an execution sequence of lenght -1 doesn't exist) so $n \geq 1$; then it comes from (14), by unpacking the definition of $\vDash$, that $s' \in [\![Q]\!]^{\mathrm{as}}_\rho$ as required. □

Theorem 5. *The (Parsed) rule is sound.*

Proof. Assume that the premises of the rule hold. Let $\rho \in \mathsf{Env}$, $n \in \mathbb{N}$ be such that $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. We must prove that $(\rho, n) \in [\![\{P\}\, l\, \{Q\}]\!]^{\mathrm{sp}}$. This amounts to showing that $\mathit{parseCmd}(l)$ has the form Just $(C', \text{“”})$ where $\rho \vDash^n \{P\}\, C'\, \{Q\}$. But the first premise is that $\mathit{parseCmd}(l) = \text{Just } (C, \text{“”})$ so it will suffice to prove that $\rho \vDash^n \{P\}\, C\, \{Q\}$. We derive this by combining the second premise with $(\rho, n-1) \in [\![\Gamma]\!]^{\mathrm{sp}}$. □

Theorem 6. *The (R) rule is sound.*

Proof. (R) is a derived rule, obtained using (H) and (Cut). □

Theorem 7. *The (ParsedA) rule is sound.*

Proof. Assume $\mathit{rep}(e, e')$. We must show $\{P[x\backslash e']\}$ “$x :=$” $+\!\!+ e\, \{P\}$. So let $\rho \in \mathsf{Env}$ and $n \in \mathbb{N}$; we are required to show $(\rho, n) \in [\![\{P[x\backslash e']\}$ “$x :=$” $+\!\!+ e\, \{P\}]\!]^{\mathrm{sp}}$.

From the semantics of *rep*, we find that $\mathit{parseExp}(\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{\rho}))$ has the form Just $(E, \text{“”})$ where all variables in E are program variables, and furthermore, for all $s \in \mathsf{Store}$, $[\![E]\!]^{\mathrm{ex}}_{s} = [\![e']\!]^{\mathrm{ex}}_{s,\rho}$. We then define C to be the command $x := E$. We will be done if we can prove the following two things:

(A) $\mathit{parseCmd}(\mathscr{S}^{-1}([\![\text{“}x :=\text{”} +\!\!+ e]\!]^{\mathrm{ex}}_{\rho})) = \text{Just } (C, \text{“”})$

(B) $\rho \vDash^n \{P[x\backslash e']\}\, C\, \{P\}$

(A) follows from the definition of *parseCmd*, using the fact that $\mathit{parseExp}(\mathscr{S}^{-1}([\![e]\!]^{\mathrm{ex}}_{\rho})) = \text{Just } (E, \text{“”})$.

We finish by proving (B). Let $s \in [\![P[x\backslash e']]\!]^{\mathrm{as}}_{\rho}$ and suppose that $(C, s) \xrightarrow{*} (\mathsf{nop}, s')$ in n steps or fewer. We need to show $s' \in [\![P]\!]^{\mathrm{as}}_{\rho}$. By the structure of the transition relation, we must get from (C, s) to (nop, s') in a single step, and we must have $s' = s[x := [\![E]\!]^{\mathrm{ex}}_{s}]$. It follows from this and $[\![E]\!]^{\mathrm{ex}}_{s} = [\![e']\!]^{\mathrm{ex}}_{s,\rho}$ that $s' = s[x := [\![e']\!]^{\mathrm{ex}}_{s,\rho}]$. From this and $s \in [\![P[x\backslash e']]\!]^{\mathrm{as}}_{\rho}$ it follows, by familiar properties of substitution, that $s' \in [\![P]\!]^{\mathrm{as}}_{\rho}$ as required. □

Theorem 8. *The (ParsedS) rule is sound.*

Proof. Let $\rho \in \mathsf{Env}$ and let $n \in \mathbb{N}$ be such that $(\rho, n-1) \in [\![\{P\}\, e_1\, \{R\}\, ,\, \{R\}\, e_2\, \{Q\}]\!]^{\mathrm{sp}}$. We need to prove that $(\rho, n) \in [\![\{P\}$ “(” $+\!\!+ e_1 +\!\!+$ “;” $+\!\!+ e_2 +\!\!+$ “)” $\{Q\}]\!]^{\mathrm{sp}}$. This amounts to showing that $\mathit{parseCmd}(\mathscr{S}^{-1}([\![\text{“(”} +\!\!+ e_1 +\!\!+ \text{“;”} +\!\!+ e_2 +\!\!+ \text{“)”}]\!]^{\mathrm{ex}}_{\rho}))$ has the form Just $(C, \text{“”})$ such that $\rho \vDash^n \{P\}\, C\, \{Q\}$.

From $(\rho, n-1) \in [\![\{P\}\, e_1\, \{R\}]\!]^{\mathrm{sp}}$ it follows that there exists C_1 such that $\mathit{parseCmd}(\mathscr{S}^{-1}([\![e_1]\!]^{\mathrm{ex}}_{\rho})) = C_1$ and either $n = 0$ or $\rho \vDash^{n-1} \{P\}\, C_1\, \{R\}$. Similarly there exists C_2 such that $\mathit{parseCmd}(\mathscr{S}^{-1}([\![e_2]\!]^{\mathrm{ex}}_{\rho})) = C_2$ and either $n = 0$ or $\rho \vDash^{n-1} \{R\}\, C_2\, \{Q\}$.

We define C to be the command $C_1\ ;\ C_2$. Using the definition of our parser, the parser's locality property (3) (page 7) and the above equations for $\mathit{parseCmd}(\mathscr{S}^{-1}([\![e_1]\!]^{\mathrm{ex}}_{\rho}))$ and $\mathit{parseCmd}(\mathscr{S}^{-1}([\![e_2]\!]^{\mathrm{ex}}_{\rho}))$ it follows that

$$\mathit{parseCmd}(\mathscr{S}^{-1}([\![\text{“(”} +\!\!+ e_1 +\!\!+ \text{“;”} +\!\!+ e_2 +\!\!+ \text{“)”}]\!]^{\mathrm{ex}}_{\rho})) = C_1; C_2 = C$$

as required. If $n = 0$ then $\rho \vDash^n \{P\}\, C\, \{Q\}$ holds trivially, because a sequential composition cannot reach a terminal configuration in 0 steps.

So suppose $n > 0$. To show $\rho \vDash^n \{P\}\, C\, \{Q\}$, let

$$(C_1; C_2, s^1) \to \cdots \to (\mathsf{nop}, s^K)$$

be an execution sequence such that $1 < K \leq n + 1$ (so the execution consists of at most n steps) and $s^1 \in [\![P]\!]^{\mathrm{as}}_{\rho}$. This execution sequence must have the form

$$(C_1; C_2, s^1) \to \cdots \to (\mathsf{nop}; C_2, s^J) \to (C_2, s^{J+1}) \to \cdots \to (\mathsf{nop}, s^K)$$

where $1 \leq J < K$ and $s^J = s^{J+1}$, and there must exist another execution sequence, of $J - 1$ steps, of the form

$$(C_1, s^1) \to \cdots \to (\mathsf{nop}, s^J)$$

By $\rho \vDash^{n-1} \{P\}\, C_1\, \{R\}$ we see that $s^J = s^{J+1} \in [\![R]\!]^{\mathrm{as}}_{\rho}$. Then applying $\rho \vDash^{n-1} \{R\}\, C_2\, \{Q\}$ to the execution sequence

$$(C_2, s^{J+1}) \to \cdots \to (\mathsf{nop}, s^K)$$

we obtain $s^K \in [\![Q]\!]^{\mathrm{as}}_{\rho}$ as required. □

Theorem 9. *The rules for rep in Fig. 8 are sound.*

Proof.

First rule:

$$\frac{}{\mathit{rep}(\text{“}x\text{”}, x)}$$

Let ρ, s be arbitrary. “x” is shorthand for $[\mathsf{VarTk}\ x]$, so we have:

$$\begin{array}{rcl} \mathit{parseExp}(\mathscr{S}^{-1}([\![\text{“}x\text{”}]\!]^{\mathrm{ex}}_{s,\rho})) & = & \mathit{parseExp}(\mathscr{S}^{-1}([\![[\mathsf{VarTk}\ x]]\!]^{\mathrm{ex}}_{s,\rho})) \\ & = & \mathit{parseExp}(\mathscr{S}^{-1}(\mathscr{S}([\mathsf{VarTk}\ x]))) \\ & = & \mathit{parseExp}([\mathsf{VarTk}\ x]) \\ & = & \mathrm{Just}(x, \text{“”}) \end{array}$$

It remains to show $[\![x]\!]^{\mathrm{ex}}_{s} = [\![x]\!]^{\mathrm{ex}}_{s,\rho}$ which is trivial.

Second rule:

$$\frac{}{\mathit{rep}(\mathrm{IntToStr}(e), e)}$$

Let ρ, s be arbitrary. Let c be the constant $[\![e]\!]^{\mathrm{ex}}_{s,\rho}$. We have:

$$\begin{array}{rcl} \mathit{parseExp}(\mathscr{S}^{-1}([\![\mathrm{IntToStr}(e)]\!]^{\mathrm{ex}}_{s,\rho})) & = & \mathit{parseExp}(\mathscr{S}^{-1}(\mathscr{S}([\mathsf{IntLitTk}\ [\![e]\!]^{\mathrm{ex}}_{s,\rho}]))) \\ & = & \mathit{parseExp}([\mathsf{IntLitTk}\ [\![e]\!]^{\mathrm{ex}}_{s,\rho}]) \\ & = & \mathit{parseExp}([\mathsf{IntLitTk}\ c]) \\ & = & \mathrm{Just}(c, \text{“”}) \end{array}$$

It remains to show $[\![c]\!]^{\mathrm{ex}}_s = [\![e]\!]^{\mathrm{ex}}_{s,\rho}$ which is trivial.

Third rule:

$$\frac{rep(e_1, e_1') \qquad rep(e_2, e_2')}{rep(\text{“(”} +\!\!+ e_1 +\!\!+ \text{“}\oplus\text{”} +\!\!+ e_2 +\!\!+ \text{“)”}, e_1' \oplus e_2')}$$

Assume the premises hold. Let ρ, s be arbitrary.

From $rep(e_1, e_1')$ it follows that $parseExp(\mathscr{S}^{-1}([\![e_1]\!]^{\mathrm{ex}}_{s,\rho}))$ has the form Just $(E_1, \text{“”})$ where all variables in E_1 are program variables, and furthermore, $[\![E_1]\!]^{\mathrm{ex}}_s = [\![e_1']\!]^{\mathrm{ex}}_{s,\rho}$. Similarly from $rep(e_2, e_2')$ it follows that $parseExp(\mathscr{S}^{-1}([\![e_2]\!]^{\mathrm{ex}}_{s,\rho}))$ has the form Just $(E_2, \text{“”})$ where all variables in E_2 are program variables, and furthermore, $[\![E_2]\!]^{\mathrm{ex}}_s = [\![e_2']\!]^{\mathrm{ex}}_{s,\rho}$.

Thus we see that $parseExp(\mathscr{S}^{-1}([\![\text{“(”} +\!\!+ e_1 +\!\!+ \text{“+”} +\!\!+ e_2 +\!\!+ \text{“)”}]\!]^{\mathrm{ex}}_{s,\rho}))$ has the form Just $(E_1 + E_2, \text{“”})$, and

$$[\![E_1 + E_2]\!]^{\mathrm{ex}}_s \;=\; [\![E_1]\!]^{\mathrm{ex}}_s + [\![E_2]\!]^{\mathrm{ex}}_s \;=\; [\![e_1']\!]^{\mathrm{ex}}_{s,\rho} + [\![e_2']\!]^{\mathrm{ex}}_{s,\rho} \;=\; [\![e_1' + e_2']\!]^{\mathrm{ex}}_{s,\rho}$$

as required. □