Efficient Substitution in Hoare Logic Expressions

Andrew W. Appel Kedar N. Swadi

Roberto Virga

Department of Computer Science
Princeton University
Princeton, NJ, U.S.A.
Email: {appel,kswadi,rvirga}@cs.princeton.edu

Abstract

Substitution plays an important role in Hoare Logic, as it is used in interpreting assignments. When writing a computer-based realization of Hoare Logic, it is therefore important to choose a good implementation for it. In this paper we compare different definitions and implementations of substitution in a logical framework, in an effort to maximize efficiency.

We start by defining substitution as a logical formula. In a conventional approach, this is done by specifying the syntactic changes substitution performs on expressions. We choose instead a semantic definition that describes the behavioral relation between the original expression and its substituted counterpart.

Next, we use this semantic definition as an abstract specification, and compare two of its concrete implementations. The first we consider is the usual one, that operates recursively over the structure of the term. This requires a number of inference steps proportional to the size of the expression, which is unacceptable for many practical applications. We therefore propose a different method, that makes better use of the primitives provided by the logical framework, and manages to reduce the complexity to a quantity proportional to the number of free variables.

We conclude the paper outlining a refinement technique that, by taking advantage of some simple program analysis information, holds promise of improving the results presented even further.

[1] This research was supported in part by DARPA grant F30602-99-1-0519, by NSF grant CCR-9974553, and by the IBM University Partnership Program.

1 Introduction

Hoare Logic [5] is a well-established formalism for proving properties of programs. Expressions in Hoare Logic are triples

$$P \ \{S\} \ Q,$$

where P and Q are logical formulas, and S is a sequence of program instructions. The meaning of this statement is that for all states for which the precondition P is true, executing the sequence S leads us (when it terminates) to a state satisfying the postcondition Q.

In real-life programs, the length and complexity of the sequences of instructions S we need to consider makes use of Hoare Logic by hand infeasible. Hence, we often make use of *Logical Frameworks* [9], such as HOL [4] or Isabelle [8]. These software packages offer us automated or semiautomated proving facilities, so that we can concentrate on the difficult theorems while the trivial ones are automatically carried out by the system. Logical frameworks typically provide a *meta-logic* to encode mathematical formalisms (in our specific case, Hoare Logic) as *object-logics.*

The rule in Hoare Logic that deals with the assignment of a value v to a variable i can be formulated as follows:

$$[v/i]Q \ \{i := v\} \ Q$$

where the notation $[v/i]Q$ represents the substitution of all the (free) occurrences of i by v in Q. Here, the semantic change of state is reflected syntactically by the notion of substitution. When implementing Hoare Logic in a logical framework, we will therefore be interested in implementing substitution efficiently.

Most frameworks offer built-in primitives for substitution of meta-logic variables. Therefore any encoding where object-logic variables coincide with (or are closely related to) their meta-logic counterparts is bound to be efficient. However there might be cases when this implementation choice may conflict with other requirements that we impose on our encoding of Hoare Logic. Therefore we want an efficient substitution method that can be adapted to whatever representation we choose. The techniques we present in this paper come very close to fulfilling this goal, as they rely on the sole assumption that the logical formulas P and Q above are written in a higher-order calculus.

We use Hoare Logic for Proof-Carrying Code (PCC) [7,1] applications, to reason about safety of machine language instructions encoded in a logical framework [2]. In Necula's formulation of PCC, mobile code is endowed with a (Hoare Logic) proof that it conforms to a specified safety policy. In our formulation we perform substitutions, but we will also require a proof that these substitutions have been performed correctly. Moreover, we will need these correctness proofs to be small, so that they can be easily transmitted, together with the program code, over a network. This has certainly influenced our definition of efficiency of substitution, making it more restrictive: judging the efficiency of a method, we will take in account proof size too.

Throughout this paper our focus will be on Hoare triples representing assignments:

$$[v/i]Q \ \{i := v\} \ Q$$

and, more specifically, we will concern ourselves with how to obtain the precondition $[v/i]Q$ by substituting the value v for the variable i in Q.

In order to make the discussion easier to understand, we will fix the logic in which the formula Q is written. We adopt the one we used in our PCC applications, which is listed in the appendix. Although this is a higher-order calculus based on a polymorphically typed lambda calculus, it is important to stress that we never make use of type polymorphism in our presentation, and that the same techniques and results hold if applied to a different calculus, possibly even an untyped one.

We will rely on the availability of primitives for functional abstraction (‘λ’) and application (‘$\cdot$’), and hence on rules for β-conversion, defined as the smallest equivalence relation containing the reduction

$$(\lambda x.\ F)\ Z \rightarrow_\beta [Z/x]F$$

and closed with respect to the term structure.

Other primitives of our logic will be implication (‘$\Rightarrow$’) and universal quantification (‘$\forall$’). Commonly used logical connectives and quantifiers, such as negation (‘$\neg$’), conjunction (‘$\wedge$’), disjunction (‘$\vee$’), and existential quantification (‘$\exists$’) can be defined in terms of these primitives. The logical constant for falsehood (‘$\perp$’), as well as the equality predicate (‘$=$’), can be similarly introduced as a defined symbols.

To reason about assignments, we will need to introduce some additional notion, and specifically:

(a) a type for variable names, together with constants I, J, K for all the variable names used in our language;

(b) a type for values, to which expressions like “$2 * j$” will belong;

(c) a syntactical representative for the semantic notion of state.

For (c), we use *environments* σ, defined as functions from variable names to values.

Since both precondition and postcondition formulas will describe state properties, we will view them as functions from environments to truth values. Hence the expression Q will take the shape of an abstraction term $(\lambda\sigma.\ Q')$, where Q' is a formula containing the bound environment variable σ. We will furthermore require that preconditions and postconditions are allowed to refer to the environment function in a point-wise fashion. This means that all the occurrences of σ in Q' will appear inside sub-expressions of the form σI, where I is one of the variable name constants mentioned in (a). Similar conventions apply to the term v: since a value can reference the content of program variables (e.g. $v = 2 * j$ references the content of the variable j), it will be also viewed as a function, mapping environments to values.

Under these conventions, the Hoare statement

$$[2 * j/i]Q \ \{i := 2 * j\} \ Q$$

is translated into

$$(\lambda\sigma.\ [(2 * \sigma J)/\sigma I]Q') \ \{I := 2 * J\} \ (\lambda\sigma.Q'),$$

where the precondition is obtained from the postcondition by substituting all the free occurrences of σI in Q' by the term $(2 * \sigma J)$. By abuse of notation, we will use the term "variable" to also describe expressions of the form σI.

Since the expression σI is an application term (the environment σ is applied to the variable name I), it is not possible for us to use the β-reduction of the framework. Even if we were to make this possible (by switching to a different representation of the problem, for example), it might still not be advisable to do so: Mason [6] shows the pitfalls of using LF *beta* conversion to model Hoare Logic substitution. This justifies our decision to model substitution as a set of inference rules. An added bonus to this approach is that every time we perform a substitution we also obtain a proof that this operation was perfomed correctly. The problem that we attack in this paper is therefore to choose an inference system that makes best use of the primitives offered by the framework, so that substitution proofs are found quickly. If we want to retain the these proofs, we will also want to keep their size to a minimum.

2 Substitution

At this point of the discussion, we have not still specified our definition of substitution. As it turns out, we have great freedom of choice in this matter. But we have to choose wisely: the "right" specification can be of great help towards an efficient implementation; on the other hand, we do not want a definition geared too much towards one specific algorithm, since we want to use it to compare different realizations.

Substitution will be modeled as a predicate "subst" of four arguments: the name I of the variable we want to substitute, the value V we want to substitute it with, and the expressions F and F' before and after the substitution, respectively

$$\text{subst } I \ V \ F \ F'.$$

$$
\begin{array}{l}
\mathrm{subst_I}\ I\ V\ F\ F' \equiv \\
\quad \forall\ r.\ (r\ I\ V\ (\lambda\sigma.\ \sigma I)\ V) \\
\qquad \wedge\ (\forall j.\ j \neq I \Rightarrow (r\ I\ V\ (\lambda\sigma.\ \sigma j)\ (\lambda\sigma.\ \sigma j))) \\
\qquad \wedge\ (r\ I\ V\ (\lambda\sigma.\ \bot)\ (\lambda\sigma.\ \bot)) \\
\qquad \wedge\ (\forall g.\ \forall g'. \\
\qquad\qquad (r\ I\ V\ (\lambda\sigma.\ g)\ (\lambda\sigma.\ g')) \\
\qquad\qquad\quad \Rightarrow (r\ I\ V\ (\lambda\sigma.\ \neg g)\ (\lambda\sigma.\ \neg g'))) \\
\qquad \vdots \\
\qquad \wedge\ (\forall g.\ \forall g'.\ \forall x.\ \forall x'. \\
\qquad\qquad (r\ I\ V\ (\lambda\sigma.\ g)\ (\lambda\sigma.\ g')) \wedge (r\ I\ V\ (\lambda\sigma.\ x)\ (\lambda\sigma.\ x')) \\
\qquad\qquad\quad \Rightarrow (r\ I\ V\ (\lambda\sigma.\ g\ x)\ (\lambda\sigma.\ g'\ x'))) \\
\quad\ \Rightarrow (r\ I\ V\ F\ F')
\end{array}
$$

Fig. 1. “Inductive” definition of substitution

We will require this notion to be *adequate*, i.e the statement

$$\text{subst}\ I\ V\ (\lambda\sigma.\ Q')\ (\lambda\sigma.\ [(V\ \sigma)/\sigma I]Q')$$

should be provable within our logical calculus.

Example 2.1 For the Hoare statement in the previous section,

$$(\lambda\sigma.\ [(2 * \sigma J)/\sigma I]Q')\ \{I := 2 * J\}\ (\lambda\sigma. Q'),$$

adequacy translates in the natural requirement that

$$\text{subst}\ I\ (\lambda\sigma.\ 2 * \sigma J)\ (\lambda\sigma.\ Q')\ (\lambda\sigma.\ [(2 * \sigma J)/\sigma I]Q')$$

holds.

A first, fairly intuitive definition of substitution proceeds by induction on the structure of the formula. Since our logic does not provide primitives for structural induction, we use the induction-less definition $\mathrm{subst_I}$ shown in Figure 1. Roughly, we describe substitution as the smallest relation r satisfying all the inductive hypotheses.

Lemma 2.2 *The definition of* $\mathrm{subst_I}$ *is adequate.*

Proof. Using the definition in Figure 1, we can construct inductive rules for

$$
\begin{aligned}
&\mathrm{subst_S}\ I\ V\ F\ F' \equiv \\
&\quad \forall\sigma.\ \forall\sigma'.\ (\forall j.\ j \neq I \Rightarrow\ \sigma j = \sigma' j) \wedge (\sigma' I = (V\ \sigma)) \\
&\qquad\qquad \Rightarrow (F\ \sigma') = (F'\ \sigma)
\end{aligned}
$$

Fig. 2. Semantic definition of substitution

deriving $\mathrm{subst_I}$ statements, e.g.

$$
\begin{aligned}
&\forall g.\ \forall g'. \\
&\quad \mathrm{subst_I}\ I\ V\ (\lambda\sigma.\ g)\ (\lambda\sigma.\ g') \\
&\qquad \Rightarrow (\mathrm{subst_I}\ I\ V\ (\lambda\sigma.\ \neg g)\ (\lambda\sigma.\ \neg g'))
\end{aligned}
$$

Using these rules, we prove the statement arguing on the structure of Q'. □

There are two problems with this approach. First, inductive specifications usually force recursive implementations, and we are striving for a notion which is as implementation-agnostic as possible. The second problem is that the definition of $\mathrm{subst_I}$ depends heavily on the set of primitive symbols we use in our logic. If we add a new primitive, we need to modify the formula defining $\mathrm{subst_I}$, and consequently all the proofs that rely on it.

We propose a different approach, more semantical in nature, as it appeals to the concept of environment. Informally, we define $\mathrm{subst_S}\ I\ V\ F\ F'$ in term of the behavior of F and F' when evaluated over some environments σ and σ' agreeing on all variable names except I. The precise definition can be found in Figure 2.

Lemma 2.3 *The definition of* $\mathrm{subst_S}$ *is adequate.*

Proof. Similarly to the proof of Lemma 2.2, we give rules to derive $\mathrm{subst_S}$. For example, let us show

$$\mathrm{subst_S}\ I\ V\ (\lambda\sigma.\ \sigma I)\ V.$$

Unfolding the definitions, we have to show

$$
\begin{aligned}
&\forall\sigma.\ \forall\sigma'.\ (\forall j.\ (j \neq I \Rightarrow\ \sigma j = \sigma' j)) \wedge (\sigma' I = (V\ \sigma)) \\
&\qquad\qquad \Rightarrow ((\lambda\sigma.\ \sigma I)\ \sigma') = (V\ \sigma)
\end{aligned}
$$

where the conclusion follows directly from the second hypothesis by a single β-reduction step. □

We may ask ourselves how the inductive and semantic definitions are related. It turns out that the latter is in general a weaker notion than the former. The two, however, can be proven to coincide if the calculus has the following additional properties:

- *Progress*

  An assignment instruction is always executable. In other words, given any environment σ, variable name I, and value V, we can always produce an environment σ' obtained from σ by updating the content of the variable name I to $(V\ \sigma)$. In symbols:

  $$\forall \sigma.\ \exists \sigma'.\ (\forall j.\ j \neq I \Rightarrow \sigma' j = \sigma j) \wedge (\sigma' I = (V\ \sigma))$$

- *Extensionality*

  If two functions are equal point-wise, then they are equal:

  $$\forall f.\ \forall g.\ (\forall x.\ f\ x = g\ x) \Rightarrow f = g$$

Lemma 2.4 *For all I, V, F, and F', the following holds:*

$$\mathrm{subst_I}\ I\ V\ F\ F' \Rightarrow \mathrm{subst_S}\ I\ V\ F\ F'.$$

The reverse implication is true, provide progress of assignments holds and the calculus is extensional.

Proof. Assume $\mathrm{subst_I}\ I\ V\ F\ F'$, instantiating the variable r in the definition with $\mathrm{subst_S}$, and using the same inductive rules we constructed in the proof of Lemma 2.3, we show $\mathrm{subst_S}\ I\ V\ F\ F'$.

Conversely, assume $\mathrm{subst_S}\ I\ V\ F\ F'$ and let σ be any environment. By Progress, there is σ' such that $F\ \sigma' = F'\ \sigma$.

If we abstract all the occurrences of σI in F,

$$F = (\lambda \sigma.\ (\lambda x.\ \hat{F})\ \sigma I)$$

we have

$$F'\ \sigma = F\ \sigma' = (\lambda \sigma.\ (\lambda x.\ \hat{F})\ \sigma I)\ \sigma' = (\lambda \sigma.\ (\lambda x.\ \hat{F})\ (V\ \sigma))\ \sigma$$

where the last equality follows from the fact that σ' was obtained from σ by updating the content of I to $(V\ \sigma)$.

Using Extensionality, we conclude

$$F' = (\lambda \sigma.\ (\lambda x.\ \hat{F})\ (V\ \sigma))$$

Applying Lemma 2.2,

$$\mathrm{subst_I}\ I\ V\ (\lambda \sigma.\ (\lambda x.\ \hat{F})\ \sigma I)\ (\lambda \sigma.\ (\lambda x.\ \hat{F})\ (V\ \sigma)),$$

and hence, replacing equals by equals, we conclude $\mathrm{subst_I}\ I\ V\ F\ F'$, which is what we intended to prove. □

$$\frac{}{\text{subst}\ I\ V\ (\lambda\sigma.\ \sigma\ I)\ V}\,\text{subst}_{=} \qquad \frac{J \neq I}{\text{subst}\ I\ V\ (\lambda\sigma.\ \sigma\ I)\ (\lambda\sigma.\ \sigma\ I)}\,\text{subst}_{\neq}$$

$$\frac{\text{subst}\ I\ V\ (\lambda\sigma.\ F)\ (\lambda\sigma.\ F') \qquad \text{subst}\ I\ V\ (\lambda\sigma.\ G)\ (\lambda\sigma.\ G')}{\text{subst}\ I\ V\ (\lambda\sigma.\ F \Rightarrow G)\ (\lambda\sigma.\ F' \Rightarrow G')}\,\text{subst}_{\Rightarrow}$$

$$\frac{\forall x.\ \text{subst}\ I\ V\ (\lambda\sigma.\ F)\ (\lambda\sigma.\ F')}{\text{subst}\ I\ V\ (\lambda\sigma.\ \forall x.\ F)\ (\lambda\sigma.\ \forall x.\ F')}\,\text{subst}_{\forall}$$

$$\frac{\forall x.\ \text{subst}\ I\ V\ (\lambda\sigma.\ F)\ (\lambda\sigma.\ F')}{\text{subst}\ I\ V\ (\lambda\sigma.\ \lambda x.\ F)\ (\lambda\sigma.\ \lambda x.\ F')}\,\text{subst}_{\lambda}$$

$$\frac{\text{subst}\ I\ V\ (\lambda\sigma.\ F)\ (\lambda\sigma.\ F') \qquad \text{subst}\ I\ V\ (\lambda\sigma.\ G)\ (\lambda\sigma.\ G')}{\text{subst}\ I\ V\ (\lambda\sigma.\ F\ G)\ (\lambda\sigma.\ F'\ G')}\,\text{subst}_{\text{ap}}$$

$$\frac{\text{subst}\ I\ V\ (\lambda\sigma.\ F)\ (\lambda\sigma.\ F') \qquad \text{subst}\ I\ V\ (\lambda\sigma.\ G)\ (\lambda\sigma.\ G')}{\text{subst}\ I\ V\ (\lambda\sigma.\ F > G)\ (\lambda\sigma.\ F' > G')}\,\text{subst}_{>}$$

$$\frac{n \in \mathcal{N}}{\text{subst}\ I\ V\ (\lambda\sigma.\ n)\ (\lambda\sigma.\ n)}\,\text{subst}_{\mathcal{N}}$$

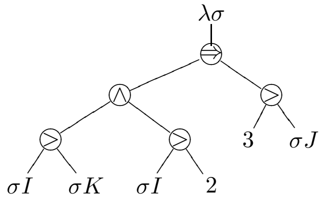
Fig. 3. Rules for recursive traversal

We adopt this latter, semantic version of substitution throughout the rest of this paper. For convenience, we will drop the subscript and write "subst" for "subst_{S}".

3 Substitution Procedures

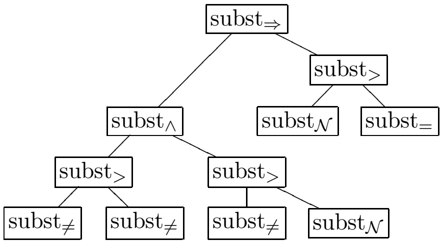
The definition of subst given in the previous section can be seen as a logical specification: it states the properties that we want substitution to satisfy, but does not tell us how to compute it. In this section we will present two different implementations that can be proved to satisfy this specification.

A first, straightforward one operates recursively on the structure of the formula. Unfortunately, a recursive implementation is not very good, both with respect to execution time and size of the proofs generated. To explain why, let us consider the expression F with the following tree representation:



and assume we want to substitute the numerical constant 2 for the variable σJ. The rules for the recursive replacement method are listed in Figure 3; they are the usual collection, introduced in the proof of Lemma 2.3, together with two new ones that deal with numbers and inequality, respectively.

The proof object generated using this inference system will be of the form:



Note that the structure of the proof term reflects that of the term F. In general, both time and proof size for this method will be proportional to the size of the expression. For large expressions, and/or for long sequences of substitutions, this is unacceptable.

Observe that σJ does not appear free in the left branch of F. If this information was available to us, we could avoid traversing it, saving in both time and proof size. Unfortunately it is not clear that information about the free variables of a subexpression can be computed without recursively traversing that subexpression. Also, since the set of free variables of an expression changes as a result of a substitution, it appears that such information would have to be recomputed after each substitution.

Searching for a better method, the key idea is to find a way to make use of the β-reduction and substitution primitives the logical framework puts at our disposal. Since substitution of meta-variables is implemented natively by the framework, it will be fast. Also, because β-reduction is one of the "trusted" operations, a method that is based on it will have to do less work in trying to convince us of its correctness, and hence will potentially generate shorter proofs.

Unfortunately, as we observed before, the substitution primitives of the framework substitute variables, while we are interested in replacing application terms σI. A first step in devising a new method is therefore to make all these application terms into (framework) variables. Consider a formula $F \equiv$

$(\lambda\sigma.\ \hat{F})$, where $\hat{F}$ contains one or more occurrences of the variable σI. Using β-expansion, we can abstract out all these occurrences and write $\hat{F}$ as

$$\hat{F} = (\lambda x.\ F_1)\ \sigma I.$$

If we repeat this operation for all the variables $I_1, \ldots, I_n$ appearing in G, we have

$$F = \lambda\sigma.\ \hat{F} = \lambda\sigma.\ (\lambda x_1.\ \ldots \lambda x_n.\ F_n)\ \sigma I_1\ \ldots\ \sigma I_n,$$

where, by what we stipulated at the beginning (σ is used only point-wise), the term F_n will not contain any occurrence of σ.

Let let_0 and let_1 be defined as follows:

$$\begin{array}{l} \mathrm{let}_1\ I\ G \equiv (\lambda\sigma.\ G\ (\sigma I)\ \sigma) \\ \mathrm{let}_0\ I\ G \equiv (\lambda\sigma.\ G). \end{array}$$

With these new definitions, the formula F above can be written as

$$F = \mathrm{let}_1\ I_1\ (\lambda x_1.\ \mathrm{let}_1\ I_2\ (\lambda x_2.\ \ldots \mathrm{let}_1\ I_n\ (\lambda x_n.\ \mathrm{let}_0\ F_n) \ldots))$$

We will use the compact notation

$$[\![I_1^{x_1}, \ldots, I_n^{x_n}]\!]\ F_n$$

for the above let-formula.

The let constructs are easily shown to satisfy the following basic properties:

- *Permutation*

  The position in which variables appear in a let-sequence can be permuted:

$$[\![\ldots, I^x, J^y, \ldots]\!]\ F = [\![\ldots, J^y, I^x, \ldots]\!]\ F$$

- *Absorption*

  Multiple occurrences of the same variable name can be compacted into one:

$$[\![\ldots I^x, I^y, \ldots]\!]\ F = [\![\ldots, I^x, \ldots]\!]\ [x/y]F$$

- *Elimination*

  If a let-bound variable no longer appears in an expression, the binding can be removed from the let-sequence:

$$[\![\ldots, I^x, \ldots]\!]\ F = [\![\ldots \ldots]\!]\ F \quad (x \notin \mathcal{FV}(F))$$

Using these simple facts, it is easy to see that let-sequences can be identified with the set of free variables of an expression: if a variable σI does not appear in F, the corresponding entry I^x can be removed from the let (Elimination);

$$\frac{}{\text{subst } I\ V\ (\llbracket\rrbracket\ F)\ (\llbracket\rrbracket\ F)}\text{subst}_{\text{let}_0}$$

$$\frac{J \neq I \quad \forall x.\ \text{subst } I\ V\ (\llbracket I_1^{x_1} \ldots I_m^{x_m}\rrbracket\ F)\ (\llbracket J_1^{y_1} \ldots J_n^{y_n}\rrbracket\ F')}{\text{subst } I\ V\ (\llbracket J^x, I_1^{x_1} \ldots I_m^{x_m}\rrbracket\ F)\ (\llbracket J^x, J_1^{y_1} \ldots J_n^{y_n}\rrbracket\ F')}\text{subst}^{\neq}_{\text{let}_1}$$

$$\frac{J = I \ \ \forall z_1.\ \ldots \forall z_l.\ \text{subst } I\ (\llbracket\rrbracket\ V)\ (\llbracket I_1^{x_1} \ldots I_m^{x_m}\rrbracket\ [V/x]F)\ (\llbracket J_1^{y_1} \ldots J_n^{y_n}\rrbracket\ F')}{\text{subst } I\ (\llbracket K_1^{z_1} \ldots K_l^{z_l}\rrbracket\ V)\ (\llbracket J^x, I_1^{x_1} \ldots I_m^{x_m}\rrbracket\ F)\ (\llbracket K_1^{z_1} \ldots K_l^{z_l}, J_1^{y_1} \ldots I_n^{y_n}\rrbracket\ F')}\text{subst}^{=}_{\text{let}_1}$$

Fig. 4. Substitution rules for let-formulas

moreover, let-sequences are order-less (Permutation) and without duplicates (Absorption).

Assuming both F and V are written as let-expressions, subst $I\ V\ F\ F'$ can now be computed with a better method, illustrated by the inference system of Figure 4.

Example 3.1 Let

$$F \equiv \lambda\sigma.\ \sigma I > \sigma K \Rightarrow 3 > \sigma J$$

and assume we want to replace 2 for σJ. Transforming everything into let notation, we want to find G_0 such that

$$\text{subst } J\ (\llbracket\rrbracket\ 2)\ (\llbracket I^x, J^y, K^z\rrbracket\ x > z \Rightarrow 3 > y)\ G_0.$$

We use the inference system of Figure 4 in a backward-chaining fashion, using the inference rules to reduce our initial statement to progressively smaller goals.

We notice that, letting $G_0 \equiv (\text{let}_1\ I\ (\lambda x.\ G_1))$, our thesis matches the conclusion of rule $\text{subst}^{\neq}_{\text{let}_1}$. Hence we are left to prove

$$\text{subst } J\ (\llbracket\rrbracket\ 2)\ (\llbracket J^y, K^z\rrbracket\ x > z \Rightarrow 3 > y)\ G_1$$

for all x. Using rule $\text{subst}^{=}_{\text{let}_1}$, we further reduce this to

$$\text{subst } J\ (\llbracket\rrbracket\ 2)\ (\llbracket K^z\rrbracket\ x > z \Rightarrow 3 > 2)\ G_1.$$

This matches again with the conclusion of rule $\text{subst}^{\neq}_{\text{let}_1}$, so that we are left to prove

$$\text{subst } J\ (\llbracket\rrbracket\ 2)\ (\llbracket\rrbracket\ x > z \Rightarrow 3 > 2)\ G_2,$$

where $G_1 \equiv (\text{let}_1\ K\ (\lambda z.\ G_2))$. Finally, we can solve this goal using $\text{subst}_{\text{let}_0}$

with $G_2 \equiv ([\![]\!]\ x > z \Rightarrow 3 > 2)$. Putting all together, we conclude

$$G_0 \equiv ([\![I^x, Z^z]\!]\ x > z \Rightarrow 3 > 2).$$

Using this approach both time and proof size become proportional to the number of variables in the formula F. Even if we account for the time required to convert a generic formula to the let representation (this will be proportional to the size of the formula), in scenarios where we need to apply (sequentially) several substitutions the new method presents clear advantages over the recursive one presented before.

4 Empirical Results

To compare the practical performance of the two methods presented in the previous section, we implemented in the Twelf logical framework [10] an automated prover that constructs substitution proofs using the rules of either system (Figures 3 and 4). We ran this prover on nine test cases, varying in both size and number of variables. Figure 5 compares the sizes of the proofs generated by the two methods. The data we collected verified our claim that the method based on the let-representation produces proofs which are proportional in size to the number of variables of the expression; the exact ratio between these two quantities turned out to be 10.

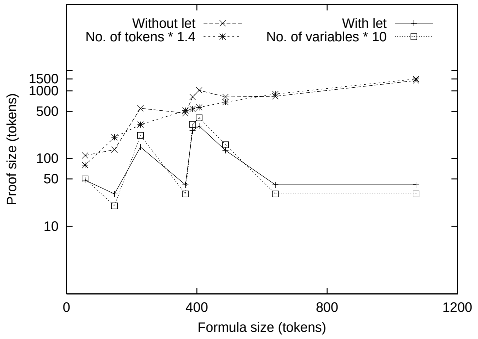


Fig. 5. Proof size

The method based on let also offered substantial savings with respect to execution time. Again, this turned out to be proportional to the number of variables with a ratio of approximately 1/20 (Figure 6).

Asymptotically, we expect the time gains of the let formulation to become less significant than what it might appear looking at Figure 6. Our method just converts object-logic substitutions to meta-logical ones; while in all examples we considered the time required by the latter was negligible, as we increase the expression size these are bound to impact performance.

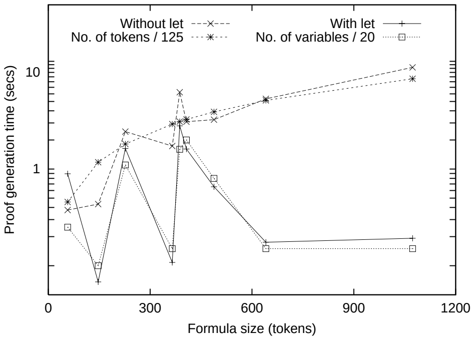


Fig. 6. Proof generation time

$$\frac{}{\text{nFree}\ S\ (\llbracket\rrbracket\ F)}\,\text{nFree}_{\text{let}_0} \qquad \frac{I \notin S \quad \forall x.\ \text{nFree}\ S\ (\llbracket J_1^{y_1} \ldots J_n^{y_n}\rrbracket\ F)}{\text{nFree}\ S\ (\llbracket I^x, J_1^{x_1} \ldots J_n^{x_n}\rrbracket\ F)}\,\text{nFree}_{\text{let}_1}$$

Fig. 7. Rules for nondependence

5 A Refinement of the Method

Substitution using let can be further improved by making use of the additional structure that our domain of application (program analysis) has.

We observe that, although the programs we wish to consider are quite large, they can be usually split into *blocks*. Assignments in a specific block B will read from and write to a subset Δ_B of the entire set of program variables Γ. The substitution techniques presented so far do not make use of this information, and require the traversal of the entire let-sequence, which will have length $|\Gamma|$.

Let us define a "nondependence" predicate nFree as

$$\text{nFree}\ S\ F\ \equiv\ \forall i.\ \forall v.\ i \in S \Rightarrow \text{subst}\ i\ v\ F\ F$$

Intuitively, this says that the variables in S are not free in F; that is, for all $I \in S$, any substitution for σI does not affect F. A set of rules to derive "nFree" is given in Figure 7.

A refinement of our method requires that, for each block B

- we move the variables Δ_B to the beginning of the let-sequence (using Permutation):

$$\llbracket\Gamma\rrbracket\ \hat{F} = \llbracket\Delta_B, \Gamma_B\rrbracket\ \hat{F};$$

- we construct a proof ϕ_B of nFree Δ_B ($\llbracket\Gamma_B\rrbracket\ \hat{F}$).

Although these two operations might be expensive (both in terms of time and proof size), they might offer a big payoff, since now each substitution of a variable σI inside the block B can be done by

(a) traversing the initial Δ_B-segment of the let-sequence;

(b) testing for membership of I to Δ_B and using ϕ_B to conclude that the remaining Γ_B-sequence is not changed by the substitution.

For (a), we know that the number of steps required is proportional to the length of the subsequence, in this case $|\Delta_B|$. For (b), we can use an encoding of sets that minimizes the time of a test for membership; assuming the set Δ is represented as a balanced binary tree, membership can be established in at most $\log_2 |\Delta_B|$ steps.

An implementation of this refinement of the original method is currently under development.

6 Conclusions

Encoding Hoare Logic in a logical framework has been a very instructive experience, in at least two distinct aspects.

First, it opened us to the idea of giving semantical definitions for syntactical concepts like substitution. The same principle can be applied to other notions: for example, we can define the set of free variables of a formula as

$$\text{freevars } S\ F \ \equiv\ \forall\sigma.\ \forall\sigma'.\ (\forall i.\ i \in S \Rightarrow \sigma i = \sigma' i) \Rightarrow (F\ \sigma) = (F\ \sigma')$$

Secondly, it has shown to us that the myth that the feature set of a framework automatically forces most implementation choices is not universally true. Our experiments with substitution have convinced us that often a bit of ingenuity goes a long way in broadening the selection of possible choices. Thus, we more likely to find an implementation that is optimal with respect to our goals, which in this case were execution time and proof size efficiency.

References

[1] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, January 2000.

[2] A. W. Appel and N. G. Michael. Machine instruction syntax and semantics in higher order logic. In *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*. Springer-Verlag, June 2000.

[3] R. Burstall and F. Honsell. Operational semantics in a natural deduction setting. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, 1992.

[4] M. J. C. Gordon. *Introduction to HOL: A Theorem Proving Environment.* Cambridge University Press, 1993.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[6] Ian A. Mason. Hoare's logic in the LF. Technical Report ECS-LFCS-87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1987.

[7] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[8] L. C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[9] F. Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers, 1999.

[10] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Berlin, July 1999.

Appendix

Our calculus is a Twelf encoding of the system presented by Appel and Felty [1]. The core symbols and rules are introduced as follows:

```
% Types
tp   : type.

form : tp.
arrow: tp -> tp -> tp.                           %infix right 14 arrow.
allt : (tp -> tp) -> tp.

% Formulas
tm    : tp -> type.

lamt  : ({T:tp} tm (F T)) -> tm (allt F).
@@    : tm (allt F) -> {T:tp} tm (F T).          %infix left 20 @@.
lam   : (tm T1 -> tm T2) -> tm (T1 arrow T2).
@     : tm (T1 arrow T2) -> tm T1 -> tm T2.      %infix left 20 @.
imp   : tm form -> tm form -> tm form.           %infix right 10 imp.
forall: (tm T -> tm form) -> tm form.

% Rules
pf     : tm form -> type.

betat_e : {F}{P} pf (P (lamt F @@ T)) -> pf (P (F T)).
betat_i : {F}{P} pf (P (F T)) -> pf (P (lamt F @@ T)).
```

```
beta_e  : {P} pf (P (lam F @ X)) ->  pf (P (F X)).
beta_i  : {P} pf (P (F X)) -> pf (P (lam F @ X)).
imp_i   : (pf A -> pf B) -> pf (A imp B).
imp_e   : pf (A imp B) -> pf A -> pf B.
forall_i: ({X} pf (A X)) -> pf (forall A).
forall_e: pf(forall A) -> {X} pf (A X).
```

Other useful logical symbols can be defined in terms of the above:

```
eq     : tm T -> tm T -> tm form =
         [A][B] forall [P] P @ B imp P @ A.
and    : tm form -> tm form -> tm form =
         [A][B] forall [C] (A imp B imp C) imp C.
or     : tm form -> tm form -> tm form =
         [A][B] forall [C] (A imp C) imp (B imp C) imp C.
exists : (tm T -> tm form) -> tm form =
         [F] forall [B] (forall [X] F X imp B) imp B.
false: tm form = forall [A] A.
not  : tm form -> tm form = [A] A imp false.
```