Part III

Tour of Interesting Aspects

CHAPTER 10

Partial Correctness

"Finally, brethren, whatever things are true, whatever things are notable, whatever things are just, whatever things are pure, whatever things are lovely, whatever things are of good report; if there is any virtue and if there is anything praiseworthy—meditate on these things."

— Philippians 4:8

In this chapter we present various interesting aspects of the VCG system, which support the proof of partial correctness of commands and the environment.

10.1 Variants

A variable is represented by a new concrete type `var`, with one constructor function, VAR: `string` $\rightarrow$ `num` $\rightarrow$ `var`. We define two deconstructor functions:

$$\begin{aligned} Base(VAR\ str\ n) &= str \\ Index(VAR\ str\ n) &= n \end{aligned}$$

The number attribute eases the creation of variants of a variable, which are made by (possibly) increasing the number.

All possible variables are considered predeclared of type `num`. In future versions, we hope to treat other data types, by introducing a more complex state and a static semantics for the language which performs type-checking. We distinguish between program variables and logical variables; the latter cannot be changed by program control. In the Sunrise language, we denote logical variables by beginning its name with a caret character ('^'), as part of its string. A "well-formed" variable, such as used in normal program code, will not have this prefix.

The *variant* function has type `var` $\rightarrow$ (`var`)`set` $\rightarrow$ `var`. *variant* x s returns a variable which is a variant of x, which is guaranteed not to be in the "exclusion" set s. If x is not in the set s, then it is its own variant. *variant* is used in defining proper substitution on quantified expressions.

The definition of *variant* is somewhat deeper than might originally appear. To have a constructive function for making variants in particular instances, we wanted

$$variant\ x\ s = (x \in s \Rightarrow variant\ (mk\_variant\ x\ 1)\ s \mid x), \tag{10.1}$$

where

$$mk\_variant\ (VAR\ str\ n)\ k = VAR\ str\ (n + k).$$

For any finite set s, this definition of *variant* will terminate, but unfortunately, it is not primitive recursive on the set s, and so does not conform to the requirements of HOLs recursive function definition. As a substitute, we wanted to define the

variant function by specifying its properties, as

$$(variant\ x\ s)\ is\_variant\ x, \text{ and} \tag{10.2}$$

$$variant\ x\ s \notin s, \text{ and} \tag{10.3}$$

$$\forall z.\ z\ is\_variant\ x \wedge z \notin s\ \Rightarrow\ Index(variant\ x\ s) \leq Index(z), \tag{10.4}$$

where *is_variant* is an infix binary predicate, defined as

$$y\ is\_variant\ x = (Base(y) = Base(x)\ \wedge\ Index(x) \leq Index(y)).$$

But even the above specification did not easily support the proof of the existence theorem, that such a variant existed for any x and s, because the set of values for z satisfying the antecedent of property 10.4 is infinite, and we were working strictly with finite sets. The solution was to introduce the function *variant_set* of type `var` $\rightarrow$ `num` $\rightarrow$ (`var`)`set`, where $variant\_set\ x\ n$ returns the set of the first n variants of x, all different from each other. Then the cardinality of the set is n, i.e.,

$$CARD\ (variant\_set\ x\ n) = n.$$

The definition of *variant_set* is

$$\begin{aligned} variant\_set\ x\ 0 &= EMPTY \\ variant\_set\ x\ (n+1) &= (mk\_variant\ x\ n)\ INSERT\ (variant\_set\ x\ n), \end{aligned}$$

where $EMPTY$ is the empty set and $INSERT$ is the infix binary operator to add an element to a set, predefined in HOL.

Then by the pigeonhole principle, we are guaranteed tha there must be at least one variable in $variant\_set\ x\ (CARD\ s + 1)$ which is not in the set s. This

led to the needed existence theorem. We then defined *variant* with the following properties:

$$(variant\ x\ s) \in variant\_set\ x\ (CARD\ s + 1),\ \text{and} \tag{10.5}$$

$$variant\ x\ s \notin s,\ \text{and} \tag{10.6}$$

$$\forall z.\ z \in variant\_set\ x\ (CARD\ s + 1) \wedge z \notin s\ \Rightarrow\ Index(variant\ x\ s) \leq Index(z). \tag{10.7}$$

From this definition, we then proved both the original set of properties (10.2–10.4), and also the constructive function definition 10.1, as theorems.

Finally, given the definition of *variant*, we defined a similar operator on lists:

$$\begin{aligned} variants\ [\,]\ s &= [\,] \\ variants\ (CONS\ x\ xs)\ s &= \textbf{let}\ x' = variant\ x\ s\ \textbf{in} \\ &\quad CONS\ x'\ (variants\ xs\ (x'\ INSERT\ s)). \end{aligned}$$

This definition has the property that the resulting list has no duplicates. We say it is a "distinct list", according to the predicate DL, which is defined as follows.

$$\begin{aligned} DL\ [\,] &= \mathrm{T} \\ DL\ (CONS\ x\ xs) &= x \notin (SL\ xs)\ \wedge\ DL\ xs \end{aligned}$$

Here SL is simply an operator to convert a list into a set, defined as follows.

$$\begin{aligned} SL\ [\,] &= EMPTY \\ SL\ (CONS\ x\ xs) &= x\ INSERT\ (SL\ xs) \end{aligned}$$

10.2 Substitution

The concept of substitution at first appears very simple, but it actually can be a mine field of subtlety and misdirection. This subtlety arises primarily from the

need to avoid the capture of free variables by bindings imposed by quantifiers in the expression receiving the substitution. Typically this is accomplished by the systematic renaming of the bound variables to preclude capturing the free variables of the expression being inserted. We have found an error in one published proof of the Substitution Lemma, and other researchers have shared their experience with the surprising difficulty of this area. The most thorough treatment we have found is by de Bakker in [dB80].

10.2.1 Assertion Language Expression Substitution

We define proper substitution on assertion language expressions using the technique of *simultaneous substitutions*, following Stoughton [Sto88]. The usual definition of proper substitution is a fully recursive function. Unfortunately, HOL only supports primitive recursive definitions. To overcome this, we use simultaneous substitutions, which are represented by functions of type `subst` = `var` $\rightarrow$ `aexp`. This describes a family of substitutions, all of which are considered to take place simultaneously. This family is in principle infinite, but in practice all but a finite number of the substitutions are the identity substitution ι. The virtue of this approach is that the application of a simultaneous substitution to an assertion language expression may be defined using only primitive recursion, not full recursion, and then the normal single substitution operation of $[v/x]$ may be defined as a special case:

$$[v/x] = \lambda y.\ (y = x \Rightarrow v \mid AVAR\ y).$$

We apply a substitution by the infix operator $\lhd$. Thus, $a \lhd ss$ denotes the application of the simultaneous substitution ss to the expression a. Therefore

$a \lhd [v/x]$ denotes the single substitution of the expression v for the variable x wherever x appears free in a.

While defining substitution on several different kinds of language phrases, we will add a subscript indicating the kind of phrase on which the substitution is being performed. For example, we will define $\lhd_v$ for substitutions on `vexp`, $\lhd_{vs}$ for substitutions on `(vexp)list`, and $\lhd_a$ for substitutions on `aexp`. However, outside this chapter we will usually simply use an undecorated $\lhd$ operator, relying on the reader to understand by context which particular substitution operator is intended. The definition of simultaneous substitution for assertion language expressions appears in Tables 10.1, 10.2, and 10.3.

$$
\begin{array}{lcl}
n \lhd_v ss & = & n \\
x \lhd_v ss & = & ss\ x \\
(v_1 + v_2) \lhd_v ss & = & (v_1 \lhd_v ss) + (v_2 \lhd_v ss) \\
(v_1 - v_2) \lhd_v ss & = & (v_1 \lhd_v ss) - (v_2 \lhd_v ss) \\
(v_1 * v_2) \lhd_v ss & = & (v_1 \lhd_v ss) * (v_2 \lhd_v ss)
\end{array}
$$

Table 10.1: Assertion Numeric Expression Simultaneous Substitution.

$$
\begin{array}{lcl}
\langle\,\rangle \lhd_{vs} ss & = & \langle\,\rangle \\
(CONS\ v\ vs) \lhd_{vs} ss & = & CONS\ (v \lhd_v ss)\ (vs \lhd_{vs} ss)
\end{array}
$$

Table 10.2: Assertion Numeric Expression List Simultaneous Substitution.

Finally, there is a dual notion of applying a simultaneous substitution to a state, instead of to an expression; this is called *semantic substitution,* and is defined as

$$s \lhd_s ss = \lambda y.\ (V\ (ss\ y)\ s).$$

$$
\begin{array}{lcl}
\textbf{true} \lhd_a ss & = & \mathrm{T} \\
\textbf{false} \lhd_a ss & = & \mathrm{F} \\
(v_1 = v_2) \lhd_a ss & = & (v_1 \lhd_v ss) = (v_2 \lhd_v ss) \\
(v_1 < v_2) \lhd_a ss & = & (v_1 \lhd_v ss) < (v_2 \lhd_v ss) \\
(vs_1 \ll vs_2) \lhd_a ss & = & (vs_1 \lhd_{vs} ss) \ll (vs_2 \lhd_{vs} ss) \\
(a_1 \wedge a_2) \lhd_a ss & = & (a_1 \lhd_a ss) \wedge (a_2 \lhd_a ss) \\
(a_1 \vee a_2) \lhd_a ss & = & (a_1 \lhd_a ss) \vee (a_2 \lhd_a ss) \\
(\sim a) \lhd_a ss & = & \sim (a \lhd_a ss) \\
(a_1 \Rightarrow a_2) \lhd_a ss & = & (a_1 \lhd_a ss) \Rightarrow (a_2 \lhd_a ss) \\
(a_1 = a_2) \lhd_a ss & = & (a_1 \lhd_a ss) = (a_2 \lhd_a ss) \\
(a_1 => a_2 \mid a_3) \lhd_a ss & = & (a_1 \lhd_a ss) => (a_2 \lhd_a ss) \mid (a_3 \lhd_a ss) \\
(\textbf{close}\ a) \lhd_a ss & = & \textbf{close}\ a \\
 & & \\
(\forall x.\ a) \lhd_a ss & = & \textbf{let}\ free = \bigcup_{z\ \in\ (FV_a\ a) - \{x\}} FV_v\ (ss\ z)\ \textbf{in} \\
 & & \textbf{let}\ y = variant\ x\ free\ \textbf{in} \\
 & & \forall y.\ a \lhd_a (ss[(AVAR\ y)/x]) \\
 & & \\
(\exists x.\ a) \lhd_a ss & = & \textbf{let}\ free = \bigcup_{z\ \in\ (FV_a\ a) - \{x\}} FV_v\ (ss\ z)\ \textbf{in} \\
 & & \textbf{let}\ y = variant\ x\ free\ \textbf{in} \\
 & & \exists y.\ a \lhd_a (ss[(AVAR\ y)/x])
\end{array}
$$

Table 10.3: Assertion Boolean Expression Simultaneous Substitution.

Most of the cases of the definition of the application of a substitution to an expression are simply the distribution of the substitution over the immediate subexpressions. For example, the application of a substitution to a conjunction is

$$(a_1 \wedge a_2) \lhd_a ss \;=\; (a_1 \lhd_a ss) \wedge (a_2 \lhd_a ss)$$

The interesting cases of the definition of $a \lhd_a ss$ are where a is a quantified expression, e.g.:

$$\begin{array}{lll} (\forall x.\, a) \lhd_a ss & = & \textbf{let } free = \bigcup_{z \,\in\, (FV_a\ a)-\{x\}} FV_v\ (ss\ z)\ \textbf{in} \\ & & \textbf{let } y = variant\ x\ free\ \textbf{in} \\ & & \forall y.\, a \lhd_a (ss[(AVAR\ y)/x]) \end{array}$$

Here FV_v is a function that returns the set of free variables in a numeric assertion expression, FV_a is a function that returns the set of free variables in a boolean assertion expression, and $variant\ x\ free$ is a function that yields a new variable as a variant of x, guaranteed not to be in the set $free$.

Once we have defined substitution as a syntactic manipulation, we can then prove the three theorems in Table 10.4 about the semantics of substitution.

$\vdash \forall v\ s\ ss.$	$V\ (v \lhd_v ss)\ s$	$=$	$V\ v\ (s \lhd_s ss)$
$\vdash \forall vs\ s\ ss.$	$VS\ (vs \lhd_{vs} ss)\ s$	$=$	$VS\ vs\ (s \lhd_s ss)$
$\vdash \forall a\ s\ ss.$	$A\ (a \lhd_a ss)\ s$	$=$	$A\ a\ (s \lhd_s ss)$

Table 10.4: Assertion Language Substitution Lemmas.

This is our statement of the Substitution Lemma of logic, and essentially says that syntactic substitution is equivalent to semantic substitution.

10.2.2 Variables-for-Variables Substitution

The substitutions discussed above replaced variables by (possibly large) numeric expressions. There is a potentially simpler version of substitution, which only replaces variables by variables. We represent these substitutions by functions of type $\texttt{vsubst} = \texttt{var} \rightarrow \texttt{var}$.

The application of these substitutions to assertion expressions is defined in Tables 10.5, 10.6, and 10.7, defining decorated versions of the operator $\lhd$, like the simultaneous substitution described above, but where ss is of type `vsubst`.

$$
\begin{array}{lcl}
n \lhd_{vv} ss & = & n \\
x \lhd_{vv} ss & = & AVAR\ (ss\ x) \\
(v_1 + v_2) \lhd_{vv} ss & = & (v_1 \lhd_{vv} ss) + (v_2 \lhd_{vv} ss) \\
(v_1 - v_2) \lhd_{vv} ss & = & (v_1 \lhd_{vv} ss) - (v_2 \lhd_{vv} ss) \\
(v_1 * v_2) \lhd_{vv} ss & = & (v_1 \lhd_{vv} ss) * (v_2 \lhd_{vv} ss)
\end{array}
$$

Table 10.5: Assertion Numeric Expression Variable-for-Variable Substitution.

$$
\begin{array}{lcl}
\langle\,\rangle \lhd_{vsv} ss & = & \langle\,\rangle \\
(CONS\ v\ vs) \lhd_{vsv} ss & = & CONS\ (v \lhd_{vv} ss)\ (vs \lhd_{vsv} ss)
\end{array}
$$

Table 10.6: Assertion Numeric Expression List Variable-for-Variable Substitution.

Most of the cases are the distribution of the substitution over the immediate subexpressions, as before. The application of ss to an assertion expression which is a simple variable is different, in that applying ss as a function to the variable name x will yield another variable, which then must be converted into an assertion expression using $AVAR$.

$\mathbf{true} \lhd_{av} ss$	$=$	T
$\mathbf{false} \lhd_{av} ss$	$=$	F
$(v_1 = v_2) \lhd_{av} ss$	$=$	$(v_1 \lhd_{vv} ss) = (v_2 \lhd_{vv} ss)$
$(v_1 < v_2) \lhd_{av} ss$	$=$	$(v_1 \lhd_{vv} ss) < (v_2 \lhd_{vv} ss)$
$(vs_1 \ll vs_2) \lhd_{av} ss$	$=$	$(vs_1 \lhd_{vsv} ss) \ll (vs_2 \lhd_{vsv} ss)$
$(a_1 \wedge a_2) \lhd_{av} ss$	$=$	$(a_1 \lhd_{av} ss) \wedge (a_2 \lhd_{av} ss)$
$(a_1 \vee a_2) \lhd_{av} ss$	$=$	$(a_1 \lhd_{av} ss) \vee (a_2 \lhd_{av} ss)$
$(\sim a) \lhd_{av} ss$	$=$	$\sim(a \lhd_{av} ss)$
$(a_1 \Rightarrow a_2) \lhd_{av} ss$	$=$	$(a_1 \lhd_{av} ss) \Rightarrow (a_2 \lhd_{av} ss)$
$(a_1 = a_2) \lhd_{av} ss$	$=$	$(a_1 \lhd_{av} ss) = (a_2 \lhd_{av} ss)$
$(a_1 => a_2 \mid a_3) \lhd_{av} ss$	$=$	$(a_1 \lhd_{av} ss) => (a_2 \lhd_{av} ss) \mid (a_3 \lhd_{av} ss)$
$(\mathbf{close}\ a) \lhd_{av} ss$	$=$	$\mathbf{close}\ a$
$(\forall x.\ a) \lhd_{av} ss$	$=$	$\forall(ss\ x).\ (a \lhd_{av} ss)$
$(\exists x.\ a) \lhd_{av} ss$	$=$	$\exists(ss\ x).\ (a \lhd_{av} ss)$

Table 10.7: Assertion Boolean Expression Variable-for-Variable Substitution.

More markedly, the cases for the substitution on quantified expressions has greatly simplified, for example

$$(\forall x.\ a) \lhd_{av} ss \ = \ \forall(ss\ x).\ (a \lhd_{av} ss).$$

There is no need here for the avoidance of capture and the selection of new variables, as the bound variable itself is also substituted, which was impossible before.

The most common variable substitutions we will use will replace the variables in one list by those of another list of equal length. We will use ι_v to denote the identity function between variables, ι_v: `var` $\rightarrow$ `var`. Then we define the operator $//_v$ to construct these variables-for-variables substitutions in Table 10.8. In the rest of this document, we will simply use a single slash to indicate this substitution-creating operator, as in $[ys/xs]$, relying on the reader to realize from

the context that since ys and xs are lists of variables, that we are referring to the variables-for-variables substitution creation operator.

$$
\begin{array}{rcl}
[\,[\,]\,//_v\,xs\,] & = & \iota_v \\
[\,ys\,//_v\,[\,]\,] & = & \iota_v \\
[\,CONS\ y\ ys\,//_v\,CONS\ x\ xs\,] & = & \textbf{let}\ ss = [ys\,//_v\,xs]\ \textbf{in} \\
 & & ss\ [(ss\ x)/(@z.\ (ss\ z) = y)]\ [y/x]
\end{array}
$$

Table 10.8: Variables-for-Variables Substitution Creation operator $//_v$.

In this definition of $//_v$, the `vsubst` ss, which is a mapping from variables to variables, is updated, first binding $(@z.\ (ss\ z) = y)$ to $ss\ x$, and then x to y. @ here is the Hilbert selection operator, choosing and yielding some variable z such that $ss\ z = y$. The reason for this double binding, rather than simply binding x to y, is to preserve the one-to-one property of the mapping; for every variable, there is exactly one variable that maps to it. This makes each such substitution one-to-one, onto, and invertible.

Once we have defined the application of variable-for-variable substitutions as a syntactic manipulation, we can then prove the theorems in Table 10.9 about the semantics of substitution.

These are only some of the shortest and simplest of the theorems proven about this kind of substitution. The ones shown describe the relationship between this kind of substitution, and the previous, where the previous kind is used to apply substitutions of the form $AVAR \circ ss$, which have type `var` $\rightarrow$ `vexp`. There are also three theorems about composing these variable-to-variable substitutions.

$$\vdash \forall v\ ss.\ \ v \lhd_{vv} ss\ =\ v \lhd_{v} (AVAR \circ ss)$$

$$\vdash \forall vs\ ss.\ \ vs \lhd_{vsv} ss\ =\ vs \lhd_{vs} (AVAR \circ ss)$$

$$\vdash \forall ss\ s.\ \ s \lhd_{s} (AVAR \circ ss)\ =\ s \circ ss$$

$$\vdash \forall v\ ss\ s.\ \ V\ (v \lhd_{vv} ss)\ s\ =\ V\ v\ (s \circ ss)$$

$$\vdash \forall vs\ ss\ s.\ \ VS\ (vs \lhd_{vsv} ss)\ s\ =\ VS\ vs\ (s \circ ss)$$

$$\vdash \forall a\ ss\ s.\ \ ONE\_ONE\ ss \Rightarrow (A\ (a \lhd_{av} ss)\ s\ =\ A\ a\ (s \circ ss))$$

$$\vdash \forall a\ ys\ xs\ s.\ \ A\ (a \lhd_{av} [ys/xs])\ s\ =\ A\ a\ (s \circ [ys/xs])$$

$$\vdash \forall a\ ys\ xs\ s.\ \ A\ (a \lhd_{av} [ys/xs])\ s\ =\ A\ (a \lhd_{a} (AVAR \circ [ys/xs]))\ s$$

$$\vdash \forall v\ ss_1\ ss_2.\ \ v \lhd_{vv} (ss_2 \circ ss_1)\ =\ (v \lhd_{vv} ss_1) \lhd_{vv} ss_2$$

$$\vdash \forall vs\ ss_1\ ss_2.\ \ vs \lhd_{vsv} (ss_2 \circ ss_1)\ =\ (vs \lhd_{vsv} ss_1) \lhd_{vsv} ss_2$$

$$\vdash \forall a\ ss_1\ ss_2.\ \ a \lhd_{av} (ss_2 \circ ss_1)\ =\ (a \lhd_{av} ss_1) \lhd_{av} ss_2$$

Table 10.9: Assertion Language Var-for-Var Substitution Lemmas.

10.2.3 Programming Language Substitution

If we wish to perform substitutions on *programming language* phrases, instead of assertion language phrases, we run into the difficulty that since expressions can have side effects, it is no longer immaterial how often an expression is evaluated. Hence it is not feasible to consider substitutions where expressions are substituted for variables. However, it turns out that the places where substitutions need to be performed on programming language phrases only require the substitution of variables for variables. Hence we only need to define one set of substitution operators for the Sunrise programming language.

In the following Tables 10.10 through 10.15, we define substitution on lists of variables, numeric expressions, lists of numeric expressions, boolean expressions, commands, and even on progress environments (i.e., *calls*).

$$
\begin{array}{lcl}
\langle\,\rangle \lhd_{xs} ss & = & \langle\,\rangle \\
(CONS\ x\ xs) \lhd_{xs} ss & = & CONS\ (ss\ x)\ (xs \lhd_{xs} ss)
\end{array}
$$

Table 10.10: Program Variable List Substitution.

$$
\begin{array}{lcl}
n \lhd_e ss & = & n \\
x \lhd_e ss & = & PVAR\ (ss\ x) \\
(++x) \lhd_e ss & = & ++(ss\ x) \\
(e_1 + e_2) \lhd_e ss & = & (e_1 \lhd_e ss) + (e_2 \lhd_e ss) \\
(e_1 - e_2) \lhd_e ss & = & (e_1 \lhd_e ss) - (e_2 \lhd_e ss) \\
(e_1 * e_2) \lhd_e ss & = & (e_1 \lhd_e ss) * (e_2 \lhd_e ss)
\end{array}
$$

Table 10.11: Program Numeric Expression Substitution.

$$
\begin{array}{lcl}
\langle\,\rangle \lhd_{es} ss & = & \langle\,\rangle \\
(CONS\ e\ es) \lhd_{es} ss & = & CONS\ (e \lhd_{e} ss)\ (es \lhd_{es} ss)
\end{array}
$$

Table 10.12: Program Numeric Expression List Substitution.

$$
\begin{array}{lcl}
(e_1 = e_2) \lhd_b ss & = & (e_1 \lhd_e ss) = (e_2 \lhd_e ss) \\
(e_1 < e_2) \lhd_b ss & = & (e_1 \lhd_e ss) < (e_2 \lhd_e ss) \\
(es_1 \ll es_2) \lhd_b ss & = & (es_1 \lhd_{es} ss) \ll (es_2 \lhd_{es} ss) \\
(b_1 \wedge b_2) \lhd_b ss & = & (b_1 \lhd_b ss) \wedge (b_2 \lhd_b ss) \\
(b_1 \vee b_2) \lhd_b ss & = & (b_1 \lhd_b ss) \vee (b_2 \lhd_b ss) \\
(\sim b) \lhd_b ss & = & \sim (b \lhd_b ss)
\end{array}
$$

Table 10.13: Program Boolean Expression Substitution.

$$
\begin{array}{lcl}
\textbf{skip} \lhd_c ss & = & \textbf{skip} \\
\textbf{abort} \lhd_c ss & = & \textbf{abort} \\
(x := e) \lhd_c ss & = & (ss\ x) := (e \lhd_e ss) \\
(c_1\ ;\ c_2) \lhd_c ss & = & (c_1 \lhd_c ss)\ ;\ (c_2 \lhd_c ss) \\
\left(\begin{array}{l} \textbf{if}\ b\ \textbf{then}\ c_1 \\ \quad \textbf{else}\ c_2\ \textbf{fi} \end{array}\right) \lhd_c ss & = & \begin{array}{l} \textbf{if}\ (b \lhd_b ss)\ \textbf{then}\ (c_1 \lhd_c ss) \\ \qquad\qquad \textbf{else}\ (c_2 \lhd_c ss)\ \textbf{fi} \end{array} \\
\left(\begin{array}{l} \textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od} \end{array}\right) \lhd_c ss & = & \begin{array}{l} \textbf{assert}\ (a \lhd_{av} ss)\ \textbf{with}\ (a_{pr} \lhd_{av} ss) \\ \textbf{while}\ (b \lhd_b ss)\ \textbf{do}\ (c \lhd_c ss)\ \textbf{od} \end{array} \\
(\textbf{call}\ p\ (xs; es)) \lhd_c ss & = & \textbf{call}\ p\ ((xs \lhd_{xs} ss)\ ;\ (es \lhd_{es} ss))
\end{array}
$$

Table 10.14: Program Command Substitution.

$$g \lhd_g ss \;=\; (\lambda p.\ (g\ p) \lhd_{av} ss)$$

Table 10.15: Program Progress Environment Substitution.

Table 10.16 has programming language versions of the Substitution Lemma.

$$\vdash \forall e\ s_1\ n\ s_2\ ss.\ \ ONE\_ONE\ ss \wedge ONTO\ ss \Rightarrow$$
$$(E\ (e \lhd_e ss)\ s_1\ n\ s_2 \;=\; E\ e\ (s_1 \circ ss)\ n\ (s_2 \circ ss))$$

$$\vdash \forall e\ s_1\ n\ s_2\ ys\ xs.$$
$$E\ (e \lhd_e [ys/xs])\ s_1\ n\ s_2 \;=\; E\ e\ (s_1 \circ [ys/xs])\ n\ (s_2 \circ [ys/xs])$$

$$\vdash \forall es\ s_1\ ns\ s_2\ ss.\ \ ONE\_ONE\ ss \wedge ONTO\ ss \Rightarrow$$
$$(ES\ (es \lhd_{es} ss)\ s_1\ ns\ s_2 \;=\; ES\ es\ (s_1 \circ ss)\ ns\ (s_2 \circ ss))$$

$$\vdash \forall es\ s_1\ ns\ s_2\ ys\ xs.$$
$$ES\ (es \lhd_{es} [ys/xs])\ s_1\ ns\ s_2 \;=\; ES\ es\ (s_1 \circ [ys/xs])\ ns\ (s_2 \circ [ys/xs])$$

$$\vdash \forall b\ s_1\ t\ s_2\ ss.\ \ ONE\_ONE\ ss \wedge ONTO\ ss \Rightarrow$$
$$(B\ (b \lhd_b ss)\ s_1\ t\ s_2 \;=\; B\ b\ (s_1 \circ ss)\ t\ (s_2 \circ ss))$$

$$\vdash \forall b\ s_1\ t\ s_2\ ys\ xs.$$
$$B\ (b \lhd_b [ys/xs])\ s_1\ t\ s_2 \;=\; B\ b\ (s_1 \circ [ys/xs])\ t\ (s_2 \circ [ys/xs])$$

$$\vdash \forall c\ g\ \rho\ ss\ s_1\ s_2.$$
$$WF_{env\_syntax}\ \rho \ \wedge\ WF_c\ c\ g\ \rho \ \wedge\ WF_{csubst}\ c\ \rho\ ss \ \Rightarrow$$
$$(C\ (c \lhd_c ss)\ \rho\ s_1\ s_2 \;=\; C\ c\ \rho\ (s_1 \circ ss)\ (s_2 \circ ss))$$

$$\vdash \forall c\ g\ \rho\ ys\ xs\ s_1\ s_2.$$
$$WF_{env\_syntax}\ \rho \ \wedge\ WF_c\ c\ g\ \rho \ \wedge\ WF_{csubst}\ c\ \rho\ [ys/xs] \ \Rightarrow$$
$$(C\ (c \lhd_c [ys/xs])\ \rho\ s_1\ s_2 \;=\; C\ c\ \rho\ (s_1 \circ [ys/xs])\ (s_2 \circ [ys/xs]))$$

Table 10.16: Programming Language Substitution Lemmas.

Finally, we exhibit some theorems that declare that if the free variables of an expression are mapped to the same results by two different variable-for-variable substitutions, then the result of applying the two substitutions to the expression must be the same. Thus the results of substitution depend only on the substitution's effect on the expression's free variables.

$$\vdash \forall e\ ss_1\ ss_2.\ (\forall x.\ x \in FV_e\ e \Rightarrow (ss_1\ x = ss_2\ x)) \Rightarrow (e \lhd_e ss_1\ =\ e \lhd_e ss_2)$$

$$\vdash \forall es\ ss_1\ ss_2.\ (\forall x.\ x \in FV_{es}\ es \Rightarrow (ss_1\ x = ss_2\ x)) \Rightarrow (es \lhd_{es} ss_1\ =\ es \lhd_{es} ss_2)$$

$$\vdash \forall b\ ss_1\ ss_2.\ (\forall x.\ x \in FV_b\ b \Rightarrow (ss_1\ x = ss_2\ x)) \Rightarrow (b \lhd_b ss_1\ =\ b \lhd_b ss_2)$$

Table 10.17: Programming Language Substitution Equality Theorems.

10.3 Translation

Expressions have typically not been trated in previous work on verification; there are some exceptions, notably Sokołowski [Sok84]. Expressions with side effects have been particularly excluded. Since expressions did not have side effects they were often considered to be a sublanguage, common to both the programming language and the assertion language. Thus one would see expressions such as $p \wedge b$, where p was an assertion and b was a boolean expression from the programming language.

One of the key realizations of this work was the need to carefully distinguish these two languages, and not confuse their expression sublanguages. This then requires us to *translate* programming language expressions into equivalent expressions in the assertion language before the two may be combined as above. In fact, since we allow expressions to have side effects, there are actually two results of translating a programming language expression e:

- an assertion language expression, representing the value of e in the state "before" evaluation, *and*
- a simultaneous substitution, representing the change in state from "before" evaluating e to "after" evaluating e.

For example, the translator for numeric expressions is defined using a helper function $VE1$: $\texttt{exp} \rightarrow \texttt{subst} \rightarrow (\texttt{aexp} \times \texttt{subst})$:

$$\begin{aligned} VE1\ (n)\ ss &= n,\ ss \\ VE1\ (x)\ ss &= ss\ x,\ ss \end{aligned}$$

$$
\begin{array}{rcl}
VE1\ (++x)\ ss & = & (ss\ x) + 1,\ ss[((ss\ x) + 1)/x] \\
VE1\ (e_1 + e_2)\ ss & = & (VE1\ e_1 \rightarrow \lambda v_1. \\
 & & \quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 + v_2,\ ss_2)))\ ss \\
VE1\ (e_1 - e_2)\ ss & = & (VE1\ e_1 \rightarrow \lambda v_1. \\
 & & \quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 - v_2,\ ss_2)))\ ss \\
VE1\ (e_1 * e_2)\ ss & = & (VE1\ e_1 \rightarrow \lambda v_1. \\
 & & \quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 * v_2,\ ss_2)))\ ss
\end{array}
$$

where $\rightarrow$ is a "translator continuation" operator, defined as

$$(f \rightarrow k)\ ss = \textbf{let}\ (v, ss') = f\ ss\ \textbf{in}\ k\ v\ ss'$$

Then define

$$
\begin{array}{rcl}
VE\ e & = & FST\ (VE1\ e\ \iota) \\
VE\_state\ e & = & SND\ (VE1\ e\ \iota)
\end{array}
$$

where ι is the identity substitution, $\iota\ x = AVAR\ x$. These two functions deliver the two results itemized above for the translation of e.

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem.

$$
\begin{array}{rcl}
\vdash \forall e\ s_1\ n\ s_2.\ (E\ e\ s_1\ n\ s_2) & = & (n = V\ (VE\ e)\ s_1\ \wedge \\
 & & \quad s_2 = s_1 \lhd (VE\_state\ e))
\end{array}
$$

In a similar fashion we can translate lists of numeric expressions. The translator for lists of numeric expressions is defined using a helper function $VES1$: (`exp`)`list` $\rightarrow$ `subst` $\rightarrow$ ((`aexp`)`list` $\times$ `subst`):

$$
\begin{array}{rcl}
VES1\ (\langle\,\rangle)\ ss & = & [\,],\ ss \\
VE1\ (CONS\ e\ es)\ ss & = & (VE1\ e \rightarrow \lambda v. \\
 & & \quad (VES1\ es \rightarrow \lambda vs\ ss_2.\ (CONS\ v\ vs,\ ss_2)))\ ss
\end{array}
$$

Then define

$$\begin{aligned} VES\ es &= FST\ (VES1\ es\ \iota) \\ VES\_state\ es &= SND\ (VES1\ es\ \iota) \end{aligned}$$

These two functions deliver the two results itemized above for the translation of es.

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem.

$$\begin{aligned} \vdash \forall es\ s_1\ ns\ s_2.\ (ES\ es\ s_1\ ns\ s_2) = {} & (ns = VS\ (VES\ es)\ s_1 \wedge \\ & s_2 = s_1 \lhd (VES\_state\ es)) \end{aligned}$$

In a similar fashion we can translate boolean expressions. The translator for boolean expressions is defined using a helper function

$AB1$: `bexp` $\rightarrow$ `subst` $\rightarrow$ (`aexp` $\times$ `subst`):

$$\begin{aligned} AB1\ (e_1 = e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\ &\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 = v_2,\ ss_2)))\ ss \\ AB1\ (e_1 < e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\ &\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 < v_2,\ ss_2)))\ ss \\ AB1\ (es_1 \ll es_2)\ ss &= (VES1\ es_1 \rightarrow \lambda vs_1. \\ &\quad (VES1\ es_2 \rightarrow \lambda vs_2\ ss_2.\ (vs_1 \ll vs_2,\ ss_2)))\ ss \\ AB1\ (b_1 \wedge b_2)\ ss &= (AB1\ b_1 \rightarrow \lambda t_1. \\ &\quad (AB1\ b_2 \rightarrow \lambda t_2\ ss_2.\ (t_1 \wedge t_2,\ ss_2)))\ ss \\ AB1\ (b_1 \vee b_2)\ ss &= (AB1\ b_1 \rightarrow \lambda t_1. \\ &\quad (AB1\ b_2 \rightarrow \lambda t_2\ ss_2.\ (t_1 \vee t_2,\ ss_2)))\ ss \\ AB1\ (\sim b)\ ss &= (AB1\ b \rightarrow \lambda t\ ss_2.\ (\sim t,\ ss_2))\ ss \end{aligned}$$

Then define

$$\begin{aligned} AB\ b &= FST\ (AB1\ b\ \iota) \\ AB\_state\ b &= SND\ (AB1\ b\ \iota) \end{aligned}$$

These two functions deliver the two results itemized above for the translation of b.

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem.

$$\vdash \forall b\ s_1\ t\ s_2.\ (B\ b\ s_1\ t\ s_2) \ = \ (t = A\ (AB\ b)\ s_1 \wedge \\ s_2 = s_1 \lhd (AB\_state\ b))$$

This theorem, along with the corresponding ones for numeric expressions and lists of numeric expressions, mean that every evaluation of a programming language expresssion has its semantics completely captured by the two translation functions for its type. These are essentially small compiler correctness proofs.

Using these translation functions, we may define functions to compute the appropriate preconditions to an executable expression, given the postcondition, as given in Table 10.18.

`vexp`	$ve\_pre\ e\ v = v \lhd_v (VE\_state\ e)$ $ves\_pre\ es\ v = v \lhd_v (VES\_state\ es)$ $vb\_pre\ b\ v = v \lhd_v (AB\_state\ b)$
`(vexp)list`	$vse\_pre\ e\ vs = vs \lhd_{vs} (VE\_state\ e)$ $vses\_pre\ es\ vs = vs \lhd_{vs} (VES\_state\ es)$ $vsb\_pre\ b\ vs = vs \lhd_{vs} (AB\_state\ b)$
`aexp`	$ae\_pre\ e\ a = a \lhd_a (VE\_state\ e)$ $aes\_pre\ es\ a = a \lhd_a (VES\_state\ es)$ $ab\_pre\ b\ a = a \lhd_a (AB\_state\ b)$

Table 10.18: Expression Precondition Functions.

As a product, we may now define the simultaneous substitution that corresponds to an assignment statement (single or multiple,) overriding the expression's state change with the change of the expression. We define

$$[x := e] = (VE\_state\ e)[(VE\ e)/x]$$

and

$$[xs := es] = (VES\_state\ es)[(VES\ es)/xs].$$

These simultaneous substitutions are used directly in defining the VCG function. The single assignment substitution is used in processing the assignment command, to compute the appropriate precondition. The multiple assignment substitution is used in processing the actual value parameters of the procedure call command, to reflect their execution's effect on the state.

We have found these translation functions to greatly condense and simplify the handling of expressions with side effects. While not an approach that can describe all possible operators with side effects, we believe this translation function approach is flexible enough to handle input/output and user-defined functions with side effects. These questions are a part of our plans for future research.

10.4 Well-Formedness

In the creation of small languages with simple features, it may be possible to define the semantics of the language sufficiently cleanly so that every program which is syntactically valid has a well-defined and proper semantics. However, as more sophisticated features are added to the language under consideration, it becomes necessary to further restrict the set of "acceptable" programs for which

one's analysis is applicable. We have found that the feature of procedure calls introduced the need to verify several restrictions on sample programs, for example that the arity of a call matched that of the definition. We have defined predicates to express these restrictions, called *well-formedness* predicates. Unless a program meets these criteria, we do not even consider it in a proof of correctness.

These well-formedness predicates describe a number of conditions, mostly simple syntactic checks like the arity check mentioned, but also including a number of semantic checks, such as the total correctness of a procedure's body with respect to its precondition and postcondition. Generally, the syntactic checks may be decided by a single, static, compile-time examination of the program. The semantic checks are satisfied by the meta-level verification of the verification condition generator, and by the proofs of the verification conditions generated by it. Since this verification includes some of the hardest parts of the proof of well-formedness, it is fortunate that much of it can be decided at the meta level. For this version of the Sunrise language, we find it unnecessary to also include dynamic checks, to be conjoined to the preconditions computed during the VCG calculation, along with the static checks instituted for compile-time. This may change in the future, for example with the introduction of arrays the checks to prevent aliasing of parameters may require dynamic checks. But for now, the only checks necessary are static, syntactic checks, that may be performed fully automatically.

It is interesting to us that there has been very little focus in the past on this issue of well-formedness. In our work, it became crucial from the beginning of the work on procedures, because it was not possible to properly relate the operational

semantics and the axiomatic semantics without constraining the set of programs considered to ones that made sense. We hope that this work will exhibit the issues involved with their proper priority.

10.4.1 Informal Description

The checks that are part of well-formedness vary with the construct being analyzed, but for the most part are simple syntactic tests on the immediate constituent constructs, and so may be defined on the structure of the constructs.

One pervasive check is the exclusion of logical variables from normal program text. Logical variables are restricted from appearing except in assertion language expressions, as part of the definition of the Sunrise language. Yet syntactically, a logical variable and a program variable are both the same kind of phrase. We rely on well-formedness checks to ensure that only program variables appear in normal program text.

For procedure calls, other checks are needed as well. The arity checks are one example, that the number of actual variable parameters matches the number of formal variable parameters, and the same for value parameters. In addition, we must ensure that aliasing does not occur; this can be done by checking that the combination of the actual variable parameters and the declared globals of the procedure being called contains no duplicates.

When it comes to procedure definitions, there are several checks that must be satisfied. These express both syntactic and semantic considerations. The syntactic considerations include checking that every variable in the parameter lists or the globals is not logical, that they have no duplicates among them, that

the body of the procedure is well-formed, and constraints on the free variables of the precondition and postcondition.

A procedure definition is fully well-formed if it is syntactically well-formed as described above, and then satisfies one additional semantic criterion; the body must be totally correct with respect to the given precondition and postcondition.

It now becomes possible to speak of an entire environment of procedure definitons being well-formed, if every individual procedure definition in the environment is itself fully well-formed.

The requirement for total correctness is quite strong. It turns out that it is quite useful to establish "stepping stones" along the way to proving total correctness, where less powerful semantic properties are established and then combined to justify more powerful ones. We have made considerable use of this approach, and in the verification of the VCG, we prove several properties in order, as described in Table 10.19.

$WF_{env\_syntax}\ \rho$	ρ is well-formed for syntax
$WF_{envk\_partial}\ \rho\ k$	ρ is well-formed for partial correctness to stage k
$WF_{envk}\ \rho\ k$	ρ is well-formed for syntax and partial correct. to stage k
$WF_{envp}\ \rho$	ρ is well-formed for syntax and partial correctness
$WF_{env\_pre}\ \rho$	ρ is well-formed for preconditions
$WF_{env\_calls}\ \rho$	ρ is well-formed for calls progress
$WF_{env\_term}\ \rho$	ρ is well-formed for conditional termination
$WF_{env\_rec}\ \rho$	ρ is well-formed for recursion
$WF_{env\_partial}\ \rho$	ρ is well-formed for partial correctness
$WF_{env\_total}\ \rho$	ρ is well-formed for termination
$WF_{env\_correct}\ \rho$	ρ is well-formed for total correctness
$WF_{env}\ \rho$	ρ is well-formed for syntax and total correctness

Table 10.19: Procedure Environment Well-Formedness Predicates.

10.4.2 Well-Formedness Predicate Definitions

A string s is well-formed ($WF_s\ s$) if the first character is not '^'.

$$WF_s\ \text{``} = \text{T}$$
$$WF_s\ (STRING\ a\ s) = (a \neq LOG\_CHAR)$$

where $LOG\_CHAR =$ '^'.

Table 10.20: Definition of Well-Formedness for Strings.

A variable x is well-formed ($WF_x\ x$) if its string is well-formed.

$$WF_x\ (VAR\ s\ n) = WF_s\ s$$

Table 10.21: Definition of Well-Formedness for Variables.

A list of variables xs is well-formed ($WF_{xs}\ xs$) if every variable in the list is well-formed.

$$WF_{xs}\ \langle\,\rangle = \text{T}$$
$$WF_{xs}\ (CONS\ x\ xs) = WF_x\ x\ \wedge\ WF_{xs}\ xs$$

Table 10.22: Definition of Well-Formedness for Lists of Variables.

A list of variables xs is not-well-formed ($NOT\_WF_{xs}\ xs$) if every variable in the list is not well-formed.

$$NOT\_WF_{xs}\ \langle\rangle = \mathrm{T}$$
$$NOT\_WF_{xs}\ (CONS\ x\ xs) = \sim(WF_x\ x)\ \wedge\ NOT\_WF_{xs}\ xs$$

Table 10.23: Definition of Not-Well-Formedness for Lists of Variables.

A numeric expression e is well-formed ($WF_e\ e$) if every part is well-formed.

$$WF_e\ (n) = \mathrm{T}$$
$$WF_e\ (x) = WF_x\ (x)$$
$$WF_e\ (++x) = WF_x\ (x)$$
$$WF_e\ (e_1 + e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2$$
$$WF_e\ (e_1 - e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2$$
$$WF_e\ (e_1 * e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2$$

Table 10.24: Definition of Well-Formedness for Numeric Expressions.

A list of numeric expressions es is well-formed ($WF_{es}\ es$) if every expression in the list is well-formed.

$$WF_{es}\ \langle\rangle = \mathrm{T}$$
$$WF_{es}\ (CONS\ e\ es) = WF_e\ e\ \wedge\ WF_{es}\ es$$

Table 10.25: Definition of Well-Formedness for Lists of Numeric Expressions.

A boolean expression b is well-formed ($WF_b\ b$) if every part is well-formed.

$$
\begin{array}{l}
WF_b\ (e_1 = e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
WF_b\ (e_1 < e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
WF_b\ (es_1 \ll es_2) = WF_{es}\ es_1\ \wedge\ WF_{es}\ es_2 \\
WF_b\ (b_1 \wedge b_2) = WF_b\ b_1\ \wedge\ WF_b\ b_2 \\
WF_b\ (b_1 \vee b_2) = WF_b\ b_1\ \wedge\ WF_b\ b_2 \\
WF_b\ (\sim b) = WF_b\ b
\end{array}
$$

Table 10.26: Definition of Well-Formedness for Boolean Expressions.

A command c is well-formed in a progress environment g and a procedure environment ρ ($WF_c\ c\ g\ \rho$) if every part is well-formed, if every **while** command's progress expression avoids other logical variables, if every call supplies the same number of actual parameters as the procedure has formal parameters, and if there is no aliasing among the variable parameters and the globals.

$$
\begin{array}{l}
WF_c\ (\textbf{skip})\ g\ \rho = \mathrm{T} \\
WF_c\ (\textbf{abort})\ g\ \rho = \mathrm{T} \\
WF_c\ (x := e)\ g\ \rho = WF_x\ x\ \wedge\ WF_e\ e \\
WF_c\ (c_1\ ;\ c_2)\ g\ \rho = WF_c\ c_1\ g\ \rho\ \wedge\ WF_c\ c_2\ g\ \rho \\
WF_c\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ g\ \rho = WF_b\ b\ \wedge\ WF_c\ c_1\ g\ \rho\ \wedge\ WF_c\ c_2\ g\ \rho \\
WF_c\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr}\textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ g\ \rho = \\
\qquad\qquad WF_b\ b\ \wedge\ WF_c\ c\ g\ \rho\ \wedge \\
\qquad\qquad (\exists v\ x.\ a_{pr} = (v < x)\ \wedge\ \sim(WF_x\ x)\ \wedge\ x \notin (FV_a\ a \cup FV_v\ v)\ \wedge \\
\qquad\qquad\qquad (\forall p.\ x \notin FV_a\ (g\ p)) \\
WF_c\ (\textbf{call}\ p\ (xs; es))\ g\ \rho = \\
\qquad\qquad WF_{xs}\ xs\ \wedge\ WF_{es}\ es\ \wedge \\
\qquad\qquad (|vars| = |xs|)\ \wedge\ (|vals| = |es|)\ \wedge\ DL\ (xs\ \&\ glbs)
\end{array}
$$

Table 10.27: Definition of Well-Formedness for Commands.

where in the last line, $\rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle$. Here $|xs|$

denotes the length of the list xs, and DL ("distinct list") is a predicate saying the variables in xs and $glbs$ have no duplicates.

A procedure specification $\langle vars, vals, glbs, pre, post, calls, rec, c\rangle$ is syntactically well-formed in an environment ρ
($WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho$) if

let $x = vars\ \&\ vals\ \&\ glbs$ **and** $x_0 = logicals\ x$ **in**

1) $WF_{xs}\ x$
2) $DL\ x$
3) $WF_c\ c\ calls\ \rho$
4) $GV_c\ c\ \rho \subseteq glbs$
5) $FV_c\ c\ \rho \subseteq x$
6) $FV_a\ pre \subseteq x$
7) $FV_a\ post \subseteq (x \cup x_0)$
8) $(\forall s.$ **let** $\langle vars', vals', glbs', pre', post', calls', rec', c'\rangle = \rho\ s$ **in**
   let $x' = vars'\ \&\ vals'\ \&\ glbs'$ **in**
   $FV_a\ (calls\ s) \subseteq SL\ (x'\ \&\ x_0))$
9) $(rec =$ **false**$) \vee$
   $(\exists v\ y.\ rec = (v < y)\ \wedge\ \sim(WF_x\ y)\ \wedge\ FV_v\ v \subseteq SL\ x)$

Table 10.28: Definition of Well-Formedness for Procedure Specification Syntax.

The several clauses of the definition of WF_{proc_syntax} are explained as follows:

1. every variable in $vars$, $vals$, and $glbs$ is well-formed, i.e., not logical,

2. the variables in $vars$, $vals$, and $glbs$ have no duplicates,

3. c is well-formed in calls progress environment $calls$ and environment ρ,

4. all globals referenced by procedures called within c are in $glbs$,

5. all the free variables of c are in x,

6. all the free variables of pre are in x,

7. all the free variables of $post$ are in x or in x_0,

8. for each procedure p, all the free variables of the progress expression contained in $calls$ for p are in x' or in x_0, where x' is the list of the variables accessible from p,

9. either rec is **false**, or else rec has the form $v < y$, where y is a logical variable not in x.

A procedure specification $\langle vars, vals, glbs, pre, post, calls, rec, c\rangle$ is fully well-formed (both syntactically and semantically) in an environment ρ ($WF_{proc}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho$) if

let $x = vars\ \&\ vals\ \&\ glbs$ **and** $x_0 = logicals\ x$ **in**
1) $WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho$
2) $\{x_0 = x\ \wedge\ pre\}\ c\ \{post\}\ /\rho$

Table 10.29: Definition of Well-Formedness for Procedure Specification.

where

1. the specification is syntactically well-formed, and

2. c is partially correct with precondition $(x_0 = x\ \wedge\ pre)$ and postcondition $post$ in environment ρ.

An environment ρ is well-formed ($WF_{env}\ \rho$) if every procedure declaration is well-formed in ρ.

$$WF_{env}\ \rho = \forall p.\ WF_{proc}\ (\rho\ p)\ \rho$$

Table 10.30: Definition of Well-Formedness for Procedure Environment.

A progress environment *calls* is well-formed in a procedure environment ρ if for every procedure p, all the free variables of ($calls\ p$) are in the variables accessible from p.

$$\begin{aligned} WF_{calls}\ calls\ \rho = (\forall p.\ &\textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \textbf{in} \\ &\textbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \textbf{in} \\ &(\exists z.\ FV_a\ (calls\ p) \subseteq SL\ (x\ \&\ logicals\ z)) \end{aligned}$$

Table 10.31: Definition of Well-Formedness for Progress Environment.

A declaration d is well-formed in an environment ρ ($WF_d\ d\ \rho$) if every individual procedure declaration is syntactically well-formed in ρ.

$$\begin{aligned} &WF_d\ (\textbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c)\ \rho = \\ &\qquad WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho \\ &WF_d\ (d_1\ ;\ d_2)\ \rho = WF_d\ d_1\ \rho\ \wedge\ WF_d\ d_2\ \rho \\ &WF_d\ (\textbf{empty})\ \rho = \mathrm{T} \end{aligned}$$

Table 10.32: Definition of Well-Formedness for Declarations.

The empty procedure environment ρ_0 is an initial environment with each procedure having no parameters or globals, a **false** precondition and a **true** postcondition, a **false** progress condition for every procedure (none of which are called), a **false** recursion expression, and a body consisting solely of the command **abort**. Declarations present in the program override these default declarations, which should never be invoked. The **false** precondition itself implies the impossibility of proving any program that calls an undeclared procedure.

$$\rho_0 \;=\; (\lambda p.\,[\,],\;[\,],\;[\,],\;\mathbf{false},\;\mathbf{true},\;(\lambda p.\;\mathbf{false},)\;\mathbf{false},\;\mathbf{abort})$$

Table 10.33: Definition of Empty Progress Environment.

The empty progress environment g_0 is **true** for all procedures. This is the progress environment used for processing the main body, for which there are no **calls** ... **with** specifications.

$$g_0 \;=\; (\lambda p.\;\mathbf{true})$$

Table 10.34: Definition of Empty Progress Environment.

A program π is well-formed ($WF_p\ \pi$) if both its declarations and its body are well-formed in the environment the declarations create.

$$
\begin{array}{l}
WF_p\ (\textbf{program}\ d\ ;\ c\ \textbf{end program}) = \\
\quad \textbf{let}\ \rho = mkenv\ d\ \rho_0\ \textbf{in} \\
\quad WF_d\ d\ \rho\ \wedge\ WF_c\ c\ g_0\ \rho
\end{array}
$$

Table 10.35: Definition of Well-Formedness for Programs.

These well-formedness predicates were indispensable prerequisites for all the reasoning of the verification condition generator. They restricted the set of programs considered to those that were consistent and proper. Without these restrictions, no deep theorems about the semantics of the VCG would have been possible; but with them, principles can be stated and proved about the wide class of normal programs which are the actual aim.

10.5 Semantic Stages

When we began proving partial correctness from the conditions generated by the VCG, we ran into a difficulty. Two of the correctness properties we wanted to prove were `vcgcp_THM` and `vcgd_THM`, repeated from Chapter 7 in Table 10.36.

`vcgcp_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\{p\}\ c\ \{q\}\ /\rho$
`vcgd_THM`	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{envp}\ \rho$

Table 10.36: Repeated VCG verification theorems.

In order to prove `vcgd_THM`, we wished to use `vcgcp_THM`, proven earlier. `vcgd_THM` is used to prove the well-formedness of an environment for partial correctness. This has both syntactic and semantic parts. The syntactic well-formedness is supported by $WF_d\ d\ \rho$, so we need only add the proof of the semantic part. For this, we wished to reason from the truth of the verification conditions produced by $vcgd\ d\ \rho$ to the partial correctness of each procedure body declared in d, with respect to its precondition and postcondition. For this task, `vcgcp_THM` appeared to be the appropriate tool, applying it to each procedure body in turn. The problem was that `vcgcp_THM` itself requires an environment well-formed for partial correctness as a precondition! Thus it seemed to be necessary to know that the environment was well-formed before we could prove that it was well-formed, a circular argument.

The solution was to cut the circle by establishing *stages* of well-formedness for the environment, indexed by number, and to show eventually by numeric induction that all stages hold, and thus the environment is well-formed. Each increase in the index signifies an ability to call procedures to one more level of calling depth. Thus, index 0 designates an environment which is well-formed as long as no procedure calls are made; index 1 designates an environment which is well-formed under calls of procedures which do not themselves issue procedure calls, etc. In pursuing this line of reasoning, it became apparent that in order to define stages of well-formedness, we needed to establish stages of command partial correctness specifications, and of the command semantic relation itself.

The new staged version of the command semantic relation C_k is described in Table 10.37.

$C_k\ c\ \rho\ k\ s_1\ s_2$	command c:`cmd` evaluated in environment ρ and state s_1 yields state s_2, without ever issuing calls beyond a nested depth of k.

Table 10.37: Staged command semantic relation description.

The definition of the new staged command semantic relation C_k is given in Table 10.38. It is similar to the definition of C in Table 5.11, but C_k adds one new argument k, which is the stage number, and every rule maintains that the stage of the resulting tuple is greater than or equal to the stages of all antecedent tuples, *except* for the procedure call rule, where the stage of the result tuple (regarding the procedure call) is exactly one greater than that of the antecedent rule (regarding the procedure's body).

Skip:

$$\frac{}{C_k\ \textbf{skip}\ \rho\ k\ s\ s}$$

Abort:

(no rules)

Assignment:

$$\frac{E\ e\ s_1\ n\ s_2}{C_k\ (x := e)\ \rho\ k\ s_1\ s_2[n/x]}$$

Sequence:

$$\frac{\begin{array}{ll} C_k\ c_1\ \rho\ k_1\ s_1\ s_2, & k_1 \leq k \\ C_k\ c_2\ \rho\ k_2\ s_2\ s_3, & k_2 \leq k \end{array}}{C_k\ (c_1\ ;\ c_2)\ \rho\ k\ s_1\ s_3}$$

Conditional:

$$\frac{B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C_k\ c_1\ \rho\ k\ s_2\ s_3}{C_k\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ k\ s_1\ s_3}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2, \qquad C_k\ c_2\ \rho\ k\ s_2\ s_3}{C_k\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ k\ s_1\ s_3}$$

Iteration:

$$\frac{\begin{array}{l} B\ b\ s_1\ \mathrm{T}\ s_2 \\ C_k\ c\ \rho\ k_1\ s_2\ s_3, \qquad k_1 \leq k \\ C_k\ (\textbf{assert}\ a\ \textbf{with}\ pr \\ \quad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ k_2\ s_3\ s_4, \qquad k_2 \leq k \end{array}}{\begin{array}{l} C_k\ (\textbf{assert}\ a\ \textbf{with}\ pr \\ \quad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ k\ s_1\ s_4 \end{array}}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2}{\begin{array}{l} C_k\ (\textbf{assert}\ a\ \textbf{with}\ pr \\ \quad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ k\ s_1\ s_2 \end{array}}$$

Call:

$$\frac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c \rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \\ C_k\ (c \lhd [xs\ \&\ vals'/vars\ \&\ vals])\ \rho\ k\ s_2[ns/vals']\ s_3 \end{array}}{C_k\ (\textbf{call}\ p(xs;\ es))\ \rho\ (k+1)\ s_1\ s_3[(\textbf{map}\ s_2\ vals')/vals']}$$

Table 10.38: Staged Command Structural Operational Semantics.

We define the staged command partial correctness specification in Table 10.39.

$$\{a_1\}\ c\ \{a_2\}\ /\rho, k\ =\ (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge C_k\ c\ \rho\ k\ s_1\ s_2 \Rightarrow A\ a_2\ s_2)$$

Table 10.39: Staged command Partial Correctness Specification.

We define the staged version of well-formedness of environments for partial correctness in Table 10.40.

$$\begin{array}{l} WF_{prock}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho\ k\ = \\ \quad \textbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \textbf{and}\ x_0 = logicals\ x\ \textbf{in} \\ \quad (WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho \wedge \\ \quad \{x_0 = x\ \wedge\ pre\}\ c\ \{post\}\ /\rho, k) \\ \\ WF_{envk}\ \rho\ k\ =\ \forall p.\ WF_{prock}\ (\rho\ p)\ \rho\ k \end{array}$$

Table 10.40: Staged Well-Formed Environment Predicate for Partial Correctness.

Using these definitions, we can prove many staged version of previous theorems about commands, for example the substitution lemmas in Table 10.41.

$$\begin{array}{l} \vdash \forall c\ g\ \rho\ k\ ss\ s_1\ s_2. \\ \quad WF_{env\_syntax}\ \rho\ \wedge\ WF_c\ c\ g\ \rho\ \wedge\ WF_{csubst}\ c\ \rho\ ss\ \Rightarrow \\ \quad (C_k\ (c \lhd_c ss)\ \rho\ k\ s_1\ s_2\ =\ C_k\ c\ \rho\ k\ (s_1 \circ ss)\ (s_2 \circ ss)) \\ \\ \vdash \forall c\ g\ \rho\ k\ ys\ xs\ s_1\ s_2. \\ \quad WF_{env\_syntax}\ \rho\ \wedge\ WF_c\ c\ g\ \rho\ \wedge\ WF_{csubst}\ c\ \rho\ [ys/xs]\ \Rightarrow \\ \quad (C_k\ (c \lhd_c [ys/xs])\ \rho\ k\ s_1\ s_2\ =\ C_k\ c\ \rho\ k\ (s_1 \circ [ys/xs])\ (s_2 \circ [ys/xs])) \end{array}$$

Table 10.41: Staged Command Substitution Lemmas.

We also prove theorems which relate C_k, $\{a_1\}\ c\ \{a_2\}\ /\rho, k$, and $WF_{envk}\ \rho\ k$ to their unstaged original counterparts. These are given in Table 10.42.

$$\vdash \forall c\ \rho\ s_1\ s_2.\ C\ c\ \rho\ s_1\ s_2\ =\ (\exists k.\ C_k\ c\ \rho\ k\ s_1\ s_2)$$

$$\vdash \forall a_1\ c\ a_2\ \rho.\ \{a_1\}\ c\ \{a_2\}\ /\rho\ =\ (\forall k.\ \{a_1\}\ c\ \{a_2\}\ /\rho, k)$$

$$\vdash \forall \rho.\ WF_{envp}\ \rho\ =\ (\forall k.\ WF_{envk}\ \rho\ k)$$

Table 10.42: Unstaged-to-Staged Correspondances.

This last theorem gives us the means to prove that an environment is well-formed for partial correctness. We first prove that for $k = 0$, the antecedents of `vcgd_THM` imply the environment is well-formed to stage 0. This is theorem `vcgd_0_THM` of Table 7.3. To prove this, we first prove theorem `vcg1_0_THM` of Table 7.1, and then `vcgc_0_THM` of Table 7.2, from which `vcgd_0_THM` follows.

Then assumming the antecedents of `vcgd_THM` and that the environment is well-formed to stage k, we prove that it is well-formed to stage $k + 1$. This is theorem `vcgd_k_THM`, built as before by first proving `vcg1_k_THM` and then `vcgc_k_THM`. By induction, the environment is then well-formed for all stages, and by the above theorem in Table 10.42, the environment is completely well-formed for partial correctness, which proves theorem `vcgd_THM`.

By proving this induction on stage numbers here at the meta-level, we obviate the need for the programmer to have to prove verification conditions that deal with these partial correctness issues of the program's recursion, for all programs.