Program Verification Based on Denotational Semantics

Wolfgang Polak
Computer Systems Laboratory
Stanford University

Abstract:

A theory of partial correctness proofs is formulated in Scott's logic of computable functions. This theory allows mechanical construction of verification conditions solely on the basis of a denotational language definition. Extensionally these conditions, the resulting proofs, and the required program augmentation are similar to those of Hoare style proofs; conventional input, output, and invariant assertions in a first order assertion language are required. The theory applies to almost any sequential language defined by a continuation semantics; for example, there are no restrictions on aliasing or side-effects. Aspects of "static semantics", such as type and declaration constraints, which are expressed in the denotational definition are validated as part of the verification condition generation process.

1. Introduction

Most existing program verification systems are based on some variant of Hoare's weak logic of programs [Fl67, Ho69]. The prevailing paradigm is to use a set of proof rules to construct "verification conditions" for a given program. Validity of these conditions implies partial correctness of the program. An "invariant assertions" has to be associated with each loop in the program; it provides an induction hypothesis to prove properties of this loop.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-80-C-0159.

An advantage of this method is its simplicity: to express the intention of a loop in an invariant in terms of program variables is easier for programmers than any other known method of performing induction proofs (short of automatic synthesis of invariants or induction hypotheses). Furthermore, a simple algorithm to construct verification conditions from a set of proof rules has been well known for quite some time [IL75].

Most existing implementations of Hoare's logic impose some restrictions on the programming language; for example aliasing and side-effects are often disallowed. This in itself is not serious. It is serious, however, that these restrictions are usually not expressed in the formal system itself, but rather as "English" comments. More generally, most systems of proof rules completely ignore questions of static semantics. Ligler [Li75] coined the term "surface properties" to characterize those aspects of semantics captured by proof rules. However, these are not principal limitations as is shown in [OC78, Sc78].

It has been argued repeatedly that a set of proof rules defines the semantics of a language. However, this is only true if rigorous soundness and consistency of proofs for these rules are given. One way to prove consistency of proof rules is to provide a model, for example a denotational or operational semantics of the language to be defined (complementary definition [Do76, HL74]).

In this paper we present a theory that enables the construction of verification conditions for a program augmented with input, output, and invariant assertions based solely on a denotational definition of the language in question. The assertion language can be any first-order language that contains certain "primitive" functions and predicates (see section 4.3). Necessary theorem proving is done within this logical language. Thus, a verification condition generator based on our theory exhibits a user interface similar to that of most rule based systems.

The class of languages to which our theory applies is virtually unrestricted; we merely require simple restrictions on the definitional language (see section 3); in particular, there are no limitations as to alias-

ing or side-effects. If the static semantics, i.e. type and declaration constraints, are part of the denotational definition, these constraints will be automatically validated during verification condition generation; no additional "English" restrictions are necessary.

Since the process of verification condition generation is completely described within Scott's functional formalism it is possible to combine proofs using fixed point induction and proofs using invariants.

The relation between Hoare's logic and denotational semantics has been investigated in several previous works [Cl77, MS76, Mi77]. It has been shown how to express predicate transformers in Scott's framework and how to prove consistency between a set of proof rules and a denotational definition.

In this paper we introduce the notion of "related" language definitions. Two definitions are related if they describe the same language from a "different point of view." We show that a continuation semantics and an equivalent predicate transformer semantics are related in this sense.

In the next section we will introduce some notations and summarize the theoretical foundations. In section 3 we introduce the notion of "related" language definitions. It is shown that under certain conditions a predicate transformer semantics can be constructed from a continuation semantics by simple textual substitution. In section 4 we introduce the assert statement. It is shown how a predicate transformer semantics can be used to generate verification conditions for programs augmented with sufficiently many assertions. Finally, in section 5 we discuss some issues relevant to the application of our theory. These include a worked example, some hints on a possible implementation, and the treatment of jumps, procedures and functions, and nondeterminism. The appendix contains the continuation semantics of a simple example language that will be used in examples throughout the paper.

2. Notations, Definitions

We assume that the reader is familiar with Scott-Strachey semantics [MS76, SS71, St77, Te76] and the underlying theory [Sc72]. In our notation we follow that of Stoy and Tennent [St77, Te76].

A *domain* D is a complete partial order (cpo) [Pl78], ordered with respect to $\sqsubseteq$ with the least element $\perp$. If D_1 and D_2 are two domains, then $D_1 + D_2$ and $D_1 \times D_2$ denote the *separated sum*, and *product spaces* respectively; these domains are ordered in the usual way. If $x \in D_1 + D_2$ then $x \mid D_i$ denotes the projection of x on D_i; it is defined to be $\perp$ if x is not in the component D_i. Conversely, if $x \in D_i$ then $x \mid D_1 + D_2$ is the imbedding of x in $D_1 + D_2$. For $x_1 \in D_1$, $x_2 \in D_2$ the pair (x_1, x_2) is an element of $D_1 \times D_2$. If $x \in D_1 \times D_2$ then x^i denotes to the i-th component of x.

D^* denotes the domain of finite lists over D. The notation $\langle x_1, \ldots, x_n \rangle$ refers to a list with the elements $x_1, \ldots, x_n$. We write $\langle\rangle$ for the empty list and use & for concatenation.

If S is a set, $S_\perp$ with $\perp \sqsubseteq s \in S$ is a *flat domain*. For a flat domain D we write *proper*(D) for the set of proper elements ($\neq \perp$) of D. $T = \{tt, ff\}_\perp$ is the domain of truthvalues. If $x_1 \in T$ and $x_2, x_3 \in D$ then the conditional **if** x_1 **then** x_2 **else** x_3 is a term in D. If $x_1 = \perp$ then the value of the conditional is $\perp$.

If D_1 and D_2 are two domains then $D_1 \to D_2$ is the domain of *continuous functions* from D_1 to D_2. If T is a term in D_2 and $x \in D_1$ then $\lambda x.T$ is a term in $D_1 \to D_2$. If $f \in D_1 \to D_2$ and $x \in D_1$ then $f\,x \in D_2$ is the result of applying f to x. Function application associates to the left. Alternatively we use the symbol ";" to denote function application; it has lower precedence than juxtaposition and associates to the right. For example $x\,y; u\,v; w\,z$ means $x\,y(u\,v(w\,z))$.

A function f is called *strict* if $f\perp = \perp$. For arbitrary f the function *strict* $f = \lambda x.$**if** $x = \perp$ **then** $\perp$ **else** $f\,x$ is strict and continuous (see e.g. [St77]).

For $f \in D_1 \to D_2$, $x_1 \in D_2$, and $x_2 \in D_1$ the term $f[x_1/x_2]$ denotes the function $\lambda y.$**if** $y = x_2$ **then** x_1 **else** $f\,y$. x_1 and x_2 may be lists in which case $f[x_1/x_2]$ denotes a simultaneous redefinition of f; if x_2 is a list whose elements are not distinct or if x_1 and x_2 are of different length we define $f[x_1/x_2] = \perp$.

If $f \in D \to D$ then $fix\,f \in D$ denotes the least fixed point of f. Continuity of f guarantees the existence of a least fixed point. The property

$$\text{If } f\,x \sqsubseteq x \text{ then } fix\,f \sqsubseteq x. \tag{1}$$

will be relevant later.

In addition to the truthvalues $T = \{tt, ff\}_\perp$ the two element cpo $T2$ of truthvalues with

$$T2 = \{true, false\} \text{ with } true \sqsubseteq false$$

will be of particular importance. $T2$ has the nice property that for t_1 and t_2 in $T2$ the implication $t_1 \supset t_2$ is true if and only if $t_2 \sqsubseteq t_1$.

A predicate over D is an element $p \in D \to T2$. That is, all predicates we consider will be continuous. This restriction is not a severe one because if D is a flat domain any strict function on D is continuous and any function defined on the proper elements of D can be extended to a strict function on D. By the ordering of $T2$ we have $\forall x.p(x) \supset q(x)$ iff $q \sqsubseteq p$.

A set S is directed if every finite subset has an upper bound in S. For a directed set S its least upper bound is written as $\bigsqcup S$. A function f is *inclusive* [MS76] if

for any directed set $S = \{x_i\}$

$$f(\bigsqcup\{x_i\}) \sqsubseteq \bigsqcup\{fx_i\}.$$

Inclusive predicates are admissible for fixed point induction.

3. Related denotational definitions

To be able to talk about language definitions as objects let us introduce the following notation. A denotational semantics $\mathcal{Q}$ is a collection of domains and functions. Domains are defined from a set of *primitive domains* $\{B_i\}$ by product, sum, function domain construction, and recursion. Functions on these domains are defined in typed λ-calculus from a fixed set of *primitive functions* $\{f_i\}$. To emphasize this dependence we write $\mathcal{Q} = \mathcal{Q}(B_1, \ldots, B_n, f_1, \ldots, f_m)$.

Primitive domains and functions are those not further defined in a language definition. For example, the flat integer cpo $N_\perp$ and addition on $N_\perp$ or the domain A of answers are considered primitive in most definitions. But one may also consider more complex domains and functions as primitive. For example, the semantics of a language could be described relative to a set of elementary statements with a fixed but not further defined meaning.

Given a definition $\mathcal{Q} = \mathcal{Q}(B_1, \ldots, B_n, f_1, \ldots, f_m)$ we will now investigate properties of a definition $\dot{\mathcal{Q}} = \mathcal{Q}(\dot{B}_1, \ldots, \dot{B}_n, \dot{f}_1, \ldots, \dot{f}_m)$ that has primitive domains and functions altered but is otherwise identical to $\mathcal{Q}$. We show that under certain conditions $\mathcal{Q}$ and $\dot{\mathcal{Q}}$ are related in a meaningful manner.

The reason for studying related definitions is that they express the semantics of a language from a different point of view, for example a definition which distinguishes different error situations and its related definition that maps all errors into the undefined element $\perp$. We will show later that for most continuation semantics one can define a related definition which is a predicate transformer semantics.

3.1. Relating domains

Let $\mathcal{Q} = \mathcal{Q}(B_1, \ldots, B_n, f_1, \ldots, f_m)$ and then $\dot{\mathcal{Q}} = \mathcal{Q}(\dot{B}_1, \ldots, \dot{B}_n, \dot{f}_1, \ldots, \dot{f}_m)$ then $\dot{\mathcal{Q}}$ consists of the same definitional clauses as $\mathcal{Q}$ with some primitive functions and domains replaced. Thus, transformations back and forth between $\mathcal{Q}$ and $\dot{\mathcal{Q}}$ are simple textual substitutions. For each domain D of $\mathcal{Q}$ there is a corresponding domain $\dot{D}$ of $\dot{\mathcal{Q}}$. For each term $t \in D$ of $\mathcal{Q}$ there is a corresponding term $\dot{t} \in \dot{D}$ of $\dot{\mathcal{Q}}$.

Let domains $B_1, \ldots, B_n$ and $\dot{B}_1, \ldots, \dot{B}_n$ be related by inclusive binary relations $\mathcal{R}_{B_i} \subseteq B_i \times \dot{B}_i$. We extend $\mathcal{R}$ to all domains of $\mathcal{Q}$ by defining relations $\mathcal{R}_D \in D \times \dot{D}$ according to

$$\mathcal{R}_D(x, \dot{x}) \quad iff \quad x = \dot{x}$$
for a primitive D that is not a parameter

$$\mathcal{R}_{D_1+D_2}(x, \dot{x}) \quad iff \quad \mathcal{R}_{D_1}(x \mid D_1, \dot{x} \mid \dot{D}_1) \wedge \mathcal{R}_{D_2}(x \mid D_2, \dot{x} \mid \dot{D}_2)$$

$$\mathcal{R}_{D_1 \times D_2}(x, \dot{x}) \quad iff \quad \mathcal{R}_{D_1}(x^1, \dot{x}^1) \wedge \mathcal{R}_{D_2}(x^2, \dot{x}^2)$$

$$\mathcal{R}_{D_1 \to D_2}(f, \dot{f}) \quad iff \quad \bigwedge\{\mathcal{R}_{D_2}(fg, \dot{f}\dot{g}) \mid \mathcal{R}_{D_1}(g, \dot{g})\}.$$

Note, if D_i is defined recursively, then so is $\mathcal{R}_{D_i}$. In this case $\mathcal{R}_{D_i}$ is defined to be the least fixed point satisfying the above definition. Unfortunately, the set constructor used in this definition is not continuous, not even monotonic. Therefore, the existence of a least fixed point is not guaranteed. To prove its existence we have to resort to the theory of inclusive predicates [MS76, Re74]. A detailed analysis is beyond the scope of this paper but corollary 5.1 in [Re74] applies to our situation and can be used to show that $\mathcal{R}_D$ is in fact well defined for any D. We will omit domain subscripts if they can be inferred from the context.

Observe, that for domains D independent of the parameters B_i we have $D = \dot{D}$ as well as $\mathcal{R}_D(d, \dot{d}) \equiv d = \dot{d}$.

Definition: Given $\mathcal{R}_{B_i}$ as above we say two definitions $\mathcal{Q} = \mathcal{Q}(B_1, \ldots, B_n, f_1, \ldots, f_m)$ and $\dot{\mathcal{Q}} = \mathcal{Q}(\dot{B}_1, \ldots, \dot{B}_n, \dot{f}_1, \ldots, \dot{f}_m)$ are *related* if for all f_i the relation $\mathcal{R}_D(f_i, \dot{f}_i)$ holds for the appropriate D (i.e. $f_i \in D$).

Note that in general suitable $\dot{f}_i$ need not exist for given $\mathcal{Q}$, B_i, $\dot{B}_i$, and $\mathcal{R}_{B_i}$.

3.2. Properties of related definitions

Theorem: Let $\mathcal{Q} = \mathcal{Q}(B_1, \ldots, B_n, f_1, \ldots, f_m)$ and $\dot{\mathcal{Q}} = \mathcal{Q}(\dot{B}_1, \ldots, \dot{B}_n, \dot{f}_1, \ldots, \dot{f}_m)$ be two related definitions. For an arbitrary term $t \in D$ of $\mathcal{Q}$ and the corresponding term $\dot{t} \in \dot{D}$ of $\dot{\mathcal{Q}}$ the relation $\mathcal{R}_D(t, \dot{t})$ holds.

Proof: The proof proceeds by induction on the structure of t.

a) Primitive functions: $\mathcal{R}(f_i, \dot{f}_i)$ holds by the definition of related semantics.

b) Application: assume inductively $\mathcal{R}(g, \dot{g})$ and $\mathcal{R}(x, \dot{x})$, then $\mathcal{R}(g\,x, \dot{g}\,\dot{x})$ follows by definition of $\mathcal{R}$.

c) Fixed points: $\mathcal{R}(\perp, \dot{\perp})$; if $\mathcal{R}(g, \dot{g})$ and $\mathcal{R}(x, \dot{x})$ part (b) yields $\mathcal{R}(g\,x, \dot{g}\,\dot{x})$. Since $\mathcal{R}$ is inclusive it is admissible for fixed point induction which yields

$$\mathcal{R}(\bigsqcup(g^n \perp), \bigsqcup(\dot{g}^n \perp)) \equiv \mathcal{R}(fix\,g, fix\,\dot{g}).$$

d) λ-abstraction: Let $t(x)$ be a term with free variable x. Given any function g such that $\mathcal{R}(g, \dot{g})$, then by induction hypothesis we have $\mathcal{R}(t(g), \dot{t}(\dot{g}))$. Since

this is true for arbitrary g, we have

$$\bigwedge\{\mathcal{R}((\lambda x.t)g, (\lambda x.\dot{t})\dot{g}) \mid \mathcal{R}(g, \dot{g})\}$$

and thus $\mathcal{R}(\lambda x.t, \lambda x.\dot{t})$.

e) Conditionals: immediate since $\mathcal{R}_T$ has to be the identity relation. Obviously, T cannot be replaced by another domain since then conditionals are not well defined.

∎

3.3. Predicate transformer semantics

Throughout the rest of this paper we only consider continuation semantics [SW74] $\mathcal{Q}_C = \mathcal{Q}(A, f_1, \ldots, f_m)$ with answer domain A. A continuation is a mapping from stores (S) to answers (A), i.e. $C = S \to A$. The meaning of a statement is described by its effect on the continuation that applies after the statement (tail function [Ma71]).

Let $q \in A \to T2$ be a predicate on answers. We define $\mathcal{R}_A \subseteq A \times T2$ as $\mathcal{R}_A(a, b) \equiv b = qa$. We will show below that given suitable functions $\dot{f}_i$ the related definition $\mathcal{Q}_q = \mathcal{Q}(T2, \dot{f}_1, \ldots, \dot{f}_m)$ is a predicate transformer semantics.

First let us consider selection of suitable functions $\dot{f}_i$. In practice the following rules suffice: if $a \in A$ is a constant take $\dot{a} = q(a)$; if $f \in D \to A$ take $\dot{f} = \lambda x.q(f\,x)$; if $d \in D$ for a D independent of A take $\dot{d} = d$.

If the language includes nondeterminism described by power domains [Pl76, Sm78] the situation is slightly more complicated. For example, in [Te77] Tennent uses the domain of continuations $C = S \to P(A)$. Alternatives are described by $\theta_1 \mid \theta_2 = \lambda\sigma.\theta_1\sigma \cup \theta_2\sigma$. For a given $q \in A \to T2$ we define $q^* \in P(A) \to T2$ according to

$$q^*(p) = \bigwedge\{qa \mid a \in p\}.$$

It follows that "$\mid$" and "$\sqcup$" are related functions, i.e. $\mathcal{R}(\text{"}\mid\text{"}, \text{"}\sqcup\text{"})$.

Our theory is applicable to any language definition $\mathcal{Q}_C$ for which $\mathcal{Q}_q$ can be constructed.

Let us now argue why $\mathcal{Q}_q$ is a predicate transformer semantics for the language defined by $\mathcal{Q}_C$. Given a program Θ, its meaning is determined by the term $\theta_{pre} = S[\![\Theta]\!]\rho_0\theta_0$ (assuming the semantics given in the appendix). Here θ_0 and ρ_0 are the initial continuation and environment respectively. Starting program Θ with store σ we get the answer $\theta_{pre}\sigma$.

Suppose we are given a predicate q on answers which we want to be true after program execution, i.e. we want $q(\theta_{pre}\sigma)$ to hold. Now by the the definition of $\mathcal{R}_A$ we have $q(\theta_{pre}\sigma) = \dot{\theta}_{pre}\dot{\sigma}$. Since S is independent of A the relation $\mathcal{R}_S$ is the identity, thus $\sigma = \dot{\sigma}$ and $q(\theta_{pre}\sigma) = \dot{\theta}_{pre}\sigma$. Consequently, $\dot{\theta}_{pre}$ is a *precondition* of Θ; i.e. if $\dot{\theta}_{pre}$ holds for the initial store σ and if program Θ terminates then q holds for the final answer of Θ. But $\dot{\theta}_{pre}$ is just the result of computing $\dot{S}[\![\Theta]\!]\dot{\rho}_0\dot{\theta}_0$ in $\mathcal{Q}_q$. Note, if Θ does not terminate its answer will be $\perp$. But unless $q = \lambda x.\mathit{false}$ monotonicity of q requires $q\perp = \perp$, i.e. q holds for the final answer of a nonterminating program. Thus, we cannot reason about termination within $\mathcal{Q}_q$.

In general, continuations of $\mathcal{Q}_q$ are predicates on stores that guarantee q for the final answer. Thus, $\dot{S}[\![\Theta]\!]\dot{\rho}$ is a predicate transformer that maps $\dot{\theta}$ into $\dot{\theta}_{pre} = \dot{S}[\![\Theta]\!]\dot{\rho}\dot{\theta}$ such that if $\dot{\theta}$ after Θ guarantees q then so does $\dot{\theta}_{pre}$ before Θ.

This notion of predicate transformer is different from Dijkstra's weakest liberal preconditions [Di76]. For example, $wlp([\![\text{goto } m]\!], P)$ cannot be defined meaningfully. In our method the use of a fixed exit condition q allows to define predicate transformers for jumps and error exits. Given the predicate q, correct label bindings can be computed for every environment; thus, the term $\dot{S}[\![\text{goto } m]\!]\dot{\rho}\dot{\theta} = \dot{\rho}[\![m]\!]$ correctly describes the precondition for the goto statement.

4. Program Proofs

In this section we introduce an assert statement in the programming language and show how it can be used together with the predicate transformer semantics constructed above to generate verification conditions. We do this for the language given in the appendix; domains used in this section refer to those of this definition. Constructions similar to those presented for our example language are immediate for many other languages.

4.1. The Assert Statement

Let $I_1, \ldots, I_n$ be identifiers and let P by a continuous predicate on values $P \in V \to T2$ an *assertion* $\in \mathit{Asrt}$ is a term of the form $P(I_1, \ldots, I_n)$.

We add a new statement of the form assert $P(I_1, \ldots, I_n)$ to our programming language. Inserting arbitrary assertions in a program will not change its semantics, thus we define

$$S[\![\text{assert } P(I_1, \ldots, I_n)]\!]\rho\theta = \theta.$$

Intuitively, an assertion serves the same purpose as in Hoare's logic. Formally, the assert statement is a means to associate a continuation with a particular point of the program. In analogy to $\mathcal{E}$ defining the meaning of expressions we define a function $\mathcal{A}$ defining the meaning

of assertions:

$$\mathcal{A} \in Asrt \to U \to S \to T2$$
$$\mathcal{A}[\![P(I_1,\ldots,I_n)]\!]\rho =$$
$$\lambda\sigma.P(\sigma(\rho[\![I_1]\!] \mid L),\ldots,\sigma(\rho[\![I_n]\!] \mid L)).$$

Note, that the assertion denotes an element of $S \to T2$, i.e. a continuation in $\mathcal{Q}_q$, while P is a predicate on values.

The predicate $q \in A \to T2$ is not an assertion in the above sense; it plays a special role since it is used to construct $\mathcal{Q}_q$ It is usually not meaningful to talk about values of program variables in q because variables do no longer exist once a program terminated. However, depending on the domain A, q may assert properties of an output file or it may assert that certain errors do not occur during program execution. For example, the predicate $q = \lambda x.x \neq \mathit{invalidindex}$ cannot in general be stated as a predicate on stores.

4.2. Generating Verification Conditions

Subsequently, we only consider a particular predicate transformer semantics $\mathcal{Q}_q$; since all terms refer to $\mathcal{Q}_q$ rather than $\mathcal{Q}_C$ we omit all primes.

The problem of generating verification conditions for a program Θ is to derive a set of conditions V_i and a predicate r on stores (precondition) such that

$$V_1,\ldots,V_n \vdash S[\![\Theta]\!]\rho_0\theta_0 \sqsubseteq r. \qquad (2)$$

If we are able to prove all V_i then this guarantees the usual partial correctness statement: if r holds for the initial state and if Θ terminates, then q is true for the final answer, i.e.

$$\forall\sigma.r\,\sigma \supset S[\![\Theta]\!]\rho_0\theta_0\sigma.$$

Formulas of the form (2) can be systematically derived by symbolic execution of meaning functions following the structure of the input program. We present several typical cases.

The semantics of the assert statement $S[\![\text{assert } P(I_1,\ldots,I_n)]\!]\rho\theta = \theta$ gives rise to

$$\theta \sqsubseteq \mathcal{A}[\![P(I_1,\ldots,I_n)]\!]\rho \vdash$$
$$S[\![\text{assert } P(I_1,\ldots,I_n)]\!]\rho\theta \sqsubseteq \mathcal{A}[\![P(I_1,\ldots,I_n)]\!]\rho. \quad (3)$$

One important point about this otherwise trivial rule is that the precondition $\mathcal{A}[\![P(I_1,\ldots,I_n)]\!]\rho$ is a constant not depending on θ.

Given a simple statement like an assignment, one can evaluate the meaning function, i.e. derive a formula

$$\vdash S[\![\Theta]\!]\rho\theta = r.$$

For a sequence of statements $\Theta_1;\Theta_2$ we get

$$V_1,\ldots,V_n \vdash S[\![\Theta_2]\!]\rho\theta \sqsubseteq r_1$$
$$V_{n+1},\ldots,V_m \vdash S[\![\Theta_1]\!]\rho r_1 \sqsubseteq r$$

and by monotonicity

$$V_1,\ldots,V_m \vdash S[\![\Theta_1;\Theta_2]\!]\rho\theta \sqsubseteq r.$$

Loops in the program are defined through terms of the form $fix\,\lambda\theta.T(\theta)$. If $T(\theta)$ is evaluated (for a symbolic θ) using the above rules a formula of the form

$$V_1(\theta),\ldots,V_n(\theta) \vdash T(\theta) \sqsubseteq r$$

is derived. If the code inside the loop is augmented with sufficiently many assertions the predicate r will be a constant not depending on θ. Since we did not make any assumption about θ the above formula holds for any θ, in particular it holds for r:

$$V_1(r),\ldots,V_n(r) \vdash T(r) \sqsubseteq r.$$

By (1) from section 2 this implies

$$V_1(r),\ldots,V_n(r) \vdash fix\,\lambda\theta.T(\theta) \sqsubseteq r$$

which again is a formula of the form (2) giving verification conditions for a loop.

4.3. Proving verification conditions

Verification conditions generated by our method are of the form $T_1 \sqsubseteq T_2$ where terms T_i are constructed from conditionals, redefinition ($x[u/v]$), predicates and functions occuring in assertions, and primitive functions on values used in the language definition. We assume that the chosen assertion language is strong enough to express all verification conditions constructed in the above way. Ignoring the special case of the initial continuation θ_0 each verification condition can be written in the form

$$\lambda\sigma.P(\sigma\alpha_1,\ldots,\sigma\alpha_n) \sqsubseteq \lambda\sigma.Q(\sigma\alpha_k,\ldots,\sigma\alpha_m)$$

where P and Q are strict predicates over V and all α_i are distinct locations ($\in L$), $1 \leq k \leq n+1, n \leq m$.

By properties of $T2$ verification conditions of the above form can be proven by showing

$$\forall\sigma.Q(\sigma\alpha_k,\ldots,\sigma\alpha_m) \supset P(\sigma\alpha_1,\ldots,\sigma\alpha_n)$$

or, after substituting x_i for $\sigma\alpha_i$

$$\forall x_i.Q(x_k,\ldots,x_m) \supset P(x_1,\ldots,x_n).$$

Note, that we have to quantify over the domain V (including $\perp$). This poses a slight problem since it rules out the use of a conventional theorem prover for first order logic. For example, axioms such as $x \neq x+1$ do not hold in $V = N_\perp$. A possible solution is to require a theorem prover which operates on domains instead of sets. But we can also argue that those cases where quantification over V is different from quantification

over proper elements of V result from erroneous programs (e.g. accessing uninitialized variables).

This rather informal argument can be made more precise as follows. Suppose we change the definition $\mathcal{Q}$ to a new definition $\hat{\mathcal{Q}}$ by adding the clause

$$S[\![\text{assert } P(I_1,\ldots,I_n)]\!]\rho\theta = \lambda\sigma.strict(\lambda\epsilon_1\ldots\epsilon_n.\theta\sigma)\,\sigma(\rho[\![I_1]\!])\cdots\sigma(\rho[\![I_n]\!]).$$

The meaning of a program Θ is the same in $\mathcal{Q}$ and $\hat{\mathcal{Q}}$ only if identifiers in assertions in Θ are declared and initialized.

To generate verification conditions for Θ in $\hat{\mathcal{Q}}$ we have to use the rule

$$\lambda\sigma.strict(\lambda\epsilon_1\ldots\epsilon_n.\theta\sigma)\,\sigma(\rho[\![I_1]\!])\cdots\sigma(\rho[\![I_n]\!]) \sqsubseteq \mathcal{A}[\![P(I_1,\ldots,I_n)]\!]\rho$$
$$\vdash\ S[\![\text{assert } P(I_1,\ldots,I_n)]\!]\rho\theta \sqsubseteq \mathcal{A}[\![P(I_1,\ldots,I_n)]\!]\rho.$$

Thus, whenever our original method generates a verification condition of the form

$$\lambda\sigma.P(\sigma\alpha_1,\ldots,\sigma\alpha_n) \sqsubseteq \lambda\sigma.Q(\sigma\alpha_k,\ldots,\sigma\alpha_m) \quad (4)$$

then the corresponding condition generated in $\hat{\mathcal{Q}}$ is

$$\lambda\sigma.strict(\lambda\,\epsilon_k\ldots\epsilon_m.P(\sigma\alpha_1,\ldots,\sigma\alpha_n))(\sigma\alpha_k)\cdots(\sigma\alpha_m) \sqsubseteq \lambda\sigma.Q(\sigma\alpha_k,\ldots,\sigma\alpha_m) \quad (5).$$

We noted above that condition (4) is equivalent to

$$\forall x_i \in V.Q(x_k,\ldots,x_m) \supset P(x_1,\ldots,x_n).$$

It is easy to see that condition (5) is equivalent to

$$\forall x_i \in proper(V).Q(x_k,\ldots,x_m) \supset P(x_1,\ldots,x_n).$$

Thus, we can conclude that it suffices to prove verification conditions for proper values only provided the program does not access undeclared or uninitialized variables in assertions – first order theorem proving is applicable.

5. Application

5.1. Example

The following simple example uses the language defined in the appendix. We assume a predicate transformer semantics with $q = \lambda\, a.true$. Thus, if the program aborts with some runtime error it will be partially correct. A meaningful exit condition is provided by the assertion placed at the end of the program. a and b are two arbitrary integer constants.

```
begin
  new x; new y; x ← a;
  while 0 ≤ (x ← x − 1) do
  begin new z = y;
    z ← z + b;
    assert P(x, y);
  end;
  assert R(y)
end
```

Let us call this program *pgm* with statement part *stmts*. With $\theta_0 = \lambda\sigma.true$ and initial store σ_0 the final answer of *pgm* is given by

$$S[\![pgm]\!]\perp\theta_0\sigma_0 = D[\![\text{new } x;\text{new } y]\!]\perp(\lambda\rho.C[\![stmts]\!]\rho\theta_0)\sigma_0.$$

(Since there are no labels J and j both evaluate to the empty list.) Abbreviating $\chi = \lambda\rho.C[\![stmts]\!]\rho\theta_0$ we have

$$\begin{aligned} &D[\![\text{new } x;\text{new } y]\!]\perp\chi\sigma_0 \\ &= D[\![\text{new } x]\!]\perp;(\lambda\rho.D[\![\text{new } y]\!]\perp\chi)\sigma_0 \\ &= D[\![\text{new } y]\!](\perp[\alpha_{\sigma_0}/x])\chi\sigma_1 \\ &= \chi(\perp[\alpha_{\sigma_0}/x][\alpha_{\sigma_1}/y])\sigma_2 \end{aligned}$$

where $\sigma_1 = \sigma_0[0/\alpha_{\sigma_0}]$ and $\sigma_2 = \sigma_1[0/\alpha_{\sigma_1}]$. We write α_σ for the α such that $\sigma\alpha = unused$. Thus, $\alpha_{\sigma_0} \neq \alpha_{\sigma_1}$ follows immediately.

$\rho_0 = \perp[\alpha_{\sigma_0}/x][\alpha_{\sigma_1}/y]$ is the environment in which we have to evaluate the statement part *stmts* of the outer block. The final answer is given as

$$\begin{aligned} &C[\![stmts]\!]\rho_0\theta_0\sigma_2 \\ &= (S[\![x \leftarrow a]\!]\rho_0; S[\![\text{while}\ldots]\!]\rho_0; S[\![\text{assert } R(y)]\!]\rho_0\theta_0)\sigma_2 \\ &= (S[\![\text{while}\ldots]\!]\rho_0; S[\![\text{assert } R(y)]\!]\rho_0\theta_0)\sigma_2[a/\rho_0[\![x]\!]] \end{aligned}$$

Let $\sigma_3 = \sigma_2[a/\rho_0[\![x]\!]] = \sigma_2[a/\alpha_{\sigma_0}]$ and

$$r = \mathcal{A}[\![\text{assert } R(y)]\!]\rho_0 = \lambda\sigma.R(\sigma\alpha_{\sigma_1}).$$

With the trivial verification condition $\theta_0 = \lambda\sigma.true \sqsubseteq q$ we get

$$\begin{aligned} &(S[\![\text{while}\ldots]\!]\rho_0; S[\![\text{assert } R(y)]\!]\rho_0\theta_0)\sigma_3 \\ &\sqsubseteq (S[\![\text{while}\ldots]\!]\rho_0\, r)\sigma_3 \\ &= fix(\lambda\theta.\mathcal{E}[\![0 \le (x \leftarrow x-1)]\!]\rho_0; \\ &\qquad cond(S[\![body]\!]\rho_0\theta,\, r))\sigma_3. \end{aligned}$$

To find an approximation for this fixed point we symbolically evaluate it with continuation θ. We first

consider the loop body:

$$
\begin{aligned}
&S[\![body]\!]\rho_0\theta \\
&= D[\![\text{new } z = y]\!]\rho_0; \\
&\quad \lambda\dot{\rho}.C[\![z \leftarrow z + b;\ \text{assert } P(x,y)]\!]\dot{\rho}\theta \\
&= C[\![z \leftarrow z + b;\ \text{assert } P(x,y)]\!]\rho_1\theta \\
&= \lambda\sigma.(C[\![\text{assert } P(x,y)]\!]\rho_1\theta)\sigma[b + \sigma(\rho_1[\![z]\!])/\rho_1[\![z]\!]] \\
&= \lambda\sigma.(C[\![\text{assert } P(x,y)]\!]\rho_1\theta)\sigma[b + \sigma\alpha_{\sigma_1}/\alpha_{\sigma_1}] \\
&\sqsubseteq \lambda\sigma.P(\sigma\alpha_{\sigma_0}, b + \sigma\alpha_{\sigma_1})
\end{aligned}
$$

where we used $\rho_1 = \rho_0[\rho_0[\![y]\!]/z] = \rho_0[\alpha_{\sigma_1}/z]$. The last step generates the verification condition

$$\theta \sqsubseteq p = \lambda\sigma.P(\sigma\alpha_{\sigma_0}, b + \sigma\alpha_{\sigma_1})$$

So far we found that given $\theta \sqsubseteq p$

$$
\begin{aligned}
&\mathcal{E}[\![0 \le (x \leftarrow x - 1)]\!]\rho_0; cond(S[\![body]\!]\rho_0\theta, r)\sigma_3 \\
&\sqsubseteq \mathcal{E}[\![0 \le (x \leftarrow x - 1)]\!]\rho_0; cond(p, r)\sigma_3 \\
&= \lambda\sigma.\text{if } 0 \le \sigma\alpha_{\sigma_0} - 1 \\
&\qquad \text{then } P(\sigma\alpha_{\sigma_0} - 1, b + \sigma\alpha_{\sigma_1}) \\
&\qquad \text{else } R(\sigma\alpha_{\sigma_0}).
\end{aligned}
$$

Since θ is not free in the right hand side we substitute the right hand side for θ in the verification condition and the fixed point term and get

$$
\begin{aligned}
&\lambda\sigma.\text{if } 0 \le \sigma\alpha_{\sigma_0} - 1 \\
&\qquad \text{then } P(\sigma\alpha_{\sigma_0} - 1, b + \sigma\alpha_{\sigma_1}) \\
&\qquad \text{else } R(\sigma\alpha_{\sigma_0}) \\
&\sqsubseteq \lambda\sigma.P(\sigma\alpha_{\sigma_0}, b + \sigma\alpha_{\sigma_1}) \\
&\vdash fix(\lambda\theta.\mathcal{E}[\![0 \le (x \leftarrow x - 1)]\!]\rho_0; \\
&\qquad cond(S[\![body]\!]\rho_0\theta, r))\sigma_3 \\
&\quad \sqsubseteq \lambda\sigma.\text{if } 0 \le \sigma\alpha_{\sigma_0} - 1 \\
&\qquad \text{then } P(\sigma\alpha_{\sigma_0} - 1, b + \sigma\alpha_{\sigma_1}) \\
&\qquad \text{else } R(\sigma\alpha_{\sigma_0}).
\end{aligned}
$$

Substituting a new variable x for $\sigma\alpha_{\sigma_0}$ and y for $\sigma\alpha_{\sigma_1}$ verification condition and precondition can be written as

$$
\begin{aligned}
&\forall x, y.P(x,y) \supset \\
&\quad \text{if } 0 \le x - 1 \text{ then } P(x - 1, b + y) \text{ else } R(y)
\end{aligned}
$$

and

$$\text{if } 0 \le a - 1 \text{ then } P(a - 1, b + 0) \text{ else } R(0)$$

respectively.

Assuming $P(x,y)$ is the predicate $a^*b = x^*b + y \wedge 0 \le x$ and $Q(y) \equiv y = a^*b$ then our technique requires proving the first order verification condition

$$
\begin{aligned}
&\forall x, y.a^*b = x^*b + y \wedge 0 \le x \supset \\
&\quad \text{if } 0 \le x - 1 \\
&\qquad \text{then } a^*b = (x - 1)^*b + b + y \wedge 0 \le x - 1 \\
&\qquad \text{else } a^*b = y \\
&\equiv true
\end{aligned}
$$

and the precondition

$$
\begin{aligned}
&\text{if } 0 \le a - 1 \\
&\qquad \text{then } a^*b = (a - 1)^*b + b + 0 \wedge 0 \le a - 1 \\
&\qquad \text{else } a^*b = 0 \\
&\equiv a \ge 0 \vee b = 0.
\end{aligned}
$$

5.2. Implementation

In principle an implementation of our theory proceeds by symbolic evaluation of the λ term denoting the meaning of a given program. The evaluator will simplify this term to a greater (wrt. $\sqsubseteq$) term; it will generate a verification condition whenever an assert statement is encountered.

Fixed points are evaluated by (possibly repeated) substitution. If this process does not eliminate the fixed point the program does not contain sufficiently many assertions. Note that such an evaluator will not necessarily require assertions for loops which are only executed a finite number of times; rather, repeated substitution will "unfold" those loops. Of course, an intelligent evaluator may apply fixed point induction if substitution is not successful.

The evaluator characterized above will automatically check all context conditions, type and declaration constraints that are part of the language definition; no distinction between static and dynamic semantics is necessary.

A simple-minded implementation will generate several isomorphic instances of the same verification condition. This is the case because the language definition may ask for the evaluation of the same term twice. Special provisions can detect this situation and eliminate duplicate conditions.

A verification condition generator including the above mentioned extensions as well as jumps and procedures (as described below) is being implemented at Stanford. The rules and strategies presented so far do not include an equivalent to the "frame rule" in Hoare's logic. In our implementation we are experimenting with various techniques to achieve the effect of frame rules.

5.3. Labels and Jumps

Loops constructed with goto's result in fixed points over environments; they can be eliminated in the very same way as fixed points over continuations. For example, consider the program fragment

```
begin
  Θ1;
  m : Θ2
end
```

in the environment ρ and with continuation θ. The environment that has label m bound to the correct continuation is given by

$$fix(\lambda\hat{\rho}.\rho[C[\![\Theta_2]\!]\hat{\rho}\theta/m]).$$

Generating verification conditions for $C[\![\Theta_2]\!]\hat{\rho}\theta$ and observing $\hat{\rho}[\![x]\!] = \rho[\![x]\!]$ if $x \neq m$ we derive

$$V(\hat{\rho}[\![m]\!], \rho, \theta) \vdash C[\![\Theta_2]\!]\hat{\rho}\theta \sqsubseteq T(\hat{\rho}[\![m]\!], \rho, \theta).$$

If sufficiently many assertions are contained in Θ_2, term T will be independent of $\hat{\rho}[\![m]\!]$, i.e. $T(\hat{\rho}[\![m]\!], \rho, \theta) = T(\rho, \theta)$. Therefore, we find

$$V(\hat{\rho}[\![m]\!], \rho, \theta) \vdash \rho[C[\![\Theta_2]\!]\hat{\rho}\theta/m] \sqsubseteq \rho[T(\rho, \theta)/m]$$

and, after substituting $\rho[T(\rho, \theta)/m]$ for $\hat{\rho}$,

$$V(T(\rho, \theta), \rho, \theta) \vdash \rho[C[\![\Theta_2]\!](\rho[T(\rho, \theta)/m])\theta/m] \sqsubseteq \rho[T(\rho, \theta)/m].$$

So, by (1) we conclude

$$V(T(\rho, \theta), \rho, \theta) \vdash fix(\lambda\hat{\rho}.\rho[C[\![\Theta_2]\!]\hat{\rho}\theta/m]) \sqsubseteq \rho[T(\rho, \theta)/m].$$

The same technique applies for several labels.

5.4. Procedures and Functions

The technique described so far is capable to deal with procedures and functions without further extensions.

First, consider a situation where a procedure is declared without any assertions. In this case the evaluator will bind the procedure name in the environment to a term denoting the proper procedure value. Upon call to the procedure this symbolic procedure value will be applied to a continuation; at this time verification conditions can be generated for the body and the call.

Clearly, this technique is very inefficient if many calls to the procedure occur. Also, it does not allow for recursion. We can improve on this situation by merely placing assert statements at the beginning and end of the procedure body. In this case the evaluator can generate verification conditions for the body (path from entry to exit assertion) at the point of declaration.

6. Conclusions

We have presented a theory which allows reasoning about predicates, predicate transformers, assertions, and inductive assertion proofs in the framework of Scott's logic of computable functions. Using this theory Floyd-Hoare style verification is possible without proof rules, thus eliminating restrictions on aliasing and side-effects and the need for consistency proofs.

The reader may argue that we have in effect given a set of "proof rules" to generate verification conditions. Yes, our theory may be looked at in this way. However, our "rules" pertain to the λ-calculus; they are not contrived for a particular programming language.

We were mainly concerned with the underlying theory and merely sketched a possible implementation of our theory. More experience with our prototype implementation is required to learn how best to handle some of the technical problems involved.

7. Acknowledgements

I'd like to thank Avra Cohn, Mike Gordon, Friedrich vonHenke, Dana Scott, and Bob Tennent for their comments on earlier versions of this paper. Olaf Owe has contributed to the implementation of this theory through valuable discussions.

8. References

[Cl77] **Clarke, E.M.:** *Program Invariants as Fixed Points*; Dept. of Computer Science, Duke University, CS—1977—5

[Di76] **Dijkstra, E.W.:** *A Discipline of Programming*; Prentice Hall, 1976

[Do76] **Donahue, J. E.:** *Complementary Definitions of Programming Language Semantics*; Lecture Notes in Computer Science 42, Springer, 1976

[Fl67] **Floyd, R. W.:** *Assigning Meanings to Programs*; Proceedings of Symp. in Applied Mathematics 19 (1967)

[GM77] **Gordon, M., Milner, R., Wadsworth, C.:** *Edinburgh LCF*; Internal report CSR-11-77, University of Edinburgh

[Ho69] **Hoare, C. A. R.:** *An Axiomatic Basis of Computer Programming*; CACM 12, Oct, pp 576-580 (1969)

[HL74] **Hoare, C. A. R., Lauer, P.E:** *Consistent and Complementary Formal Theories of the Semantics of Programming Languages*; Acta Informatica 3, pp135-154, (1974)

[IL75] **Igarashi, S., London, R. L., Luckham, D. C.:** *Automatic Program Verification 1: Logical Basis and Its Implementation*; Acta Informatica, Vol 4, pp 145-182 (1975)

[Li75] **Ligler, G.:** *Surface Properties of Programming Language Constructs*; in Proving and improving programs, G.Huet, G. Kahn (ed.), Arc-et-Senans, 1975

[Ma71] Mazurkiewicz, A.: *Proving Algorithms by Tail Functions*; Information and Control, 18 (1971), pp220-226

[MS76] Milne, R., Strachey, C.: *A theory of programming language semantics*; Chapman and Hall, London 1976

[Mi77] Milne, R.: *Transforming Predicate Transformers*; IFIP working conference on formal description of programming concepts, Saint Andrews, 1977

[OC78] Oppen, D. C., Cartwright, R: *Reasoning about recursively defined data structures*; Proc. fifth ACM Symp. on Principles of Programming languages, 1978

[Pl76] Plotkin G.: *A powerdomain construction.*; SIAM Journal of Computing 5, 1976, 452-487.

[Pl78] Plotkin, G.: T^{ω} *as a Universal Domain*; Journal of computer and system sciences 17, pp 209-236 (1978)

[Re74] Reynolds, J.C.: *On the Relation between Direct and Continuation Semantics*; Proc. 2nd Coll. on Automata, Languages and Programming, Saarbrücken, pp. 157 - 168, 1974

[Sc78] Schwartz, R.L.: *An Aziomatic Semantic Definition of ALGOL 68*; CS Dept, UCLA, UCLA-34-P214-75, Aug. 78

[Sc72] Scott, D.: *Lattice Theory, Data Types and Semantics*; NYU Symp. on Formal Semantics, Prentice-Hall, New York (1972)

[SS71] Scott, D., Strachey, C.: *Toward a Mathematical Semantics for Computer Languages*; Tech. Monograph PRG-6, Programming Research Group, University of Oxford (1971)

[Sm78] Smyth M.B.: *Power domains.*; Journal of Computer and System Sciences 16, 1978, 23-36.

[St77] Stoy, J.: *Denotational Semantics — The Scott-Strachey Approach to Language Theory*; MIT Press, Cambridge (1977)

[SW74] Strachey, C., Wadsworth, C. P.: *Continuations, a Mathematical Semantics for Handling Full Jumps*; Technical Monograph PRG-11, Oxford University, 1974

[Te76] Tennent, R.D.: *The Denotational Semantics of Programming Languages*; CACM, 19 (1976) pp437-453

[Te77] Tennent, R.D.: *A Denotational Definition of the Programming Language Pascal*; Tech. Report 77-47, Queen's University, Kingston, Ontario (1977), revised 1978

9. Appendix: An example language

The following simple example language is used throughout the paper to demonstrate our approach. The language has been kept simple; for example static semantics are not described. There are no data types and only simple control structures.

However, some unconventional features are included in order to demonstrate the treatment of aliasing and side effects:

— the declaration *new* $I = J$ declares I to be an alias for J;

— assignment is allowed as an expression to allow for side effects.

The language contains an "assert statement"; its sole purpose is to include predicates in the program text; its execution has no effect.

The definition is written in the style of [Te77]. Details of the memory allocation are described axiomatially (for an α such that ...); suitable continuous functions for memory allocation can be found in [MS76].

9.1. Abstract syntax

$\Delta \in Dec$ — Declarations
$E \in Exp$ — Expressions
$\Gamma \in Com$ — Commands
$\Theta \in Stm$ — Statements
$I \in Id$ — Identifiers
$F \in Con$ — Constant symbols

$\Delta ::=$ **new** I | **new** $I_1 = I_2$ | $\Delta_1; \Delta_2$
$E ::= I := E \mid F(E_1, \ldots, E_n) \mid I \mid N$
$\Gamma ::= I : \Theta \mid \Gamma_1; \Gamma_2 \mid \Theta$
$\Theta ::= I := E$ | **while** E **do** Θ |
if E **then** Θ_1 **else** Θ_2 | **dummy** |
goto I | **assert** $P(I_1, \ldots, I_n)$ |
begin $\Delta; \Gamma$ **end**

9.2. Semantic Domains

$\epsilon \in V$ — Values
$\alpha \in L$ — Locations
$\sigma \in S = L \to V + \{unused\}$ — Stores
A — Answers
$\theta \in C = S \to A$ — Continuations
$\kappa \in K = V \to C$ — Expression continuations
$\chi \in X = U \to C$ — Declaration continuations
$\rho \in U = Id \to (L + C)$ — Environments

The domains A, L and V are left unspecified; both are flat lattices.

9.3. Auxilliary functions

$$update \in L \to V \to C \to C$$
$$content \in L \to K \to C$$
$$cond \in C \to C \to K$$

$$update\ \alpha\epsilon\theta = \lambda\sigma.\theta(\sigma[\epsilon/\alpha])$$
$$content\ \alpha\kappa = \lambda\sigma.\kappa(\sigma\alpha)\sigma$$
$$cond\ \theta_1\theta_2 = \lambda\epsilon.\text{if } \epsilon \text{ then } \theta_1 \text{ else } \theta_2)$$

9.4. Valuations

$$M \in Con \to V^* \to V$$
$$D \in Dec \to U \to X \to C$$
$$\mathcal{E} \in Exp \to U \to K \to C$$
$$C \in Com \to U \to C \to C$$
$$S \in Stm \to U \to C \to C$$
$$J \in Com \to U \to C \to C^*$$
$$j \in Com \to Id^*$$

M is not further defined here.

$$D[\![\Delta_1;\Delta_2]\!]\rho\chi = D[\![\Delta_1]\!]\rho;\lambda\dot{\rho}.D[\![\Delta_2]\!]\dot{\rho}\chi$$
$$D[\![\text{new } I]\!]\rho\chi = \lambda\sigma.\chi\rho[\alpha/I]\sigma[0/\alpha]$$
$$\text{for some } \alpha \text{ such that } \sigma\alpha = unused$$
$$D[\![\text{new } I_1 = I_2]\!]\rho\chi = \chi\rho[\rho[\![I_2]\!] \mid L/I_1]$$

$$\mathcal{E}[\![I := E]\!]\rho\kappa = \mathcal{E}[\![E]\!]\rho;\lambda\epsilon.update\ (\rho[\![I]\!] \mid L)\epsilon\ (\kappa\epsilon)$$
$$\mathcal{E}[\![I]\!]\rho\kappa = content\ (\rho[\![I]\!] \mid L)\kappa$$
$$\mathcal{E}[\![F(E_1,\ldots,E_n)]\!]\rho\kappa = \mathcal{E}[\![E_1]\!]\rho;\ \lambda\epsilon_1.\mathcal{E}[\![E_2]\!]\rho;\ \lambda\epsilon_2.\mathcal{E}[\![E_3]\!]\rho;\ \ldots\ \lambda\epsilon_{n-1}.\mathcal{E}[\![E_n]\!]\rho;\ \lambda\epsilon_n.M[\![F]\!]\kappa\langle\epsilon_1,\ldots,\epsilon_n\rangle$$

$$C[\![I : \Theta]\!] = S[\![\Theta]\!]$$
$$C[\![\Theta]\!] = S[\![\Theta]\!]$$
$$C[\![\Gamma_1;\Gamma_2]\!]\rho\theta = C[\![\Gamma_1]\!]\rho;C[\![\Gamma_2]\!]\rho\theta$$

$$S[\![\text{dummy}]\!]\rho\theta = \theta$$
$$S[\![\text{assert } P(I_1,\ldots,I_n)]\!]\rho\theta = \theta$$
$$S[\![I := E]\!]\rho\theta = \mathcal{E}[\![E]\!]\rho;\lambda\epsilon.update\ (\rho[\![I]\!] \mid L)\epsilon\theta$$
$$S[\![\text{while } E \text{ do } \Theta]\!]\rho\theta = fix(\lambda\dot{\theta}.\mathcal{E}[\![E]\!]\rho;cond(S[\![\Theta]\!]\rho\dot{\theta})\theta)$$
$$S[\![\text{if } E \text{ then } \Theta_1 \text{ else } \Theta_2]\!]\rho\theta = \mathcal{E}[\![E]\!]\rho;cond(S[\![\Theta_1]\!]\rho\theta)(S[\![\Theta_2]\!]\rho\theta)$$
$$S[\![\text{goto } I]\!]\rho\theta = \rho[\![I]\!] \mid C$$
$$S[\![\text{begin } \Delta;\Gamma \text{ end}]\!]\rho\theta = D[\![\Delta]\!]\rho;\lambda\rho_0.C[\![\Gamma]\!]\rho_1\theta$$
$$\text{where } \rho_1 = fix(\lambda\dot{\rho}.\rho_0[J[\![\Gamma]\!]\dot{\rho}\theta/j[\![\Gamma]\!]])$$

$$J[\![I : \Theta]\!]\rho\theta = \langle S[\![\Theta]\!]\rho\theta\rangle$$
$$J[\![\Theta]\!]\rho\theta = \langle\rangle$$
$$J[\![\Gamma_1;\Gamma_2]\!]\rho\theta = J[\![\Gamma_1]\!]\rho(C[\![\Gamma_2]\!]\rho\theta)\&J[\![\Gamma_2]\!]\rho\theta$$

$$j[\![I : \Theta]\!] = \langle I\rangle$$
$$j[\![\Theta]\!] = \langle\rangle$$
$$j[\![\Gamma_1;\Gamma_2]\!] = j[\![\Gamma_1]\!]\&j[\![\Gamma_2]\!]$$