Proving Refinement Transformations Using Extended Denotational Semantics

Victor L. Winter*
Intelligent Systems and Robotics Center
Sandia National Laboratories
Dept 9622, P.O. Box 5800
Albuquerque, NM 87185-0660, U.S.A.
vlwinte@sandia.gov

James M. Boyle†
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, U.S.A.
boyle@mcs.anl.gov

Abstract

TAMPR is a fully automatic transformation system based on syntactic rewrites. Our approach in a correctness proof is to map the transformation into an axiomatized mathematical domain where formal (and automated) reasoning can be performed. This mapping is accomplished via an extended denotational semantic paradigm.

In this approach, the abstract notion of a program *state* is distributed between an *environment* function and a *store* function. Such a distribution introduces properties that go beyond the *abstract state* that is being modeled. The reasoning framework needs to be aware of these properties in order to successfuly complete a correctness proof.

This paper discusses some of our experiences in proving the correctness of TAMPR transformations.

*This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

†This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by the BM/C3 directorate, Ballistic Missle Defense Organization, U.S. Department of Defense.

1 Denotational Semantic Paradigm

Denotational semantics is an approach that is used to give formal semantics to a language whose syntax is defined in terms of a context free grammar. Given a language $L(G)$, one constructs a set of *valuation functions* that map elements of $L(G)$ to *expressions* belonging to some mathematical language $\mathcal{L}_M$, whose semantics is assumed to be known.

In denotational semantics an element, $p \in L(G)$, is represented in terms of its *syntax derivation tree* (SDT). This SDT shows, in a graphical form, the grammar productions that were used to generate p, and it is this SDT that is given a semantics (i.e., defined) through *valuation functions.*

The standard description of valuation functions is that they "define the meaning of an SDT, having root n, in terms of an expression consisting of (1) elements of $\mathcal{L}_M$ and (2) valuation functions for the nodes that are the immediate descendents of n". Since the relationship between n and its immediate descendants has a one-to-one correspondence with a grammar production, another way to view valuation functions is as follows:

1. Given a grammar production of the from: $<S> \rightarrow <A>\ b\ <C>$, where b is a terminal symbol.

2. $V_s \overset{\text{def}}{=} \lambda$ input_sdt. λ other_inputs. $e((V_a[[<A>]]\ x),(V_b[[<B>]]\ y))$. Here V_s is a valuation function for the set of SDT's having $<S>$ as the root and $<A>\ b\ <C>$ as its immediate descendants. The definition of V_s is given in terms of a λ-Calculus expression. The first parameter of this expression represents the SDT whose semantics is being defined by the valuation function V_s.
Generally, the semantics of an SDT will itself be a function. In this case, the semantics is a function from *other_inputs* to the "type of the expression e". The expression e consists of objects (e.g., constants and functions) from $\mathcal{L}_M$ in addition to $(V_a[[<A>]]\ x)$ and $(V_b[[<B>]]\ y)$ which themselves are valuation functions. The function V_a has a signature similar to V_s (i.e., it takes an *input_sdt* and *other_inputs*) and is passed $<A>$ and x as its actual parameters. The notational convention is that SDT's are surrounded in "double square brackets" to remind the reader that the object is a syntax derivation tree an not just a "flat" string. Hence we write $(V_a[[<A>]]\ x)$ instead of $(V_a<A>\ x)$.

When defining the semantics of a language $L(G)$ using denotational semantics, one begins with (selects) a mathematical foundation $\mathcal{L}_M$ and constructs *valuation functions* that assign *meanings* to the nonterminal/terminal symbols in G. The goal is to construct a valuation function, C_{PROG}, that can be used to determine the *meaning* of entire programs (i.e., elements of $L(G)$).

Programs in most programming languages can be viewed as functions from *states* to *states.* Here *state* is used in the standard sense and refers to "variables and the values to which they are bound". The *meaning* of a program then, is the sequence of states it passes through during the course of its execution. Denotational semantics is used to define programs (i.e., the execution of programs) in terms of such sequences. This is accomplished by constructing a mathematical state space, $\mathcal{M}$ within $\mathcal{L}_M$. Generally, $\mathcal{M}$ will consist of an environment function, ε, and a store function, s. The environment function ε maps identifiers to storage locations, and the store function s maps storage locations to *denotable values.* A *denotable value* is a value that an identifier in the language can represent (e.g., integers, reals, etc.).

After a suitable $\mathcal{M}$ has been created, denotational definitions can now be constructed that define the execution semantics of a program in terms of state sequences in $\mathcal{M}$.

2 The Semantics of Schemas

2.1 Overview

The objective of this work is to use denotational semantics as a vehicle to enable reasoning about the correctness of refinement transformations. In particular, we are interested in proving the correctness of TAMPR transformations [1][2].

In general, refinement transformations can be viewed as rewrite rules having the form:

$$t_{pattern} \Rightarrow t_{replacement}$$

In TAMPR, the *patterns* and *replacements* of refinement transformations are viewed in terms of SDT's, whose roots referred to as their *dominating symbols.* Furthermore, these SDT's, which in this context we refer to as *schemas*, are generally incomplete in the sense that they contain nonterminal symbols as leaf elements. A nonterminal symbol in a leaf position is referred to as a *schema variable.* A *schema variable* will "match" (i.e., unify with) all SDT's having the same dominating symbol. It is through *schema variables*, that schemas have the ability to match more than one SDT, and it is through this matching ability that a transformation whose *pattern* contains one or more *schema variables* will have general applicability.

Unfortunately, standard denotational semantics is not concerned with the semantics of schema variables. Standard valuation functions are not defined in cases where they encounter a nonterminal symbol having no subtrees. Since schemas often contain such nonterminals (i.e., schema variables), the semantics of schemas are also undefined. This needs to be remedied if one wishes to use the denotational semantics of a language as a basis for reasoning about the correctness of transformations.

2.2 *Delta-Functions*

Fortunately, the denotational semantics of a language provides enough information to allow the semantics for nonterminals (i.e., schema variables) to be determined. For example, consider the following partial grammar:

$$\begin{array}{ll} <\!expr\!> & \rightarrow <\!id\!> \mid \ldots \\ <\!id\!> & \rightarrow x \mid y \mid z \end{array}$$

A continuation semantics of <*expr*> and <*id*> might be:

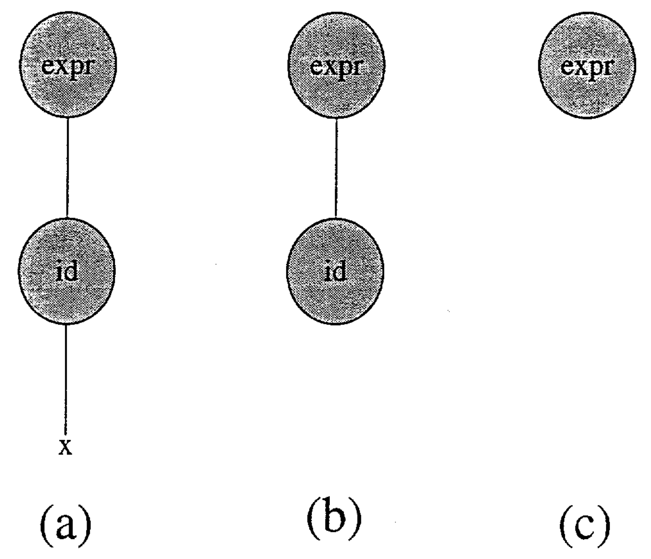
$$\begin{array}{l} E_{expr}[[<\!id\!>]] \overset{\text{def}}{=} \lambda\ \varepsilon.\ \lambda\ s.\ \lambda\ k.\ E_{id}[[<\!id\!>]]\ \varepsilon\ s\ k \\ E_{id}[[x]] \overset{\text{def}}{=} \lambda\ \varepsilon.\ \lambda\ s.\ \lambda\ k.\ k(s(\varepsilon(x))) \\ E_{id}[[y]] \overset{\text{def}}{=} \lambda\ \varepsilon.\ \lambda\ s.\ \lambda\ k.\ k(s(\varepsilon(y))) \\ E_{id}[[z]] \overset{\text{def}}{=} \lambda\ \varepsilon.\ \lambda\ s.\ \lambda\ k.\ k(s(\varepsilon(z))) \end{array}$$

Here k is a traditional continuation function having the signature:

$$k:\ denotable\_value \rightarrow\ store$$

For more on continuations see [5].

Using the denotational semantic definitions given above we consider the semantics of the following three SDT's:



Three SDT's

In this example, the SDT (a) evaluates to a *denotable value* that is bound to the identifier x. SDT (b) has a schema variable as its leaf, and so does SDT (c). What can we say about the semantics of (b) and (c)? Well, we know that k is the continuation for both SDT's. We also know that k expects an input of type *denotable value*. Thus we conclude that SDT (b) and SDT (c) both will ultimately evaluate to a *denotable value* (actually, evaluation to an undefined value is also possible but that is beyond the scope of this paper). With respect to the above example we cannot pin down the semantics of (b) and (c) any tighter. In summary then, the *meaning* of any syntax derivation tree having <*expr*> as its root will be an element belonging to the set of *denotable values*.

What else can one say about the denotable value corresponding to an SDT having <*expr*> as its dominating symbol? Consider an occurrence, in an actual program p, of an SDT e_1, having <*expr*> as its dominating symbol. Consider a point in the execution of p when the SDT e_1 is evaluated (i.e., e_1 is executed). The evaluation of e_1 will take place with respect to a specific environment and store. In particular the evaluation of e_1 is generally *dependent upon* (i.e., a function of) the environment and store in which it is evaluated. Thus one thing that we know about the schema variable <*expr*>, is that the semantics of every SDT having this root is of the form:

$$\lambda\ (\varepsilon, s).\ \Delta_v(\varepsilon, s)$$

Where Δ_v is a semantic function having the signature:

$$\Delta_v : \ \textit{environment} \times \ \textit{store} \rightarrow \ \textit{denotable value.}$$

A similar result holds for the schema variable <*id*>.

Exploring this idea further, let us consider the nonterminal <*assign*> denoting the set of assignment statements for a side-effect free Pascal-like language. For such a language, executing an instantiation of <*assign*> will result in a (single) change to the store. If the denotational semantics of the language under consideration defines assignments as "commands that take an environment and a store as input and produce a store as output", then the corresponding delta-function for <*assign*> will be:

$$\lambda\ (\varepsilon, s).\ \Delta_s(\varepsilon, s)$$

Where Δ_s is a semantic function having the signature:

$$\Delta_s : \ \textit{environment} \times \ \textit{store} \rightarrow \ \textit{store.}$$

Similarly, execution of an arbitrary declaration will result in a change to the environment (i.e., Δ_ε).

Note that the abstract semantics of a nonterminal like <*assign*> can be described in terms of a function that takes an environment and a store as input and returns a new "changed" store. Because many nonterminals have an abstract semantics that can be described in terms of such a "change", we have coined the term *delta-function* to describe an abstract valuation function that gives the semantics of a nonterminal. In general, we think of *delta-functions* as describing the "change in meaning" across a nonterminal.

2.3 The Importance of *Delta-Functions*

We would like to point out that *delta-functions*, in the semantics of schemas, play a role similar to that played by *variables* in standard algebraic expressions. However it would be incorrect to simply use generic *variables* in place of *delta-functions*. Consider the following transformation which replaces "if <*be*> then <*stmt*>$_1$ else <*stmt*>$_1$;<*stmt_tail*>$_1$" with "<*stmt*>$_1$;<*stmt_tail*>$_1$".

$$\mathcal{T}_1 \stackrel{\text{def}}{=} \begin{cases} <\!\mathit{stmt\_tail}\!>\{\ \text{if } <\!\mathit{be}\!> \text{ then } <\!\mathit{stmt}\!>_1 \text{ else } <\!\mathit{stmt}\!>_1; \\ \qquad\qquad <\!\mathit{stmt\_tail}\!>_1\} \\ \qquad\qquad \} \\ \Rightarrow \\ <\!\mathit{stmt\_tail}\!>\{<\!\mathit{stmt}\!>_1; \\ \qquad\qquad <\!\mathit{stmt\_tail}\!>_1 \\ \qquad\qquad \} \end{cases}$$

Given the standard semantics for the if-then-else construct, one might conclude that this transformation is correct. However, the correctness of this transformation not only depends upon the semantics of the if-then-else construct, but also upon whether the evaluation of boolean expressions, in the language under consideration, can cause side-effects. When using *delta-function* semantics for the nonterminal <*be*> this constraint becomes explicit, and a correctness proof will not "go through" for languages where such side-effects are possible. In contrast, when using a generic *variable* in place of <*be*> this information will not be present and must be accounted for by some other means.

2.4 Theoretical Considerations

It should be noted that there are many factors that determine just how specific a *delta-function* can be. For example, in a language that supports parallel assignments, the most general *delta-function* for a <*parallel_assign*> can only say that one or more identifiers will be assigned new values. Contrast this with an assignment statement in a sequential language where a side-effect free assignment will change the value of exactly one identifier.

Also, the examples we have considered above are quite simple. Nontrivial valuation functions and continuations can exist within a denotational semantics. For some of these, it is not immediately obvious what the appropriate and relevant *delta-functions* are.

Finally, in addition to inherent properties of the language, properties established by preceding transformations can also have an effect on *delta-functions.* Note that, applying a sequence of transformations to a specification/program s will result in a program p having certain syntactic and semantic properties deriving from the canonical forms achieved by the transformations in the sequence. For example, a program can be transformed into a cannonical form where evaluation of boolean expressions in conditional statements will **not** cause side-effects regardless of the general policy regarding side-effects that is supported by the language. To see this consider the following transformation:

$$\mathcal{T}_2 \overset{\text{def}}{=} \left\{ \begin{array}{l} <\!\mathit{stmt\_tail}\!>\{x := <\!\mathit{be}\!>\ ; \\ \qquad\qquad\quad \text{if } x \text{ then } <\!\mathit{stmt}\!>_1 \text{ else } <\!\mathit{stmt}\!>_1; \\ \qquad\qquad\quad <\!\mathit{stmt\_tail}\!>_1 \\ \qquad\qquad\quad \} \\ \Rightarrow \\ <\!\mathit{stmt\_tail}\!>\{x := <\!\mathit{be}\!>\ ; \\ \qquad\qquad\quad <\!\mathit{stmt}\!>_1; \\ \qquad\qquad\quad <\!\mathit{stmt\_tail}\!>_1 \\ \qquad\qquad\quad \} \end{array} \right.$$

This transformation can be applied in general, because it provides the context for its application—namely that the boolean expression of a conditional test consist of a single variable. However, suppose a transformation sequence has been applied to a program so that this property holds for all conditional statements within the program. For such a program the transformation $\mathcal{T}_1$ given earlier is correct! The explaination of this comes from the realization that transformation sequences can alter the semantics of *delta-functions.*

In general, the properties established by preceding transformations can impact the semantics of delta-functions of future transformations that are used to further refine p. In the presence of such properties, one can think of a nonterminal as having a family of *delta-functions*: a most general *delta-function* which results from the semantics of the language, and other more specific ones that incorporate properties established by prior transformations.

We have found that for many transformations, using the most general *delta-function*, which can usually be determined by inspection, is sufficient to permit a correctness proof to be obtained. However, because of the theoretical subtleties in determining the exact semantics of delta-functions (as mentioned above), we are developing an automated procedure for determining the semantics of *delta-functions* with respect to a given set of denotational semantic definitions. We are also looking into how transformations can effect *delta-functions.*

3 The Refinement Relation in $\mathcal{M}$

3.1 Motivation

The objective of TAMPR transformations is to introduce and restructure computation in a manner consistent with the notion of refinement. In general, refinement can have two possible effects on the precondition (initial state) and postcondition (final state) of a code segment, c, corresponding to a schema: refinement may logically weaken the precondition of c, or refinement may logically strengthen the postcondition of c.

To prove the general correctness of a refinement transformation, one must prove that *any* code segment matching the *pattern* of the transformation will be *refined* by the correspondingly instantiated *replacement* schema. In our paradigm, this is accomplished by demonstrating that the replacement schema will, for all possible instantiations, produce an *(abstract) state* that is a refinement of the *(abstract) state* produced by the correspondingly instantiated *pattern.*

The previous paragraph motivates the need for reasoning about *abstract states.* In our denotational semantics, abstract state is captured by $\mathcal{M}$. Therefore we need to be able to reason about refinement relationships within $\mathcal{M}$.

3.2 The State Space of a Denotationally Defined Computation

To give a full and correct description of the scope of identifiers, the state space $\mathcal{M}$, for most denotationally defined languages is represented by the cross product of an environment function, ε, and a store function, s (and possibly some additional constructs such as counters). Note that in this representation, obtaining the value corresponding to an identifier requires two steps: the *environment* function maps the identifier to a storage location, and the *store* function maps that storage location to a *denotable value.*

For example, consider the following code segment:

$x := 5;$
$y := x + 3;$

Let $(\varepsilon_1,\ s_1)$ denote the environment and store tuple that exist at the point in the program before the execution of the above code segment. After the code segment has been executed, the following environment-store tuple

(ε_2, s_2) will be produced. Here $\varepsilon_1 = \varepsilon_2$, and s_2 is a store that is identical to s_1 except that the storage locations corresponding to x and y will have the values 5 and 8 respectively. From this example, one can see that, when taken together, the environment and store functions provide the *(abstract) state* information of a program.

The abstract state is important because it provides a basis for verification. In traditional verification of programs, a code segment is proved to be correct by showing that if the execution of the code segment is begun in an abstract state satisfying a given precondition, then it will terminate in an abstract state satisfying a given postcondition.

The transformational perspective is somewhat different, but nevertheless related. In an application of a transformation of the form $t_{pattern} \Rightarrow t_{replacement}$, a fragment of code matching $t_{pattern}$ is replaced with the fragment of code corresponding to $t_{replacement}$. If the semantics of the programming language allow us to conclude that the execution of any fragment of code corresponding to $t_{replacement}$ will result in an *abstract state* that is a refinement of the *abstract state* produced by executing the fragment of code matched by $t_{pattern}$, then we can conclude that the substitution (i.e., the transformation) is *correctness preserving.* It is easy to show that correctness preservation is simply a projection of the traditional notions of program correctness onto program substitution (i.e., transformation).

3.3 Refinement Properties in $\varepsilon \times s$

The domain $\mathcal{M} \overset{\text{def}}{=} \varepsilon \times s$ forms a refinement lattice with $m_\perp \overset{\text{def}}{=} (\varepsilon_\perp, s_\perp)$ being the bottom element and $m_\top \overset{\text{def}}{=} (\varepsilon_\top, s_\top)$ denoting the top element. The components of $m_\perp$ and $m_\top$ are defined as follows:

$$\begin{aligned}
\varepsilon_\perp &\overset{\text{def}}{=} (\lambda\, x.\ \perp) \\
s_\perp &\overset{\text{def}}{=} (\lambda\, x.\ \perp) \\
\varepsilon_\top &\overset{\text{def}}{=} (\lambda\, x.\ \top) \\
s_\top &\overset{\text{def}}{=} (\lambda\, x.\ \top)
\end{aligned}$$

Since one of the purposes of the environment and the store is to capture the notion of *state*, we generally consider environment and store functions in pairs. That is, we only consider a function to be an environment function (or a store function) when it is part of a tuple belonging to $\mathcal{M}$. For example, when we mention the store s_1, the implication is that this store is part of $m_1 \overset{\text{def}}{=} (\varepsilon_1, s_1)$, similarly ε is part of $m \overset{\text{def}}{=} (\varepsilon, s)$. From here on out, we will use the terms "element of $\mathcal{M}$" and "*state*" interchangeably. We make explicit only that portion of the state that is necessary to facilitate understanding.

Before discussing refinement on $\varepsilon \times s$ we begin with a few definitions.

Definition 1 ***(Function Alteration.)*** *Let ε denote an arbitrary environment function. The notation $[x \mapsto \alpha]\varepsilon$ denotes an environment having the same mapping as ε for all identifiers except x. For $[x \mapsto \alpha]\varepsilon$ the storage location (i.e., the output of the function) associated with x is α. For more on this notation see [5].*

Definition 2 *General refinement on functions. Given any two functions f and g such that $f : D_1 \rightarrow D_2$ and $g : D_1 \rightarrow D_2$.*

$$f \sqsubseteq g \overset{\text{def}}{=} \forall x \in D_1, f(x) \sqsubseteq g(x)$$

Definition 3 $f \equiv g \Leftrightarrow (f \sqsubseteq g \wedge g \sqsubseteq f)$

Definition 4 *General refinement on tuples.*

$$(f_1, g_1) \sqsubseteq (f_2, g_2) \overset{\text{def}}{=} (f_1 \sqsubseteq f_2) \wedge (g_1 \sqsubseteq g_2).$$

Note that the preceding definition gives the standard definition of refinement for tuples [4], which is applicable to all tuples.

In contrast, environment and store functions enjoy special properties with respect to refinement that are not shared by other functions. These properties are important for proving the correctness of transformations, because they enable proofs in cases that could not be proved from the general definition of refinement alone. To emphasize the difference between general refinement for functions and refinement for the domain $\mathcal{M}$, we introduce a the symbol, $\sqsubseteq^{\mathcal{M}}$ to denote the refinement relation as it manifests itself in $\mathcal{M}$. The semantics of $\sqsubseteq^{\mathcal{M}}$ is given below.

For *states*, definition 4 should be weakened from an equality to an implication as stated in Axiom 1.

Axiom 1 $(\varepsilon_1 \sqsubseteq \varepsilon_2) \wedge (s_1 \sqsubseteq s_2) \Rightarrow (\varepsilon_1, s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, s_2)$

Axiom 2 *Refinement within $\mathcal{M}$*

$$\begin{aligned}(\varepsilon_1, s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, s_2) \overset{\text{def}}{=} \quad (\forall x \in id, \quad & ((\varepsilon_2(x) = \perp) \Rightarrow (\varepsilon_1(x) = \perp)) \wedge \\ & (s_1(\varepsilon_1(x)) \sqsubseteq s_2(\varepsilon_2(x)))).\end{aligned}$$

This axiom states that the address that a variable gets mapped to in the store is not important with respect to our abstract notion of state. Note that $((\varepsilon_2(x) = \perp) \Rightarrow (\varepsilon_1(x) = \perp))$ is critical for most imperative languages. This expression distinguishes the case where a variable is undefined because it has not be declared from the case where the variable is undefined because it has not been assigned a value.

For some languages, program commands can be cleanly partitioned to those that alter the environment and those that change the store. For this reason the following two instantiations of Axiom 2 are of special interest.

- $$\begin{aligned}(\varepsilon_1, s) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, s) = \quad (\forall x \in id, \quad & ((\varepsilon_2(x) = \perp) \Rightarrow (\varepsilon_1(x) = \perp)) \wedge \\ & (s(\varepsilon_1(x)) \sqsubseteq s(\varepsilon_2(x)))).\end{aligned}$$
- $(\varepsilon, s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon, s_2) = (\forall x \in id, (s_1(\varepsilon(x)) \sqsubseteq s_2(\varepsilon(x))))$

Axiom 3 *For a given* α. $(\neg \exists x \in id, \varepsilon(x) = \alpha) \Rightarrow (\varepsilon, s) \sqsubseteq^{\mathcal{M}} (\varepsilon, [\alpha \mapsto \perp]s)$.

This axiom states that the value of any location in the store that does not have a corresponding identifier is irrelevant. This axiom is for convenience more than anything else, for it allows the denotational semantics to omit "storage cleanup" operations between scope boundaries.

Lemma 1 $((\varepsilon, s) \sqsubseteq^{\mathcal{M}} (\varepsilon, [\alpha \mapsto \perp]s)) \Rightarrow ((\varepsilon, s) \equiv (\varepsilon, [\alpha \mapsto \perp]s))$.

Axiom 4 $x \neq y \Rightarrow (\varepsilon(x) = \perp) \vee (\varepsilon(x) \neq \varepsilon(y))$. *We do not permit aliasing.*

Axiom 5 $\exists \alpha \in$ *storage_locations*, $\forall \alpha' \in$ *storage_locations*, $(|\alpha| < |\alpha'|) \Rightarrow \varepsilon(\alpha') = \perp$. The number of storage locations in an environment is finite. This is a necessary restriction to enable a constructive realization of the function *new* which is defined in Section *4.1.*

Lemma 2 *(recursive definition of* $\sqsubseteq^{\mathcal{M}}$*)*

$$\begin{aligned}(\forall x \in id, \quad & \varepsilon_1(x) \neq \perp \wedge \varepsilon_2(x) \neq \perp \wedge \\ & (s_1(\varepsilon_1(x)) \sqsubseteq s_2(\varepsilon_2(x))) \wedge \\ & ((\varepsilon_1, [\varepsilon_1(x) \mapsto \perp]s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, [\varepsilon_2(x) \mapsto \perp]s_2)) \quad \Rightarrow (\varepsilon_1, s_1) \sqsubseteq^{\mathcal{M}} (\varepsilon_2, s_2)))\end{aligned}$$

4 Refinement on *schemas*

We can extend the above definition of refinement of *states* to define refinement for transformation schemas. Given a transformation schema t (a syntactic object), we use the symbol $\widehat{t}$ to denote the expression in the mathematical domain (i.e., the semantic object) that corresponds to t according to our extended denotational semantics.

Definition 5 *(general refinement – unconditional correctness)*

$$t_1 \sqsubseteq t_2 \overset{\text{def}}{=} \forall state_i \in \mathcal{M},\ \widehat{t}_1(state_i) \sqsubseteq^{\mathcal{M}} \widehat{t}_2(state_i)$$

This is the most general form of refinement on schemas. Also note that what we have just extended our definition of refinement from a semantic domain into a syntactic domain. From this point on, it makes sense to talk about "refinement of schemas".

4.1 Semantic Properties

In Section 3.3 we discussed (semantic) properties of $\mathcal{M}$. Additional semantic predicates and functions are often useful for showing that one schema is a refinement of another. A common predicate is *uniqueness* (for variables) and a common function is *new* (for addresses). These are defined as follows:

Definition 6 $unique(x, (\varepsilon, s)) \overset{\text{def}}{=} (\varepsilon(x) = \perp)$

Definition 7 $new \overset{def}{=} (\lambda\ \varepsilon.\ \alpha)$ such that $(\neg\exists x \in id,\ \varepsilon(x) = \alpha)$ holds.

Note that the latter definition places a requirement on the storage allocation and management strategy that it be able to generate an α with respect to a specific ε in accordance with the definition of *new*.

5 Correctness Proofs

In this section we prove the correctness of a simple TAMPR transformation. We stress that this transformation is simple and is used for illustrative purposes only. Nevertheless in spite of its simplicity, $\mathcal{T}_1$ is interesting because in other semantic systems this transformation is give as an axiom. This is in contrast to our semantic framework, where we can prove the correctness of $\mathcal{T}_1$.

To date, we have proved the correctness of several practical transformations, having substantially greater complexity, using this methodology [6]. For a partial grammar of Poly and its denotational semantics see [6]. For more information on TAMPR and the syntax of transformations see [1].

5.1 A simple transformation

- Declaration Order Interchange. Interchanging the order in which two variables are declared in a Poly program is a refinement.

$$\mathcal{T}_3 \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \textit{<spec stmt>}\{< \textit{ standard type} >_1 \; x, y\} \\ \Rightarrow \\ \textit{<spec stmt>}\{< \textit{ standard type} >_1 \; y, x\} \end{array} \right.$$

Theorem 1 *(declarations are commutative).*

$$\begin{array}{l} < \textit{ spec stmt} > \{< \textit{ standard type} >_1 x, y\} \\ \sqsubseteq \\ < \textit{ spec stmt} > \{< \textit{ standard type} >_1 y, x\} \end{array}$$

Proof: If we omit some technical details, then the denotational definitions will map the schema *<spec stmt>* $\{$*<standard type>*$_1$ $x, y\}$ to the semantic function

$$t_{\,pattern} = \left\{ \begin{array}{l} \lambda\,(\varepsilon, s).\; ([x \mapsto \alpha_1][y \mapsto \alpha_2]\varepsilon, s) \\ \\ \text{where } new([y \mapsto \alpha_2]\varepsilon) = \alpha_1 \wedge new(\varepsilon) = \alpha_2 \end{array} \right.$$

Similarly, the schema *<spec stmt* > $\{$*declare* $y, x\}$ will get mapped to

$$t_{\,replacement} = \left\{ \begin{array}{l} \lambda\,(\varepsilon, s).\; ([y \mapsto \alpha'_1][x \mapsto \alpha'_2]\varepsilon, s) \\ \\ \text{where } new([x \mapsto \alpha'_2]\varepsilon) = \alpha'_1 \wedge new(\varepsilon) = \alpha'_2 \end{array} \right.$$

Let (ε_i, s_i) denote a particular but arbitrarily chosen state from the domain of states. From this we get

$$\hat{t}_{\,pattern}(\varepsilon_i, s_i) = \left\{ \begin{array}{l} ([x \mapsto \alpha_1][y \mapsto \alpha_2]\varepsilon_i, s_i) \\ \\ \text{where } new([y \mapsto \alpha_2]\varepsilon_i) = \alpha_1 \wedge new(\varepsilon_i) = \alpha_2 \end{array} \right.$$

and

$$\hat{t}_{\mathit{replacement}}(\varepsilon_i, s_i) = \begin{cases} ([y \mapsto \alpha_1'][x \mapsto \alpha_2']\varepsilon_i, s_i) \\ \text{where } \mathit{new}([x \mapsto \alpha_2']\varepsilon_i) = \alpha_1' \wedge \mathit{new}(\varepsilon_i) = \alpha_2' \end{cases}$$

Axioms 2 and 3 together with Definition 7 (the definition of *new*) and the fact that $\varepsilon_i \sqsubseteq \varepsilon_i$ gives us

$$([x \mapsto \alpha_1][y \mapsto \alpha_2]\varepsilon_i, s_i) \sqsubseteq^m ([y \mapsto \alpha_1'][x \mapsto \alpha_2']\varepsilon_i, s_i).$$

which in turn allows us to conclude that

$$\forall(\varepsilon_i, s_i) \in \text{states}, \ \hat{t}_{\mathit{pattern}}(\varepsilon_i, s_i) \sqsubseteq^m \hat{t}_{\mathit{replacement}}(\varepsilon_i, s_i)$$

which leads to

$$t_{\mathit{pattern}} \sqsubseteq t_{\mathit{replacement}}$$

Q.E.D.

6 Conclusions and Future Work

In this paper we identified a deficiency of the traditional denotational semantic paradigm with respect to schema variables. Since schema variables occur frequently in TAMPR transformations, this motivated our work in extending the denotational semantic paradigm with *delta-functions. Delta-functions* can have a straightforward semantics, however languages and contexts within transformation sequences can exist where the semantics of *delta-functions* can be quite complex. For these reasons, an automated procedure for determining the semantics of *delta-functions* with respect to a given grammar and its denotational semantics is being developed.

In denotational semantics, a computational state space $\mathcal{M}$ is constructed. This state space generally consists of an environment and a store function. The execution semantics of programs (syntactic objects) are then defined in terms of $\mathcal{M}$. The environment and store functions when considered together capture the notion of the *abstract state* of a computation. Since information about the abstract state is spread out over two functions, dependencies are introduced. These dependencies must be factored out in order to allow reasoning about the abstract state to proceed. The axioms, definitions, and lemmas in Section 3.3 permit reasoning with respect to the state space $\mathcal{M}$.

In conclusion, we believe that a properly extended denotational semantic framework together with a correspondingly modified definition of refinement provide an environment that is well suited for proving the correctness of refinement transformations such as those used by TAMPR.

References

[1] James M. Boyle. Abstract programming and program transformation – an approach to reusing programs. In T. J. Biggerstaff and A. Perlis, editors, *Software Reusability*, pages 361-413. Addison-Wesley, 1989.

[2] James M. Boyle and Manohar N. Muralidharan. Program Reusability through program transformation. *IEEE Transactions on Software Engineering*, (5):574-588, September 1984

[3] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, Inc. San Diego, California, 1973.

[4] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill Inc., New York, New York, 1974.

[5] David A. Schmidt. *Denotational Semantics.* Wm. C. Brown Publishers, Dubuque, Iowa, 1986.

[6] Victor L. Winter. *Proving the Correctness of Program Transformations.* Ph.D. dissertation, University of New Mexico, 1994.

7 Biography

Victor L. Winter received his Ph.D. from the University of New Mexico in 1994. His dissertation research focused on proving the correctness of program transformations. Currently, Dr. Winter is a member of a newly constructed High Integrity Software (HIS) group at Sandia National Laboratories. His research interests include trusted software, formal semantic models, theory of computation, automated reasoning and robotics. Dr. Winter can be reached by phone in the United States at (505) 284-2696 or by email at vlwinte@sandia.gov.

James M. Boyle received his Ph.D. from Northwestern University in 1970. He has been active in the field of program transformation since writing his

dissertation on the initial design of the TAMPR transformation system. He is a member of the Mathematics and Computer Science Division at Argonne National Laboratory. Dr. Boyle's other research interests include trusted software, parallel processing, and automated reasoning. He is coauthor of the books *Automated Reasoning—Introduction and Applications* and *Portable Programs for Parallel Processors.* He can be reached at +1 708-252-7227 or by email at boyle@mcs.anl.gov