On the Semantics of Refinement Calculi

Hongseok Yang[1] and Uday S. Reddy[2]

[1] University of Illinois at Urbana-Champaign, `hyang@cs.uiuc.edu`
[2] University of Birmingham, `u-reddy@cs.uiuc.edu`

Abstract. Refinement calculi for imperative programs provide an integrated framework for programs and specifications and allow one to develop programs from specifications in a systematic fashion. The semantics of these calculi has traditionally been defined in terms of predicate transformers and poses several challenges in defining a state transformer semantics in the denotational style. We define a novel semantics in terms of sets of state transformers and prove it to be isomorphic to positively multiplicative predicate transformers. This semantics disagrees with the traditional semantics in some places and the consequences of the disagreement are analyzed.

1 Introduction

Two dominant semantic views of imperative programs are in terms of *state transformers*, initiated by McCarthy [17], Scott and Strachey [30], and *predicate transformers*, initiated by Dijkstra [11]. State transformers give a clear correspondence with the operational semantics, where commands do, after all, transform the state of a machine. The predicate transformer view, on the other hand, has been argued to be suitable for showing that programs achieve certain goals, i.e., to questions of correctness. A definitive relationship between the two views was established by Plotkin [28], following other work [9, 31, 4], where it is shown that Dijkstra's predicate transformers are isomorphic to nondeterministic state transformers defined using the Smyth powerdomain. The isomorphism establishes a tight connection between the predicate transformer view and operational behavior, which is not obvious otherwise. It is also of important conceptual value as it allows the two semantic views to coexist side by side. The ideas expressed using either view can be converted into the other, and there is no conflict between the two views.

In more recent work, predicate transformers have been put to new uses. Refinement calculi, developed by Hehner [16], Back [3, 5], Morris [24], Morgan [19] and Nelson [27], extend Dijkstra's programming language with "specification statements." Typically written as $[\varphi, \psi]$, a specification statement stands for some statement that is yet to be developed but which is expected to satisfy the specification $\langle \varphi, \psi \rangle$, i.e., transform states satisfying φ to states satisfying ψ. Such specification statements serve as space fillers in the initial stages of program development, and they are refined to actual program statements in later stages.

The semantics of such extended languages for program refinement has only been defined in terms of predicate transformers. No semantics is known in terms of state transformers. Moreover, the predicate transformers involved in the semantics go beyond Dijkstra's predicate transformers. (They do not satisfy Dijkstra's healthiness conditions such as continuity.) Since, by Plotkin's result, state transformers are isomorphic to Dijkstra's predicate transformers, we already know that there are no conventional state transformers corresponding to these new predicate transformers. This leaves the operational interpretation of the refinement calculi very much in the dark.

In this paper, we develop a semantic interpretation of refinement calculi in terms of state transformers. The basic idea is that statements in refinement calculi are to be interpreted as *sets of state transformers* that satisfy the specifications embedded in the statements. In Denney's terminology [10], this interpretation represents "under-determinism" as opposed to "nondeterminism." We also need a notion of *guarded* state transformers, similar to the idea of partial functions, which are defined only for some subset of the set of all states. We are able to show that suitable sets of guarded state transformers are isomorphic to positively multiplicative predicate transformers. This parallels Plotkin's original isomorphism result for Dijkstra's predicate transformers.

All the constructs of refinement calculi can be interpreted using sets of guarded state transformers. This gives a natural semantics of specification statements as collections of program statements that meet those specifications. However, this semantics does not match up exactly with the traditional predicate transformer semantics of refinement calculi. The predicate transformers used in the latter are not in general positively multiplicative, a property used in our isomorphism result.

We examine the consequences of this mismatch, and show that there are refinement laws that are intuitively *unreasonable* but hold in the traditional semantics though not in ours. The conclusion is that a better semantics of refinement calculus is obtained by restricting to positively multiplicative predicate transformers which have a natural equivalence with state transformer sets.

We believe these results go a long way towards demystifying refinement calculi. The absence of an operational reading for the constructs of refinement calculi has contributed to some of the mysteries surrounding the traditional treatment of the subject. The predicate transformer semantics implies that the theory of these calculi is internally consistent. However, the mysteries point to problems in *interpreting* the theory. Our contribution is in clarifying the interpretation which, we hope, might lead to a wider appreciation of the theory itself.

Related Work The early work on relating state transformers and predicate transformers is mentioned in Plotkin [28]. In later work, Apt and Plotkin [1, 2] extended [28] to countable nondeterminism, and Smyth [29] to non-flat state spaces. Bonsangue and Kok [7] found correspondences for safety and liveness predicate transformers and, in [8], for Nelson's predicate transformers [27]. We should remark that all this work is for *programming* languages, not for specification languages used in refinement. However, there are close relationships between the results needed in this paper and the earlier results, especially those of Apt and Plotkin [1, 2]. Morgan [20], Gardiner [12] and Naumann [25] also considered multiplicative predicate transformers and the correspondence with relations (which may be seen as infinitely nondeterministic state transformers). Gardiner et al. [13] and Naumann [25] used this correspondence to lift type structure to specification languages.

After the present work was completed, we were made aware of Ewen Denney's dissertation [10], which echoes very similar ideas to our work. In particular, it interprets specifications via under-determinism. On the other hand, Denney focuses on functional programming languages whereas we are looking at imperative programming and the correspondence between state transformer and predicate transformer interpretations. We also highlight the interaction between nondeterminism and under-determinism (cf. Sec. 3.2).

Overview In Sec. 2, we give a brief summary of the refinement calculus we use in this paper and define its predicate transformer semantics. Section 3 introduces the state transformer concepts that are used in our semantics and show their isomorphism with positively multiplicative predicate transformers. The state transformer semantics of the calculus is defined in Sec. 4. Finally, in Sec. 5, we discuss problems and issues that lie outside our isomorphism.

2 Refinement Calculus

Refinement calculi are obtained by extending a programming language with additional notations for expressing specifications. Program statements and specification statements are then freely intermixed. A refinement relation $\sqsubseteq$ is defined between statements in the extended language. A collection of refinement laws, axiomatizing the refinement relation, is devised, by which a specification can be refined to an executable program in a series of steps. The subject is extensively covered in the two text books [6, 21] as well as the collection [23].

Here, we use a variant of the Morgan-Gardiner refinement calculus [19, 22] as the basis of our study. For simplicity, we treat basic imperative programs over a fixed collection of program variables. However, we will allow locally bound constant identifiers in specifications.

Assume a finite set $\mathcal{V}$ of typed *variable identifiers*, and a countably infinite set $\mathcal{I}$ of *constant identifiers*, disjoint from $\mathcal{V}$. Using these we form a collection of *expressions*, *assertions* and *atomic commands*, whose structure we unspecified except to note that both variable identifiers and constant identifiers can occur in them. The collection of statements in the Dijkstra's programming language is given by the context-free syntax:

$$\begin{array}{rcl} C & ::= & A \mid \mathbf{skip} \mid \mathbf{abort} \mid C_1; C_2 \mid \mathbf{if}\, G\, \mathbf{fi} \mid \mathbf{do}\, G\, \mathbf{od} \\ G & ::= & \epsilon \mid E \to C \mid G_1 \,[\!]\, G_2 \end{array}$$

where A and E range over atomic commands and boolean expressions respectively, and G stands for guarded commands.

To obtain a refinement calculus, we extend the collection of statements by two clauses:

$$C \quad ::= \quad \ldots \mid v_1, \ldots, v_n \colon [\varphi, \psi] \mid \textbf{con}\ i \colon \tau = E\ \textbf{in}\ C$$

The statement $v_1, \ldots, v_n \colon [\varphi, \psi]$ is called a *specification statement* or a *prescription*. The intended meaning is that it stands for some arbitrary program statement that satisfies the specification $\langle \varphi, \psi \rangle$, i.e., transforms states satisfying φ to those satisfying ψ, by modifying at most the variables $v_1, \ldots, v_n$. The variables $v_1, \ldots, v_n$ are said to constitute the *frame* of the statement. When the frame includes all the variables in $\mathcal{V}$, we use the abbreviation $[\varphi, \psi]$ for $\mathcal{V} \colon [\varphi, \psi]$. For example, the statement $r \colon [n > 0,\ (r-1)^2 < n \leq r^2]$ specifies the action of assigning to r the integer square root of n.

The construct $\textbf{con}\ i \colon \tau = E\ \textbf{in}\ C$ specifies an action that satisfies the specification C when i is given the value of E in the current state. For example,

$$\textbf{con}\ k \colon \mathsf{int} = |n|\ \textbf{in}\ n \colon [\text{true},\ |n| = k + 1]$$

specifies that n must be modified so as to increase its absolute value by 1. This is a variant of the constant-introduction construct of Morgan and Gardiner [22] where we require the initial value to be explicitly declared. We consider the Morgan-Gardiner construct in Section 5 as it raises interesting semantic issues.

Predicate Transformer Semantics

A predicate transformer interpretation for the refinement calculus has been defined by Morgan and Gardiner [19, 14] (as well as other authors on the subject). Here, we use a semantic version of this interpretation by taking predicates as sets of states. Our treatment closely follows Plotkin [28]. See also [26, 8, 7] for similar presentations.

Let Σ be the set of states for the variables in $\mathcal{V}$. For technical reasons, we assume that Σ is countable. A *predicate* is a subset $a \subseteq \Sigma$. A *predicate transformer* is a monotone function $t : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. Predicate transformers are partially ordered by the pointwise ordering:[1] $t_1 \sqsubseteq t_2 \iff \forall a' \in \mathcal{P}(\Sigma).\, t_1(a') \subseteq t_2(a')$. The poset of predicate transformers is denoted PT.

A predicate transformer is said to be *completely multiplicative* if for any family $F' \subseteq \mathcal{P}(\Sigma)$, $t(\bigcap F') = \bigcap t(F')$. We call it *positively multiplicative* if this property holds for all nonempty families $F' \subseteq \mathcal{P}(\Sigma)$. Define the poset:

$$\mathsf{PTM}^+ = \{t : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma) \mid t \text{ is positively multiplicative }\}, \quad \text{ordered pointwise}$$

If t is any predicate transformer and $x \in t(\Sigma)$, define

$$L_t(x) = \bigcap \{a' \mid x \in t(a')\}.$$

[1] We often use primed variable names (such as a') as arguments for predicate transformers to denote the fact that they are sets of post-states.

Program Operations

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{Skip} & : & \mathcal{D} & \mathsf{Do} & : & \mathsf{Bool} \times \mathcal{D} \to \mathcal{D} \\
\mathsf{Conv} & : & \mathsf{d\text{-}ST} \to \mathcal{D} & \mathsf{Empty} & : & \mathsf{Bool} \times \mathcal{D} \\
\mathsf{Comp} & : & \mathcal{D}^2 \to \mathcal{D} & \mathsf{Guard} & : & \mathsf{Bool} \times \mathcal{D} \to \mathsf{Bool} \times \mathcal{D} \\
\mathsf{Cond} & : & \mathsf{Bool} \times \mathcal{D} \to \mathcal{D} & \mathsf{Bar} & : & (\mathsf{Bool} \times \mathcal{D})^2 \to \mathsf{Bool} \times \mathcal{D}
\end{array}
$$

Specification Operations

$$
\begin{array}{rcl}
\mathsf{Pres}_R & : & \mathcal{P}(\Sigma)^2 \to \mathcal{D} \\
\mathsf{Con}_A & : & (\Sigma \to A) \times (A \to \mathcal{D}) \to \mathcal{D}
\end{array}
$$

where $R \subseteq \Sigma \times \Sigma$ is an equivalence relation
A is a countable set
$\mathsf{d\text{-}ST} = \Sigma \to \Sigma$, ordered discretely
$\mathsf{Bool} = \Sigma \to \{\mathsf{tt}, \mathsf{ff}\}$, ordered discretely

Table 1. Signature of the Semantic Algebra

Lemma 1. *A predicate transformer t is positively multiplicative iff, for all $x \in t(\Sigma)$, $x \in t(L_t(x))$. In this case, $L_t(x)$ is the least a' such that $x \in t(a')$.*

Lemma 2. *For any predicate transformer t, there is a least positively multiplicative predicate transformer t^* above t, given by $t^*(a') = \{x \in t(\Sigma) \mid L_t(x) \subseteq a'\}$.*

Note that $L_{t^*}(x)$ and $L_t(x)$ are the same. By forcing $t^*(L_{t^*}(x))$ to include x, we obtain a positively multiplicative predicate transformer. We call t^* the *positively multiplicative closure* of t.

Lemma 3. PTM^+ *is a complete lattice with least upper bounds given by $\bigsqcup_{i \in I} t_i = (\lambda a.\, \bigcup_{i \in I} t_i(a))^*$. The least element $\bot_{\mathsf{PTM}^+}$ is $\lambda a'.\, \emptyset$.*

Note that PTM^+ is not a complete sublattice of PT because the least upper bounds in PTM^+ are different from those in PT.

We work in the category of complete lattices with monotone functions as morphisms. By Tarski's fixed point theorem, every monotone function $f : L \to L$ has a least fixed point, given by $\mathbf{fix}(f) = \bigsqcap\{t \in L \mid f(t) \sqsubseteq t\}$.

To define the semantics of the refinement calculus, we use an algebraic approach as in [28]. Table 1 shows the signature of a semantic algebra $\mathcal{D}$, where all the operations are meant to be monotone maps. For the predicate transformer semantics $\mathcal{D} = \mathsf{PTM}^+$, but the same signature will be used for the state transformer semantics to be introduced later. The program operations are as in [28] and we recall their definitions in Table 2. The only difference from [28] is that we are using *positively multiplicative* predicate transformers instead of continuous ones.

For interpreting specification constructs, we define two new operators:

1. **Prescription:** The operation Pres_R captures the semantics of Morgan's specification statement $\boldsymbol{v}\colon[\varphi, \psi]$. The idea that only variables $\boldsymbol{v}$ can be modified can be represented by an equivalence relation $R \subseteq \Sigma \times \Sigma$, which equates

$$
\begin{array}{rcl}
\mathsf{Skip} & = & \lambda a'.a' \\
\mathsf{Conv}(m) & = & \lambda a'.m^{-1}(a') \\
\mathsf{Comp}(t_1,t_2) & = & t_1 \circ t_2 \\
\mathsf{Cond}(p,t) & = & \lambda a'.p^+ \cap t(a') \\
\mathsf{Do}(p,t) & = & \mathbf{fix}_{\mathsf{PTM}^+}(\lambda t'.\,\lambda a'.\,(p^- \cap a') \cup (p^+ \cap (t \circ t')(a'))) \\
\mathsf{Empty} & = & (\lambda x.\,\mathsf{ff}, \lambda a'.\,\emptyset) \\
\mathsf{Guard}(p,t) & = & (p,t) \\
\mathsf{Bar}((p_1,t_1),(p_2,t_2)) & = & (p_1 \vee p_2,\ \ \lambda a'.\,(p_1^+ \cup p_2^+) \cap (p_1^- \cup t_1(a')) \cap (p_2^- \cup t_2(a')))
\end{array}
$$

where $p^+ = p^{-1}(\mathsf{tt})$, $p^- = p^{-1}(\mathsf{ff})$ and $(p \vee q)(x) = p(x) \vee q(x)$

Table 2. Program Operations for PTM^+

states that possibly differ only in variables $\boldsymbol{v}$. We write $[x]_R$ for the equivalence class of x under R. Define a family of operations indexed by equivalence relations $R \subseteq \Sigma \times \Sigma$:

$$
\begin{array}{rcl}
\mathsf{Pres}_R & : & \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \to \mathsf{PTM}^+ \\
\mathsf{Pres}_R(b,b') & = & \lambda a'.\{x \in b \mid b' \cap [x]_R \subseteq a'\}
\end{array}
$$

2. **Constant introduction:** The family of operations $\mathsf{Con}_A : (\Sigma \to A) \times (A \to \mathsf{PTM}^+) \to \mathsf{PTM}^+$ captures the introduction of constant identifiers of type A.

$$
\mathsf{Con}_A(e,f) \quad = \quad \lambda a'.\,\{x \in \Sigma \mid x \in f(e(x))(a')\}
$$

We note that two of Dijkstra's healthiness conditions are violated by these predicate transformers. When R relates all pairs of states,

– $\mathsf{Pres}_R(b,\emptyset)$ is not strict (unless $b = \emptyset$).
– $\mathsf{Pres}_R(b,\Sigma)$ is not continuous. Note that Σ can be expressed as a lub $\bigcup_i a'_i$ for an increasing sequence of finite sets a'_i. $\mathsf{Pres}_R(b,\Sigma)(\Sigma) = b$, but for every finite a', $\mathsf{Pres}_R(b,\Sigma)(a') = \emptyset$.

Lemma 4. *All the operators above are well-defined and monotone.*

Note that the operators are not necessarily continuous. For example, Comp is not continuous.

The semantics of the refinement language is as follows. Since commands have free identifiers (for constants), we use *environments* for giving values to the identifiers [15, 18]. *Env* denotes the set of environments. The semantic functions are defined in Table 3, parameterized by a semantic algebra $\mathcal{D}$. By instantiating the definition by $\mathcal{D} = \mathsf{PTM}^+$, we obtain the predicate transformer semantics. We denote these semantic functions by $\mathcal{C}_{\mathsf{M}}$ and $\mathcal{G}_{\mathsf{M}}$ for commands and guarded commands respectively.

The fact that all the semantic algebra operations are monotonic implies that program contexts preserve refinement, i.e., $C \sqsubseteq C'$ implies $P\{C\} \sqsubseteq P\{C'\}$ for any program context $P\{\ \}$. This result is essential for program refinement because it allows one to refine whole programs by refining their components one at a time.

$$
\begin{array}{rcl}
\mathcal{P} & : & \text{Predicates} \to \text{Env} \to \mathcal{P}(\Sigma) \\
\mathcal{E} & : & \text{Expressions} \to \text{Env} \to (\Sigma \to \text{Value}) \\
\mathcal{A} & : & \text{Atomic commands} \to \text{Env} \to \mathsf{d\text{-}ST} \\
\mathcal{C} & : & \text{Statements} \to \text{Env} \to \mathcal{D} \\
\mathcal{G} & : & \text{Guarded Commands} \to \text{Env} \to \mathsf{Bool} \times \mathcal{D}
\end{array}
$$

Interpretation of Commands

$$
\begin{array}{rcl}
\mathcal{C}[\![A]\!]e & = & \mathsf{Conv}(\mathcal{A}[\![A]\!]e) \\
\mathcal{C}[\![\mathbf{skip}]\!]e & = & \mathsf{Skip} \\
\mathcal{C}[\![\mathbf{abort}]\!]e & = & \bot \\
\mathcal{C}[\![C_1; C_2]\!]e & = & \mathsf{Comp}(\mathcal{C}[\![C_1]\!]e, \mathcal{C}[\![C_2]\!]e) \\
\mathcal{C}[\![\mathbf{if}\, G\, \mathbf{fi}]\!]e & = & \mathsf{Cond}(\mathcal{G}[\![G]\!]e) \\
\mathcal{C}[\![\mathbf{do}\, G\, \mathbf{od}]\!]e & = & \mathsf{Do}(\mathcal{G}[\![G]\!]e) \\
\mathcal{C}[\![\boldsymbol{v}\colon [\varphi, \psi]]\!]e & = & \mathsf{Pres}_{R(\boldsymbol{v})}(\mathcal{P}[\![\varphi]\!]e, \mathcal{P}[\![\psi]\!]e) \\
\mathcal{C}[\![\mathbf{con}\ i\colon \tau = E\ \mathbf{in}\ C]\!]e & = & \mathsf{Con}_{[\![\tau]\!]}(\mathcal{E}[\![E]\!]e,\ \lambda k \in [\![\tau]\!].\, \mathcal{C}[\![C]\!]e[i \mapsto k])
\end{array}
$$

where $R(\boldsymbol{v})$ denotes the equivalence relation on states given by

$$x[R(\boldsymbol{v})]x' \iff \forall v \notin \boldsymbol{v}.\, x(v) = x'(v)$$

Interpretation of Guarded Commands

$$
\begin{array}{rcl}
\mathcal{G}[\![\epsilon]\!]e & = & \mathsf{Empty} \\
\mathcal{G}[\![E \to C]\!]e & = & \mathsf{Guard}(\mathcal{E}[\![E]\!]e, \mathcal{C}[\![C]\!]e) \\
\mathcal{G}[\![G_1 \mathbin{[\,]} G_2]\!]e & = & \mathsf{Bar}(\mathcal{G}[\![G_1]\!]e, \mathcal{G}[\![G_2]\!]e)
\end{array}
$$

Table 3. Semantics of Refinement Calculus

3 State Transformers and Predicate Transformers

Consider the set of states Σ. The set obtained by adding an element $\bot$ (for the undefined state) is denoted $\Sigma_\bot$. We make $\Sigma_\bot$ into a poset by defining the partial order $x \sqsubseteq y \iff x = \bot \vee x = y$. The Smyth powerdomain of $\Sigma_\bot$ is defined as follows:

$$
\begin{array}{rcl}
\mathcal{P}_S(\Sigma_\bot) & = & \text{the set of nonempty finite subsets of } \Sigma \text{ and the infinite set } \Sigma_\bot, \\
 & & \text{ordered by superset order.}
\end{array}
$$

So, the least element of $\mathcal{P}_S(\Sigma_\bot)$ is $\Sigma_\bot$.

The domain of state transformers is

$$\mathsf{ST} \ = \ (\Sigma \to \mathcal{P}_S(\Sigma_\bot)), \quad \text{ordered pointwise}$$

The intuition is as follows. If $c \sqsubseteq c'$, then

1. c' terminates (possibly) more often than c, and
2. c' is (possibly) more deterministic than c.

We say that c' is "better" than c. Say that a state transformer c *satisfies* a specification $\langle a,\, a'\rangle$, written $c \models \langle a,\, a'\rangle$, if running c from a state in a gives a state in a'. Formally,

$$c \models \langle a,\, a'\rangle \iff \forall x \in a.\, c(x) \subseteq a'$$

Then, it is easy to see that $c \models \langle a,\, a'\rangle \wedge c \sqsubseteq c' \Longrightarrow c' \models \langle a,\, a'\rangle$. That is, better state transformers continue to satisfy all the old specifications.

By regarding a predicate transformer t as a collection of specifications $\{\langle t(a'), a'\rangle\}_{a' \in \mathcal{P}(\Sigma)}$, we have a notion of satisfaction for predicate transformers:

$$c \models t \iff \forall x.\forall a'.\, x \in t(a') \Rightarrow c(x) \subseteq a'$$

The strongest predicate transformer satisfied by c is denoted Tc:

$$Tc(a') \;=\; \{x \in \Sigma \mid c(x) \subseteq a'\}$$

Tc is nothing but the "weakest precondition" operator of c. It satisfies the following properties:

- **continuity**: $Tc(\bigcup_i a'_i) \;=\; \bigcup_i Tc(a'_i)$ for every ascending chain $\{a'_i\}_i$. The reason is that $Tc(\bigcup_i a'_i)$ includes all and only those initial states x whose results $c(x)$ are included in finite subsets of $\bigcup_i a'_i$.
- **positive multiplicativity**: $Tc(\bigcap_{i \in \mathcal{I}} a_i) \;=\; \bigcap_{i \in \mathcal{I}} Tc(a_i)$ for nonempty $\mathcal{I}$. The reason is that x is in $\bigcap_i Tc(a_i)$ only when for all i, $c(x)$ is a subset of a_i. This is equivalent to $x \in Tc(\bigcap_i a_i)$.
- **strictness**: $Tc(\emptyset) \;=\; \emptyset$. The reason is that $c(x)$ is always nonempty. So, $c(x) \subseteq \emptyset$ is impossible.

It is possible to recover c from Tc. For any predicate transformer t that satisfies these properties, let[2] $T^{-1}(t) \;=\; \lambda x.\, x \in t(\Sigma) \leadsto L_t(x); \Sigma_\perp$. It can be verified that $T^{-1}(t)$ is a state transformer.

Theorem 1 (Plotkin). *There is an order-isomorphism between* ST *and the poset of predicate transformers that are continuous, positively multiplicative and strict.*

Recall that the predicate transformers used in refinement calculus do not generally satisfy the properties mentioned above. We examine a series of state transformer concepts that correspond to wider classes of predicate transformers.

3.1 Guarded State Transformers

The idea of a guarded state transformer is similar to that of a partial function. A guarded state transformer is meant to be run only starting from certain initial states and not from others. Formally, a guarded state transformer is a pair

$$(p \subseteq \Sigma, c : p \to \mathcal{P}_S(\Sigma_\perp))$$

[2] We use the notation $p \leadsto x; y$ to mean "if p then x else y."

Note that c is only defined for states in p (which is called the "domain of definition") and undefined for others. This notion of "undefined" is different from nontermination. (The state transformer c might still map states in p to $\Sigma_\perp$.) A guarded state transformer is simply never meant to be used outside its domain of definition. The notion of satisfaction is:

$$(p, c) \models \langle a, a' \rangle \iff \forall x \in p.\, x \in a \Rightarrow c(x) \subseteq a'$$

So, we only worry about initial states within the domain of definition. As a result, the completely undefined state transformer satisfies every specification. In particular, $(\emptyset, \lambda x.\Sigma_\perp) \models \langle \Sigma, \emptyset \rangle$. Recall that there are no ordinary state transformers satisfying $\langle \Sigma, \emptyset \rangle$. But this is not the case for guarded state transformers. In refinement calculus literature, this (sneaky!) way of satisfying specifications is termed "miraculous" [19].

We define a partial order on guarded state transformers by

$$(p, c) \sqsubseteq (p', c') \iff p \supseteq p' \;\wedge\; (\forall x \in p'.\, c'(x) \subseteq c(x)).$$

This partial order may seem surprising. We get a better state transformer by *reducing* the domain of definition. However, this order is consistent with the notion of satisfaction:

$$(p, c) \sqsubseteq (p', c') \wedge (p, c) \models \langle a, a' \rangle \implies (p', c') \models \langle a, a' \rangle$$

Just as partial functions $A \rightharpoonup B$ can be regarded as total functions of type $A \to B_\perp$ with an adjoined $\perp$ element denoting the undefined result, guarded state transformers can be regarded as state transformers with an adjoined *top* element in the codomain: $\Sigma \to \mathcal{P}_S^\top(\Sigma_\perp)$. Here $\mathcal{P}_S^\top(\Sigma_\perp)$ is like the Smyth powerdomain but also includes the empty set $\emptyset$ (which serves as the top element under the superset order). A guarded state transformer (p, c) is represented under this representation as the function $\lambda x \in \Sigma.\, x \in p \leadsto c(x); \emptyset$. Conversely, a state transformer $d : \Sigma \to \mathcal{P}_S^\top(\Sigma_\perp)$ represents the guarded state transformer $(\mathrm{dom}(d),\, d \restriction \mathrm{dom}(d))$ where $\mathrm{dom}(d) = \overline{d^{-1}(\emptyset)}$. From here on, we will identify guarded state transformers with this alternative representation, which is technically convenient to work with.

Define GST as the poset:

$$\mathsf{GST} \;=\; \Sigma \to \mathcal{P}_S^\top(\Sigma_\perp), \qquad \text{ordered pointwise}$$

For every guarded state transformer $d \in \mathsf{GST}$, we define a predicate transformer $Td : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ by

$$Td(a') \;=\; \{x \in \Sigma \mid d(x) \subseteq a'\}$$

This predicate transformer is *continuous* and *positively multiplicative* for the same reasons as before. But it is not strict. We have $Td(\emptyset) = \{x \in \Sigma \mid d(x) = \emptyset\} = \overline{\mathrm{dom}(d)}$, which has no reason to be empty. There is an inverse to T:

$$T^{-1}(t) \;=\; \lambda x.\, x \in t(\Sigma) \leadsto L_t(x); \Sigma_\perp$$

Theorem 2. *There is an order-isomorphism between* GST *and the poset of predicate transformers that are continuous and positively multiplicative.*

3.2 State Transformer Sets

Given a specification $\langle \varphi, \psi \rangle$, we have a collection S of state transformers satisfying it. Any such collection is *closed under union* in the following sense: if c is a state transformer and, for every $x \in \Sigma$, there are $c_1, \ldots, c_n \in S$ such that $c(x) \subseteq c_1(x) \cup \ldots \cup c_n(x)$, then $c \in S$. There is a simpler statement of this. Let the "lower bound" map $\widehat{S} : \Sigma \to \mathcal{P}(\Sigma_\perp)$ be the pointwise union $\widehat{S}(x) = \bigcup\{c(x) \mid c \in S\}$ (which is not a state transformer). Closure under union says that any state transformer c such that $c(x) \subseteq \widehat{S}(x)$ is in S. The same idea can also be used for guarded state transformers. In this case, the collection S must be nonempty. If S is a nonempty set of guarded state transformers, we define its *closure under union* by $S^\dagger = \{c \mid \forall x.\, c(x) \subseteq \widehat{S}(x)\}$.

Remark 1. The lower bound maps $\widehat{S}$ can be regarded as maps of type $\Sigma \to \mathcal{P}^\infty_S(\Sigma_\perp)$ where $\mathcal{P}^\infty_S$ is the infinitely nondeterministic Smyth powerdomain [2], whose elements include $\Sigma_\perp$ and *all* subsets of Σ. Sets of state transformers closed under union are one-to-one with such infinitely nondeterministic maps. This is in fact an order-isomorphism.

Let PGST denote the poset with nonempty sets of guarded state transformers that are closed under union, ordered by superset order. We call the elements of PGST *state transformer sets*.

Lemma 5. PGST *is a complete lattice with the least upper bounds given by intersection:* $\bigsqcup_i S_i = \bigcap_i S_i$.

For any $S \in \mathsf{PGST}$, we define a predicate transformer $TS : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ by

$$TS(a') = \bigcap_{c \in S} Tc(a') = \{x \in \Sigma \mid \forall c \in S.\, c(x) \subseteq a'\}$$

This predicate transformer is *positively multiplicative*: $TS(\bigcap_{i\in\mathcal{I}} a_i) = \bigcap_{i\in\mathcal{I}} TS(a_i)$ for nonempty $\mathcal{I}$. But, it is not continuous. We have $TS(\bigcup_i a_i) = \{x \in \Sigma \mid \forall c \in S.\, c(x) \subseteq \bigcup_i a_i\}$. If S is the set of all terminating state transformers, then $TS(\Sigma) = \Sigma$, but $TS(a) = \emptyset$ for every finite $a \subseteq \Sigma$.

Conversely, every positively multiplicative predicate transformer corresponds to a state transformer set: $T^{-1}(t) = \{c \mid c \models t\}$.

Theorem 3. *There is an order-isomorphism between* PGST *and* PTM^+.

4 State Transformer Semantics

We define a semantics of the refinement calculus using state transformer sets introduced in the previous section. We proceed as in Sec 2 by defining a semantic algebra over PGST. The operations for program statements are lifted versions of Plotkin's operations in [28]. They are shown in Table 4. The operations for specification statements are as follows:

$$
\begin{array}{rcl}
\mathsf{Skip} & = & \{\lambda x.\,\{x\}\}^{\dagger} \\
\mathsf{Conv}(m) & = & \{\lambda x.\,\{m(x)\}\}^{\dagger} \\
\mathsf{Comp}(S_1, S_2) & = & \{\lambda x.\,\mathsf{App}(c_2, c_1(x)) \mid c_1 \in S_1,\ c_2 \in S_2\}^{\dagger} \\
\mathsf{Cond}(p, S) & = & \{\lambda x.\,p(x) \rightsquigarrow c(x); \Sigma_{\perp} \mid c \in S\}^{\dagger} \\
\mathsf{Do}(p, S) & = & \mathbf{fix}_{\mathsf{PGST}}\,\lambda S'.\,\{\lambda x.\,p(x) \rightsquigarrow \mathsf{App}(c', c(x)); \{x\} \ \mid\ c \in S,\ c' \in S'\}^{\dagger} \\
\mathsf{Empty} & = & (\lambda x.\,\mathsf{ff},\ \{\lambda x.\,\Sigma_{\perp}\}^{\dagger}) \\
\mathsf{Guard}(p, S) & = & (p, S) \\
\mathsf{Bar}((p_1, S_1), & = & (p_1 \vee p_2, \\
(p_2, S_2)) & & \{\lambda x.\,p_1(x) \rightsquigarrow (p_2(x) \rightsquigarrow c_1(x) \cup c_2(x); c_1(x)); (p_2(x) \rightsquigarrow c_2(x); \Sigma_{\perp}) \\
 & & \qquad\qquad \mid c_1 \in S_1, c_2 \in S_2\}^{\dagger})
\end{array}
$$

where $\mathsf{App} : \mathsf{GST} \times \mathcal{P}_S^{\top}(\Sigma_{\perp}) \to \mathcal{P}_S^{\top}(\Sigma_{\perp})$ is defined by

$$\mathsf{App}(c, a') \quad = \quad (a' = \Sigma_{\perp}) \rightsquigarrow \Sigma_{\perp}; \textstyle\bigcup_{x' \in a'} c(x')$$

Table 4. Program Operations for PGST

1. **Prescription:** For any equivalence relation $R \subseteq \Sigma \times \Sigma$,

$$
\begin{array}{rcl}
\mathsf{Pres}_R & : & \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \to \mathsf{PGST} \\
\mathsf{Pres}_R(b, b') & = & \{c \in \mathsf{GST} \mid \forall x \in b.\, c(x) \subseteq b' \cap [x]_R\}
\end{array}
$$

   This defines our under-determinism semantics for specification statements. A specification statement stands for an arbitrary command that satisfies the specification.

2. **Constant introduction:**

$$
\begin{array}{rcl}
\mathsf{Con}_A & : & (\Sigma \to A) \times (A \to \mathsf{PGST}) \to \mathsf{PGST} \\
\mathsf{Con}_A(e, f) & = & \{c \in \mathsf{GST} \mid \forall x \in \Sigma.\, \exists c' \in f(e(x)).\, c(x) = c'(x)\}
\end{array}
$$

   This looks a bit intricate, but it is easier to see in terms of lower bound maps: $(\widehat{\mathsf{Con}_A(e, f)})(x) \quad = \quad \widehat{f(e(x))}(x)$.

Lemma 6. *The order-isomorphism $T : \mathsf{PGST} \cong \mathsf{PTM}^{+}$ is an isomorphism of the semantic algebra.*

The semantic equations in Table 4 now give a state transformer semantics for the refinement calculus. We denote these semantic functions by $\mathcal{C}_{\mathsf{S}}$ and $\mathcal{G}_{\mathsf{S}}$.

Theorem 4. *The isomorphism $T : \mathsf{PGST} \cong \mathsf{PTM}^{+}$ is an isomorphism of the semantics of refinement calculus in the sense that the following diagrams commute:*

$$
\begin{tikzcd}
\text{Statements} \times \text{Env} \arrow[d, "\mathcal{C}_{\mathsf{S}}"] \arrow[dr, "\mathcal{C}_{\mathsf{M}}"] & \\
\mathsf{PGST} \arrow[r, shift right, "T"'] & \mathsf{PTM}^{+} \arrow[l, shift right, "T^{-1}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{Guarded Commands} \times \text{Env} \arrow[d, "\mathcal{G}_{\mathsf{S}}"] \arrow[dr, "\mathcal{G}_{\mathsf{M}}"] & \\
\mathsf{Bool} \times \mathsf{PGST} \arrow[r, shift right, "\mathsf{id} \times T"'] & \mathsf{Bool} \times \mathsf{PTM}^{+} \arrow[l, shift right, "\mathsf{id} \times T^{-1}"']
\end{tikzcd}
$$

5 Beyond the Isomorphism

In the last section, we focused on giving a state transformer semantics to a refinement calculus in such a way that it matches the traditional predicate transformer semantics. The benefit of this exercise is that it gives an intuitive support for the traditional approach. However, we believe this semantics is not ideal. The state transformer set approach gives us a better handle on specifications which does not seem possible in the predicate transformer approach. In this section, we explore the new opportunities.

Consider the semantics of a **do** statement of the form **do** $B \to \boldsymbol{v}\colon[\varphi, \psi]$ **od**. The intent is that the specification $\boldsymbol{v}\colon[\varphi, \psi]$ will eventually be refined to a concrete program statement which will then be repeated during execution. If there are several possible refinements, one of them must be chosen before the execution ever begins. In contrast, the predicate transformer semantics as well as our matching state transformer semantics allow the statement of the loop body to be chosen each time the loop is repeated. In other words, they represent non-determinism instead of under-determinism. To arrive at a better semantics, we redefine the Do operator as follows:

$$\begin{array}{rcl} \mathsf{Do} & : & \mathsf{Bool} \times \mathsf{PGST} \to \mathsf{PGST} \\ \mathsf{Do}(p, S) & = & \{\mathsf{Do}_{\mathsf{GST}}(p, c) \mid c \in S\}^{\dagger} \\ \mathsf{Do}_{\mathsf{GST}}(p, c) & = & \mathbf{fix}_{\mathsf{GST}} \lambda d.\, \lambda x.\, p(x) \leadsto \mathsf{App}(d, c(x)); \{x\} \end{array}$$

In this under-determinism semantics, a fixed command is chosen for the loop body which is then repeated during execution. It does not seem possible to express such an interpretation in the predicate transformer setting.

Morgan's refinement calculus contains a general constant-introduction operator of the form **con** $i\colon\tau.\, C(i)$, where there is no initialization of the constant identifier. This operator is termed "conjunction," and its meaning is explained as the worst program that is better than every $C(i)$. In other words, it is the least upper bound of all $C(i)$'s. Formally, the interpretation is

$$\mathcal{C}[\![\mathbf{con}\ i\colon\tau.\, C]\!]e \;=\; \bigsqcup_{k \in [\![\tau]\!]} \mathcal{C}[\![C]\!]e[i \to k]$$

Since, in PGST, least upper bounds are given by intersections, we obtain

$$\mathcal{C}_{\mathsf{S}}[\![\mathbf{con}\ i\colon\tau.\, C]\!]e \;=\; \bigcap_{k \in [\![\tau]\!]} \mathcal{C}_{\mathsf{S}}[\![C]\!]e[i \to k]$$

which says that a state transformer satisfying **con** $i\colon\tau.\, C(i)$ must satisfy $C(i)$ for every value of i. The semantics in PTM^+ amounts to:

$$\mathcal{C}_{\mathsf{M}}[\![\mathbf{con}\ i\colon\tau.\, C]\!]e \;=\; (\lambda a'.\, \bigcup_{k \in [\![\tau]\!]} \mathcal{C}_{\mathsf{M}}[\![C]\!]e[i \to k](a'))^{*}$$

Given that PGST and PTM^+ are order-isomorphic, these two interpretation match up in the sense of Theorem 4.

However, the traditional semantics [14] is given in PT where all monotone predicate transformers are present and least upper bounds are given pointwise. So, the interpretation of **con** amounts to

$$\mathcal{C}_{\mathsf{P}}[\![\mathbf{con}\ i\colon\tau.\, C]\!]e \;=\; \lambda a'.\, \bigcup_{k \in [\![\tau]\!]} \mathcal{C}_{\mathsf{P}}[\![C]\!]e[i \to k](a')$$

where the subscript P identifies the semantics in PT. This predicate transformer is not positively multiplicative even if every $\mathcal{C}_{\mathsf{P}}[\![C]\!]e[i \to k]$ is positively multiplicative.

What are the consequences of this mismatch? Since positively multiplicative predicate transformers form a proper subset of predicate transformers, our semantics identifies statements which would be semantically distinct in the traditional semantics. The following is an example. For convenience, we use a binary conjunction operator $C_1 \wedge C_2$, which can be regarded as a special case of the general one, for example as (**con** i: bool. **if** $i \to C_1 \,[\!]\, \neg i \to C_2$ **fi**). Consider the two statements:

$$C = [\text{true}, n \geq 0] \wedge [\text{true}, n \leq 0] \quad \text{and} \quad C' = [\text{true}, n = 0]$$

The collection of state transformers satisfying the two specifications is exactly the same. It is $\{\lambda x.\, \{0\}\}^{\dagger}$. (We are taking states to be the values of the variable n.) Hence, $C \equiv C' \equiv (n := 0)$ in our semantics. However, the traditional semantics interprets the two statements as the respective predicate transformers

$$\begin{array}{rcl} t(a') & = & ((a' \supseteq \mathbb{Z}^{+} \cup \{0\}) \vee (a' \supseteq \mathbb{Z}^{-} \cup \{0\})) \leadsto \Sigma; \emptyset \\ t'(a') & = & a' \supseteq \{0\} \leadsto \Sigma; \emptyset \end{array}$$

which are clearly distinct. Whereas t' is equivalent to $n := 0$, t is not equivalent to any program statement. Nevertheless, $n := 0$ is the only nontrivial statement that C can be refined to. These distinctions have nontrivial consequences under sequential composition. Consider

$$D = C; [n = 0, n = 9] \quad \text{and} \quad D' = C'; [n = 0, n = 9].$$

The traditional semantics equates D to **abort**, whereas D' is equivalent to $n := 9$. The equivalence $D \equiv$ **abort** is surprising. We are hard put to find any intuitive explanation of why D should be equivalent to **abort**.

To pin down the difference between the traditional semantics and ours, we consider the following (hypothetical) $\wedge$-distributivity law:

$$(C_1 \wedge C_2); S \;\sqsubseteq\; (C_1; S) \wedge (C_2; S)$$

To us, this law seems unreasonable. Basically, it says that the requirements for a composite command $(C_1; S) \wedge (C_2; S)$ entail requirements for the component commands $(C_1 \wedge C_2)$. However, the law is validated by Morgan's semantics and the fact $D \sqsubseteq$ **abort** can be derived using it. This law is *not valid* in our semantics.

6 Conclusion

Refinement calculi have been proposed as integrated frameworks for combining programs and specifications and as vehicles for deriving programs from specifications. But their traditional semantics, defined in the predicate transformer setting, leaves several questions unanswered. The most important of these is what specification statements mean in terms of one's operational intuitions. By giving a semantics in terms of sets of state transformers, we hope to have answered these questions. We showed that the mysterious concept of "miracle" has a natural explanation in terms of partially defined state transformers. We also proposed that the non-multiplicative predicate transformers used in the traditional semantics may not be ideal, whereas a semantics based on positively multiplicative predicate transformers has a natural correspondence with the state transformer semantics.

We leave open the question of what it means for a semantics to be ideal. For programming languages, the ideal semantics is often taken to be a fully abstract semantics, i.e., one whose equality relation is the same as observational equivalence. For specification languages, it is not yet clear what observational equivalence might mean.

We have considered a very simple language here to focus on the main ideas. The extension of the ideas to cover procedures, abstract data types and object-oriented concepts remains to be addressed.

Acknowledgements We have benefited from discussions with David Naumann and Peter O'Hearn. This research was carried out as part of a joint US-Brazil project on refinement of object-oriented programs whose members include David Naumann, Ana Cavalcanti, Augusto Sampaio and Paulo Borba. It is supported by NSF grant INT-98-13845.

References

[1] K. Apt and G. Plotkin. A Cook's tour of countable non-determinism. In *8th ICALP*. Springer-Verlag, 1981.

[2] K. Apt and G. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, October 1986.

[3] R.-J. R. Back. On the correctness of refinement steps in program development. Report A-1978-4, Department of Computer Science, University of Helsinki, 1978.

[4] R.-J. R. Back. On the notion of correct refinement of programs. Technical report, University of Helsinki, 1979.

[5] R.-J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[6] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, Berlin, 1998.

[7] M. M. Bonsangue and J. N. Kok. Isomorphism between state and predicate transformers. In *Math. Foundations of Comput. Sci.*, volume 711 of *LNCS*, pages 301–310. Springer-Verlag, Berlin, 1993.

[8] M. M. Bonsangue and J. N. Kok. The weakest precondition calculus: Recursion and duality. *Formal Aspects of Computing*, 6, 1994.

[9] J. W. de Bakker. Recursive programs as predicate transformers. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*. North-Holland, Amsterdam, 1978.

[10] E. Denney. *A Theory of Programm Refinement*. PhD thesis, Univ. of Edinburgh, 1999.
[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.
[12] P. H. B. Gardiner. Algebraic proofs of consistency and completeness. *Theoretical Comput. Sci.*, 150:161–191, 1995.
[13] P. H. B. Gardiner, C. E. Martin, and O. de Moor. An algebraic construction of predicate transformers. *Science of Computer Programming*, 22:21–44, 1994.
[14] P. H. B. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Comput. Sci.*, 87:143–162, 1991. Reprinted in [23].
[15] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
[16] E. C. R. Hehner. *The Logic of Programming*. Prentice-Hall, London, 1984.
[17] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *Information Processing 62: Proceedings of IFIP Congress 1962*, pages 21–28. North-Holland, Amsterdam, 1963.
[18] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1997.
[19] C. C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3), Jul 1988. Reprinted in [23].
[20] C. C. Morgan. The cuppest capjunctive capping, and Galois. In A. W. Roscoe, editor, *A Classical Mind: Essays in Honor of C. A. R. Hoare*. Prentice-Hall International, 1994.
[21] C. C. Morgan. *Programming from Specifications, 2nd Edition*. Prentice-Hall, 1994.
[22] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27, 1991. Reprinted in [23].
[23] C. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1992.
[24] J. M. Morris. The theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.
[25] D. Naumann. A categorical model for higher order imperative programming. *Math. Struct. Comput. Sci.*, 8(4):351–399, Aug 1998.
[26] D. Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of Computer Programming*, 1999. To appear.
[27] G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, October 1989.
[28] G. D. Plotkin. Dijkstra's predicate transformers and Smyth's power domains. In D. Bjorner, editor, *Abstract Software Specifications*, volume 86 of *LNCS*, pages 527–553. Springer-Verlag, 1980.
[29] M. B. Smyth. Powerdomains and predicate transformers: A topological view. In J. Diaz, editor, *Intern. Colloq. Aut., Lang. and Program.*, volume 154 of *LNCS*, pages 662–675. Springer-Verlag, 1983.
[30] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.
[31] M. Wand. A characterization of weakest preconditions. *J. Comput. Syst. Sci.*, 15(2):209–212, 1977.