Semantics of Separation-logic Typing and Higher-order Frame Rules

Lars Birkedal Noah Torp-Smith
IT University of Copenhagen
{birkedal, noah}@itu.dk

Hongseok Yang
ERC-ACI, Seoul National University
hyang@ropas.snu.ac.kr

Abstract

We show how to give a coherent semantics to programs that are well-specified in a version of separation logic for a language with higher types: idealized algol extended with heaps (but with immutable stack variables). In particular, we provide simple sound rules for deriving higher-order frame rules, allowing for local reasoning.

1. Introduction

Separation logic [20, 18, 5, 14, 9, 4] is a Hoare-style program logic, and variants of it have been applied to prove correct interesting pointer algorithms such as copying a dag, disposing a graph, the Schorr-Waite graph algorithm, and Cheney's copying garbage collector. The main advantage of separation logic compared to ordinary Hoare logic is that it facilitates *local reasoning*, formalized via the so-called *frame rule* using a connective called *separating conjunction*. The development of separation logic has mostly focused on *low-level* languages with heaps and pointers, although in recent work [10] it was shown how to extend separation logic to a language with a simple kind of procedures, and a second-order frame rule was proved sound.

Our aim here is to extend the study of separation logic to *high-level* languages, in particular to *higher-order* languages, in such a way that a wide collection of frame rules are sound, thus allowing for local reasoning in the presence of higher-order procedures. For concreteness, we choose to focus on the language of idealized algol extended with heaps and pointers and we develop a semantics for this language in which all commands and procedures are appropriately local. Our approach is to refine the type system of idealized algol extended with heaps, essentially by making specifications be types, and give semantics to well-specified programs. Thus we develop a *separation-logic type system* for idealized algol extended with heaps. It is a dependent type theory and the types include Hoare triples, rules corresponding to the rules of separation logic, and subtyping rules formalizing higher-order versions of the frame rule of separation logic.

Our type system is related to modern proposals for type systems for low-level imperative languages, such as TAL [7], in that types may express state changes (since they include forms of Hoare triples as types). The type system for TAL was proved sound using an operational semantics. We provide a soundness proof of our type system using a denotational semantics which we, moreover, formally relate to the standard semantics for idealized algol [11, 15]. The denotational semantics of a well-typed program is given by induction on its typing derivation and the relation to the standard semantics for idealized algol is then used to prove that the semantics is *coherent* (i.e., is independent of the chosen typing derivation). We should perhaps stress that soundness is not a trivial issue: Reynolds has shown [10] that already the soundness of the second-order frame rule is tricky, by proving that if a proof system contains the second-order frame rule and the conjunction rule, together with the ordinary frame rule and rule of Consequence, then the system becomes inconsistent. The semantics of our system proves that if we drop the conjunction rule, then we get soundness of all higher-order frame rules, including the second-order one.

In idealized algol, variables are allocated on a stack and they are mutable (i.e., one can assign to variables). We only consider *immutable* variables (as in the ML programming language) for simplicity. The reason for this choice is that all mutation then takes place in the heap and thus we need not bother with so-called *modifies clauses* on frame rules, which become complicated to state already for the second-order frame rule [10].

We now give an intuitive overview of the technical development. Recall that the standard semantics of idealized algol is given using the category CPO of pointed complete partial orders and continuous functions. Thus types are interpreted as pointed complete partial orders and terms (programs) are interpreted as continuous functions. The semantics of our refined type system is given by refining the standard semantics. A type θ in our refined type system specifies which elements of the "underlying" type in the standard semantics satisfy the specification corresponding to θ and are appropriately local (to ensure soundness of the frame

rules), that is, it "extracts" those elements. Moreover, the semantics also equates elements, which cannot be distinguished by clients, that is, it quotients some of the extracted elements. Corresponding to these two aspects of the semantics we introduce two categories, $\mathcal{C}$ and $\mathcal{D}$, where $\mathcal{C}$ just contains the extracted elements and $\mathcal{D}$ is a quotient of $\mathcal{C}$. Thus there is a faithful functor from $\mathcal{C}$ to CPO and a full functor from $\mathcal{C}$ to $\mathcal{D}$. We show that the categories $\mathcal{C}$ and $\mathcal{D}$ are cartesian closed and have additional structure to interpret the higher-order frame rules, and that the mentioned functors preserve all this structure. The semantics of our type system is then given in the category $\mathcal{D}$ and the functors relating $\mathcal{C}$, $\mathcal{D}$, and CPO are then used to prove coherence of the semantics. In fact, as mentioned above, our type system is a *dependent* type theory, with dependent product type $\Pi_i\theta$ intuitively corresponding to the specification given by universally quantifying i in the specification corresponding to θ (the usual Curry-Howard correspondence). For this reason the semantics is really not given in $\mathcal{D}$ but rather in the family fibration $Fam(\mathcal{D}) \to \mathsf{Set}$ over $\mathcal{D}$.

The remainder of this paper is organized as follows. In Section 2, we define the storage model and assertion language used in this paper, thus setting the stage for our model. In Section 3, we provide the syntax of the version of idealized algol we use in this paper. In particular, we introduce our separation-logic type system, which includes an extended subtyping relation. In Section 4, we present the main contribution of the paper, a model which allows a sound interpretation, which we also show to be coherent and in harmony with the standard semantics. In the last sections we give pointers to related and future work, and conclude.

2. Storage Model and Assertion Language

We use the usual storage model of separation logic with one minor modification: we make explicit the shape of stack storage. Let $\mathsf{Ids} = \{i, j, \ldots\}$ be a countably infinite set of variables, and let Δ range over finite subsets of Ids. We use the following semantic domains:

$$\begin{array}{rcrcl} \eta & \in & [\![\Delta]\!] & \stackrel{\text{def}}{=} & \Delta \to Int, \\ h & \in & Heap & \stackrel{\text{def}}{=} & Nat \rightharpoonup_{\mathsf{fin}} Int, \\ (\eta, h) & \in & State(\Delta) & \stackrel{\text{def}}{=} & [\![\Delta]\!] \times Heap. \end{array}$$

The set Δ models the set of variables in scope, and an element η in $[\![\Delta]\!]$ specifies the values of those stack variables. We sometimes call η an *environment* instead of a *stack*, in order to emphasize that all variables are immutable. An element h in $Heap$ denotes a heap; the domain of h specifies the set of allocated cells, and the actual action of h determines the contents of those allocated cells. We recall the disjointness predicate $h\#h'$ and the (partial) heap combination operator $h \cdot h'$ from separation logic. The predicate $h\#h'$ means that $\mathsf{dom}(h) \cap \mathsf{dom}(h') = \emptyset$; and, $h \cdot h'$ is defined only for such disjoint heaps h and h', and in that case, it denotes the combined heap $h \cup h'$.

Properties of states are expressed using the assertion language of classical separation logic [20]: [1]

$$\begin{array}{rcl} E & ::= & i \mid 0 \mid 1 \mid E + E, \\ P & ::= & E = E \mid E \mapsto E \mid \mathsf{emp} \mid P * P \\ & & \mid \mathsf{true} \mid P \wedge P \mid \neg P \mid \forall i.\, P. \end{array}$$

The assertion $E \mapsto E'$ means that the current heap has only one cell E and, moreover, that the content of the cell is E'. When we do not care about the contents, we write $E \mapsto -$; formally, this is an abbreviation of $\neg(\forall i.\, \neg E \mapsto i)$ for some i not occurring in E. The next two assertions, emp and $P * Q$, are the most interesting features of this assertion language. The empty predicate emp means that the current heap is empty, and the separating conjunction $P * Q$ means that the current heap can be partitioned into two parts, one satisfying P and another satisfying Q.

As in the storage model, we make explicit which set of free variables we are considering an expression or an assertion under. Thus, letting fiv be a function that takes an expression or an assertion and returns the set of free variables, we often write assertions as $\Delta \vdash P$ to indicate that $\mathsf{fiv}(P) \subseteq \Delta$, and that P is currently being considered for environments of the shape Δ. Likewise, we often write $\Delta \vdash E$ for expressions.

The interpretations of an expression $\Delta \vdash E$ and an assertion $\Delta \vdash P$ are of the forms

$$[\![\Delta \vdash E]\!] : [\![\Delta]\!] \to Int, \qquad [\![\Delta \vdash P]\!] : [\![\Delta]\!] \to \mathcal{P}(Heap).$$

The interpretation of expressions is standard, just like that of assertions. We include part of the definition of the interpretation of assertions here.

$$\begin{array}{rcl} [\![\Delta \vdash E \mapsto E']\!]_\eta & = & \{[[\![\Delta \vdash E]\!]_\eta \to [\![\Delta \vdash E']\!]_\eta]\} \\ [\![\Delta \vdash \mathsf{emp}]\!]_\eta & = & \{[]\} \end{array}$$

$$[\![\Delta \vdash P * P']\!]_\eta = \{h * h' \mid h\#h' \wedge h \in [\![\Delta \vdash P]\!]_\eta \wedge h' \in [\![\Delta \vdash P']\!]_\eta\}$$

3. Programming Language

The programming language is Reynolds's idealized algol [15] adapted for "separation-logic typing." It is a call-by-name typed lambda calculus, extended with heap operations, dependent functions, and Hoare-triple types. As explained in the introduction, we only consider immutable variables.

[1] The assertion language of separation logic also contains the separating implication $-\!\!*$. Since that connective does not raise any new issues in connection with the present work, we omit it here.

The types of the language are defined as follows. We write $\Delta \vdash \theta : \mathsf{Type}$ for a type θ in context Δ. The set of types is defined by the following inference rules (in which P and Q range over assertions):

$$\frac{\mathsf{fiv}(P) \subseteq \Delta \quad \mathsf{fiv}(Q) \subseteq \Delta}{\Delta \vdash \{P\}{-}\{Q\}{:}\mathsf{Type}} \qquad \frac{\Delta \vdash \theta{:}\mathsf{Type} \quad \mathsf{fiv}(P) \subseteq \Delta}{\Delta \vdash \theta \otimes P{:}\mathsf{Type}}$$

$$\frac{\Delta \cup \{i\} \vdash \theta : \mathsf{Type} \quad i \notin \Delta}{\Delta \vdash \Pi_i\theta{:}\mathsf{Type}} \qquad \frac{\Delta \vdash \theta{:}\mathsf{Type} \quad \Delta \vdash \theta'{:}\mathsf{Type}}{\Delta \vdash \theta \to \theta'{:}\mathsf{Type}}$$

Note that the types are *dependent types*, in that they may depend on variables i (see the first rule above). One way to understand a type is to read it as a specification for terms, i.e., through the Curry-Howard correspondence. A Hoare-triple type $\{P\}{-}\{Q\}$ is a direct import from separation logic; it denotes a set of commands c that satisfy the Hoare triple $\{P\}c\{Q\}$. An invariant extension $\theta \otimes P$ is satisfied by a term M if and only if for one part of the heap, the behavior of M satisfies θ and for the other part of the heap, M maintains the invariant P. For instance, $\{P\}{-}\{Q\} \otimes P_0$ intuitively consists of commands that given an input state satisfying $P * P_0$, so that the input state may be split into a P-part and a P_0-part, change the P-part so that the result satisfies Q; and for the P_0-part, the commands modify it freely, but maintain the invariant P_0.

The type $\Pi_i\theta$ is a dependent product type, as in standard dependent type theory (under Curry-Howard it corresponds to the specification given by universally quantifying i in the specification corresponding to θ). Intuitively, $\Pi_i\theta$ denotes functions from integers such that given an integer n, they return a value satisfying $\theta[n/i]$. For example, the type $\Pi_i\{j \mapsto -\}{-}\{j \mapsto i!\}$ specifies a factorial function that computes the factorial of i and stores the result in the heap cell j.

The pre-terms of the language are given by the following grammar:

$$\begin{aligned} M ::= {} & x \mid \lambda x{:}\,\theta.M \mid MM \mid \lambda i.M \mid ME \\ & \mid \mathsf{fix}\,M \mid \mathsf{ifz}\,E\,M\,M \mid M;M \mid \mathsf{let}\,i = \mathsf{new}\,\mathsf{in}\,M \\ & \mid \mathsf{free}(E) \mid [E] := E \mid \mathsf{let}\,i = [E]\,\mathsf{in}\,M, \end{aligned}$$

where E is an integer expression defined in Section 2. The language has the usual constructs for a higher-order imperative language with heap operations, but it has two distinct features. First, it treats the integer expressions as "second class": the terms M never have the integer type, and all integer expressions inside a term are from the separate grammar for E defined in Section 2. Second, no "integer variables" i can be modified in this language; only heap cells can be modified. Note that the language has two forms of abstraction and application, one for general terms and the other for integer expressions. A consequence of this stratification is that all integer expressions terminate, because the grammar for E does not contain the recursion operator.

The language has four heap operations. Command $\mathsf{let}\,i = \mathsf{new}\,\mathsf{in}\,M$ allocates a heap cell, binds i to the address of the allocated cell, and executes the command M.[2] An allocated cell i can be disposed by $\mathsf{free}(i)$. The remaining two commands access the content of a cell. The command $[i] := E'$ changes the content of cell i by E'; and $\mathsf{let}\,j = [i]\,\mathsf{in}\,M$ reads the content of cell i, binds j to the read value, and executes M. Note that the allocation and lookup commands involve the "continuation", and make the bound variable available in the continuation; such indirect-style commands are needed because all variables are immutable.

The typing rules of the language decide a judgment of the form $\Gamma \vdash_\Delta M{:}\,\theta$, where Γ is a list of type assignments to identifiers $\Gamma = x_1 : \theta_1, \ldots, x_n : \theta_n$, and where the set Δ contains all the free variables appearing in Γ, M, θ.

The type system is shown in Figure 1. For notational simplicity we have omitted some obvious side-conditions of the form $\Delta \vdash \theta : \mathsf{Type}$ which ensure that, for a judgment $\Gamma \vdash_\Delta M{:}\,\theta$, the set Δ always contains all the free variables appearing in Γ, M, θ, and that the type assignment Γ is always well-formed. There are three classes of rules. The first class consists of the rules from the simply typed lambda calculus extended with dependent product types and recursion. The second class consists of the rules for the imperative constructs, all of which come from separation logic. The last class consists of the subsumption rule based on the subtyping relation $\preceq_\Delta$, which is the most interesting part of our type system. The proof rules for $\preceq_\Delta$ define a preorder between types with free variables in Δ, and include all the usual structural subtyping rules [13, chp. 15]. The rules specific to our system are: the encoding of Consequence in Hoare logic; the generalized frame rule that adds an invariant to all types; and the distribution rules for an added invariant assertion.

The generalized frame rule, $\theta \preceq_\Delta \theta \otimes P_0$, means that if a program satisfies θ and an assertion P_0 does not "mention" any cells described by θ, then the program preserves P_0. Note that this rule indicates that the types in our system are *tight* [5, 20]: if a program satisfies θ, it can only access heap cells "mentioned" in θ. This is why an assertion P_0 for "unmentioned" cells is preserved by the program. For instance, if a program M has a type of the form

$$\theta_1 \to \ldots \to \theta_n \to \{P\}{-}\{Q\},$$

the tightness of the type says that all the cells that M can directly access must appear in the precondition P. Thus, if no cells in an assertion P_0 appear in P, program M maintains P_0, as long as argument procedures maintain it. Such a fact can, indeed, be inferred by the generalized frame rule

[2] We consider single-cell allocation only in order to simplify the presentation; it is straightforward to adapt our results to a language with allocation of n consecutive cells.

$$\frac{}{\Gamma, x{:}\theta \vdash_\Delta x : \theta}\ (\mathsf{fiv}(\Gamma,\theta) \subseteq \Delta) \qquad \frac{\Gamma, x{:}\theta \vdash_\Delta M : \theta'}{\Gamma \vdash_\Delta \lambda x{:}\theta.M : \theta \to \theta'} \qquad \frac{\Gamma \vdash_\Delta M : \theta' \to \theta \quad \Gamma \vdash_\Delta M' : \theta'}{\Gamma \vdash_\Delta MM' : \theta}$$

$$\frac{\Gamma \vdash_{\Delta\cup\{i\}} M : \theta'}{\Gamma \vdash_\Delta \lambda i.M : \Pi_i\theta'}\ (\mathsf{fiv}(\Gamma) \subseteq \Delta) \quad \frac{\Gamma \vdash_\Delta M : \Pi_i\theta}{\Gamma \vdash_\Delta ME : \theta[E/i]}\ (\mathsf{fiv}(E) \subseteq \Delta) \quad \frac{\Gamma \vdash_\Delta M : \theta \to \theta}{\Gamma \vdash_\Delta \mathsf{fix}\, M : \theta}$$

$$\frac{\Gamma \vdash_\Delta M : \{P \wedge E{=}0\}{-}\{Q\} \quad \Gamma \vdash_\Delta M' : \{P \wedge E{\neq}0\}{-}\{Q\}}{\Gamma \vdash_\Delta \mathsf{if}\mathsf{z}\, E\, M\, M' : \{P\}{-}\{Q\}} \qquad \frac{\Gamma \vdash_\Delta M : \{P\}{-}\{P'\} \quad \Gamma \vdash_\Delta M' : \{P'\}{-}\{Q\}}{\Gamma \vdash_\Delta (M; M') : \{P\}{-}\{Q\}}$$

$$\frac{\Gamma \vdash_{\Delta\cup\{i\}} M : \{P * i \mapsto -\}{-}\{Q\}}{\Gamma \vdash_\Delta \mathsf{let}\, i = \mathsf{new}\, \mathsf{in}\, M : \{P\}{-}\{Q\}}\ (i \notin \mathsf{fiv}(\Gamma, P, Q)) \quad \frac{\Gamma \vdash_{\Delta\cup\{i\}} M : \{P * E \mapsto i\}{-}\{Q\}}{\Gamma \vdash_\Delta \mathsf{let}\, i = [E]\, \mathsf{in}\, M : \{P * E \mapsto -\}{-}\{Q\}}\ (i \notin \mathsf{fiv}(\Gamma, E, P, Q))$$

$$\frac{}{\Gamma \vdash_\Delta \mathsf{free}(E) : \{E \mapsto -\}{-}\{\mathsf{emp}\}}\ (\mathsf{fiv}(\Gamma, E) \subseteq \Delta) \quad \frac{}{\Gamma \vdash_\Delta [E] := E' : \{E \mapsto -\}{-}\{E \mapsto E'\}}\ (\mathsf{fiv}(\Gamma, E, E') \subseteq \Delta)$$

$$\frac{\Gamma \vdash_\Delta M : \theta \quad \theta \preceq_\Delta \theta'}{\Gamma \vdash_\Delta M : \theta'}$$

where $\preceq_\Delta$ is the usual structural subtyping relation [13, chp. 15] for types over Δ, extended with the following rules:

$$\{P'\}{-}\{Q'\} \preceq_\Delta \{P\}{-}\{Q\}\ \text{(when for all } \eta \in [\![\Delta]\!],\ [\![P]\!]_\eta \subseteq [\![P']\!]_\eta \text{ and } [\![Q']\!]_\eta \subseteq [\![Q]\!]_\eta)$$
$$\theta \preceq_\Delta \theta \otimes P \qquad (\{P\}{-}\{Q\}) \otimes P_0 \simeq_\Delta \{P * P_0\}{-}\{Q * P_0\} \qquad (\Pi_i\theta) \otimes P \simeq_\Delta \Pi_i\theta \otimes P$$
$$(\theta \otimes Q) \otimes P \simeq_\Delta \theta \otimes (Q * P) \qquad (\theta \to \theta') \otimes P \simeq_\Delta (\theta \otimes P \to \theta' \otimes P)$$

Figure 1. Typing Rules

together with the distribution rules:

$$\begin{array}{ll} & \theta_1 \to \ldots \to \theta_n \to \{P\}{-}\{Q\} \\ \preceq_\Delta & \quad (\because \theta \preceq_\Delta \theta \otimes P_0) \\ & (\theta_1 \to \ldots \to \theta_n \to \{P\}{-}\{Q\}) \otimes P_0 \\ \preceq_\Delta & \quad (\because (\theta \to \theta') \otimes P_0 \preceq_\Delta (\theta \otimes P_0 \to \theta' \otimes P_0)) \\ & (\theta_1 \otimes P_0 \to \ldots \to \theta_n \otimes P_0 \to \{P\}{-}\{Q\} \otimes P_0) \\ \preceq_\Delta & \quad (\because \{P\}{-}\{Q\} \otimes P_0 \preceq_\Delta \{P * P_0\}{-}\{Q * Q_0\}) \\ & (\theta_1 \otimes P_0 \to \ldots \to \theta_n \otimes P_0 \to \{P * P_0\}{-}\{Q * P_0\}). \end{array}$$

The generalized frame rule, the distribution rules, and the structural subtyping rule for function types all together give many interesting higher-order frame rules, including the second-order frame rule. The common mechanism for obtaining such a rule is: first, add an invariant assertion by the generalized frame rule, and then, propagate the added assertion all the way down to a base triple type by the distribution rules. The structural subtyping rule for the function type allows us to apply this construction for a sub type-expression in an appropriate covariant or contravariant way. For instance, we can derive a third-order frame rule as follows:

$$\begin{array}{ll} & (\{P_1\}{-}\{Q_1\} \to \{P_2\}{-}\{Q_2\}) \to \{P_3\}{-}\{Q_3\} \\ \preceq_\Delta & \quad (\because \theta \preceq_\Delta \theta \otimes P) \\ & \Big((\{P_1\}{-}\{Q_1\} \to \{P_2\}{-}\{Q_2\}) \to \{P_3\}{-}\{Q_3\}\Big) \otimes P \\ \preceq_\Delta & \quad (\because (\theta \to \theta') \otimes P \preceq_\Delta (\theta \otimes P \to \theta' \otimes P)) \\ & (\{P_1\}{-}\{Q_1\}{\otimes}P \to \{P_2\}{-}\{Q_2\}{\otimes}P) \to \{P_3\}{-}\{Q_3\}{\otimes}P \\ \preceq_\Delta & \quad (\because \text{structural subtyping}) \\ & (\{P_1\}{-}\{Q_1\}{\otimes}P \to \{P_2\}{-}\{Q_2\}) \to \{P_3\}{-}\{Q_3\}{\otimes}P \\ \preceq_\Delta & \quad (\because \{P_0\}{-}\{Q_0\} \otimes P \simeq_\Delta \{P_0 * P\}{-}\{Q_0 * P\}) \\ & (\{P_1{*}P\}{-}\{Q_1{*}P\} \to \{P_2\}{-}\{Q_2\}) \to \{P_3{*}P\}{-}\{Q_3{*}P\}. \end{array}$$

4. Semantics

In this section we present our main contribution, the semantics that formalizes the underlying intuitions of the separation-logic type system. In particular, we formalize the following three intuitive properties of the type system:

1. The types in the separation-logic type system refine the conventional types. A separation-logic type specifies a stronger property of a term, and restricts clients of such terms by asking them to only depend upon what can be known from the type. For instance, the type $\{1 \mapsto 3\}{-}\{1 \mapsto 0\}$ of a term M indicates not just that M is a command, but also that M stores 0 to cell 1 if cell 1 contains 3 initially. Moreover, this type forces clients to run M only when cell 1 contains 3.

2. The higher-order frame rules in the type system imply that all programs behave locally.

3. The type system, however, does not change the computational behavior of each program.

We formalize the first intuitive property by means of partial equivalence relations. Roughly, each type θ in our semantics determines a partial equivalence relation (in short, per) over the meaning of the "underlying type" $\overline{\theta}$. The domain of a per over a set A is a subset of A; this indicates that θ indeed specifies a stronger property than $\overline{\theta}$. The other part of a per, namely the equivalence relation part, explains that the type system restricts the clients, so that no type-checked clients can tell apart two equivalent programs. For

instance, $\{1 \mapsto 3\}-\{1 \mapsto 0\}$ determines a per over the set of all commands. The domain of this per consists of commands satisfying $\{1 \mapsto 3\}-\{1 \mapsto 0\}$, and the per equates two such commands if they behave identically when cell 1 contains 3 initially. The equivalence relation implies that type-checked clients run a command of $\{1 \mapsto 3\}-\{1 \mapsto 0\}$ only when cell 1 contains 3.

We justify the other two intuitive properties by proving technical lemmas about our semantics. For number 2, we prove the soundness of all the subtyping rules, including the generalized frame rule and the distribution rules. For number 3, we prove that our semantics has been obtained by extracting and then quotienting semantic elements in the conventional semantics; yet, this extraction and quotienting does not reduce the computational information of semantic elements.

In this section, we first define categories $\mathcal{C}$ and $\mathcal{D}$, corresponding to the extraction and quotienting, respectively. Next we give the interpretation of types and terms. Finally, we connect our semantics with the conventional semantics, and prove that our semantics is indeed obtained by extracting and quotienting from the conventional semantics.

4.1. Categories $\mathcal{C}$ and $\mathcal{D}$

We construct $\mathcal{C}$ and $\mathcal{D}$ by modifying the category CPO of pointed cpos and continuous functions. For $\mathcal{C}$, we impose a parameterized per on each cpo, and extract only those morphisms in CPO that preserve such pers (at all instantiations). Each per formalizes that each type θ corresponds to a specification over the underlying type $\overline{\theta}$, and the preservation of the pers ensures that all the morphisms in $\mathcal{C}$ satisfy the corresponding specifications. The parameterization of each per guarantees that all morphisms in $\mathcal{C}$ behave locally (in the sense of higher-order frame rules). The other category $\mathcal{D}$ is a quotient of $\mathcal{C}$. Intuitively, the quotienting of $\mathcal{C}$ reflects that our type system also restricts the clients of a term; thus, more terms cannot be distinguished observationally.

We define the "extracting" category $\mathcal{C}$ first. Let $Pred$ be the set of *predicates*, i.e., subsets of $Heap$. We recall the semantic version of separating connectives, emp and $*$, on $Pred$. For $p, q \in Pred$,

$$h \in emp \iff h = \lambda n.undef$$
$$h \in p * q \iff \exists h_1 h_2.\, h_1 * h_2 = h \wedge h_1 \in p \wedge h_2 \in q.$$

The category $\mathcal{C}$ is defined as follows:

- objects: (A, R) where A is a pointed cpo, and R is a family of admissible pers[3] indexed by predicates such that
$$\forall p, q \in Pred.\, R(p) \subseteq R(p * q);$$
- morphisms: $f\colon (A, R) \to (B, S)$ is a continuous function from A to B such that
$$\forall p.\, f[R(p) \to S(p)]f,$$
i.e., f maps $R(p)$ related elements to $S(p)$ related elements.

Intuitively, each object (A, R) denotes a specification parameterized by invariant extension. The first component A denotes the underlying set from which we select "correct" elements. $R(emp)$ denotes the initial specification of this object where no invariant is added by the frame rule. The domain $|R(emp)|$ of per $R(emp)$ indicates which elements satisfy the specification, and the equivalence relation on $|R(emp)|$ expresses how the specification is also used to limit the interaction of a client: the client can only do what the specification guarantees, so more elements become equivalent observationally. The per $R(p)$ at another predicate p denotes an extended specification by the invariant p.

We illustrate the intuition of $\mathcal{C}$ with a "Hoare-triple" object $[p, q]$ for some $p, q \in Pred$. Let comm be the set of all functions c from $Heap$ to $\mathcal{P}(Heap \cup \{wrong\})$ that satisfy safety monotonicity and the frame property:

- *Safety Monotonicity*: for all $h, h_0 \in State$, if $h \# h_0$ and $wrong \notin c(h)$, then $wrong \notin c(h * h_0)$;
- *Frame Property*: for all $h, h_0, h_1' \in State$, if $h \# h_0$, $wrong \notin c(h)$, and $h_1' \in c(h * h_0)$, then there exists h' such that $h_1' = h_0 * h'$ and $h' \in c(h)$.

Intuitively, comm contains all commands that satisfy the (first-order) frame rule [9], and it forms the underlying set for all Hoare-triple specifications. Indeed, it is the first component of the Hoare-triple object $[p, q]$, where the order on comm is given by:

$$c \sqsubseteq c' \iff \forall h.\, c(h) \subseteq c'(h).$$

The real meaning of $[p, q]$ is given by the second component R. For each predicate p_0, the domain of $R(p_0)$ consists of all "commands" in comm that satisfy $\{p * p_0\}-\{q * p_0\}$:

$$c \in |R(p_0)| \iff \forall h \in p * p_0.\, c(h) \subseteq q * p_0.$$

The equivalence relation $R(p_0)$ relates c and c' in $|R(p_0)|$ iff c and c' behave the same for the inputs in $p * p_0 * true$:

$$true = \{h \mid h \in Heap\}$$
$$c[R(p_0)]c' \iff \forall h \in p * p_0 * true.\, c(h) = c'(h).$$

Intuitively, this equivalence relation means that the type system allows a client to execute c or c' in h only when h satisfies $p * p_0 * p'$ for some p'; the predicate p' reflects that the frame rule in the type system allows the "widening" of a specification by an invariant.

[3] A per R_0 on A is admissible iff $(\bot, \bot) \in R_0$ and R_0 is a sub-cpo of $A \times A$.

The category $\mathcal{C}$ is cartesian closed, and it has all small products. The terminal object is $(\{\bot\}, R)$ where $R(p)$ is $\{(\bot, \bot)\}$ for all p, and all the small products are given pointwise; for instance, $(A, R) \times (B, S)$ is $(A \times B, \{R(p) \times S(p)\}_p)$. The exponential of (A, R) and (B, S) is subtle, and its per component involves the quantification over all predicates. Let $R \Rightarrow S$ be the following family of pers on the continuous function space $A \Rightarrow B$:

$$f[(R \Rightarrow S)(p)]g \iff \forall q \in Pred.\, f[R(p*q) \to S(p*q)]g.$$

Here f and g range over the domain of $(R \Rightarrow S)(p)$, which contains only local functions: it consists of functions f in $|R(p) \to S(p)|$ such that

$$\forall q.\, f[R(p * q) \to S(p * q)]f.$$

Note that this condition is precisely the semantic version of the frame rule. The exponential $(A, R) \Rightarrow (B, S)$ is given by $(A \Rightarrow B, R \Rightarrow S)$.

Lemma 1 *$\mathcal{C}$ is cartesian closed, and has all small limits.*

Another important feature of $\mathcal{C}$ is that it validates higher-order frame rules. Let $\mathcal{P}_r$ be the preorder $(Pred, \sqsubseteq)$ with $\sqsubseteq$ defined by predicate extension:

$$p \sqsubseteq r \iff \exists q. p * q = r.$$

Category $\mathcal{C}$ has an "invariant-extension" functor inv from $\mathcal{C} \times \mathcal{P}_r$ to $\mathcal{C}$ defined by:

$$\mathsf{inv}((A, R), p) = (A, R(p * -)) \text{ and } \mathsf{inv}(f, p \sqsubseteq q) = f.$$

Functor inv corresponds to the type constructor $\otimes$ in our language; given a "type" (A, R) and a predicate p, inv extends (A, R) by adding an invariant p. For instance, when a triple object $[p', q']$ is extended with p, it becomes $[p' * p, q' * p]$.

Functor inv validates the subtyping rules that express higher-order frame rules: the generalized frame rule $\theta \preceq \theta \otimes P$ and the rules for distributing $\otimes$ over each type constructor. We first show that the functoriality of inv gives the soundness of the generalized frame rule. Note that for all predicates p, $emp \sqsubseteq p$, and that $\mathsf{inv}(-, emp)$ is the identity functor on $\mathcal{C}$. Thus, for each (A, R), the functoriality of inv gives a morphism from (A, R) to $\mathsf{inv}((A, R), p)$. This morphism gives the soundness of the subtyping rule $\theta \preceq \theta \otimes P$.

The soundness of the other distribution rules follows from the fact that for all p, $\mathsf{inv}(-, p)$ preserves most of the structure of $\mathcal{C}$. For instance, $\mathsf{inv}(-, p)$ preserves the exponential of $\mathcal{C}$, because for all objects (A, R) and (B, S) and all predicates q, we have that

$$\begin{array}{ll} & f[(R(p * -) \Rightarrow S(p * -))(q)]g \\ \iff & \forall q'.f[R(p * (q * q')) \to S(p * (q * q'))]g \\ \iff & \forall q'.f[R((p * q) * q') \to S((p * q) * q')]g \\ \iff & f[(R \Rightarrow S)(p * q)]g. \end{array}$$

Lemma 2 *For each predicate p, $\mathsf{inv}(-, p)$ preserves the cartesian closed structure and all the small products of $\mathcal{C}$ on the nose.*

Lemma 3 *For all predicates p and q, $\mathsf{inv}(-, p) \circ \mathsf{inv}(-, q) = \mathsf{inv}(-, p * q)$.*

For now, the final remark on $\mathcal{C}$ is that the triple-object generator $[-, -]$ can be made into a functor, whose morphism action validates the subtyping rule for Consequence. Let $\mathcal{P}$ be the set of predicates ordered by the subset inclusion $\subseteq$. Generator $[-, -]$ can be extended to a functor tri from $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$ to $\mathcal{C}$:

$$\mathsf{tri}(p, q) = [p, q] \text{ and } \mathsf{tri}(p \subseteq p', q' \subseteq q)(c) = c.$$

Note that tri is contravariant in the first argument and covariant on the second argument. This mixed variance reflects that the pre-condition of a triple can be strengthened, and the post-condition can be weakened; thus, it validates the subtyping rule for Consequence. We also note that the subtyping rule that moves an invariant assertion into the pre- and post-conditions is sound.

Lemma 4 *For each predicate p, let $- * p : \mathcal{P} \to \mathcal{P}$ be a functor that maps a predicate q to $q * p$. Then,*

$$\mathsf{inv}(-, p) \circ \mathsf{tri} = \mathsf{tri}(- * p, - * p).$$

The category $\mathcal{D}$ is obtained from $\mathcal{C}$ by equating morphisms according to an equivalence relation $\sim$. Morphisms f and g in $\mathcal{C}[(A, R), (B, S)]$ are related by $\sim$ iff

$$\forall p \in Pred.\, f[R(p) \to S(p)]g.$$

Relation $\sim$ is an equivalence relation; it is reflexive, because each morphism in $\mathcal{C}[(A, R), (B, S)]$ should map $R(p)$-related elements to $S(p)$-related elements, for all p; and it is symmetric and transitive because, for all p, $R(p)$ and $S(p)$ are symmetric and transitive. The interesting property of $\sim$ is that it is preserved by all the structure of $\mathcal{C}$:

Lemma 5 (Preservation) *The relation $\sim$ is preserved by the following operators in $\mathcal{C}$:*

- *the composition of morphisms;*
- *the currying of morphisms;*
- *the pairing into all the small products; and*
- *the functor $\mathsf{inv}(-, p \sqsubseteq q)$ on $\mathcal{C}$, for all predicates p, q such that $p \sqsubseteq q$.*

This lemma ensures that taking a quotient of morphisms in $\mathcal{C}$ gives a well-defined category, which we call $\mathcal{D}$. Category $\mathcal{D}$ inherits all the interesting structure of $\mathcal{C}$ by Lemma 5; it is cartesian closed, has all small products, and has a functor

$\mathsf{inv}' : \mathcal{D} \times \mathcal{P}_r \to \mathcal{D}$ that preserves the CCC structure and the small products of $\mathcal{D}$. Let E be the "quotienting" functor from $\mathcal{C}$ to $\mathcal{D}$, and $\mathsf{tri}' : \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \to \mathcal{D}$ the composition of E with tri. We summarize the main property of $\mathcal{D}$ in the following two lemmas:

Lemma 6 *The category $\mathcal{D}$ is a CCC with all small products, and has two functors $\mathsf{inv}' : \mathcal{D} \times \mathcal{P}_r \to \mathcal{D}$ and $\mathsf{tri}' : \mathcal{P}^{\mathsf{op}} \times \mathcal{P} \to \mathcal{D}$ such that*

1. *$\mathsf{inv}'(-, p)$ preserves all the CCC structure and the small products of $\mathcal{D}$;*
2. *$\mathsf{inv}'(-, p) \circ \mathsf{inv}'(-, q) = \mathsf{inv}'(-, p * q)$; and*
3. *$\mathsf{inv}'(-, p) \circ \mathsf{tri}' = \mathsf{tri}'(- * p, - * p)$.*

Lemma 7 *The functor E from $\mathcal{C}$ to $\mathcal{D}$ is full, preserves the CCC structure as well as small products, and makes the following diagrams commute:*

$$\begin{array}{ccc} \mathcal{C} \times \mathcal{P}_r & \xrightarrow{\mathsf{inv}} & \mathcal{C} \\ {\scriptstyle E \times \mathsf{Id}}\downarrow & & \downarrow{\scriptstyle E} \\ \mathcal{D} \times \mathcal{P}_r & \xrightarrow{\mathsf{inv}'} & \mathcal{D} \end{array} \qquad \begin{array}{ccc} \mathcal{P}^{\mathsf{op}} \times \mathcal{P} & \xrightarrow{\mathsf{tri}} & \mathcal{C} \\ {\scriptstyle \mathsf{Id}}\downarrow & & \downarrow{\scriptstyle E} \\ \mathcal{P}^{\mathsf{op}} \times \mathcal{P} & \xrightarrow{\mathsf{tri}'} & \mathcal{D} \end{array}$$

4.2. Interpretation of the Language

We interpret the language in the family fibration $Fam(\mathcal{D}) \to \mathsf{Set}$. Each base set in the fibration models all the possible environments for a fixed shape of the stack (i.e., a fixed set of integer variables Δ). For instance, the object $\{(A, R)_\eta\}_{\eta \in [\![\Delta]\!]}$ assumes that all the available integer variables are in Δ, and it specifies a type dependent on the values of such variables, given by η. The types and terms of our language are interpreted using the categorical structure of this fibration.

The interpretation is explicit about the set of variables under which we consider types, type assignments, and terms. Write $\Delta \vdash \Gamma$ to mean that $\Delta \vdash \Gamma(x) : \mathsf{Type}$, for all x in the domain of Γ.

The semantics of $\Delta \vdash \theta(: \mathsf{Type})$ and $\Delta \vdash \Gamma$ is given by a family of objects in $\mathcal{D}$ indexed by the environments in $[\![\Delta]\!]$. The precise definition of $[\![\theta]\!]$ and $[\![\Gamma]\!]$ is given as follows: for η in $[\![\Delta]\!]$,

$$\begin{array}{rcl} [\![\Delta \vdash \{P\}{-}\{Q\}]\!]_\eta & = & \mathsf{tri}'([\![\Delta \vdash P]\!]_\eta, [\![\Delta \vdash Q]\!]_\eta) \\ [\![\Delta \vdash \theta \otimes P]\!]_\eta & = & \mathsf{inv}'([\![\Delta \vdash \theta]\!]_\eta, [\![\Delta \vdash P]\!]_\eta) \\ [\![\Delta \vdash \theta \to \theta']\!]_\eta & = & [\![\Delta \vdash \theta]\!]_\eta \Rightarrow [\![\Delta \vdash \theta']\!]_\eta \\ [\![\Delta \vdash \Pi_i \theta]\!]_\eta & = & \Pi_{n \in Int} [\![\Delta \cup \{i\} \vdash \theta]\!]_{\eta[i \to n]} \\ [\![\Delta \vdash \Gamma]\!]_\eta & = & \Pi_{x \in \mathsf{dom}(\Gamma)} [\![\Delta \vdash \Gamma(x)]\!]_\eta \end{array}$$

Note that tri' is used to interpret the triple type $\{P\}{-}\{Q\}$, and inv' to interpret the invariant extension $\theta \otimes P$.

We interpret each subtype relation $\theta \preceq_\Delta \theta'$ as a family of morphisms in $\mathcal{D}$ of the following shape:

$$\{f_\eta : [\![\Delta \vdash \theta]\!]_\eta \to [\![\Delta \vdash \theta']\!]_\eta\}_{\eta \in [\![\Delta]\!]}.$$

The semantics is given by induction on the derivation. The subtyping rule for Consequence is interpreted using the morphism action of functor tri':

$$\begin{array}{l} [\![\{P'\}{-}\{Q'\} \preceq_\Delta \{P\}{-}\{Q\}]\!]_\eta \\ = \mathsf{tri}'([\![P]\!]_\eta \subseteq [\![P']\!]_\eta, [\![Q']\!]_\eta \subseteq [\![Q]\!]_\eta) \end{array}$$

The other important subtyping rules are the ones for the higher-order frame rules. The interpretation of these rules uses the property of inv'. The generalized frame rule is interpreted by the morphism action of inv':

$$[\![\theta \preceq_\Delta \theta \otimes P]\!]_\eta = \mathsf{inv}'([\![\theta]\!]_\eta, emp \sqsubseteq [\![P]\!]_\eta)$$

and the rules for distributing an invariant is interpreted by the identity; this interpretation "typechecks," because $\mathsf{inv}'(-, p)$ preserves the exponentials and the small products on the nose, and because of (items 2 and 3 of) Lemma 6.

Finally, we define the semantics of each typing judgment $\Gamma \vdash_\Delta M : \theta$ by an indexed family of morphisms in $\mathcal{D}$ of the form:

$$\{f_\eta : [\![\Delta \vdash \Gamma]\!]_\eta \to [\![\Delta \vdash \theta]\!]_\eta\}_{\eta \in [\![\Delta]\!]}.$$

The semantics is given by induction on the derivation of the judgment, and it is shown in Figure 2. The interpretation of terms is given using the categorical structure of $\mathcal{D}$ in a standard way. The only specific parts are the interpretation of basic imperative operations, where we use five basic semantic constants

$$seq,\ new,\ read,\ free,\ \text{and}\ write,$$

which are also defined in the figure.

For this interpretation of terms, the question of well-definedness arises, because of the introduction and elimination of dependent function type $\Pi_i \theta$. The semantic definition of $\lambda i.M$ assumes that if Γ does not contain the variable i, it is interpreted as the same object in $\mathcal{D}$ no matter how we change or even drop the value of i in the index. The definition of $[\![ME]\!]$ assumes that the reindexing precisely models the substitution. The following lemmas show that these two assumptions indeed hold.

Lemma 8 *If $\mathsf{fiv}(\Gamma) \subseteq \Delta$, then*

$$\forall \eta \in [\![\Delta]\!]. \forall n \in Int.\ [\![\Delta \vdash \Gamma]\!]_\eta = [\![\Delta \cup \{i\} \vdash \Gamma]\!]_{\eta[i \to n]}.$$

Lemma 9 *If $\mathsf{fiv}(\theta) \subseteq \Delta \cup \{i\}$ and $\mathsf{fiv}(E) \subseteq \Delta$, then*

$$\forall \eta \in [\![\Delta]\!]. [\![\Delta \vdash \theta[E/i]]\!]_\eta = [\![\Delta \cup \{i\} \vdash \theta]\!]_{\eta[i \to [\![E]\!]_\eta]}.$$

$$
\begin{array}{rcl}
[\![\Gamma, x:\theta \vdash_\Delta x:\theta]\!]_\eta & = & \pi_x \\
[\![\Gamma \vdash_\Delta \lambda x{:}\theta.M:\theta\to\theta']\!]_\eta & = & \mathsf{curry}([\![\Gamma, x{:}\theta \vdash_\Delta M:\theta']\!]_\eta \circ \mathsf{iso}([\![\Gamma]\!]_\eta \times [\![\theta]\!]_\eta, [\![\Gamma, x{:}\theta]\!]_\eta)) \\
[\![\Gamma \vdash_\Delta MM':\theta]\!]_\eta & = & \mathsf{ev} \circ \langle [\![\Gamma \vdash_\Delta M:\theta'\to\theta]\!]_\eta, [\![\Gamma \vdash_\Delta M':\theta']\!]_\eta\rangle \\
[\![\Gamma \vdash_\Delta \lambda i.M{:}\Pi_i\theta]\!]_\eta & = & \langle [\![\Gamma \vdash_{\Delta\cup\{i\}} M{:}\theta]\!]_{\eta[i\to n]}\rangle_{n\in Int} \\
[\![\Gamma \vdash_\Delta ME{:}\theta[E/i]]\!]_\eta & = & \pi_{[\![E]\!]_\eta} \circ [\![\Gamma \vdash_\Delta M{:}\Pi_i\theta]\!]_\eta \\
[\![\Gamma \vdash_\Delta M:\theta']\!]_\eta & = & [\![\theta \preceq_\Delta \theta']\!]_\eta \circ [\![\Gamma \vdash_\Delta M:\theta]\!]_\eta \\
[\![\Gamma \vdash_\Delta \mathsf{fix}\, M{:}\theta]\!]_\eta & = & \mathsf{lfix} \circ [\![\Gamma \vdash_\Delta M{:}\theta\to\theta]\!]_\eta \\
[\![\Gamma \vdash_\Delta \mathsf{ifz}\, E\, M\, M'{:}\{P\}{-}\{Q\}]\!]_\eta & = & \mathsf{if}\ ([\![\Delta \vdash E]\!]_\eta = 0)\ \mathsf{then}\ [\![\Gamma \vdash_\Delta M{:}\{P\wedge E{=}0\}{-}\{Q\}]\!]_\eta \\
 & & \qquad\qquad \mathsf{else}\ [\![\Gamma \vdash_\Delta M'{:}\{P\wedge E{\neq}0\}{-}\{Q\}]\!]_\eta \\
[\![\Gamma \vdash_\Delta M; M'{:}\{P\}{-}\{Q\}]\!]_\eta & = & \mathit{seq} \circ \langle [\![\Gamma \vdash_\Delta M{:}\{P\}{-}\{P'\}]\!]_\eta, [\![\Gamma \vdash_\Delta M'{:}\{P'\}{-}\{Q\}]\!]_\eta\rangle \\
[\![\Gamma \vdash_\Delta \mathsf{let}\, i = \mathsf{new}\, \mathsf{in}\, M : \{P\}{-}\{Q\}]\!]_\eta & = & \mathit{new} \circ \langle [\![\Gamma \vdash_{\Delta\cup\{i\}} M : \{P * i\mapsto -\}{-}\{Q\}]\!]_{\eta[i\to n]}\rangle_{n\in Int} \\
[\![\Gamma \vdash_\Delta \mathsf{let}\, i = [E]\, \mathsf{in}\, M : \{P * E\mapsto -\}{-}\{Q\}]\!]_\eta & = & \mathit{read}([\![\Delta \vdash E]\!]_\eta) \circ \langle [\![\Gamma \vdash_{\Delta\cup\{i\}} M : \{P * E\mapsto i\}{-}\{Q\}]\!]_{\eta[i\to n]}\rangle_{n\in Int} \\
[\![\Gamma \vdash_\Delta \mathsf{free}(E) : \{E\mapsto -\}{-}\{\mathsf{emp}\}]\!]_\eta & = & \mathit{free}([\![\Delta \vdash E]\!]_\eta) \circ !_{[\![\Delta\vdash\Gamma]\!]_\eta} \\
[\![\Gamma \vdash_\Delta [E] := E' : \{E\mapsto -\}{-}\{E\mapsto E'\}]\!]_\eta & = & \mathit{write}([\![\Delta \vdash E]\!]_\eta, [\![\Delta \vdash E']\!]_\eta) \circ !_{[\![\Delta\vdash\Gamma]\!]_\eta}
\end{array}
$$

where seq, new, $\mathit{read}(n)$, $\mathit{free}(n)$, and $\mathit{write}(n,n')$ are defined as a morphism in $\mathcal{D}$ (which is an equivalence class) as follows:

$$
\begin{array}{rcl}
\mathit{seq}_{p,q,q'} & : & \mathsf{tri}(p,q) \times \mathsf{tri}(q,q') \to \mathsf{tri}(p,q') \\
\mathit{seq} & = & [\lambda(c,c').\, \lambda h.\, \{\mathit{wrong} \mid \mathit{wrong} \in c(h)\} \cup \bigcup\{c'(h') \mid h' \in c(h)\}] \\
\mathit{new}_{p,q} & : & (\Pi_{n\in Int}\mathsf{tri}(p * n \mapsto -, q)) \to \mathsf{tri}(p,q) \\
\mathit{new} & = & [\lambda c.\, \lambda h.\, \bigcup\{c(n)(h[n\mapsto m]) \mid m,n \in Int\ \wedge\ n \notin \mathsf{dom}(h)\}] \\
\mathit{read}(n)_{p,q} & : & (\Pi_{m\in Int}\mathsf{tri}(p * n \mapsto m, q)) \to \mathsf{tri}(p * n \mapsto -, q) \\
\mathit{read}(n) & = & [\lambda c.\, \lambda h.\, \mathsf{if}\ n \in \mathsf{dom}(h)\ \mathsf{then}\ c(h(n))(h)\ \mathsf{else}\ \{\mathit{wrong}\}] \\
\mathit{free}(n) & : & 1 \to (\{n\mapsto -\}{-}\{\mathit{emp}\}) \\
\mathit{free}(n) & = & [\lambda x.\, \lambda h.\, \mathsf{if}\ n \in \mathsf{dom}(h)\ \mathsf{then}\ \{h[n{\to}\mathit{undef}]\}\ \mathsf{else}\ \{\mathit{wrong}\}] \\
\mathit{write}(n,n') & : & 1 \to (\{n\mapsto -\}{-}\{n\mapsto n'\}) \\
\mathit{write}(n,n') & = & [\lambda x.\, \lambda h.\, \mathsf{if}\ n \in \mathsf{dom}(h)\ \mathsf{then}\ \{h[n{\to}n']\}\ \mathsf{else}\ \{\mathit{wrong}\}]
\end{array}
$$

Figure 2. Interpretation of Terms

4.3. Adequacy

Our semantics of terms needs further justification in two ways. First, the interpretation of a typing judgment needs to be shown coherent. The interpretation is defined over a proof derivation of the judgment, so two different derivations of the same judgment might have different denotations. This is troublesome for us especially, because our goal is to give a semantics of a programming language with a separation-logic type system, instead of a semantics of a proof in separation logic. Second, the connection with the standard semantics needs to be provided. Our semantics uses subsumption which never arises in the standard interpretation. Thus, our interpretation could be substantially different from the standard interpretation. In this section, we provide justification for both of these two issues.

We consider two other interpretations of our language. The first interpretation, called the standard interpretation, ignores all assertions in the types. In the standard interpretation, $\{P\}{-}\{Q\}$ means the same thing no matter what P and Q are, and for all P, $\theta \otimes P$ and θ have identical interpretations. Let tri'' be the constant functor from $\mathcal{P}^{\mathsf{op}} \times \mathcal{P}$ to CPO such that $\mathsf{tri}''(p,q) = \mathsf{comm}$, and let inv'' be a functor given by the first projection from $\mathsf{CPO} \times \mathcal{P}_r$ to CPO. Then, the standard interpretation is the interpretation of the previous section, where we use CPO, tri'' and inv'' instead of $\mathcal{D}$, tri' and inv', and we take an equivalence class representative in the definition of constants; for instance, the constant $\mathit{read}(n)$ now means

$$\lambda c.\, \lambda h.\, \mathsf{if}\ n \in \mathsf{dom}(h)\ \mathsf{then}\ c(h(n))(h)\ \mathsf{else}\ \{\mathit{wrong}\}.$$

The second interpretation uses the category $\mathcal{C}$. It is obtained by simply replacing tri' and inv' in the previous section by tri and inv, and eliminating the embedding $[-]$ to the equivalence class in the definition of constants.

The three interpretations are very closely related. Note that from the category $\mathcal{C}$ to CPO, there is a forgetful functor F that maps an object (A,R) to A, and a morphism f to f. This forgetful functor preserves all the categorical structure of $\mathcal{C}$ that we use to interpret the types of our language:

Lemma 10 *F is a faithful functor that preserves the CCC structure and the small products of $\mathcal{C}$, and makes the following diagrams commute.*

$$
\begin{array}{ccc}
\mathcal{C} \times \mathcal{P}_r & \xrightarrow{\mathsf{inv}} & \mathcal{C} \\
{\scriptstyle F\times \mathsf{Id}}\downarrow & & \downarrow{\scriptstyle F} \\
\mathsf{CPO} \times \mathcal{P}_r & \xrightarrow{\mathsf{inv}''} & \mathsf{CPO}
\end{array}
\qquad
\begin{array}{ccc}
\mathcal{P}^{\mathsf{op}} \times \mathcal{P} & \xrightarrow{\mathsf{tri}} & \mathcal{C} \\
{\scriptstyle \mathsf{Id}}\downarrow & & \downarrow{\scriptstyle F} \\
\mathcal{P}^{\mathsf{op}} \times \mathcal{P} & \xrightarrow{\mathsf{tri}''} & \mathsf{CPO}
\end{array}
$$

Note that Lemmas 5 and 10 imply that the interpretation of the types in $\mathcal{D}$ and CPO factors through the interpretation in $\mathcal{C}$. The following lemma shows that the interpretation of the terms has a similar property.

Lemma 11 *Both the forgetful functor F from $\mathcal{C}$ to CPO and the quotienting functor E from $\mathcal{C}$ to $\mathcal{D}$ preserve the interpretation of terms. That is, for all typing judgments $\Gamma \vdash_\Delta M\colon\theta$ and all $\eta \in [\![\Delta]\!]$,*

$$F([\![\Gamma \vdash_\Delta M\colon\theta]\!]^{\mathcal{C}}_\eta) = [\![\Gamma \vdash_\Delta M\colon\theta]\!]^{\mathsf{CPO}}_\eta$$
$$E([\![\Gamma \vdash_\Delta M\colon\theta]\!]^{\mathcal{C}}_\eta) = [\![\Gamma \vdash_\Delta M\colon\theta]\!]^{\mathcal{D}}_\eta.$$

Since E is full and F is faithful, intuitively, Lemma 11 says that our semantics is obtained by first selecting some elements, and then quotienting those selected elements.

Corollary 12 *Our semantics is coherent: the semantics of a typing judgment does not depend on derivations.*

Proof: Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two derivations of a judgment $\Gamma \vdash_\Delta M : \theta$. We note that the standard semantics is coherent; only the subsumption rule is not syntax-directed, but in the standard semantics, this rule does not contribute to the interpretation, because all the subtyping rules denote the identity function. Thus, for all environments η, we have

$$[\![\mathcal{P}_1]\!]^{\mathsf{CPO}}_\eta = [\![\mathcal{P}_2]\!]^{\mathsf{CPO}}_\eta.$$

Then, by Lemma 11 and the faithfulness of F,

$$\begin{array}{rcl} [\![\mathcal{P}_1]\!]^{\mathsf{CPO}}_\eta = [\![\mathcal{P}_2]\!]^{\mathsf{CPO}}_\eta & \Longrightarrow & F([\![\mathcal{P}_1]\!]^{\mathcal{C}}_\eta) = F([\![\mathcal{P}_2]\!]^{\mathcal{C}}_\eta) \\ & \Longrightarrow & [\![\mathcal{P}_1]\!]^{\mathcal{C}}_\eta = [\![\mathcal{P}_2]\!]^{\mathcal{C}}_\eta \\ & \Longrightarrow & E([\![\mathcal{P}_1]\!]^{\mathcal{C}}_\eta) = E([\![\mathcal{P}_2]\!]^{\mathcal{C}}_\eta) \\ & \Longrightarrow & [\![\mathcal{P}_1]\!]^{\mathcal{D}}_\eta = [\![\mathcal{P}_2]\!]^{\mathcal{D}}_\eta \end{array}$$

□

5. Related Work

The (first order) frame rule was discovered in the early days of separation logic [5], and it was a main reason for the success of that logic. For example, it was vital in the proofs of garbage collection algorithms in [22] and [4]. Recently, the second-order frame rule, which allows reasoning about simple first-order modules, was discovered [10]. This naturally encouraged the question of whether there are more general frame rules that apply to higher types.

Unlike in [10], our soundness result of higher-order frame rules does not require that invariant assertions be *precise*. This preciseness requirement is needed because higher-order frame rules interact badly with the conjunction rule [10]. In our system, such bad interaction does not exist, because the system does not have the conjunction rule.

Other type systems which track state changes have been proposed in the work on typed assembly languages [7, 2, 21]. Their main focus is to obtain sound rules for proving the safety of programs. Thus, they mostly use easy-to-define conventional operational semantics, and prove the soundness of the proof system syntactically (i.e., by subject reduction and progress lemmas), or "logically" [21]: each type is interpreted as a subset of a single universe of "meanings," and a typing judgment is interpreted as a specification for the behavior of programs, like a Hoare triple in separation logic. Our separation-logic type system is more refined in that it allows the full power of separation logic in the types and, moreover, we also treat higher-order procedures.

The semantics of idealized algol has been studied intensively [11, 15, 8, 14]. Normally, the semantics is parameterized by the shape of the memory. The indexing in the fibration in our semantics follows this tradition, and it models the shape of the stack. However, the other indexing of our semantics, the indexing by invariant predicates over heaps, has not been used in the literature before.

The construction of the category $\mathcal{D}$ is an instance of the Kripke quotient by Mitchell and Moggi [6]. The families of pers in $\mathcal{D}$ form a Kripke logical relation on CPO indexed by the preorder category $\mathcal{P}_r$; our condition on each family ensures that the requirement of Kripke monotonicity holds. This Kripke logical relation produces $\mathcal{D}$ by Mitchell and Moggi's construction.

The idea of proving coherence by relating two languages comes from Reynolds [19]. Reynolds proved the coherence of the semantics of typed lambda calculus with subtyping, by connecting it with the semantics of untyped lambda calculus. We use the general direction of Reynolds's proof, but the details of our proof are quite different from Reynolds's, because we consider very different languages.

6. Conclusion and Future Directions

We have presented a type system for idealized algol extended with heaps that includes separation-logic specifications as types and, moreover, defined the coherent semantics of idealized algol typed with this system.

One shortcoming of our type system is that the higher-order frame rules in the system allow only static modularity [12]. With the higher-order frame rules alone, we cannot capture all the the information hiding aspect of dynamically allocated data structures as needed for modeling abstract data types. However, it is well-known that abstract data types can be modeled using existential types and we are currently considering to enrich the assertion language with predicate variables, as in the recently introduced higher-order version of separation logic [3], and to extend the types with dependent product and sums over *predicates*.

Another shortcoming of the type system is that it cannot have the disjunction rule in separation logic. The disjunction rule has two judgments in the premise, and requires

that both judgments are about the same program. Such a requirement about the same program cannot be expressed in our type system. We plan to overcome this problem by extending the type system with intersection types [17].

Yet another future direction is to define a parametric model. Uday Reddy pointed out that separation-logic types should validate stronger reasoning principles for data abstraction than ordinary types, because they let us control what clients can access more precisely. Formalizing his intuition is the goal of the parametricity semantics. We currently plan to use category $\mathcal{C}'$ which replaces each *predicate-indexed* family of *pers* in $\mathcal{C}$ by a *relation-indexed* family of *saturated relations*: an object in $\mathcal{C}'$ is a cpo paired with a family T of *binary* relations such that (1) T is indexed by a "typed" relation $r\colon p \leftrightarrow q$ on heaps (i.e., $r \subseteq p \times q$); (2) for each predicate p, T at the diagonal relation Δ_p is a per; (3) for all $r\colon p \leftrightarrow q$, $T(r)$ is a saturated relation between pers $T(\Delta_p)$ and $T(\Delta_q)$; (4) $T(r) \subseteq T(r * r')$. The morphisms in $\mathcal{C}'$ are continuous functions that preserve the families of relations. This category has all the categorical structure of $\mathcal{C}$ that we used in the semantics of this paper. However, it is difficult to interpret the triple types such that the memory allocator new live in the category. Overcoming this problem will be the focus of our research in this direction.

Finally, we would like to extend the relational separation logic [23] to higher-order, following the style of system R [1], and we want to explore the Curry-Howard correspondence of our type system with specification logic [16].

Acknowledgements

We have benefitted greatly from discussions with Uday Reddy and Peter O'Hearn. Yang was supported by grant No. R08-2003-000-10370-0 from the Basic Research Program of the Korea Science & Engineering Foundation. Yang, Birkedal and Torp-Smith were supported by Danish Technical Research Council Grant 56-00-0309.

References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Comput. Sci.*, 121(1-2):9–58, December 1993.

[2] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *Proc. of LICS'03*, Ottawa, Canada, June 2003.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP'05*, Edinburgh, Scotland, April 2005.

[4] L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. of POPL'04*, pages 220 – 231, Venice, Italy, 2004.

[5] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL'01*, London, England, January 2001.

[6] J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Appl. Logic*, 51:99–124, 1991.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. and Syst.*, 21(3):527 – 568, 1999.

[8] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.

[9] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Local reasoning about programs that alter data structures. In *Proc. of CSL'01*, pages 1 – 19, Paris, France, September 2001.

[10] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL'04*, pages 268 – 280, Venice, Italy, 2004.

[11] F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, 1982.

[12] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Proc. of POPL'05*, Long Beach, CA, USA, January 2005.

[13] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] U. Reddy and H. Yang. Correctness of data representations involving heap data structures. *Science of Computer Programming*, 50(1):129 – 160, March 2004.

[15] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[16] J. C. Reynolds. Idealized Algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge University Press, 1982.

[17] J. C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28, 1996.

[18] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, Houndsmill, Hampshire, 2000.

[19] J. C. Reynolds. The meaning of types — from intrinsic to extrinsic semantics. Research Series RS–00–32, BRICS, DAIMI, Department of Computer Science, University of Aarhus, December 2000. http://www.brics.dk/RS/00/32/.

[20] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55 – 74, Copenhagen, Denmark, 2002.

[21] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In *Proc. of VMCAI '04*, Venice, Italy, January 2004.

[22] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.

[23] H. Yang. Relational separation logic. Submitted to Theoretical Comput. Sci., October 2004.