ELSEVIER

Theoretical Computer Science 278 (2002) 271–301

Theoretical Computer Science

www.elsevier.com/locate/tcs

# Soundness of data refinement for a higher-order imperative language

David A. Naumann

*Computer Science, Stevens Institute of Technology, Hoboken, NJ 07030, USA*

**Abstract**

Using a set-theoretic model of predicate transformers and ordered data types, we give a semantics for an Oberon-like higher-order imperative language with record subtyping and procedure-type variables and parameters. Data refinement is shown to be sound for this language: It implies algorithmic refinement when suitably localized. All constructs are shown to preserve simulation, so data refinement can be carried out piecewise. © 2002 Elsevier Science B.V. All rights reserved.

*Keywords:* Program correctness; Programming Calculi; Data refinement; Formal semantics; Predicate transformers

## 1. Introduction

Data refinement is a method of verifying correctness of data representations by means of simulation relations. Soundness of the method means that if there is a simulation from one representation to another, a program's correctness is preserved when one representation is replaced by the other. Soundness has not been seriously questioned and proofs have appeared for first-order imperative languages [10] and for higher-order functional languages [12]. But proofs of soundness have not appeared for higher-order imperative languages because of problems in the semantics of local variables and higher-order procedures. The problems are close to being solved for what are known as Algol-like languages, and soundness has been announced for Algol (see below). The difficulties are not so acute when higher-order procedures are restricted as in what we call *Oberon-like languages*, *e.g.* Oberon, C, Modula-3, C++, and Ada. The key restriction is that if a procedure is passed as an argument then its external variables are declared in the outermost scope. Such languages go beyond Algol in having stored

*E-mail address:* naumann@cs.stevens-tech.edu (D.A. Naumann).


PII: S0304-3975(00)00339-X

procedures, again subject to the restriction on external variables. (They also differ in not using call-by-name.)

This paper shows soundness of data refinement for an Oberon-like language with record subtyping and non-determinacy. The language differs from the ones mentioned in that reference types are omitted and aliasing is restricted in the usual ways needed for proof rules: Result arguments are required to be distinct from each other and from externals of called procedures. These restrictions make it possible to give a total-correctness predicate-transformer semantics that extends the usual semantics for first-order programs and also models specification constructs for a higher-order refinement calculus. Our semantics has been used to prove refinement laws and rules for reasoning about procedure calls, non-interference, and record subtyping, allowing higher-order predicates in specifications [25]. The semantics also justifies program equivalences like those used to test semantics of Algol [14]; two such laws are shown in the sequel using data refinement. This does not mean that we solve the problems of Algol; instead, our results show that Oberon-like languages are sufficiently different from Algol that they admit a simple model adequate for reasoning about data refinement. Indeed, a conventional Scott–Strachey semantics could also be given and results like ours should be provable using logical relations; but we argue in the sequel that such semantics is inadequate for a calculus of program refinement.

The rest of this section explains data refinement and introduces the language. Section 2 surveys what we need of the calculus of relations and predicate transformers (*transformers* henceforth). Section 3 gives the syntax of the language, except for extensible records and subtyping which are deferred to streamline the exposition. Section 4 defines the semantics of data types, along with notations for predicates and transformers. Section 5 gives the semantics of expressions and commands. Section 6 is an intermezzo on specification constructs, motivating our semantic model. Soundness and preservation of data refinement is shown in Section 8, using couplings defined in Section 7. Section 9 extends the language to include extensible records, and extends the above results. Applications are in Section 10 and concluding remarks in Section 11.

A less-condensed presentation of our idealized Oberon [31], called *Io* after another planetary moon, can be found in [25]. An introduction to data refinement can be found in, e.g. [16]; a clear and concise account appears in [30] which also explains a semantics that can be used to show soundness of data refinement in an Algol-like language.[1] For first-order imperative programs without procedures, soundness of data refinement has been shown using both state-transformer [10] and predicate-transformer semantics [7]. For simply typed higher-order functional programs, soundness and completeness results appear in [12] and references cited therein.

[1] The language of [30] is quite different from ours: Only zero-order types like **int** can be stored, thus parameters are passed by name but not by value. The emphasis is on program equivalence rather than refinement. Although the paper is expository, neither stating nor proving a result on soundness of data refinement, such a result has been proved and can be extended to some form of program refinement (Tennant, pers. comm.).

Table 1
Summary of notations; rows ordered by decreasing binding power

| | |
|---|---|
| $(x :\approx f)(x \mapsto f)$ | (postfix) substitution, overriding (bind tightly) |
| $\cdot$ | (left associative) function application |
| $\circ :$ | function composition, typing |
| , | listing |
| $\downharpoonleft \dagger$ | function restriction, predicate lifting |
| ; | composition |
| $\otimes \cup \times \to \sim \approx >$ | set-theoretic operations |
| $= \in \subseteq \sqsubseteq R$ | infix binary relations |
| $\wedge \vee (\forall x : - : -)$ | logic (meta-language) |
| $\equiv \Rightarrow$ | logic (bind least tightly) |

An account of data refinement begins with program refinement, also called *algorithmic refinement*, because data refinement is a means to design and verify an algorithmic refinement that is accomplished by a change of data representation. Unqualified, *refinement* means the relation of algorithmic refinement, which can only hold between programs of the same type – i.e. in the same state space, for imperative programs. In previous work on higher-order programs [12, 30], refinement is taken to mean equality, but improvement by increasing determinacy or termination is of more interest in practice. For $g$ to refine $f$ means that $g$ meets all specifications that $f$ does, and in this paper that means total-correctness (pre-post) specifications. In our semantics, commands denote (predicate) transformers, for which refinement coincides with the pointwise order: $f \sqsubseteq g$ is defined to mean $(\forall \varphi :: f.\varphi \subseteq g.\varphi)$. Infix dot denotes function application, and the range of quantification is written between colons or omitted. Table 1 summarizes notations used in the sequel.

The main advantage of transformers over other models is that they can be used for semantics of specification constructs (prescriptions and angelic variables) [16] that internalize correctness statements in the sense that a pre-post pair $\varphi, \psi$ determines an "imaginary" or "infeasible" program that is the greatest lower bound among programs satisfying $\varphi, \psi$. Such constructs can be used to internalize conditional data refinement [4, 17], i.e. data refinement in the context of a specification, which is what is often needed in practice. The full language of [25] includes these constructs, which is why we have chosen a transformer semantics – see Section 6. In the rest of the paper we omit these constructs so we can omit the syntax and semantics of predicate formulas. The specification constructs are not type-constructors, and it is straightforward to show soundness and preservation for them in our model.

Development by data refinement means replacing a data type $D$ in program $p$ by a type $D'$ (typically, more concrete or efficient) in a program $p'$ using operations on $D'$ in place of those on $D$. Customers are seldom keen on video displays being replaced by paper tapes, so the method is applied when $D$ and $D'$ are not the types of observable variables. Typically, they are encapsulated in modules, but for expository purposes it is simpler to ignore modules and consider local variables (as in the cited work on data

refinement). Data refinement of $p$ to $p'$ is used to achieve an algorithmic refinement

$$(\mathbf{var}\, x : D \bullet p) \sqsubseteq (\mathbf{var}\, x' : D' \bullet p'). \tag{1}$$

In Io syntax, $p$ has type $\mathbf{com}(\pi, x : D)$ for typing $\pi$ of variables other than $x$, and $p'$ has type $\mathbf{com}(\pi, x' : D')$. Both sides of (1) have type $\mathbf{com}(\pi)$, i.e. they are commands in state space $\pi$.

The method for proving (1) is to define a *coupling* relation $R$ connecting the two state spaces so the programs $p, p'$ can be compared in terms of the touchstone, algorithmic refinement. For the visible coordinates $\pi$, $R$ should be the identity, but not necessarily so for $x$ and $x'$. Let us ignore variables for a moment and consider the case where $p, p'$ are functions $p : A \to A$ and $p' : A' \to A'$. Then we say $p'$ *simulates* $p$ *by coupling* $R \subseteq A \times A'$ if

$$(\forall a, a' : a\ R\ a' :\ p.a\ R\ p'.a'). \tag{2}$$

Function application binds more tightly than infix relations, so this is to be parsed as $(p\,.\,a)\,R\,(p'\,.\,a')$. Property (2) is equivalent to the following inclusion:

$$p; R \supseteq R; p' \qquad \begin{array}{ccc} & \xrightarrow{\quad p' \quad} & \\ R\uparrow & \subseteq & \uparrow R \\ & \xrightarrow{\quad p \quad} & \end{array} \tag{3}$$

(We write ";" for composition of relations, including functions.) A special case is a representation function, i.e. a coupling that is a function $h$ from concrete to abstract values; in that case $R$ is the converse of $h$.

If $p, p'$ are binary relations viewed as non-deterministic programs, then $\supseteq$ represents algorithmic refinement by increasing determinacy; (3) can still be interpreted as simulation of $p$ by $p'$. Data refinement of transformers is defined in a similar way below (see (16) and (28)). The soundness theorem for data refinement says, roughly, that to prove (1) it suffices to find $R$ that is the identity on $\pi$ and satisfies (3). This is not quite true, because (1) omits initialization of variables. There are also problems if procedure types are included among the observable types, as discussed later. The exact soundness result for Io is Theorem 4.

In practice, $p'$ is obtained from $p$ by simulating just the primitive operations on $D$ with operations on $D'$. Data refinement can be carried out in this "piecewise" manner only if simulation of a part implies simulation of the whole. This is analogous to algorithmic refinement, with respect to which we want program constructs to be monotonic. (In the case that algorithmic refinement is just equality, this is compositionality of semantics.) It is well established that the standard first-order imperative constructs

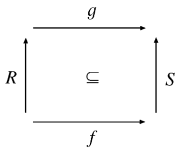
*preserve simulations*, as do the pure functional programming constructs. All constructs of Io preserve simulation (Theorem 5).

To prove that program constructs preserve simulation, we need to obtain couplings of constructed types from couplings on their constituent types. (This is not explicit in some treatments of data refinement of transformers, where the issue is hidden by the syntactic view of predicates and the use of a single fixed state space.) For example, from relation $R_D : D \to D'$ the relation $R_{\pi,x:D} : (\pi, x : D) \to (\pi, x : D')$ needed for (1) is defined by pairing with the identity on $\pi$.

The coupling for the function-space construct is based on the observation that the equivalence of (2) and (3) generalizes to differing types, i.e.

$$f ; S \supseteq R ; g \equiv (\forall x, y : x \ R \ y : f.x \ S \ g.y) \tag{4}$$

for all relations $R, S$ and functions $f, g$ in configuration




(5)

This indicates that for the data type of functions the coupling should relate $f$ to $g$ just if (5), and that is the definition of *logical relation*. Logical relations have been studied extensively in the context of functional languages [15] and for relational parametricity which is a key element in semantics of Algol [26, 30]. To deal with a wider variety of program paradigms and constructs, there are general results that give sufficient conditions for preservation of simulation; the conditions are expressed in the language of ordered categories [13] or 2-categories [18]. Here we do not use results or definitions from that theory,[2] although it is implicit in some proofs in the sequel.[3]

For our purposes it is convenient to take the following view of simulation. Type $D$ and primitive operations on it are "built in" to the language. Instead of $D, D'$ in (1) we have a single type $D$ with two interpretations $[\![D]\!]$ and $[\![D]\!]'$. Instead of $p, p'$ in (1) we have two interpretations $[\![p]\!]$ and $[\![p]\!]'$ of the same text; the interpretations differ only in the semantics of the built-in operations on $D$. Put differently, we consider a semantics $[\![ - ]\!]$ induced from $[\![D]\!]$ and its primitive operations, and a semantics $[\![ - ]\!]'$ induced the same way from $[\![D]\!]'$. Treating $D$ as a built-in data type suffices to capture the relevant aspects of abstract data type modules, without the need for an explicit module construct.

[2] But the restriction to "total simulation" [13] or adjoint simulation [18] does not rule out any simulations in our results, answering a question in [13] about higher order languages. Preservation of simulation for a deterministic first order imperative language without procedures or local variables is shown in [13] as an illustrative application of the general theory.

[3] In its present form the categorical theory does not apply to the rather lax structure of types in the category of transformers, because the theory treats inequational laws but not conditional inequations.

The semantics is based on a typing system that assigns data types to expressions and command types to commands. An expression $e$ is given a data type $T$. Expressions are typed in context with constants $\Gamma$ and (state) variables $\pi$. Such a typing is written $\Gamma;\pi \triangleright e : T$, and its meaning $[\![\Gamma;\pi \triangleright e : T]\!]$ is a function from environments and states to values. Given a semantics $[\![-]\!]$ of types, a *$\Gamma$-environment* is a type-respecting valuation of the constants in $dom.\Gamma$, and a *$\pi$-state* is a type-respecting valuation of variables in $dom.\pi$. Writing $[\![\pi]\!]$ for the set of $\pi$-states and $[\![\Gamma]\!]$ for the set of $\Gamma$-environments, the meaning of an expression is a function $[\![\Gamma;\pi \triangleright e : T]\!] \in [\![\Gamma]\!] \to [\![\pi]\!] \to [\![T]\!]$. Writing $\mathscr{P}[\![\pi]\!]$ for the set of predicates, a command of type $\mathbf{com}(\pi)$ maps environments to transformers over $[\![\pi]\!]$, i.e.

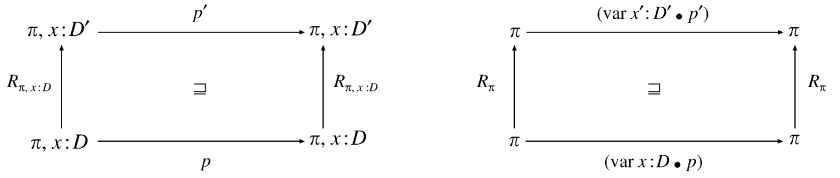
$$[\![\Gamma \triangleright p : \mathbf{com}(\pi)]\!] \in [\![\Gamma]\!] \to \mathscr{P}[\![\pi]\!] \to \mathscr{P}[\![\pi]\!]. \tag{6}$$

For expression $[\![e]\!]'$ to simulate $[\![e]\!]$ by $R$ means (omitting the typing of $e$)

$$\eta \; R_\Gamma \; \eta' \wedge \sigma \; R_\pi \; \sigma' \Rightarrow [\![e]\!]_\eta.\sigma \; R_T \; [\![e]\!]'_{\eta'}.\sigma' \tag{7}$$

for all environments $\eta, \eta'$ and states $\sigma, \sigma'$. Here $R_\Gamma$ is the relation on environments induced from given relations on the built-in types, $R_T$ is the induced relation for type $T$, and $R_\pi$ is the induced relation on $\pi$-states. Section 7 gives these inductive definitions. The preservation theorem (Section 8) says that (7) holds for any expression $e$ provided that it holds for all built-ins; the theorem gives a similar result for commands.

In the methodological literature, soundness is usually proved directly from the definitions. In the semantics literature (e.g. [30]) and in this paper, soundness is a consequence of preservation. Here is a sketch of the argument, dropping the semantic brackets. Simulation for $p$ will amount to some kind of refinement like the left diagram




and preservation yields the right diagram. If $R$ is the identity relation on $\pi$ then the right diagram collapses to (1). For $R$ to be the identity on $\pi$ depends on an *identity extension lemma* (Lemma 3) which says that the coupling on a type is the identity provided that the coupling on its primitive constituents is the identity. If $D$ is to be replaced it should not appear in types in $\pi$, and we can assume the other base types are coupled by identities. It should then follow that constructed types like $\pi$ are also coupled by the identity – but this fails in a language like Io. Even if $D$ does not occur in $T$ it can occur in values of extensible-record type $\mathbf{record}(F : T)$ because they can have additional fields: $[\![\mathbf{record}(F : T)]\!]$ need not be equal to $[\![\mathbf{record}(F : T)]\!]'$, thus the coupling cannot be the identity. Similarly, if $D$ does not

occur in a procedure type **proc**$(x:T\ \ \mathbf{var}\ y:U)$ it can still occur in the type of an external variable of a procedure of this type. Our soundness result requires the global coordinates $\pi$ to have no record or procedure types. Such types are not appropriate observables.

## 2. Relations, ideals, and transformers

In most works on transformer semantics, predicates are either treated syntactically, i.e. as formulas, or they are interpreted as arbitrary sets of states. Indeed, for many purposes it is safe and convenient to blur the distinction. But higher-order features warrant caution; for the model-theoretic foundation of a higher-order refinement calculus, we choose to use sets as predicates. However, arbitrary subsets $\varphi$ of state space $[\![\pi]\!]$ cannot be allowed as predicates. For a higher type, we would take the set $\mathscr{P}[\![\pi]\!]\to\mathscr{P}[\![\pi]\!]$ of transformers to be a data type. But its full powerset $\mathscr{P}(\mathscr{P}[\![\pi]\!]\to\mathscr{P}[\![\pi]\!])$ includes predicates disrespectful of the refinement order – these make undesirable distinctions that invalidate some operationally natural refinement laws (see Section 6). Our solution is to replace sets by posets and to require predicates to be *updeals* (upward closed sets). Updeals make sense for any type where the order relation (generically written $\preccurlyeq$) represents approximation or refinement: If $x$ satisfies predicate $\varphi$ and $x\preccurlyeq x'$ then $x'$ should satisfy $\varphi$ because its approximant $x$ is sufficient. Applications of this model in program construction can be found in [21, 23]. Results mentioned without proof in this section can be shown easily (cf. [8, 22]).[4]

Updeals are familiar in domain theory: Scott-open sets are upward closed and inaccessible by directed join. The latter condition is appropriate for "observable predicates", but not for specifications [29], hence we use the alternative that makes the least commitment, the so-called Alexandrov topology of all updeals. Io's record and procedure type constructors create non-discrete orders, but built-in types can denote arbitrary posets. (For an order to be *discrete* means $x\preccurlyeq y\equiv x=y$.) The order on types like **int** can be discrete, because divergence is modelled in transformer semantics without using $\bot$.

Let $\mathscr{U}A$ denote the set of updeals of poset $A$. Henceforth, a *transformer* is a monotonic function of type $\mathscr{U}A\to\mathscr{U}B$, and in (6) we replace $\mathscr{P}$ by $\mathscr{U}$. For any poset $A$, $\mathscr{U}A$ is a complete lattice, as is each set of transformers $\mathscr{U}A\to\mathscr{U}B$, which provides a

[4] The underlying structure here is robust and elegant: Transformers are lax spans over ideals, which are themselves lax spans over monotonic functions [22]. None of this is explicit in the sequel, but it lurks in the correspondence between relations and transformers that we use to streamline proofs. We also use, without explicit categorical formalization, a lax product of transformers, and the hom-object which is a lax exponent. One of the main motivations for this model was the discovery that the lax exponent of transformers in the powerset model does not preserve data refinement. Even though the exponent of transformers on updeals is still rather lax, its properties suffice for preservation of simulation [19]. But those categorical constructs do not match closely with those found in Oberon-like languages, and it is simpler to give direct proofs of the results in the sequel than to derive them at great length from [19].

straightforward fixpoint semantics for recursive commands. (Io does not include recursively defined function expressions or data types.) States and environments are ordered pointwise and expressions denote monotonic functions. We write $\to$ for monotonic functions throughout the paper.

To connect coupling relations with transformers, we need the direct image function of a coupling to be a transformer, i.e. to map updeals to updeals. For that the coupling needs to respect order in the following sense. For posets $A, A'$ say relation $R \subseteq A \times A'$ is an *ideal* just if

$$\preccurlyeq ; R ; \preccurlyeq \subseteq R. \tag{8}$$

On the left, $\preccurlyeq$ is the relation $(\preccurlyeq_A) \subseteq A \times A$ and on the right it is $\preccurlyeq_{A'}$. The subscripts will often be omitted. In (8) the relations $\preccurlyeq$ are composed with another relation, rather than appearing between arguments like $\subseteq$ in (8). The explicit composition symbol ";" makes such formulas readable. At the level of points, (8) says $b \preccurlyeq a \wedge a\ R\ a' \wedge a' \preccurlyeq b' \Rightarrow b\ R\ b'$. The converse holds for any relation $R$ by reflexivity of $\preccurlyeq$ (i.e. $id \subseteq (\preccurlyeq)$ where $id$ is the identity function). If $\preccurlyeq_A$ and $\preccurlyeq_B$ are discrete then any relation $R$ satisfies (8), so all relations on discretely ordered base types (and hence their function spaces) are ideals.

By contrast with some treatments, where couplings are in different category from program meanings, we embed couplings into the same category. For ideal $R \subseteq A \times A'$, define the *direct image* $\langle R \rangle : \mathcal{U}A \to \mathcal{U}A'$ and *inverse image* transformer $[R] : \mathcal{U}A' \to \mathcal{U}A$ as follows:

$$a' \in \langle R \rangle.\varphi \equiv (\exists a : a\ R\ a' : a \in \varphi) \quad \text{for all } a', \varphi$$
$$a \in [R].\varphi' \equiv (\forall a' : a\ R\ a' : a' \in \varphi') \quad \text{for all } a, \varphi'$$

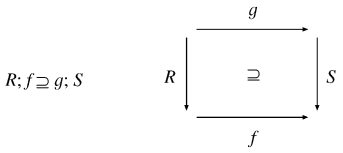
For updeal $\varphi$ and any relation $S$, $\langle S \rangle.\varphi$ is an updeal just if $S ; \preccurlyeq \subseteq S$, and $[S].\varphi$ is an updeal just if $\preccurlyeq ; S \subseteq S$. We require couplings to be ideals, which makes sense if $\preccurlyeq$ means approximation or refinement. Suppose $R \subseteq [\![T]\!] \times [\![T]\!]'$ is a relation from one interpretation $[\![T]\!]$ of type $T$ to another interpretation $[\![T]\!]'$. Suppose $t\,R\,t'$, i.e. $t'$ simulates $t$. If $u \preccurlyeq t$ then $t'$ certainly has enough information to simulate $u$, which is no more refined than $t$. If $t' \preccurlyeq u'$ then $u'$ is certainly adequate to simulate $t$, as it is at least as refined as $t'$. Hence $u\,R\,u'$.

Composition preserves the ideal property, but the converse $R^o$ of an ideal $R$ is not necessarily an ideal, unless the types are discretely ordered. In that case, using the fact that

$$f ; S \supseteq R ; g \equiv S ; g^o \supseteq f^o ; R \tag{9}$$

for all functions $f, g$ and relations $R, S$, one can show that (5) is equivalent to $g$; $S^o \supseteq R^o$; $f$. Thus, if we ignore order, $f$ simulates $g$ by $R$ and $S$ iff $g$ simulates $f$ by $S^o$ and $R^o$. Moreover, (5) is complete for deterministic first-order programs: if (1) holds then there is a coupling that simulates in the sense of (5) [10]. For non-deterministic

programs, a second form of simulation is necessary, viz.,




(10)

Although (10) appears symmetric with (5), (9) does not apply to (10) and at the level of points (10) is more complex than (4). For first-order programs with bounded non-determinacy, *forward simulation* (5) and *backward simulation* (10) are jointly complete [10] (see also [6]).

Although all functions are relations, monotonic functions are only ideals when the orders are discrete. But the first of the following facts implies that $(f;\preccurlyeq)$ is an ideal – as is $\preccurlyeq;f^o$ – for any monotonic $f$:

$$\text{If } R \text{ is an ideal and } f \text{ is monotonic then } (f;R) \text{ is an ideal.} \tag{11}$$

$$\text{If } R \text{ is an ideal and } S \text{ is any relation then } S \subseteq R \equiv S;\preccurlyeq \subseteq R. \tag{12}$$

In particular, the identity function $id \in A \to A$ is not an ideal unless $A$ is ordered discretely. The appropriate "identity coupling" on $A$ is $\preccurlyeq_A$.

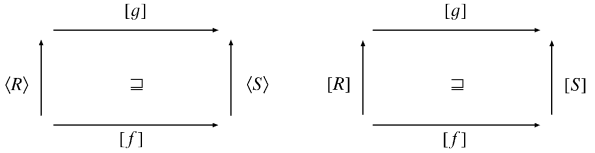
Henceforth we use letters $R, S$ for ideals, and $f, g$ for monotonic functions of various kinds. For any $R$, $\langle R\rangle$ is universally disjunctive and $[R]$ is universally conjunctive, whence the first of the following facts:

$$f;\langle R\rangle \sqsubseteq \langle S\rangle;g \equiv [S];f \sqsubseteq g;[R] \tag{13}$$

$$R \subseteq S \equiv \langle R\rangle \sqsubseteq \langle S\rangle, \quad R \subseteq S \equiv [R] \sqsupseteq [S] \tag{14}$$

$$\langle R;S\rangle = \langle R\rangle;\langle S\rangle, \quad [R;S] = [S];[R], \quad \langle\preccurlyeq\rangle = id, \quad [\preccurlyeq] = id \tag{15}$$

By facts (13)–(15) we have $f;S \sqsupseteq R;g \equiv [f];\langle R\rangle \sqsubseteq \langle S\rangle;[g]$ and by (14) and (15) we have $R;f \sqsupseteq g;S \equiv [f];[R] \sqsubseteq [S];[g]$. Thus, both (5) and (10) form configurations oriented like (5):




(16)

This is used in [9] for a single complete rule, but completeness is beyond the scope of this paper. We only treat the more commonly used forward simulations, i.e. (5) and the left-hand side of (16).

Couplings as transformers can streamline the derivation of commands ([17] versus [10]), but their use for data refinement of expressions is limited to those of type

Boolean. (Much of the literature deals only with data refinement of commands, not expressions). Here we work directly with relations in order to deal with expressions in assignments and as procedure arguments. For expressions, as functions of state, our simulations take the form of (5). For commands we use the left-hand side of (16): in place of the horizontal arrows $[f],[g]$ will be command meanings $[\![p]\!]$, $[\![p]\!]'$.

To use relation calculus in proofs, we need a few facts about monotonic functions and ideals. To get a feel for these facts, you may enjoy using (9) to show

$$(f \text{ is monotonic}) \equiv (\preccurlyeq; f) \subseteq (f; \preccurlyeq) \tag{17}$$

(Hint: monotonicity of $f$ is equivalent to $(\preccurlyeq) \subseteq (f; \preccurlyeq; f^o)$.) For reasoning about the pointwise ordering monotonic functions, we use the equivalence

$$f \preccurlyeq g \equiv (g; \preccurlyeq) \subseteq (f; \preccurlyeq). \tag{18}$$

The following lemma shows how we can combine simulations of functions with algorithmic refinements of them:

$$g \preccurlyeq f \wedge f; S \supseteq R; f' \wedge f' \preccurlyeq g' \Rightarrow g; S \supseteq R; g' \tag{19}$$

We need the extension $\otimes$ of Cartesian products to transformers. For posets $A, B$, define $A \otimes B = A \times B$ (the order-theoretic product), and for transformers $f \in \mathscr{U}A \to \mathscr{U}B$ and $g \in \mathscr{U}C \to \mathscr{U}D$ define the "lax product" transformer $(f \otimes g) \in \mathscr{U}(A \otimes C) \to \mathscr{U}(B \otimes D)$ by

$$(b,d) \in (f \otimes g).\delta \equiv (\exists \alpha, \beta : \alpha \times \beta \subseteq \delta : b \in f.\alpha \wedge d \in g.\beta) \tag{20}$$

This extends the Cartesian product in the sense that $\langle R \rangle \otimes \langle S \rangle = \langle R \times S \rangle$ where we use the ideal $R \times S$ defined by $(x,y)\ (R \times S)\ (z,w) \equiv x\,R\,z \wedge y\,S\,w$. The "laxity" of $\otimes$ is evident in its weak distribution over composition: In general, we have only $f; f' \otimes g; g' \sqsubseteq (f \otimes g); (f' \otimes g')$. For Io semantics, it is enough that $f; f' \otimes id = (f \otimes id); (f' \otimes id)$, and this does hold for all $f$, $f'$. In Section 4 the definition of $f \otimes g$ is specialized for the case where $g$ is an identity and $A, B$ are sets of states; $\otimes$ is then associative up to equality.

## 3. Syntax of Io except records

Procedures are important and non-trivial, so we defer the added complication of records and subtyping until Section 9. The main syntactic classes are data types, expressions, and commands. We assume given disjoint sets of

- variables (or lists thereof), with typical elements $x, y, z, w, pv$,
- constants (or lists thereof), with typical elements $P, Q$,
- built-in constants, with typical element $c$.

Data types $T$ and type assignments are generated from built-in base types $B$ by the following grammar:

$$T ::= B \mid \mathbf{bool} \mid T \times T \mid T \to T \mid \mathbf{proc}(x : T \ \mathbf{var}\, x : T)$$

$$\pi ::= \emptyset \mid \pi, x : T \qquad \text{data type assignment to variables}$$

$$\Gamma ::= \emptyset \mid \Gamma, Q : T \mid \Gamma, Q : \mathbf{com}(\pi) \quad \text{type assignment to constants}$$

The typing rules are simplified by allowing an identifier "$x$" to stand for a list of variables (e.g. a parameter list can be written as just $x$). We also adopt the list convention for data types $T$, constants $Q$, type assignments $x : T$, and expressions $e$ – but not for commands. Type assignments are included in the grammar for reference, but we abuse notation and treat them as finite functions; e.g. $\pi.x$ is the type of $x$, for $x \in dom.\pi$. The type $\mathbf{proc}(x : T \ \mathbf{var}\, y : U)$ is for procedures with value parameter $x$ and value-result parameter $y$. The keyword **var** is used to separate $x$ from $y$ since both may be lists.

To enforce the restriction that externals of procedures are in outermost scope, we assume variables are partitioned into *Globals* and *Locals*; the typing rules ensure that only Globals may be externals of procedures and only Locals may be bound as parameters and as local variables. Oberon-like languages enforce the restriction by requiring the relevant procedure declarations to be in outermost scope.

Expressions $e$ and commands $p$ are generated from built-in constants $c$ by

$$e ::= c \mid x \mid Q \mid e(e) \qquad \text{built-in, var., const., applic.}$$

$$\mid (\mathbf{pro}\, x : T \ \mathbf{var}\, x : T \bullet p) \qquad \text{procedure abstraction}$$

$$p ::= Q \mid x := e \mid \mathbf{call}\ e(e,x) \mid p; p \mid (\mathbf{if}\ e\ \mathbf{then}\ p\ \mathbf{else}\ p)$$

$$\mid p \sqcap p \mid (\mathbf{var}\, x : T \bullet p) \qquad \text{demonic choice, local var.}$$

$$\mid (\mathbf{let}\, Q : T = e \bullet p) \mid (\mathbf{rec}\, Q : \mathbf{com}(\pi) \bullet p) \quad \text{local constant, recursion}$$

An *expression typing* has the form $\Gamma; \pi \triangleright e : T$ where neither $e$ nor $T$ are lists. A *command typing* has the form $\Gamma \triangleright p : \mathbf{com}(\pi)$. The typing rules are in Tables 2 and 3. Section 9 extends the language with records, for which there is a non-trivial relation $\Subset$ of structural subtyping. For now, the rules can be read with $\Subset$ as equality of types. (We consider parameter names in proc types to be significant, to avoid un-illuminating renamings in semantics [25].)

Procedure expressions (**pro**) are distinct from declarations (**let**). The rules ensure that **let** binds constants to state-independent expressions only, so it is referentially transparent. This and the other claims about typings and semantics are proved in [25].

Table 2
Rules for expressions

$$\frac{}{\Gamma;Q:T;\ \pi \triangleright Q:T}\ (use\ const) \qquad \frac{}{\Gamma;\pi,x:T \triangleright x:T}(use\ var)$$

$$\frac{\Gamma;\pi \triangleright f:T \to U \quad \Gamma;\pi \triangleright e:T' \quad T' \Subset T}{\Gamma;\pi \triangleright f(e):U}(apply) \qquad \frac{c:T \text{ is given}}{\Gamma;\pi \triangleright c:T}(built\ in)$$

$$\frac{\Gamma \triangleright p:\mathbf{com}(x:T,y:U,z:V) \quad x,y \text{ Local} \quad z \text{ Global} \quad \text{rank}.V \leqslant \text{rank}.(T,U)}{\Gamma;\pi \triangleright (\mathbf{pro}\,x:T\,\mathbf{var}\,y:U \bullet p):\mathbf{proc}(x:T\,\mathbf{var}\,y:U)}(proc)$$

Table 3
Rules for commands

$$\frac{}{\Gamma,Q:\mathbf{com}(\pi) \triangleright Q:\mathbf{com}(\pi,x:T)}(com\ const)$$

$$\frac{\Gamma \triangleright p:\mathbf{com}(\pi) \quad \Gamma \triangleright p:\mathbf{com}(\pi)}{\Gamma \triangleright (p \sqcap p'):\mathbf{com}(\pi)}(choice)$$

$$\frac{\Gamma;\pi \triangleright e:\mathbf{bool} \quad \Gamma \triangleright p:\mathbf{com}(\pi) \quad \Gamma \triangleright p':\mathbf{com}(\pi)}{\Gamma \triangleright (\mathbf{if}\ e\ \mathbf{then}\ p\ \mathbf{else}\ p'):\mathbf{com}(\pi)}(cond)$$

$$\frac{\Gamma;\pi,x:T \triangleright e:T' \quad T' \Subset T}{\Gamma \triangleright (x:=e):\mathbf{com}(\pi,x:T)}(assign) \qquad \frac{\Gamma \triangleright p:\mathbf{com}(\pi) \quad \Gamma \triangleright p':\mathbf{com}(\pi)}{\Gamma \triangleright (p;p'):\mathbf{com}(\pi)}(seq)$$

$$\frac{\Gamma;\pi \triangleright e:\mathbf{proc}(x:T\ \mathbf{var}\,y:U) \quad \Gamma;\pi \triangleright e':T' \quad T' \Subset T \quad \pi.w=U \quad w \text{ Local}}{\Gamma \triangleright (\mathbf{call}\,e(e',w)):\mathbf{com}(\pi)}(call)$$

$$\frac{\Gamma,Q:T \triangleright p:\mathbf{com}(\pi) \quad \Gamma;\emptyset \triangleright e:T}{\Gamma \triangleright (\mathbf{let}\,Q:T=e \bullet p):\mathbf{com}(\pi)}(let)$$

$$\frac{\Gamma,P:\mathbf{com}(\pi) \triangleright q:\mathbf{com}(\pi)}{\Gamma \triangleright (\mathbf{rec}\,P:\mathbf{com}(\pi) \bullet q):\mathbf{com}(\pi,x:T)}(rec)$$

$$\frac{\Gamma \triangleright p:\mathbf{com}(\pi,x:T) \quad \Gamma;\pi \triangleright e:T' \quad T' \Subset T \quad x \text{ Local}}{\Gamma \triangleright (\mathbf{var}\,x:T:=e \bullet p):\mathbf{com}(\pi)}(var)$$

The differences between this language and that in [25] are minimal.[5] For technical convenience, we assume each variable or constant identifier is implicitly associated with a fixed data type, written $typ.x$, and we require that the type of each identifier in a type assignment is its fixed type.

[5] We combine local variable declarations with initializing assignments (alternatively, they can be kept separate, but initialization is then needed in the soundness theorem). The remaining changes are also to simplify the exposition. We omit lambda abstraction, for which soundness of data refinement is straightforward (see [24]), but we keep function types and functional constants. Specification constructs (see Section 6) and structural rules for state space extension are omitted. The structural rules are used in [25] to prove results about non-interference, but every derivable typing has a canonical derivation without structural rules, so here we confine attention to canonial derivations.

The hypothesis in rule (*assign*) is written for a single expression $e$. But $x$ and $e$ can be lists so, strictly speaking, the hypothesis should be a list of typings, one for each expression. We refrain from cluttering the notation here and for the analogous situation with $e'$ in rule (*call*) (and later in rule (*record*) of Table 7). Rules (*com const*) and (*rec*) allow recursive commands to occur in contexts with arbitrary extra variables $x$. This obviates the need for a separate coercion rule. Command types do not appear as the types of variables, but for **rec** and **let** we have command-type constants.

To extend couplings from base types to all kinds of phrases, we will define couplings that correspond to semantic categories not directly denotable by phrases in the language. For that purpose, *coupling types* $\theta$ are defined by

$$\theta ::= T \mid \Gamma \mid \pi \mid \mathbf{com}(\pi).$$

The notion of *bound constant* is standard: $Q$ is bound in $(\mathbf{let}\, Q : T = e \bullet p)$ and in $(\mathbf{rec}\, Q : \mathbf{com}(\pi) \bullet p)$. Fat dot begins the scope of a binding, so the scope of $Q$ is $p$ in both cases. For variables of **pro** expressions we need a slightly non-standard definition; as a reminder, the terms Free and Bound (i.e. not Free) are capitalized in reference to variables.

**Definition 1.** As usual, $x$ is not *Free* in $(\mathbf{var}\, x : T \bullet p)$. Built-in constants do not have *Free* variables. There are no *Free* variables in $(\mathbf{pro}\, x : T\, \mathbf{var}\, y : U \bullet p)$: not only $x$ and $y$ but all the *Free* variables of $p$ are Bound by **pro**. For all other expressions and commands the definition is as usual.

The notion of Free variable for **pro** expressions may seem strange, but in procedure variables we confront a similar situation. In the command $\mathbf{call}\, pv$ where $pv$ is a procedure type variable, the Free variable is $pv$, even though in a given state the value of $pv$ will be a procedure which can have external variables. The externals only come into play in the call of the procedure; the value of $pv$ in a state does not depend on the values of its externals in that state. Note that command types like $\mathbf{com}(x : T)$ involve what is in some sense a "binding" of $T$ to $x$. But these are not bindings; e.g. $x$ can be Free in $(\mathbf{rec}\, Q : \mathbf{com}(x : \mathbf{int}) \bullet \ldots)$. Further explanation, and results showing that the rules prevent problems with aliasing, can be found in [25]. [6]

Typings are *derivable* iff they have derivations using allowed instances of the rules. An *allowed instance* of a rule is one such that: (i) in typing $\Gamma; \pi \triangleright e : T$, the bound variables of $e$ are disjoint from $dom.\pi$, and (ii) in $\Gamma \triangleright p : \mathbf{com}(\pi)$ the bound variables in $p$ are disjoint from $dom.\pi$ – except that $\pi$ may include externals of **pro**-expressions in both (i) and (ii). It can be shown that in a derivable typing $\Gamma \triangleright p : \mathbf{com}(\pi)$, no variable in $dom.\pi$ has a binding occurrence in $p$ and no variable has more than one binding occurrence in $p$ – except for externals of **pro**-expressions (several procedures can bind the same global variable, but semantically it is shared as if it were free). Absence

[6] In [25], constants $Q : \mathbf{com}(\pi)$ of command type are considered to have $dom.\pi$ as Free variables but this is for minor technical reasons not relevant here.

of variable re-declarations is not essential, but it spares us tiresome renamings in the semantics of rules (*var*) and (*call*), while retaining lexical scope. Typing derivations are unique, so semantics of a typing can be defined by induction on its derivation.

The typing rule for procedures imposes a stratification condition which makes it possible to define semantics of types by an ordinary induction rather than as the solution of a domain equation. That in turn means that we need not require posets to be complete or predicates to be Scott-open, so we can retain the expressive power of refinement calculus in combination with procedure variables. The *rank* of a data type $T$ (or list thereof) is the depth of nesting of **proc** (but not $\to$): $rank.B = 0$ for built-in $B$, for lists, etc., we define $rank.(T,U) = rank.(T \times U) = rank.(T \to U) = max.(rank.T, rank.U)$. Finally, $rank.(\mathbf{proc}(x:T\,\mathbf{var}\ y:U)) = 1 + max.(rank.T, rank.U)$. As an example, a procedure of type **proc**() can have an external variable $z$ that is an array $J \to \mathbf{int}$ of integers, taking **int** and the index type $J$ as built-ins; but $z$ cannot itself be of type **proc**(). Because there are closed terms at every rank, which can be passed as dummy arguments, the language remains very expressive despite disallowing, e.g. self-application.

## 4. Types, predicates and transformers

This section defines the semantics of types, as well as various operations on predicates and transformers. We assume given for each built-in data type $B$ a non-empty poset $[\![B]\!]$. The semantics for other types is given in Table 4. For each coupling type $\theta$ we define a set $[\![\theta]\!]$ and an order $\preccurlyeq_\theta$. Often the subscript on $\preccurlyeq$ is omitted; note also that $\preccurlyeq_{\mathbf{com}(\pi)}$ is $\sqsubseteq$.

Table 4
Semantics of types

$$
\begin{array}{rcl}
[\![\mathbf{bool}]\!] & = & \{true, false\} \\
j \preccurlyeq k & \equiv & j = k \\
[\![T \times U]\!] & = & [\![T]\!] \times [\![U]\!] \\
\preccurlyeq & = & \preccurlyeq_T \times \preccurlyeq_U \\
[\![T \to U]\!] & = & [\![T]\!] \to [\![U]\!] \\
f \preccurlyeq g & \equiv & g;\preccurlyeq \subseteq f;\preccurlyeq \ \ i.e.,\ (\forall x :: f.x \preccurlyeq g.x) \\
[\![\mathbf{proc}(x:T\,\mathbf{var}\ y:U)]\!] & = & \{(f,z) \mid z \text{ is Global} \wedge rank.V \leqslant rank.(T,U) \\
 & & \quad \wedge f \in [\![\mathbf{com}(x:T, y:U, z:V)]\!]\} \text{ where } V = typ.z \\
(f,z) \preccurlyeq (g,u) & \equiv & (\exists z_0 : z = u, z_0 : f \preccurlyeq_{\mathbf{com}(x,y,z)} (g \otimes id_{z_0}))
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathbf{com}(\pi)]\!] & = & \mathscr{U}[\![\pi]\!] \to \mathscr{U}[\![\pi]\!] \\
f \preccurlyeq g & \equiv & (\forall \varphi :: f.\varphi \subseteq g.\varphi) \\
[\![\pi]\!] & = & \{\sigma \mid dom.\sigma = dom.\pi \wedge (\forall x : x \in dom.\pi : \sigma.x \in [\![\pi.x]\!])\} \\
\sigma \preccurlyeq \tau & \equiv & (\forall x : x \in dom.\pi : \sigma.x \preccurlyeq \tau.x) \\
[\![\Gamma]\!] & = & \{\eta \mid dom.\eta = dom.\Gamma \wedge (\forall Q : Q \in dom.\eta : \eta.Q \in [\![\Gamma.Q]\!])\} \\
\eta \preccurlyeq \zeta & \equiv & (\forall Q : Q \in dom.\Gamma : \eta.Q \preccurlyeq \zeta.Q)
\end{array}
$$

The data type constructors $\times$ and $\to$ are interpreted by the usual order-theoretic definitions. The discrete order on **bool** is needed so that conditionals have the standard meaning and are monotonic in their guards; moreover, negation is monotonic and hence admissible as a built-in constant.

For procedure types, note that without restriction on the rank of the external variables $z$ the definition would be circular, i.e. we would have to solve a recursive equation rather than it being a simple inductive definition. We include all monotonic transformers as procedure values, as is needed for the full refinement calculus of [25] with specification constructs, but see Section 6.

The rest of this section uses semantics of types to define various notations about transformers. Conjunction and disjunction of $\pi$-predicates are given set-theoretically, but the set-theoretic complement of an updeal need not be an updeal. The negation $\sim\varphi$ of $\varphi$ is defined to be $\varphi \approx\!> \emptyset$. The implication $\pi$-predicate is defined by $\varphi \approx\!> \psi = (\cup\delta : \delta \cap \varphi \subseteq \psi : \delta)$ where $\delta$ ranges over $\pi$-predicates. For $\varphi \in \mathscr{U}A$ with $A$ discretely ordered, $\sim\varphi$ is just the complement $A - \varphi$. We write $\downharpoonleft x$ for the projection of states onto variables other than $x$, i.e. $\sigma \in [\![\pi, x : T]\!]$ and $y \in dom.\pi$ imply $\sigma \downharpoonleft x \in [\![\pi]\!]$ and $(\sigma \downharpoonleft x).y = \sigma.y$. The notation $\downharpoonleft$ is also used for restricting the domains of other kinds of functions. We also use an alternative postfix notation $(\dagger x)$ for the inverse image function $[\downharpoonleft x]$; thus for any $x \notin dom.\pi$ and $\varphi \in \mathscr{U}[\![\pi]\!]$, the subset $\varphi \dagger x$ of $[\![\pi, x : T]\!]$ is defined by $\sigma \in \varphi \dagger x \equiv \sigma \downharpoonleft x \in \varphi$. Both $\downharpoonleft x$ and $\dagger x$ are identity functions if $x$ is the empty list. For $(\pi, x : T)$-predicate $\varphi$, define the $\pi$-predicates $(\forall x \bullet \varphi)$ and $(\exists x \bullet \varphi)$ by $\sigma \in (\exists x \bullet \varphi) \equiv (\exists \tau : \sigma = \tau \downharpoonleft x : \tau \in \varphi)$ and $\sigma \in (\forall x \bullet \varphi) \equiv (\forall \tau : \sigma = \tau \downharpoonleft x : \tau \in \varphi)$. These have the usual logical properties, e.g. $(\exists x \bullet \varphi \dagger x) = \varphi$.

The state space with no coordinates is used for semantics of **let**. We write $*$ for the empty function as a state, as opposed to $\emptyset$ for the empty function as a type assignment, so $*$ is the only $\emptyset$-state. For any function $f$, the notation $f(\alpha \mapsto \beta)$ is used for the function sending $\alpha$ to $\beta$ and otherwise acting as $f$; the domain of $f$ may or may not contain $\alpha$. Syntactic substitution is not used to interpret assignment commands because predicates are not formulas. If $\varphi$ is a $\pi$-predicate and $f$ is a function from $\pi$-states to values of type $\pi.x$, then define the *semantic substitution* so that $\varphi(x :\approx f)$ is the $\pi$-predicate defined by $\sigma \in \varphi(x :\approx f) \equiv \sigma(x \mapsto f.\sigma) \in \varphi$ ($x$ may be a disjoint list of variables, $f$ corresponding thereto). Thus $(x :\approx f)$ is the inverse image $[(\lambda\sigma :: \sigma(x \mapsto f.\sigma))]$ of the function that updates coordinate $x$. For projection of states onto particular coordinates, define the function $\hat{x}$ in $[\![x : T, y : U]\!] \to [\![T]\!]$ by $\hat{x}.\sigma = \sigma.x$. Dependence on $y, T, U$ is suppressed in the notation $\hat{x}$.

For any $\pi$ and transformer $g$ over $[\![\pi]\!]$ we need the transformer $g \otimes id_x$ over $[\![\pi, x : T]\!]$. For brevity we write $id_x$ for the identity function on $\mathscr{U}[\![x : T]\!]$. Context distinguishes between $id_x$ for $\mathscr{U}[\![x : T]\!]$ and for $\mathscr{U}[\![x : T]\!]'$. To simplify and specialize definition (20) of $\otimes$ for this situation, we first define an abbreviation which is the semantic analog of substitution of a literal for a variable in a formula. For $(\pi, x : T)$-predicate $\varphi$ and element $k$ of $[\![T]\!]$, define the $\pi$-predicate $\varphi(x \leadsto k)$ by $\varphi(x \leadsto k) = (\exists x \bullet \varphi(x :\approx (\lambda\sigma :: k)))$. Expanding the definitions gives $\sigma \in \varphi(x \leadsto k) \equiv \sigma(x \mapsto k) \in \varphi$, for all $\sigma \in [\![\pi]\!]$. Now define $(\otimes id_x)$ by $\sigma \in (g \otimes id_x).\varphi \equiv \sigma \downharpoonleft x \in g.\varphi(x \leadsto \sigma.x)$.

Table 5
Semantics of expression rules

| | |
|---|---|
| $[\![\Gamma, Q:T;\pi \rhd Q:T]\!]_\eta\cdot\sigma = \eta.Q$ | (*use const*) |
| $[\![\Gamma;\pi, x:T \rhd x:T]\!]_\eta.\sigma = \sigma.x \quad (i.e.,\ [\![x]\!] = \hat{x})$ | (*use var*) |
| $[\![\Gamma;\pi \rhd c:T]\!]_\eta\cdot\sigma = [\![c]\!]$ | (*built in*) |
| $[\![\Gamma;\pi \rhd e(e'):T]\!]_\eta\cdot\sigma$ | (*apply*) |
| $= [\![\Gamma;\pi \rhd e: U \to T]\!]_\eta\cdot\sigma.([\![\Gamma;\pi \rhd e':U]\!]_\eta.\sigma)$ | |
| $[\![\Gamma;\pi \rhd (\mathbf{pro}\, x:T\ \mathbf{var}\ y:U \bullet p):\mathbf{proc}(x:T\ \mathbf{var}\ y:U)]\!]_\eta\cdot\sigma$ | (*proc*) |
| $= ([\![\Gamma \rhd p:\mathbf{com}(x:T, y:U, z:V)]\!]_\eta, z)$ | |

Table 6
Semantics of command rules except (*call*)

| | |
|---|---|
| $[\![\Gamma, Q:\mathbf{com}(\pi) \rhd Q:\mathbf{com}(\pi, x:T)]\!]_\eta.\varphi = (\eta.Q \otimes id_x).\varphi$ | (*com const*) |
| $[\![\Gamma \rhd (p \sqcap q):\mathbf{com}(\pi)]\!]_\eta.\varphi$ | (*choice*) |
| $= [\![\Gamma \rhd p:\mathbf{com}(\pi)]\!]_\eta.\varphi \cap [\![\Gamma \rhd q:\mathbf{com}(\pi)]\!]_\eta.\varphi$ | |
| $[\![\Gamma \rhd (\mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ q):\mathbf{com}(\pi)]\!]_\eta.\varphi$ | (*cond*) |
| $= ([\![\Gamma;\pi \rhd b:\mathbf{bool}]\!]_\eta.\approx\!\!> [\![\Gamma \rhd p:\mathbf{com}(\pi)]\!]_\eta.\varphi) \cap (\sim [\![b]\!]_\eta \approx\!\!> [\![q]\!]_\eta.\varphi)$ | |
| $[\![\Gamma \rhd x := e:\mathbf{com}(\pi, x:T)]\!]_\eta.\varphi = \varphi(x :\approx [\![\Gamma;\pi, x:T \rhd e:T']\!]_\eta)$ | (*assign*) |
| $[\![\Gamma \rhd (p;q):\mathbf{com}(\pi)]\!]_\eta.\varphi = [\![\Gamma \rhd p:\mathbf{com}(\pi)]\!]_\eta.([\![\Gamma \rhd q:\mathbf{com}(\pi)]\!]_\eta.\varphi)$ | (*seq*) |
| $[\![\Gamma \rhd (\mathbf{let}\ Q:T = e \bullet p):\mathbf{com}(\pi)]\!]_\eta.\varphi$ | (*let*) |
| $= [\![\Gamma, Q:T \rhd p:\mathbf{com}(\pi)]\!]_{\eta(Q \mapsto [\![\Gamma;\emptyset \rhd e:T]\!]_\eta.*)}.\varphi$ | |
| $[\![\Gamma \rhd (\mathbf{rec}\, P:\mathbf{com}(\pi) \bullet q):\mathbf{com}(\pi, x:T)]\!]_\eta.\varphi$ | (*rec*) |
| $= ((\mu f :: [\![\Gamma, P:\mathbf{com}(\pi) \rhd q:\mathbf{com}(\pi)]\!]_{\eta(P \mapsto f)}) \otimes id_x).\varphi$ | |
| $[\![\Gamma \rhd (\mathbf{var}\, x:T := e \bullet p):\mathbf{com}(\pi)]\!]_\eta.\varphi$ | (*var*) |
| $= (\exists x \bullet ([\![\Gamma \rhd p:\mathbf{com}(\pi, x:T)]\!]_\eta.(\varphi \dagger x))(x :\approx [\![\Gamma;\pi \rhd e:T']\!]_\eta \circ (\!|x|\!)))$ | |

## 5. Semantics of expressions and commands

Recall from Section 1 that for expression typing $\Gamma;\pi \rhd e:T$, environment $\eta$, and state $\sigma$, the meaning $[\![\Gamma;\pi \rhd e:T]\!]_\eta.\sigma$ is an element of poset $[\![T]\!]$. For command $\Gamma \rhd p:\mathbf{com}(\pi)$, the meaning $[\![\Gamma \rhd p:\mathbf{com}(\pi)]\!]$ is a monotonic function $\mathscr{U}[\![\pi]\!] \to \mathscr{U}[\![\pi]\!]$. We assume that for each built-in constant $c$ of type $T$ an element of $[\![T]\!]$, denoted by $[\![c]\!]$, is given. The value of a built-in constant does not depend on state or environment, so $[\![c]\!]_\eta.\sigma$ is $[\![c]\!]$.

The semantics is defined by induction on typing derivations. In other words, there is an interpretation for each rule, as a function of the interpretations of its hypotheses. Identifiers in the semantic definitions are as in the corresponding typing rules. Environments and types are sometimes omitted when they are obvious or irrelevant. Tables 5 and 6 give the semantics of rules except (*call*). The semantics of rule (*var*) can also be written as follows, for all $\sigma, \varphi$:

$$\equiv \begin{array}{l} \sigma \in [\![\Gamma \rhd (\mathbf{var}\, x:T := e \bullet p):\mathbf{com}(\pi)]\!]_\eta.\varphi \\ \sigma(x \mapsto [\![\Gamma;\pi \rhd e:T']\!]_\eta.\sigma) \in [\![\Gamma \rhd p:\mathbf{com}(\pi, x:T)]\!]_\eta.(\varphi \dagger x) \end{array} \tag{21}$$

For semantics of (*call*) we use a Boolean-valued function **wp** to define

$$\begin{array}{l} \sigma \in [\![\Gamma \triangleright \textbf{call } e(e', w) : \textbf{com}(\pi)]\!]_\eta.\varphi \\ \equiv \\ \textbf{wp}.\sigma.[\![\Gamma; \pi \triangleright e]\!]_\eta.[\![\Gamma; \pi \triangleright e']\!]_\eta.w.\varphi \end{array} \tag{22}$$

for all $\sigma \in [\![\pi]\!]$ and $\varphi \in \mathscr{U}[\![\pi]\!]$. Note that **wp** depends on the name $w$ of the **var** argument, not on its value. We take **wp** to be a total function of type

$$[\![\pi]\!] \to ([\![\pi]\!] \to [\![\textbf{proc}(x \textbf{ var } y)]\!]) \to ([\![\pi]\!] \to [\![T]\!]) \to dom.\pi \to \mathscr{U}[\![\pi]\!] \to \{tt, ff\}$$

Note that **wp** also depends on $\pi$, but we suppress that as we are defining the semantics of a generic instance of rule (*call*); $\pi$ is fixed throughout this discussion. For all $\sigma, h, g, w, \varphi$, define **wp** by

$$\begin{array}{l} \textbf{wp}.\sigma.h.g.w.\varphi \\ \equiv \\ (\exists f, z, t : h.\sigma = (f, z) \wedge dom.\pi = w, z, t : \sigma \in \textbf{wpc}.f.z.g.w.\varphi) \end{array} \tag{23}$$

The first conjuncts serve two purposes. They ensure that externals $z$ are in scope (the call is divergent otherwise) and force $t$ to name the remaining coordinates; $f, z, t$ are uniquely determined by $\pi$ and $h.\sigma$. The subsidiary notation **wpc** is defined for all $\sigma, f, z, g, w, \varphi$ by

$$\begin{array}{l} \sigma \in \textbf{wpc}.f.z.g.w.\varphi \\ \equiv \\ \sigma(x, y \mapsto g.\sigma, \sigma.w) \in (f \otimes id_{w,t}).(\varphi \dagger x, y)(w :\approx \hat{y}) \end{array} \tag{24}$$

This is the standard semantics for value and result parameters, and there is a copy rule replacing a call by local variables and assignments for parameter passing (in cases where the called procedure is not a variable) [25].

## 6. Intermezzo on specification constructs

For the language without specification constructs, a straightforward Scott–Strachey semantics can be given based on domains and state transformers, and results on data refinement should be provable in the standard way using logical relations. Perhaps 'standard' is a bit strong, as most work using logical relations is concerned with program equivalence rather than refinement. Moreover, standard logical relations do not compose, and thus are not adequate for data refinement; promising generalizations have recently appeared [12] for functional languages. In any case, the primary reason for our use of predicate transformer semantics is that it models specification constructs for refinement calculi. This short section discusses these constructs, and in particular the issues that arise in a calculus of higher-order programs, to motivate our model. But the constructs themselves pose no problems for the results in the sequel, so we omit them in the rest of the paper.

To give a proper semantics for specification constructs, the companion paper [25] includes syntax and semantics for state predicates. Here, for expository purposes, we

confuse formulas with semantic predicates, i.e. state sets, and we ignore typing. The first construct, *prescription*, takes the form $\mathbf{frame}\, x : T\, \mathbf{pre}\, \varphi\, \mathbf{post}\, \psi$ and is used to specify a program with precondition $\varphi$ and postcondition $\psi$, which modifies at most the variable(s) $x$. This construct is taken to be a command, albeit one which is not feasible. The semantics is remarkably simple: $[\![\mathbf{frame}\, x\, \mathbf{pre}\, \varphi\, \mathbf{post}\, \psi]\!].\psi' = \varphi \cap (\forall x \bullet \psi \approx\!> \psi')$. This yields the fundamental theorem of refinement calculus, which says that $\mathbf{frame}\, x : T\ \mathbf{pre}\, \varphi\, \mathbf{post}\, \psi \sqsubseteq p$ holds just if $p$ satisfies the specification. The point of the theorem is that satisfaction can be abandoned in favor of refinement. (To formalize such a theorem, one needs an independent semantics for specifications and for programs, as well as a notion of satisfaction, which is beyond our scope.)

Consider, as an example, the specification

$$\mathbf{frame}\, x : \mathbf{int}\, \mathbf{pre}\, \mathit{true}\, \mathbf{post}\, x > 0 \tag{25}$$

which requires $x$ to be set to a positive value and all other variables to be left unchanged. It is refined by the specification $\mathbf{frame}\, x : \mathbf{int}\, \mathbf{pre}\, \mathit{true}\, \mathbf{post}\, x = 1$ as well as by the equivalent assignment $x := 1$. The first of these refinements is an instance of a general refinement law which corresponds to the rule of consequence in Hoare logic: $\mathbf{frame}\, x : T\ \mathbf{pre}\, \varphi\, \mathbf{post}\, \psi \sqsubseteq \mathbf{frame}\, x : T\ \mathbf{pre}\, \varphi'\ \mathbf{post}\, \psi'$ provided $\varphi \Rightarrow \varphi'$ and $\psi' \Rightarrow \psi$. So far, it might appear that we have achieved nothing more than an alternative notation. But because specifications are considered to be commands, and are given semantics in the same category as feasible commands, they may be freely intermixed in a calculus formalizing stepwise and piecewise refinement of programs [11, 16]. The goal is for development to proceed by a series of refinements $p \sqsubseteq \cdots \sqsubseteq p'$ from a specification $p$ to a feasible program $p'$, i.e. one that includes no specification constructs.

The fundamental theorem can be viewed as saying that a prescription is the greatest lower bound among programs satisfying the specification. Such a thing need not exist as a state-transformer. For example, (25) must be unboundedly non-deterministic; but even countable non-determinacy is incompatible with ordinary domain theoretic semantics [1].[7] In fact, transformer semantics also models demonic and angelic choice operators with arbitrary index sets, but we omit those constructs.

An adequate form of specification must allow postconditions to refer to the initial state. This is achieved by the *auxiliary variable* construct $(\mathbf{aux}\, x : T \bullet p)$ as in the example $(\mathbf{aux}\, y : \mathbf{int} \bullet \mathbf{frame}\, x : \mathbf{int}\, \mathbf{pre}\, x = y\, \mathbf{post}\, x > y)$ which specifies a program to increase $x$. The program is required to meet the specification $\mathbf{pre}\, x = y\, \mathbf{post}\, x > y$ for all values of $y$, so this construct can be seen as a least upper bound operator on specifications indexed by states of their auxiliary variables. Thus, it is not surprising that the predicate transformer semantics is given by existential quantification:

[7] An obvious alternative is for specifications to denote the set of satisfying state transformers; for semantics of a full calculus, this means lifting the usual semantic operators to sets of state transformers. To date, this idea has been explored only for simple functional [5] and first-order imperative languages. For the latter, [32] exposes but does not resolve difficulties in accurately matching the standard semantics of refinement calculus.

$[\![\mathbf{aux}\, x : T \bullet p]\!].\varphi = (\exists x \bullet [\![p]\!].\varphi)$. Note that $p$ need not be a prescription; if $p$ begins with an initializing assignment to $x$, the semantics is the same as for our (initialized) **var** block. But auxiliaries are not normally used as targets of assignment. Uninitialized **var** blocks can be given the same semantics as for auxiliaries except for universal rather than existential quantification. An operational interpretation can be given in terms of a game between an angel and a demon; then auxiliaries are initialized angelically whereas **var** is demonic [2].

Transformers for feasible programs satisfy Dijkstra's *healthiness conditions*: strictness, conjunctivity, and continuity. There is an isomorphism between state transformers and healthy predicate transformers [28]. Prescriptions denote predicate transformers that need not be strict or continuous, and auxiliary variables denote predicate transformers that need not be conjunctive. There is no need to drop monotonicity, however, and it is essential in order for program constructs to be monotonic with respect to refinement. So much for justifying the use of transformers.

The need for monotonic predicates (updeals) on posets only obtrudes for higher-order languages. Because prescriptions and auxiliaries are commands, they can occur as bodies of procedures. (Indeed, the idea of refinement calculus is particularly interesting for object oriented programs because an abstract class may be given as a record of procedures, some of which have specifications for bodies.) Because procedural types internalize programs as data, the refinement order on programs needs to be taken into account for predicates. As an example, suppose we have a refinement $p \sqsubseteq p'$. Then $v := p$ should be refined by $v := p'$, but the postcondition $v = p$ would make an undesired distinction between the two assignments; the problem occurs with any postcondition that is not upward closed. But in practice, the basic predicate for a procedure variable should be a form of specification, and specifications are upward closed with respect to $\sqsubseteq$; such higher-order predicates are formalized in [25]. The limited treatment of higher-order programs in [11] also restricts predicates to be upward closed.

The data type of procedures must include all transformers, if general laws of, e.g. currying and uncurrying, are to hold for the language including specification constructs [20, 22]. On the other hand, if the final program $p'$ in a development $p \sqsubseteq \cdots \sqsubseteq p'$ is in a state space that includes procedure types, the infeasible procedures should be removed from the type by a final data refinement step to an interpretation of procedure types that only includes only healthy transformers. This is shown in [23] for a simpler language and it works as well for Io.

## 7. Definition of the induced coupling

Assume that for each built-in type $B$ we are given posets $[\![B]\!]$ and $[\![B]\!]'$. This determines an interpretation $[\![\theta]\!]$ for each $\theta$ by the definitions in Table 4. We write $[\![\theta]\!]'$ for the interpretation determined the same way from the posets $[\![B]\!]'$. Assume further that for each $B$ an ideal $R_B \subseteq [\![B]\!] \times [\![B]\!]'$ is given. This section defines ideals $R_\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ by

induction on the structure of $\theta$.

- $R_\pi$ and $R_\Gamma$ are instances of (5) (and are ideals by (19)):

$$\sigma\ R_\pi\ \sigma' \equiv (\forall x : x \in dom.\pi : \sigma.x R_{\pi.x} \sigma'.x) \quad \text{for } \sigma \in [\![\pi]\!],\ \sigma' \in [\![\pi]\!]'$$

$$\eta\ R_\Gamma\ \eta' \equiv (\forall Q : Q \in dom.\Gamma : \eta.Q R_{\Gamma.Q} \eta'.Q) \quad \text{for } \eta \in [\![\Gamma]\!],\ \eta' \in [\![\Gamma]\!]'$$

  For brevity, $R_{\pi,x:T}$ is often written $R_{\pi,x}$. The following facts are straightforward to prove, for all $\pi, x, f, f'$:

$$\langle R_{\pi,x}\rangle.(\varphi \dagger x) \subseteq \langle R_\pi\rangle.\varphi \dagger x \quad \text{for all } \varphi \in \mathscr{U}[\![\pi]\!] \tag{26}$$

$$f; \langle R_\pi\rangle \sqsubseteq \langle R_\pi\rangle; f' \Rightarrow (f \otimes id_x); \langle R_{\pi,x}\rangle \sqsubseteq \langle R_{\pi,x}\rangle; (f' \otimes id_x) \tag{27}$$

- For commands, define relation $R_{\mathbf{com}(\pi)}$ from transformers over $[\![\pi]\!]$ to transformers over $[\![\pi]\!]'$ by

$$f R_{\mathbf{com}(\pi)} f' \equiv f; \langle R_\pi\rangle \sqsubseteq \langle R_\pi\rangle; f' \tag{28}$$

  as in (16). This is an ideal: if $g \sqsubseteq f$ and $f' \sqsubseteq g'$ then $g; \langle R_\pi\rangle \sqsubseteq \langle R_\pi\rangle; g'$ by monotonicity of ";" and transitivity of $\sqsubseteq$. The following characterization, which follows directly from the definitions, is used in some proofs:

$$f R_{\mathbf{com}(\pi)} f' \equiv (\forall \sigma, \sigma', \varphi : \sigma R \sigma' : \sigma \in f.\varphi \Rightarrow \sigma' \in f'.(\langle R\rangle.\varphi)) \tag{29}$$

  From (27) we obtain a fact used later:

$$f R_{\mathbf{com}(\pi)} f' \Rightarrow (f \otimes id_x) R_{\mathbf{com}(\pi,x)} (f' \otimes id_x). \tag{30}$$

- Turning to data types, define $R_{\mathbf{bool}}$ by $k\,R\,k' \equiv k = k'$. It is an ideal because $[\![\mathbf{bool}]\!]$ is discretely ordered. Define $R_{U\times V} = R_U \times R_V$, and define $R_{U\to V}$ by $f\ R_{U\to V} f' \equiv f; R_V \supseteq R_U; f'$ so that $R_{U\to V}$ is an ideal by (19).
- For the most interesting data type, **proc**, define $R_{\mathbf{proc}(x:T\mathbf{var}\,y:U)}$ by

$$(f,z) R (f',z') \equiv (\exists z_0 : z = z_0, z' : f R_{\mathbf{com}(x,y,z)} (f' \otimes id_{z_0}))$$

To show that $R_{\mathbf{proc}(x:T,\mathbf{var}\,y:U)}$ is an ideal, suppose $f, f', z, z'$ are as above, so that $z = z_0$, $z'$ and $f R_{\mathbf{com}(x,y,z)} (f' \otimes id_{z_0})$. Consider procedure values such that $(g,w) \preccurlyeq (f,z)$ and $(f',z') \preccurlyeq (g',w')$. By definition of $\preccurlyeq$, we have some $w_0, z_0'$ such that $g \sqsubseteq f \otimes id_{w_0}$, $w = (w_0, z)$, $f' \sqsubseteq g' \otimes id_{z_0'}$, and $z' = (z_0', w')$.

Using the obvious fact $id \otimes id = id$, observe that

$$\begin{array}{ll}
 & f R_{\mathbf{com}(x,y,z)} (f' \otimes id_{z_0}) \\
\Rightarrow & (f \otimes id_{w_0}) R_{\mathbf{com}(x,y,z,w_0)} (f' \otimes id_{z_0,w_0}) \quad \text{fact (30)} \\
\Rightarrow & g R_{\mathbf{com}(x,y,z,w_0)} (f' \otimes id_{z_0,w_0}) \quad g \sqsubseteq f \otimes id_{w_0}, R \text{ ideal} \\
\Rightarrow & g R_{\mathbf{com}(x,y,z,w_0)} (g' \otimes id_{z_0',z_0,w_0}) \quad f' \sqsubseteq g' \otimes id_{z_0'}, R \text{ ideal}
\end{array}$$

Also $w = (w_0, z) = (w_0, z_0, z') = (w_0, z_0, z_0', w')$, so taking $z_0$ to be $w_0, z_0, z_0'$ in the definition of $R$ yields $(g,w)\,R\,(g',w')$.

## 8. Soundness and preservation

In this section we assume given a pair $[\![-]\!], [\![-]\!]'$ of interpretations connected by $R$ as defined in the preceding section. We prove soundness using an identity extension lemma, and then prove preservation. We sometimes omit $\Gamma, \pi, \eta, T$ and subscripts on $R$. In $[\![\ldots]\!]\ R\ [\![\ldots]\!]'$ we omit context on the right side as the typings are the same on both sides.

**Definition 2.** For coupling $R$ to be *identical except for* $D$ means that both (i) $[\![B]\!] = [\![B]\!]'$ (as posets) for each base type $B$ except $D$, and (ii) $R_B$ is $\preccurlyeq_B$ for each base type $B$ except $D$.

As discussed in Section 1, this condition does not ensure that, e.g. $[\![\mathbf{proc}()]\!] = \mathbf{proc}()]\!]'$, because procedures can have external variables of type $D$.

**Lemma 3.** *Suppose coupling $R$ is identical except for $D$. Then for any $\theta$ which has no occurrences of $D$ or* **proc** *we have $[\![\theta]\!] = [\![\theta]\!]'$ and $R_\theta = (\preccurlyeq_\theta)$.*

**Proof.** The notation $\preccurlyeq_\theta$ does not indicate whether it is the order on $[\![\theta]\!]$ or $[\![\theta]\!]'$, but $R_\theta = (\preccurlyeq_\theta)$ implies $[\![\theta]\!] = [\![\theta]\!]'$. We show $R_\theta = (\preccurlyeq_\theta)$ by induction on the structure of $\theta$. If $\theta$ is $B$, it must be different from $D$, so $R_B = (\preccurlyeq_B)$ by definition of "identical except for $D$". If $\theta$ is $U \to V$, we have for any $f, f'$

$$
\begin{aligned}
& f R_{U\to V} f' \\
\equiv\ & f; R_V \supseteq R_U; f' && \text{definition of } R_{U\to V} \\
\equiv\ & f; \preccurlyeq \supseteq \preccurlyeq; f' && \text{induction for } U, V \\
\equiv\ & f; \preccurlyeq \supseteq \preccurlyeq; f'; \preccurlyeq && \text{facts (11) and (12)} \\
\equiv\ & f; \preccurlyeq \supseteq f'; \preccurlyeq && (\Rightarrow)\ \preccurlyeq \text{ reflexive}, (\Leftarrow)\ \preccurlyeq \text{ transitive}, f \text{ mono. (17)} \\
\equiv\ & f \preccurlyeq_{U\to V} f' && \text{characterization (18) of } \preccurlyeq
\end{aligned}
$$

If $\theta$ is $\mathbf{com}(\pi)$ we have for all $f, f'$

$$
\begin{aligned}
& f R_{\mathbf{com}(\pi)} f' \\
\equiv\ & f; \langle R_\pi \rangle \sqsubseteq \langle R_\pi \rangle; f' && \text{definition of } R_{\mathbf{com}(\pi)} \\
\equiv\ & f; \langle \preccurlyeq \rangle \sqsubseteq \langle \preccurlyeq \rangle; f' && \text{induction} \\
\equiv\ & f \preccurlyeq_{\mathbf{com}(\pi)} f' && (15),\ \sqsubseteq \text{ is alternate notation for } \preccurlyeq_{\mathbf{com}(\pi)}
\end{aligned}
$$

We omit the remaining cases, which are similar.

By contrast with the syntactic formulation (1) of soundness, the formal statement below does not rename $x$ because we use $[\![-]\!]$ and $[\![-]\!]'$. What is more significant is that in addition to the analog of (1) we need a second statement for Global variables,

which cannot be bound by **var** but would be encapsulated in a module. The semantics would be the same as local variables, as is evident in (vi) below.

**Theorem 4.** *Suppose coupling $R$ is identical except for $D$, and suppose*

(i) $\eta\, R_\Gamma\, \eta'$,
(ii) $[\![\Gamma;\pi \triangleright e : D]\!]_\eta.\sigma\, R_D\, [\![e]\!]'_{\eta'}.\sigma'$ *for all* $\sigma,\sigma'$ *with* $\sigma\, R_\pi\, \sigma'$
(iii) $[\![\Gamma \triangleright p : \mathbf{com}(\pi, x : D)]\!]_\eta\, R_{\mathbf{com}(\pi,x:D)}\, [\![p]\!]'_{\eta'}$
(iv) $\pi$ *has no occurrences of* $D$ *or* **proc**

*Then we have*

(v) $[\![\Gamma \triangleright (\mathbf{var}\, x : D := e \bullet p) : \mathbf{com}(\pi)]\!]_\eta \sqsubseteq [\![(\mathbf{var}\, x : D := e \bullet p)]\!]'_{\eta'}$
(vi) $(\exists x \bullet [\![(x := e;\, p) : \mathbf{com}(\pi, x : D)]\!]_\eta.(\varphi \dagger x)) \subseteq (\exists x \bullet [\![x := e;\, p]\!]'_{\eta'}.(\varphi \dagger x))$ *for all* $\varphi \in \mathscr{U}[\![\pi]\!]$

In (vi), $p$ can contain calls to procedures with externals, but those calls diverge unless their externals are in scope. The only external involving $D$ that can be in scope is $x$. In (v), $x$ must be Local for the typing to be derivable so it cannot be an external.

**Proof.** We show (v) first. Suppose $\sigma \in [\![\pi]\!]$ and $\varphi \in \mathscr{U}[\![\pi]\!]$. We have $\sigma \preccurlyeq \sigma$ by reflexivity, and $\pi$ satisfies the conditions of Lemma 3, which yields $\sigma\, R\, \sigma$. Thus by (ii) and definition of $R$ we have $[\![e]\!].\sigma\ R\ [\![e]\!]'.\sigma$, whence

$$\sigma(x \mapsto [\![e]\!].\sigma)\ R\ \sigma(x \mapsto [\![e]\!]'.\sigma). \tag{31}$$

We also have $\langle R_\pi \rangle.\varphi = \varphi$ because $R_\pi = (\preccurlyeq)$ by Lemma 3, and $\langle \preccurlyeq \rangle = id$ by (15). We obtain $(v)$ by calculating

$$\begin{aligned}
&\sigma \in [\![\Gamma \triangleright (\mathbf{var}\, x : D := e \bullet p) : \mathbf{com}(\pi)]\!]_\eta.\varphi \\
&\equiv \sigma(x \mapsto [\![e]\!].\sigma) \in [\![p]\!]_\eta.(\varphi \dagger x) && \text{sem. } (var)\ (21)\\
&\Rightarrow \sigma(x \mapsto [\![e]\!]'.\sigma) \in [\![p]\!]'_{\eta'}.(\langle R \rangle.(\varphi \dagger x)) && (31),\ (iv) \text{ and } (29)\\
&\Rightarrow \sigma(x \mapsto [\![e]\!]'.\sigma) \in [\![p]\!]'_{\eta'}.(\langle R \rangle.\varphi \dagger x) && \text{fact } (26)\\
&\equiv \sigma(x \mapsto [\![e]\!]'.\sigma) \in [\![p]\!]'_{\eta'}.(\varphi \dagger x) && \langle R \rangle.\varphi = \varphi\\
&\equiv \sigma \in [\![\Gamma \triangleright (\mathbf{var}\, x : D := e \bullet p)]\!]'_{\eta'}.\varphi && \text{sem. } (var)
\end{aligned}$$

For (vi), we show the containment by observing first that for any $\sigma$

$$\begin{aligned}
&\sigma \in (\exists x \bullet [\![x := e;\, p]\!].(\varphi \dagger x))\\
&\equiv (\exists \tau :\ \tau \downarrow x = \sigma :\ \tau \in [\![x := e;\, p]\!].(\varphi \dagger x)) \quad \text{def. } (\exists x \bullet -)\\
&\equiv (\exists \tau :\ \tau \downarrow x = \sigma :\ \tau(x \mapsto [\![e]\!].\tau) \in [\![p]\!].(\varphi \dagger x)) \quad \text{sem. } (seq) \text{ and } (assign)
\end{aligned}$$

By a calculation similar to the one for (v) (using that $[\![e]\!].\tau$ is $[\![e]\!].\sigma$), the last line implies $(\exists \tau : \tau \downarrow x = \sigma : \tau(x \mapsto [\![e]\!]'.\tau) \in [\![p']\!].(\varphi \dagger x))$ whence by semantics we have $\sigma \in (\exists x \bullet [\![x := e; p]\!'.(\varphi \dagger x))$, proving (vi).

**Theorem 5.** *Suppose that for each built-in constant $c : U$ we have $[\![c]\!] R_U [\![c]\!]'$. Then for all expression typings $\Gamma; \pi \triangleright e : T$, command typings $\Gamma \triangleright p : \mathbf{com}(\pi)$, and environments $\eta, \eta'$ with $\eta R \eta'$ we have*

$$[\![\Gamma; \pi \triangleright e : T]\!]_\eta.\sigma \quad R_T \quad [\![\Gamma; \pi \triangleright e : T]\!]'_{\eta'}.\sigma' \quad \text{for all } \sigma, \sigma' \text{ with } \sigma R_\pi \sigma'$$
$$[\![\Gamma \triangleright p : \mathbf{com}(\pi)]\!]_\eta R_{\mathbf{com}(\pi)} [\![\Gamma \triangleright p : \mathbf{com}(\pi)]\!]'_{\eta'}$$

**Proof.** By induction on typing derivations. For each typing rule we show that if the meanings of its hypotheses are $R$-related then so are the meanings of its conclusion. We use without mentioning the fact that all program constructs are monotonic in their expression and command constituents; e.g. $\mathbf{call}\, e(e', w)$ is monotonic in $e$ and in $e'$. Throughout the proof we assume the hypothesis $\eta R \eta'$. We omit the most straightforward cases (see [24]):

(*proc*) Let $A$ abbreviate ($\mathbf{pro}\ x : T\ \mathbf{var}\ y : U \bullet p$). We have to show that $[\![A : \mathbf{proc}(x : T\ \mathbf{var}\ y : U)]\!].\sigma\, R\, [\![A]\!]'.\sigma'$ which, by semantics of (*proc*), is equivalent to $([\![p : \mathbf{com}(x, y, z)]\!], z)\, R\, ([\![p]\!]', z)$. By definition of $R$ that is equivalent to $[\![p : \mathbf{com}(x, y, z)]\!]_\eta\, R_{\mathbf{com}(x,y,z)}\, [\![p]\!]'_{\eta'}$ which holds by induction.

(*assign*) Here we use that $[\![(x := e) : \mathbf{com}(\pi, x : T)]\!]$ is the inverse-image function $[(\lambda\sigma : \sigma \in [\![\pi, x : T]\!] : \sigma(x \mapsto [\![e]\!].\sigma))]$, so we can use the calculus of $[-]$ and $\langle - \rangle$ as follows:

$$\begin{aligned}
&[\![x := e]\!]; \langle R_{\pi,x} \rangle \sqsubseteq \langle R_{\pi,x} \rangle; [\![x := e]\!]' \\
&\equiv [(\lambda\sigma\ ::\ \sigma(x \mapsto [\![e]\!].\sigma))]; \langle R \rangle \sqsubseteq \langle R \rangle; [(\lambda\sigma'\ ::\ \sigma'(x \mapsto [\![e]\!]'.\sigma'))] \\
&\equiv [R]; [(\lambda\sigma\ ::\ \sigma(x \mapsto [\![e]\!].\sigma))) \sqsubseteq [(\lambda\sigma'\ ::\ \sigma'(x \mapsto [\![e]\!]'.\sigma'))]; [R] \qquad \text{by} \quad (13) \\
&\equiv [(\lambda\sigma\ ::\ \sigma(x \mapsto [\![e]\!].\sigma)); R] \sqsubseteq [R; (\lambda\sigma'\ ::\ \sigma'(x \mapsto [\![e]\!]'.\sigma'))] \qquad \text{by} \quad (15) \\
&\equiv (\lambda\sigma\ ::\ \sigma(x \mapsto [\![e]\!].\sigma)); R \supseteq R; (\lambda\sigma'\ ::\ \sigma'(x \mapsto [\![e]\!]'.\sigma')) \qquad \text{by} \quad (14)
\end{aligned}$$

The last line follows by definition of $R_\pi$ and induction for $e$.

(*call*) To avoid confusion we consider a call $e(d, w)$ rather than $e(e', w)$ as in rule (*call*). Using characterization (29) of $R_{\mathbf{com}(\pi)}$, we are to show, for all $\varphi$ and all $\pi$-states $\sigma, \sigma'$ with $\sigma\, R\, \sigma'$, that

$$\sigma \in [\![\mathbf{call}\ e(d, w) : \mathbf{com}(\pi)]\!].\varphi \Rightarrow \sigma' \in [\![\mathbf{call}\ e(d, w)]\!]'.(\langle R \rangle.\varphi) \tag{32}$$

Suppose $\sigma\, R\, \sigma'$. By induction for $d$, using the definitions of $R$ and $\mapsto$, we have

$$\sigma(x, y \mapsto [\![d]\!].\sigma, \sigma.w) R \sigma'(x, y \mapsto [\![d]\!]'.\sigma', \sigma'.w) \tag{33}$$

Define $f, f', z, z'$ by $(f, z) = [\![\pi \triangleright e : \mathbf{proc}(x\ \mathbf{var}\ y)]\!].\sigma$ and $(f', z') = [\![e]\!]'.\sigma'$. By semantics (22), (23) of (*call*), the antecedent of (32) holds just if there is $t$ with $dom.\pi = w, z, t$ and $\sigma \in \mathbf{wpc}.f.z\ [\![d]\!].w.\varphi$. By (24) that is equivalent to

$$\sigma(x, y \mapsto [\![d]\!].\sigma, \sigma.w) \in (f \otimes id_{w,t}).(\varphi \dagger x, y, )(w :\approx \hat{y}) \tag{34}$$

By $\sigma\, R\, \sigma'$ and induction for $e$ we have $[\![\pi \triangleright e]\!].\sigma\, R_{\mathbf{proc}(x\ \mathbf{var}\ y)}[\![e]\!]'.\sigma'$, i.e. (by definition of $R$) $z = z'$ and $f;\langle R_{x,y,z}\rangle \sqsubseteq \langle R_{x,y,z}\rangle; f'$. It follows by (27) that

$$(f \otimes id_{w,t});\langle R_{x,y,z,w,t}\rangle \sqsubseteq \langle R_{x,y,z,w,t}\rangle;(f' \otimes id_{w,t}).$$

By (29) and (28), this says that for all $v \in [\![x,y,z,w,t]\!]$, $v' \in [\![x,y,z,w,t]\!]'$, and $\psi \in \mathscr{U}[\![x,y,z,w,t]\!]$ we have

$$v\ R\ v' \wedge v \in (f \otimes id_{w,t}).\psi \Rightarrow v' \in (f' \otimes id_{w,t}).(\langle R\rangle.\psi). \tag{35}$$

Using (33) and (34) and instantiating (35) by $\psi := (\varphi \dagger x, y)(w :\approx \hat{y})$ and $v := \sigma(x, y \mapsto [\![d]\!].\sigma, \sigma.w)$ and $v' := \sigma'(x, y \mapsto [\![d]\!]'.\sigma', \sigma'.w)$ we get

$$\sigma'(x, y \mapsto [\![d]\!]'.\sigma', \sigma'.w) \in (f' \otimes id_{w,t}).(\langle R\rangle.((\varphi \dagger x, y)(w :\approx \hat{y}))). \tag{36}$$

We claim

$$(\langle R\rangle.\varphi \dagger x, y)(w :\approx \hat{y}) \supseteq \langle R\rangle.((\varphi \dagger x, y)(w :\approx \hat{y})). \tag{37}$$

Using the claim and monotonicity of $f' \otimes id$, (36) implies

$$\sigma'(x, y \mapsto [\![d]\!]'.\sigma', \sigma'.w) \in (f' \otimes id_{w,t}).(\langle R\rangle.\varphi \dagger x, y)(w :\approx \hat{y}))$$

which, by semantics of (*call*) and definition of **wpc**, is equivalent to the consequent in (32). It remains to prove (37), to which end observe for any $\rho'$

$$\begin{array}{ll}
& \rho' \in \langle R\rangle.((\varphi \dagger x, y)(w :\approx \hat{y})) \\
\equiv & (\exists \rho :\ \rho\ R\ \rho' :\ \rho \in (\varphi \dagger x, y)(w :\approx \hat{y})) \quad \text{definition of } \langle -\rangle \\
\equiv & (\exists \rho :\ \rho\ R\ \rho' :\ \rho(w \mapsto \rho.y)\!\downarrow\! x, y \in \varphi) \quad \text{definitions of } :\approx, \dagger \\
\Rightarrow & (\exists \tau :\ \tau\ R\ \rho'(w \mapsto \rho'.y)\!\downarrow\! x, y :\ \tau \in \varphi) \quad \text{Note below} \\
\equiv & \rho'(w \mapsto \rho'.y)\!\downarrow\! x, y \in \langle R\rangle.\varphi \quad \text{definition of } \langle -\rangle \\
\equiv & \rho' \in (\langle R\rangle.\varphi \dagger x, y)(w :\approx \hat{y}) \quad \text{definitions of } \dagger, :\approx
\end{array}$$

**Note**. Given any $\rho$ witnessing the antecedent, taking $\tau$ to be $\rho(w \mapsto \rho.y)\!\downarrow\! x, y$ gives $\rho\, R\, \rho' \Rightarrow \tau\, R\ (\rho'(w \mapsto \rho'.y)\!\downarrow\! x, y)$ by definitions of $R, \mapsto, \downarrow$.

(*var*) Let $\varphi \in \mathscr{U}[\![\pi]\!]$, $\sigma \in [\![\pi]\!]$, and $\sigma' \in [\![\pi]\!]'$ with $\sigma\, R\, \sigma'$. By induction for $e$, it follows from $\sigma\, R\, \sigma'$ that $[\![e]\!].\sigma\, R\, [\![e]\!]'.\sigma'$, whence we have

$$\sigma(x \mapsto [\![e]\!].\sigma)\ R_{\pi,x}\ \sigma'\ (x \mapsto [\![e]\!]'.\sigma') \tag{38}$$

by definition of $R_{\pi,x}$. Now, we have

$$\begin{array}{ll}
& \sigma \in [\![(\mathbf{var}\, x :\ T := e \bullet p) :\ \mathbf{com}(\pi)]\!].\varphi \\
\equiv & \sigma(x \mapsto [\![e]\!].\sigma) \in [\![p :\ \mathbf{com}(\pi, x)]\!].(\varphi \dagger x) \quad \text{sem. } (var)\ (21) \\
\Rightarrow & \sigma'(x \mapsto [\![e]\!]'.\sigma') \in [\![p :\ \mathbf{com}(\pi, x)]\!]'.(\langle R\rangle.(\varphi \dagger x)) \quad (38),\ \text{induction for } p \\
\Rightarrow & \sigma'(x \mapsto [\![e]\!]'.\sigma') \in [\![p :\ \mathbf{com}(\pi, x)]\!]'.(\langle R\rangle.\varphi \dagger x) \quad (26),\ [\![p]\!]' \text{ mono.} \\
\equiv & \sigma' \in [\![(\mathbf{var}\, x :\ T := e \bullet p) :\ \mathbf{com}(\pi)]\!]'.(\langle R\rangle.\varphi) \quad \text{sem. } (var)
\end{array}$$

## 9. Extensible records

In this section we extend soundness and preservation to the full language with extensible records. The syntactic classes are augmented as follows. In addition to variable and constant identifiers we assume given a set of field labels (or lists), with typical elements $F$, $G$. Using type assignments $\lambda$ to labels, we add extensible records to the grammar of types.

$$T ::= \mathbf{record}(\lambda) \qquad \lambda ::= \emptyset \mid \lambda, F : T$$

We identify types up to re-arrangement of typings, e.g. $\mathbf{record}(F : T, G : U)$ is the same as $\mathbf{record}(G : U, F : T)$. Coupling types are extended to include type assignments to labels: $\theta ::= \lambda$. The record construct does not increase rank, i.e. $rank.(\mathbf{record}(F : T)) = rank.T$. There is no subsumption rule, but the effect of subsumption is embodied in rules like (*assign*) and (*call*).[8] The structural *subtype* relation $\Subset$ can now be defined as follows:

- $\lambda \Subset \lambda'$ iff $dom.\lambda' \subseteq dom.\lambda$ and $\lambda.F \Subset \lambda'.F$ for each $F \in dom.\lambda'$, and moreover $rank.(\lambda'.G) \leqslant rank.(\lambda.F)$ where $G = dom.\lambda' - dom.\lambda$.
- $T \Subset T'$ iff $T$ and $T'$ are identical; or $T, T'$ are record types $\mathbf{record}(\lambda), \mathbf{record}(\lambda')$ with $\lambda \Subset \lambda'$; or $T$ is $U \times V$ and $T'$ is $U' \times V'$ with $U \Subset U'$ and $V \Subset V'$.

Absence of non-trivial subtyping for function and **proc** types simplifies the semantics a little: No coercions are needed, because $T \Subset T'$ implies $[\![T]\!] \subseteq [\![T']\!]$ for all data types $T$. (Although $\Subset$ is also defined on data type assignments $\lambda$ to labels, it is not the case that $[\![\lambda]\!] \subseteq [\![\lambda']\!]$ when $\lambda \Subset \lambda'$.) There are no non-trivial subtyping relations between built-in types. A typical application might involve a base type $B$ with the set $[\![B]\!]$ being the set $[\![(\mathbf{int} \to \mathbf{int}) \times \mathbf{record}(F : \mathbf{int}, G : \mathbf{proc}(\ldots))]\!]$ but ordered discretely.

The grammar of expressions and commands is augmented as follows:

$$e ::= e.F \mid e(F : e) \mid (\mathbf{rcd}\ F = e) \quad \text{selection, update, record formation}$$

$$K ::= (\mathbf{with}\ x : T\ \mathbf{do}\ p) \quad \text{type guard}$$

The typing rules are in Table 7. There is no selective update of data structures; as usual, assignment $r.F := e$ to a record component is treated as assignment $r := r(F : e)$ of an updated record. (The presence of function types allows arrays to be treated the same way, as a built-in tuple.) Operationally, the command $(\mathbf{with}\ x : T'\ \mathbf{do}\ p)$ executes $p$ if $x$ initially has type $T'$, otherwise it aborts. The semantics for records is in Table 8. In semantics of (*record*), $\{F \mapsto \ldots\}$ is notation for formation of an $F$-labelled tuple. In [25], $\mathbf{record}(\lambda)$ denotes the set of all tuples with at least the fields designated by $\lambda$ and possibly more (this is why subtypes are, semantically, subsets). Here, to facilitate

[8] This style of type-system is commonly used in type-theoretic work to avoid algorithmic difficulty of unrestricted subsumption. Subsumption is not an admissible rule for our system, for the trivial reason that the variable-introduction rule gives only its declared type. Our system is convenient in that derivations are unique.

Table 7
Rules for record constructs

$$\frac{\Gamma;\pi \rhd e : \mathbf{record}(\lambda, F:T)}{\Gamma;\pi \rhd e.F : T}(select) \qquad \frac{\Gamma;\pi \rhd e : T}{\Gamma;\pi \rhd (\mathbf{rcd}\, F = e) : \mathbf{record}(F:T)}(record)$$

$$\frac{\Gamma;\pi \rhd e : \mathbf{record}(\lambda, F:T) \quad \Gamma;\pi \rhd e' : T' \quad T' \Subset T}{\Gamma;\pi \rhd e(F:e') : \mathbf{record}(\lambda, F:T)}(update)$$

$$\frac{\Gamma \rhd p : (\mathbf{com}(\pi, x:T') \quad T' \Subset T}{\Gamma \rhd (\mathbf{with}\ x:T'\ \mathbf{do}\ p) : \mathbf{com}(\pi, x:T)}(with)$$

Table 8
Semantics of record constructs

$$\begin{aligned}
[\![\mathbf{record}(\lambda)]\!] &= \{(t,\lambda') \mid \lambda' \Subset \lambda \wedge t \in [\![\lambda']\!]\} \\
(t,\lambda) \preccurlyeq (u,\mu) &\equiv \mu \Subset \lambda \wedge (\forall F : F \in dom.t : t.F \preccurlyeq u.F) \\
[\![\lambda]\!] &= \{t \mid dom.t = dom.\lambda \wedge (\forall F : F \in dom.t : t.F \in [\![\lambda.F]\!])\} \\
t \preccurlyeq u &\equiv (\forall F : F \in dom.t : t.F \preccurlyeq u.F) \\
[\![\Gamma;\pi \rhd e.F : T]\!]_\eta.\sigma &= t.F \qquad (select) \\
\quad \text{where } (t,\lambda') &= [\![\Gamma;\pi \rhd e : \mathbf{record}(\lambda, F:T)]\!]_\eta.\sigma \\
[\![\Gamma;\pi \rhd (\mathbf{rcd}\ F = e) : \mathbf{record}(F:T)]\!]_\eta.\sigma & \qquad (record) \\
\quad = (\{F \mapsto [\![\Gamma;\pi \rhd e : T]\!]_\eta.\sigma\},\ F:T) & \\
[\![\Gamma;\pi \rhd e(F:e') : \mathbf{record}(\lambda, F:T)]\!]_\eta.\sigma &= (t(F \mapsto [\![\Gamma;\pi \rhd e : T]\!]_\eta.\sigma),\ \lambda') \qquad (update) \\
\quad \text{where } (t,\lambda') &= [\![\Gamma;\pi \rhd e : \mathbf{record}(\lambda, F:T)]\!]_\eta.\sigma \\
[\![\Gamma \rhd (\mathbf{with}\, x:T'\ \mathbf{do}\ p) : \mathbf{com}(\pi, x:T)]\!]_\eta.\varphi & \qquad (with) \\
\quad = [\![\pi, x:T']\!] \cap [\![\Gamma \rhd p : \mathbf{com}(\pi, x:T')]\!]_\eta.(\varphi \cap [\![\pi, x:T']\!]) &
\end{aligned}$$

definition of the induced coupling, we pair tuples with their typings, rather like type tags used in implementations of record type extension.

Both soundness (Theorem 4) and identity extension (Lemma 3) hold for the extended language, with the proviso in both cases that the state space or coupling type has no record types (just as it has no procedures). Indeed, the proofs go through without change. As with procedure types, a coupling can be identical except for $D$ (Definition 2) yet, e.g. $[\![\mathbf{record}()]\!] \neq [\![\mathbf{record}()]\!]'$ due to extended fields of type $D$.

The induced coupling for label assignments is like that for states and environments: $t\, R_\lambda\, t' \equiv (\forall F : F \in dom.\lambda : t.F\ \ R_{\lambda.F}\, t'.F)$. For $R_{\mathbf{record}(\lambda'')}$, a record value $(t',\lambda')$ simulates $(t,\lambda)$ if $\lambda'$ has at least the fields of $\lambda$ and for each of those fields $F$ the value $t'.F$ simulates $t.F$:

$$(t,\lambda)\ R_{\mathbf{record}(\lambda'')}\ (t',\lambda') \equiv \lambda' \Subset \lambda \wedge tR_\lambda(t' \downarrow (dom.\lambda' - dom.\lambda))$$

Note that for field $F$, $\lambda'.F$ can be a subtype of $\lambda.F$, so we choose the relation $R_{\lambda.F}$ on the supertype. Because $\lambda'.F \Subset \lambda.F$ implies $[\![\lambda'.F]\!] \subseteq [\![\lambda.F]\!]$, values of the subtype are related by $R_{\lambda.F}$ as needed. The ideal property of $R_{\mathbf{record}(\lambda'')}$ follows from transitivity of $\Subset$ and the ideal property of each $R_{\lambda.F}$.

The preservation Theorem 5 extends to the full language. We omit the straightforward proofs for field selection, update, and record formation. We show preservation for **with** as follows. Suppose $T' \Subset T$. By definition of $R_\pi$ we have $\langle R_\pi\rangle.[\![\pi]\!] \subseteq [\![\pi]\!]'$ for any $\pi$. Now for any $\varphi$ we have (using monotonicity of $[\![p]\!]'$)

$$
\begin{aligned}
&\langle R\rangle.([\![(\mathbf{with}\, x : T' \ \mathbf{do}\ p) : \mathbf{com}(\pi, x : T)]\!].\varphi) \\
&= \langle R\rangle.([\![\pi, x : T']\!] \cap [\![p : \mathbf{com}(\pi, x : T')]\!].(\varphi \cap [\![\pi, x : T']\!])) && \text{sem. } (with) \\
&\subseteq \langle R\rangle.[\![\pi, x : T']\!] \cap \langle R\rangle.([\![p : \mathbf{com}(\pi, x : T')]\!].(\varphi \cap [\![\pi, x : T']\!])) && \langle R\rangle \text{ monotonic} \\
&\subseteq [\![\pi, x : T']\!]' \cap \langle R\rangle.([\![p : \mathbf{com}(\pi, x : T')]\!].(\varphi \cap [\![\pi, x : T']\!])) && \langle R\rangle.[\![\pi]\!] \subseteq [\![\pi]\!]' \\
&\subseteq [\![\pi, x : T']\!]' \cap [\![p : \mathbf{com}(\pi, x : T')]\!]'.(\langle R\rangle.(\varphi \cap [\![\pi, x : T']\!])) && \text{induction for } p \\
&\subseteq [\![\pi, x : T']\!]' \cap [\![p : \mathbf{com}(\pi, x : T')]\!]'.(\langle R\rangle.\varphi \cap [\![\pi, x : T']\!]') && \langle R\rangle.[\![\pi]\!] \subseteq [\![\pi]\!]' \\
&= [\![(\mathbf{with}\, x : T' \ \mathbf{do}\ p) : \mathbf{com}(\pi, x : T)]\!].(\langle R\rangle.\varphi) && \text{sem. } (with)
\end{aligned}
$$

## 10. Examples

Substantial examples of data refinement in development of first-order programs can be found in many places, e.g. [16]. This section gives trivial but typical examples of data refinement for program development, followed by examples of data refinement used to prove general laws. Environments are omitted in the examples, and we assume **int** is a built-in type with standard semantics.

### 10.1. Data refinement

Consider built-ins $D$, $init : D$, and $inc : D \to \mathbf{int}$ in

$$(\mathbf{var}\, x : D := init \bullet y := y + inc(x)) : \mathbf{com}(y : \mathbf{int}). \tag{39}$$

Our first interpretation takes $[\![D]\!] = \{true, false\}$, $[\![init]\!] = true$, and $[\![inc]\!]$ is the function that returns 1 if its argument is true and 0 if false. Our second interpretation takes $[\![D]\!]' = \mathbb{Z}$, $[\![init]\!]' = 1$, and $[\![inc]\!]'$ sends positive $k$ to 1 and $k \leqslant 0$ to 0. Clearly the effect of (39) under either interpretation is to increase $y$ by one. We show $[\![(39)]\!] \sqsubseteq [\![(39)]\!]'$ by data refinement (in fact the reverse $\sqsupseteq$ can be shown similarly). Define $R_D \subseteq [\![D]\!] \times [\![D]\!]'$ by $j\, R_D\, k$ iff $(j = true \equiv k > 0)$. This is an ideal because both interpretations of $D$ are ordered discretely. The other built-ins, like **int**, should have the identity coupling. It is immediate that $[\![init]\!]\, R_D\, [\![init]\!]'$. For the other built-in constant we have $[\![inc : D \to \mathbf{int}]\!]\, R_{D \to \mathbf{int}}\, [\![inc : D \to \mathbf{int}]\!]'$ because for any $j, k$ with $j\, R_D\, k$ we have $[\![inc]\!].j\ R_{\mathbf{int}}\ [\![inc]\!]'.k$ (that is, $[\![inc]\!].j = [\![inc]\!]'.k$). Thus by preservation (Theorem 5) we have $[\![y := y + inc(x)]\!]\ R_{\mathbf{com}(x,y)}\ [\![y := y + inc(x)]\!]'$. So by soundness (Theorem 4(v)) we have $[\![(39)]\!] \sqsubseteq [\![(39)]\!]'$.

To illustrate the use of stored procedures, here is a phrase of type $\mathbf{com}(y:\mathbf{int})$:

$$\mathbf{let}\ Q:\mathbf{proc}() = (\mathbf{pro}\bullet(\mathbf{var}\,x:D := \mathit{init}\bullet y := y + \mathit{inc}(x)))$$
$$\bullet\,(\mathbf{var}\ p:\mathbf{proc}() := Q \bullet \mathbf{call}\ p())$$

The interpretations and coupling above apply to this example as well.

### 10.2. Program equivalence

To illustrate the use of data refinement in proving laws, we turn to the literature on Algol-like languages [14, 26] which has a number of interesting laws that have been used to test semantics for local variables and procedures. Some of these "test equivalences" are applicable to Io despite its syntactic restrictions. Some of the laws can be shown directly, e.g. the following is proved in [25]: $[\![(\mathbf{var}\,x:\mathbf{int}\bullet x:=0;\ p):\mathbf{com}(y:U)]\!] = [\![p:\mathbf{com}(y:U)]\!]$ for all $p$ with $x$ not free. Here we use data refinement to show two other laws.

In Algol (using Io syntax), the first equivalence, of type $\mathbf{com}(\pi)$, is

$$(\mathbf{var}\,z:\mathbf{int} := \mathit{init}\bullet\mathbf{call}\ e(\mathbf{pro}\bullet z := z+1)) = \mathbf{call}\ e(\mathbf{skip})$$

for all procedure constants $e$ declared outside the scope of $z$. In Io, external variable $z$ cannot be bound by **var** so a formulation with $z$ Global is needed; in this way our law is less general than the one for Algol. It is more general in the sense that here $e$ can be a variable. The point of this kind of example for Algol is that $e$ cannot interfere with $z$ for reasons of scope; the usual proof technique is data refinement. Here we simply show the data refinement.

Our formalization of data refinement requires the two programs to have the same structure. We take *inc* to be a built-in constant using a fresh type $D$ that does not appear in $\pi$, and we let $K$ abbreviate the command

$$(z := \mathit{init};\mathbf{call}\ e(\mathit{inc})):\mathbf{com}(\pi, z:D)$$

which we treat under two interpretations. In the first, $[\![D]\!]=\mathbb{Z}$, $[\![\mathit{init}:D]\!]=0$, and $[\![\mathit{inc}:\mathbf{proc}()]\!]=(f,z)$ with $f.\varphi=\varphi(z:\approx(\lambda i:\ :i+1))$. In effect, *inc* is interpreted as $(\mathbf{pro}\bullet z:=z+1)$. The second interpretation takes *inc* to be **skip** on a trivial state space: $[\![D]\!]'=\{*\}$, $[\![\mathit{init}:D]\!]'=*$, and $[\![\mathit{inc}]\!]'=(\mathit{id}_{\pi,z},z)$. For any $e:\mathbf{proc}(y:\mathbf{proc}())$, we claim for all $\varphi\in\mathscr{U}[\![\pi]\!]$ that $(\exists z\bullet[\![K]\!].(\varphi\dagger z))\subseteq(\exists z\bullet[\![K]\!]'.(\varphi\dagger z))$. We prove the claim; it is also straightforward to prove the reverse inclusion. By soundness (Theorem 4(vi)), the inclusion holds provided that there is some $R_D$ with $[\![\mathit{init}]\!]R_D[\![\mathit{init}]\!]'$ and $[\![\mathbf{call}\ e(\mathit{inc})]\!]R_{\mathbf{com}(\pi,z)}[\![\mathbf{call}\ e(\mathit{inc})]\!]'$. Define $R_D$ to be everywhere true. Clearly $[\![\mathit{init}]\!]R_D[\![\mathit{init}]\!]'$. Now $[\![\mathit{inc}]\!]R_{\mathbf{proc}()}[\![\mathit{inc}]\!]'$ is equivalent to $(f,z)R_{\mathbf{proc}()}(\mathit{id},z)$. By definition (29) of $R$, that is equivalent to $\sigma R\sigma'\wedge\sigma\in f.\varphi\Rightarrow\sigma'\in\mathit{id}.(\langle R\rangle.\varphi)$ for all $\sigma,\sigma',\varphi$, which follows easily by definitions from the assumption $\sigma R\sigma'$. We are done if $[\![e]\!]R[\![e]\!]'$ because then $[\![\mathbf{call}\ e(\mathit{inc})]\!]R[\![\mathbf{call}\ e(\mathit{inc})]\!]'$ by preservation (Theorem 5). For $[\![e]\!]R[\![e]\!]'$ it is sufficient but not necessary for $z$ not to be free (in the ordinary sense) in $e$. The result holds

provided $e$ contains no access to $z$ except via *init* and *inc*. It could fail if there are other built-ins that access $z$ but are not simulated. If $e$ is a variable, this non-interference property can be expressed as a precondition on $e$ [Nau00] rather than a hypothesis for the law.

Our final example shows equivalence of two interpretations of the following command, abbreviated $L:(z:=\mathit{init};\ \mathbf{call}\ e(\mathit{flick},\ \mathit{test})):\mathbf{com}(\pi, z:D)$, where $e$ is any procedure $e:\mathbf{proc}(f:\mathbf{proc}(), t:\mathbf{proc}(\mathbf{var}\ y:\mathbf{bool}))$ not involving $z$. Here *init*, *flick*, *test* are built-ins. The idea is that $z$ is a switch object, which can be tested to see whether it has been flicked. Informally, the first interpretation has $D=\mathbf{bool}$, with procedures $(\mathbf{pro}\bullet z:=\mathit{true})$ and $(\mathbf{pro}\ y:\mathbf{bool}\bullet y:=z)$. The second interpretation has $D=\mathbf{int}$, with $(\mathbf{pro}\bullet z:=z+1)$ and $(\mathbf{pro}\ y:\mathbf{bool}\bullet y:=z>0)$. Formally, for the first interpretation let $[\![D]\!]=\{\mathit{true},\mathit{false}\}$ $[\![\mathit{init}]\!]=\mathit{false}$, $[\![\mathit{flick}:\mathbf{proc}()]\!]=(f,z)$ with $f.\varphi=\varphi(z:\approx \mathit{true})$, and $[\![\mathit{test}:\mathbf{proc}(\mathbf{var}\ y:\mathbf{bool})]\!]=(g,z)$ with $g.\varphi=\varphi(y:\approx\hat{z})$. For the second interpretation let $[\![D]\!]'=\mathbb{Z}$, $[\![\mathit{init}]\!]'=0$, $[\![\mathit{flick}]\!]'=(f,z)$ with $f.\varphi=\varphi(z:\approx[\![z+1]\!])$, and $[\![\mathit{test}]\!]'=(g,z)$ with $g.\varphi=\varphi(y:\approx[\![z>0]\!])$. Using $R_D$ defined by $j\,R\,k\equiv(j=\mathit{false}\wedge k=0)\vee(j=\mathit{true}\wedge k>0)$, we can show that for all $\varphi$, $(\exists z\bullet [\![L]\!].(\varphi\dagger z))\subseteq(\exists z\bullet [\![L]\!]'.(\varphi\dagger z))$ in much the same way as the preceding example; so too the reverse inclusion.

## 11. Conclusion

Our main contribution is to justify the use of data refinement for some of the languages used in practice, in particular, to show soundness and preservation for data refinement in the presence of stored procedures and record subtyping. The restriction on external variables is in accord with languages including C, C++, Modula-3, Ada, and Oberon; it is such a dramatic simplification that we can treat stored procedures (which are difficult in Algol, e.g. [27]). The stratification we impose on procedures in rule (*proc*) is less natural; but it can be circumvented using dummy parameters, as there are closed terms at every procedure type. Stratification makes possible a set-theoretic semantics that models unbounded angelic and demonic non-determinacy and unobservable predicates. For example, correctness statements are used in [25] as predicates at higher types. Using transformers and taking fixpoints past $\omega$ (as in [7] and most work in transformer semantics) simplifies the conditions on couplings (e.g. chain-completeness is not needed). That makes for simpler derivations of concrete programs from abstract ones [17].

The semantic definitions allow arbitrary sets as predicates [25]. Using that alternative semantics, and dropping the restriction that couplings are ideals, one can still prove soundness and preservation of data refinement.[9] But the alternative is unattractive because some important refinement laws fail [11, 20, 25]. For the language without

[9] This does not contradict [19] because the latter has products and exponents of transformers in a more general form than in Io.

specification constructs, it is straightforward to drop the rank restriction and restrict built-in types to be CPOs; a state-transformer semantics then poses no difficulty.

The examples in Section 10 from the Algol literature illustrate phenomena involving hidden local state. But Oberon-like languages express hiding using separate module constructs rather than local variables. Our semantics extends easily to such constructs; the semantics is the basis for current work on refinement calculus for class-based object-oriented programming [3].

Although it appears that Io preserves backward simulations (recall (10)), the details have not been checked. The question of completeness remains open.

## Acknowledgements

I had helpful discussions of data refinement with a number of people including Paul Gardiner, Tony Hoare, Carroll Morgan, and Jeff Sanders. John Power and Yoshiki Kinoshita helped me understand categorical theories of data refinement. Bob Tennant convinced me that although internalization of specifications is successful for some refinement calculi, it should not be institutionalized in a general theory of simulation. Uday Reddy and Peter O'Hearn helped me understand Algol. Special thanks to Uday for spurring me to write up these results, and to anonymous referees for substantial expository improvements. Thanks to Southwestern University for supporting this research, and to NSF grant INT-9813854 for support when the final revisions were made.

## References

[1] K.R. Apt, G.D. Plotkin, Countable nondeterminism and random assignment, J. ACM 33 (1986) pp. 724–767.
[2] R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Graduate Texts in Computer Science, Springer, Berlin, 1998.
[3] A.L.C. Cavalcanti, D.A. Naumann, in: A weakest precondition semantics for an object-oriented language of refinement, FM'99-Formal Methods, vol. II, Lecture Notes in Computer Science, vol. 1709, Springer, Berlin, 1999.
[4] W. Chen, J.T. Udding, Towards a calculus of data refinement, Proceedings, Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 375, Springer, Berlin, 1989, pp. 197–218.
[5] E. Denney, A theory of program refinement, Dissertation, University of Edinburgh, 1998.
[6] W.-P. de Roever, K. Engelhardt, Data Refinement: Model-Oriented Proof Methods and their Comparison, Cambridge University Press, Cambridge, 1998.
[7] P. Gardiner, C. Morgan, Data refinement of predicate transformers, Theoret. Comput. Sci. 87 (1991) 143–162.
[8] P.H.B. Gardiner, C.E. Martin, O. de Moor, An algebraic construction of predicate transformers, Sci. Comput. Programming 22 (1994) 21–44.
[9] P.H.B. Gardiner, C. Morgan, A single complete rule for data refinement, Formal Aspects Comput. 5 (4) (1993) 367–382.
[10] J. He, C.A.R. Hoare, J.W. Sanders, in: Data refinement refined (resumé) European Symp. on Programming, Lecture Notes in Computer Science, vol. 213, Springer, Berlin, 1986.
[11] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1998.

[12] F. Honsell, J. Longley, D. Sannella, A. Tarlecki, Constructive data refinement in typed lambda calculus, FOSSACS 2000, Lecture Notes in Computer Science, vol. 1784, Springer, Berlin, 2000, pp. 149–164.
[13] Y. Kinoshita, J. Power, Enriched categories and data refinement, Tech. Report 94-23, University of Edinburgh Department of Computer Science, 1994.
[14] A.R. Meyer, K. Sieber, Towards fully abstract semantics for local variables: preliminary report, Proc. 15th POPL, 1988, pp. 191–203.
[15] J.C. Mitchell, Type systems for programming languages, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, MIT Press Cambridge, MA/Elsevier, Amsterdam, 1990, pp. 365–458.
[16] C. Morgan, Programming from Specifications, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1994.
[17] C. Morgan, P.H.B. Gardiner, Data refinement by calculation, Acta Inform. 27 (1990) 481–503.
[18] D.A. Naumann, Two-categories and program structure: data types, refinement calculi, and predicate transformers, Dissertation, University of Texas at Austin, 1992.
[19] D.A. Naumann, Data refinement, call by value, and higher order programs, Formal Aspects Comput. 7 (1995) 652–662.
[20] D.A. Naumann, Predicate transformers and higher order programs, Theoret. Comput. Sci. 150 (1995) 111–159.
[21] D.A. Naumann, Beyond fun: order and membership in polytypic imperative programming, in: J. Jeuring (Ed.), Mathematics of Program Construction, Lecture Notes in Computer Science, vol. 1422, Springer, Berlin, 1998, pp. 286–314.
[22] D.A. Naumann, A categorical model for higher order imperative programming, Math. Struct. Comput. Sci. 8 (4) (1998) 351–399.
[23] D.A. Naumann, Towards squiggly refinement algebra, in: D. Gries, W.-P. de Roever (Eds.), Programming Concepts and Methods, Chapman & Hall, London, 1998,, pp. 346–365.
[24] D.A. Naumann, Validity of data refinement for a higher order imperative language, Tech. Report 9905, Computer Science, Stevens Institute of Technology, 1999.
[25] D.A. Naumann, Predicate transformer semantics of a higher order imperative language with record subtyping, Sci. Comput. Programming 41 (2001) 1–51.
[26] P.W. O'Hearn, R.D. Tennent, Parametricity and local variables, J. ACM 42 (3) (1995) 658–709.
[27] A.M. Pitts, Reasoning about local variables with operationally-based logical relations, Proc. 9th Annual IEEE Symp. on Logic in Computer Science, 1996.
[28] G.D. Plotkin, Dijkstra's predicate transformers and Smyth's powerdomains, in: D. Bjørner (Ed.), Abstract Software Specifications, Lecture Notes in Computer Science, vol. 86, Springer, Berlin, 1979,, pp. 527–553.
[29] M.B. Smyth, Topology, in: S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 1, Oxford, University Press 1992, pp. 641–761.
[30] R.D. Tennant, Correctness of data representations in Algol-like languages, in: A.W. Roscoe (Ed.), A Classical Mind: Essays Dedicated to C.A.R. Hoare, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[31] N. Wirth, The programming language Oberon, Software – Practice Experience 18 (7) (1988) pp. 671–690.
[32] H. Yang, U.S. Reddy, On the Semantics of Refinement Calculi, FOSSACS 2000, Lecture Notes in Computer Science, vol. 1784, Springer, Berlin, 2000, pp. 359–374.