State-level and value-level simulations in data refinement

William Leal [1,2], Anish Arora [1,3]

Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, USA

Abstract

Simulations are a popular way to show data refinement. Simulations that have been proposed are either state level, relating concrete to abstract states in a given state space, or value level, relating individual concrete to abstract values and hence holding for all state spaces. Value-level simulations are less complex and easier to use, but the extent of their completeness has not been well studied. We show that in fact known value-level simulations are in general incomplete but are complete when operations are limited to a single argument.

Key words: Data refinement, program correctness, formal verification, components

0 Introduction

Suppose we have a program $pgm(\mathcal{A})$ that uses the operations of a data type $\mathcal{A}$. We wish to substitute a more concrete data type $\mathcal{C}$ while guaranteeing that the behavior of $pgm(\mathcal{C})$ will not surprise us. In fact, we would like to know if we can do this for all programs, not just a particular one, in which case we can say that $\mathcal{C}$ refines $\mathcal{A}$. What "not surprised" means varies from one author to another but does include the notion of subsetting of visible behavior. Proving

[1] This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and a grant from Microsoft Research.

[2] E-mail: leal@cis.ohio-state.edu. Sponsored in part by a fellowship from the General Electric Fund's Faculty for the Future Program and by a Multiple-Year Dean's Fellowship provided by The Ohio State University's Graduate School.

[3] Corresponding author. E-mail: anish@cis.ohio-state.edu. Arora is currently on sabbatical leave at Microsoft Research.

this subsetting directly for all programs is hard; a popular and more practical approach is to exhibit a simulation relation between the concrete and abstract state spaces (or alternatively, data values) and then to show that this relation holds for each operation.

Simulations come in many different forms but can be broadly classified as either state-level or value-level. As the names imply, a state-level simulation relates concrete states to abstract states while a value-level simulation relates the values that make up the states. For example, suppose we wish to refine a bag that holds integers with operations *add* and *remove* by an integer stack with operations *push* and *pop*. Given a program, a state-level simulation would relate a given state of that program using the stack to the corresponding states of that same program using the bag; a value-level simulation would instead relate a stack with some configuration to those bags containing the same elements and would similarly relate the integers, in this case with an identity.

Value-level reasoning has two attractions. In the first place, this kind of reasoning is in accord with the usual programming language constructs for data abstraction: modules define data types (bags, queues), not state spaces. By contrast, much of the treatment in the literature on data refinement supposes a particular state space and does not deal with the issue of using a data type in a different state space.

In the second place, the complexity of value-level reasoning is typically less than that of state-level. Suppose we were to fix state spaces for the bag and the queue data types, respectively. To verify that the stack refines the bag we would have to show, for the state-level simulation, that the relation holds for both operations on all instances for all states. For the value-level simulation, we need to show only that the relation holds for the combinations of values mentioned in the operations, since the value-level simulation induces a state space in a pointwise way, a process that is less complex. In addition, a state-level proof for a given state space would assure us of refinement for that state space only; we might then have to repeat the proof for each possible state space, whereas a value-level simulation assures us of refinement for any state space.

The purpose of this paper is to compare the relative power of value-level and state-level simulations. We use for our model a partial-correctness semantics of first-order input-output programs, although the results can be applied to total correctness and to reactive programs. There are two important state-level simulations, forward and backward, and these are known to be complete with respect to refinement [7]. For value-level we propose two analogous simulations, value-level forward and backward, and show by means of an example that the completeness of these simulations depends on the language available. If operations can read and write multiple values, then these value-level simulations are not complete. But if the language is constrained in such a way that each operation has only a single argument, then they are complete. Hence completeness depends on the language restrictions chosen. We also observe that the value-level simulations proposed in the literature for data refinement are all examples of our value-level forward or backward simulations, so known value-level simulations are no more complete than shown here.

The rest of this paper is organized as follows. In Section 1 we give a definition of data type suitable for value-level reasoning and recall from [7] the definition of data refinement and the forward and backward state-level simulations. We then define the value-level forward and backward simulations. In Section 2 we show that these value-level simulations, while sound, are incomplete unless the data types are restricted to have only a single argument. We also discuss a stronger completeness result for a more restricted set-up. In Section 3 we show that other value-level simulations proposed in the literature are examples of our value-level forward and backward simulations. In the interest of space, some of the proofs are not shown but are included in an expanded on-line technical report version [1].

1 Data Types, Data Refinement and Simulations

We begin by giving a definition of a data type as a structure for a signature, similar to [6]. Then we define a state space in the usual way and associate the data type definition with the state space. This gives us a definition of data type that is of the form specified by de Roever and Engelhardt [7] and permits us to compare state-level and value-level results.

Definition 1 (Signature) *A signature is a triple* $\Sigma \stackrel{\text{def}}{=} (v, h, Op)$ *where* v *is a visible sort name,* h *is a hidden sort name, and* Op *is a set of pairs* $(P : w)$*, where* P *is an operation name,* $w \in T^*$ *and* $T = \{v, h\}$*.* □

Note that an operation $(P : w)$ has a domain and co-domain that are both of sort w, consistent with [7]. For ease of exposition we have restricted signatures to a single hidden and a single visible sort. The important distinction is between hidden and visible, and the results presented here would not be affected by adopting multiple hidden or multiple visible sorts. Each of the proofs is easily generalized to multiple sorts.

Definition 2 (Data type) *A data type* A *for signature* Σ *is a* Σ*-structure* $A \stackrel{\text{def}}{=} (D, I, O, F)$ *where*

- *Domain* D *is partitioned into sets* $D.t$ *for each* $t \in T$*.*
- *Initialization* I *is partitioned into relations* $I.v \subseteq D.v \times D.v$ *and* $I.h \subseteq$

$\{\cdot\} \times D.h$ *that initialize visible and hidden values, respectively, where* $\cdot$ *is an anonymous hidden value.*

- *For each operation* $(P : w) \in Op$*,* O *contains a relation* $P \subseteq D.w \times D.w$*, where for* $w \in T^*$*,* $D.w$ *is the cross-product of the domains of the sorts in* w*.*
- *Finalization* F *is partitioned into relations* $F.v \subseteq D.v \times D.v$ *and* $F.h \subseteq D.h \times \{\cdot\}$ *that finalize visible and hidden values, respectively.* $\square$

Without loss of generality we assume that the sort domains are disjoint; if necessary we may subscript every element of each domain with the sort name. Notationally, for $(P : w) \in Op$ and $b' \in D.w$, $P[b'] \stackrel{\text{def}}{=} \{b \mid (b', b) \in P\}$. Data types A and C are *compatible* if they share the same signature and the same visible domain.

A data type A may be placed in the context of a state space and so become a state-level data type. Each of the variables in the state space corresponds to one of the data type sorts and ranges over the domain of the sort. State-level operations are written as $P(\vec{g})$ for each operation $(P : w) \in Op$ and tuple $\vec{g}$ of disjoint variables where the sort of $\vec{g}$ is w. We require disjointness of variables in $\vec{g}$ to avoid aliasing problems for output. Informally, $P(\vec{g})$ is a state-level relation that is the identity for variables not in $\vec{g}$ and uses the relation of P for the values of the variables in $\vec{g}$. A state-level initialization operation is defined based on A's initialization that relates initial visible states to initial composite states and finalization that similarly relates composite final states to visible final states.

More precisely, let R be the set of variables for a state-level data type, with R partitioned into $R.v$ and $R.h$ for the visible and hidden variables, respectively. Then we may define a hidden state space *hss* and a visible state space *vss* in the usual way based on $D.h$ and $D.v$, respectively, and a composite state space *css* as the cross product of the hidden and visible spaces.

We extend the notion of sort to tuples of values and tuples of sort names in the expected way. If $\vec{g}$ is a tuple of unique variables and s is a state, then for any variable y, $s.y$ is the value of y in s and $s.\vec{g}$ is the tuple of values drawn from s according to $\vec{g}$. Now we define a data type for a state space.

Definition 3 (State-level data type) *Let* $A = (D, I, O, F)$ *be a data type and let* R *be the variables for a state space. The state-level data type,* $\mathcal{A}$*, is given by* $\mathcal{A} \stackrel{\text{def}}{=} (R, D, \mathcal{I}, \mathcal{O}, \mathcal{F})$*, or just* $\mathcal{A} \stackrel{\text{def}}{=} (\mathcal{I}, \mathcal{O}, \mathcal{F})$ *when the other parts are understood.* $\mathcal{I}, \mathcal{O}$ *and* $\mathcal{F}$ *are given as:*

- $\mathcal{I} \subseteq vss \times css$ *is a state-level initialization operation defined as the set of all* (s', s) *s.t. if* y *is a visible variable then* $s.y \in I.v[s.y']$ *and if* y *is a hidden variable then* $s.y \in I.h[\cdot]$*.*

- *$\mathcal{O}$ is a set of operations $\{(P(\vec{g}) : w) \mid (P : w) \in Op$ and $\vec{g}$, a vector of disjoint variables, is sort $w\}$. The meaning of $P(\vec{g})$ is given as the set of all state pairs (s', s) s.t. for all variables y of $\mathcal{A}$, if y is not in $\vec{g}$ then $s.y = s'.y$ and otherwise $s.\vec{g} \in P[s'.\vec{g}]$.*
- *$\mathcal{F} \subseteq css \times vss$ is a state-level finalization operation defined as the set of all (s', s) s.t. if y is a visible variable then $s.y \in F.v[s'.y]$ and if y is a hidden variable then $F.h[s'.y] = \{\cdot\}$.* □

Definition 3 is a restriction of the general definition of data type given in [7],[4] and we now recall those definitions of program, data refinement and simulation, adapted to our notation for state-level data types.

Given state-level data type $\mathcal{C}$, $Pgms(\mathcal{C})$ consists of all programs built up out of sequential composition, nondeterministic choice and recursion of operations in $\mathcal{C}$ [7, Section 3.4]. If there is a bijection J from the operations of $\mathcal{C}$ to those of $\mathcal{A}$ and if $pgm(\mathcal{C})$ is a program of $\mathcal{C}$ then $pgm(\mathcal{A})$ is that program of $\mathcal{A}$ obtained by replacing the operations $\mathcal{P}$ of $\mathcal{C}.\mathcal{O}$ with $J(\mathcal{P})$ of $\mathcal{A}.\mathcal{O}$, where the qualifications "$\mathcal{C}$." and "$\mathcal{A}$." are added to distinguish between data types.

Definition 4 (Data refinement) *Let $\mathcal{C}$ and $\mathcal{A}$ be state-level data types derived from data types C and A, respectively. $\mathcal{C}$ is a data refinement (or just refinement) of $\mathcal{A}$ if there is a bijection J as given above s.t. for all programs $pgm(\mathcal{A}) \in Pgms(\mathcal{A})$ the following inclusion holds:*

$$\mathcal{C}.\mathcal{I}; pgm(\mathcal{C}); \mathcal{C}.\mathcal{F} \subseteq \mathcal{A}.\mathcal{I}; pgm(\mathcal{A}); \mathcal{A}.\mathcal{F} \qquad \square$$

Next we recall the definitions of state-level forward and backward simulations (L and L^{-1}, respectively, in the terminology of [7]). For a concise and readable presentation of forward and backward simulations, see [4].

Definition 5 (State-level simulation) *Let $\mathcal{C}$ and $\mathcal{A}$ be state-level data types with a bijection J from the operations of $\mathcal{C}$ to those of $\mathcal{A}$.*

- *There is a state-level forward simulation from $\mathcal{C}$ to $\mathcal{A}$ (denoted $\mathcal{C} \subseteq_{\mathrm{F}} \mathcal{A}$) iff there is $fw \subseteq \mathcal{C}.css \times \mathcal{A}.css$ s.t.*

$$\begin{array}{rll} & \mathcal{C}.\mathcal{I} \subseteq \mathcal{A}.\mathcal{I}; fw^{-1} & \textit{(fw init)} \\ \textit{for all } \mathcal{P} \in \mathcal{C}.\mathcal{O}: & fw^{-1}; \mathcal{C}.\mathcal{P} \subseteq \mathcal{A}.(J(\mathcal{P})); fw^{-1} & \textit{(fw opn)} \\ & fw^{-1}; \mathcal{C}.\mathcal{F} \subseteq \mathcal{A}.\mathcal{F} & \textit{(fw final)} \end{array}$$

[4] In [7], any two data types must have disjoint sets of hidden variables. Since this complicates our discussion needlessly, we allow state-level data types to have the same hidden variables. Also, we use a bijection to relate the operations of two data types rather than indexing the operations. Finally, $(R, D, \mathcal{I}, \mathcal{O}, \mathcal{F})$ may be read as $(R.v, R.h, D.v, D.h, \mathcal{I}, \mathcal{O}, \mathcal{F})$, the format specified by [7].

- *There is a state-level backward simulation from $\mathcal{C}$ to $\mathcal{A}$ (denoted $\mathcal{C} \subseteq_{\mathrm{B}} \mathcal{A}$) iff there is $bk \subseteq \mathcal{C}.css \times \mathcal{A}.css$ s.t.*

$$\mathcal{C}.\mathcal{I}; bk \subseteq \mathcal{A}.\mathcal{I} \qquad \textit{(bk init)}$$

$$\textit{for all } \mathcal{P} \in \mathcal{C}.\mathcal{O}: \quad \mathcal{C}.\mathcal{P}; bk \subseteq bk; \mathcal{A}.(J(\mathcal{P})) \qquad \textit{(bk opn)}$$

$$\mathcal{C}.\mathcal{F} \subseteq bk; \mathcal{A}.\mathcal{F} \qquad \textit{(bk final)} \quad \square$$

We know from [7] that state-level simulations are sound with respect to data refinement. We are now ready to give our definition of value-level simulation. A sorted relation L from C to A is $L_t \subseteq A.D.t \times C.D.t$ for $t \in T$. For $(P : w) \in Op$, L_w is the pointwise extension of L according to the sorts of w: if $w = (t_1, t_2, \ldots, t_n)$ then $L_w = L_{t_1} \times L_{t_2} \times \ldots \times L_{t_n}$.

Definition 6 (Value-level simulation) *Let A and C be compatible data types.*

- *There is a value-level forward simulation from C to A (denoted by $C \subseteq_{\mathrm{LF}} A$) iff there is a sorted relation lfw from C to A s.t.*

$$\textit{for all } t \in T: \quad C.I.t \subseteq A.I.t; lfw^{-1} \qquad \textit{(vlf init)}$$

$$\textit{for all } (P : w) \in Op: \quad lfw_w^{-1}; C.P \subseteq A.P; lfw_w^{-1} \qquad \textit{(vlf opn)}$$

$$\textit{for all } t \in T: \quad lfw^{-1}; C.F.t \subseteq A.F.t \qquad \textit{(vlf final)}$$

- *There is a value-level backward simulation from C to A (denoted by $C \subseteq_{\mathrm{LB}} A$) iff there is a sorted relation lbk from C to A s.t.*

$$\textit{for all } t \in T: \quad C.I.t; lbk \subseteq A.I.t \qquad \textit{(vlb init)}$$

$$\textit{for all } (P : w) \in Op: \quad C.P; lbk_w \subseteq lbk_w; A.P \qquad \textit{(vlb opn)}$$

$$\textit{for all } t \in T: \quad C.F.t \subseteq lbk; A.F.t \qquad \textit{(vlb final)} \quad \square$$

Theorem 7 (Soundness of value-level simulations) *Let C and A be compatible data types and let $\mathcal{C}$ and $\mathcal{A}$ be state-level data types derived from C and A, respectively, using variables R. If there is a value level simulation from C to A then $\mathcal{C}$ refines $\mathcal{A}$.* $\square$

The proof of the theorem is given in the on-line technical report version [1]. The essential idea is that, given $\mathcal{C}$ and $\mathcal{A}$, a state-level simulation may be lifted from a value-level simulation by a pointwise extension based on the variables in R, similar to the pointwise extension L_w above.

2 Value-Level (In)Completeness Results

Assume in the sequel that A and C are compatible data types and that $\mathcal{A}$ and $\mathcal{C}$ are state level data types derived from A and C, respectively, using the same variables R. From [7] we know that state-level forward and backward simulations together are complete, but the construction used in the proof is inconsistent with our value-level setup. Completeness also holds in our setting: that is, we can show that if $\mathcal{C}$ is a data refinement of $\mathcal{A}$ then there is a data type B and a state-level data type $\mathcal{B}$ s.t. $\mathcal{C} \subseteq_{\mathrm{F}} \mathcal{B} \subseteq_{\mathrm{B}} \mathcal{A}$. A proof of this completeness is given in the on-line technical report version [1]. As we show in this section, if we limit ourselves to value-level simulations we do not have completeness except in the case that the data types are so restricted that each operation accepts only a single visible or hidden argument.

Consider Fig. 0. The abstract data type *EagerToss* models an ambitious but incompetent juggler that tries to toss two ready plates (R) so that they are both right side up (Up) or upside down (Dn); he notes which way they came up (tt if both were up, ff if both were down, and either tt or ff if they don't match) but he can't catch them, so they smash at the end (H). A programmer implementing this (*LazyToss*) observes that deciding how they came up can be postponed, so takes two available plates (L), makes them match (M), and then nondetermistically decides whether they were right side up or not (tt or ff) before they are broken (K). In this example, plates are hidden sorts. For *EagerToss*, the domain for plates is $\{R, Up, Dn, H\}$ while for *LazyToss*, the domain is $\{L, M, K\}$. For both, the visible values are three-state Booleans $\{tt, ff, ?\}$, where "?" indicates an undefined value.

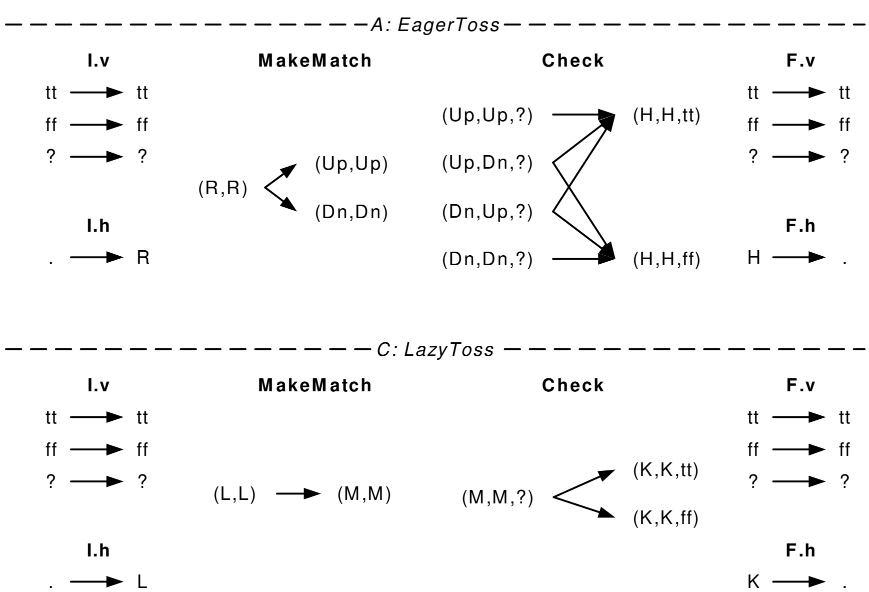


Fig. 0. Tossing Matching Plates: Data Types

LazyToss, C, is a refinement of *EagerToss*, A; a proof is included in the expanded on-line technical report version [1]. Consider a program $pgm(\mathcal{C})$ for some $\mathcal{C}$ s.t. $\mathcal{C}.\mathcal{I}; pgm(\mathcal{C}); \mathcal{C}.\mathcal{F} \neq \emptyset$. Note that in any such non-empty program every hidden variable is matched exactly once and is checked exactly once; variables that are unmatched or unchecked will fail finalization. For any $\mathcal{C}$ and $\mathcal{A}$ we have the invariant that for any $pgm(\mathcal{C})$, if state s is in the codomain of $\mathcal{C}.\mathcal{I}; pgm(\mathcal{C})$ there is state u in the codomain of $\mathcal{A}.\mathcal{I}; pgm(\mathcal{A})$ and for each hidden variable y, if $u.y$ is Up or Dn then $s.y = M$; and for each visible variable z, $u.z = s.z$. Let x and y be distinct hidden variables and z a visible variable, and consider two cases. Recall that, according to Definition 3, arguments to operations must be distinct variables.

- Case 1. x and y were matched together. For $\mathcal{A}$ the matching will nondeterministically give us abstract states u_1' and u_2' where x and y are both Up or both Dn, respectively. In $\mathcal{C}$ the matching gives us a state s' in which the variables are both M. Suppose $s'.z =?$ so that $u'.z =?$. Then applying $\mathcal{A}.Check(x, y, z)$ to the abstract states gives us states u_1 and u_2 where z is tt and $f\!f$, respectively. Applying $\mathcal{C}.Check(x, y, z)$ to s' nondeterministically gives us two states in which z is tt and $f\!f$, respectively. Hence the outcomes are visibly identical.
- Case 2. x and y were not matched together; that is, each was matched with other variables. Then we will have an abstract state u' where x is Up and y is Dn or vice versa. In this case applying $\mathcal{A}.Check(x, y, z)$ to u' nondeterministically gives us states u_1 and u_2 where z is tt and $f\!f$, respectively. In the concrete, x and y are both M in s'. As in the first case, we nondeterministically have two checked states in which z is tt and $f\!f$, respectively, so again the outcomes are visibly identical.

Despite the refinement, there is no value-level simulation between the data types, as the theorem below shows. The problem lies in the fact that a value-level simulation relation would have to relate M in the concrete to both Up and Dn in the abstract. Hence for any value-level backward simulation, lbk, $C.MakeMatch; lbk$ would include $((M, M), (Up, Dn))$. This is not in $lbk; A.MakeMatch$, so it violates the condition for value-level backward simulation. A similar problem occurs for any proposed value-level forward simulation.

Since state-level forward and backward simulations are individually incomplete but jointly complete, we might suppose that value-level forward and backward simulations are also jointly complete. However, this is not the case, as the following theorem shows.

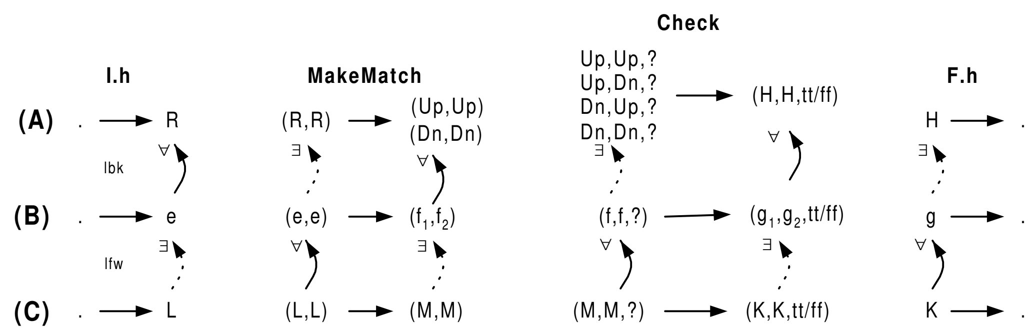
Theorem 8 (Incompleteness of value-level simulations) *Refer to Fig. 0. LazyToss is a refinement of EagerToss but there is no data type B s.t. LazyToss* $\subseteq_{\mathrm{LF}} B \subseteq_{\mathrm{LB}}$ *EagerToss or s.t. LazyToss* $\subseteq_{\mathrm{LB}} B \subseteq_{\mathrm{LF}}$ *EagerToss.*

PROOF. Notationally, let $\vec{c}' \overset{C.P}{\dashrightarrow} \vec{c}$ stand for $(\vec{c}', \vec{c}) \in C.P$ where $(P : w) \in Op$ and $\vec{c}$ is a tuple of values of C of sort w. Note that for value-level forward simulation, $lfw_w^{-1}; C.P \subseteq A.P; lfw_w^{-1}$ is equivalent to $(\vec{a}' \in lfw[\vec{c}'] \wedge \vec{c}' \overset{C.P}{\dashrightarrow} \vec{c} \implies \exists \vec{a} \in lfw_w[\vec{c}] : \vec{a}' \overset{A.P}{\dashrightarrow} \vec{a})$. Likewise, for value-level backward simulation, $C.P; lbk_w \subseteq lbk_w; A.P$ is equivalent to $(\vec{c}' \overset{C.P}{\dashrightarrow} \vec{c} \wedge \vec{a} \in lfw[\vec{c}] \implies \exists \vec{a}' \in lbk_w[\vec{c}'] : \vec{a}' \overset{A.P}{\dashrightarrow} \vec{a})$. Let $mm \overset{\text{def}}{=} (h, h)$, the sort of *MakeMatch*, and $ck \overset{\text{def}}{=} (h, h, v)$, the sort of *Check*.

- **Forward-backward**. Let B be any data type such that lfw is a value-level forward simulation from C to B. We show that there is no value-level backward simulation lbk from B to A. For the sake of contradiction, suppose that lbk is such a value-level backward simulation.

  First we show that $lfw[lbk[tt]] = \{\{tt\}\}$, and similarly for ff and ?, so it suffices to consider the visible values of B to be tt, ff and ?. By (*vlf init*), $\exists a \in lfw[tt] : tt \overset{B.I.v}{\dashrightarrow} a$. By (*vlf final*), $\forall a \in lfw[tt] : a \overset{B.F.v}{\dashrightarrow} tt$. Let $a \in lfw[tt]$. Since $a \overset{B.F.v}{\dashrightarrow} tt$, by (*vlb final*) $\exists n \in lbk[a] : n \overset{A.F.v}{\dashrightarrow} tt$. By (*vlb init*), $\forall n \in lbk[a] : tt \overset{A.I.v}{\dashrightarrow} n$. So $n = tt$ and $lfw[lbk[tt]] = \{\{tt\}\}$. The reasoning for ff and ? are similar.

  Figure 1 shows the commutativity relationships for the example. From



Quantifications shown are implied by the definitions of value-level simulation. For instance, $\forall (e,e) \in lfw_{mm}[(L,L)]\ \exists (f_1,f_2) \in lfw_{mm}[(M,M)] : (\ (e,e) \overset{B.MakeMatch}{\dashrightarrow} (f_1,f_2)\ )$.

Fig. 1. Diagram for incompleteness proof (forward-backward)

  this figure we may read off several facts. First, $\forall c \in \{L, M, K\} \wedge \forall b \in lfw[c] : lfw[c] \neq \emptyset \wedge lbk[b] \neq \emptyset$. Second, $\exists e \in lfw[L] : (\cdot, e) \in B.I.h$ and for all such e, $lbk[e] = \{R\}$.

  Next we observe that since there are $f_1, f_2 \in lfw[M]$ s.t. $(e, e) \overset{B.MakeMatch}{\dashrightarrow} (f_1, f_2)$ and since $lbk_{mm}[(e, e)] = \{(R, R)\}$, then $lbk[f_1], lbk[f_2] \subseteq \{Up, Dn\}$. Now we consider the three cases for $lbk[f_1]$.

  · $lbk[f_1] = \{Dn\}$. We have $(f_1, f_1, ?) \overset{B.Check}{\dashrightarrow} (g_1, g_2, ff)$ and $(H, H, ff) \in lbk_{ck}[(g_1, g_2, ff)]$. We have $lbk_{ck}[(f_1, f_1, ?)] = \{(Up, Up, ?)\}$ but this and (*vlb opn*) would falsely imply that $(Up, Up, ?) \overset{A.Check}{\dashrightarrow} (H, H, ff)$.

  · $lbk[f_1] = \{Up\}$. The reasoning is similar to the preceding case.

· $lbk[f_1] = \{Up, Dn\}$. Suppose $lbk[f_2]$ contains Dn. Then we have $(e, e) \overset{B.MakeMatch}{\dashrightarrow} (f_1, f_2)$ and $(Up, Dn) \in lbk_{mm}[(f_1, f_2)]$. We have $lbk_{mm}[(e, e)] = \{(R, R)\}$ but this and (*vlb opn*) would falsely imply that $(R, R) \overset{A.MakeMatch}{\dashrightarrow} (Up, Dn)$. The case for $lbk[f_2]$ containing Up is similar.

Since all possibilities for lbk have failed, we conclude there is no value-level backward simulation from B to A.

- **Backward-forward**. Let B be any data type such that lbk is a value-level backward simulation from C to B. We show that there is no value-level forward simulation lfw from B to A. For the sake of contradiction, suppose that lfw is such a value-level forward simulation.

  By reasoning similar to that for forward-backward, we have that $lbk[lfw[tt]] = \{\{tt\}\}$, and similarly for $f\!f$ and ?, so again it suffices to consider the visible values of B to be tt, $f\!f$ and ?.

  Figure 2 shows the commutativity relationships for the example. From

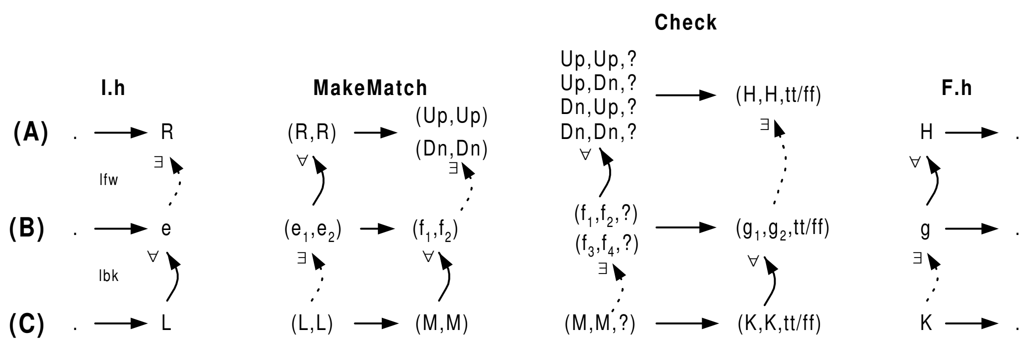


Fig. 2. Diagram for incompleteness proof (backward-forward)

this figure we may read off several facts. First, $\forall c \in \{L, M, K\} \wedge \forall b \in lfw[c] : lbk[c] \neq \emptyset \wedge lfw[b] \neq \emptyset$. Second, $\forall e \in lbk[L] : (\cdot, e) \in B.I.h \wedge R \in lfw[e]$.

Next we observe that since there are $f_1, f_2, f_3, f_4 \in lbk[M]$ (not necessarily unique) as shown, $lfw[f_1]$, $lfw[f_2]$, $lfw[f_3]$, $lfw[f_4]$ must contain Up or Dn (or both). We have the following two facts.

· Either $lfw[f_1]$ or $lfw[f_2]$ does not contain Dn. If both do, then we have $(f_1, f_2, ?) \overset{B.Check}{\dashrightarrow} (g_1, g_2, tt)$ and $(Dn, Dn) \in lfw_{ck}[(f_1, f_2)]$, but for no $(m_1, m_2, tt) \in lfw_{ck}[(g_1, g_2, tt)]$ is it the case that $(Dn, Dn) \overset{A.Check}{\dashrightarrow} (m_1, m_2, tt)$, violating (*vlb opn*).

· Either $lfw[f_3]$ or $lfw[f_4]$ does not contain Up. The reasoning is similar to the preceding case.

Now we show the contradiction. Suppose $lfw[f_1]$ contains Up but not Dn and $lfw[f_3]$ contains Dn but not Up. Since $f_1, f_3 \in lbk[M]$, there is $(e_1, e_2) \in lbk_{mm}[(L, L)]$ s.t. $(e_1, e_2) \overset{B.MakeMatch}{\dashrightarrow} (f_1, f_3)$. We have $(R, R) \in lfw_{mm}[(e_1, e_2)]$ but since $lfw_{mm}[(f_1, f_3)]$ does not contain (Up, Up) or (Dn, Dn), we do not meet the (*vlb opn*) condition for *MakeMatch*. The case for $lfw[f_1]$ containing Dn but not Up and $lfw[f_3]$ containing Up but not Dn is similar.

So we conclude there is no value-level forward simulation from B to A. $\square$

Although value-level forward and backward simulations are not in general complete, there is a restriction to data types for which they are complete. The example of Fig. 0 depends on relations that can modify more than one value. We can obtain a value-level completeness result by restricting data types to being *monadic* so that each operation has only a single argument. The idea is that for a program in a state-level data type derived from a monadic data type, the values computed for any variable are independent of the other variables. The full proof may be found on-line in [1].

Say that *data type C is a refinement of data type A* if for all sets of variables R, there is a refinement from the state-level data type $\mathcal{C}$ derived from C using R to the state-level data type $\mathcal{A}$ derived from A using R.

Theorem 9 (Monadic completeness) *Let C and A be value-level compatible monadic data types. If C is a refinement of A then there is a data type B s.t. $C \subseteq_{\mathrm{LF}} B \subseteq_{\mathrm{LB}} A$.* $\square$

This completeness result can be lifted to apply to data types whose operations have multiple visible arguments but at most a single hidden argument, provided we add other restrictions. (1) Visible values may be changed between program operations. We could view this as an "environment" or "client program" making the changes or as requiring each data type to include operations that each assign a visible value to its single argument. (2) For visible values, initialization and finalization are the identity relation. (3) For hidden values, initialization is not empty and finalization is total. This *weak monadic* definition is complete for value-level reasoning and is consistent with the set-up of RESOLVE [8] and of behavioral subtyping [3].

3 Related Work and Conclusions

For ease of exposition we have used a partial correctness definition of data refinement for first-order input-output programs. Since the proof of our main counterexample, Fig. 0, depends neither on termination nor on input-output, our results hold for total correctness definitions and for reactive programs as well. And even though we have relied heavily on [7], the results can be adapted to other state-level simulations such as the refinement calculus [0].

Now let us consider the value-level simulations mentioned in the literature. Nipkow presents a value-level forward simulation [6], as do Leavens and Pigozzi [2]. Liskov and Wing's behavioral subtyping [3] gives an abstraction function that is both a value-level forward and backward simulation. Naumann [5] gives forward and backward simulations that are induced from value-level relations for a higher-order language.

RESOLVE [8] defines components that import existing data types and export new data types with parameterized operations. For our purposes this is equivalent to a data type with visible sorts (for the imported types) and hidden sorts (for the exported types). The simulation used is called a *correspondence*, which is a form of value-level backward simulation. For verification purposes, RESOLVE permits an augmentation to data values provided this does not affect the behavior of any operation; this is a restricted version of the value-level forward simulation.

Since the value-level simulations proposed in the literature are either forward or backward, we conclude that known value-level simulations are incomplete for showing data refinement in the case that language constructs permit reading and writing of multiple variables. These value-level simulations are complete for the more restricted constructs of monadic and weak monadic. This relationship between language constructs and completeness is deserving of more study.

Acknowledgements. Insightful comments by the anonymous reviewers as well as by David Naumann and Willem-Paul de Roever helped sharpen the paper. We also thank Bill Ogden for many fruitful discussions on the topic.

References

[0] P. Gardiner and C. Morgan. A single complete rule for data refinement. In C. Morgan and T. Vickers (eds.), *On the Refinement Calculus*. (Springer-Verlag, London, 1992), pp. 111-126.

[1] W. Leal and A. Arora. State-level and value-level simulations in data type refinement. Tech. Rpt. OSU-CISRC-5/00-TR13, Dept. of Computer Information Science, The Ohio State University, 2000. ftp://ftp.cis.ohio-state.edu/pub/tech-report/

[2] G.T. Leavens and D. Pigozzi. The behavior-realization adjunction and generalized homomorphic relations. *Theoretical Computer Science 177*, 1 (1997), 183–216.

[3] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*, 6 (1994), 1811–1841.

[4] N. Lynch and F. Vaandrager. Forward and backward simulations, Part I: untimed systems. *Information and Computation 121*, 2 (1995), 214–233.

[5] D.A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science.* To appear, 2000.

[6] T. Nipkow, Observing nondeterministic data types. In D. Sannella and A. Tarlecki (eds.), *Recent Trends in Data Type Specification*, Lecture Notes in Computer Science 332 (Springer Verlag, 1987), pp. 170–183.

[7] W.P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison* (Cambridge University Press, Cambridge, 1998).

[8] M. Sitaraman, B. Weide, and B. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering 23*, 3 (1997), 157-170.