Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Symbolic Analysis via Semantic Reinterpretation*

Junghee Lim[1]**, Akash Lal**[2]**, Thomas Reps**[1,3]

[1] University of Wisconsin; Madison, WI; USA. e-mail: {junghee,reps}@cs.wisc.edu
[2] Microsoft Research India; Bangalore; India. e-mail: akashl@microsoft.com
[3] GrammaTech, Inc.; Ithaca, NY; USA.

Received: Nov. 1, 2009 / Revised version: date

Abstract. The paper presents a novel technique to create implementations of the basic primitives used in symbolic program analysis: *forward symbolic evaluation*, *weakest liberal precondition*, and *symbolic composition*. We used the technique to create a system in which, for the cost of writing just *one* specification—an interpreter for the programming language of interest—one obtains automatically-generated, mutually-consistent implementations of all *three* symbolic-analysis primitives. This can be carried out even for languages with pointers and address arithmetic. Our implementation has been used to generate symbolic-analysis primitives for the x86 and PowerPC instruction sets.

1 Introduction

The use of symbolic-reasoning primitives for *forward symbolic evaluation*, *weakest liberal precondition* ($\mathcal{WLP}$), and *symbolic composition* has experienced a resurgence in program-analysis tools because of the power that they provide when exploring a program's state space.

Model-checking tools, such as SLAM [1] and BLAST [15], as well as hybrid concrete/symbolic program-exploration tools, such as DART [11], CUTE [34], YOGI [14], SAGE [12], BITSCOPE [5], and DASH [3] use forward symbolic evaluation, $\mathcal{WLP}$, or both. An important subroutine in these tools is to determine the following: given a path π in the program, is π feasible (i.e., executable)?

Given path π, symbolic evaluation is used to construct a path formula ψ for π such that π is feasible if and only if ψ is satisfiable. Moreover, a model of ψ can be used to create an input for the program that causes execution to follow path π.

Some of the aforementioned tools also use $\mathcal{WLP}$ to identify new predicates that split part of a program's state space [1,3]. Proof-carrying code systems [30], use $\mathcal{WLP}$ to create verification conditions.

Bug-finding tools, such as ARCHER [37] and SATURN [36], as well as commercial bug-finding products, such as Coverity's PREVENT [7] and GrammaTech's CODESONAR [13], use symbolic composition. Formulas are used to summarize a portion of the behavior of a procedure. Suppose that procedure P calls Q at call-site c, and that r is the site in P to which control returns after the call at c. When c is encountered during the exploration of P, such tools perform the symbolic composition of the formula that expresses the behavior along the path $[entry_P, \ldots, c]$ explored in P with the formula that captures the behavior of Q to obtain a formula that expresses the behavior along the path $[entry_P, \ldots, r]$.

Motivation. The standard approach to implementing each of the symbolic-analysis primitives for a programming language of interest (which we call the *subject* language[1]) is to create hand-written *translation proce-*

* Portions of this work appeared in the Proc. of the 16th Int. SPIN Workshop [23]. The research was supported by NSF under grants CCF-0540955, CCF-0810053, and CCF-0904371, by ONR under grants N00014-09-1-0510 and N00014-09-1-0776, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA8750-06-C-0249 and FA9550-09-1-0279.

** Supported by a Symantec Research Labs Graduate Fellowship.

*** Supported by a Microsoft Research Fellowship. The work was performed while A. Lal was affiliated with the University of Wisconsin.

† T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

[1] Semantic reinterpretation is a program-generation technique, and thus we follow the terminology of the partial-evaluation literature [19], where the program on which the partial evaluator operates is called the *subject program*. (§9 discusses the connections between our approach and partial evaluation.)

dures—one per symbolic-analysis primitive—that convert subject-language commands into appropriate formulas. Such an approach can be extremely tedious. It is also error prone: a system can contain subtle inconsistency bugs if the different translation procedures adopt different "views" of the semantics.

One manifestation of an inconsistency bug would be that if one performs symbolic evaluation of a path π starting from a state that satisfies $\psi = \mathcal{WLP}(\pi, \varphi)$, the resulting symbolic state does not entail φ. Such bugs undermine the soundness of an analysis tool.

The consistency problem is compounded by the issue of aliasing: most subject languages permit memory states to have complicated aliasing patterns, but usually it is not obvious that aliasing is treated consistently across implementations of symbolic evaluation, $\mathcal{WLP}$, and symbolic composition.

Such bugs are easy to introduce because each translation procedure must encode the subject language's semantics; however, the encodings for symbolic evaluation, $\mathcal{WLP}$, and symbolic composition have different flavors.

Our own interest is in analyzing machine code, such as x86 and PowerPC. Unfortunately, machine-code instruction sets have hundreds of instructions, as well as other complicating factors, such as the use of separate instructions to set flags (based on the condition that is tested) and to branch according to the flag values, the ability to perform address arithmetic and dereference computed addresses, etc. To appreciate the need for tool support for creating symbolic-analysis primitives for real machine-code languages, consult the Intel instruction-set reference manual ([16, §3.2] and [17, §4.1]), and imagine writing three separate encodings of each instruction's semantics to implement symbolic evaluation, $\mathcal{WLP}$, and symbolic composition. Some tools (e.g., [12,5]) need an instruction-set emulator, in which case a fourth encoding of the semantics is also required. Moreover, most instruction sets have evolved over time, so that each instruction-set family has a bewildering number of variants.

Our approach. To address these issues, this paper presents a way to automatically obtain mutually-consistent, correct-by-construction implementations of symbolic primitives, by *generating* them from a specification of the subject language's concrete semantics.

The semantics of the basic symbolic-reasoning primitives are easy to state; for instance, if $\tau(\sigma, \sigma')$ is a 2-state formula that represents the semantics of an instruction, then $\mathcal{WLP}(\tau, \varphi)$ can be expressed as $\forall\sigma'.(\tau(\sigma, \sigma') \Rightarrow \varphi(\sigma'))$. However, this formula uses quantification over states—i.e., *second-order quantification*—whereas SMT solvers, such as Yices [9] and Z3 [8], support only *quantifier-free first-order* logic. Hence, such a formula cannot be used directly.

For a simple language that has only `int`-valued variables, it is easy to recast matters in first-order logic. For instance, the $\mathcal{WLP}$ of postcondition φ with respect to an assignment statement $var = rhs$; can be obtained by substituting *rhs* for all (free) occurrences of *var* in φ: $\varphi[var \leftarrow rhs]$. For real-world programming languages, however, the situation is more complicated. For instance, for languages with pointers, Morris's rule of substitution [27] requires taking into account all possible aliasing combinations. In general, tool builders need to create implementations of symbolic primitives for full languages, and hence must be prepared to accommodate whatever features the language supports.

We present a method to obtain quantifier-free, first-order-logic formulas for (a) symbolic evaluation of a single command, (b) $\mathcal{WLP}$ with respect to a single command, and (c) symbolic composition for a class of formulas that express state transformations. The generated implementations are guaranteed to be mutually consistent, and also to be consistent with an instruction-set emulator (for concrete execution) that is generated from the same specification of the subject language's concrete semantics.

Primitives (a) and (b) immediately extend to compound operations over a given program path for use in forward and backwards symbolic evaluation, respectively; see §6. (The design of client algorithms that use such primitives to perform state-space exploration is an orthogonal issue that is outside the scope of this paper.)

Semantic Reinterpretation. Our approach is based on factoring the concrete semantics of a language into two parts: (i) a *client* specification, and (ii) a semantic *core*. The interface to the core consists of certain base types, function types, and operators (sometimes called a *semantic algebra* [28]), and the client is expressed in terms of this interface. Such an organization permits the core to be *reinterpreted* to produce an alternative semantics for the subject language. The idea of exploiting such a factoring comes from the field of abstract interpretation [6], where semantic reinterpretation has been proposed as a convenient tool for formulating abstract interpretations [29,20,32,26] (see §2).

Achievements and Contributions. We used the approach described in the paper to create a "Yacc-like" tool for generating mutually-consistent, correct-by-construction implementations of symbolic-analysis primitives for instruction sets (§8). The input is a specification of an instruction set's concrete semantics; the output is a triple of C++ functions that implement the three symbolic-analysis primitives. The tool has been used to generate such primitives for x86 and PowerPC. To accomplish this, we leveraged an existing tool, TSL [25], as the implementation platform for defining the necessary reinterpretations. However, we wish to stress that the

In logic and linguistics, the programming language would be called the "object language". We avoid that terminology because of possible confusion in §5, which discusses the application of semantic reinterpretation to machine-language programs. In the compiler literature, an object program is a machine-code program produced by a compiler.

ideas presented in the paper are not TSL-specific; other ways of implementing the necessary reinterpretations are possible (see §2).

The contributions of this paper lie in the insights that went into defining the specific reinterpretations that we use to obtain mutually-consistent, correct-by-construction implementations of the symbolic-analysis primitives, and the discovery that $\mathcal{WLP}$ could be obtained by using two different reinterpretations working in tandem. The paper's other contributions are summarized as follows:

- We present a new application for semantic reinterpretation, namely, to create implementations of the basic primitives for symbolic reasoning (§4 and §5). In particular, two key insights allowed us to obtain the primitives for $\mathcal{WLP}$ and symbolic composition:
  - The first insight was that we could apply semantic reinterpretation in a new context, namely, to the interpretation function of a *logic* (§4).
  - The second insight was to define a particular form of state-transformation formula—called a structure-update expression (see §3.1)—to be a first-class notion in the logic, which allows such formulas (i) to serve as a replacement domain in various reinterpretations, and (ii) to be reinterpreted themselves (§4).
- We show how reinterpretation can automatically create a $\mathcal{WLP}$ primitive that implements Morris's rule of substitution for a language with pointers [27] (§4).
- We conducted an experiment that used the generated symbolic-evaluation primitive on real x86 code. The experiment showed that using an exact symbolic-evaluation primitive, as opposed to one that approximates the real semantics, is slower by a factor of 1.07 but is dramatically more accurate (§8).

Moreover, we demonstrate that this approach to creating symbolic-analysis primitives can handle languages with pointers and address arithmetic (§4 and §5). For expository purposes, simplified languages are used throughout. Our discussion of machine code (§3.3 and §5) is based on a greatly simplified fragment of the x86 instruction set; however, our implementation (§8) works on code from real x86 programs compiled from C++ source code, including C++ STL, using Visual Studio.

Organization. §2 presents the basic principles of semantic reinterpretation by means of an example in which reinterpretation is used to create abstract transformers for abstract interpretation. §3 defines the logic that we use, as well a simple source-code language (PL) and an idealized machine-code language (MC). §4 discusses how to use reinterpretation to obtain the three symbolic-analysis primitives for PL. §5 addresses reinterpretation for MC. §6 explains how other language constructs beyond those found in PL and MC can be handled. §7 describes how non-determinism can be incorporated into our approach. §8 describes our implementation and the experiment carried out with it. §9 discusses related work. §10 presents some conclusions. App. A presents correctness proofs.

2 Semantic Reinterpretation for Abstract Interpretation

This section presents the basic principles of semantic reinterpretation in the context of abstract interpretation. We use a simple language of assignments, and define the concrete semantics and an abstract sign-analysis semantics via semantic reinterpretation.

Example 1. [Adapted from [26].] Consider the following fragment of a denotational semantics, which defines the meaning of assignment statements over variables that hold signed 32-bit `int` values (where $\oplus$ denotes exclusive-or):

$$I \in \mathit{Id} \qquad E \in \mathit{Expr} ::= I \mid E_1 \oplus E_2 \mid \ldots$$
$$S \in \mathit{Stmt} ::= I = E; \qquad \sigma \in \mathit{State} = \mathit{Id} \to \mathit{Int32}$$

$$\mathcal{E} : \mathit{Expr} \to \mathit{State} \to \mathit{Int32}$$
$$\mathcal{E}[\![I]\!]\sigma = \sigma I$$
$$\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma \oplus \mathcal{E}[\![E_2]\!]\sigma$$

$$\mathcal{I} : \mathit{Stmt} \to \mathit{State} \to \mathit{State}$$
$$\mathcal{I}[\![I = E;]\!]\sigma = \sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma]$$

We use the notation "$\sigma[I \mapsto v]$," to mean the *State* that acts like σ except that argument I is mapped to v. The function $\mathcal{I}$ can be understood as an *interpreter* for the language: $(\mathcal{I}[\![s]\!]\sigma)$ is the state that results from executing statement s on the state σ. A sequence of statements can be executed by repeatedly calling $\mathcal{I}$. For instance, consider the program shown in Fig. 1(a), which swaps two `int`s. Execution of this code, starting from the state $\sigma_0 = \{x \mapsto -1, y \mapsto 2\}$ can be achieved as follows:

$$\begin{aligned}
&\sigma_0 := \{x \mapsto -1, y \mapsto 2\}\\
&\sigma_1 := \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 = \{x \mapsto -3, y \mapsto 2\}\\
&\sigma_2 := \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 = \{x \mapsto -3, y \mapsto -1\}\\
&\sigma_3 := \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 = \{x \mapsto 2, y \mapsto -1\}
\end{aligned}$$

The languages derivable from *Expr* and *State* define the subject language. The semantics is defined using a *meta-language*. In this example, the meta-language has one base type (*Int32*). It supports defining map types (*State* = *Id* → *Int32*) and user-defined functions ($\mathcal{E}$ and $\mathcal{I}$). It also supports operations on base-type values (e.g., "$\_ \oplus \_$"), map-access operations (σI), map-update operations ($\sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma]$), and invocation of user-defined functions ($\mathcal{E}[\![E]\!]\sigma$).

To highlight better the role of the meta-language, we introduce names for certain aspects of the meta-language. For instance, the one base type, whose standard interpretation is *Int32*, will be called *Val*. We also introduce names for the following operators:

– "_ *xor* _", whose standard interpretation is "_ ⊕ _".
– *lookup*, for map-access operations.
– *store*, for map-update operations.

The specification given earlier is thus rewritten as follows:

$$
\begin{array}{l}
xor : Val \to Val \to Val \\
lookup : State \to Id \to Val \\
store : State \to Id \to Val \to State
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E} : Expr \to State \to Val \\
\mathcal{E}[\![I]\!]\sigma = lookup\ \sigma\ I \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma\ xor\ \mathcal{E}[\![E_2]\!]\sigma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{I} : Stmt\ \to State\ \to State \\
\mathcal{I}[\![I = E;]\!]\sigma = store\ \sigma\ I\ \mathcal{E}[\![E]\!]\sigma
\end{array}
$$

For the concrete (or "standard") semantics, the meta-language types and operators are defined as follows:

$$
\begin{array}{ll}
v \in Val_{std} = Int32 & lookup_{std} = \lambda\sigma.\lambda I.\sigma I \\
State_{std} = Id \to Val & store_{std} = \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v] \\
 & xor_{std} = \lambda v_1.\lambda v_2.v_1 \oplus v_2
\end{array}
$$

Different abstract interpretations of the same language can be defined by using the same semantic specification, but by giving different interpretations of the base types, function types, and operators of the meta-language. For example, for sign analysis, assuming that *Int32* values are represented in two's complement, the meta-language is reinterpreted as follows:[2]

$$
\begin{array}{rcl}
v \in Val_{abs} & = & \{neg, zero, pos, \top\} \\
State_{abs} & = & Id \to Val_{abs} \\
lookup_{abs} & = & \lambda\sigma.\lambda I.\sigma I \\
store_{abs} & = & \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]
\end{array}
$$

$xor_{abs} = \lambda v_1.\lambda v_2.$

		v_2			
		neg	*zero*	*pos*	$\top$
v_1	*neg*	$\top$	*neg*	*neg*	$\top$
	zero	*neg*	*zero*	*pos*	$\top$
	pos	*neg*	*pos*	$\top$	$\top$
	$\top$	$\top$	$\top$	$\top$	$\top$

Essentially, this redefines (or abstracts) the set of values Val_{std} to Val_{abs} and redefines the operators (like *xor*) to operate on the abstract values.

For the code fragment shown in Fig. 1(a), sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$, produce the abstract states shown in Fig. 2. □

Remark. As originally proposed by Mycroft and Jones [29,20], semantic reinterpretation involves refactoring the specification of a language's concrete semantics into a suitable form by introducing appropriate *combinators* that are subsequently redefined to create the different subject-language interpretations. While that style of semantic reinterpretation is also supported by our implementation (see the discussion of the TSL system [25], which is our implementation platform, in §10), we generally work with a fixed set of combinators, namely, the operations supported by the meta-language. The advantage of our approach is that it allows our implementation to act as a "Yacc"-like tool for generating symbolic-analysis primitives from a semantic description of an instruction set. Further discussion of these issues can be found in §8 and §10. □

Semantic Reinterpretation Versus Standard Abstract Interpretation. Semantic reinterpretation [29, 20,32,26] is a form of abstract interpretation [6], but differs from the way abstract interpretation is normally applied: in standard abstract interpretation, one reinterprets the constructs of each *subject language*; in contrast, with semantic reinterpretation one reinterprets the constructs of the *meta-language*. Standard abstract interpretation helps in creating semantically sound *tools*; semantic reinterpretation helps in creating semantically sound *tool generators*. In particular, if you have N subject languages and M analyses, with semantic reinterpretation you obtain $N \times M$ analyzers by writing just $N + M$ specifications: concrete semantics for N subject languages and M reinterpretations. With the standard approach, one must write $N \times M$ abstract semantics.

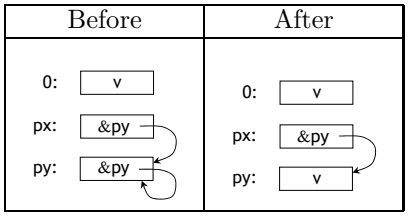
Semantic Reinterpretation Versus Translation to a Common Intermediate Representation. The mapping of subject-language constructs to meta-language operations that one defines as part of the semantic-reinterpretation approach resembles a translation to a common intermediate representation (CIR) *data structure*. Thus, another approach to obtaining "systematic" reinterpretations that are similar to semantic reinterpretations—in that they apply to multiple subject languages—would be to translate subject-language programs to a CIR, and then create various interpreters that implement different abstract interpretations of the node types of the CIR data structure. Each interpreter would then be applied to (the translation of) programs in any subject language L for which one has defined an L-to-CIR translator. Compared with interpreting objects of a CIR data type, the advantages of semantic reinterpretation (i.e., reinterpreting the constructs of the *meta-language*) are

1. The presentation of our ideas is simpler because one does not have to introduce an additional language of trees for representing CIR objects.
2. With semantic reinterpretation, there is no explicit CIR data structure to be interpreted. In essence, semantic reinterpretation removes a level of interpre-

[2] For the two's-complement representation, $pos\ xor_{abs}\ neg = neg\ xor_{abs}\ pos = neg$ because, for all combinations of values represented by *pos* and *neg*, the high-order bit of the result is set, which means that the result is always negative. However, $pos\ xor_{abs}\ pos = neg\ xor_{abs}\ neg = \top$ because the concrete result could be either 0 or positive, and $zero \sqcup pos = \top$.

(a)
s_1: $x = x \oplus y$;
s_2: $y = x \oplus y$;
s_3: $x = x \oplus y$;

(b)
t_1: $*px = *px \oplus *py$;
t_2: $*py = *px \oplus *py$;
t_3: $*px = *px \oplus *py$;



(c)

```
[1] mov  eax, [ebp−10]
[2] xor  eax, [ebp−14]
[3] mov  [ebp−10], eax
[4] mov  eax, [ebp−10]
[5] xor  eax, [ebp−14]
[6] mov  [ebp−14], eax
[7] mov  eax, [ebp−10]
[8] xor  eax, [ebp−14]
[9] mov  [ebp−10], eax
```

(d)

Fig. 1. (a) Code fragment that swaps two `int`s; (b) code fragment that swaps two `int`s using pointers; (c) possible before and after configurations for code fragment (b): the swap is unsuccessful due to aliasing; (d) x86 machine code (in Intel syntax) corresponding to (a).

$$\sigma_0 := \{x \mapsto neg, y \mapsto pos\}$$
$$\sigma_1 := \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 = store_{abs}\ \sigma_0\ x\ (neg\ xor_{abs}\ pos) = \{x \mapsto neg, y \mapsto pos\}$$
$$\sigma_2 := \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 = store_{abs}\ \sigma_1\ y\ (neg\ xor_{abs}\ pos) = \{x \mapsto neg, y \mapsto neg\}$$
$$\sigma_3 := \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 = store_{abs}\ \sigma_2\ x\ (neg\ xor_{abs}\ neg) = \{x \mapsto \top, y \mapsto neg\}.$$

Fig. 2. Application of the abstract transformers created by the sign-analysis reinterpretation to the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$.

tation, and hence generated analyzers should run faster.

To some extent, however, the decision to explain our ideas in terms of semantic reinterpretation is just a matter of presentational style. The goal of the paper is *not* to argue the merits of semantic reinterpretation *per se*; on the contrary, the goal is to present *particular interpretations that yield three desirable symbolic-analysis primitives* for use in program-analysis tools. Semantic reinterpretation is used because it allows us to present our ideas in a concise manner. The ideas introduced in §4 and §5 can be implemented using semantic reinterpretation—as we did (see §8); alternatively, they can be implemented by defining a suitable CIR datatype and creating appropriate interpretations of the CIR's node types—again using ideas similar to those presented in §4 and §5.

3 A Logic and Two Programming Languages

This section defines quantifier-free first-order bit-vector logic, L, a simple source-code language, PL, which only has `int`-valued variables and pointer variables, and a simple machine-code language MC.

3.1 L: A Quantifier-Free Bit-Vector Logic with Finite Functions

The logic L is quantifier-free first-order bit-vector logic over a vocabulary of constant symbols ($I \in Id$) and function symbols ($F \in FuncId$). Strictly speaking, we work with various instantiations of L, denoted by L[PL] and L[MC], in which the vocabularies of function symbols are chosen to describe aspects of the values used by, and computations performed by, the programming languages PL and MC, respectively.

We distinguish the syntactic symbols of L from their counterparts in PL (§2 and §3.2) by using boxes around L's symbols.

$$\begin{aligned} c \in C_{Int32} &= \{0, 1, \ldots\} \\ op2_L \in BinOp_L &= \{\boxed{+}, \boxed{-}, \boxed{\oplus}, \ldots\} \\ rop_L \in RelOp_L &= \{\boxed{=}, \boxed{\neq}, \boxed{<}, \boxed{>}, \ldots\} \\ bop_L \in BoolOp_L &= \{\boxed{\&\&}, \boxed{||}, \ldots\} \end{aligned}$$

The rest of the syntax of $L[\cdot]$ is defined as follows:

$$I \in Id, T \in Term, \varphi \in Formula,$$
$$F \in FuncId, FE \in FuncExpr, U \in StructUpdate$$
$$T ::= c \mid I \mid T_1\ op2_L\ T_2 \mid ite(\varphi, T_1, T_2) \mid FE(T)$$
$$\varphi ::= \boxed{\mathbb{T}} \mid \boxed{\mathbb{F}} \mid T_1\ rop_L\ T_2 \mid \boxed{\neg}\varphi_1 \mid \varphi_1\ bop_L\ \varphi_2$$
$$FE ::= F \mid FE_1[T_1 \mapsto T_2]$$
$$U ::= (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$$

A *Term* of the form $ite(\varphi, T_1, T_2)$ represents an if-then-else expression. Names of the form $F \in FuncId$, possibly with subscripts and/or primes, are function symbols. A *FuncExpr* of the form $FE_1[T_1 \mapsto T_2]$ denotes a *function-update expression*. A *StructUpdate* of the form $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$ is called a *structure-update expression*. It specifies a structure-transformation operation that yields a structure in which the identifier I_i is updated to the value of term T_i, and the function identifier F_j is updated to the value of function-expression FE_j. The subscripts i and j implicitly range over certain index sets, which will be omitted to reduce clutter.

To emphasize that I_i and F_j refer to next-state quantities, we sometimes write structure-update expressions with primes: $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow FE_j\})$. $\{I'_i \hookleftarrow T_i\}$ specifies the updates to the interpretations of the constant symbols and $\{F'_j \hookleftarrow FE_j\}$ specifies the updates to the interpretations of the function symbols (see below). Thus, a structure-update expression $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow FE_j\})$ can be thought of as a kind of restricted 2-vocabulary (i.e., 2-state) formula $\bigwedge_i (I'_i = T_i) \wedge \bigwedge_j (F'_j = FE_j)$. We define U_{id} to be

$$(\{I' \hookleftarrow I \mid I \in Id\}, \{F' \hookleftarrow F \mid F \in FuncId\}).$$

Semantics of L. The semantics of $L[\cdot]$ is defined in terms of a *logical structure*, which gives meaning to the *Id* and *FuncId* symbols of the logic's vocabulary:

$$\iota \in LogicalStruct = (Id \to Val) \times (FuncId \to (Val \to Val)).$$

$(\iota\uparrow 1)$ assigns meanings to constant symbols, and $(\iota\uparrow 2)$ assigns meanings to function symbols. ("$(p\uparrow 1)$" and "$(p\uparrow 2)$" denote the 1st and 2nd components, respectively, of a pair p.)

The factored semantics of L is presented in Fig. 3. Motivated by the needs of later sections, we retain the convention from §2 of working with the domain *Val* rather than *Int32*. Similarly, we also use *BVal* rather than *Bool*. The standard interpretations of $binop_L$, $relop_L$, and $boolop_L$ are as one would expect, e.g., $v_1\, binop_L(\boxed{\oplus})\, v_2 = v_1\, xor\, v_2$, etc. The standard interpretations for $lookupId_{std}$ and $lookupFuncId_{std}$ select from the first and second components, respectively, of a *LogicalStruct*: $lookupId_{std}\, \iota\, I = (\iota\uparrow 1)(I)$ and $lookupFuncId_{std}\, \iota\, F = (\iota\uparrow 2)(F)$. The standard interpretations for *access* and *update* select from, and store to, a map, respectively.

Let $U = (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$. Because $\mathcal{U}[\![U]\!]\iota$ retains from ι the value of each constant I and function F for which an update is not defined explicitly in U (i.e., $I \in (Id - \{I_i\})$ and $F \in (FuncId - \{F_j\})$), as a notational convenience we sometimes treat U as if it contains an identity update for each such symbol; that is, we say that $(U\uparrow 1)I = I$ for $I \in (Id - \{I_i\})$, and $(U\uparrow 2)F = F$ for $F \in (FuncId - \{F_j\})$.

3.2 PL : A Simple Source-Level Language

PL is the language from §2, extended with some additional kinds of `int`-valued expressions, an address-generation expression, a dereferencing expression, and an indirect-assignment statement. Note that arithmetic operations can also occur inside a dereference expression; i.e., PL allows arithmetic to be performed on addresses (including bitwise operations on addresses: see Ex. 2).

$$c \in C_{Int32}, I \in Id, E \in Expr, BE \in BoolExpr, S \in Stmt$$

$$\begin{aligned} c &::= 0 \mid 1 \mid ... \\ E &::= c \mid I \mid \&I \mid *E \mid E_1\ op2\ E_2 \mid BE\ ?\ E_1 : E_2 \\ BE &::= \mathbb{T} \mid \mathbb{F} \mid E_1\ rop\ E_2 \mid \neg BE_1 \mid BE_1\ bop\ BE_2 \\ S &::= I = E; \mid *I = E; \mid S_1\ S_2 \end{aligned}$$

Semantics of PL. The factored semantics of PL is presented in Fig. 4. The semantic domain *Loc* stands for *locations* (or memory addresses). We identify *Loc* with the set *Val* of values. A state $\sigma \in State$ is a pair (η, ρ), where, in the standard semantics, *environment* $\eta \in Env = Id \to Loc$ maps identifiers to their associated locations and *store* $\rho \in Store = Loc \to Val$ maps each location to the value that it holds.

The standard interpretations of the operators used in the PL semantics are

$$\begin{aligned} BVal_{std} &= BVal \\ Val_{std} &= Int32 \\ Loc_{std} &= Int32 \\ \eta \in Env_{std} &= Id \to Loc_{std} \\ \rho \in Store_{std} &= Loc_{std} \to Val_{std} \end{aligned}$$

$$\begin{aligned} cond_{std} &= \lambda b.\lambda v_1.\lambda v_2.\,(b\,?\,v_1 : v_2) \\ lookupState_{std} &= \lambda(\eta,\rho).\lambda I.\rho(\eta(I)) \\ lookupEnv_{std} &= \lambda(\eta,\rho).\lambda I.\eta(I) \\ lookupStore_{std} &= \lambda(\eta,\rho).\lambda l.\rho(l) \\ updateStore_{std} &= \lambda(\eta,\rho).\lambda l.\lambda v.(\eta, \rho[l \mapsto v]) \end{aligned}$$

Handling Computations that "Go Wrong". In accounts of axiomatic semantics [31] and relational semantics [35], one generally considers four outcomes of an execution: an execution *terminates* (in some final state), *goes wrong*, *blocks*, or *diverges*. Because we are only providing the semantics of individual statements/instructions, to simplify matters, we consider only semantic specifications that are terminating. This eliminates outcomes that block or diverge.

We sidestep the need for an explicit outcome for "goes wrong" by introducing an additional *BVal* variable in the state, `isRunning`, which is set to false to model computations that "go wrong". In the extended semantics, a state $\sigma \in State$ is a triple $(\eta, \rho, \texttt{isRunning})$. Fig. 5 shows a sketch of how to add the semantics of the outcome for "divide-by-zero". For the moment, we consider only deterministic specifications. §7 discusses how we handle non-determinism.

3.3 MC: A Simple Machine-Code Language

MC is based on the x86 instruction set, but greatly simplified to have just four registers, one flag, and four in-

$$
\begin{aligned}
const &: C_{Int32} \to Val \\
cond_L &: BVal \to Val \to Val \to Val \\
lookupId &: LogicalStruct \to Id \to Val \\
binop_L &: BinOp_L \to (Val \times Val \to Val) \\
relop_L &: RelOp_L \to (Val \times Val \to BVal) \\
boolop_L &: BoolOp_L \to (BVal \times BVal \to BVal) \\
lookupFuncId &: LogicalStruct \to FuncId \to (Val \to Val) \\
access &: (Val \to Val) \times Val) \to Val \\
update &: ((Val \to Val) \times Val \times Val) \to (Val \to Val)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T} &: Term \to LogicalStruct \to Val \\
\mathcal{T}[\![c]\!]\iota &= const(c) \\
\mathcal{T}[\![I]\!]\iota &= lookupId\ \iota\ I \\
\mathcal{T}[\![T_1\ op2_L\ T_2]\!]\iota &= \mathcal{T}[\![T_1]\!]\iota\ binop_L(op2_L)\ \mathcal{T}[\![T_2]\!]\iota \\
\mathcal{T}[\![ite(\varphi, T_1, T_2)]\!]\iota &= cond_L(\mathcal{F}[\![\varphi]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota) \\
\mathcal{T}[\![FE(T_1)]\!]\iota &= access(\mathcal{FE}[\![FE]\!]\iota, \mathcal{T}[\![T_1]\!]\iota)
\end{aligned}
\qquad
\begin{aligned}
\mathcal{F} &: Formula \to LogicalStruct \to BVal \\
\mathcal{F}[\![\boxed{\mathbb{T}}]\!]\iota &= \mathbb{T} \\
\mathcal{F}[\![\boxed{\mathbb{F}}]\!]\iota &= \mathbb{F} \\
\mathcal{F}[\![T_1\ rop_L\ T_2]\!]\iota &= \mathcal{T}[\![T_1]\!]\iota\ relop_L(rop_L)\ \mathcal{T}[\![T_2]\!]\iota \\
\mathcal{F}[\![\boxed{\neg}\ \varphi_1]\!]\iota &= \neg\mathcal{F}[\![\varphi_1]\!]\iota \\
\mathcal{F}[\![\varphi_1\ bop_L\ \varphi_2]\!]\iota &= \mathcal{F}[\![\varphi_1]\!]\iota\ boolop_L(bop_L)\ \mathcal{F}[\![\varphi_2]\!]\iota
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{FE} &: FuncExpr \to LogicalStruct \to (Val \to Val) \\
\mathcal{FE}[\![F]\!]\iota &= lookupFuncId\ \iota\ F \\
\mathcal{FE}[\![FE_1[T_1 \mapsto T_2]]\!]\iota &= update(\mathcal{FE}[\![FE_1]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{U} : StructUpdate \to LogicalStruct \to LogicalStruct \\
&\mathcal{U}[\![(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})]\!]\iota = ((\iota\uparrow 1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], (\iota\uparrow 2)[F_j \mapsto \mathcal{FE}[\![FE_j]\!]\iota])
\end{aligned}
$$

Fig. 3. The factored semantics of L.

$$
\begin{aligned}
v &\in Val \\
l &\in Loc = Val \\
\eta &\in Env = Id \to Loc \\
\rho &\in Store = Loc \to Val \\
\sigma &\in State = Store \times Env
\end{aligned}
\qquad
\begin{aligned}
\mathcal{E} &: Expr \to State \to Val \\
\mathcal{E}[\![c]\!]\sigma &= const(c) \\
\mathcal{E}[\![I]\!]\sigma &= lookupState\ \sigma\ I \\
\mathcal{E}[\![\&I]\!]\sigma &= lookupEnv\ \sigma\ I \\
\mathcal{E}[\![*E]\!]\sigma &= lookupStore\ \sigma\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathcal{E}[\![E_1\ op2\ E_2]\!]\sigma &= \mathcal{E}[\![E_1]\!]\sigma\ binop(op2)\ \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{E}[\![BE\ ?\ E_1 : E_2]\!]\sigma &= cond(\mathcal{B}[\![BE]\!]\sigma, \mathcal{E}[\![E_1]\!]\sigma, \mathcal{E}[\![E_2]\!]\sigma)
\end{aligned}
$$

$$
\begin{aligned}
const &: C_{Int32} \to Val \\
cond &: BVal \to Val \to Val \to Val \\
lookupState &: State \to Id \to Val \\
lookupEnv &: State \to Id \to Loc \\
lookupStore &: State \to Loc \to Val \\
updateStore &: State \to Loc \to Val \to State
\end{aligned}
\qquad
\begin{aligned}
\mathcal{B} &: BoolExpr \to State \to BVal \\
\mathcal{B}[\![\mathbb{T}]\!]\sigma &= \mathbb{T} \\
\mathcal{B}[\![\mathbb{F}]\!]\sigma &= \mathbb{F} \\
\mathcal{B}[\![E_1\ rop\ E_2]\!]\sigma &= \mathcal{E}[\![E_1]\!]\sigma\ relop(rop)\ \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{B}[\![\neg BE_1]\!]\sigma &= \neg\mathcal{B}[\![BE_1]\!]\sigma \\
\mathcal{B}[\![BE_1\ bop\ BE_2]\!]\sigma &= \mathcal{B}[\![BE_1]\!]\sigma\ boolop(bop)\ \mathcal{B}[\![BE_2]\!]\sigma
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I} &: Stmt \to State \to State \\
\mathcal{I}[\![I = E;]\!]\sigma &= updateStore\ \sigma\ (lookupEnv\ \sigma\ I)\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathcal{I}[\![*I = E;]\!]\sigma &= updateStore\ \sigma\ (\mathcal{E}[\![I]\!]\sigma)\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathcal{I}[\![S_1\ S_2]\!]\sigma &= \mathcal{I}[\![S_2]\!](\mathcal{I}[\![S_1]\!]\sigma)
\end{aligned}
$$

Fig. 4. The factored semantics of PL.

structions.

$$
\begin{aligned}
r \in register, do &\in dst\_operand, \\
so \in src\_operand, i &\in instruction \\
r &::= EAX \mid EBX \mid EBP \mid EIP \\
flagName &::= ZF \\
do &::= Indirect(r, Val) \mid DirectReg(r) \\
so &::= do \cup Immediate(Val) \\
instruction &::= MOV(do, so) \mid CMP(do, so)
\end{aligned}
$$

Semantics of MC. The factored semantics of MC is presented in Fig. 6. It is similar to the semantics of PL, although MC exhibits two features not part of PL: there is an explicit program counter (*EIP*), and MC includes the typical feature of machine-code languages that a branch is split across two instructions (*CMP ... JZ*).

$$
\begin{array}{rl}
\mathit{Loc} = \mathit{Val} & \mathcal{E} : \mathit{Expr} \to \mathit{State} \to (\mathit{Val}, \mathit{BVal}) \\
\mathit{Env} = \mathit{Id} \to \mathit{Loc} & \mathcal{E}[\![c]\!]\sigma = (\mathit{const}(c),\ \mathbb{T}) \\
\mathit{Store} = \mathit{Loc} \to \mathit{Val} & \mathcal{E}[\![I]\!]\sigma = (\mathit{lookupState}\ \sigma\ I,\ \mathbb{T}) \\
\mathit{State} = \mathit{Store} \times \mathit{Env} \times \mathit{BVal} & \mathcal{E}[\![E_1/E_2]\!]\sigma = (\mathcal{E}[\![E_2]\!]\sigma = 0) \\
 & \quad ?\ (1,\ \mathbb{F}) \\
\mathit{const} : C_{\mathit{Int32}} \to \mathit{Val} & \quad :\ (\mathcal{E}[\![E_1]\!]\sigma / \mathcal{E}[\![E_2]\!]\sigma,\ \mathbb{T}) \\
\mathit{lookupState} : \mathit{State} \to \mathit{Id} \to \mathit{Val} & \\
\mathit{getIsRunning} : (\mathit{Val}, \mathit{BVal}) \to \mathit{BVal} & \mathcal{I} : \mathit{Stmt} \to \mathit{State} \to \mathit{State} \\
\mathit{lookupIsRunning} : \mathit{State} \to \mathit{BVal} & \mathcal{I}[\![I = E;]\!]\sigma = (\mathit{lookupIsRunning}\ \sigma) = \mathbb{T} \\
\mathit{updateIsRunning} : \mathit{State} \to \mathit{BVal} \to \mathit{State} & \quad ?\ (\mathit{getIsRunning}\ \mathcal{E}[\![E]\!]\sigma) = \mathbb{T} \\
\mathit{getIsRunning} = \lambda(v, b).b & \quad\quad ?\ \mathit{updateStore}\ \sigma\ (\mathit{lookupEnv}\ \sigma\ I)\ (\mathcal{E}[\![E]\!]\sigma) \\
\mathit{lookupIsRunning} = \lambda(\eta, \rho, b).b & \quad\quad :\ \mathit{updateIsRunning}\ \sigma\ \mathbb{F} \\
\mathit{updateIsRunning} = \lambda(\eta, \rho, b).\lambda b'.(\eta, \rho, b') & \quad :\ \sigma
\end{array}
$$

Fig. 5. An extended semantics of PL to accommodate the outcome of "divide-by-zero" execution.

An MC state $\sigma \in \mathit{State}$ is a triple $(\mathit{mem}, \mathit{reg}, \mathit{flag})$, where *mem* is a map $\mathit{Val} \to \mathit{Val}$, *reg* is a map $\mathit{register} \to \mathit{Val}$, and *flag* is a map $\mathit{flagName} \to \mathit{BVal}$. We assume that each instruction is 4 bytes long; hence, the execution of a *MOV*, *CMP* or *XOR* increments the program-counter register *EIP* by 4. *CMP* sets the value of *ZF* according to the difference of the values of the two operands; *JZ* updates *EIP* depending on the value of flag *ZF*.

4 Symbolic Analysis for PL via Reinterpretation

A PL state (η, ρ) can be modeled in L[PL] by using a function symbol F_ρ for store ρ, and a constant symbol $c_x \in \mathit{Id}$ for each PL identifier x. (To reduce clutter, we will use `x` for such constants instead of c_x.) Given $\iota \in \mathit{LogicalStruct}$, the constant symbols and their interpretations in ι correspond to environment η, and the interpretation of F_ρ in ι corresponds to store ρ.

Symbolic Evaluation. A primitive for forward symbolic-evaluation must solve the following problem: *Given the semantic definition of a programming language, together with a specific statement s, create a logical formula that captures the semantics of s.* The following table illustrates how the semantics of PL statements can be expressed as L[PL] structure-update expressions:

PL	L[PL]
$x = 17;$	$(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto 17]\})$
$x = y;$	$(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})]\})$
$x = *q;$	$(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(F_\rho(\mathtt{q}))]\})$

To create such expressions automatically using semantic reinterpretation, we use formulas of the logic L[PL] as a reinterpretation domain for the meta-language primitives used to define PL. The base types and the state type of the meta-language are reinterpreted as follows (our convention is to mark each reinterpreted base type, function type, and operator with an overbar): $\overline{\mathit{Val}} = \mathit{Term}$, $\overline{\mathit{BVal}} = \mathit{Formula}$, and $\overline{\mathit{State}} = \mathit{StructUpdate}$. The operators used in PL's meaning functions $\mathcal{E}$, $\mathcal{B}$, and $\mathcal{I}$ are reinterpreted over these domains as follows:

- The arithmetic, bitwise, relational, and logical operators are interpreted as syntactic constructors of L[PL] *Term*s and *Formula*s, e.g.,
$$\overline{\mathit{binop}}(\oplus) = \lambda T_1.\lambda T_2.T_1 \boxed{\oplus} T_2.$$
Straightforward simplifications are also performed; e.g., $0 \boxed{\oplus} a$ simplifies to a, etc. Other simplifications that we perform are similar to ones used by others, such as the preprocessing steps used in decision procedures (e.g., the ite-lifting and read-over-write transformations for operations on functions [10]).
- $\overline{\mathit{cond}}$ residuates an $\mathit{ite}(\cdot,\cdot,\cdot)$ *Term* when the result cannot be simplified to a single branch.

The other operations used in the PL semantics are reinterpreted as follows:

$$
\begin{array}{rl}
\overline{\mathit{lookupState}} & : \mathit{StructUpdate} \to \mathit{Id} \to \mathit{Term} \\
\overline{\mathit{lookupState}} & = \lambda U.\lambda I.((U{\uparrow}2)F_\rho)((U{\uparrow}1)I) \\
\overline{\mathit{lookupEnv}} & : \mathit{StructUpdate} \to \mathit{Id} \to \mathit{Term} \\
\overline{\mathit{lookupEnv}} & = \lambda U.\lambda I.(U{\uparrow}1)I \\
\overline{\mathit{lookupStore}} & : \mathit{StructUpdate} \to \mathit{Term} \to \mathit{Term} \\
\overline{\mathit{lookupStore}} & = \lambda U.\lambda T.((U{\uparrow}2)F_\rho)(T) \\
\overline{\mathit{updateStore}} & : \mathit{StructUpdate} \to \mathit{Term} \to \mathit{Term} \\
 & \quad \to \mathit{StructUpdate} \\
\overline{\mathit{updateStore}} & = \lambda U.\lambda T_1.\lambda T_2. \\
 & \quad ((U{\uparrow}1), (U{\uparrow}2)[F_\rho \mapsto ((U{\uparrow}2)F_\rho)[T_1 \mapsto T_2]])
\end{array}
$$

By extension, this produces functions $\overline{\mathcal{E}}$, $\overline{\mathcal{B}}$, and $\overline{\mathcal{I}}$ with the types shown in Fig. 7.

In particular, given a *StructUpdate* U, function $\overline{\mathcal{I}}$ translates a statement s of PL to the *StructUpdate* $\overline{\mathcal{I}}[\![s]\!]U$ in logic L[PL]. To perform symbolic evaluation along a path π, one starts with the *StructUpdate* $U_{id} = (\emptyset, \{F'_\rho \hookleftarrow F_\rho\})$ and repeatedly calls function $\overline{\mathcal{I}}$ with the next statement in π and the current *StructUpdate*.

Example 2. The steps of symbolic evaluation of Fig. 1(a) via semantic reinterpretation, starting with U_{id}, are

$$
\begin{array}{rl}
const : C_{Int32} \to Val & cond : BVal \to Val \to Val \to Val \\
lookup_{reg} : State \to register \to Val & lookup_{mem} : State \to Val \to Val \\
store_{reg} : State \to register \to Val \to State & store_{mem} : State \to Val \to Val \to State \\
lookup_{flag} : State \to flagName \to BVal & incr_{eip} : State \to State \\
store_{flag} : State \to flagName \to BVal \to State & incr_{eip} = \lambda\sigma.store_{reg}(\sigma, EIP, \mathcal{R}[\![EIP]\!]\sigma\ binop(+)\ 4)
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{R} : reg \to State \to Val & \mathcal{O} : src\_operand \to State \to Val \\
\mathcal{R}[\![r]\!]\sigma = lookup_{reg}(\sigma, r) & \mathcal{O}[\![Indirect(r,c)]\!]\sigma = lookup_{mem}(\sigma, \mathcal{R}[\![r]\!]\sigma\ binop(+)\ const(c)) \\
 & \mathcal{O}[\![DirectReg(r)]\!]\sigma = \mathcal{R}[\![r]\!]\sigma \\
\mathcal{K} : flagName \to State \to BVal & \mathcal{O}[\![Immediate(c)]\!]\sigma = const(c) \\
\mathcal{K}[\![ZF]\!]\sigma = lookup_{flag}(\sigma, ZF) &
\end{array}
$$

$$
\begin{array}{rl}
\multicolumn{2}{c}{\mathcal{I} : instruction \to State \to State} \\
\mathcal{I}[\![MOV(Indirect(r,c), so)]\!]\sigma &= incr_{eip}(store_{mem}(\sigma, \mathcal{R}[\![r]\!]\sigma\ binop(+)\ const(c), \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![MOV(DirectReg(r), so)]\!]\sigma &= incr_{eip}(store_{reg}(\sigma, r, \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![CMP(do, so)]\!]\sigma &= incr_{eip}(store_{flag}(\sigma, ZF, \mathcal{O}[\![do]\!]\sigma\ binop(-)\ \mathcal{O}[\![so]\!]\sigma\ relop(=)\ 0)) \\
\mathcal{I}[\![XOR(do{:}Indirect(r,c), so)]\!]\sigma &= incr_{eip}(store_{mem}(\sigma, \mathcal{R}[\![r]\!]\sigma\ binop(+)\ const(c), \mathcal{O}[\![do]\!]\sigma\ binop(\oplus)\ \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![XOR(do{:}DirectReg(r), so)]\!]\sigma &= incr_{eip}(store_{reg}(\sigma, r, \mathcal{O}[\![do]\!]\sigma\ binop(\oplus)\ \mathcal{O}[\![so]\!]\sigma)) \\
\mathcal{I}[\![JZ(do)]\!]\sigma &= store_{reg}(\sigma, EIP, cond(\mathcal{K}[\![ZF]\!]\sigma, \mathcal{R}[\![EIP]\!]\sigma\ binop(+)\ 4, \mathcal{O}[\![do]\!]\sigma))
\end{array}
$$

Fig. 6. The factored semantics of MC.

Standard	Reinterpreted
$\mathcal{E}$: $Expr \to State \to Val$	$\overline{\mathcal{E}}$: $Expr \to StructUpdate \to Term$
$\mathcal{B}$: $BoolExpr \to State \to BVal$	$\overline{\mathcal{B}}$: $BoolExpr \to StructUpdate \to Formula$
$\mathcal{I}$: $Stmt \to State \to State$	$\overline{\mathcal{I}}$: $Stmt \to StructUpdate \to StructUpdate$

Fig. 7. Standard types of the PL meaning functions, and the reinterpreted types used to obtain an implementation of symbolic evaluation.

shown in Fig. 8. The resulting *StructUpdate*, U_{swap}, can be considered to be the 2-vocabulary formula

$$F'_\rho = F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})],$$

which expresses a state change in which the values of program variables x and y are swapped.

Algebraic simplification plays an important role. For example, when y is updated in U_1 by

$$[\mathtt{y} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{y}))]$$

(see Fig. 8), the update is simplified to $[\mathtt{y} \mapsto F_\rho(\mathtt{x})]$. □

Example 3. To illustrate symbolic evaluation for an example that involves pointers and pointer-dereferencing operations, Fig. 9 shows the steps of symbolic evaluation of Fig. 1(b) via semantic reinterpretation, starting with a *StructUpdate* that corresponds to the "Before" column of Fig. 1(c). The program from Fig. 1(b) works correctly when there is no aliasing; however, it does not always work correctly when started from the kind of state shown in the "Before" column of Fig. 1(c). The *StructUpdate* U_4 obtained via our symbolic-evaluation primitive can be considered to be the 2-vocabulary formula

$$F'_\rho = F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto v],$$

which expresses a state change that does not usually perform a successful swap. The example shows that the symbolic-evaluation method can faithfully track non-trivial situations that involve pointer aliasing. □

The correctness of our method for performing symbolic evaluation is captured by the following theorem:

Theorem 1. *For all $s \in Stmt$, $U \in StructUpdate$, and $\iota \in LogicalStruct$, the meaning of $\overline{\mathcal{I}}[\![s]\!]U$ in ι (i.e., $\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota$) is equivalent to running $\mathcal{I}$ on s with an input state obtained from $\mathcal{U}[\![U]\!]\iota$. That is,*

$$\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota = \mathcal{I}[\![s]\!](\mathcal{U}[\![U]\!]\iota).$$

Proof. See App. A.1.

WLP. $\mathcal{WLP}(s, \varphi)$ characterizes the set of states σ such that the execution of s starting in σ either fails to terminate or results in a state σ' such that $\varphi(\sigma')$ holds. For a language that only has `int`-valued variables, the $\mathcal{WLP}$ of a postcondition (specified by formula φ) with respect to an assignment statement $var = rhs$; can be expressed as the formula obtained by substituting *rhs* for all (free) occurrences of *var* in φ: $\varphi[var \leftarrow rhs]$.

For a language with pointer variables, such as PL, syntactic substitution is not adequate for finding $\mathcal{WLP}$ formulas. For instance, suppose that we are interested in finding a formula for the $\mathcal{WLP}$ of postcondition $x = 5$ with respect to $*p = e$;. It is not correct merely to perform the substitution $(x = 5)[*p \leftarrow e]$. That substitution yields $x = 5$, whereas the $\mathcal{WLP}$ depends on the execution context in which $*p = e$; is evaluated:

$$
\begin{aligned}
\overline{\mathcal{I}}[\![x = x \oplus y;]\!]U_{id} &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (\overline{\mathcal{E}}[\![x]\!]U_{id} \boxed{\oplus} \overline{\mathcal{E}}[\![y]\!]U_{id})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))]\}) = U_1 \\
\overline{\mathcal{I}}[\![y = x \oplus y;]\!]U_1 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))][\mathtt{y} \mapsto (\overline{\mathcal{E}}[\![x]\!]U_1 \boxed{\oplus} \overline{\mathcal{E}}[\![y]\!]U_1)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))][\mathtt{y} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{y}))]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y}))][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) = U_2 \\
\overline{\mathcal{I}}[\![x = x \oplus y;]\!]U_2 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto (\overline{\mathcal{E}}[\![x]\!]U_2 \boxed{\oplus} \overline{\mathcal{E}}[\![y]\!]U_2)][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{x}))][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) = U_{swap}
\end{aligned}
$$

Fig. 8. Symbolic evaluation of Fig. 1(a) via semantic reinterpretation, starting with the *StructUpdate* $U_{id} = (\emptyset, \{F'_\rho \hookleftarrow F_\rho\})$.

$$
\begin{aligned}
U_1 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto \mathtt{py}]\}) \\
\overline{\mathcal{I}}[\![*px = *px \oplus *py;]\!]U_1 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto (\overline{\mathcal{E}}[\![*px]\!]U_1 \boxed{\oplus} \overline{\mathcal{E}}[\![*py]\!]U_1)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto (\mathtt{py} \boxed{\oplus} \mathtt{py})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto 0]\}) = U_2 \\
\overline{\mathcal{I}}[\![*py = *px \oplus *py;]\!]U_2 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto (\overline{\mathcal{E}}[\![*px]\!]U_2 \boxed{\oplus} \overline{\mathcal{E}}[\![*py]\!]U_2)][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto 0]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto (0 \boxed{\oplus} v)][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto 0]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto 0]\}) = U_3 \\
\overline{\mathcal{I}}[\![*px = *px \oplus *py;]\!]U_3 &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto (\overline{\mathcal{E}}[\![*px]\!]U_3 \boxed{\oplus} \overline{\mathcal{E}}[\![*py]\!]U_3)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto (0 \boxed{\oplus} v)]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathtt{px} \mapsto \mathtt{py}][\mathtt{py} \mapsto v]\}) = U_4
\end{aligned}
$$

Fig. 9. Symbolic evaluation of Fig. 1(b) via semantic reinterpretation, starting with a *StructUpdate* that corresponds to the "Before" column of Fig. 1(c).

- If p points to x, then the $\mathcal{WLP}$ formula should be $e = 5$.
- If p does not point to x, then the $\mathcal{WLP}$ formula should be $x = 5$.

The desired formula can be expressed informally as

$$((p = \&x)\ ?\ e : x) = 5.$$

For a program fragment that involves multiple pointer variables, the $\mathcal{WLP}$ formula may have to take into account all possible aliasing combinations. This is the essence of Morris's rule of substitution [27]. One of the most important features of our approach is its ability to create correct implementations of Morris's rule of substitution automatically—and basically for free.

Example 4. In L[PL], such a formula would be expressed as shown in the lower row below.

Informal	$\mathcal{WLP}(*p = e, x = 5)$ $= ((p = \&x)\ ?\ e : x) = 5$
L[PL]	$\mathcal{WLP}(*p = e, F_\rho(\mathtt{x}) \boxed{=} 5)$ $= ite(F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}, F_\rho(\mathtt{e}), F_\rho(\mathtt{x})) \boxed{=} 5$

In Ex. 6, we will show how the latter formula is created via semantic reinterpretation. □

To create primitives for $\mathcal{WLP}$ and symbolic composition via semantic reinterpretation, we again use L[PL] as a reinterpretation domain; however, there is a trick: in contrast with what is done to generate symbolic-evaluation primitives, we use the *StructUpdate* type of L[PL] to reinterpret the meaning functions $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$ of L[PL] itself! By this means, the "alternative meaning" of a *Term*/*Formula*/*FuncExpr*/*StructUpdate* is a (usually different) *Term*/*Formula*/*FuncExpr*/*StructUpdate* in which some substitution and/or simplification has taken place. The general scheme is outlined in the following table:

Meaning Functions	Type Reinterpreted	Replacement Type	Function Created
$\mathcal{I}, \mathcal{E}, \mathcal{B}$	*State*	*StructUpdate*	Symbolic evaluation
$\mathcal{F}, \mathcal{T}$	*LogicalStruct*	*StructUpdate*	$\mathcal{WLP}$
$\mathcal{U}, \mathcal{FE}, \mathcal{F}, \mathcal{T}$	*LogicalStruct*	*StructUpdate*	Symbolic composition

In §3.1, we defined the semantics of $L[\cdot]$ in a form that would make it amenable to semantic reinterpretation. However, one small point needs adjustment: in §3.1, the type signatures of *LogicalStruct*, *lookupFuncId*, *access*, *update*, and $\mathcal{FE}$ include occurrences of $Val \rightarrow Val$. This was done to make the types more intuitive; however, for reinterpretation to work, an additional level of factoring is necessary. In particular, the occurrences of $Val \rightarrow Val$ need to be replaced by *FVal*. The standard semantics of *FVal* is $Val \rightarrow Val$; however, for creating symbolic-analysis primitives, *FVal* is reinterpreted as *FuncExpr*.

The reinterpretation used for $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$ is similar to what was used for symbolic evaluation of PL programs:

- $\overline{Val} = Term$, $\overline{BVal} = Formula$, $\overline{FVal} = FuncExpr$, and $\overline{LogicalStruct} = StructUpdate$.
- The arithmetic, bitwise, relational, and logical operators are interpreted as syntactic *Term* and *Formula* constructors of L, e.g.,

$$\overline{binop_L}(\boxed{\oplus}) = \lambda T_1.\lambda T_2.T_1 \boxed{\oplus} T_2,$$

  although straightforward simplifications are also performed.
- $\overline{cond_L}$ residuates an $ite(\cdot,\cdot,\cdot)$ *Term* when the result cannot be simplified to a single branch.
- $\overline{lookupId}$ and $\overline{lookupFuncId}$ are resolved immediately, rather than residuated:
  - $\overline{lookupId}\ (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})\ I_k = T_k$
  - $\overline{lookupFuncId}\ (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})\ F_k = FE_k$.
- $\overline{access}$ and $\overline{update}$ are discussed below.

By extension, this produces reinterpreted meaning functions $\overline{\mathcal{U}}$, $\overline{\mathcal{FE}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{T}}$.

Somewhat surprisingly, we do not need to introduce an explicit operation of substitution for our logic because *a substitution operation is produced as a by-product of reinterpretation.* In particular, in the standard semantics for L, the return types of meaning function $\mathcal{T}$ and helper function *lookupId* are both *Val*. However, in the reinterpreted semantics, a $\overline{Val}$ is a *Term*—i.e., something *symbolic*—which is used in subsequent computations. Thus, when $\iota \in LogicalStruct$ is reinterpreted as $U \in StructUpdate$, the reinterpretation of formula φ via $\overline{\mathcal{F}}[\![\varphi]\!]U$ substitutes *Term*s found in U into φ: $\overline{\mathcal{F}}[\![\varphi]\!]U$ calls $\overline{\mathcal{T}}[\![T]\!]U$, which may call $\overline{lookupId}\ U\ I$; the latter would return a *Term* fetched from U, which would be a subterm of the answer returned by $\overline{\mathcal{T}}[\![T]\!]U$, which in turn would be a subterm of the answer returned by $\overline{\mathcal{F}}[\![\varphi]\!]U$.

To create a formula for $\mathcal{WLP}$ via semantic reinterpretation, we make use of both $\overline{\mathcal{F}}$, the reinterpreted logic semantics, and $\overline{\mathcal{I}}$, the reinterpreted programming-language semantics. The $\mathcal{WLP}$ formula for φ with respect to statement s is obtained by performing the following computation:

$$\mathcal{WLP}(s,\varphi) = \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id}). \tag{1}$$

Example 5. In Ex. 2 and Fig. 8, we derived the following *StructUpdate*, which expresses in L[PL] the semantics of the swap-code fragment *swap* from Fig. 1(a):

$$\begin{aligned} U_{swap} &= \overline{\mathcal{I}}[\![swap]\!]U_{id} \\ &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}). \end{aligned}$$

Using the method given in Eqn. (1), we obtain the following *Formula* of L[PL] for $\mathcal{WLP}(swap, F_\rho(\mathtt{x}) \boxed{=} 2)$:

$$\begin{aligned} &\mathcal{WLP}(swap, F_\rho(\mathtt{x}) \boxed{=} 2) \\ &\quad= \overline{\mathcal{F}}[\![F_\rho(\mathtt{x}) \boxed{=} 2]\!]U_{swap} \\ &\quad= (\overline{\mathcal{T}}[\![F_\rho(\mathtt{x})]\!]U_{swap}) \boxed{=} (\overline{\mathcal{T}}[\![2]\!]U_{swap}) \\ &\quad= (\overline{access}(\overline{\mathcal{FE}}[\![F_\rho]\!]U_{swap}, \overline{\mathcal{T}}[\![\mathtt{x}]\!]U_{swap})) \boxed{=} (\overline{const}(2)) \\ &\quad= \left(\overline{access}\left(\begin{array}{l}\overline{lookupFuncId}\ U_{swap}\ F_\rho, \\ \overline{lookupId}\ U_{swap}\ \mathtt{x}\end{array}\right)\right) \boxed{=} 2 \\ &\quad= (\overline{access}(F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})], \mathtt{x})) \boxed{=} 2 \\ &\quad= F_\rho(\mathtt{y}) \boxed{=} 2 \end{aligned}$$

(To understand the last step, see the discussion of $\overline{access}$ below.) □

To understand how pointers are handled during the $\mathcal{WLP}$ operation, the key reinterpretations to concentrate on in L[PL] are the ones for the operations of the metalanguage that manipulate *FVal*s (i.e., arguments of type $Val \to Val$)—in particular, *access* and *update*. We want $\overline{access}$ and $\overline{update}$ to enjoy the following semantic properties:

$$\begin{aligned} \mathcal{T}[\![\overline{access}(FE_0, T_0)]\!]\iota &= (\mathcal{FE}[\![FE_0]\!]\iota)(\mathcal{T}[\![T_0]\!]\iota) \\ \mathcal{FE}[\![\overline{update}(FE_0, T_0, T_1)]\!]\iota &= (\mathcal{FE}[\![FE_0]\!]\iota)[\mathcal{T}[\![T_0]\!]\iota \mapsto \mathcal{T}[\![T_1]\!]\iota] \end{aligned}$$

Note that these properties require evaluating the results of $\overline{access}$ and $\overline{update}$ with respect to an arbitrary $\iota \in LogicalStruct$. As mentioned earlier, it is desirable for reinterpreted base-type operations to perform simplifications whenever possible, when they construct *Term*s, *Formula*s, *FuncExpr*s, and *StructUpdate*s. However, because the value of ι is unknown, $\overline{access}$ and $\overline{update}$ operate in an uncertain environment.

To use semantic reinterpretation to create a $\mathcal{WLP}$ primitive that implements Morris's rule, simplifications are performed by $\overline{access}$ and $\overline{update}$ according to the definitions given in Fig. 10. The possible-equality case for $\overline{access}$ Fig. 10 introduces *ite* terms. As illustrated in Ex. 6, it is these *ite* terms that cause the reinterpreted operations to account for possible aliasing combinations, and thus are the reason that the semantic-reinterpretation method automatically carries out the actions of Morris's rule of substitution [27].

Example 6. We now demonstrate how semantic reinterpretation produces the L[PL] formula for $\mathcal{WLP}(*p = e, x = 5)$ claimed in Ex. 4.

$$\begin{aligned} U &:= \overline{\mathcal{I}}[\![*p = e]\!]U_{Id} \\ &= \overline{updateStore}(U_{Id}, \overline{\mathcal{E}}[\![p]\!]U_{Id}, \overline{\mathcal{E}}[\![e]\!]U_{Id}) \\ &= \overline{updateStore}(U_{Id}, \overline{lookupState}(U_{Id}, \mathtt{p}), \overline{lookupState}(U_{Id}, \mathtt{e}) \\ &= \overline{updateStore}(U_{Id}, F_\rho(\mathtt{p}), F_\rho(\mathtt{e})) \\ &= ((U_{Id}\uparrow 1), \{F_\rho \hookleftarrow F_\rho[F_\rho(\mathtt{p}) \mapsto F_\rho(\mathtt{e})]\}) \end{aligned}$$

$$\begin{aligned} &\mathcal{WLP}(*p = e, F_\rho(\mathtt{x}) \boxed{=} 5) \\ &\quad= \overline{\mathcal{F}}[\![F_\rho(\mathtt{x}) \boxed{=} 5]\!]U \\ &\quad= (\overline{\mathcal{T}}[\![F_\rho(\mathtt{x})]\!]U) \boxed{=} (\overline{\mathcal{T}}[\![5]\!]U) \\ &\quad= (\overline{access}(\overline{\mathcal{FE}}[\![F_\rho]\!]U, \overline{\mathcal{T}}[\![\mathtt{x}]\!]U)) \boxed{=} 5 \\ &\quad= (\overline{access}(\overline{lookupFuncId}(U, F_\rho), \overline{lookupId}(U, \mathtt{x}))) \boxed{=} 5 \\ &\quad= (\overline{access}(F_\rho[F_\rho(\mathtt{p}) \mapsto F_\rho(\mathtt{e})], \mathtt{x})) \boxed{=} 5 \\ &\quad= ite(F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}, F_\rho(\mathtt{e}), \overline{access}(F_\rho, \mathtt{x})) \boxed{=} 5 \\ &\quad= ite(F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}, F_\rho(\mathtt{e}), F_\rho(\mathtt{x})) \boxed{=} 5 \end{aligned}$$

$$\overline{access}(F, k_1) = F(k_1)$$
$$\overline{access}(FE[k_2 \mapsto d_2]), k_1) = \begin{cases} d_2 & \text{if } (k_1 \equiv k_2) \\ \overline{access}(FE, k_1) & \text{if } (k_1 \neq k_2) \\ ite(k_1 \boxed{=} k_2, d_2, \overline{access}(FE, k_1)) & \text{if } (k_1 \doteq k_2) \end{cases}$$

$$\overline{update}(F, k_1, d_1) = F[k_1 \mapsto d_1]$$
$$\overline{update}(FE[k_2 \mapsto d_2], k_1, d_1) = \begin{cases} FE[k_1 \mapsto d_1] & \text{if } (k_1 \equiv k_2) \\ \overline{update}(FE, k_1, d_1)[k_2 \mapsto d_2] & \text{if } (k_1 \neq k_2) \\ FE[k_2 \mapsto d_2][k_1 \mapsto d_1] & \text{if } (k_1 \doteq k_2) \end{cases}$$

Fig. 10. Simplifications performed by $\overline{access}$ and $\overline{update}$. The operations $\equiv$, $\neq$, and $\doteq$ denote *equality-as-terms*, *definite-disequality*, and *possible-equality*, respectively. (The possible-equality tests, "$k_1 \doteq k_2$", are really "otherwise" cases of three-pronged comparisons.)

Note how the case for $\overline{access}$ that involves a possible-equality comparison causes an *ite* term to arise that tests "$F_\rho(\mathtt{p}) \boxed{=} \mathtt{x}$". The test determines whether the value of p is the address of x, which is the only aliasing condition that matters for this example. □

Although $\mathcal{WLP}$ is sometimes confused with the formula-manipulation operations used to obtain a formula that expresses it, or with the formula ψ that results, $\mathcal{WLP}$ is really a semantic notion—the set of states *described* by ψ. For example, for any statement s: $var = rhs$; in a language that only has `int`-valued variables, and postcondition formula φ, the formula $\varphi[var \leftarrow rhs]$ obtained by substitution is not the only formula that expresses $\mathcal{WLP}(s, \varphi)$. In fact, there are an infinity of acceptable formulas. A formula ψ is acceptable if ψ holds in the pre-state structure ι exactly when φ holds in the post-state structure $\mathcal{I}[\![s]\!]\iota$.

Definition 1 (Acceptable $\mathcal{WLP}$ Formula). ψ is an *acceptable* formula for $\mathcal{WLP}(s, \varphi)$ iff, for all $\iota \in$ *LogicalStruct*,

$$\mathcal{F}[\![\psi]\!]\iota = \mathcal{F}[\![\varphi]\!](\mathcal{I}[\![s]\!]\sigma),$$

where σ is the *State* that corresponds to *LogicalStruct* ι (i.e., $\sigma = ((\iota\uparrow 1), (\iota\uparrow 2)F_\rho)$; see App. A).

The correctness of the $\mathcal{WLP}$ primitive defined in Eqn. (1) is captured by the following theorem:

Theorem 2. *For any Stmt s and Formula φ, $\psi := \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$ is an acceptable $\mathcal{WLP}$ formula for φ with respect to s.*

Proof. See App. A.2.

Symbolic Composition. The goal of symbolic composition is to have a method that, given two symbolic representations of state changes, computes a symbolic representation of their composed state change. In our approach, each state change is represented in logic L[PL] by a *StructUpdate*, and the method computes a new *StructUpdate* that represents their composition. To accomplish this, L[PL] is used as a reinterpretation domain, exactly as for $\mathcal{WLP}$. Moreover, $\overline{\mathcal{U}}$ *turns out to be exactly the symbolic-composition function that we seek.* In particular, $\overline{\mathcal{U}}$ works as follows:

$$\begin{aligned} &\overline{\mathcal{U}}[\![(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})]\!]U \\ &\quad = ((U\uparrow 1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U], (U\uparrow 2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U]) \end{aligned}$$

Example 7. At the syntactic level, we can demonstrate the ability of $\overline{\mathcal{U}}$ (plus simple algebraic simplification) to perform symbolic composition by showing that for the swap-code fragment from Fig. 1(a)

$$\overline{\mathcal{I}}[\![s_1; s_2; s_3]\!]U_{id} = \overline{\mathcal{U}}[\![\overline{\mathcal{I}}[\![s_3]\!]U_{id}]\!](\overline{\mathcal{I}}[\![s_1; s_2]\!]U_{id}).$$

First, consider the left-hand side. As shown in Fig. 8, $\overline{\mathcal{I}}[\![s_1; s_2; s_3]\!]U_{id} = (\emptyset, F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})[\mathtt{y} \mapsto F_\rho(\mathtt{x})]) = U_{swap}$. Now consider the right-hand side. Let $U_{1,2}$ and U_3 be defined as follows:

$$\begin{aligned} U_{1,2} &= \overline{\mathcal{I}}[\![s_1; s_2]\!]U_{id} \\ &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\ U_3 &= \overline{\mathcal{I}}[\![s_3]\!]U_{id} \\ &= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]\}). \end{aligned}$$

As shown in Fig. 11,

$$\overline{\mathcal{U}}[\![U_3]\!]U_{1,2} = (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}).$$

Therefore, $\overline{\mathcal{U}}[\![U_3]\!]U_{1,2} = U_{swap}$. □

The semantic correctness of the symbolic-composition primitive $\overline{\mathcal{U}}$ is captured by the following theorem, which shows that the meaning of $\overline{\mathcal{U}}[\![U_2]\!]U_1$ is the composition of the meanings of U_2 and U_1:

Theorem 3. *For all $U_1, U_2 \in$ StructUpdate,*

$$\mathcal{U}[\![\overline{\mathcal{U}}[\![U_2]\!]U_1]\!] = \mathcal{U}[\![U_2]\!] \circ \mathcal{U}[\![U_1]\!].$$

Proof. See App. A.3.

$$
\begin{aligned}
\overline{\mathcal{U}}[\![U_3]\!]U_{1,2} &= \overline{\mathcal{U}}[\![(\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]\})]\!]U_{1,2} \\
&= (\emptyset, (U_{1,2}\!\uparrow\!2)[F_\rho \mapsto \overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]]\!]U_{1,2}]) \\
&= (\emptyset, \{F'_\rho \hookleftarrow \overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{y})]]\!]U_{1,2}\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow \overline{update} \begin{pmatrix} \overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]]\!]U_{1,2}, \\ \overline{\mathcal{T}}[\![\mathtt{y}]\!]U_{1,2}, \\ \overline{\mathcal{T}}[\![F_\rho(\mathtt{y})]\!]U_{1,2} \end{pmatrix}\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow (\overline{FE}[\![F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]]\!]U_{1,2})[\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow \overline{update} \begin{pmatrix} \overline{FE}[\![F_\rho]\!]U_{1,2}, \\ \overline{\mathcal{T}}[\![\mathtt{x}]\!]U_{1,2}, \\ \overline{\mathcal{T}}[\![F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]\!]U_{1,2} \end{pmatrix} [\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow (F_\rho[\mathtt{x} \mapsto \overline{\mathcal{T}}[\![F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})]\!]U_{1,2}][\mathtt{y} \mapsto F_\rho(\mathtt{x})])[\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto ((F_\rho(\mathtt{x}) \boxed{\oplus} F_\rho(\mathtt{y})) \boxed{\oplus} F_\rho(\mathtt{x}))][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= (\emptyset, \{F'_\rho \hookleftarrow F_\rho[\mathtt{x} \mapsto F_\rho(\mathtt{y})][\mathtt{y} \mapsto F_\rho(\mathtt{x})]\}) \\
&= U_{swap}
\end{aligned}
$$

Fig. 11. Example of symbolic composition.

5 Symbolic Analysis for MC via Reinterpretation

To obtain the three symbolic-analysis primitives for MC, we use a reinterpretation of MC's semantics that is essentially identical to the reinterpretation for PL, modulo the fact that the semantics of PL is written in terms of the combinators *lookupEnv*, *lookupStore*, and *updateStore*, whereas the semantics of MC is written in terms of $lookup_{reg}$, $store_{reg}$, $lookup_{flag}$, $store_{flag}$, $lookup_{mem}$, and $store_{mem}$.

Symbolic Evaluation. The base types are redefined as $\overline{BVal} = Formula$, $\overline{Val} = Term$, $\overline{State} = StructUpdate$, where the vocabulary for *LogicalStruct*s is

$$(\{ZF, EAX, EBX, EBP, EIP\}, \{F_{mem}\}).$$

Lookup and store operations for MC, such as $\overline{lookup}_{mem}$ and $\overline{store}_{mem}$, are handled the same way that $\overline{lookupStore}$ and $\overline{updateStore}$ are handled for PL.

$$
\begin{aligned}
\overline{lookup}_{mem} &: StructUpdate \to Term \to Term \\
\overline{lookup}_{mem} &= \lambda U.\lambda T.((U\!\uparrow\!2)F_{mem})(T) \\
\overline{store}_{mem} &: StructUpdate \to Term \to Term \to StructUpdate \\
\overline{store}_{mem} &= \lambda U.\lambda T_1.\lambda T_2. \\
&\qquad ((U\!\uparrow\!1), (U\!\uparrow\!2)[F_{mem} \mapsto ((U\!\uparrow\!2)F_{mem})[T_1 \mapsto T_2]]) \\
\overline{lookup}_{reg} &: StructUpdate \to register \to Term \\
\overline{lookup}_{reg} &= \lambda U.\lambda r.(U\!\uparrow\!1)(r) \\
\overline{store}_{reg} &: StructUpdate \to register \to Term \\
&\qquad \to StructUpdate \\
\overline{store}_{reg} &= \lambda U.\lambda r.\lambda T.((U\!\uparrow\!1)[r \mapsto T], (U\!\uparrow\!2))
\end{aligned}
$$

Because we placed *ZF* in the set of constant symbols (which denote *Int32* values), we use the following definitions of $\overline{lookup}_{flag}$ and $\overline{store}_{flag}$, where in $\overline{store}_{flag}$ the *Int32* values 1 and 0 encode $\mathbb{T}$ and $\mathbb{F}$, respectively.[3]

$$
\begin{aligned}
\overline{lookup}_{flag} &: StructUpdate \to flagName \to Formula \\
\overline{lookup}_{flag} &= \lambda U.\lambda f.((U\!\uparrow\!1)(f) \boxed{=} 1) \\
\overline{store}_{flag} &: StructUpdate \to flagName \to Formula \\
&\qquad \to StructUpdate \\
\overline{store}_{flag} &= \lambda U.\lambda f.\lambda \varphi.((U\!\uparrow\!1)[f \mapsto ite(\varphi, 1, 0)], (U\!\uparrow\!2))
\end{aligned}
$$

Example 8. Fig. 1(d) shows the MC code that corresponds to the swap code in Fig. 1(a): lines 1–3, lines 4–6, and lines 7–9 correspond to lines 1, 2, and 3 of Fig. 1(a), respectively. For the MC code in Fig. 1(d), $\overline{\mathcal{I}}_{\mathrm{MC}}[\![swap]\!]U_{id}$, which denotes the symbolic evaluation of *swap*, produces the *StructUpdate*

$$
\begin{pmatrix} \{EAX' \hookleftarrow F_{mem}(EBP \boxed{-} 14)\}, \\ \left\{ \begin{aligned} F'_{mem} \hookleftarrow F_{mem}[&EBP \boxed{-} 10 \mapsto F_{mem}(EBP \boxed{-} 14)] \\ [&EBP \boxed{-} 14 \mapsto F_{mem}(EBP \boxed{-} 10)] \end{aligned} \right\} \end{pmatrix}
$$

Fig. 1(d) illustrates why it is essential to be able to handle address arithmetic: an access on a source-level variable is compiled into machine code that dereferences an address in the stack frame computed from the frame pointer (*EBP*) and an offset. This example shows that $\overline{\mathcal{I}}_{\mathrm{MC}}$ is able to handle address arithmetic correctly. □

WLP. To create a formula for the $\mathcal{WLP}$ of φ with respect to instruction i via semantic reinterpretation, we use the reinterpreted MC semantics $\overline{\mathcal{I}}_{\mathrm{MC}}$, together with

[3] To simplify the exposition, L is intentionally a limited logic over values of type *Int32*. To define $\overline{lookup}_{flag}$ and $\overline{store}_{flag}$, it would be more convenient to use a logic with Boolean-valued constant symbols $B_j \in BoolId$, in which case a *StructUpdate* would be a triple of the form

$$(\{I_i \hookleftarrow T_i\}, \{B_j \hookleftarrow \varphi_j\}, \{F_k \hookleftarrow FE_k\}),$$

and $\overline{lookup}_{flag}$ and $\overline{store}_{flag}$ could be defined as follows:

$$
\begin{aligned}
\overline{lookup}_{flag} &= \lambda U.\lambda f.(U\!\uparrow\!2)(f) \\
\overline{store}_{flag} &= \lambda U.\lambda f.\lambda \varphi.((U\!\uparrow\!1), (U\!\uparrow\!2)[f \mapsto \varphi], (U\!\uparrow\!3))
\end{aligned}
$$

```
[1] void foo(int e, int x, int* p) {
[2]     ...
[3]     *p = e;
[4]     if(x == 5)
[5]         goto ERROR;
[6] }
```

(a)

```
[1] mov  eax, p;
[2] mov  ebx, e;
[3] mov  [eax], ebx;
[4] cmp  x, 5;
[5] jz   ERROR;
[6] ...
[7] ERROR: ...
```

(b)

Fig. 12. (a) A simple source-code fragment written in PL; (b) the MC code for (a).

the reinterpreted L[MC] meaning function $\overline{\mathcal{F}}_{\text{MC}}$, where $\overline{\mathcal{F}}_{\text{MC}}$ is created via the same approach used in §4 to reinterpret L[PL]. $\mathcal{WLP}(i, \varphi)$ is obtained by performing $\overline{\mathcal{F}}_{\text{MC}}[\![\varphi]\!](\overline{\mathcal{I}}_{\text{MC}}[\![i]\!]U_{id})$.

Example 9. Fig. 12(a) shows a source-code fragment; Fig. 12(b) shows the corresponding MC code. (To simplify the MC code, source-level variable names are used.) In Fig. 12(a), the largest set of states just before line `[3]` that cause the branch to `ERROR` to be taken at line `[4]` is described by $\mathcal{WLP}(*p = e, x = 5)$. In Fig. 12(b), an expression that characterizes whether the branch to `ERROR` is taken is $\mathcal{WLP}(s_{\texttt{[1]-[5]}}, (EIP \boxed{=} c_{\texttt{[7]}}))$, where $s_{\texttt{[1]-[5]}}$ denotes instructions `[1]`–`[5]` of Fig. 12(b), and $c_{\texttt{[7]}}$ is the address of `ERROR`. Using semantic reinterpretation,

$$\overline{\mathcal{F}}_{\text{MC}}[\![(EIP \boxed{=} c_{\texttt{[7]}})]\!](\overline{\mathcal{I}}_{\text{MC}}[\![s_{\texttt{[1]-[5]}}]\!]U_{id})$$

produces the formula

$$(ite((F_{mem}(\texttt{p}) \boxed{=} \texttt{x}), F_{mem}(\texttt{e}), F_{mem}(\texttt{x})) \boxed{-} 5) \boxed{=} 0,$$

which, transliterated to informal source-level notation, is $(((p = \&x)\ ?\ e : x) - 5) = 0$.

Even though the (source-level) branch is split across two instructions in Fig. 12(b), $\mathcal{WLP}$ can be used to recover the branch condition. First,

$$\mathcal{WLP}(\texttt{cmp x,5; jz ERROR}, (EIP \boxed{=} c_{\texttt{[7]}}))$$

returns the formula

$$ite(((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0), c_{\texttt{[7]}}, c_{\texttt{[6]}}) \boxed{=} c_{\texttt{[7]}},$$

as shown by the following derivation:

$$\overline{\mathcal{I}}_{\text{MC}}[\![\texttt{cmp x,5}]\!]U_{id} = (\{ZF' \hookleftarrow ite((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0, 1, 0)\}, \emptyset) = U_1$$

$$\overline{\mathcal{I}}_{\text{MC}}[\![\texttt{jz ERROR}]\!]U_1 = \left(\left\{\begin{array}{l} ZF' \hookleftarrow ite((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0, 1, 0) \\ EIP' \hookleftarrow ite\begin{pmatrix}((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0), \\ c_{\texttt{[7]}}, \\ c_{\texttt{[6]}}\end{pmatrix}\end{array}\right\}, \emptyset\right) = U_2$$

$$\overline{\mathcal{F}}_{\text{MC}}[\![EIP \boxed{=} c_{\texttt{[7]}}]\!]U_2 = ite\begin{pmatrix}((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0), \\ c_{\texttt{[7]}}, \\ c_{\texttt{[6]}}\end{pmatrix} \boxed{=} c_{\texttt{[7]}}$$

Second, because $c_{\texttt{[7]}} \neq c_{\texttt{[6]}}$, the formula in the last line simplifies to $(F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0$; i.e., in source-level terms, $(x - 5) = 0$. □

Symbolic Composition. For MC, symbolic composition can be performed using $\overline{\mathcal{U}}_{\text{MC}}$.

6 Other Language Constructs

Branching. Ex. 9 illustrated a $\mathcal{WLP}$ computation across a machine-code branch instruction. We now illustrate forward symbolic evaluation across a branch.

Example 10. Suppose that an if-statement is represented by

$$\mathit{IfStmt}(\mathit{BE}, \mathit{Int32}, \mathit{Int32}),$$

where BE is the condition and the two *Int32*s are the addresses of the true-branch and false-branch, respectively. Its factored semantics would specify how the value of the program counter PC changes:

$$\mathcal{I}[\![\mathit{IfStmt}(\mathit{BE}, c_T, c_F)]\!]\sigma = \mathit{updateStore}\ \sigma\ PC\ \mathit{cond}(\mathcal{B}[\![\mathit{BE}]\!]\sigma, \mathit{const}(c_T), \mathit{const}(c_F)).$$

In the reinterpretation for symbolic evaluation, the *StructUpdate* U obtained by $\overline{\mathcal{I}}[\![\mathit{IfStmt}(\mathit{BE}, c_T, c_F)]\!]U_{id}$ would be $(\{PC' \hookleftarrow \mathit{ite}(\varphi_{\mathit{BE}}, c_T, c_F)\}, \emptyset)$, where φ_{BE} is the *Formula* obtained for BE under the reinterpreted semantics. To obtain the branch condition for a specific branch, say the true-branch, we evaluate $\overline{\mathcal{F}}[\![PC \boxed{=} c_T]\!]U$. The result is $(\mathit{ite}(\varphi_{\mathit{BE}}, c_T, c_F) \boxed{=} c_T)$, which (assuming that $c_T \neq c_F$) simplifies to φ_{BE}. (A similar formula simplification was performed in Ex. 9 on the result of the $\mathcal{WLP}$ formula.)

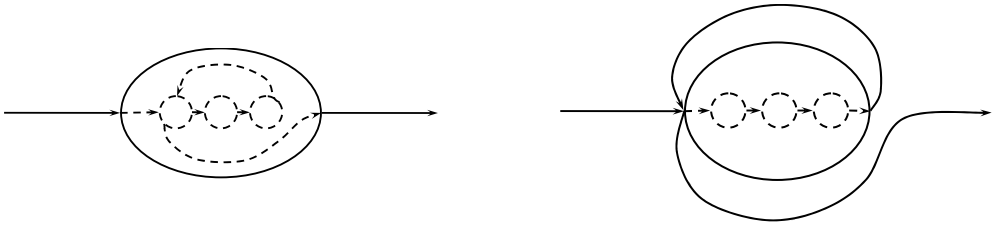
□

Loops. One kind of intended client of our approach to creating symbolic-analysis primitives is hybrid concrete/symbolic state-space exploration [11,34,12,5]. Such tools use a combination of concrete and symbolic evaluation to generate inputs that increase coverage. In such tools, a program-level loop is executed concretely a specific number of times as some path π is followed. The symbolic-evaluation primitive for a single instruction is applied to each instruction of π to obtain symbolic states at each point of π. A *path-constraint formula* that characterizes which initial states must follow π can be obtained by collecting the branch formula φ_{BE} obtained at each branch condition by the technique described above; the algorithm is shown in Fig. 13.

X86 String Instructions. X86 string instructions can involve actions that perform an *a priori* unbounded

```
Formula ObtainPathConstraintFormula(Path π) {
  Formula φ = [𝕋];    // Initial path-constraint formula
  StructUpdate U = U_id;    // Initial symbolic state-transformer
  let [PC_1 : i_1, PC_2 : i_2, ..., PC_n : i_n, PC_{n+1} : skip] = π in
  for (k = 1; k ≤ n; k++) {
    U = [I][[i_k]]U;    // Symbolically execute i_k
    φ = φ [&&] [F][[PC [=] PC_{k+1}]]U;    // Conjoin the branch condition for i_k
  }
  return φ;
}
```

Fig. 13. An algorithm to obtain a path-constraint formula that characterizes which initial states must follow path π.



(a) (b)

Fig. 14. Conversion of a recursively defined instruction—portrayed in (a) as a "microcode loop" over the actions denoted by the dashed circles and arrows—into (b), an explicit loop in the control-flow graph whose body is an instruction defined without using recursion. The three microcode operations in (b) correspond to the three operations in the body of the microcode loop in (a).

amount of work (e.g., the amount performed is determined by the value held in register *ECX* at the start of the instruction). This can be reduced to the loop case discussed above by giving a semantics in which the instruction itself is one of its two successors. In essence, the "microcode loop" is converted into an explicit loop (see Fig. 14).

Procedures. A call statement's semantics (i.e., how the state is changed by the call action) would be specified with some collection of operations. Again, the reinterpretation of the state transformer is induced by the reinterpretation of each operation:

- For a call statement in a high-level language, there would be an operation that creates a new activation record. The reinterpretation of this would generate a fresh logical constant to represent the location of the new activation record.
- For a call instruction in a machine-code language, register operations would change the stack pointer and frame pointer, and memory operations would initialize fields of the new activation record. These are reinterpreted in exactly the same way that register and memory operations are reinterpreted for other constructs.

Dynamic Allocation. Two approaches are possible:

- The allocation package is implemented as a library. One can apply our techniques to the machine code from the library.
- If a formula is desired that is based on a high-level semantics, a call statement that calls `malloc` or `new` can be reinterpreted using the kind of approach used in other systems (a fresh logical constant denoting a new location can be generated).

7 Incorporating Non-Determinism

Many formalisms for symbolic analysis of programs support the use of non-determinism, which is useful for writing "harness code" (code that models the possible client environments from which the code being analyzed might be called), as well as for modeling the possible inputs to a program. A common approach is to provide a primitive that returns an arbitrary value of a given type. Examples include the `SdvMakeChoice` primitive of SLAM [1] and the `havoc(x)` primitive of BoogiePL [2]. In this section, we discuss adding such a primitive, *CALL randInt32*, to MC. *CALL randInt32* is an instruction that assigns an arbitrary value to register *EAX*.[4] We refer to MC extended with *CALL randInt32* as NDMC.

This section describes how implementations of the basic primitives used in symbolic program analysis are obtained for NDMC. (Essentially the same method can be applied to a version of PL extended with its own primitive for generating an arbitrary *Int32* value.)

[4] In the x86 instruction set, register *EAX* is used to pass back the return value from a function call.

Because our approach to creating implementations of the primitives used in symbolic program analysis is based on semantic reinterpretation, our goal is to give a concrete semantics for *CALL randInt32* whose reinterpretation produces the desired effect. At an intuitive level, we would like to treat each invocation of *CALL randInt32* as reading the next input value, and have the semantics of the program arrange to record all of the input values. To carry out something equivalent to this, we assume that the meta-language in which semantic specifications are written supports a primitive for creating a *random map*, which is a map initialized with arbitrary values.[5] Rather than recording input values, we will materialize—in a random map that is part of the input state—the sequence of non-deterministic values that *EAX* will receive on successive calls to *CALL randInt32*. The state will also contain an index-variable, which indicates the index of the next choice. Thus, all non-determinism in the concrete semantics is pushed onto the initialization of the random map in the initial state; all transitions thereafter are deterministic.

The *CALL randInt32* instruction and its semantics are defined as an extension of the MC language presented in §3.3:

$$instruction := \ldots \mid CALL\ randInt32$$

An NDMC state is defined in terms of

$$\begin{aligned} choiceMap &\in Val \rightarrow Val \\ choiceIndex &\in Val \end{aligned}$$

and an NDMC state $\sigma \in State$ is now a quintuple

$$(mem, reg, flag, choiceMap, choiceIndex),$$

where *choiceMap* is a random map.

$$\begin{aligned} &lookup_{choiceMap} : State \rightarrow Val \\ &lookup_{choiceMap} = \\ &\quad \lambda(mem, reg, flag, choiceMap, choiceIndex) \\ &\qquad\qquad .choiceMap(choiceIndex) \end{aligned}$$

$$\begin{aligned} &incr_{choiceIndex} : State \rightarrow State \\ &incr_{choiceIndex} = \\ &\quad \lambda(mem, reg, flag, choiceMap, choiceIndex) \\ &\qquad .(mem, reg, flag, choiceMap, choiceIndex + 1) \end{aligned}$$

The concrete semantics of *CALL randInt32* is defined as follows:

$$\begin{aligned} &\mathcal{I}[\![CALL\ randInt32]\!]\sigma \\ &\qquad = incr_{eip}\left(store_{reg}\begin{pmatrix} incr_{choiceIndex}(\sigma), \\ EAX, \\ lookup_{choiceMap}(\sigma) \end{pmatrix}\right) \end{aligned}$$

Reinterpretation in Logic. As before, *State* is reinterpreted as a *StructUpdate*: $\overline{State} = StructUpdate$, where the vocabulary for *LogicalStruct*s is

$$\begin{pmatrix} \{choiceIndex, ZF, EAX, EBX, EBP, EIP\}, \\ \{F_{choiceMap}, F_{mem}\} \end{pmatrix},$$

and U_{id} is

$$\begin{pmatrix} \{choiceIndex' \hookleftarrow choiceIndex, ZF' \hookleftarrow ZF, \ldots\}, \\ \{F'_{choiceMap} \hookleftarrow F_{choiceMap}, F'_{mem} \hookleftarrow F_{mem}\} \end{pmatrix}.$$

$\mathcal{WLP}$ in the Presence of Non-Determinism. In previous sections, we have referred to the backwards-reasoning primitive generated by our method as $\mathcal{WLP}$, which is correct for the situation considered in §4 and §5, namely languages whose primitive statements/instructions are total and deterministic.

In the terminology of relational semantics [35], one considers two backwards-reasoning primitives, *pre* and $\widetilde{pre}$, defined as follows (where R is a binary relation on Q, and φ defines a subset of Q):

$$\begin{aligned} pre[R](\varphi) &= \exists q'.\ (R(q, q') \wedge \varphi(q')) \\ \widetilde{pre}[R](\varphi) &= \forall q'.\ (R(q, q') \Rightarrow \varphi(q')) \end{aligned}$$

pre specifies the set of all predecessors in R of states that satisfy φ. $\widetilde{pre}$ specifies the largest set of states such that for each state q all successors of q (possibly the empty set) satisfy φ.

The backwards-reasoning primitive considered in §4 and §5 could be referred to as either *pre* or $\widetilde{pre}$, because the two operators are identical for total, deterministic transitions. For a non-deterministic transition system, however, *pre* and $\widetilde{pre}$ are different. For instance, execution of the `havoc(x)` primitive of BoogiePL [2] assigns an arbitrary value to `x`. For `havoc(x)`, *pre* and $\widetilde{pre}$ are defined as follows:

$$\begin{aligned} pre[\![\texttt{havoc(x)}]\!](\varphi) &= \exists x.\ \varphi \\ \widetilde{pre}[\![\texttt{havoc(x)}]\!](\varphi) &= \forall x.\ \varphi \end{aligned}$$

The following example shows that the backwards-reasoning primitive created by our technique behaves similarly to *pre*.

Example 11. Consider what the backwards-reasoning primitive creates for $EAX \boxed{=} 5$ with respect to *CALL randInt32*:

$$\begin{aligned} &\overline{\mathcal{I}}[\![CALL\ randInt32]\!]U_{id} \\ &= \left(\begin{pmatrix} \{choiceIndex' \hookleftarrow (U_{id}{\uparrow}1)(choiceIndex) \boxed{+} 1, \\ EAX' \hookleftarrow ((U_{id}{\uparrow}2)(F_{choiceMap}))((U_{id}{\uparrow}1)(choiceIndex))\} \\ (U_{id}{\uparrow}2) \end{pmatrix},\right) \\ &= \left(\begin{pmatrix} \{choiceIndex' \hookleftarrow choiceIndex \boxed{+} 1, \\ EAX' \hookleftarrow F_{choiceMap}(choiceIndex)\} \\ (U_{id}{\uparrow}2) \end{pmatrix},\right) \\ &= U_1 \end{aligned}$$

$$\begin{aligned} &\mathcal{WLP}(CALL\ randInt32, EAX \boxed{=} 5) \\ &\qquad = \overline{\mathcal{F}}[\![EAX \boxed{=} 5]\!]U_1 \\ &\qquad = F_{choiceMap}(choiceIndex) \boxed{=} 5 \end{aligned}$$

□

$F_{choiceMap}$ can be thought of as an array of logical variables. In the quantifier-free logic we work with, formulas are implicitly existentially quantified.

[5] A random map is easy to model in logic L using a function that is unconstrained.

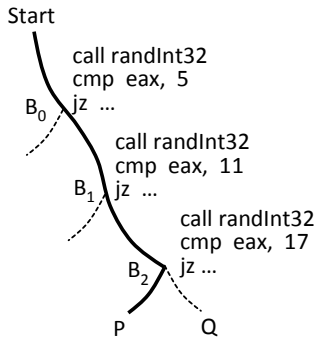


Fig. 15. In a symbolic evaluation of the trace from *Start* to P, the three path constraints obtained from the branch instructions at B_0, B_1, and B_2 constrain the values of $F_{choiceMap}(0)$, $F_{choiceMap}(1)$, and $F_{choiceMap}(2)$, respectively. To create a new initial state that causes a concrete execution of the program to follow the same path, except to branch the opposite way at B_2 (to reach Q), we need the satisfying assignment returned by the theorem prover to satisfy the constraints on $F_{choiceMap}(0)$ and $F_{choiceMap}(1)$ and the negated constraint on $F_{choiceMap}(2)$.

Letting v denote $F_{choiceMap}(choiceIndex)$, the formula $F_{choiceMap}(choiceIndex) \boxed{=} 5$ can be thought of as the quantifier-free version of the formula $\exists v.v \boxed{=} 5$, which corresponds to $pre[\![\texttt{havoc(v)}]\!](v \boxed{=} 5)$.

Thus, in earlier sections it would have been more precise to have referred to the backwards-reasoning primitive as *pre*, rather than $\mathcal{WLP}$—although the term $\mathcal{WLP}$ was also correct because earlier sections dealt only with languages whose primitive statements/instructions are total and deterministic.

Guaranteed Replay in the Presence of Non-Determinism. The application of directed test generation [11,34,12,5] requires path constraints that enable the test-generation system to create new test inputs that are guaranteed to follow a particular path through the program.[6] In particular, during forward symbolic evaluation, we want path-constraint generation (Fig. 13) to produce a formula such that when a theorem prover is able to provide an assignment that satisfies the formula, the satisfying assignment serves as an initial state that will cause concrete execution of the program to follow a specific path. The paths of interest are ones that replay at least part of a previous execution trace.

The situation is illustrated in Fig. 15. During directed test generation, suppose that a concrete execution trace T follows the path from *Start* to P. Associated with T are three path constraints obtained from the branch instructions at B_0, B_1, and B_2. The three constraints constrain the values of $F_{choiceMap}(0)$, $F_{choiceMap}(1)$, and $F_{choiceMap}(2)$, respectively. To increase branch coverage, a directed-test-generation tool would like to obtain an initial state that drives the program along the same path, except when it reaches B_2, when the program should proceed to Q.

[6] See §8 and §9 for more detailed discussion of systems for directed test generation.

With the scheme presented in this section, the theorem prover is able to create such an initial state by providing initial values for the first three entries of $F_{choiceMap}$ (which models the random map *choiceMap*).

Repeatability comes from the fact that we have kept the concrete semantics deterministic by, in essence, recording all non-deterministically chosen values in a kind of shadow input stream. As a result, repeatability is automatically obtained for both symbolic evaluation as well as $\mathcal{WLP}$. In each case, for a given path we obtain an assignment for the input that forces execution along that path: in symbolic evaluation, one works forwards and collects path constraints; in $\mathcal{WLP}$, one works backwards starting from $\mathbb{T}$; the solver is constrained to return an assignment that, at each branch instruction, causes a concrete execution to branch in the direction that stays on the path.

8 Implementation and Evaluation

Implementation. In our implementation, the abstract syntax and concrete semantics of an instruction set are specified using a language called TSL (**T**ransformer **S**pecification **L**anguage) [25]. Decoding (i.e., translation of binary-encoded instructions to abstract syntax trees) is specified using a tool called ISAL (**I**nstruction **S**et **A**rchitecture **L**anguage). The relationship between ISAL and TSL is similar to the relationship between Flex and Bison—i.e., a Flex-generated lexer passes tokens in a shared language of tokens to a Bison-generated parser. In our case, a shared language of abstract syntax trees serves as the formalism for communicating values—namely, abstract syntax trees for instances of specific instructions—from ISAL to TSL.

Compared with other specification languages for instruction sets, TSL has one unique feature: from a *single* specification of the concrete semantics of an instruction set, a *multiplicity* of static-analysis, dynamic-analysis, and symbolic-analysis components can be *generated automatically*. The TSL system consists of two parts:

- The TSL language for specifying an instruction set's abstract syntax and concrete semantics. TSL is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching.
- The TSL compiler, which translates a specification to a common intermediate representation (CIR). The CIR generated for a given TSL specification is a C++ template that can be used to create multiple analysis components by instantiating the template in different ways.

TSL has two classes of users: (1) *instruction-set specifiers* use the TSL language to specify the concrete semantics of

different instruction sets; (2) *analysis developers* create new analyses by instantiating the CIR in different ways.

The TSL language is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Writing a TSL specification for an instruction set is similar to writing an interpreter in first-order ML. One specifies (i) the abstract syntax of the instruction set, by defining the constructors for a (reserved, but user-defined) type `instruction`; (ii) a type for concrete states, by defining—e.g., for 32-bit Intel x86—the type `state` as a triple of maps:

$$\texttt{state} : \texttt{State} \begin{pmatrix} \texttt{INT32} \rightarrow \texttt{INT8}, \\ \texttt{reg32} \rightarrow \texttt{INT32}, \\ \texttt{flag} \rightarrow \texttt{BOOL} \end{pmatrix};$$

where `INT32` and `INT8` refer to 32-bit and 8-bit integers, respectively, and `reg32` and `flag` are types for the names of 32-bit registers and the names of condition-codes, respectively; and (iii) the concrete semantics of each instruction by writing a TSL function

```
state interpInstr(instruction I, state S){...};
```

Each analysis component is defined by reinterpreting the constructs of the TSL meta-language. The meta-language supports a fixed set of base-types; a fixed set of arithmetic, bitwise, relational, and logical operators; and a facility for defining map-types. Each TSL *reinterpretation* is defined over the *meta-language constructs*, by redefining (in C++) the TSL base-types and base-type operators, and (if necessary) the map-types and map-type operators (i.e., *access* and *update*). These are used to instantiate the CIR template. Each instantiation defines an alternative interpretation of each expression and function in a semantic definition, and thereby yields an alternative semantics.

We used TSL to (1) define the syntax of $L[\cdot]$ as a user-defined datatype; (2) create a reinterpretation based on $L[\cdot]$ formulas; (3) define the semantics of $L[\cdot]$ by writing functions that correspond to $\mathcal{T}$, $\mathcal{F}$, etc.; and (4) apply reinterpretation (2) to the meaning functions of $L[\cdot]$ itself. (We already had in hand TSL specifications of x86 and PowerPC.)

When semantic reinterpretation is performed in the manner supported by TSL, it is *independent* of any given subject language. Consequently, now that we have carried out steps (1)–(4), all three symbolic-analysis primitives can be generated automatically for a new instruction set *IS* merely by writing a TSL specification of *IS*, and then applying the TSL compiler. In essence, TSL acts as a "Yacc-like" tool for generating symbolic-analysis primitives from a semantic description of an instruction set.

To illustrate the leverage gained by using the approach presented in this paper, the table shown in Fig. 16 lists the number of (non-blank) lines of C++ that are generated from the TSL specifications of the x86 and PowerPC instruction sets. The number of (non-blank) lines of TSL are indicated in bold.

The C++ code is emitted as a template, which can be instantiated with different interpretations. For instance, instantiations that create C++ implementations of $\mathcal{I}_{\text{x86}}[\![\cdot]\!]$ and $\mathcal{I}_{\text{PowerPC}}[\![\cdot]\!]$ (i.e., emulators for x86 and PowerPC, respectively) can be obtained trivially. Thus, for a hybrid concrete/symbolic tool for x86, our tool essentially furnishes 23,109 lines of C++ for the concrete-execution component and 23,109 lines of C++ for the symbolic-evaluation component. Note that the 1,510 lines of TSL that defines $\mathcal{F}[\![\cdot]\!]$, $\mathcal{T}[\![\cdot]\!]$, $FE[\![\cdot]\!]$, and $\mathcal{U}[\![\cdot]\!]$ needs to be written only once.

In addition to the components for concrete and symbolic evaluation, one also obtains an implementation of $\mathcal{WLP}$—via the method described in §4—by calling the C++ implementations of $\overline{\mathcal{F}}[\![\cdot]\!]$ and $\overline{\mathcal{I}}[\![\cdot]\!]$: $\mathcal{WLP}(s, \varphi) = \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$. By Thm. 2 of App. A, $\mathcal{WLP}$ is guaranteed to be consistent with the components for concrete and symbolic evaluation (modulo bugs in the implementation of TSL).

Evaluation. Some tools that use symbolic reasoning employ formula transformations that are not faithful to the actual semantics. For instance, the SAGE system for directed test generation [12] uses an approximate x86 symbolic evaluation in which concrete values are used when non-linear operators or symbolic pointer dereferences are encountered. As a result, its symbolic evaluation of a path can produce an "unfaithful" path-constraint formula φ; that is, an actual execution path may not match the program path predicted by the path-constraint formula φ. This situation is called a *divergence* [12]. Because the intended use of SAGE is to generate inputs that increase coverage, it can be acceptable for the tool to have a substantial divergence rate (due to the use of unfaithful symbolic techniques) if the cost of performing symbolic operations is lowered in most circumstances.

In contrast with directed test generation, to model check machine code [21] an implementation of a faithful symbolic technique is required. A faithful symbolic technique could raise the cost of performing symbolic operations because faithful path-constraint formulas could be a great deal more complex than unfaithful ones. Thus, our experiment was designed to answer the question

> "What is the cost of using exact symbolic-evaluation primitives instead of unfaithful ones?"

It would have been an error-prone task to implement a faithful symbolic-evaluation primitive for x86 machine code manually. Using TSL, however, we were able to generate a faithful symbolic-evaluation primitive from an existing, well-tested TSL specification of the semantics of x86 instructions. We also generated an unfaithful symbolic-evaluation primitive that adopts SAGE's approximate approach. We used these to create two

	TSL Specifications		Generated C++ Templates	
	$\mathcal{I}[\![\cdot]\!]$	$\mathcal{F}[\![\cdot]\!] \cup \mathcal{T}[\![\cdot]\!] \cup FE[\![\cdot]\!] \cup \mathcal{U}[\![\cdot]\!]$	$\overline{\mathcal{I}}[\![\cdot]\!]$	$\overline{\mathcal{F}}[\![\cdot]\!] \cup \overline{\mathcal{T}}[\![\cdot]\!] \cup \overline{FE}[\![\cdot]\!] \cup \overline{\mathcal{U}}[\![\cdot]\!]$
x86	**3,524**	**1,510**	23,109	15,632
PowerPC	**1,546**	(already written)	12,153	15,632

Fig. 16. The number of (non-blank) lines of C++ that are generated from the TSL specifications of the x86 and PowerPC instruction sets. The number of (non-blank) lines of TSL are indicated in bold.

Name (STL)	#Tests	|Trace| (#Instrs)	#Branches	Faithful					Approximate					Slowdown
				CE	SE	SMT	$\|\varphi\|$	Div.	CE+SE	SMT	$\|\varphi\|$	Div.	Dist.	(T_F/T_A)
copy	12	1462	19	0.3	3.44	0.017	**6**	**0%**	3.58	0.013	**1**	**50%**	93%	1.05
equal	202	1604	64	0.33	5.56	0.48	**54**	**0%**	5.75	0.46	**24**	**60%**	73%	1.11
find	344	1240	174	0.15	5.34	0.2	**144**	**0%**	5.31	0.17	**85**	**50%**	82%	1.07
partition	19	1293	43	0.24	5.26	0.79	**43**	**0%**	5.43	0.26	**1**	**73%**	87%	1.16
random_shuffle	94	2448	71	0.48	7.56	0.028	**37**	**0%**	7.88	0.014	**1**	**48%**	99%	1.03
search	274	1422	107	0.33	6.3	0.17	**59**	**0%**	6.37	0.13	**31**	**55%**	89%	1.07
transform	200	3749	95	0.82	18.56	0.05	**85**	**0%**	19.36	0.012	**1**	**64%**	99%	1.00

Table 1. Experimental results. Key: CE = time for concrete execution; SE = time for symbolic execution; SMT = solver time; $|\varphi|$ = avg. number of constraints found; Div. = divergence rate; CD+SE = time for concrete + symbolic execution (when run in lock-step); Dist. = avg. distance before a diverging test diverges. T_F/T_A denotes the ratio of the times (CE+SE+SMT) for the faithful version and the approximate version. (All times are in seconds.)

directed-test-generation tools that perform state-space exploration—one that uses the faithful primitive, and one that uses the unfaithful primitive.

Although the presentation in earlier sections was couched in terms of simplified core languages, the implemented tools work with real x86 programs. Our experiment used seven C++ programs, each exercising a single algorithm from the C++ STL, compiled under Visual Studio 2005.

To compare the two tools' divergence rates and running times, we used the algorithm shown in Fig. 17. All execution runs were performed on a single core of a quad-core 3.0GHz Pentium Xeon processor running Windows XP, configured so that a user process has 4 GB of memory. Tab. 1 shows the divergence rates and running times that we measured.

Tab. 1 reports the number of tests executed, the average length of the trace obtained from the tests, and the average number of branches in the traces. For the faithful version, we report the average time taken for concrete execution (CE) and symbolic evaluation (SE). In the approximate ("unfaithful") version, concrete execution and symbolic evaluation were done in lock step and their total time is reported in (CE+SE). (All times are in seconds.) For each version, we also report the average time taken by the SMT solver (Yices [9]), the average number of constraints found ($|\varphi|$), and the divergence rate. For the approximate version, we also show the average distance (as a percentage of the total length of the trace) before a diverging test diverged. T_F/T_A denotes the ratio of the times (CE+SE+SMT) for the faithful version and the approximate version.

On average, the unfaithful primitive had a 57% divergence rate (computed as the arithmetic mean of the seven measured divergence rates), whereas no divergences were reported for the faithful primitive. The faithful primitive had 9.27 times more constraints in φ than the unfaithful primitive (computed as the geometric mean of the ratios of the two versions for the seven programs), and was about 1.07 times slower than the unfaithful version (geometric mean).

9 Related Work

Symbolic analysis is used in many recent systems for testing and verification:

- Hybrid concrete/symbolic tools for directed test generation [11,34,12,5] use a combination of concrete and symbolic evaluation to generate inputs that increase coverage. They use concrete evaluation to identify an executable path π. They use symbolic evaluation to obtain a path formula for π, then change the formula to be one for a path π' that follows the same sequence of branches as π, except that at the final branch node π' branches in the direction opposite to the one taken by π, and call an SMT solver to determine if there is an input that drives the program down π'.
- $\mathcal{WLP}$ can be used to create new predicates that split part of a program's abstract state space [1,3].
- Symbolic composition is useful when a tool has access to a formula that summarizes a called procedure's behavior [36]; re-exploration of the procedure is avoided by symbolically composing a path formula with the procedure-summary formula.

However, compared with the way such symbolic-analysis primitives are implemented in existing program-analysis

```
σ := a random initial input state
Perform concrete execution, starting with input state σ, and obtain the trace T
numTracesConsidered := 0; divergences_faithful := 0; divergences_unfaithful := 0
Worklist := {⟨σ, T⟩}; AlreadyConsideredTraces := ∅
while Worklist ≠ ∅ and numTracesConsidered < threshold do
  Select and remove a pair ⟨σ, T⟩ from Worklist
  Perform two symbolic evaluations of T using the faithful and unfaithful symbolic primitives, respectively, generating branch predicates for each branch instruction in T
  Let B_1, B_2, ..., B_k be the branch instructions, in order, in T
  for i := k downto 1 do
    For each of the two symbolic evaluations, conjoin all the branch predicates in T prior to B_i with the negation of the branch predicate for B_i in T, creating path formulas φ_faithful and φ_unfaithful, respectively
    T_B+ := the prefix of T up to and including B_i, plus the intended successor of B_i
    if T_B+ ∈ AlreadyConsideredTraces then
      Break /* Exit the for loop; all prefixes of T_B+ are in AlreadyConsideredTraces, too */
    else
      Insert T_B+ into AlreadyConsideredTraces
    end if
    if φ_faithful is unsatisfiable then
      Continue /* Go to the next iteration of the for loop */
    end if
    σ'_faithful := a satisfying assignment for φ_faithful
    Perform concrete execution, starting with input state σ'_faithful, and obtain the trace T'
    numTracesConsidered := numTracesConsidered + 1
    if T' does not match T_B+ then
      Increment divergences_faithful by 1
    end if
    if φ_unfaithful is unsatisfiable then
      Increment divergences_unfaithful by 1
    else
      σ'_unfaithful := a satisfying assignment for φ_unfaithful
      Perform concrete execution, starting with input state σ'_faithful, and obtain the trace T''
      if T'' does not match T_B+ then
        Increment divergences_unfaithful by 1
      end if
    end if
    Insert ⟨σ'_faithful, T'⟩ into Worklist
  end for
end while
```

Fig. 17. Directed-test-generation algorithm used for comparing the divergence rates of the faithful and unfaithful symbolic-evaluation primitives.

tools, our work has one definite advantage: it creates the key concrete-execution and symbolic-analysis components in a way that ensures by construction that they are *mutually consistent*.

We use a *declarative approach*: one provides a specification of the subject language's *standard* semantics; then, as described in §4 and §5, mutually-consistent implementations of symbolic evaluation, $\mathcal{WLP}$, and symbolic composition are obtained from the subject language's standard semantics by (i) reinterpreting metalanguage constructs in terms of logic, and (ii) reinterpreting a logic's meaning functions. The advantage of this approach is that one obtains implementations of (a) concrete execution, (b) symbolic evaluation, (c) $\mathcal{WLP}$, and (d) symbolic composition from a *single* specification, which removes the possibility of different analysis components having different "views" of the semantics.

It appears to be the case that in most tools, the concrete-execution and symbolic-analysis primitives are not implemented in a way that guarantees such a consistency property. For instance, in the source code for B2 [18] (the next-generation Blast), one finds symbolic evaluation (*post*) and $\mathcal{WLP}$ implemented with different pieces of code, and hence mutual consistency is not guaranteed. $\mathcal{WLP}$ is implemented via substitution, with special-case code for handling pointers. Any modification of the B2 intermediate representation would require

changing both *post* and $\mathcal{WLP}$, and possibly rethinking the substitution method.

Recently, directed-test-generation tools have been created for x86 executables—e.g., SAGE [12] and BITSCOPE [5].

- BITSCOPE is a framework that takes an x86 executable and provides information about execution paths that can be used for additional, more specific analyses, such as finding out what inputs cause erroneous behavior. To perform symbolic evaluation, they first translate each x86 instruction into an intermediate representation that is designed to model the semantics of the original x86 instruction, including all implicit side effects (such as flags that are set), register addressing modes, and other issues. Symbolic evaluation is performed on the IR with a symbolic transformer for each IR statement.
- SAGE is a *white-box fuzz-testing tool* for x86 Windows applications [12]. The system uses *offline, trace-based* constraint generation: concrete execution and symbolic evaluation are performed over a separately recorded, replayable execution trace in which the outcome of each nondeterministic event encountered during the recorded run has been captured. To generate path constraints, SAGE maintains a concrete state and a symbolic state—a pair of stores that associate each memory location and register to a byte-sized value and a *symbolic tag*, which is an expression that represents either an input value or a function of some input values. A symbolic tag is propagated on the trace during the process of symbolic evaluation by using a symbolic transformer written specifically for each instruction. The concrete store is sometimes used to *concretize* symbolic values that are overly complex. In SAGE, symbolic pointer dereferences are intentionally ignored to reduce complexity. SAGE could be improved to increase coverage by using more precise path constraints created from the symbolic-evaluation primitive produced by our technique. §8 shows that the faithful constraints created by our technique dramatically reduce the number of divergences with only a modest (7%) increase in running time.

BITSCOPE uses the approach of translating each instruction to a common intermediate representation (CIR) (see §2), which provides a level of assurance that the concrete-execution and symbolic-evaluation components are mutually consistent. SAGE uses independently created components for capturing execution traces and for path-constraint generation. It also uses approximate techniques during the symbolic-evaluation part of constraint generation; hence, the treatment of program semantics in SAGE is definitely inconsistent, which causes divergences. ($\mathcal{WLP}$ and symbolic composition do not play a role in either SAGE or BITSCOPE.)

Readers should not confuse the topic of the present paper—which focuses on particular reinterpretations that yield three desirable symbolic-analysis primitives for use in program-analysis tools—with the authors' previous paper about the TSL system [25]. As explained in §8, we used TSL as our implementation platform to create the various logic-based reinterpretations that are used to obtain the three primitives. Although logic-based reinterpretation was mentioned in [25] as a way to translate the semantics of an instruction to a formula, it was just one of several reinterpretations sketched in that paper. The idea of applying logic-based reinterpretation to the meaning functions of the logic itself—thereby generating implementations of $\mathcal{WLP}$ and symbolic composition—is entirely new to the present paper.

Moreover, as discussed in §2, the technique of *semantic* reinterpretation itself (i.e., reinterpreting the constructs of the meta-language) is not even required if one wants to obtain consistent implementations of the three symbolic-analysis primitives. Semantic reinterpretation was used in the paper because it allowed us to present the ideas from §4 and §5 in a concise manner. While those ideas can be implemented via semantic reinterpretation (see §8), it is also possible to create the needed reinterpretations by defining a suitable CIR datatype and creating appropriate interpretations of the CIR's node types—as sketched in §2.

Relationship to Partial Evaluation, Binding-Time Analysis, and 2-Level Semantics. In general, the semantic definition of an imperative programming language is a meaning function $\mathcal{I}$ with type $\mathcal{I} : \mathit{Stmt} \times \mathit{State} \rightarrow \mathit{State}$. The objective of a primitive for symbolic evaluation can be stated as follows:

> Given the semantic definition of a programming language, $\mathcal{I} : \mathit{Stmt} \times \mathit{State} \rightarrow \mathit{State}$, together with a specific programming-language statement (or instruction) $s \in \mathit{Stmt}$, create a logical formula that captures the semantics of s.

Given such a goal for the primitive to be created, it is not surprising that partial-evaluation techniques come into play in the tool that generates implementations of such primitives. In essence, we wish to partially evaluate $\mathcal{I}$ with respect to *Stmt* s so that the residual object captures the semantics of s, while at the same time the result is translated to L. Semantic reinterpretation permits us to do this: Let U_s be the *StructUpdate* $\overline{\mathcal{I}}[\![s]\!]U_{id}$. Then U_s is the partial evaluation of $\mathcal{I}$ with respect to s, translated to logic.

In our implementation, the TSL system is supplied with a TSL program for the meaning function $\mathcal{I}$ (i.e., `interpInstr`). Although TSL is not a partial-evaluation system *per se*, for reasons discussed in [24, §3.4], the TSL compiler performs binding-time analysis [19], and annotates the code for `interpInstr` to create an intermediate representation in a two-level language [33]. In our case, Level 1 corresponds to parameter `I` of `interpInstr`, and

Level 2 corresponds to parameter `state`. To generate implementations of symbolic-analysis primitives via semantic reinterpretation, we use two different reinterpretations for the two levels:

- Concrete semantics (C) for Level 1.
- Something close to the Herbrand interpretation (H) for Level 2: operators of L are used as syntactic constructors, but algebraic simplifications are performed whenever possible.

Let `interpInstr-CH` denote `interpInstr-2level` reinterpreted in this fashion. When `interpInstr-CH` is executed, it creates a residual expression as output. Because concrete semantics is used for level 1, all parts of `interpInstr` that are not relevant to the form of `I` are eliminated.

Overall, the TSL compiler and the two interpretations create something that is very similar to a generating extension [19] `interpInstr-gen` for `interpInstr`. If `p` is a two-input program, a *generating extension* `p-gen` is any program with the property that for every input pair a and b,

$$[\![\texttt{p-gen}]\!](a) = \texttt{p}_a, \text{ where } [\![\texttt{p}_a]\!](b) = [\![\texttt{p}]\!](a, b).$$

Thus, $\mathcal{I}$`-gen` is a program such that for every statement s and *State* σ,

$$[\![\mathcal{I}\texttt{-gen}]\!](s) = \mathcal{I}_s, \text{ where } [\![\mathcal{I}_s]\!](\sigma) = [\![\mathcal{I}]\!](s, \sigma).$$

Generating extension `interpInstr-gen` would be a program with the following property:

$$\begin{aligned}[\![\texttt{interpInstr-gen}]\!](\texttt{I}) &= \texttt{interpInstr}_\texttt{I}, \text{where} \\ [\![\texttt{interpInstr}_\texttt{I}]\!](\texttt{S}) &= [\![\texttt{interpInstr}]\!](\texttt{I}, \texttt{S}).\end{aligned}$$

`interpInstr-CH` has similar properties:

$$\begin{aligned}[\![\texttt{interpInstr-CH}]\!](\texttt{I}, U_{id}) &= U_\texttt{I}, \text{where} \\ \mathcal{U}[\![U_\texttt{I}]\!](\texttt{S}) &= [\![\texttt{interpInstr}]\!](\texttt{I}, \texttt{S}).\end{aligned}$$

Consequently, `interpInstr-gen` and `interpInstr-CH` are not the same, although the difference between is quite small. `interpInstr-CH` still requires *two* inputs to be supplied (but we could use the trivial value U_{id} for the second input).

When partial-evaluation machinery is included in the discussion, the explanation is complicated by the number of language levels involved. Consequently, in the body of the paper we chose to base the discussion on the simpler principle of semantic reinterpretation, which has benefits and drawbacks:

- The benefit is that the explanation is simpler, and could also be useful for direct hand implementation when a meta-system such as TSL is not available.
- The drawback is that in some of the sections it may appear that many steps perform rather trivial transliteration of expressions from programming language PL_i into expressions of the corresponding logic $L[\text{PL}_i]$. In part, this is an artifact of trying to present the method in an easy-to-digest manner; in part, it mimics the behavior of a generating extension: copying (or transliterating) the appropriate residual expression is one of the principles of "writing a generating extension by hand" [4, 22].

10 Conclusion

This paper presents a way to obtain automatically mutually-consistent, correct-by-construction implementations of symbolic primitives—in particular, quantifier-free, first-order-logic formulas for (a) symbolic evaluation of a single command, (b) $\mathcal{WLP}$ with respect to a single command, and (c) symbolic composition for a class of formulas that express state transformations. The approach presented in the paper involves *generating* implementations of each of the primitives from a single specification of the subject language's concrete semantics. The generated implementations are guaranteed to be mutually consistent (modulo bugs in the implementation of the program-generation implementation), and also to be consistent with an instruction-set emulator (for concrete execution) that is generated from the same specification of the subject language's concrete semantics.

In the paper, the method used to generate such implementations is semantic reinterpretation, a technique originally introduced by Mycroft and Jones [29, 20] as a method for formulating abstract interpretations. In this paper, we are not doing abstract interpretation *per se* (i.e., to over-approximate the concrete semantics [6]), but we take two-fold advantage of their methodology: we use two separate semantic reinterpretations—(i) reinterpretation of a *programming language's* meaning function(s), and (ii) reinterpretation of a *logic's* meaning function(s). The two kinds of reinterpretations define the key primitives $\overline{\mathcal{I}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{U}}$ from which the desired implementations of symbolic evaluation, $\mathcal{WLP}$, and symbolic composition are obtained.

As far as we are aware, the application of semantic reinterpretation to a logic is a new idea. A related innovation on which our results rest was to define a particular form of state-transformation formula (structure-update expressions) as a first-class notion in the logic. By this device, such formulas could (i) serve as a replacement domain in the reinterpretations of both the programming language's meaning functions and the logic's meaning functions, and (ii) be reinterpreted themselves.

We applied our technique to both the x86 and PowerPC instruction sets, using the TSL system [25] as our implementation platform.[7] §8 discusses the substantial leverage that we obtained using TSL's facilities for semantic reinterpretation: from 6,580 lines of TSL, 101,788

[7] As discussed in §2, other ways of implementing the necessary reinterpretations are possible.

lines of C++ were produced that implement $\mathcal{I}$, $\overline{\mathcal{I}}$, $\overline{\mathcal{F}}$, $\overline{\mathcal{T}}$, $\overline{FE}$, and $\overline{\mathcal{U}}$ for x86 and PowerPC. Moreover, for each instruction set all six primitives are guaranteed to be mutually consistent (modulo bugs in the implementation of TSL and in the implementations of the primitives for the two kinds of reinterpretations).

As proposed by Mycroft and Jones [29,20], in a semantic reinterpretation one refactors the specification of a language's concrete semantics into a suitable form by introducing appropriate combinators that are subsequently redefined. While this style of semantic reinterpretation is supported by the TSL system, ordinarily one never has to be concerned with refactoring a specification. Instead, each reinterpretation is performed at the meta-level; that is, each reinterpretation involves redefining the approximately 40 primitives of the TSL meta-language.[8] In our TSL-based semantic reinterpretations of specifications of the concrete semantics of x86 and PowerPC, we did not have to refactor the specification to introduce any special combinators.

Finally, we conducted an experiment that used the generated primitives on x86 code, compiled under Visual Studio 2005 from C++ STL source code, to gain insight on the question "What is the cost of using exact symbolic-evaluation primitives instead of unfaithful ones in a system for directed test generation?" The experiment showed that using exact symbolic-analysis primitives, as opposed to ones that approximate the real semantics, is slower by a factor of 1.07, but is dramatically more accurate.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
2. M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, 2005.
3. N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *ISSTA*, 2008.
4. L. Birkedal and M. Welinder. Hand-writing program generator generators. In *PLILP*, 1994.
5. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Analysis and Defense*. Springer, 2008.
6. P. Cousot and R. Cousot. Abstract interpretation. In *POPL*, 1977.
7. Coverity, Inc. Coverity Prevent. `www.coverity.com/html/coverity-prevent.html`.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems*, 2008.
9. B. Dutertre and L. de Moura. Yices: An SMT solver, 2006. http://yices.csl.sri.com/.
10. V. Ganesh and D. Dill. A decision procesure for bit-vectors and arrays. In *CAV*, 2007.
11. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.
12. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
13. GrammaTech, Inc. CodeSonar. `www.grammatech.com/products/codesonar`.
14. B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE*, 2006.
15. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
16. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M.* download.intel.com/design/processor/manuals/253666.pdf.
17. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z.* download.intel.com/design/processor/manuals/253667.pdf.
18. R. Jhala and R. Majumdar. B2: Software model checking for C, 2009. www.cs.ucla.edu/∼rupak/b2/.
19. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
20. N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *POPL*, pages 296–306, 1986.
21. A. Lal, J. Lim, and T. Reps. McDash: Refinement-based property verification for machine code. TR-1649, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, June 2009.
22. P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI*, 1996.
23. J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. In *Spin Workshop*, 2009.
24. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. TR-1622, CS Dept., Univ. of Wisconsin, Madison, WI, Oct. 2007.
25. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
26. K. Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., 1993.
27. J. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theor. Found. of Program. Methodology*. Reidel, 1982.
28. P. Mosses. A semantic algebra for binding constructs. In *ICFPC*, 1981.
29. A. Mycroft and N. Jones. A relational framework for abstract interpretation. In *PADO*, 1985.
30. G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.
31. G. Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4), 1989.
32. F. Nielson. Two-level semantics and abstract interpretation. *TCS*, 69:117–242, 1989.

[8] Each of the numeric primitives comes in four bit-widths: 8-bit, 16-bit, 32-bit, and 64-bit. All four must be reinterpreted; however, generally the reinterpretation of a given family of four such numeric primitives can be parameterized on bit-width, so we only count each family as a single primitive.

33. F. Nielson and H. Nielson. *Two-Level Functional Languages*. Cambridge Univ. Press, 1992.
34. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.
35. J. Sifakis. A unified approach for studying the properties of transition systems. *TCS*, 18:227–258, 1982.
36. Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. *TOPLAS*, 29(3), 2007.
37. Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, 2003.

A Appendix

In this section, we give correctness proofs for our generated primitives for symbolic evaluation, $\mathcal{WLP}$, and symbolic composition. These apply to the language PL (§3.2) and reinterpretations given in §4; the proofs for MC differ only slightly.

As a notational convenience, we do not distinguish between a *State* and a *LogicalStruct*. A *LogicalStruct* ι corresponds to the *State*: $((\iota{\uparrow}1), (\iota{\uparrow}2)F_\rho)$. Because, for PL, logical structures only contain the single function F_ρ, there is a one-to-one correspondence with states. Hence, whenever necessary (e.g. in the applications of $\mathcal{E}[\![.]\!]$, $\mathcal{B}[\![.]\!]$, and $\mathcal{I}[\![.]\!]$), we assume that that a *LogicalStruct* ι is coerced to $((\iota{\uparrow}1), (\iota{\uparrow}2)F_\rho)$.

A.1 Correctness of the Symbolic-Evaluation Primitive

Lemma 1 (Relationship of $\overline{\mathcal{E}}$ to $\mathcal{E}$ and $\overline{\mathcal{B}}$ to $\mathcal{B}$).

$$(1)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!]U]\!]\iota = \mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)$$
$$(2)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![BE]\!]U]\!]\iota = \mathcal{B}[\![BE]\!](\mathcal{U}[\![U]\!]\iota)$$

Proof. The two lemmas are simultaneously proved using structural induction on E and BE, as shown below. Let U be $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$.

Note that the standard interpretations of *binop*, *relop*, and *boolop* coincide with those of $binop_L$, $relop_L$, and $boolop_L$. Thus, reasoning steps of the form $binop_L(op2_L) \rightsquigarrow binop(op2)$ are shorthands for reasoning about each case, such as $binop_L(\boxed{+}) \rightsquigarrow binop(+)$, etc.

(1) (*i*)
$$\begin{aligned}\mathcal{T}[\![\overline{\mathcal{E}}[\![c]\!]U]\!]\iota &= \mathcal{T}[\![\overline{const}(c)]\!]\iota\\ &= \mathcal{T}[\![c]\!]\iota\\ &= const(c)\\ &= \mathcal{E}[\![c]\!](\mathcal{U}[\![U]\!]\iota)\end{aligned}$$

(*ii*)
$$\begin{aligned}&\text{lhs}: \mathcal{T}[\![\overline{\mathcal{E}}[\![I]\!]U]\!]\iota\\ &\quad= \mathcal{T}[\![\overline{lookupState}\ U\ I]\!]\iota\\ &\quad= \mathcal{T}[\![((U{\uparrow}2)F_\rho)((U{\uparrow}1)I)]\!]\iota\\ &\text{rhs}: \mathcal{E}[\![I]\!](\mathcal{U}[\![U]\!]\iota)\\ &\quad= \mathcal{E}[\![I]\!]\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)\end{pmatrix}\\ &\quad= lookupState\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)I\end{pmatrix}\\ &\quad= ((\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)\ (\mathcal{T}[\![(U{\uparrow}1)I]\!]\iota)\\ &\quad= (\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota)\ (\mathcal{T}[\![(U{\uparrow}1)I]\!]\iota)\\ &\quad= access(\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota, \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota)\\ &\quad= \mathcal{T}[\![((U{\uparrow}2)F_\rho)((U{\uparrow}1)I)]\!]\iota\end{aligned}$$

(*iii*)
$$\begin{aligned}&\text{lhs}: \mathcal{T}[\![\overline{\mathcal{E}}[\![\&I]\!]U]\!]\iota = \mathcal{T}[\![\overline{lookupEnv}\ U\ I]\!]\iota = \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota\\ &\text{rhs}: \mathcal{E}[\![\&I]\!](\mathcal{U}[\![U]\!]\iota)\\ &\quad= \mathcal{E}[\![\&I]\!]\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)\end{pmatrix}\\ &\quad= lookupEnv\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)I\end{pmatrix}\\ &\quad= \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota\end{aligned}$$

(*iv*)
$$\begin{aligned}&\text{lhs}:\\ &\quad= \mathcal{T}[\![\overline{\mathcal{E}}[\![*E]\!]U]\!]\iota\\ &\quad= \mathcal{T}[\![\overline{lookupStore}\ U\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota\\ &\quad= \mathcal{T}[\![((U{\uparrow}2)F_\rho)(\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota\\ &\text{rhs}: \mathcal{E}[\![*E]\!](\mathcal{U}[\![U]\!]\iota)\\ &\quad= \mathcal{E}[\![*E]\!]\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)\end{pmatrix}\\ &\quad= lookupStore\begin{pmatrix}(\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![(U{\uparrow}1)I_i]\!]\iota],\\ (\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![(U{\uparrow}2)F_j]\!]\iota)(\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota))\end{pmatrix}\\ &\quad= (\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota)\ (\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota))\\ &\quad= (\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota)\ (\mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!]U]\!]\iota)\ //\ \text{by ind. via (1)}\\ &\quad= access(\mathcal{FE}[\![(U{\uparrow}2)F_\rho]\!]\iota, \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!]U]\!]\iota)\\ &\quad= \mathcal{T}[\![((U{\uparrow}2)F_\rho)(\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota\end{aligned}$$

(*v*)
$$\begin{aligned}&\mathcal{T}[\![\overline{\mathcal{E}}[\![E_1\ op2\ E_2]\!]U]\!]\iota\\ &\quad= \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U\ op2_L\ \overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota\\ &\quad= \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota\ binop_L(op2_L)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota\\ &\quad= \mathcal{E}[\![E_1]\!](\mathcal{U}[\![U]\!]\iota)\ binop(op2)\ \mathcal{E}[\![E_2]\!](\mathcal{U}[\![U]\!]\iota)\\ &\qquad\qquad //\ \text{by ind. via (1)}\\ &\quad= \mathcal{E}[\![E_1\ op2\ E_2]\!](\mathcal{U}[\![U]\!]\iota)\end{aligned}$$

(*vi*)
$$\begin{aligned}&\mathcal{T}[\![\overline{\mathcal{E}}[\![BE\ ?\ E_1 : E_2]\!]U]\!]\iota\\ &\quad= \mathcal{T}[\![ite(\overline{\mathcal{B}}[\![BE]\!]U, \overline{\mathcal{E}}[\![E_1]\!]U, \overline{\mathcal{E}}[\![E_2]\!]U)]\!]\iota\\ &\quad= cond_L(\mathcal{F}[\![\overline{\mathcal{B}}[\![BE]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota)\\ &\quad= \mathcal{F}[\![\overline{\mathcal{B}}[\![BE]\!]U]\!]\iota\ ?\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota : \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota\\ &\quad= \mathcal{B}[\![BE]\!](\mathcal{U}[\![U]\!]\iota)\ ?\ \mathcal{E}[\![E_1]\!](\mathcal{U}[\![U]\!]\iota) : \mathcal{E}[\![E_2]\!](\mathcal{U}[\![U]\!]\iota)\\ &\qquad\qquad //\ \text{by ind. via (1) and (2)}\\ &\quad= \mathcal{E}[\![BE\ ?\ E_1 : E_2]\!](\mathcal{U}[\![U]\!]\iota)\end{aligned}$$

(2) (*i*) $\mathcal{F}[\![\overline{\mathcal{B}}[\![\mathbb{T}]\!]U]\!]\iota = \mathcal{F}[\![\mathbb{T}]\!]\iota = \mathbb{T} = \mathcal{B}[\![\mathbb{T}]\!](\mathcal{U}[\![U]\!]\iota)$

(*ii*) $\mathcal{F}[\![\overline{\mathcal{B}}[\![\mathbb{F}]\!]U]\!]\iota = \mathcal{F}[\![\mathbb{F}]\!]\iota = \mathbb{F} = \mathcal{B}[\![\mathbb{F}]\!](\mathcal{U}[\![U]\!]\iota)$

(*iii*)
$$\begin{aligned}&\mathcal{F}[\![\overline{\mathcal{B}}[\![E_1\ rop\ E_2]\!]U]\!]\iota\\ &\quad= \mathcal{F}[\![\overline{\mathcal{E}}[\![E_1]\!]U\ rop_L\ \overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota\\ &\quad= \mathcal{T}[\![\overline{\mathcal{E}}[\![E_1]\!]U]\!]\iota\ relop_L(rop_L)\ \mathcal{T}[\![\overline{\mathcal{E}}[\![E_2]\!]U]\!]\iota\\ &\quad= \mathcal{E}[\![E_1]\!](\mathcal{U}[\![U]\!]\iota)\ relop(rop)\ \mathcal{E}[\![E_2]\!](\mathcal{U}[\![U]\!]\iota)\\ &\qquad\qquad //\ \text{by ind. via (1)}\\ &\quad= \mathcal{B}[\![E_1\ rop\ E_2]\!](\mathcal{U}[\![U]\!]\iota)\end{aligned}$$

(iv) $\mathcal{F}[\![\overline{\mathcal{B}}[\![\neg BE_1]\!]U]\!]\iota$
$= \mathcal{F}[\![\boxed{\neg}\,\overline{\mathcal{B}}[\![BE_1]\!]U]\!]\iota$
$= \neg\mathcal{F}[\![\overline{\mathcal{B}}[\![BE_1]\!]U]\!]\iota$
$= \neg\mathcal{B}[\![BE_1]\!](\mathcal{U}[\![U]\!]\iota)$ // by ind. via (2)
$= \mathcal{B}[\![\neg BE_1]\!](\mathcal{U}[\![U]\!]\iota)$

(v) $\mathcal{F}[\![\overline{\mathcal{B}}[\![BE_1\ bop\ BE_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\overline{\mathcal{B}}[\![BE_1]\!]U\ bop_L\ \overline{\mathcal{B}}[\![BE_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\overline{\mathcal{B}}[\![BE_1]\!]U]\!]\iota\ boolop_L(bop_L)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![BE_2]\!]U]\!]\iota$
$= \mathcal{B}[\![BE_1]\!](\mathcal{U}[\![U]\!]\iota)\ boolop(bop)\ \mathcal{B}[\![BE_2]\!](\mathcal{U}[\![U]\!]\iota)$
// by ind. via (2)
$= \mathcal{B}[\![BE_1\ bop\ BE_2]\!](\mathcal{U}[\![U]\!]\iota)$

Theorem 1. *For all $s \in Stmt$, $U \in StructUpdate$, and $\iota \in LogicalStruct$, the meaning of $\overline{\mathcal{I}}[\![s]\!]U$ in ι (i.e., $\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota$) is equivalent to running $\mathcal{I}$ on s with an input state obtained from $\mathcal{U}[\![U]\!]\iota$. That is,*

$$\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U]\!]\iota = \mathcal{I}[\![s]\!](\mathcal{U}[\![U]\!]\iota).$$

Proof.

(i) $\mathcal{U}[\![\overline{\mathcal{I}}[\![I = E;]\!]U]\!]\iota$
$= \mathcal{U}[\![\overline{updateStore}\ U\ (\overline{lookupEnv}\ U\ I)\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota$
$= \mathcal{U}[\![\overline{updateStore}\ U\ ((U{\uparrow}1)I)\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota$
$= \mathcal{U}\left[\!\!\left[\begin{pmatrix}(U{\uparrow}1),\\ (U{\uparrow}2)[F_\rho \mapsto ((U{\uparrow}2)F_\rho)[(U{\uparrow}1)I \mapsto \overline{\mathcal{E}}[\![E]\!]U]])\end{pmatrix}\right]\!\!\right]\iota$
$= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[\mathcal{T}[\![(\mathcal{U}[\![U]\!]\iota)I]\!] \mapsto \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\!]]\end{pmatrix}$
$= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[(\mathcal{U}[\![U]\!]\iota{\uparrow}1)I \mapsto \mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\end{pmatrix}$
// by Lem. 1(1)
$= updateStore\ (\mathcal{U}[\![U]\!]\iota)$
$(lookupEnv\ (\mathcal{U}[\![U]\!]\iota)\ I)$
$(\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota))$
$= \mathcal{I}[\![I = E;]\!](\mathcal{U}[\![U]\!]\iota)$

(ii) $\mathcal{U}[\![\overline{\mathcal{I}}[\![*I = E;]\!]U]\!]\iota$
$= \mathcal{U}[\![\overline{updateStore}\ U\ (\overline{\mathcal{E}}[\![I]\!]U)\ (\overline{\mathcal{E}}[\![E]\!]U)]\!]\iota$
$= \mathcal{U}\left[\!\!\left[\begin{pmatrix}(U{\uparrow}1),\\ (U{\uparrow}2)[F_\rho \mapsto ((U{\uparrow}2)F_\rho)[\overline{\mathcal{E}}[\![I]\!]U \mapsto \overline{\mathcal{E}}[\![E]\!]U]]\end{pmatrix}\right]\!\!\right]\iota$
$= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[\mathcal{T}[\![\overline{\mathcal{E}}[\![I]\!](\mathcal{U}[\![U]\!]\iota)]\!] \mapsto \mathcal{T}[\![\overline{\mathcal{E}}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\!]]\end{pmatrix}$
$= \begin{pmatrix}(\mathcal{U}[\![U]\!]\iota{\uparrow}1),\\ (\mathcal{U}[\![U]\!]\iota{\uparrow}2)[\mathcal{E}[\![I]\!](\mathcal{U}[\![U]\!]\iota) \mapsto \mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota)]\end{pmatrix}$
// by Lem. 1(1)
$= updateStore\ (\mathcal{U}[\![U]\!]\iota)\ (\mathcal{E}[\![I]\!](\mathcal{U}[\![U]\!]\iota))\ (\mathcal{E}[\![E]\!](\mathcal{U}[\![U]\!]\iota))$
$= \mathcal{I}[\![*I = E;]\!](\mathcal{U}[\![U]\!]\iota)$

(iii) $(\mathcal{U}[\![\overline{\mathcal{I}}[\![S_1S_2]\!]U]\!]\iota)$
$= (\mathcal{U}[\![\overline{\mathcal{I}}[\![S_2]\!](\overline{\mathcal{I}}[\![S_1]\!]U)]\!]\iota)$
$= \mathcal{I}[\![S_2]\!](\mathcal{U}[\![\overline{\mathcal{I}}[\![S_1]\!]U]\!]\iota)$ // by induction
$= \mathcal{I}[\![S_2]\!](\mathcal{I}[\![S_1]\!](\mathcal{U}[\![U]\!]\iota))$ // by induction
$= \mathcal{I}[\![S_1S_2]\!](\mathcal{U}[\![U]\!]\iota)$

A.2 Correctness of $\mathcal{WLP}$

Lemma 2 (Relationship of $\overline{\mathcal{T}}$ to $\mathcal{T}$, $\overline{\mathcal{F}}$ to $\mathcal{F}$, $\overline{\mathcal{FE}}$ to $\mathcal{FE}$).

(1) $\mathcal{T}[\![\overline{\mathcal{T}}[\![T]\!]U]\!]\iota = \mathcal{T}[\![T]\!](\mathcal{U}[\![U]\!]\iota)$
(2) $\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota = \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![U]\!]\iota)$
(3) $\mathcal{FE}[\![\overline{\mathcal{FE}}[\![FE]\!]U]\!]\iota = \mathcal{FE}[\![FE]\!](\mathcal{U}[\![U]\!]\iota)$

Proof. The three lemmas are simultaneously proved using structural induction on T, φ, and FE, as shown below. Let U be $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$. (Thus, $T_i = (U{\uparrow}1)I_i$ and $FE_j = (U{\uparrow}2)F_j$.) Let f be $(\iota{\uparrow}2)[F_j \mapsto \mathcal{FE}[\![FE_j]\!]\iota]$.

(1) (i) $\mathcal{T}[\![\overline{\mathcal{T}}[\![c]\!]U]\!]\iota = \mathcal{T}[\![c]\!]\iota = const(c) = \mathcal{T}[\![c]\!](\mathcal{U}[\![U]\!]\iota)$

(ii)
lhs $= \mathcal{T}[\![\overline{\mathcal{T}}[\![I]\!]U]\!]\iota = \mathcal{T}[\![\overline{lookupId}\ U\ I]\!]\iota = \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota$
rhs $= \mathcal{T}[\![I]\!](\mathcal{U}[\![U]\!]\iota) = \mathcal{T}[\![I]\!]((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)$
$= lookupId\ ((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)\ I$
$= \mathcal{T}[\![(U{\uparrow}1)I]\!]\iota$

(iii) $\mathcal{T}[\![\overline{\mathcal{T}}[\![T_1\ op2_L\ T_2]\!]U]\!]\iota$
$= \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U\ op2_L\ \overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$
$= \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota\ binop_L(op2_L)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$
$= \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota)\ binop_L(op2_L)\ \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)$
// by ind. via (1)
$= \mathcal{T}[\![T_1\ op2_L\ T_2]\!](\mathcal{U}[\![U]\!]\iota)$

(iv) $\mathcal{T}[\![\overline{\mathcal{T}}[\![ite(\varphi, T_1, T_2)]\!]U]\!]\iota$
$= \mathcal{T}[\![ite(\overline{\mathcal{F}}[\![\varphi]\!]U, \overline{\mathcal{T}}[\![T_1]\!]U, \overline{\mathcal{T}}[\![T_2]\!]U)]\!]\iota$
$= cond_L(\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota)$
$= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota\ ?\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota : \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![U]\!]\iota)\ ?\ \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota) : \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)$
// by ind. via (1) and (2)
$= \mathcal{F}[\![\varphi\ ?\ T_1 : T_2]\!](\mathcal{U}[\![U]\!]\iota)$

(v) $\mathcal{T}[\![\overline{\mathcal{T}}[\![FE(T)]\!]U]\!]\iota$
$= \mathcal{T}[\![\overline{\mathcal{FE}}[\![FE]\!]U(\overline{\mathcal{T}}[\![T]\!]U)]\!]\iota$
$= (\mathcal{FE}[\![\overline{\mathcal{FE}}[\![FE]\!]U]\!]\iota)(\mathcal{T}[\![\overline{\mathcal{T}}[\![T]\!]U]\!]\iota)$
$= (\mathcal{FE}[\![FE]\!](\mathcal{U}[\![U]\!]\iota))(\mathcal{T}[\![T]\!](\mathcal{U}[\![U]\!]\iota))$
// by ind. via (3)
$= \mathcal{T}[\![FE(T)]\!](\mathcal{U}[\![U]\!]\iota)$

(2) (i) $\mathcal{F}[\![\overline{\mathcal{F}}[\![\boxed{\mathbb{T}}]\!]U]\!]\iota = \mathcal{F}[\![\boxed{\mathbb{T}}]\!]\iota = \mathbb{T} = \mathcal{F}[\![\boxed{\mathbb{T}}]\!](\mathcal{U}[\![U]\!]\iota)$

(ii) $\mathcal{F}[\![\overline{\mathcal{F}}[\![\boxed{\mathbb{F}}]\!]U]\!]\iota = \mathcal{F}[\![\boxed{\mathbb{F}}]\!]\iota = \mathbb{F} = \mathcal{F}[\![\boxed{\mathbb{F}}]\!](\mathcal{U}[\![U]\!]\iota)$

(iii) $\mathcal{F}[\![\overline{\mathcal{F}}[\![T_1\ rop_L\ T_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\overline{\mathcal{T}}[\![T_1]\!]U\ relop_L(rop_L)\ \overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$
$= \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota\ relop_L(rop_L)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota$
$= \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota)\ relop_L(rop_L)\ \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)$
// by ind. via (1)
$= \mathcal{F}[\![T_1\ rop_L\ T_2]\!](\mathcal{U}[\![U]\!]\iota)$

(iv) $\mathcal{F}[\![\overline{\mathcal{F}}[\![\boxed{\neg}\,\varphi_1]\!]U]\!]\iota$
$= \mathcal{F}[\![\boxed{\neg}\,\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota$
$= \neg\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota$
$= \neg\mathcal{F}[\![\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)$ // by ind. via (2)
$= \mathcal{F}[\![\boxed{\neg}\,\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)$

(v) $\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1\ bop_L\ \varphi_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U\ boolop_L(bop_L)\ \overline{\mathcal{F}}[\![\varphi_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota\ boolop_L(bop_L)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_2]\!]U]\!]\iota$
$= \mathcal{F}[\![\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)\ boolop_L(bop_L)\ \mathcal{F}[\![\varphi_2]\!](\mathcal{U}[\![U]\!]\iota)$
// by ind. via (2)
$= \mathcal{F}[\![\varphi_1\ bop_L\ \varphi_2]\!](\mathcal{U}[\![U]\!]\iota)$

(3) (i)
lhs $= \mathcal{FE}[\![\overline{\mathcal{FE}}[\![F]\!]U]\!]\iota$
$= \mathcal{FE}[\![\overline{lookupId}\ U\ F]\!]\iota$
$= \mathcal{FE}[\![(U{\uparrow}2)F]\!]\iota$
rhs $= \mathcal{FE}[\![F]\!](\mathcal{U}[\![U]\!]\iota)$
$= \mathcal{FE}[\![F]\!]((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)$
$= lookupFuncId\ ((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)\ F$
$= \mathcal{FE}[\![(U{\uparrow}2)F]\!]\iota$

(ii) $\mathcal{FE}[\![\overline{\mathcal{FE}}[\![FE_0[T_1 \mapsto T_2]]\!]U]\!]\iota$
$= \mathcal{FE}[\![(\overline{\mathcal{FE}}[\![FE_0]\!]U)[\overline{\mathcal{T}}[\![T_1]\!]U \mapsto \overline{\mathcal{T}}[\![T_2]\!]U]]\!]\iota$
$= \mathcal{FE}[\![(\overline{\mathcal{FE}}[\![FE_0]\!]U)]\!]\iota[\mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota \mapsto \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota]$
$= \mathcal{FE}[\![FE_0]\!](\mathcal{U}[\![U]\!]\iota)[\mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota) \mapsto \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)]$
// by ind. via (1)
$= \mathcal{FE}[\![FE_0[T_1 \mapsto T_2]]\!](\mathcal{U}[\![U]\!]\iota)$

Theorem 2. *For any Stmt s and Formula φ, $\psi := \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$ is an acceptable $\mathcal{WLP}$ formula for φ with respect to s.*

Proof. For all $\iota \in LogicalStruct$,

$$\begin{aligned}\mathcal{F}[\![\psi]\!]\iota &= \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})]\!]\iota \\ &= \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![\overline{\mathcal{I}}[\![s]\!]U_{id}]\!]\iota) \quad \text{// by Lem. 2} \\ &= \mathcal{F}[\![\varphi]\!](\mathcal{I}[\![s]\!](\mathcal{U}[\![U_{id}]\!]\iota)) \quad \text{// by Thm. 1} \\ &= \mathcal{F}[\![\varphi]\!](\mathcal{I}[\![s]\!]\iota)\end{aligned}$$

and therefore, by Defn. 1, $\overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$ is an acceptable $\mathcal{WLP}$ formula for φ with respect to s.

A.3 Correctness of the Symbolic-Composition Primitive

We now show that the meaning of $\overline{\mathcal{U}}[\![U_2]\!]U_1$ is the composition of the meanings of U_2 and U_1.

Theorem 3. *For all $U_1, U_2 \in StructUpdate$,*

$$\mathcal{U}[\![\overline{\mathcal{U}}[\![U_2]\!]U_1]\!] = \mathcal{U}[\![U_2]\!] \circ \mathcal{U}[\![U_1]\!].$$

Proof. Let $U_2 = (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow FE_j\})$; let I_k and F_m range over Id and $FuncId$, respectively; and let $\iota \in LogicalStruct$ be an arbitrary logical structure.

$$\begin{aligned}&\mathcal{U}[\![\overline{\mathcal{U}}[\![U_2]\!]U_1]\!]\iota \\ &= \mathcal{U}\left[\!\!\left[\begin{pmatrix}(U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U_1], \\ (U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U_1]\end{pmatrix}\right]\!\!\right]\iota \\ &= \mathcal{U}\left[\!\!\left[\begin{pmatrix}\{I_k \mapsto ((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U_1])I_k\}, \\ \{F_m \mapsto ((U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U_1])F_m\}\end{pmatrix}\right]\!\!\right]\iota \\ &= \begin{pmatrix}(\iota{\uparrow}1)[I_k \mapsto \mathcal{T}[\![((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}[\![T_i]\!]U_1])I_k]\!]\iota], \\ (\iota{\uparrow}2)[F_m \mapsto \mathcal{FE}[\![((U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}[\![FE_j]\!]U_1])F_m]\!]\iota]\end{pmatrix} \\ &= \begin{pmatrix}(\iota{\uparrow}1)[I_{k(\neq i)} \mapsto \mathcal{T}[\![(U_1{\uparrow}1)I_k]\!]\iota][I_i \mapsto \mathcal{T}[\![\overline{\mathcal{T}}[\![T_i]\!]U_1]\!]\iota], \\ (\iota{\uparrow}2)[F_{m(\neq j)} \mapsto \mathcal{FE}[\![(U_1{\uparrow}2)F_m]\!]\iota][F_j \mapsto \mathcal{FE}[\![\overline{\mathcal{FE}}[\![FE_j]\!]U_1]]\!]\iota\end{pmatrix} \\ &= \begin{pmatrix}(\iota{\uparrow}1)[I_{k(\neq i)} \mapsto \mathcal{T}[\![(U_1{\uparrow}1)I_k]\!]\iota][I_i \mapsto \mathcal{T}[\![T_i]\!](\mathcal{U}[\![U_1]\!]\iota)], \\ (\iota{\uparrow}2)[F_{m(\neq j)} \mapsto \mathcal{FE}[\![(U_1{\uparrow}2)F_m]\!]\iota][F_j \mapsto \mathcal{FE}[\![FE_j]\!](\mathcal{U}[\![U_1]\!]\iota)]\end{pmatrix} \\ &\qquad \text{// by Lem. 2} \\ &= \begin{pmatrix}((\mathcal{U}[\![U_1]\!]\iota){\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!](\mathcal{U}[\![U_1]\!]\iota)], \\ ((\mathcal{U}[\![U_1]\!]\iota){\uparrow}2)[F_j \mapsto \mathcal{FE}[\![FE_j]\!](\mathcal{U}[\![U_1]\!]\iota)]\end{pmatrix} \\ &= \mathcal{U}[\![U_2]\!](\mathcal{U}[\![U_1]\!]\iota)\end{aligned}$$