Symbolic Analysis via Semantic Reinterpretation*

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

[1] University of Wisconsin; Madison, WI; USA. {junghee,akash,reps}@cs.wisc.edu
[2] GrammaTech, Inc.; Ithaca, NY; USA.

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 x86 and PowerPC.

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], as well as hybrid concrete/symbolic program-exploration tools, such as DART [6], CUTE [13], SAGE [7], BITSCOPE [3], and DASH [2] use forward symbolic evaluation, $\mathcal{WLP}$, or both. Symbolic evaluation is used to create path formulas. To determine whether a path π is executable, an SMT solver is used to determine whether π's path formula is satisfiable, and if so, to generate an input that drives the program down π. $\mathcal{WLP}$ is used to identify new predicates that split part of a program's state space [1, 2].
- Bug-finding tools, such as ARCHER [15] and SATURN [14], 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

* Supported by NSF under grants CCF-{0540955, 0524051, 0810053}, by AFRL under contract FA8750-06-C-0249, and by ONR under grant N00014-09-1-0510.
** Supported by a Symantec Research Labs Graduate Fellowship.
*** Supported by a Microsoft Research Fellowship.

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 $[entry_P, \ldots, r]$.

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 and Z3, 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 [11] requires taking into account all possible aliasing combinations.

The standard approach to implementing each of the symbolic-analysis primitives for a programming language of interest (which we call the *subject* language) is to create hand-written *translation procedures*—one per symbolic-analysis primitive—that convert subject-language commands into appropriate formulas. With this approach, a system can contain subtle inconsistency bugs if the different translation procedures adopt different "views" of the semantics. 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. One manifestation of an inconsistency bug would be that if one performs symbolic execution 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.

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 Section 3.2 of the Intel manual (http://download.intel.com/design/processor/manuals/253666.pdf), and imagine writing three separate encodings of each instruction's semantics to implement symbolic evaluation, $\mathcal{WLP}$, and symbolic composition. Some tools (e.g., [7, 3]) need an instruction-set emulator, in which case a fourth encoding of the semantics is also required.

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. More precisely, 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, 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 [4], where semantic reinterpretation has been proposed as a convenient tool for formulating abstract interpretations [12, 10] (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 (§7). 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 [9], as the implementation platform for defining the necessary reinterpretations. However, we wish to stress that the 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 [11] (§4).
– We conducted an experiment on real x86 code using the generated primitives (§7).

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 (§7) 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 our implementation and the experiment carried out with it. §8 discusses related work.

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 [10].] 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):

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

$$\begin{array}{l} \mathcal{E} : \mathit{Expr} \to \mathit{State} \to \mathit{Int32} \\ \mathcal{E}[\![I]\!]\sigma = \sigma I \\ \mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma \oplus \mathcal{E}[\![E_2]\!]\sigma \end{array} \qquad \begin{array}{l} \mathcal{I} : \mathit{Stmt} \to \mathit{State} \to \mathit{State} \\ \mathcal{I}[\![I = E;]\!]\sigma = \sigma[I \mapsto \mathcal{E}[\![E]\!]\sigma] \end{array}$$

By "$\sigma[I \mapsto v]$," we mean the function that acts like σ except that argument I is mapped to v. The specification given above can be factored into client and core specifications by introducing a domain *Val*, as well as operators *xor*, *lookup*, and *store*. The client specification is defined by

$$\mathit{xor} : \mathit{Val} \to \mathit{Val} \to \mathit{Val} \quad \mathit{lookup} : \mathit{State} \to \mathit{Id} \to \mathit{Val} \quad \mathit{store} : \mathit{State} \to \mathit{Id} \to \mathit{Val} \to \mathit{State}$$

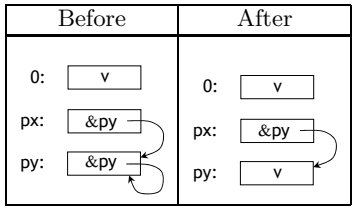
$$\begin{array}{l} \mathcal{E} : \mathit{Expr} \to \mathit{State} \to \mathit{Val} \\ \mathcal{E}[\![I]\!]\sigma = \mathit{lookup}\ \sigma\ I \\ \mathcal{E}[\![E_1 \oplus E_2]\!]\sigma = \mathcal{E}[\![E_1]\!]\sigma\ \mathit{xor}\ \mathcal{E}[\![E_2]\!]\sigma \end{array} \qquad \begin{array}{l} \mathcal{I}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

For the concrete (or "standard") semantics, the semantic core is defined by

$$\begin{array}{lll} v \in \mathit{Val}_{std} = \mathit{Int32} & \mathit{lookup}_{std} = \lambda\sigma.\lambda I.\sigma I & \mathit{xor}_{std} = \lambda v_1.\lambda v_2.v_1 \oplus v_2 \\ \mathit{State}_{std} = \mathit{Id} \to \mathit{Val} & \mathit{store}_{std} = \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v] & \end{array}$$

(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 corresponding to (a).

Different abstract interpretations can be defined by using the same client semantics, but giving a different interpretation of the base types, function types, and operators of the core. For example, for sign analysis, assuming that *Int32* values are represented in two's complement, the semantic core is reinterpreted as follows:[3]

$$
\begin{aligned}
v \in Val_{abs} &= \{neg, zero, pos\}^{\top} \\
State_{abs} &= Id \to Val_{abs} \\
lookup_{abs} &= \lambda\sigma.\lambda I.\sigma I \\
store_{abs} &= \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]
\end{aligned}
\qquad
xor_{abs} = \lambda v_1.\lambda v_2.
$$

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

For the code fragment shown in Fig. 1(a), which swaps two `int`s, sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$, produce the following abstract states:

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

Semantic Reinterpretation Versus Standard Abstract Interpretation. Semantic reinterpretation [12, 10] is a form of abstract interpretation [4], 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 $N + M$

[3] 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 $zero \sqcup pos = \top$.

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 a client specification to the operations of the semantic core that one defines in a semantic reinterpretation 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—is 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 can 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 interpretation, 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 §7); 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

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.

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

$$
\begin{aligned}
\mathit{const} &: C_{\mathit{Int32}} \to \mathit{Val} \\
\mathit{cond}_L &: \mathit{BVal} \to \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\
\mathit{lookupId} &: \mathit{LogicalStruct} \to \mathit{Id} \to \mathit{Val} \\
\mathit{binop}_L &: \mathit{BinOp}_L \to (\mathit{Val} \times \mathit{Val} \to \mathit{Val}) \\
\mathit{relop}_L &: \mathit{RelOp}_L \to (\mathit{Val} \times \mathit{Val} \to \mathit{BVal}) \\
\mathit{boolop}_L &: \mathit{BoolOp}_L \to (\mathit{BVal} \times \mathit{BVal} \to \mathit{BVal}) \\
\mathit{lookupFuncId} &: \mathit{LogicalStruct} \to \mathit{FuncId} \to (\mathit{Val} \to \mathit{Val}) \\
\mathit{access} &: (\mathit{Val} \to \mathit{Val}) \times \mathit{Val}) \to \mathit{Val} \\
\mathit{update} &: ((\mathit{Val} \to \mathit{Val}) \times \mathit{Val} \times \mathit{Val}) \to (\mathit{Val} \to \mathit{Val})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T} &: \mathit{Term} \to \mathit{LogicalStruct} \to \mathit{Val} \\
\mathcal{T}[\![c]\!]\iota &= \mathit{const}(c) \\
\mathcal{T}[\![I]\!]\iota &= \mathit{lookupId}\ \iota\ I \\
\mathcal{T}[\![T_1\ \mathit{op2}_L\ T_2]\!]\iota &= \mathcal{T}[\![T_1]\!]\iota\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![T_2]\!]\iota \\
\mathcal{T}[\![\mathit{ite}(\varphi, T_1, T_2)]\!]\iota &= \mathit{cond}_L(\mathcal{F}[\![\varphi]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota) \\
\mathcal{T}[\![\mathit{FE}(T_1)]\!]\iota &= \mathit{access}(\mathcal{FE}[\![\mathit{FE}]\!]\iota, \mathcal{T}[\![T_1]\!]\iota)
\end{aligned}
\qquad
\begin{aligned}
\mathcal{F} &: \mathit{Formula} \to \mathit{LogicalStruct} \to \mathit{BVal} \\
\mathcal{F}[\![\boxed{\mathbb{T}}]\!]\iota &= \mathbb{T} \\
\mathcal{F}[\![\boxed{\mathbb{F}}]\!]\iota &= \mathbb{F} \\
\mathcal{F}[\![T_1\ \mathit{rop}_L\ T_2]\!]\iota &= \mathcal{T}[\![T_1]\!]\iota\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![T_2]\!]\iota \\
\mathcal{F}[\![\boxed{\neg}\,\varphi_1]\!]\iota &= \neg\mathcal{F}[\![\varphi_1]\!]\iota \\
\mathcal{F}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!]\iota &= \mathcal{F}[\![\varphi_1]\!]\iota\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\varphi_2]\!]\iota
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{FE} &: \mathit{FuncExpr} \to \mathit{LogicalStruct} \to (\mathit{Val} \to \mathit{Val}) \\
\mathcal{FE}[\![F]\!]\iota &= \mathit{lookupFuncId}\ \iota\ F \\
\mathcal{FE}[\![\mathit{FE}_1[T_1 \mapsto T_2]]\!]\iota &= \mathit{update}(\mathcal{FE}[\![\mathit{FE}_1]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota)
\end{aligned}
$$

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

Fig. 2. The factored semantics of L.

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

$$I \in \mathit{Id}, T \in \mathit{Term}, \varphi \in \mathit{Formula}, F \in \mathit{FuncId}, \mathit{FE} \in \mathit{FuncExpr}, U \in \mathit{StructUpdate}$$

$$
\begin{aligned}
T &::= c \mid I \mid T_1\ \mathit{op2}_L\ T_2 \mid \mathit{ite}(\varphi, T_1, T_2) \mid \mathit{FE}(T) \\
\varphi &::= \boxed{\mathbb{T}} \mid \boxed{\mathbb{F}} \mid T_1\ \mathit{rop}_L\ T_2 \mid \boxed{\neg}\,\varphi_1 \mid \varphi_1\ \mathit{bop}_L\ \varphi_2
\end{aligned}
\qquad
\begin{aligned}
\mathit{FE} &::= F \mid \mathit{FE}_1[T_1 \mapsto T_2] \\
U &::= (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})
\end{aligned}
$$

A *Term* of the form $\mathit{ite}(\varphi, T_1, T_2)$ represents an if-then-else expression. A *FuncExpr* of the form $\mathit{FE}_1[T_1 \mapsto T_2]$ denotes a *function-update expression*. A *StructUpdate* of the form $(\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})$ is called a *structure-update expression*. 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: $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow \mathit{FE}_j\})$. $\{I'_i \hookleftarrow T_i\}$ specifies the updates to the interpretations of the constant symbols and $\{F'_j \hookleftarrow \mathit{FE}_j\}$ specifies the updates to the interpretations of the function symbols (see below). Thus, a structure-update expression $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow \mathit{FE}_j\})$ can be thought of as a kind of restricted 2-vocabulary (i.e., 2-state) formula $\bigwedge_i (I'_i = T_i) \wedge \bigwedge_j (F'_j = \mathit{FE}_j)$. We define U_{id} to be $(\{I' \hookleftarrow I \mid I \in \mathit{Id}\}, \{F' \hookleftarrow F \mid F \in \mathit{FuncId}\})$.

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.

$$\iota \in \mathit{LogicalStruct} = (\mathit{Id} \to \mathit{Val}) \times (\mathit{FuncId} \to (\mathit{Val} \to \mathit{Val}))$$

$(\iota{\uparrow}1)$ 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. 2. Motivated by the needs of later sections, we retain the convention from §2 of working with the domain *Val*

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

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

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

Fig. 3. The factored semantics of PL.

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., $v_1\ \mathit{binop}_L(\boxed{\oplus})\ v_2 = v_1\ \mathit{xor}\ v_2$, etc. The standard interpretations for $\mathit{lookupId}_{\mathit{std}}$ and $\mathit{lookupFuncId}_{\mathit{std}}$ select from the first and second components, respectively, of a *LogicalStruct*: $\mathit{lookupId}_{\mathit{std}}\ \iota\ I = (\iota{\uparrow}1)(I)$ and $\mathit{lookupFuncId}_{\mathit{std}}\ \iota\ F = (\iota{\uparrow}2)(F)$. 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., $I \in (\mathit{Id} - \{I_i\})$ and $F \in (\mathit{FuncId} - \{F_j\})$), 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 $I \in (\mathit{Id} - \{I_i\})$, and $(U{\uparrow}2)F = F$ for $F \in (\mathit{FuncId} - \{F_j\})$.

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).

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

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

Semantics of PL. The factored semantics of PL is presented in Fig. 3. The semantic domain *Loc* stands for *locations* (or memory addresses). We identify *Loc* with the set *Val* of values. A state $\sigma \in \mathit{State}$ is a pair (η, ρ), where, in the standard semantics, *environment* $\eta \in \mathit{Env} = \mathit{Id} \to \mathit{Loc}$ maps identifiers to their

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

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

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

Fig. 4. The factored semantics of MC.

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

$$
\begin{array}{rl}
BVal_{std} = BVal & cond_{std} = \lambda b.\lambda v_1.\lambda v_2.\ (b\,?\,v_1 : v_2) \\
Val_{std} = Int32 & lookupState_{std} = \lambda(\eta,\rho).\lambda I.\rho(\eta(I)) \\
Loc_{std} = Int32 & lookupEnv_{std} = \lambda(\eta,\rho).\lambda I.\eta(I) \\
\eta \in Env_{std} = Id \to Loc_{std} & lookupStore_{std} = \lambda(\eta,\rho).\lambda l.\rho(l) \\
\rho \in Store_{std} = Loc_{std} \to Val_{std} & updateStore_{std} = \lambda(\eta,\rho).\lambda l.\lambda v.(\eta, \rho[l \mapsto v])
\end{array}
$$

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 instructions.

$$
r \in register, do \in dst\_operand, so \in src\_operand, i \in instruction
$$

$$
\begin{array}{rl}
r ::= EAX \mid EBX \mid EBP \mid EIP & do ::= Indirect(r, Val) \mid DirectReg(r) \\
flagName ::= ZF & so ::= do \cup Immediate(Val) \\
\multicolumn{2}{c}{instruction ::= MOV(do, so) \mid CMP(do, so) \mid XOR(do, so) \mid JZ(do)}
\end{array}
$$

Semantics of MC. The factored semantics of MC is presented in Fig. 4. 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*). An MC state $\sigma \in State$ is a triple (*mem*, *reg*, *flag*), where *mem* is a map $Val \to Val$, *reg* is a map $register \to Val$, and *flag* is a map $flagName \to BVal$. 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 $\mathtt{x}$ 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 logic L[PL] as a reinterpretation domain for the semantic core of PL. The base types and the state type of the semantic core 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 [5]).
- $\overline{\mathit{cond}}$ 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:

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

By extension, this produces functions $\overline{\mathcal{E}}$, $\overline{\mathcal{B}}$, and $\overline{\mathcal{I}}$ with the following types:

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

Function $\overline{\mathcal{I}}$ translates a statement s of PL to a phrase in logic L[PL].

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

Fig. 5. Symbolic evaluation of Fig. 1(b) via semantic reinterpretation, starting with a *StructUpdate* that corresponds to the "Before" column of Fig. 1(c).

Example 2. The steps of symbolic evaluation of Fig. 1(b) via semantic reinterpretation, starting with a *StructUpdate* that corresponds to Fig. 1(c), are shown in Fig. 5. The *StructUpdate* U_4 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.

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:

- 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 [11]. 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 3. In L[PL], such a formula would be expressed as shown below on the right. (This formula will be created using semantic reinterpretation in Ex. 4.)

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

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 function(s)	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 $Val \to Val$. 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 $Val \to Val$ need to be replaced by *FVal*. The standard semantics of *FVal* is $Val \to Val$; 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* of the semantic core 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:

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

To understand how pointers are handled during the $\mathcal{WLP}$ operation, the key reinterpretations to concentrate on are the ones for the operations of the semantic core of L[PL] 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 below, where $\equiv$, $\neq$, and $\doteq$ denote *equality-as-terms*, *definite-disequality*, and *possible-equality*, respectively.

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

(The possible-equality tests, "$k_1 \doteq k_2$", are really "otherwise" cases of three-pronged comparisons.) The possible-equality case for $\overline{access}$ introduces *ite* terms. As illustrated in Ex. 4, 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 [11].

Example 4. We now demonstrate how semantic reinterpretation produces the L[PL] formula for $\mathcal{WLP}(*p = e, x = 5)$ claimed in Ex. 3.

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

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

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 $\mathtt{p}$ is the address of $\mathtt{x}$, which is the only aliasing condition that matters for this example.

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:

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

Example 5. For the swap-code fragment from Fig. 1(a), we can demonstrate the ability of $\overline{\mathcal{U}}$ to perform symbolic composition by showing that

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

First, consider the left-hand side. It is not hard to show that $\overline{\mathcal{I}}[\![s_1; s_2; s_3]\!]U_{id} = (\{\mathtt{x}' \hookleftarrow \mathtt{y}, \mathtt{y}' \hookleftarrow \mathtt{x}\}, \emptyset)$. Now consider the right-hand side. Let $U_{1,2}$ and U_3 be

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

We want to compute

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

Therefore, $\overline{\mathcal{I}}[\![s_1; s_2; s_3]\!]U_{id} = \overline{\mathcal{U}}[\![U_3]\!]U_{1,2}$.

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 $store_{mem}$.

```
[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. 6. (a) A simple source-code fragment written in PL; (b) the MC code for (a).

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.

Example 6. 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 execution of *swap*, produces the *StructUpdate*

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

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

Example 7. Fig. 6(a) shows a source-code fragment; Fig. 6(b) shows the corresponding MC code. (To simplify the MC code, source-level variable names are used.) In Fig. 6(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. 6(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. 6(b), and $c_{\texttt{[7]}}$ is the address of `ERROR`. Using semantic reinterpretation, $\overline{\mathcal{F}}_{\mathrm{MC}}[\![(EIP \boxed{=} c_{\texttt{[7]}})]\!](\overline{\mathcal{I}}_{\mathrm{MC}}[\![s_{\texttt{[1]-[5]}}]\!]U_{id})$ 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 branch is split across two instructions, $\mathcal{WLP}$ can be used to recover the branch condition. $\mathcal{WLP}(\texttt{cmp x,5; jz ERROR}, (EIP \boxed{=} c_{\texttt{[7]}}))$ returns the formula $ite(((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0), c_{\texttt{[7]}}, c_{\texttt{[6]}}) \boxed{=} c_{\texttt{[7]}}$ as follows:

$$\overline{\mathcal{I}}_{\mathrm{MC}}[\![\texttt{cmp x,5}]\!]U_{id} = (\{ZF' \hookleftarrow (F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0\}, \emptyset) = U_1$$
$$\overline{\mathcal{I}}_{\mathrm{MC}}[\![\texttt{jz ERROR}]\!]U_1 = (\{EIP' \hookleftarrow ite(((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0), c_{[7]}, c_{[6]})\}, \emptyset) = U_2$$
$$\overline{\mathcal{F}}_{\mathrm{MC}}[\![EIP \boxed{=} c_{[7]}]\!]U_2 = ite(((F_{mem}(\texttt{x}) \boxed{-} 5) \boxed{=} 0), c_{[7]}, c_{[6]}) \boxed{=} c_{[7]}$$

Because $c_{[7]} \neq c_{[6]}$, this 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 $\overline{\mathcal{U}}_{\mathrm{MC}}$.

6 Other Language Constructs

Branching. Ex. 7 illustrated a $\mathcal{WLP}$ computation across a branch. We now illustrate forward symbolic evaluation across a branch.

Suppose that an if-statement is represented by *IfStmt*(*BE*, *Int32*, *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:

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

In the reinterpretation for symbolic evaluation, the *StructUpdate U* obtained by $\mathcal{I}[\![IfStmt(BE, c_T, c_F)]\!]U_{id}$ would be $(\{PC' \hookleftarrow ite(\varphi_{BE}, c_T, c_F)\}, \emptyset)$, 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 $(ite(\varphi_{BE}, c_T, c_F) \boxed{=} c_T)$, which (assuming that $c_T \neq c_F$) simplifies to φ_{BE}. (A similar formula simplification was performed in Ex. 7 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 [6, 13, 7, 3]. 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. 7.

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

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:

```
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
    if (i_k is a branch instruction)
      φ = φ [&&] ℱ⟦PC = PC_{k+1}⟧U;    // Conjoin the branch condition for i_k
  }
  return φ;
}
```

Fig. 7. An algorithm to obtain a path-constraint formula that characterizes which initial states must follow path π.

- 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 Implementation and Evaluation

Implementation. Our implementation uses the TSL system [9]. (TSL stands for "**T**ransformer **S**pecification **L**anguage".) 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. For instance, the meaning function $\mathcal{I}$ of §3.3 is written as a TSL function

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

where `instruction` and `state` are user-defined datatypes that represent the syntactic objects (in this case, instructions) and the semantic states, respectively.

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 TSL specifications of x86 and PowerPC.)

TSL's meta-language provides 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 reinterpreting the TSL base-types, base-type operators, map-types, and map-type operators (i.e., *access* and *update*). When semantic reinterpretation is performed in this way, 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 act 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 following table 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.

	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

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 $\overline{\mathcal{I}}[\![\cdot]\!]$: $\mathcal{WLP}(s, \varphi) = \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$. $\mathcal{WLP}$ 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, SAGE [7] 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, φ can be unsatisfiable when the path is executable, or satisfiable when the path is not executable. Both situations are called a *divergence* [7]. 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.

However, if we eventually hope to model check x86 machine code, implementations of faithful symbolic techniques will be required. Using faithful symbolic techniques could raise the cost of performing symbolic operations because faithful path-constraint formulas could end up being 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 symbolic-evaluation 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 experiments used six C++ programs, each exercising a single algorithm from the C++ STL, compiled under Visual Studio 2005. We compared the two tools' divergence rates and running times (see Tab. 1). On average, the approximate version had 5.2X fewer constraints in φ, had a 79% divergence rate, and was about 2X faster than the faithful version; the faithful version reported no divergences.

8 Related Work

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

- Hybrid concrete/symbolic tools [6, 13, 7, 3] use a combination of concrete and symbolic evaluation to generate inputs that increase coverage.
- $\mathcal{WLP}$ can be used to create new predicates that split part of a program's abstract state space [1, 2].

Name (STL)	# Tests	|Trace| #instrs	# branch	Faithful CE	Faithful SE	Faithful SMT	Faithful $\|\varphi\|$	Faithful Div.	Approximate C+SE	Approximate SMT	Approximate $\|\varphi\|$	Approximate Div.	Approximate Dist.
search	18	770	28	0.26	8.68	0.26	**10.5**	**0%**	9.13	0.10	**4.8**	**61%**	55%
random_shuffle	48	1831	51	0.59	21.6	0.17	**27.3**	**0%**	21.9	0.03	**1.0**	**95%**	93%
copy	5	1987	57	0.69	55.0	0.15	**5.4**	**0%**	55.8	0.03	**1.0**	**60%**	57%
partition	13	2155	76	0.72	26.4	0.43	**35.2**	**0%**	27.4	0.02	**1.0**	**92%**	58%
max_element	101	2870	224	0.94	17.0	3.59	**153.0**	**0%**	18.0	2.90	**78.4**	**83%**	6%
transform	11	10880	476	4.22	720.8	1.12	**220.6**	**0%**	713.6	0.03	**1.0**	**82%**	89%

Table 1. Experimental results. We report 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 version, these were done in lock step and their total time is reported in (C+SE). (All times are in seconds.) For each version, we also report the average time taken by the SMT solver (Yices), the average number of constraints found ($|\varphi|$), and the divergence rate. For the approximate version, we also show the average distance (in % of the total length of the trace) before a diverging test diverged.

– Symbolic composition is useful when a tool has access to a formula that summarizes a called procedure's behavior [14]; 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 tools, our work has one key advantage: it creates the core concrete-execution and symbolic-analysis components in a way that ensures by construction that they are *mutually consistent*. We are not aware of existing tools in which the concrete-execution and symbolic-analysis primitives are implemented in a way that guarantees such a consistency property. For instance, in the source code for B2 [8] (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.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
2. N. Beckman, A. Nori, S. Rajamani, and R. Simmons. Proofs from tests. In *ISSTA*, 2008.
3. 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.
4. P. Cousot and R. Cousot. Abstract interpretation. In *POPL*, 1977.
5. V. Ganesh and D. Dill. A decision procesure for bit-vectors and arrays. In *CAV*, 2007.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.
7. P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.
8. R. Jhala and R. Majumdar. B2: Software model checking for C, 2009. www.cs.ucla.edu/∼rupak/b2/.
9. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
10. K. Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., 1993.
11. J. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theor. Found. of Program. Methodology*. Reidel, 1982.
12. A. Mycroft and N. Jones. A relational framework for abstract interpretation. In *PADO*, 1985.
13. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.
14. Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. *TOPLAS*, 29(3), 2007.
15. Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, 2003.