Symbolic Analysis via Semantic Reinterpretation

Junghee Lim   Akash Lal   Thomas Reps
University of Wisconsin
{junghee, akash, reps}@cs.wisc.edu

Abstract

In recent years, the use of symbolic analysis in systems for testing and verifying programs has experienced a resurgence. By "symbolic program analysis", we mean logic-based techniques to analyze state changes along individual program paths. The three basic primitives used in symbolic analysis are functions that perform *forward symbolic evaluation*, *weakest precondition*, and *symbolic composition* by manipulating formulas.

The conventional approach to implementing systems that use symbolic analysis is to write each of the three symbolic-analysis functions by hand for the programming language of interest. In this paper, we develop a method to create implementations of these primitives so that they can be made available easily for multiple programming languages—particularly for multiple machine-code instruction sets. In particular, we have created a system in which, for the cost of writing just *one* specification—of the semantics of the programming language of interest, in the form of an interpreter expressed in a functional language—one obtains automatically-generated implementations of all *three* symbolic-analysis functions. We show that this can be carried out even for programming languages with pointers, aliasing, dereferencing, and address arithmetic. The technique has been implemented, and used to automatically generate symbolic-analysis primitives for multiple machine-code instruction sets.

1. Introduction

This paper presents new ways to create implementations of the basic primitives used in certain kinds of verification and testing tools that are based on symbolic program analysis. By "symbolic program analysis", we mean logic-based techniques to analyze state changes along individual program paths.[1] The basic primitives used in symbolic analysis are functions that perform *forward symbolic evaluation*, *weakest precondition*, and *symbolic composition* by manipulating formulas.

The conventional approach to implementing systems that use symbolic analysis is to write each of the three symbolic-analysis functions by hand for the programming language of interest (which we call the *subject* language).[2] Our goal is to develop a method to create implementations of symbolic-analysis primitives easily, so that they can be made available for different subject languages—particularly for different machine-code instruction sets. Such instruction sets typically have (i) several hundred instructions, (ii) a variety of architecture-specific features that are incompatible with other architectures, and (iii) the ability to perform address arithmetic and dereferencing of addresses, which means that memory states can have complicated aliasing patterns. Moreover, most instruction sets have evolved over time, so that each instruction-set family has a bewildering number of variants.[3] Consequently, our goal is to *generate* implementations of such primitives automatically from a specification of the subject language's concrete semantics.

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* [27]), and the client is expressed in terms of this interface. This organization permits the core to be *reinterpreted* to produce an alternative semantics for the subject language.

Semantic reinterpretation for abstract interpretation. The idea of exploiting such a factoring comes from the field of abstract interpretation [7], where factoring-plus-reinterpretation has been proposed as a convenient tool for formulating abstract interpretations and proving them to be sound [23, 24, 21]. In particular, soundness of the *entire* abstract semantics can be established via purely *local* soundness arguments for each of the reinterpreted operators. (An example of semantic reinterpretation for abstract interpretation is presented in §2.)

Semantic reinterpretation for symbolic analysis. This paper presents a new application for semantic reinterpretation, namely, to create implementations of the basic primitives used in symbolic program analysis.

In recent years, the use of symbolic analysis in systems for testing and verifying programs has experienced a resurgence because of the power that they provide in exploring a program's state space.

[1] This is in contrast to the situation addressed by many abstract-interpretation/dataflow-analysis techniques, which usually consider the problem of analyzing the effects of a *collection* of program paths—e.g., to identify program invariants.

[2] Semantic reinterpretation is a program-generation technique, and thus we follow the terminology of the partial-evaluation literature [16], where the program on which the partial evaluator operates is called the *subject program*. (§3.2 and §8 discuss the connections between our approach and partial evaluation.)

In logic and linguistics, the programming language would be called the "object language". We avoid that terminology because of possible confusion in §7, 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.

[3] See http://en.wikipedia.org/wiki/{X86,ARM_architecture,PowerPC}. For instance, the article about ARM lists 18 different architectural versions.

- Model-checking tools, such as SLAM [1] and BLAST [14], as well as hybrid concrete/symbolic program-exploration tools, such as DART [10], CUTE [28], YOGI [13], SAGE [11], BITSCOPE [5], and DASH [2] use forward symbolic evaluation, weakest precondition, or both.

  Symbolic evaluation can be used to create path formulas. When it is possible that a path π being analyzed might not be executable, a call on an SMT solver to determine whether π's path formula is satisfiable can be used to decide whether π is executable, and if so, to generate inputs that drive the program down π. Weakest precondition can be used to create new predicates that split part of a program's state space [1, 13, 2].

- Bug-finding tools, such as ARCHER [32] and SATURN [31], as well as commercial bug-finding products, such as Coverity's PREVENT [8] and GrammaTech's CODESONAR [12] 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 $[\mathit{entry}_P, \ldots, c]$ explored in P with the formula that captures the behavior of Q to obtain a formula that expresses the behavior along the path $[\mathit{entry}_P, \ldots, r]$.

The aforementioned systems apply symbolic analysis to programs written in languages with pointers, aliasing, dereferencing, and address arithmetic. This paper demonstrates that the reinterpretation technique provides a way to create symbolic-analysis primitives for such languages.

As mentioned earlier, our motivation is to be able to create implementations of symbolic-analysis primitives for multiple *machine-code instruction sets* (including multiple variants of a given machine-code instruction set). However, our work is also useful for creating tools to analyze *high-level-language programs*, starting from source code. Moreover, most of the principles that we make use of can be explained using two variants of a simple high-level language: PL$_1$, defined in §4, and PL$_2$, defined in §6. For this reason, the paper is couched in terms of high-level languages up until §7, which discusses an idealized machine-code language, MC. This has the benefit of making the paper accessible to a wider number of readers, but might cause readers who are mainly familiar with analysis techniques for C, C++, C#, or Java to under-appreciate the benefits that one obtains from our approach when creating machine-code-analysis tools.

Three for the price of one! In §8, we describe how, using binding-time analysis [16] and a two-level intermediate language [25], the reinterpretation technique can be used to generate implementations of symbolic-analysis primitives automatically, using a meta-system that generates program-analysis components from a specification of the subject language's semantics. In particular, we have created a system in which, for the cost of writing just *one* specification—of the semantics of the programming language of interest, in the form of an interpreter expressed in a functional language—one obtains automatically-generated implementations of all *three* symbolic-analysis functions. We show that this can be carried out even for programming languages with pointers, aliasing, dereferencing, and address arithmetic.

This has been achieved using the TSL system [20], and the implementation has been used to generate symbolic-analysis primitives for multiple machine-code instruction sets. TSL[4] consists of (i) a language for specifying the concrete semantics of a machine-code instruction set (i.e., a collection of concrete-state transformers), (ii) a mechanism to create implementations of different abstract interpretations easily by reinterpreting the TSL base types, function types, and operators, and (iii) a run-time system to support the (re-)interpretation and analysis of executables written in that instruction set.

Moreover, with TSL each reinterpretation is defined at the *meta-level*, by reinterpreting the collection of TSL base types, function types, and operators. When a reinterpretation is performed in this way, it is independent of any given subject language. Consequently, with our implementation, all three of the symbolic-analysis primitives can be generated automatically for *every* instruction set for which one has a TSL specification.

The contributions of the paper can be summarized as follows:

- From the conceptual standpoint, we present a new application for semantic reinterpretation. In particular, the paper shows how semantic reinterpretation can be applied to create analysis functions that compute formulas for forward symbolic evaluation, weakest precondition, and symbolic composition (§5.1, §5.2, and §5.3, respectively).
- From the systems-building perspective, we show that this observation has algorithmic content: the paper describes how we created a meta-system that, given an interpreter that specifies a subject language's concrete semantics, uses binding-time analysis, a two-level intermediate language, and semantic reinterpretation to automatically generate implementations of all three of symbolic-analysis primitives, for *every* instruction set for which one has a TSL specification (§8).
- We demonstrate that semantic reinterpretation can handle languages with pointers, aliasing, dereferencing, and address arithmetic (§3, §6, and §7). In particular, in §3 and §6.4.1, we show how reinterpretation can automatically generate a weakest-precondition primitive that implements Morris's rule of substitution for a language with pointer variables [22].
- §6.4.2 shows how the semantic-reinterpretation approach can also generate a weakest-precondition primitive that implements the pure substitution-based approach of Cartwright and Oppen [6] (again for a language with pointer variables). This provides insight on how Morris's rule and Cartwright and Oppen's rule are related: both are based on substitution; the difference is merely the degree of algebraic simplification that is performed.

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 provides an overview of our techniques and the results obtained with the symbolic-analysis primitives that are created by semantic reinterpretation. §4 defines the logic that we use, as well as the programming languages PL$_1$. §5 discusses how to use reinterpretation to obtain the primitives for forward symbolic evaluation, weakest precondition, and symbolic composition. §6 defines PL$_2$, which includes pointer variables and dereferencing, and shows how the weakest-precondition operation that is obtained automatically via semantic reinterpretation implements Morris's rule of substitution. §7 introduces a simplified machine-code language, which includes address arithmetic and dereferencing, and shows that the reinterpretation technique applies at the machine-code level, as well. §8 describes how these ideas are implemented using the TSL system [20]. §9 discusses related work. (Proofs of two lemmas appear in App. A.)

[4] TSL stands for "**T**ransformer **S**pecification **L**anguage".

$$
\begin{array}{llll}
s_1: & x = x \oplus y; & t_1: & *px = *px \oplus *py; \\
s_2: & y = x \oplus y; & t_2: & *py = *px \oplus *py; \\
s_3: & x = x \oplus y; & t_3: & *px = *px \oplus *py; \\
 & \text{(a)} & & \text{(b)}
\end{array}
$$

Figure 1. (a) Code fragment that swaps two `ints`; (b) (buggy) code fragment that swaps two `ints` using pointers.

2. Semantic Reinterpretation for Abstract Interpretation

To illustrate factoring-plus-reinterpretation in the context of abstract interpretation, and as a warm-up exercise for the rest of the paper, this section presents the basic principle of semantic reinterpretation using a simple example in which, both the concrete semantics, for a language of assignment statements, and an abstract sign-analysis semantics are defined via semantic reinterpretation.

Example 2.1. [Adapted from [21].] 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):

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

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

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

This specification 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

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

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

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

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

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

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, the semantic core is reinterpreted as follows:

$$
v \in \mathit{Val}_{\mathit{abs}} = \{\mathit{neg}, \mathit{zero}, \mathit{pos}\}^{\top} \qquad \mathit{State}_{\mathit{abs}} = \mathit{Id} \to \mathit{Val}_{\mathit{abs}}
$$

$\mathit{xor}_{\mathit{abs}} = \lambda v_1.\lambda v_2.$

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

$$
\begin{array}{rcl}
\mathit{lookup}_{\mathit{abs}} & = & \lambda\sigma.\lambda I.\sigma I \\
\mathit{store}_{\mathit{abs}} & = & \lambda\sigma.\lambda I.\lambda v.\sigma[I \mapsto v]
\end{array}
$$

For instance, for the code fragment shown in Fig. 1, which swaps two `ints`, sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\}$, produce the following abstract states:[5]

$$
\begin{array}{rclcl}
\sigma_0 & := & & & \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\} \\
\sigma_1 & := & \mathcal{I}[\![s_1 : x = x \oplus y;]\!]\sigma_0 & = & \mathit{store}_{\mathit{abs}}\ \sigma_0\ x\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos}) \\
 & & & = & \{x \mapsto \mathit{neg}, y \mapsto \mathit{pos}\} \\
\sigma_2 & := & \mathcal{I}[\![s_2 : y = x \oplus y;]\!]\sigma_1 & = & \mathit{store}_{\mathit{abs}}\ \sigma_1\ y\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos}) \\
 & & & = & \{x \mapsto \mathit{neg}, y \mapsto \mathit{neg}\} \\
\sigma_3 & := & \mathcal{I}[\![s_3 : x = x \oplus y;]\!]\sigma_2 & = & \mathit{store}_{\mathit{abs}}\ \sigma_2\ x\ (\mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg}) \\
 & & & = & \{x \mapsto \top, y \mapsto \mathit{neg}\}
\end{array}
$$

□

3. Overview

This section presents intuition about some of the elements that are used in our work, and provides an overview of how it is possible to automatically generate the three symbolic-analysis primitives. §3.1 defines a stripped-down version of a logic L that is sufficient for the discussion in this section. (The full logic is defined in §4.1.) §3.2 presents examples of semantic reinterpretation applied to forward symbolic evaluation; §3.3 discusses issues relevant to weakest precondition and symbolic composition. We use the two swap-code fragments shown in Fig. 1 as a running example.

Because tools that check path feasibility (à la SLAM [1]) or perform path exploration (à la DART [10], CUTE [28], SAGE [11], and DASH [2]) only analyze traces, we can concentrate on non-branching statement sequences. For this reason, our programming-language definitions contain only assignment statements and statement sequences, and do not have either if-then-else statements or loop constructs.

3.1 A Simple Logic

The syntax of L is defined as follows:

$$
\begin{array}{rcl}
I & \in & \mathit{Id}, T \in \mathit{Term}, \varphi \in \mathit{Formula} \\
F & \in & \mathit{FuncId}, \mathit{FE} \in \mathit{FuncExpr}, U \in \mathit{FOUpdate} \\
 & & \\
T & ::= & I \mid T_1 \boxed{\oplus} T_2 \mid \mathit{FE}(T) \\
\varphi & ::= & T_1 \boxed{=} T_2 \mid \varphi_1 \boxed{\&\&} \varphi_2 \mid \ldots \\
\mathit{FE} & ::= & F \mid \mathit{FE}_1[T_1 \mapsto T_2] \\
U & ::= & (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})
\end{array}
$$

Names of the form $F \in \mathit{FuncId}$, possibly with subscripts and/or primes, are function symbols. We distinguish the *xor* constructor of L from the programming-language *xor* (§2) by putting the former in a box. A *FuncExpr* of the form $\mathit{FE}_1[T_1 \mapsto T_2]$ denotes a *function-update expression*.

An expression 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: $(\{I'_i \hookleftarrow T_i\}, \{F'_j \hookleftarrow \mathit{FE}_j\})$. (Also, if a component has only a singleton set, we omit the set brackets.) $\{I'_i \hookleftarrow T_i\}$ specifies the updates to the constants and $\{F'_j \hookleftarrow \mathit{FE}_j\}$ specifies the updates to the functions. 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)

[5] For numbers represented in two's complement notation,

$$
\mathit{pos}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg} = \mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos} = \mathit{neg}
$$

because, for all combinations of values represented by *pos* and *neg*, the sign bit of the result is set, which means that the result is guaranteed to be negative. However,

$$
\mathit{pos}\ \mathit{xor}_{\mathit{abs}}\ \mathit{pos} = \mathit{neg}\ \mathit{xor}_{\mathit{abs}}\ \mathit{neg} = \top
$$

because the concrete result could be either 0 or positive, and $\mathit{zero} \sqcup \mathit{pos} = \top$.

formula

$$\bigwedge_i (I_i' = T_i) \wedge \bigwedge_j (F_j' = \mathit{FE}_j).$$

We define U_{id} to be

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

Example 3.1. In §5, we work with a simple high-level language, PL$_1$, that only has `int`-valued variables. (PL$_1$ is the language from §2, extended with some additional kinds of expressions.) In §6, we introduce PL$_2$, which extends PL$_1$ with pointers. Here we confine ourselves to sketching how the semantics of various kinds of assignment statements can be expressed in $L[\text{PL}_1]$ and $L[\text{PL}_2]$.

- In PL$_1$, a state $\sigma \in \mathit{State}$ is a map $\mathit{Id} \to \mathit{Int32}$. This is modeled in $L[\text{PL}_1]$ by using a constant $c_x \in \mathit{Id}$ for each PL$_1$ identifier x. (However, to reduce clutter, we will merely use x for such constants instead of c_x.)
- In PL$_2$, a state σ is a pair (η, ρ), where, *environment* $\eta \in \mathit{Env} = \mathit{Id} \to \mathit{Loc}$ maps identifiers to their associated locations and *store* $\rho \in \mathit{Store} = \mathit{Loc} \to \mathit{Int32}$ maps each location to the value that it holds. (*Loc* stands for *locations*—e.g., memory addresses—and we identify *Loc* with the set *Int32* of values.) This is modeled in $L[\text{PL}_2]$ by using a function symbol F_ρ for store ρ, and a constant symbol $c_x \in \mathit{Id}$ for each PL$_2$ identifier x. (Again, to reduce clutter, we will use x for such constants instead of c_x.) The constants and their values correspond to the environment η.

The following table illustrates how the semantics of a few assignment statements are expressed as $L[\text{PL}_1]$ and $L[\text{PL}_2]$ structure-update expressions:

PL$_1$	$L[\text{PL}_1]$
$x = 17;$	$(x' \hookleftarrow 17, \emptyset)$
$x = y;$	$(x' \hookleftarrow y, \emptyset)$

PL$_2$	$L[\text{PL}_2]$
$x = 17;$	$(\emptyset, F_\rho' \hookleftarrow F_\rho[x \mapsto 17])$
$x = y;$	$(\emptyset, F_\rho' \hookleftarrow F_\rho[x \mapsto F_\rho(y)])$
$x = *q;$	$(\emptyset, F_\rho' \hookleftarrow F_\rho[x \mapsto F_\rho(F_\rho(q))])$

□

The semantics of L is defined in terms of a *logical structure*, which gives meaning to the *Id* and *FuncId* symbols of the logic's vocabulary.

$$\iota \in \mathit{LogicalStruct} = (\mathit{Id} \to \mathit{Int32}) \times (\mathit{FuncId} \to (\mathit{Int32} \to \mathit{Int32}))$$

We use $(\iota{\uparrow}1)$ and $(\iota{\uparrow}2)$ to denote the first and second components of ι, respectively. $(\iota{\uparrow}1)$ assigns meanings to constant symbols; $(\iota{\uparrow}2)$ assigns meanings to function symbols.

$$\begin{aligned}
\mathcal{T} : \mathit{Term} &\to \mathit{LogicalStruct} \to \mathit{Int32} \\
\mathcal{T}\llbracket I \rrbracket \iota &= (\iota{\uparrow}1)\ I \\
\mathcal{T}\llbracket T_1 \boxed{\oplus} T_2 \rrbracket \iota &= \mathcal{T}\llbracket T_1 \rrbracket \iota \oplus \mathcal{T}\llbracket T_2 \rrbracket \iota \\
\mathcal{T}\llbracket \mathit{FE}(T) \rrbracket \iota &= (\mathcal{FE}\llbracket \mathit{FE} \rrbracket \iota)(\mathcal{T}\llbracket T_1 \rrbracket \iota)
\end{aligned}$$

$$\begin{aligned}
\mathcal{F} : \mathit{Formula} &\to \mathit{LogicalStruct} \to \mathit{Bool} \\
\mathcal{F}\llbracket T_1 \boxed{=} T_2 \rrbracket \iota &= \mathcal{T}\llbracket T_1 \rrbracket \iota = \mathcal{T}\llbracket T_2 \rrbracket \iota \\
\mathcal{F}\llbracket \varphi_1 \boxed{\&\&} \varphi_2 \rrbracket \iota &= \mathcal{F}\llbracket \varphi_1 \rrbracket \iota \wedge \mathcal{F}\llbracket \varphi_1 \rrbracket \iota
\end{aligned}$$

$$\begin{aligned}
\mathcal{FE} : \mathit{FuncExpr} &\to \mathit{LogicalStruct} \to (\mathit{Int32} \to \mathit{Int32}) \\
\mathcal{FE}\llbracket F \rrbracket \iota &= (\iota{\uparrow}2)\ F \\
\mathcal{FE}\llbracket \mathit{FE}_1[T_1 \mapsto T_2] \rrbracket \iota &= (\mathcal{FE}\llbracket \mathit{FE}_1 \rrbracket \iota)[(\mathcal{T}\llbracket T_1 \rrbracket \iota) \mapsto (\mathcal{T}\llbracket T_2 \rrbracket \iota)]
\end{aligned}$$

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

$$\begin{aligned}
U_{\mathit{id}} &:= (\{x \hookleftarrow x, y \hookleftarrow y\}, \emptyset) \\
\overline{\mathcal{I}}\llbracket x = x \boxed{\oplus} y; \rrbracket U_{\mathit{id}} &= (\{x \hookleftarrow (\overline{\mathcal{E}}\llbracket x \rrbracket U_{\mathit{id}} \boxed{\oplus} \overline{\mathcal{E}}\llbracket y \rrbracket U_{\mathit{id}}), y \hookleftarrow y\}, \emptyset) \\
&= (\{x \hookleftarrow (x \boxed{\oplus} y), y \hookleftarrow y\}, \emptyset) \\
&= U_1 \\
\overline{\mathcal{I}}\llbracket y = x \boxed{\oplus} y; \rrbracket U_1 &= (\{x \hookleftarrow (x \boxed{\oplus} y), y \hookleftarrow (\overline{\mathcal{E}}\llbracket x \rrbracket U_1 \boxed{\oplus} \overline{\mathcal{E}}\llbracket y \rrbracket U_1)\}, \emptyset) \\
&= (\{x \hookleftarrow (x \boxed{\oplus} y), y \hookleftarrow ((x \boxed{\oplus} y) \boxed{\oplus} y)\}, \emptyset) \\
&= (\{x \hookleftarrow (x \boxed{\oplus} y), y \hookleftarrow x\}, \emptyset) \\
&= U_2 \\
\overline{\mathcal{I}}\llbracket x = x \boxed{\oplus} y; \rrbracket U_2 &= (\{x \hookleftarrow (\overline{\mathcal{E}}\llbracket x \rrbracket U_2 \boxed{\oplus} \overline{\mathcal{E}}\llbracket y \rrbracket U_2), y \hookleftarrow x\}, \emptyset) \\
&= (\{x \hookleftarrow ((x \boxed{\oplus} y) \boxed{\oplus} x), y \hookleftarrow x\}, \emptyset) \\
&= (\{x \hookleftarrow y, y \hookleftarrow x\}, \emptyset) \\
&= U_3
\end{aligned}$$

Figure 2. Symbolic execution of Fig. 1(a) via semantic reinterpretation, starting with the *FOUpdate* $U_{\mathit{id}} = (\{x \hookleftarrow x, y \hookleftarrow y\}, \emptyset)$.

Note how the meaning of a structure-update expression is a function that maps a pre-state logical structure ι to a post-state logical structure: $\{I_i \hookleftarrow T_i\}$ specifies the updates to the constants and $\{F_j \hookleftarrow \mathit{FE}_j\}$ specifies the updates to the functions.

3.2 Symbolic Evaluation via Reinterpretation

A primitive for forward symbolic-evaluation must solve the following problem:

> Given the semantic definition of a programming language, together with a specific programming-language statement (or instruction) s, create a logical formula that captures the semantics of s.

To apply semantic reinterpretation to this problem, we use formulas of logic L as a reinterpretation domain for the semantic core of PL$_1$. 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):

$$\overline{\mathit{Val}} = \mathit{Term} \qquad \overline{\mathit{BVal}} = \mathit{Formula} \qquad \overline{\mathit{State}} = \mathit{FOUpdate}$$

The operators used in the factored versions of PL$_1$'s meaning functions $\mathcal{E}$, $\mathcal{B}$, and $\mathcal{I}$ are reinterpreted over these domains; in particular, operations that are used in the PL$_1$ semantics—e.g., *xor*—are interpreted as syntactic constructors of $L[\text{PL}_1]$ expressions—e.g., $\boxed{\oplus}$. By extension, this produces reinterpreted meaning functions $\overline{\mathcal{E}}$, $\overline{\mathcal{B}}$, and $\overline{\mathcal{I}}$ with the types listed below:

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

The reinterpreted function $\overline{\mathcal{I}}$ translates a statement s of PL$_1$ to a phrase in logic $L[\text{PL}_1]$.

Example 3.2. The steps of symbolic execution of Fig. 1(a) via semantic reinterpretation, starting with the *FOUpdate* $U_{\mathit{id}} = (\{x \hookleftarrow x, y \hookleftarrow y\}, \emptyset)$ are shown in Fig. 2. The final *FOUpdate* U_3 can be considered to be the 2-vocabulary formula

$$(x' = y) \wedge (y' = x).$$

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

Algebraic simplification of the resulting terms and formulas also plays an important role. The simplification techniques that we use 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 [29, 9, 17]).

We assume that the reinterpreted $\boxed{\oplus}$ performs bit-vector simplification according to the algebraic laws for *xor*. For example, when y is updated in U_1 by $y \hookleftarrow ((x \boxed{\oplus} y) \boxed{\oplus} y)$ (see Fig. 2), this is simplified to $y \hookleftarrow x$. We assume that the other bit-vector, relational, and Boolean constructors of the logic behave similarly.

Relationship to partial evaluation. In general, the semantic definition of an imperative programming language is a meaning function $\mathcal{I}$ with type $\mathcal{I} : \mathit{Stmt} \times \mathit{State} \to \mathit{State}$. Given our goal, namely,

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

it is not surprising that partial-evaluation techniques come into play.

In essence, we wish to partially evaluate $\mathcal{I}$ with respect to *Stmt* s, while at the same time translating to L. Semantic reinterpretation permits us to do this: Let U_s be the *FOUpdate* $\overline{\mathcal{I}}\llbracket s \rrbracket U_{id}$. Then U_s is the partial evaluation of $\mathcal{I}$ with respect to s, translated to logic.

We show in §5.1 that U_s has the desired semantics. Note that to model PL$_1$ programs in $L[\mathrm{PL}_1]$, we do not require any function symbols. Thus, a PL$_1$ state σ can be identified with the *LogicalStruct* $(\sigma, \emptyset)$.[6] In §5.1, we show that for all $\iota \in$ *LogicalStruct*, evaluating U_s is equivalent to running $\mathcal{I}$ on s—i.e., $((\mathcal{U}\llbracket U_s \rrbracket \iota)\uparrow 1) = \mathcal{I}\llbracket s \rrbracket(\iota\uparrow 1)$ (see Cor. 5.3).

In our implementation, discussed in §8, the TSL system is supplied with a TSL program for the meaning function $\mathcal{I}$, and the way that it performs semantic reinterpretation is to create a kind of generating extension [16] $\mathcal{I}$-gen for $\mathcal{I}$.[7] The full explanation is complicated by the number of language levels involved when the partial-evaluation machinery is included in the discussion. For this reason, we have chosen to delay the discussion of generating extensions and partial-evaluation machinery until §8, and instead to base the discussion on the simpler principle of semantic reinterpretation. This 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 before §8 it may appear that many steps perform rather trivial transliteration of expressions from programming language PL$_i$ into expressions of the corresponding logic $L[\mathrm{PL}_i]$. 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" [3, 18].

3.3 Other Symbolic-Analysis Operations

For weakest precondition and symbolic composition, we again use $L[\cdot]$ as a reinterpretation domain; however, there is a trick: in contrast with what is done to generate symbolic-evaluation primitives, we use the *FOUpdate* type of $L[\cdot]$ to reinterpret the meaning functions $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$ of $L[\cdot]$ itself! 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*	*FOUpdate*	Symbolic evaluation
$\mathcal{F}, \mathcal{T}$	*LogicalStruct*	*FOUpdate*	Weakest precondition
$\mathcal{U}, \mathcal{FE}, \mathcal{F}, \mathcal{T}$	*LogicalStruct*	*FOUpdate*	Symbolic composition

To keep things simple in §3.2, we did not present the semantics of $L[\cdot]$ in factored form (see §4.1). Thus, the discussion in the rest of this section merely surveys a few of the results that are obtained by the techniques presented in later sections.

Weakest precondition. The weakest (liberal) precondition $\mathcal{WLP}(s, \varphi)$ 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 like PL$_1$, which 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: $\varphi[\mathit{var} \leftarrow \mathit{rhs}]$.

For the swap-code fragment shown in Fig. 1(a), repeated substitution and simplification shows that the weakest precondition of the program *swap* with respect to postcondition $x = 2$ is $y = 2$. (This will be derived using semantic reinterpretation in §5.2.)

Complications from pointers. When Hoare logic is extended for a language with pointer variables, such as PL$_2$, syntactic substitution is no longer adequate for finding weakest-precondition formulas. For instance, suppose that we are interested in finding a formula for the $\mathcal{WLP}$ of postcondition $x = 5$ with respect to $*p = e$;. This cannot be accomplished merely by performing the substitution $(x = 5)[*p \leftarrow e]$: the substitution yields the formula $x = 5$, 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 $x = 5$.

In this case, the $\mathcal{WLP}$ formula can be expressed informally as

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

Example 3.3. In §5.2, such formulas are expressed as shown below on the right.

	Informal	Formal
Query	$\mathcal{WLP}(*p = e, x = 5)$	$\mathcal{WLP}(*p = e, F_\rho(x) \boxed{=} 5)$
Result	$(p = \&x) \; ? \; (e = 5) : (x = 5)$	$\mathit{ite}(F_\rho(p) \boxed{=} x,$ $F_\rho(e) \boxed{=} 5,$ $F_\rho(x) \boxed{=} 5)$

□

For a program fragment that involves multiple pointer variables, the $\mathcal{WLP}$ formula may have to take into account all possible aliasing combinations. One of the most important features of our approach is its ability to create correct implementations of Morris's rule of substitution [22] automatically—and basically for free.

Symbolic analysis of machine code.

Example 3.4. Fig. 4(a) shows a source-code fragment; Fig. 4(b) shows the corresponding assembly code. To simplify the discussion, the source-level variables are used in the assembly code instead of having operations to access variable locations based on their frame-pointer-relative offsets in the activation record.

[6] Similarly, for PL$_2$ a *State* $\sigma = (\eta, \rho)$ can be identified with the *LogicalStruct* $(\eta, [F_\rho \mapsto \rho])$.

[7] If `p` is a two-input program, then `p-gen` is any program with the property that for every input pair a and b,

$$\llbracket \texttt{p-gen} \rrbracket(a) = \texttt{p}_a, \text{ where } \llbracket \texttt{p}_a \rrbracket(b) = \llbracket \texttt{p} \rrbracket(a, b).$$

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

$$\llbracket \mathcal{I}\texttt{-gen} \rrbracket(s) = \mathcal{I}_s, \text{ where } \llbracket \mathcal{I}_s \rrbracket(\sigma) = \llbracket \mathcal{I} \rrbracket(s, \sigma).$$

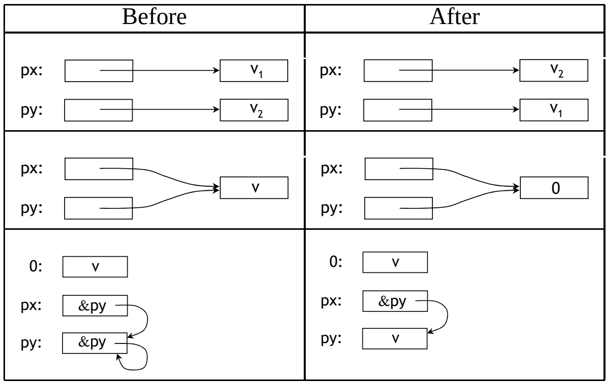

Figure 3. Before and after configurations for the (buggy) code fragment shown in Fig. 1(b), which attempts to swap two `int`s using pointers. Note that the swap is not successful in the second and third examples.

```
[1] void foo(int e,              [1] mov  eax, p;
[2]     int x, int* p) {         [2] mov  ecx, e;
[3]     ...                      [3] mov  [eax], ecx;
[4]     *p = e;                  [4] cmp  x, 5;
[5]     if(x == 5)               [5] jz   ERROR;
[6]         goto ERROR;          [6] ...
[7] }                            [7] ERROR: ...
              (a)                              (b)
```

Figure 4. (a) A simple source-code example written in PL$_2$. (b) A snippet of the assembly code for (a).

The answer to the query $\mathcal{WLP}(*p = e, x = 5)$ discussed in Ex. 3.3 describes the largest set of states just before line 4 in Fig. 4(a) that will cause the branch to ERROR to be taken at line 5. For the machine-code program shown in Fig. 4(b), the equivalent query is $\mathcal{WLP}(\texttt{mov eax},\ p;\ \texttt{mov ecx},\ e;\ \texttt{mov [eax]},\ \texttt{ecx}, x \boxed{=} 5)$, which describes the largest set of states just before line 1 in Fig. 4(b) that will cause the branch to ERROR to be taken.

Even when starting from the machine-code semantics, semantic reinterpretation will obtain the formula discussed in Ex. 3.3: $\mathit{ite}(F_{\mathit{mem}}(p) \boxed{=} x, F_{\mathit{mem}}(e) \boxed{=} 5, F_{\mathit{mem}}(x) \boxed{=} 5)$, or, using informal notation in source-level terms, $(p = \&x)\ ?\ (e = 5) : (x = 5)$. □

4. Definitions and Terminology

This section defines quantifier-free first-order bit-vector logic and a simple high-level language, PL$_1$, which only has `int`-valued variables.

4.1 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 ($I \in \mathit{Id}$) and function symbols ($F \in \mathit{FuncId}$). Strictly speaking, we work with various instantiations of L, denoted by L[PL$_1$], L[PL$_2$], 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$_1$, PL$_2$, and MC, respectively.

$$\begin{array}{rcl} c \in C_{\mathit{Int32}} & = & \{0, 1, \ldots\} \\ \mathit{op2}_L \in \mathit{BinOp}_L & = & \{\boxed{+}, \boxed{-}, \boxed{\oplus}, \ldots\} \\ \mathit{rop}_L \in \mathit{RelOp}_L & = & \{\boxed{=}, \boxed{\neq}, \boxed{<}, \boxed{>}, \ldots\} \\ \mathit{bop}_L \in \mathit{BoolOp}_L & = & \{\boxed{\&\&}, \boxed{||}, \ldots\} \end{array}$$

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

$$\begin{array}{rcl} I & \in & \mathit{Id}, T \in \mathit{Term}, \varphi \in \mathit{Formula} \\ F & \in & \mathit{FuncId}, \mathit{FE} \in \mathit{FuncExpr}, U \in \mathit{FOUpdate} \end{array}$$

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

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. (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*.)

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

The types of the functions that operate on *Term*s, *Formula*s, and *FuncExpr*s are as follows:

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

The meaning functions are defined as follows:

$$\begin{array}{rcl} \mathcal{T} : \mathit{Term} & \to & \mathit{LogicalStruct} \to \mathit{Val} \\ \mathcal{T}[\![c]\!]\iota & = & \mathit{const}(c) \\ \mathcal{T}[\![I]\!]\iota & = & \mathit{lookupId}\ \iota\ I \\ \mathcal{T}[\![T_1\ \mathit{op2}_L\ T_2]\!]\iota & = & \mathcal{T}[\![T_1]\!]\iota\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![T_2]\!]\iota \\ \mathcal{T}[\![\mathit{ite}(\varphi, T_1, T_2)]\!]\iota & = & \mathit{cond}_L(\mathcal{F}[\![\varphi]\!]\iota, \mathcal{T}[\![T_1]\!]\iota, \mathcal{T}[\![T_2]\!]\iota) \\ \mathcal{T}[\![\mathit{FE}(T_1)]\!]\iota & = & \mathit{access}(\mathcal{FE}[\![\mathit{FE}]\!]\iota, \mathcal{T}[\![T_1]\!]\iota) \end{array}$$

$$\begin{array}{rcl} \mathcal{F} : \mathit{Formula} & \to & \mathit{LogicalStruct} \to \mathit{BVal} \\ \mathcal{F}[\![\boxed{\mathbb{T}}]\!]\iota & = & \mathbb{T} \\ \mathcal{F}[\![\boxed{\mathbb{F}}]\!]\iota & = & \mathbb{F} \\ \mathcal{F}[\![T_1\ \mathit{rop}_L\ T_2]\!]\iota & = & \mathcal{T}[\![T_1]\!]\iota\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![T_2]\!]\iota \\ \mathcal{F}[\![\boxed{\neg}\,\varphi_1]\!]\iota & = & \neg\mathcal{F}[\![\varphi_1]\!]\iota \\ \mathcal{F}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!]\iota & = & \mathcal{F}[\![\varphi_1]\!]\iota\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\varphi_2]\!]\iota \end{array}$$

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

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

Let $U = (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\})$. 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 $F \in (\mathit{FuncId} - \{F_j\})$.

4.2 PL$_1$: A Simple High-Level Language

PL$_1$ is the language from §2, extended with some additional kinds of expressions. It is a simple high-level language that only has `int`-valued variables. (§6 discusses PL$_2$, which is PL$_1$ extended with pointers.)

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

$$\begin{aligned} c \in C_{\mathit{Int32}} &= \{0, 1, \ldots\} \\ \mathit{op2} \in \mathit{BinOp} &= \{+, -, \oplus, \ldots\} \\ \mathit{rop} \in \mathit{RelOp} &= \{=, \neq, <, >, \ldots\} \\ \mathit{bop} \in \mathit{BoolOp} &= \{\&\&, ||, \ldots\} \end{aligned}$$

$$\begin{aligned} E &::= c \mid I \mid E_1\ \mathit{op2}\ E_2 \mid \mathit{BE}\ ?\ E_1 : E_2 \\ \mathit{BE} &::= \mathbb{T} \mid \mathbb{F} \mid E_1\ \mathit{rop}\ E_2 \mid \neg \mathit{BE}_1 \mid \mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2 \\ S &::= I = E; \mid S_1\ S_2 \end{aligned}$$

The (factored) semantics of PL$_1$ is defined in terms of the following operators:

$$\begin{aligned} \mathit{const} &: C_{\mathit{Int32}} \to \mathit{Val} \\ \mathit{binop} &: \mathit{BinOp} \to (\mathit{Val} \times \mathit{Val} \to \mathit{Val}) \\ \mathit{relop} &: \mathit{RelOp} \to (\mathit{Val} \times \mathit{Val} \to \mathit{BVal}) \\ \mathit{boolop} &: \mathit{BoolOp} \to (\mathit{BVal} \times \mathit{BVal} \to \mathit{BVal}) \\ \mathit{cond} &: \mathit{BVal} \to \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\ \mathit{lookup} &: \mathit{State} \to \mathit{Id} \to \mathit{Val} \\ \mathit{store} &: \mathit{State} \to \mathit{Id} \to \mathit{Val} \to \mathit{State} \end{aligned}$$

These appear in the meaning functions $\mathcal{E}$, $\mathcal{B}$, and $\mathcal{I}$:

$$\begin{aligned} \mathcal{E} : \mathit{Expr} &\to \mathit{State} \to \mathit{Val} \\ \mathcal{E}\llbracket c \rrbracket \sigma &= \mathit{const}(c) \\ \mathcal{E}\llbracket I \rrbracket \sigma &= \mathit{lookup}\ \sigma\ I \\ \mathcal{E}\llbracket E_1\ \mathit{op2}\ E_2 \rrbracket \sigma &= \mathcal{E}\llbracket E_1 \rrbracket \sigma\ \mathit{binop}(\mathit{op2})\ \mathcal{E}\llbracket E_2 \rrbracket \sigma \\ \mathcal{E}\llbracket \mathit{BE}\ ?\ E_1 : E_2 \rrbracket \sigma &= \mathcal{B}\llbracket \mathit{BE} \rrbracket \sigma\ ?\ \mathcal{E}\llbracket E_1 \rrbracket \sigma : \mathcal{E}\llbracket E_2 \rrbracket \sigma \end{aligned}$$

$$\begin{aligned} \mathcal{B} : \mathit{BoolExpr} &\to \mathit{State} \to \mathit{BVal} \\ \mathcal{B}\llbracket \mathbb{T} \rrbracket \sigma &= \mathbb{T} \\ \mathcal{B}\llbracket \mathbb{F} \rrbracket \sigma &= \mathbb{F} \\ \mathcal{B}\llbracket E_1\ \mathit{rop}\ E_2 \rrbracket \sigma &= \mathcal{E}\llbracket E_1 \rrbracket \sigma\ \mathit{relop}(\mathit{rop})\ \mathcal{E}\llbracket E_2 \rrbracket \sigma \\ \mathcal{B}\llbracket \neg \mathit{BE}_1 \rrbracket \sigma &= \neg \mathcal{B}\llbracket \mathit{BE}_1 \rrbracket \sigma \\ \mathcal{B}\llbracket \mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2 \rrbracket \sigma &= \mathcal{B}\llbracket \mathit{BE}_1 \rrbracket \sigma\ \mathit{boolop}(\mathit{bop})\ \mathcal{B}\llbracket \mathit{BE}_2 \rrbracket \sigma \end{aligned}$$

$$\begin{aligned} \mathcal{I} : \mathit{Stmt} &\to \mathit{State} \to \mathit{State} \\ \mathcal{I}\llbracket I = E; \rrbracket \sigma &= \mathit{store}\ \sigma\ I\ (\mathcal{E}\llbracket E \rrbracket \sigma) \\ \mathcal{I}\llbracket S_1\ S_2 \rrbracket \sigma &= \mathcal{I}\llbracket S_2 \rrbracket (\mathcal{I}\llbracket S_1 \rrbracket \sigma) \end{aligned}$$

5. Symbolic-Analysis Primitives via Reinterpretation

This section gives technical details of how to use semantic reinterpretation to obtain primitives for forward symbolic evaluation (§5.1), weakest precondition (§5.2), and symbolic composition (§5.3).

5.1 Symbolic Evaluation via Reinterpretation

The discussion in §3.2 of how semantic reinterpretation can be used to generate a symbolic-evaluation primitive was already fairly comprehensive. No new issues arise in extending the material presented in §3.2 to handle the full definitions of $L[\cdot]$ and PL$_1$ from §4, and thus the extensions will not be discussed here.

Correctness considerations. We now show that $\overline{\mathcal{I}}$ and $\mathcal{I}$ have the right relationship.

Lemma 5.1 (Relationship of $\overline{\mathcal{E}}$ to $\mathcal{E}$ and $\overline{\mathcal{B}}$ to $\mathcal{B}$)**.**

$$\begin{gathered} (1)\ \mathcal{T}\llbracket \overline{\mathcal{E}}\llbracket E \rrbracket U \rrbracket \iota = \mathcal{E}\llbracket E \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1) \\ (2)\ \mathcal{F}\llbracket \overline{\mathcal{B}}\llbracket \mathit{BE} \rrbracket U \rrbracket \iota = \mathcal{B}\llbracket \mathit{BE} \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1) \end{gathered}$$

Proof. See App. A. □

Theorem 5.2. *For all $\iota \in$ LogicalStruct, evaluating $\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket s \rrbracket U \rrbracket \iota$ is equivalent to running $\mathcal{I}$ on s with an input state obtained from $\mathcal{U}\llbracket U \rrbracket \iota$; that is, $((\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket s \rrbracket U \rrbracket \iota)\uparrow 1) = \mathcal{I}\llbracket s \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1)$.*

Proof.

$$\begin{aligned} (i)\ & ((\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket I = E; \rrbracket U \rrbracket \iota)\uparrow 1) \\ &= ((\mathcal{U}\llbracket ((U\uparrow 1)[I \mapsto \overline{\mathcal{E}}\llbracket E \rrbracket U], (U\uparrow 2)) \rrbracket \iota)\uparrow 1) \\ &= ((((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1)[I \mapsto \mathcal{T}\llbracket \overline{\mathcal{E}}\llbracket E \rrbracket U \rrbracket \iota], ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 2))\uparrow 1) \\ &= ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1)[I \mapsto \mathcal{E}\llbracket E \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1)] \text{ // by Lem. 5.1} \\ &= \mathcal{I}\llbracket I = E; \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1) \end{aligned}$$

$$\begin{aligned} (ii)\ & ((\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket S_1 S_2 \rrbracket U \rrbracket \iota)\uparrow 1) \\ &= ((\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket S_2 \rrbracket (\overline{\mathcal{I}}\llbracket S_1 \rrbracket U) \rrbracket \iota)\uparrow 1) \\ &= \mathcal{I}\llbracket S_2 \rrbracket ((\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket S_1 \rrbracket U \rrbracket \iota)\uparrow 1) \text{ // by induction} \\ &= \mathcal{I}\llbracket S_2 \rrbracket (\mathcal{I}\llbracket S_1 \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1)) \text{ // by induction} \\ &= \mathcal{I}\llbracket S_1 S_2 \rrbracket ((\mathcal{U}\llbracket U \rrbracket \iota)\uparrow 1) \end{aligned}$$

□

Corollary 5.3. *For all $\iota \in$ LogicalStruct,*

$$((\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket s \rrbracket U_{\mathit{id}} \rrbracket \iota)\uparrow 1) = \mathcal{I}\llbracket s \rrbracket (\iota\uparrow 1).$$

5.2 Weakest Liberal Precondition

In this section, we discuss how to use semantic reinterpretation to obtain a symbolic-analysis primitive for weakest liberal precondition. As mentioned in §3.3, one trick is to use $L[\cdot]$ to reinterpret the meaning functions $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$ of $L[\cdot]$ itself. By this means, the "alternative meaning" of a *Term*/*Formula*/*FuncExpr*/*FOUpdate* is a (usually different) *Term*/*Formula*/*FuncExpr*/*FOUpdate* in which some substitution and/or simplification has taken place.

In §4.1, we wrote the semantics of $L[\cdot]$ in factored form so that it would be possible to perform semantic reinterpretation. However, one small point needs adjustment: in §4.1, the type signatures of *LogicalStruct*, *lookupFuncId*, *access*, *update*, and $\mathcal{FE}$ include occurrences of *Val* → *Val*. This was done to make the types more intuitive; however, for a reinterpretation scheme to work, an additional level of factoring is necessary. In particular, the occurrences of *Val* → *Val* need to be replaced by *FVal*. The standard semantics of *FVal* is *Val* → *Val* (i.e., *Int32* → *Int32*); for creating symbolic-analysis primitives, *FVal* is reinterpreted as *FuncExpr*.

After this change, we use the logic L as a reinterpretation domain for the semantic core of L, defined in §4.1. The base types and the state type of the semantic core are reinterpreted as follows:

$$\begin{aligned} \overline{\mathit{Val}} &= \mathit{Term} & \overline{\mathit{FVal}} &= \mathit{FuncExpr} \\ \overline{\mathit{BVal}} &= \mathit{Formula} & \overline{\mathit{LogicalStruct}} &= \mathit{FOUpdate} \end{aligned}$$

The operators used in the factored versions of $\mathcal{U}$, $\mathcal{FE}$, $\mathcal{F}$, and $\mathcal{T}$, are reinterpreted over these domains. (In particular, $\overline{\mathit{binop}}_L$, $\overline{\mathit{relop}}_L$, and $\overline{\mathit{boolop}}_L$ are interpreted basically as syntactic *Term* and *Formula* constructors of L—although, as discussed in §3, the reinterpreted base-type operations perform simplification, whenever possible, when constructing *Term*s and *Formula*s.) By extension, this produces reinterpreted meaning functions $\overline{\mathcal{U}}$, $\overline{\mathcal{FE}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{T}}$ with the types listed in Fig. 5.

WLP via semantic reinterpretation. To compute 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. As we show in Thm. 5.6, we can compute a weakest-precondition formula for φ with respect to statement s by performing the following computation:

$$\overline{\mathcal{F}}\llbracket \varphi \rrbracket (\overline{\mathcal{I}}\llbracket s \rrbracket U_{\mathit{id}}) \tag{1}$$

Example 5.1. In Ex. 3.2 and Fig. 2, we derived the following *FOUpdate*, which expresses in logic L the semantics of the swap-code fragment *swap* from Fig. 1(a):

$$U_{\mathit{swap}} := \overline{\mathcal{I}}\llbracket \mathit{swap} \rrbracket U_{\mathit{id}} = (\{x' \hookleftarrow y, y' \hookleftarrow x\}, \emptyset)$$

Standard Reinterpreted	$\mathcal{U} : \mathit{FOUpdate} \rightarrow \mathit{LogicalStruct} \rightarrow \mathit{LogicalStruct}$ $\overline{\mathcal{U}} : \mathit{FOUpdate} \rightarrow \overline{\mathit{LogicalStruct}} \rightarrow \overline{\mathit{LogicalStruct}}$ $: \mathit{FOUpdate} \rightarrow \mathit{FOUpdate} \rightarrow \mathit{FOUpdate}$
Standard Reinterpreted	$\mathcal{FE} : \mathit{FuncExpr} \rightarrow \mathit{LogicalStruct} \rightarrow \mathit{FVal}$ $\overline{\mathcal{FE}} : \mathit{FuncExpr} \rightarrow \overline{\mathit{LogicalStruct}} \rightarrow \overline{\mathit{FVal}}$ $: \mathit{FuncExpr} \rightarrow \mathit{FOUpdate} \rightarrow \mathit{FuncExpr}$
Standard Reinterpreted	$\mathcal{F} : \mathit{Formula} \rightarrow \mathit{LogicalStruct} \rightarrow \mathit{BVal}$ $\overline{\mathcal{F}} : \mathit{Formula} \rightarrow \overline{\mathit{LogicalStruct}} \rightarrow \overline{\mathit{BVal}}$ $: \mathit{Formula} \rightarrow \mathit{FOUpdate} \rightarrow \mathit{Formula}$
Standard Reinterpreted	$\mathcal{T} : \mathit{Term} \rightarrow \mathit{LogicalStruct} \rightarrow \mathit{Val}$ $\overline{\mathcal{T}} : \mathit{Term} \rightarrow \overline{\mathit{LogicalStruct}} \rightarrow \overline{\mathit{Val}}$ $: \mathit{Term} \rightarrow \mathit{FOUpdate} \rightarrow \mathit{Term}$

Figure 5. Types of the reinterpreted meaning functions $\overline{\mathcal{U}}$, $\overline{\mathcal{FE}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{T}}$.

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

$$\begin{aligned}
\mathcal{WLP}(\mathit{swap}, x \boxed{=} 2) &= \overline{\mathcal{F}}\llbracket x \boxed{=} 2 \rrbracket U_{\mathit{swap}} \\
&= \overline{\mathcal{T}}\llbracket x \rrbracket U_{\mathit{swap}} \boxed{=} \overline{\mathcal{T}}\llbracket 2 \rrbracket U_{\mathit{swap}} \\
&= (\overline{\mathit{lookupId}}\ U_{\mathit{swap}}\ x) \boxed{=} \overline{\mathit{const}}(2) \\
&= y \boxed{=} 2
\end{aligned}$$

□

Correctness considerations. Although weakest liberal precondition is sometimes confused with the formula-manipulation operations used to obtain a formula that expresses it, or with the formula ψ that results, weakest liberal precondition is really a semantic notion—the set of states *described* by ψ. For example, for any statement s: *var* $=$ *rhs*; in a language like PL$_1$ that only has `int`-valued variables, and postcondition formula φ, the formula $\varphi[\mathit{var} \leftarrow \mathit{rhs}]$ obtained by substitution is not the only formula that expresses $\mathcal{WLP}(s, \varphi)$. In fact, there are an infinity of acceptable formulas. Thus, to address the correctness of the answers obtained via Eqn. (1), we characterize what constitutes an acceptable formula as follows:

Definition 5.4 (Acceptable $\mathcal{WLP}$ Formula.)**.** *ψ is an* acceptable *formula for $\mathcal{WLP}(s, \varphi)$ iff, for all $\iota \in$ LogicalStruct,*

$$\mathcal{F}\llbracket \psi \rrbracket \iota = \mathcal{F}\llbracket \varphi \rrbracket (\mathcal{I}\llbracket s \rrbracket (\iota{\uparrow}1), (\iota{\uparrow}2)). \qquad (2)$$

That is, ψ holds in the pre-state structure ι exactly when φ holds in the post-state structure $(\mathcal{I}\llbracket s \rrbracket (\iota{\uparrow}1), (\iota{\uparrow}2))$. (Recall that a PL$_1$ state σ is identified with the *LogicalStruct* $(\sigma, \emptyset)$, and thus $(\iota{\uparrow}2) = \emptyset$ in Eqn. (2).)

It is pleasing to observe that the method for computing a weakest-precondition formula, Eqn. (1), is nearly identical syntactically to the right-hand side of Eqn. (2), which defines when a formula is an acceptable formula for $\mathcal{WLP}(s, \varphi)$.

Lemma 5.5 (Relationship of $\overline{\mathcal{F}}$ to $\mathcal{F}$, $\overline{\mathcal{T}}$ to $\mathcal{T}$, $\overline{\mathcal{FE}}$ to $\mathcal{FE}$)**.**

$$\begin{aligned}
(1)\ \mathcal{F}\llbracket \overline{\mathcal{F}}\llbracket \varphi \rrbracket U \rrbracket \iota &= \mathcal{F}\llbracket \varphi \rrbracket (\mathcal{U}\llbracket U \rrbracket \iota) \\
(2)\ \mathcal{T}\llbracket \overline{\mathcal{T}}\llbracket T \rrbracket U \rrbracket \iota &= \mathcal{T}\llbracket T \rrbracket (\mathcal{U}\llbracket U \rrbracket \iota) \\
(3)\ \mathcal{FE}\llbracket \overline{\mathcal{FE}}\llbracket \mathit{FE} \rrbracket U \rrbracket \iota &= \mathcal{FE}\llbracket \mathit{FE} \rrbracket (\mathcal{U}\llbracket U \rrbracket \iota)
\end{aligned}$$

Proof. See App. A. □

Theorem 5.6. *For any Stmt s and Formula φ, $\psi := \overline{\mathcal{F}}\llbracket \varphi \rrbracket (\overline{\mathcal{I}}\llbracket s \rrbracket U_{\mathit{id}})$ is an acceptable weakest-precondition formula for φ with respect to s.*

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

$$\begin{aligned}
\mathcal{F}\llbracket \psi \rrbracket \iota &= \mathcal{F}\llbracket \overline{\mathcal{F}}\llbracket \varphi \rrbracket (\overline{\mathcal{I}}\llbracket s \rrbracket U_{\mathit{id}}) \rrbracket \iota \\
&= \mathcal{F}\llbracket \varphi \rrbracket (\mathcal{U}\llbracket \overline{\mathcal{I}}\llbracket s \rrbracket U_{\mathit{id}} \rrbracket \iota) \quad \text{// by Lem. 5.5} \\
&= \mathcal{F}\llbracket \varphi \rrbracket (\mathcal{I}\llbracket s \rrbracket (\iota{\uparrow}1), (\iota{\uparrow}2)) \quad \text{// by Cor. 5.3}
\end{aligned}$$

and therefore, by Defn. 5.4, $\overline{\mathcal{F}}\llbracket \varphi \rrbracket (\overline{\mathcal{I}}\llbracket s \rrbracket U_{\mathit{id}})$ is an acceptable weakest-precondition formula for φ with respect to s. □

5.3 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 this section, we describe how to use semantic reinterpretation to create such a method automatically: each state change is represented in logic $L[\cdot]$ by an *FOUpdate*, and the method computes a new *FOUpdate* that represents their composition.

To accomplish this, $L[\cdot]$ is used as a reinterpretation domain, exactly as in §5.2; the reinterpreted meaning functions $\overline{\mathcal{U}}$, $\overline{\mathcal{FE}}$, $\overline{\mathcal{F}}$, and $\overline{\mathcal{T}}$ have the types listed in Fig. 5. Moreover, $\overline{\mathcal{U}}$ turns out to be exactly the symbolic-composition function that we seek.

$\overline{\mathcal{U}}$, the reinterpretation of $\mathcal{U}$, works as follows:

$$\begin{aligned}
&\overline{\mathcal{U}} : \mathit{FOUpdate} \rightarrow \mathit{FOUpdate} \rightarrow \mathit{FOUpdate} \\
&\overline{\mathcal{U}}\llbracket (\{I_i \hookleftarrow T_i\}, \{F_j \hookleftarrow \mathit{FE}_j\}) \rrbracket U \\
&\qquad = ((U{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}\llbracket T_i \rrbracket U], (U{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}\llbracket \mathit{FE}_j \rrbracket U])
\end{aligned}$$

In the remainder of this section, we show that $\overline{\mathcal{U}}$ performs symbolic composition of *FOUpdate*s.

Example 5.2. 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

$$\overline{\mathcal{I}}\llbracket s_1; s_2; s_3 \rrbracket U_{\mathit{id}} = \overline{\mathcal{U}}\llbracket \overline{\mathcal{I}}\llbracket s_3 \rrbracket U_{\mathit{id}} \rrbracket (\overline{\mathcal{I}}\llbracket s_1; s_2 \rrbracket U_{\mathit{id}}), \qquad (3)$$

where in this case $U_{\mathit{id}} = (\{x \hookleftarrow x, y \hookleftarrow y\}, \emptyset)$.

Consider the left-hand side of Eqn. (3). As shown in Ex. 3.2 and Fig. 2, $U_{\mathit{swap}} := \overline{\mathcal{I}}\llbracket \mathit{swap} \rrbracket U_{\mathit{id}} = (\{x \hookleftarrow y, y \hookleftarrow x\}, \emptyset)$.

Now consider the right-hand side of Eqn. (3). Let $U_{1,2}$ and U_3 be defined as follows:

$$\begin{aligned}
U_{1,2} &= \overline{\mathcal{I}}\llbracket s_1; s_2 \rrbracket U_{\mathit{id}} &&= (\{x \hookleftarrow x \boxed{\oplus} y, y \hookleftarrow x\}, \emptyset) \\
U_3 &= \overline{\mathcal{I}}\llbracket s_3 \rrbracket U_{\mathit{id}} &&= (\{x \hookleftarrow x \boxed{\oplus} y, y \hookleftarrow y\}, \emptyset).
\end{aligned}$$

We want to compute

$$\begin{aligned}
\overline{\mathcal{U}}\llbracket U_3 \rrbracket U_{1,2} &= \overline{\mathcal{U}}\llbracket (\{x \hookleftarrow x \boxed{\oplus} y, y \hookleftarrow y\}, \emptyset) \rrbracket U_{1,2} \\
&= ((U_{1,2}{\uparrow}1)[x \mapsto \mathcal{T}\llbracket x \boxed{\oplus} y \rrbracket U_{1,2}, y \mapsto \mathcal{T}\llbracket y \rrbracket U_{1,2}], \emptyset) \\
&= ((U_{1,2}{\uparrow}1)[x \mapsto ((x \boxed{\oplus} y) \boxed{\oplus} x), y \mapsto x], \emptyset) \\
&= ((U_{1,2}{\uparrow}1)[x \mapsto y, y \mapsto x], \emptyset) \\
&= (\{x \hookleftarrow y, y \hookleftarrow x\}, \emptyset) \\
&= U_{\mathit{swap}}
\end{aligned}$$

Therefore, $\overline{\mathcal{I}}\llbracket s_1; s_2; s_3 \rrbracket U_{\mathit{id}} = \overline{\mathcal{U}}\llbracket U_3 \rrbracket U_{1,2}$. □

At the semantic level, the ability of $\overline{\mathcal{U}}$ to perform symbolic composition is captured by the following theorem:

Theorem 5.7. *For all $U_1, U_2 \in$ FOUpdate and $\iota \in$ LogicalStruct, $\mathcal{U}\llbracket \overline{\mathcal{U}}\llbracket U_2 \rrbracket U_1 \rrbracket \iota = \mathcal{U}\llbracket U_2 \rrbracket (\mathcal{U}\llbracket U_1 \rrbracket \iota)$.*

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.

$$
\begin{array}{l}
\mathcal{U}\llbracket\overline{\mathcal{U}}\llbracket U_2\rrbracket U_1\rrbracket\iota \\
\quad = \mathcal{U}\llbracket((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}\llbracket T_i\rrbracket U_1], (U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}\llbracket FE_j\rrbracket U_1])\rrbracket\iota \\
\quad = \mathcal{U}\left\llbracket\begin{pmatrix}\{I_k \mapsto ((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}\llbracket T_i\rrbracket U_1])I_k\}, \\ \{F_m \mapsto ((U_1{\uparrow}2)[F_j \mapsto \overline{\mathcal{FE}}\llbracket FE_j\rrbracket U_1])F_m\}\end{pmatrix}\right\rrbracket\iota \\
\quad = ((\iota{\uparrow}1)[I_k \mapsto \mathcal{T}\llbracket((U_1{\uparrow}1)[I_i \mapsto \overline{\mathcal{T}}\llbracket T_i\rrbracket U_1])I_k\rrbracket\iota], \\
\qquad (\iota{\uparrow}2)[F_m \mapsto \mathcal{FE}\llbracket((U_2{\uparrow}1)[F_j \mapsto \overline{\mathcal{FE}}\llbracket FE_j\rrbracket U_1])F_m\rrbracket\iota]) \\
\quad = ((\iota{\uparrow}1)[I_k \mapsto \mathcal{T}\llbracket(U_1{\uparrow}1)I_k\rrbracket\iota][I_i \mapsto \mathcal{T}\llbracket\overline{\mathcal{T}}\llbracket T_i\rrbracket U_1\rrbracket\iota], \\
\qquad (\iota{\uparrow}2)[F_m \mapsto \mathcal{FE}\llbracket(U_1{\uparrow}2)F_m\rrbracket\iota][F_j \mapsto \mathcal{FE}\llbracket\overline{\mathcal{FE}}\llbracket FE_j\rrbracket U_1\rrbracket\iota) \\
\quad = \ \text{// by Lem. 5.5} \\
\quad\ \ ((\iota{\uparrow}1)[I_k \mapsto \mathcal{T}\llbracket(U_1{\uparrow}1)I_k\rrbracket\iota][I_i \mapsto \mathcal{T}\llbracket T_i\rrbracket(\mathcal{U}\llbracket U_1\rrbracket\iota)], \\
\qquad (\iota{\uparrow}2)[F_m \mapsto \mathcal{FE}\llbracket(U_1{\uparrow}2)F_m\rrbracket\iota][F_j \mapsto \mathcal{FE}\llbracket FE_j\rrbracket(\mathcal{U}\llbracket U_1\rrbracket\iota)]) \\
\quad = (((\mathcal{U}\llbracket U_1\rrbracket\iota){\uparrow}1)[I_i \mapsto \mathcal{T}\llbracket T_i\rrbracket(\mathcal{U}\llbracket U_1\rrbracket\iota)], \\
\qquad ((\mathcal{U}\llbracket U_1\rrbracket\iota){\uparrow}2)[F_j \mapsto \mathcal{FE}\llbracket FE_j\rrbracket(\mathcal{U}\llbracket U_1\rrbracket\iota)]) \\
\quad = \mathcal{U}\llbracket U_2\rrbracket(\mathcal{U}\llbracket U_1\rrbracket\iota)
\end{array}
$$

□

6. Reinterpretation for a Language with Pointers

In this section, we extend our subject language to have pointer variables, and show how reinterpretation of the factored semantics provides an easy way to deal with aliasing issues that arise with such a language. In particular, §6.4.1 and §6.4.2 discuss two different reinterpretations, one that automatically carries out the actions of Morris's rule of substitution [22] (see Ex. 6.2), and one that automatically carries out those of Cartwright and Oppen's rule (see Ex. 6.3).

6.1 PL$_2$: PL$_1$ Extended with Pointers

PL$_2$ is PL$_1$ extended with an address-generation expression, a dereferencing expression, and an indirect-assignment statement. The syntax of PL$_2$ is defined below, with changes from PL$_1$ highlighted in bold:

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

$$
\begin{array}{rcl}
c & ::= & 0 \mid 1 \mid \ldots \\
E & ::= & c \mid I \mid \boldsymbol{\&I} \mid \boldsymbol{*E} \mid E_1\ \mathit{op2}\ E_2 \mid \mathit{BE}\ ?\ E_1 : E_2 \\
\mathit{BE} & ::= & \mathbb{T} \mid \mathbb{F} \mid E_1\ \mathit{rop}\ E_2 \mid \neg \mathit{BE}_1 \mid \mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2 \\
S & ::= & I = E; \mid \boldsymbol{*I = E;} \mid S_1\ S_2
\end{array}
$$

6.2 Semantics of PL$_2$

The semantics for PL$_2$ is given in Fig. 6. The semantic domain *Loc* stands for *locations* (or memory addresses). We identify *Loc* with the set *Val* of values. (Note that we do not worry about type-checking issues in this paper, and locations can be freely manipulated as values.) In contrast with PL$_1$, where a state $\sigma \in \mathit{State}$ is a map $\mathit{Id} \to \mathit{Val}$, σ is now a pair (η, ρ), where, in the concrete semantics, *environment* $\eta \in \mathit{Env} = \mathit{Id} \to \mathit{Loc}$ maps identifiers to their associated locations and *store* $\rho \in \mathit{Store} = \mathit{Loc} \to \mathit{Val}$ maps each location to the value that it holds.

The standard interpretation of the operators used in the PL$_2$ semantics is as follows:

$$
\begin{array}{rcl}
\mathit{BVal}_{\mathit{std}} & = & \mathit{BVal} \\
\mathit{Val}_{\mathit{std}} & = & \mathit{Int32} \\
\mathit{Loc}_{\mathit{std}} & = & \mathit{Int32}
\end{array}
\qquad
\begin{array}{rcl}
\eta \in \mathit{Env}_{\mathit{std}} & = & \mathit{Id} \to \mathit{Loc}_{\mathit{std}} \\
\rho \in \mathit{Store}_{\mathit{std}} & = & \mathit{Loc}_{\mathit{std}} \to \mathit{Val}_{\mathit{std}}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{lookupState}_{\mathit{std}} & = & \lambda(\eta,\rho).\lambda I.\rho(\eta(I)) \\
\mathit{lookupEnv}_{\mathit{std}} & = & \lambda(\eta,\rho).\lambda I.\eta(I) \\
\mathit{lookupStore}_{\mathit{std}} & = & \lambda(\eta,\rho).\lambda l.\rho(l) \\
\mathit{updateStore}_{\mathit{std}} & = & \lambda(\eta,\rho).\lambda l.\lambda v.(\eta, \rho[l \mapsto v])
\end{array}
$$

$$v \in \mathit{Val}, l \in \mathit{Loc} = \mathit{Val}, \sigma \in \mathit{State} = \mathit{Store} \times \mathit{Env}$$

$$
\begin{array}{rcl}
\mathit{const} & : & C_{\mathit{Int32}} \to \mathit{Val} \\
\mathit{lookupState} & : & \mathit{State} \to \mathit{Id} \to \mathit{Val} \\
\mathit{lookupEnv} & : & \mathit{State} \to \mathit{Id} \to \mathit{Loc} \\
\mathit{lookupStore} & : & \mathit{State} \to \mathit{Loc} \to \mathit{Val} \\
\mathit{updateStore} & : & \mathit{State} \to \mathit{Loc} \to \mathit{Val} \to \mathit{State} \\
 & & \\
\mathcal{E} : \mathit{Expr} & \to & \mathit{State} \to \mathit{Val} \\
\mathcal{E}\llbracket c\rrbracket\sigma & = & \mathit{const}(c) \\
\mathcal{E}\llbracket I\rrbracket\sigma & = & \mathit{lookupState}\ \sigma\ I \\
\mathcal{E}\llbracket \&I\rrbracket\sigma & = & \mathit{lookupEnv}\ \sigma\ I \\
\mathcal{E}\llbracket *E\rrbracket\sigma & = & \mathit{lookupStore}\ \sigma\ (\mathcal{E}\llbracket E\rrbracket\sigma) \\
\mathcal{E}\llbracket E_1\ \mathit{op2}\ E_2\rrbracket\sigma & = & \mathcal{E}\llbracket E_1\rrbracket\sigma\ \mathit{binop}(\mathit{op2})\ \mathcal{E}\llbracket E_2\rrbracket\sigma \\
\mathcal{E}\llbracket \mathit{BE}\ ?\ E_1 : E_2\rrbracket\sigma & = & \mathcal{B}\llbracket \mathit{BE}\rrbracket\sigma\ ?\ \mathcal{E}\llbracket E_1\rrbracket\sigma : \mathcal{E}\llbracket E_2\rrbracket\sigma \\
 & & \\
\mathcal{B} : \mathit{BoolExpr} & \to & \mathit{State} \to \mathit{BVal} \\
\mathcal{B}\llbracket \mathbb{T}\rrbracket\sigma & = & \mathbb{T} \\
\mathcal{B}\llbracket \mathbb{F}\rrbracket\sigma & = & \mathbb{F} \\
\mathcal{B}\llbracket E_1\ \mathit{rop}\ E_2\rrbracket\sigma & = & \mathcal{E}\llbracket E_1\rrbracket\sigma\ \mathit{relop}(\mathit{rop})\ \mathcal{E}\llbracket E_2\rrbracket\sigma \\
\mathcal{B}\llbracket \neg \mathit{BE}_1\rrbracket\sigma & = & \neg\mathcal{B}\llbracket \mathit{BE}_1\rrbracket\sigma \\
\mathcal{B}\llbracket \mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2\rrbracket\sigma & = & \mathcal{B}\llbracket \mathit{BE}_1\rrbracket\sigma\ \mathit{boolop}(\mathit{bop})\ \mathcal{B}\llbracket \mathit{BE}_2\rrbracket\sigma \\
 & & \\
\mathcal{I} : \mathit{Stmt} & \to & \mathit{State} \to \mathit{State} \\
\mathcal{I}\llbracket I = E;\rrbracket\sigma & = & \mathit{updateStore}\ \sigma\ (\mathit{lookupEnv}\ \sigma\ I)\ (\mathcal{E}\llbracket E\rrbracket\sigma) \\
\mathcal{I}\llbracket *I = E;\rrbracket\sigma & = & \mathit{updateStore}\ \sigma\ (\mathcal{E}\llbracket I\rrbracket\sigma)\ (\mathcal{E}\llbracket E\rrbracket\sigma) \\
\mathcal{I}\llbracket S_1\ S_2\rrbracket\sigma & = & \mathcal{I}\llbracket S_2\rrbracket(\mathcal{I}\llbracket S_1\rrbracket\sigma)
\end{array}
$$

Figure 6. The factored semantics of PL$_2$.

6.3 Reinterpretation and Symbolic Execution for PL$_2$

The reinterpretation used for symbolic execution is similar to the one given in §3.2. Some of the reinterpreted operations from the core semantics of PL$_2$ are as follows:

$$
\begin{array}{lcl}
\overline{\mathit{lookupState}} & : & \mathit{FOUpdate} \to \mathit{Id} \to \mathit{Term} \\
\overline{\mathit{lookupState}} & = & \lambda U.\lambda I.((U{\uparrow}2)F_\rho)(I) \\
\overline{\mathit{updateStore}} & : & \mathit{FOUpdate} \to \mathit{Term} \to \mathit{Term} \to \mathit{FOUpdate} \\
\overline{\mathit{updateStore}} & &
\end{array}
$$

$$\quad = \lambda U.\lambda T_1.\lambda T_2.\begin{pmatrix}(U{\uparrow}1), \\ (U{\uparrow}2)[F_\rho \mapsto \overline{\mathit{update}}((U{\uparrow}2)F_\rho, T_1, T_2)]\end{pmatrix}$$

Example 6.1. The steps of symbolic execution of Fig. 1(b) via semantic reinterpretation, starting with an *FOUpdate* that corresponds to the third configuration of Fig. 3 are shown in Fig. 7. The *FOUpdate* shown in the last line of Fig. 7 can be considered to be the 2-vocabulary formula $F'_\rho = F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto v]$. This expresses a state change that does not usually perform a successful swap. □

Thm. 5.7 holds for PL$_2$, although it is now stated as follows:

Theorem 6.1. *For all $U_1, U_2 \in \mathit{FOUpdate}$ and $\iota \in \mathit{LogicalStruct}$, $\mathcal{U}\llbracket\overline{\mathcal{I}}\llbracket s\rrbracket U\rrbracket\iota = \mathcal{I}\llbracket s\rrbracket(\mathcal{U}\llbracket U\rrbracket\iota)$.*

6.4 Reinterpretation and WLP for PL$_2$

The semantic-reinterpretation approach provides insight on reconciling two different approaches that have been developed for expressing the weakest precondition of a formula with respect to a state transformation. On the one hand, one has Morris's rule of substitution [22], which generalizes Hoare's axiom of assignment [15] to a language with pointer variables by explicitly considering possible aliasing combinations. On the other hand, one has the approach of Cartwright and Oppen [6], the essence of which is to use a referentially transparent meta-language; this permits pre-state properties to be expressed using *only* formula substitution, even in the presence of aliasing.

Although these approaches do not seem to be connected, the semantic-reinterpretation approach provides an explanation of

$$
\begin{aligned}
\overline{\mathcal{I}}[\![*px = *px \boxed{\oplus} *py;]\!](\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto \mathit{py}]) &= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (F_\rho(\mathit{px}) \boxed{\oplus} F_\rho(\mathit{py}))]) \\
&= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (\mathit{py} \boxed{\oplus} \mathit{py})]) \\
&= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]) \\
\overline{\mathcal{I}}[\![*py = *px \boxed{\oplus} *py;]\!](\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]) &= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto (F_\rho(\mathit{py}) \boxed{\oplus} F_\rho(0))][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]) \\
&= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto (0 \boxed{\oplus} v)][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]) \\
&= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]) \\
\overline{\mathcal{I}}[\![*px = *px \boxed{\oplus} *py;]\!](\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto 0]) &= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (F_\rho(\mathit{py}) \boxed{\oplus} F_\rho(0))]) \\
&= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto (0 \boxed{\oplus} v)]) \\
&= (\emptyset, F_\rho \hookleftarrow F_\rho[0 \mapsto v][\mathit{px} \mapsto \mathit{py}][\mathit{py} \mapsto v])
\end{aligned}
$$

Figure 7. Symbolic execution of Fig. 1(b) via semantic reinterpretation, starting with an *FOUpdate* that corresponds to the third configuration in Fig. 3.

how they are related. In particular, the semantic-reinterpretation approach can actually achieve *both* methods—the difference is merely the degree of algebraic simplification that is performed.

It may not be obvious why this is so, particularly because we have never introduced an explicit operation of substitution for our logic. However, *a symbolic 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 $\overline{\mathit{Val}}$ is a *Term*—i.e., something *symbolic*—which is used in subsequent computations. Thus, when $\iota \in \mathit{LogicalStruct}$ is reinterpreted as $U \in \mathit{FOUpdate}$, the reinterpretation of formula φ via $\overline{\mathcal{F}}[\![\varphi]\!]U$ substitutes *Terms* found in U into φ: $\overline{\mathcal{F}}[\![\varphi]\!]U$ calls $\overline{\mathcal{T}}[\![T]\!]U$, which may call $\overline{\mathit{lookupId}}\ U\ I$; 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 understand how pointers, aliasing, and dereferencing are handled during the $\mathcal{WLP}$ operation, the key reinterpretations to concentrate on are the ones for the operations of the semantic core of $L[\mathrm{PL}_2]$ that manipulate *FVal*s (i.e., arguments of type $\mathit{Val} \to \mathit{Val}$)—in particular, *access* and *update*.

We want $\overline{\mathit{access}}$ and $\overline{\mathit{update}}$ to enjoy the following semantic properties:

$$
\begin{aligned}
\mathcal{T}[\![\overline{\mathit{access}}(\mathit{FE}_0, T_0)]\!]\iota &= (\mathcal{FE}[\![\mathit{FE}_0]\!]\iota)(\mathcal{T}[\![T_0]\!]\iota) \\
\mathcal{T}[\![\overline{\mathit{update}}(\mathit{FE}_0, T_0, T_1)]\!]\iota &= (\mathcal{FE}[\![\mathit{FE}_0]\!]\iota)[\mathcal{T}[\![T_0]\!]\iota \mapsto \mathcal{T}[\![T_1]\!]\iota]
\end{aligned}
$$

Note that these properties require evaluating an $\overline{\mathit{access}}$ or $\overline{\mathit{update}}$ expression with respect to an arbitrary $\iota \in \mathit{LogicalStruct}$. As discussed in §3, it can be desirable for reinterpreted base-type operations to perform simplifications whenever possible, when they construct *Terms*, *Formulas*, *FuncExprs*, and *FOUpdates*. However, because the value of ι is unknown, $\overline{\mathit{access}}$ and $\overline{\mathit{update}}$ operate in an uncertain environment. As shown below, the essential difference between Morris's rule and Cartwright and Oppen's rule is the degree of algebraic simplification attempted in the absence of information about ι.

§6.4.1 shows how reinterpretation automatically generates a weakest-precondition primitive that implements Morris's rule; §6.4.2 shows how the semantic-reinterpretation approach can also generate a weakest-precondition primitive that implements Cartwright and Oppen's rule. Thm. 5.6 holds for PL_2 with both $\mathcal{WLP}$ primitives, although the proofs are more involved than the one given for PL_1 in §5.2.

6.4.1 Reinterpretation and WLP à la Morris

To use semantic reinterpretation to create a weakest-precondition primitive that implements Morris's rule, simplifications are performed during $\overline{\mathit{access}}$ and $\overline{\mathit{update}}$ using the rewriting rules given below, where $\equiv$, $\not\equiv$, and $\doteq$ denote *equality-as-terms*, *definite-disequality*, and *possible-equality*, respectively.

$$
\begin{aligned}
&\overline{\mathit{access}}(a_1, k_1):\\
&(a_1 \equiv F) &&\Longrightarrow F(k_1) \\
&(a_1 \equiv a_2[k_2 \mapsto d_2]) \wedge (k_1 \equiv k_2) &&\Longrightarrow d_2 \\
&(a_1 \equiv a_2[k_2 \mapsto d_2]) \wedge (k_1 \not\equiv k_2) &&\Longrightarrow \overline{\mathit{access}}(a_2, k_1) \\
&(a_1 \equiv a_2[k_2 \mapsto d_2]) \wedge (k_1 \doteq k_2) && \\
&&&\Longrightarrow \mathit{ite}(k_1 \boxed{=} k_2, d_2, \overline{\mathit{access}}(a_2, k_1))
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathit{update}}(a_1, k_1, d_1):\\
&(a_1 \equiv F) &&\Longrightarrow F[k_1 \mapsto d_1] \\
&(a_1 \equiv a_2[k_2 \mapsto d_2]) \wedge (k_1 \equiv k_2) &&\Longrightarrow a_2[k_1 \mapsto d_1] \\
&(a_1 \equiv a_2[k_2 \mapsto d_2]) \wedge (k_1 \not\equiv k_2) &&\Longrightarrow \overline{\mathit{update}}(a_2, k_1, d_1)[k_2 \mapsto d_2] \\
&(a_1 \equiv a_2[k_2 \mapsto d_2]) \wedge (k_1 \doteq k_2) &&\Longrightarrow a_1[k_1 \mapsto d_1]
\end{aligned}
$$

In particular, the rules for $\overline{\mathit{access}}$ that involve *possible-equality* comparisons cause *ite* terms to arise. As illustrated in Ex. 6.2, 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 [22].

Example 6.2. We now demonstrate how semantic reinterpretation arrives at the weakest-precondition formula for $\mathcal{WLP}(*p = e, x = 5)$ that was claimed in Ex. 3.3. (The definition of $\overline{\mathit{lookupState}}$ and $\overline{\mathit{updateStore}}$ were given in §6.3.)

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

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

Note how the formula-simplification rules of $\overline{\mathit{access}}$ that involve *possible-equality* comparisons causes an *ite* term to arise that tests the condition "$F_\rho(p) \boxed{=} x$". The test determines whether the value of p is an alias for the address of x, which, as discussed in §3.3, is the only aliasing combination that matters for this example. □

6.4.2 Reinterpretation and WLP à la Cartwright and Oppen

§6.4.1 gave one version of $\overline{\mathit{access}}$ and $\overline{\mathit{update}}$, where $\overline{\mathit{access}}$ performs simplification using a set of rules that can introduce *ite* terms.

A less ambitious $\overline{\mathit{access}}$ just creates a residual *Term*:

$$\overline{\mathit{access}}(a,k) : a(k) \;\; \text{// constructs a } \mathit{Term} \qquad (4)$$

Similarly $\overline{\mathit{update}}$ just creates a residual *FuncExpr*:

$$\overline{\mathit{update}}(a,k,d) : a[k \mapsto d] \;\; \text{// constructs a } \mathit{FuncExpr} \qquad (5)$$

When these definitions are used in the context of the method given in §5.2 for computing a weakest-precondition formula via semantic reinterpretation, i.e., $\mathcal{WLP}(s,\varphi) = \overline{\mathcal{F}}[\![\varphi]\!](\overline{\mathcal{I}}[\![s]\!]U_{id})$, the $\mathcal{WLP}$ operation performs substitution (and no simplification), and the result is the same as the one obtained using Cartwright and Oppen's method.

Example 6.3. When $\overline{\mathit{access}}$ is defined as in Eqn. (5), the second half of Ex. 6.2 changes as follows:

$$\begin{aligned} &\mathcal{WLP}(*p = e, F_\rho(x) \boxed{=} 5) \\ &\quad = \overline{\mathcal{F}}[\![F_\rho(x) \boxed{=} 5]\!]U \\ &\quad = \ldots \;\; \text{// same as Ex. 6.2} \\ &\quad = \overline{\mathit{access}}(F_\rho[F_\rho(p) \mapsto F_\rho(e)], x) \boxed{=} 5 \\ &\quad = F_\rho[F_\rho(p) \mapsto F_\rho(e)](x) \boxed{=} 5 \end{aligned}$$

□

7. Reinterpretation of Machine-Code Semantics

In this section, we discuss how to apply the reinterpretation technique at the machine-code level.

7.1 MC : A Simple Machine-Code Language

We first define a simple machine-code language MC; it is based on the x86 instruction set, but greatly simplified.

$$\begin{aligned} r \in \mathit{reg} &:= \mathit{EAX} \mid \mathit{EBP} \mid \mathit{EIP} \\ f \in \mathit{flag} &:= \mathit{ZF} \\ do \in \mathit{dst\_operand} &:= \mathit{Indirect}(\mathit{reg}\ \mathit{Val}) \mid \mathit{DirectReg}(\mathit{reg}) \\ so \in \mathit{src\_operand} &:= \mathit{dst\_operand} \cup \mathit{Immediate}(\mathit{Val}) \\ o \in \mathit{operand} &:= \mathit{src\_operand} \\ \mathit{Instr} \in \mathit{instruction} &:= \mathit{MOV}(\mathit{dst\_operand}\ \mathit{src\_operand}) \\ &\mid\ \mathit{CMP}(\mathit{dst\_operand}\ \mathit{src\_operand}) \\ &\mid\ \mathit{XOR}(\mathit{dst\_operand}\ \mathit{src\_operand}) \\ &\mid\ \mathit{JZ}(\mathit{dst\_operand}) \end{aligned}$$

MC has 3 registers and one flag. There are 3 kinds of operands and 4 instructions.

7.2 Semantics of MC

The informal semantics of MC is as follows: *MOV* moves the value of the source operand to the destination operand. *CMP* sets *ZF* according to the difference of the values from the two operands. *XOR* computes the exclusive-or of the values from the two operands and stores the result to the destination operand. Each instruction increments the program-counter register *EIP*. *JZ* updates *EIP* depending on the value of flag *ZF*.

A formal semantics for MC is given in Fig. 8. In contrast with PL$_1$ and PL$_2$, in MC a state σ is a triple $(\mathit{mem}, \mathit{reg}, \mathit{flag})$. $\mathcal{R}$, $\mathcal{K}$, $\mathcal{O}$, and $\mathcal{I}$ are the evaluation functions for *reg*, *flag*, *operand*, and MC, respectively.

7.3 Reinterpretation of MC

Semantic reinterpretation works similarly to what was done for PL$_1$, PL$_2$, and L. The base types are redefined as follows:

$$\overline{\mathit{BVal}} = \mathit{Formula} \quad \overline{\mathit{Val}} = \mathit{Term}$$
$$\overline{\mathit{State}} = \mathit{FOUpdate} = (\{\mathit{ZF}, \mathit{EAX}, \mathit{EBP}, \mathit{EIP}\}, \{F_{\mathit{mem}}\})$$

Example 7.1. Fig. 9 shows the assembly code that corresponds to the swap code in Fig. 1(a). lines 1–3, lines 4–6, and lines 7–9 correspond to line 1, line 2, and line 3 in Fig. 1(a), respectively.

$$\begin{aligned} \mathit{const} &: C_{\mathit{Int32}} \rightarrow \mathit{Val} \\ \mathit{store}_{\mathit{reg}} &: \mathit{State} \rightarrow \mathit{reg} \rightarrow \mathit{Val} \rightarrow \mathit{State} \\ \mathit{lookup}_{\mathit{reg}} &: \mathit{State} \rightarrow \mathit{reg} \rightarrow \mathit{Val} \\ \mathit{store}_{\mathit{flag}} &: \mathit{State} \rightarrow \mathit{flag} \rightarrow \mathit{BVal} \rightarrow \mathit{State} \\ \mathit{lookup}_{\mathit{flag}} &: \mathit{State} \rightarrow \mathit{flag} \rightarrow \mathit{BVal} \\ \mathit{store}_{\mathit{mem}} &: \mathit{State} \rightarrow \mathit{Val} \rightarrow \mathit{Val} \rightarrow \mathit{State} \\ \mathit{lookup}_{\mathit{mem}} &: \mathit{State} \rightarrow \mathit{Val} \rightarrow \mathit{Val} \\ \mathit{store}_{\mathit{eip}} &: \mathit{State} \rightarrow \mathit{State} \\ \mathit{store}_{\mathit{eip}} &= \lambda\sigma.\mathit{store}_{\mathit{reg}}(\sigma, \mathit{EIP}, \mathcal{R}[\![\mathit{EIP}]\!]\sigma + 4) \end{aligned}$$

$$\begin{aligned} \mathcal{R} : \mathit{reg} \rightarrow \mathit{State} \rightarrow \mathit{Val}, &\quad \mathcal{R}[\![r]\!]\sigma = \mathit{lookup}_{\mathit{reg}}(\sigma, r) \\ \mathcal{K} : \mathit{flag} \rightarrow \mathit{State} \rightarrow \mathit{BVal}, &\quad \mathcal{K}[\![f]\!]\sigma = \mathit{lookup}_{\mathit{flag}}(\sigma, f) \end{aligned}$$

$$\begin{aligned} \mathcal{O} : \mathit{operand} &\rightarrow \mathit{State} \rightarrow \mathit{Val} \\ \mathcal{O}[\![\mathit{Indirect}(r\ c)]\!]\sigma &= \mathit{lookup}_{\mathit{mem}}(\sigma, \mathcal{R}[\![r]\!]\sigma + \mathit{const}(c)) \\ \mathcal{O}[\![\mathit{DirectReg}(r)]\!]\sigma &= \mathcal{R}[\![r]\!]\sigma \\ \mathcal{O}[\![\mathit{Immediate}(c)]\!]\sigma &= \mathit{const}(c) \end{aligned}$$

$$\begin{aligned} &\mathcal{I} : \mathit{instruction} \rightarrow \mathit{State} \rightarrow \mathit{State} \\ &\mathcal{I}[\![\mathit{MOV}(\mathit{Indirect}(r\ c)\ so)]\!]\sigma \\ &\quad = \mathit{store}_{\mathit{eip}}(\mathit{store}_{\mathit{mem}}(\sigma, \mathcal{R}[\![r]\!]\sigma + \mathit{const}(c), \mathcal{O}[\![so]\!]\sigma)) \\ &\mathcal{I}[\![\mathit{MOV}(\mathit{DirectReg}(r)\ so)]\!]\sigma = \mathit{store}_{\mathit{eip}}(\mathit{store}_{\mathit{reg}}(\sigma, r, \mathcal{O}[\![so]\!]\sigma)) \\ &\mathcal{I}[\![\mathit{CMP}(do\ so)]\!]\sigma = \mathit{store}_{\mathit{eip}}(\mathit{store}_{\mathit{flag}}(\sigma, \mathit{ZF}, \mathcal{O}[\![do]\!]\sigma - \mathcal{O}[\![so]\!]\sigma = 0)) \\ &\mathcal{I}[\![\mathit{XOR}(\mathit{Indirect}(r\ c)\ so)]\!]\sigma = \mathcal{I}[\![\mathit{XOR}(do\ so)]\!]\sigma \\ &\quad = \mathit{store}_{\mathit{eip}}(\mathit{store}_{\mathit{mem}}(\sigma, \mathcal{R}[\![r]\!]\sigma + \mathit{const}(c), \mathcal{O}[\![do]\!]\sigma \oplus \mathcal{O}[\![so]\!]\sigma)) \\ &\mathcal{I}[\![\mathit{XOR}(\mathit{DirectReg}(r)\ so)]\!]\sigma = \mathcal{I}[\![\mathit{XOR}(do\ so)]\!]\sigma \\ &\quad = \mathit{store}_{\mathit{eip}}(\mathit{store}_{\mathit{reg}}(\sigma, r, \mathcal{O}[\![do]\!]\sigma \oplus \mathcal{O}[\![so]\!]\sigma)) \\ &\mathcal{I}[\![\mathit{JZ}(do)]\!]\sigma = \mathit{store}_{\mathit{reg}}(\sigma, \mathit{EIP}, \mathcal{K}[\![\mathit{ZF}]\!]\sigma\ ?\ \mathcal{R}[\![\mathit{EIP}]\!]\sigma + 4 : \mathcal{O}[\![do]\!]\sigma) \end{aligned}$$

Figure 8. The factored semantics of MC.

```
[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
```

Figure 9. The assembly code corresponding to Fig. 1(a).

For the swap assembly code in Fig. 9, $\overline{\mathcal{I}}(\mathit{swap}, U_{id})$ produces the following *FOUpdate*.

$$\begin{aligned} U = (\mathit{EAX}' &\hookleftarrow F_{\mathit{mem}}(\mathit{EBP} \boxed{-} 14), \\ F'_{\mathit{mem}} &\hookleftarrow F_{\mathit{mem}}[\mathit{EBP} \boxed{-} 10 \mapsto F_{\mathit{mem}}(\mathit{EBP} \boxed{-} 14)] \\ &\qquad\quad [\mathit{EBP} \boxed{-} 14 \mapsto F_{\mathit{mem}}(\mathit{EBP} \boxed{-} 10)]) \end{aligned}$$

□

8. Implementation

We have implemented the approach to creating symbolic-analysis primitives described in the paper using the TSL system [20]. The implementation has been used to generate primitives for forward symbolic evaluation, weakest precondition, and symbolic composition for multiple machine-code instruction sets.

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. Much of what a TSL user writes when developing an instruction-set specification is similar to writing an interpreter for an instruction set in first-order ML. That is, the meaning function $\mathcal{I}$ of §7.2 is written as a TSL function

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

where `instruction` and `state` are user-defined data types that represent the syntactic objects (i.e., instructions, statements, expressions, or formulas) and the semantic states, respectively.

To implement the work described in this paper, we used TSL to create semantic reinterpretations that are based on logical formulas. We specified both the syntax and semantics of logic $L[\cdot]$ in TSL—the latter involved writing functions in TSL that correspond to $\mathcal{T}$, $\mathcal{F}$, etc.—and then reinterpreted the semantic core of $L[\cdot]$, as described in §3.3 and §5.2.

These reinterpretations were applied to two TSL instruction-set specifications that we had on hand from our work on generating abstract interpretations [20]. The specification of the Intel x86 instruction set is about 2700 lines of TSL; the specification of the PowerPC instruction set is about 1200 lines. Using TSL, we obtained automatically-generated implementations of all three symbolic-analysis functions from each of the specifications.

Moreover, each of the reinterpretations can be reused. In TSL, reinterpretation is performed at the meta-level: the set of TSL primitive operations on base types forms the semantic core of all languages specified using TSL. An analysis designer adds a new analysis component to the TSL system by (i) redefining the TSL base types (e.g., *INT32*, *INT8*, *BOOL*, etc.), and (ii) providing a set of alternative interpretations for the primitive operations on base types (e.g., $+_{INT32}$, $+_{INT8}$, etc.). This implicitly defines an alternative interpretation of each expression and function in an instruction-set's standard semantics (including `interpInstr`). Consequently, implementations of all *three* symbolic-analysis functions can be generated for the next instruction set of interest, say *IS*, merely by writing a TSL specification of the standard semantics of *IS*.

8.1 Binding-Time Analysis and 2-Level Semantics

As mentioned earlier, in §3.2, one of the key techniques that we use is related to partial evaluation. In essence, we 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.

TSL is not a partial-evaluation system *per se*; however, for reasons discussed in [19, §3.4], the TSL compiler performs binding-time analysis [16], and annotates the code for `interpInstr` to create an intermediate representation in a two-level language [25]. 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 [16] `interpInstr-gen` for `interpInstr` (see footnote 7). 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:

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

The difference between `interpInstr-gen` and `interpInstr-CH` is very small: `interpInstr-CH` still requires *two* inputs to be supplied (but we can use the trivial value U_{id} for the second input).

9. Related Work

Forward symbolic evaluation. Symbolic execution has been employed in many recently developed systems for program testing and verification. In particular, hybrid concrete/symbolic execution tools, which are able to generate inputs that increase test coverage, start with the path formula for an executable path π, change the formula to be one for a nearby path π' that follows the same sequence of edges 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'. Recently these techniques have been employed on x86 executables in the SAGE [11] and BITSCOPE [5] tools.

Our work provides a way to create the core primitives of such systems automatically. It would allow one to easily build versions of such tools that can be applied to other instruction sets. However, there is a significant difference between the approach that we use and the way symbolic-analysis primitives are implemented in existing tools.

- In existing tools, the semantics of the subject language is encapsulated in a *translation procedure* that translates the subject-language instructions into a form more suitable for symbolic manipulation.
- We use a *declarative approach*: the tool writer provides a specification of the subject language's standard semantics, in the form of an interpreter expressed in a functional language. Our system then applies semantic reinterpretation to the semantic core, using the methods explained in the paper.

  Our approach does not need any of the preprocessing steps that are sometimes performed on the subject program, such as converting the program into a single assignment form and translating it into an intermediate form on which symbolic operations are carried out [4].

Weakest liberal precondition. An intriguing aspect of the semantic-reinterpretation approach is that it provides insight on reconciling two different approaches that have been developed for expressing the weakest precondition of a formula with respect to a state transformation, for languages with pointer variables and aliasing: (i) Morris's rule of substitution [22], which explicitly considers possible aliasing combinations, and (ii) the pure substitution-based approach of Cartwright and Oppen [6]. §6.4.1 shows how reinterpretation can automatically generate a weakest-precondition primitive that implements Morris's rule; §6.4.2 shows how the semantic-reinterpretation approach can also generate a weakest-precondition primitive that implements Cartwright and Oppen's rule. This provides an account of how Morris's rule and Cartwright and Oppen's rule are related: both are based on substitution; the difference is merely the degree of algebraic simplification that is performed on *Term*s that express function accesses.

In particular, in §6.4.1 the rules for $\overline{\textit{access}}$ that involve *possible-equality* comparisons cause *ite* terms to arise. As illustrated in Ex. 6.2, it is these *ite* terms that cause the reinterpreted operations to account for possible aliasing combinations, and are the reason that the semantic reinterpretation from §6.4.1 carries out the actions of Morris's rule. In contrast, the simpler rules used in §6.4.2 cause semantic reinterpretation to implement the pure substitution-based approach of Cartwright and Oppen.

Symbolic composition. Symbolic composition arises in many computational contexts. It provides one way to addresses the problem of "dissolving" module boundaries in software systems for the sake of run-time efficiency. In the context of imperative programs, in-line expansion (followed by simplification/optimization) can be thought of as a kind of symbolic composition. However, the techniques used to perform symbolic composition are often a good deal more sophisticated than what is obtained by simple in-line expansion. For example, *deforestation* is a particular kind of symbolic composition relevant when the producer function f creates a tree-structured output that is consumed by g [30]. The idea behind deforestation is to transform the program so that the intermediate tree structure is never constructed. Similarly, *filter fusion* [26] looks for situations in which `read` operations cancel with `write` operations.

Our work addresses symbolic composition of logic formulas. As explained in §1, this operation is useful when a tool has access to a formula that summarizes a called procedure's behavior. Re-exploration of the procedure can be avoided by symbolically composing a path formula with the procedure-summary formula. The potential gain in efficiency comes from cancellations of *update*s and *access*s, as well as simplification of *update*s and subsuming *update*s.

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] L. Birkedal and M. Welinder. Hand-writing program generator generators. In *PLILP*, 1994.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Sec. Symp.*, Aug. 2007.

[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] R. Cartwright and D. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.

[7] P. Cousot and R. Cousot. Abstract interpretation. In *POPL*, 1977.

[8] Coverity, Inc. Coverity Prevent. `www.coverity.com/html/coverity-prevent.html`.

[9] V. Ganesh and D. Dill. A decision procesure for bit-vectors and arrays. In *CAV*, 2007.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[11] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[12] GrammaTech, Inc. CodeSonar. `www.grammatech.com/products/codesonar`.

[13] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: A new algorithm for property checking. In *FSE*, 2006.

[14] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[15] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, Oct. 1969.

[16] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[17] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag, 2008.

[18] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI*, 1996.

[19] 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.

[20] J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.

[21] K. Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms*. PhD thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas, 1993.

[22] J. Morris. A general axiom of assignment. In M. Broy and G. Schmidt, editors, *Theor. Found. of Program. Methodology, Proc. of the 1981 Marktoberdorf Summer School*, pages 25–34. Reidel, 1982.

[23] A. Mycroft and N. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects*, 1985.

[24] F. Nielson. Two-level semantics and abstract interpretation. *TCS*, 69:117–242, 1989.

[25] F. Nielson and H. Nielson. *Two-Level Functional Languages*. Cambridge Univ. Press, 1992.

[26] T. Proebsting and S. Watterson. Filter fusion. In *POPL*, 1996.

[27] D. Schmidt. *Denotational Semantics*. Allyn and Bacon, Inc., Boston, MA, 1986.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *FSE*, 2005.

[29] A. Stump, C. Barrett, and D. Dill. A decision procedure for an extensional theory of arrays. In *LICS*, 2001.

[30] P. Wadler. Deforestation: Transforming programs to eliminate trees. *TCS*, 73:231–248, 1990.

[31] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using Boolean satisfiability. *TOPLAS*, 29(3), 2007.

[32] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, 2003.

A. Appendix

Lemma 5.1.

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

□

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

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 $\mathit{binop}_L(\mathit{op2}_L) \rightsquigarrow \mathit{binop}(\mathit{op2})$ are shorthands for reasoning about each case, such as $\mathit{binop}_L(\boxed{+}) \rightsquigarrow \mathit{binop}(+)$, etc.

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

$$\begin{aligned}
&(ii) \\
\text{lhs}\ \ :\ \ & \mathcal{T}[\![\overline{\mathcal{E}}[\![I]\!]U]\!]\iota = \mathcal{T}[\![\overline{\mathit{lookup}}\ U\ I]\!]\iota = \mathcal{T}[\![(U\uparrow 1)I]\!]\iota \\
\text{rhs}\ \ :\ \ & \mathcal{E}[\![I]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1) \\
=\ \ & \mathcal{E}[\![I]\!]((\iota\uparrow 1)[I_i \mapsto \mathcal{T}[\![(U\uparrow 1)I_i]\!]\iota], (\iota\uparrow 2)[F_j \mapsto \mathcal{FE}[\![\mathit{FE}_j]\!]\iota) \\
=\ \ & \mathit{lookup}((\iota\uparrow 1)[I_i \mapsto \mathcal{T}[\![(U\uparrow 1)I_i]\!]\iota], (\iota\uparrow 2)[F_j \mapsto \mathcal{FE}[\![\mathit{FE}_j]\!]\iota)I \\
=\ \ & \mathcal{T}[\![(U\uparrow 1)I]\!]\iota
\end{aligned}$$

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

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

$$(2)\ (i)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathbb{T}]\!]U]\!]\iota = \mathcal{F}[\![\mathbb{T}]\!]\iota = \mathbb{T} = \mathcal{B}[\![\mathbb{T}]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1)$$

$$(ii)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathbb{F}]\!]U]\!]\iota = \mathcal{F}[\![\mathbb{F}]\!]\iota = \mathbb{F} = \mathcal{B}[\![\mathbb{F}]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1)$$

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

$$\begin{aligned}
&(iv)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\neg \mathit{BE}_1]\!]U]\!]\iota = \mathcal{F}[\![\boxed{\neg}\,\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U]\!]\iota = \neg\mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U]\!]\iota \\
&\quad = \neg\mathcal{B}[\![\mathit{BE}_1]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1)\ \text{// by ind. via (2)} \\
&\quad = \mathcal{B}[\![\neg \mathit{BE}_1]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1)
\end{aligned}$$

$$\begin{aligned}
&(v)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2]\!]U]\!]\iota \\
&\quad = \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U\ \mathit{bop}_L\ \overline{\mathcal{B}}[\![\mathit{BE}_2]\!]U]\!]\iota \\
&\quad = \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_1]\!]U]\!]\iota\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\overline{\mathcal{B}}[\![\mathit{BE}_2]\!]U]\!]\iota \\
&\quad = \text{// by ind. via (2)} \\
&\qquad \mathcal{B}[\![\mathit{BE}_1]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1)\ \mathit{boolop}(\mathit{bop})\ \mathcal{B}[\![\mathit{BE}_2]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1) \\
&\quad = \mathcal{B}[\![\mathit{BE}_1\ \mathit{bop}\ \mathit{BE}_2]\!]((\mathcal{U}[\![U]\!]\iota)\uparrow 1)
\end{aligned}$$

□

Lemma 5.5.

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

□

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

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

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

$$\begin{aligned}
&(iii)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1\ \mathit{op2}_L\ T_2]\!]U]\!]\iota \\
&\quad = \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U\ \mathit{op2}_L\ \overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota \\
&\quad = \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota \\
&\quad = \text{// by ind. via (1)} \\
&\qquad \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota)\ \mathit{binop}_L(\mathit{op2}_L)\ \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota) \\
&\quad = \mathcal{T}[\![T_1\ \mathit{op2}_L\ T_2]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}$$

$$\begin{aligned}
&(iv)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![\mathit{ite}(\varphi, T_1, T_2)]\!]U]\!]\iota \\
&\quad = \mathcal{T}[\![\mathit{ite}(\overline{\mathcal{F}}[\![\varphi]\!]U, \overline{\mathcal{T}}[\![T_1]\!]U, \overline{\mathcal{T}}[\![T_2]\!]U)]\!]\iota \\
&\quad = \mathit{cond}_L(\mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota, \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota) \\
&\quad = \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi]\!]U]\!]\iota\ ?\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota : \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota \\
&\quad = \text{// by ind. via (1) and (2)} \\
&\qquad \mathcal{F}[\![\varphi]\!](\mathcal{U}[\![U]\!]\iota)\ ?\ \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota) : \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota) \\
&\quad = \mathcal{F}[\![\varphi\ ?\ T_1 : T_2]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}$$

$$\begin{aligned}
&(v)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![\mathit{FE}(T)]\!]U]\!]\iota \\
&\quad = \mathcal{T}[\![\overline{\mathcal{FE}}[\![\mathit{FE}]\!]U(\overline{\mathcal{T}}[\![T]\!]U)]\!]\iota \\
&\quad = (\mathcal{FE}[\![\overline{\mathcal{FE}}[\![\mathit{FE}]\!]U]\!]\iota)(\mathcal{T}[\![\overline{\mathcal{T}}[\![T]\!]U]\!]\iota) \\
&\quad = (\mathcal{FE}[\![\mathit{FE}]\!](\mathcal{U}[\![U]\!]\iota))(\mathcal{T}[\![T]\!](\mathcal{U}[\![U]\!]\iota))\ \ \text{// by ind. via (3)} \\
&\quad = \mathcal{T}[\![\mathit{FE}(T)]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}$$

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

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

$$\begin{aligned}
&(iii)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![T_1\ \mathit{rop}_L\ T_2]\!]U]\!]\iota \\
&\quad = \mathcal{F}[\![\overline{\mathcal{T}}[\![T_1]\!]U\ \mathit{relop}_L(\mathit{rop}_L)\ \overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota \\
&\quad = \mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota \\
&\quad = \text{// by ind. via (1)} \\
&\qquad \mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota)\ \mathit{relop}_L(\mathit{rop}_L)\ \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota) \\
&\quad = \mathcal{F}[\![T_1\ \mathit{rop}_L\ T_2]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}$$

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

$$\begin{aligned}
&(v)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!]U]\!]\iota \\
&\quad = \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U\ \mathit{boolop}_L(\mathit{bop}_L)\ \overline{\mathcal{F}}[\![\varphi_2]\!]U]\!]\iota \\
&\quad = \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_1]\!]U]\!]\iota\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\overline{\mathcal{F}}[\![\varphi_2]\!]U]\!]\iota \\
&\quad = \text{// by ind. via (2)} \\
&\qquad \mathcal{F}[\![\varphi_1]\!](\mathcal{U}[\![U]\!]\iota)\ \mathit{boolop}_L(\mathit{bop}_L)\ \mathcal{F}[\![\varphi_2]\!](\mathcal{U}[\![U]\!]\iota) \\
&\quad = \mathcal{F}[\![\varphi_1\ \mathit{bop}_L\ \varphi_2]\!](\mathcal{U}[\![U]\!]\iota)
\end{aligned}$$

(3) (i)

$$\begin{array}{ll}
\text{lhs} & = \mathcal{FE}[\![\overline{\mathcal{FE}}[\![F]\!]U]\!]\iota = \mathcal{FE}[\![\overline{\textit{lookupId}}\ U\ F]\!]\iota = \mathcal{FE}[\![(U{\uparrow}2)F]\!]\iota \\
\text{rhs} & = \mathcal{FE}[\![F]\!](\mathcal{U}[\![U]\!]\iota) \\
& = \mathcal{FE}[\![F]\!]((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f) \\
& = \textit{lookupFuncId}\ ((\iota{\uparrow}1)[I_i \mapsto \mathcal{T}[\![T_i]\!]\iota], f)\ F \\
& = \mathcal{FE}[\![(U{\uparrow}2)F]\!]\iota
\end{array}$$

$$\begin{array}{l}
(ii)\ \mathcal{FE}[\![\overline{\mathcal{FE}}[\![\mathit{FE}_0[T_1 \mapsto T_2]]\!]U]\!]\iota \\
= \mathcal{FE}[\![(\overline{\mathcal{FE}}[\![\mathit{FE}_0]\!]U)[\overline{\mathcal{T}}[\![T_1]\!]U \mapsto \overline{\mathcal{T}}[\![T_2]\!]U]]\!]\iota \\
= \mathcal{FE}[\![(\overline{\mathcal{FE}}[\![\mathit{FE}_0]\!]U)]\!]\iota[\mathcal{T}[\![\overline{\mathcal{T}}[\![T_1]\!]U]\!]\iota \mapsto \mathcal{T}[\![\overline{\mathcal{T}}[\![T_2]\!]U]\!]\iota] \\
= //\ \text{by ind. via } (1) \\
\quad \mathcal{FE}[\![\mathit{FE}_0]\!](\mathcal{U}[\![U]\!]\iota)[\mathcal{T}[\![T_1]\!](\mathcal{U}[\![U]\!]\iota) \mapsto \mathcal{T}[\![T_2]\!](\mathcal{U}[\![U]\!]\iota)] \\
= \mathcal{FE}[\![\mathit{FE}_0[T_1 \mapsto T_2]]\!](\mathcal{U}[\![U]\!]\iota)
\end{array}$$

□