TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis

JUNGHEE LIM
GrammaTech, Inc.
and
THOMAS REPS
University of Wisconsin and GrammaTech, Inc.

This paper describes the design and implementation of a system, called TSL (for "Transformer Specification Language"), that provides a systematic solution to the problem of creating retargetable tools for analyzing machine code. TSL is a tool generator—i.e., a meta-tool—that automatically creates different abstract interpreters for machine-code instruction sets.

The most challenging technical issue that we faced in designing TSL was how to automate the generation of the set of *abstract transformers* for a given abstract interpretation of a given instruction set. From a description of the *concrete operational semantics* of an instruction set, together with the datatypes and operations that define an abstract domain, TSL automatically creates the set of abstract transformers for the instructions of the instruction set. TSL advances the state of the art in program analysis because it provides two dimensions of parameterizability: (i) a given analysis component can be retargeted to different instruction sets; (ii) multiple analysis components can be created automatically from a single specification of the concrete operational semantics of the language to be analyzed.

TSL is an *abstract-transformer-generator generator*. The paper describes the principles behind TSL, and discusses how one uses TSL to develop different abstract interpreters.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers; model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution; testing tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages; macro and assembly languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Program-

Authors' addresses: J. Lim, GrammaTech, Inc., 531 Esty Street, Ithaca, NY 14850, junghee@grammatech.com. T. Reps, Computer Sciences Dept., Univ. of Wisconsin, 1210 West Dayton Street, Madison, WI 53703, and GrammaTech, Inc., 531 Esty St., Ithaca, NY 14850; reps@cs.wisc.edu. At the time the research reported in the paper was carried out, J. Lim was affiliated with the University of Wisconsin.

The work was supported in part by NSF under grants CCF-{0524051, 0540955, 0810053, 0904371}; by ONR under grants N00014-{01-1-0708, 01-1-0796, 09-1-0510, 09-1-0776, 10-M-0251, 11-C-0447}; by ARL under grant W911NF-09-1-0413; by AFRL under grants FA8750-05-C-0179, FA8750-06-C-0249, FA9550-09-1-0279 and FA8650-10-C-7088; by DARPA under cooperative agreement HR0011-12-2-0012; by a donation from GrammaTech, Inc.; and by a Symantec Research Labs Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring companies or agencies.

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

Portions of this work appeared in the 17th Int. Conf. on Compiler Construction [Lim and Reps 2008], the 16th Int. SPIN Workshop [Lim et al. 2009] and a subsequent journal article [Lim et al. 2011], and the 22nd Int. Conf. on Computer Aided Verification [Thakur et al. 2010], as well as in J. Lim's Ph.D. dissertation [Lim 2011].

ming Languages—*Program analysis*

General Terms: Algorithms, Languages, Security, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, machine-code analysis, dynamic analysis, symbolic analysis, static analysis, dataflow analysis

1. INTRODUCTION

In recent years, methods to analyze machine-code programs have been receiving increased attention. Two of the factors that motivate such work are (i) source code is often unavailable, and (ii) machine code is closer than source code to what is actually executed, which is often important in the context of computer security. While the tools and techniques that have been developed for analyzing machine code are, in principle, language-independent, implementations are often tied to one specific instruction set. Retargeting them to another instruction set can be an expensive and error-prone process.

This paper describes the design and implementation of a system, called TSL (for "**T**ransformer **S**pecification **L**anguage"),[1] that provides a systematic solution to the problem of creating retargetable tools for analyzing machine code. TSL is a tool generator—i.e., a meta-tool—that automatically creates different abstract interpreters for machine-code instruction sets. More precisely, TSL is an *abstract-transformer-generator generator*. The TSL system provides a language in which a user specifies the concrete operational semantics of an instruction set; from a TSL specification, the TSL compiler generates an intermediate representation that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the primitives of the TSL language (i.e., the TSL base-types, map-types, and operations on values of those types). TSL's run-time system supports the use of such generated abstract-transformer generators for dynamic analysis, static analysis, and symbolic execution.[2]

TSL advances the state of the art in program analysis by providing a YACC-like mechanism for creating the key components of machine-code analyzers: from a *description* of the concrete operational semantics of a given instruction set, TSL automatically creates *implementations* of different abstract interpreters for the instruction set.

In designing the TSL system, the most challenging technical issue that we faced was how to automate the generation of the set of *abstract transformers* for a given abstract interpretation of a given instruction set. There have been a number of past efforts to create generator tools to support abstract interpretation, including MUG2 [Wilhelm 1981], SPARE [Venkatesh 1989; Venkatesh and Fischer 1992], Steffen's work on harnessing model checking for dataflow analysis [Steffen 1991; 1993], Sharlit [Tjiang and Hennessy 1992], Z [Yi and Harrison, III 1993], PAG [Alt

[1] TSL is also used as the name of the system's meta-language.

[2] Because an abstract transformer is a component of a tool, an abstract-transformer generator is a meta-tool. Thus, technically, an abstract-transformer-generator generator is a meta-meta-tool. To avoid such an awkward locution, we will not distinguish between the "meta-" role and the "meta-meta-" role and refer to TSL generically as a "meta-tool".

and Martin 1995], OPTIMIX [Assmann 2000], TVLA [Lev-Ami and Sagiv 2000; Reps et al. 2010], HOIST [Regehr and Reid 2004], and RHODIUM [Scherpelz et al. 2007]. However, in all but the last three, the user specifies an *abstract semantics*, but not the *concrete semantics* of the language to be analyzed. Moreover, it is the responsibility of the user to establish, outside of the system, the soundness of the abstract semantics with respect to the (generally not-written-down) concrete semantics.

In contrast, a major goal of our work was to adhere closely to the credo of abstract interpretation [Cousot and Cousot 1977]:

—specify the concrete semantics

—obtain an abstract semantics as an abstraction of the concrete semantics.

In particular, a specification of the concrete semantics of the language to be analyzed is an explicit artifact that the TSL compiler receives as input. Consequently, TSL differs from most past work that has attempted to automate the creation of abstract interpreters.

A language's concrete semantics is specified in TSL's meta-language. The meta-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. Thus, writing a TSL specification for a language is similar to writing an interpreter for that language in first-order ML.

TSL provides a fixed set of basetypes and operators, as well as map-types with map-access and (applicative) map-update operations. From a TSL specification, the TSL compiler generates a common intermediate representation (CIR) that allows the meanings of the input-language constructs to be redefined by supplying alternative interpretations of the basetypes, map-types, and the operations on them (also known as "*semantic reinterpretation*"). Because all the abstract operations are defined at the *meta-level*, semantic reinterpretation is independent of any given language defined in TSL. Therefore, each implementation of an analysis component's driver serves as the unchanging driver for use in different instantiations of the analysis component to different languages. The TSL language becomes the specification language for retargeting that analysis component for different languages. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of a language, and N analysis implementations, i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

Problem Statement. Our work addresses the following fundamental problem in abstract interpretation:

> Given the concrete semantics for a language, how can one systematically create the associated abstract transformers?

In addition to the theory of abstract interpretation itself [Cousot and Cousot 1977], the inspiration for our work is two-fold:

—Prior work on systems that generate analyzers from the concrete semantics of a language: TVLA, HOIST, and RHODIUM.

—Prior work on semantic reinterpretation [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993].

The use of semantic reinterpretation in TSL as the basis for generating abstract transformers is what distinguishes our work from TVLA, HOIST, and RHODIUM. Semantic reinterpretation is discussed in more detail in §2.2 and §3.2.

Our work also addresses the *retargeting problem*. The literature on program analysis is vast, and essentially all of the results described in the literature are, in principle, language-independent. However, their implementations are often tied to one specific language. Retargeting them to another language (as well as implementing a new analysis for the same language) can be an expensive and error-prone process. TSL represents one point in the design space of tools to support retargetable program analyzers, namely, a meta-tool—a tool generator—that automatically creates different abstract interpreters for a language.

Contributions. TSL advances the state of the art in program analysis because it provides two dimensions of parameterizability. In particular,

—a given analysis component can be retargeted to different instruction sets. One merely has to write a TSL specification of the concrete semantics of a given instruction set. In this respect, TSL provides a YACC-like mechanism for creating different instantiations of an analysis component for different languages: from a *description* of the concrete operational semantics, TSL automatically creates *implementations* of different analysis components.

—multiple analysis components can be created automatically from a single specification of the concrete operational semantics of the language to be analyzed. For each new analysis component, the analysis designer merely has to provide a reinterpretation of the basetypes, map-types, and operators of the TSL meta-language.

Other notable aspects of our work include

—*Support for multiple analysis types.* The system supports several analysis types:
  —classical worklist-based value-propagation analyses
  —transformer-composition-based analyses [Cousot and Cousot 1979; Sharir and Pnueli 1981], which are particularly useful for context-sensitive interprocedural analysis, and for relational analyses
  —unification-based analyses for flow-insensitive interprocedural analysis
  —dynamic analyses (including concrete emulation using concrete semantics)
  —symbolic analyses

—*Implemented analyses.* These mechanisms have been instantiated for a number of specific analyses that are useful for analyzing machine code, including value-set analysis [Balakrishnan 2007; Balakrishnan and Reps 2004] (§4.1.1), affine-relation analysis [Müller-Olm and Seidl 2005; Elder et al. 2011] (§4.1.2), def-use analysis (for memory, registers, and flags) (§4.1.4), aggregate structure identification [Ramalingam et al. 1999] (§4.1.3), and generation of symbolic expressions for an instruction's semantics (§4.1.5).

  Using TSL, we also developed a novel way of applying semantic reinterpretation to create symbolic-analysis primitives automatically for symbolic evaluation, preimage computation, and symbolic composition [Lim et al. 2011].

—*Established applicability.* The capabilities of our approach have been demonstrated by writing specifications for IA32 (also known as x86[3]) and PowerPC. These are nearly complete specifications of the integer subset of these languages, and include such features as (1) aliasing among 8-, 16-, and 32-bit registers, e.g., al, ah, ax, and eax (for IA32), (2) endianness, (3) issues arising due to bounded-word-size arithmetic (overflow/underflow, carry/borrow, shifting, rotation, etc.), and (4) setting of condition codes (and their subsequent interpretation at jump instructions). We have also experimented with sufficiently complex features of other machine-code languages (e.g., register windows for Sun SPARC and conditional execution of instructions for ARM) to know that they fit our specification and implementation models.

TSL has been used to recreate the analysis components employed by CodeSurfer/x86 [Balakrishnan et al. 2005; Balakrishnan and Reps 2010], which is a static-analysis framework for analyzing stripped x86 executables. The TSL-generated analysis components include value-set analysis, affine-relation analysis, def-use analysis (for memory, registers, and flags), and aggregate structure identification. From the TSL specification of PowerPC, we also generated the analysis components needed for a PowerPC version, CodeSurfer/ppc32.

In addition, using TSL-generated primitives for symbolic analysis, we developed a machine-code verification tool, called MCVETO [Thakur et al. 2010] (§4.3), and a concolic-execution-based program-exploration tool, called BCE [Lim and Reps 2010] (§4.4).

—*Evaluation of the benefits of the approach.* As discussed in §5, TSL provides benefits from several standpoints, including (i) development time, and (ii) the precision of TSL-generated abstract transformers.

Organization of the Paper. The remainder of the paper is organized as follows. §2 presents an overview of the TSL system, the principles that lie behind it, and the kinds of applications to which it has been applied. §3 describes TSL in more detail, and considered from three perspectives: (i) how to write a TSL specification (from the point of view of instruction-set-specification developers), (ii) how to write domains for (re)interpreting the TSL basetypes and map-types (from the point of view of reinterpretation developers), and (iii) how to use TSL-generated abstract transformers (from the point of view of tool developers). §4 summarizes some of the analyses that have been written using TSL. §5 presents an evaluation of the costs and benefits of the TSL approach. §6 discusses related work. §7 concludes.

2. OVERVIEW OF THE TSL SYSTEM

The goal of TSL is to provide a systematic way of implementing analyzers that work on machine code. TSL has three classes of users: (i) instruction-set-specification (ISS) developers, (ii) reinterpretation developers, and (iii) tool developers. The ISS developers are involved in specifying the semantics of different instruction sets; the reinterpretation developers are involved in defining abstract domains and reinterpretations for the TSL basetypes; the tool developers are involved in extending the analysis framework. The TSL language allows ISS developers to specify the concrete

[3]IA32 and x86 will be used interchangeably in this paper.

semantics of an instruction set. The TSL run-time system—defined by a collection of C++ classes—allows analysis developers to easily create analyzers that support dynamic analysis, static analysis, and symbolic analysis of executables written in any instruction set for which a TSL semantic specification has been written.

2.1 Design Principles

In designing the TSL language, we were guided by the following principles:

(1) There should be a formal language for specifying the semantics of the language to be analyzed. Moreover, an ISS developer should specify only the abstract syntax and a concrete operational semantics of the language to be analyzed. Each analyzer should be generated automatically from this specification.
(2) Concrete syntactic issues—including (i) decoding (machine code to abstract syntax), (ii) encoding (abstract syntax to machine code), (iii) parsing assembly (assembly code to abstract syntax), and (iv) assembly pretty-printing (abstract syntax to assembly code)—should be handled separately from the abstract syntax and concrete semantics.
(3) There should be a clean interface for reinterpretation developers to specify the abstract semantics for each analysis. An abstract semantics consists of an *interpretation*: an abstract domain and a set of abstract operators (i.e., for the operations of TSL).
(4) The abstract semantics for each analysis should be separated from the languages to be analyzed so that one does not need to specify multiple versions of an abstract semantics for multiple languages.

Each of these objectives has been achieved in the TSL system: The TSL system translates the TSL specification of each instruction set to a common intermediate representation (CIR) that can be used to create multiple analyzers (§2.3 and §3.1.3). Each analyzer is specified at the level of the meta-language (i.e., by reinterpreting the operations of TSL), which—by extension to TSL expressions and functions—provides the desired reinterpretation of the instructions of an instruction set.

ISAL: Concrete Syntactic Issues. Item (2) in the list above—the translation of various concrete syntaxes to and from the abstract syntax—is actually handled by another meta-tool, called ISAL, which is separate from TSL. ISAL is roughly comparable to SLED [Ramsey and Fernández 1997]. The input to ISAL specifies the representation of an instruction set's instructions, and ISAL generates components that implement several tasks having to do with syntactic issues, including

—a decoder that converts machine-code bits/bytes to a TSL-defined abstract-syntax tree
—an instruction disassembler that converts machine-code bits/bytes to an assembly-language instruction in ASCII text
—a generator of random samples of instructions for testing purposes (which are produced in three forms: machine code (bits), assembly language (text), and TSL abstract-syntax trees).

The relationship between ISAL and TSL is similar to the one between `flex` and `bison`. With `flex` and `bison`, the specification of a collection of token identifiers

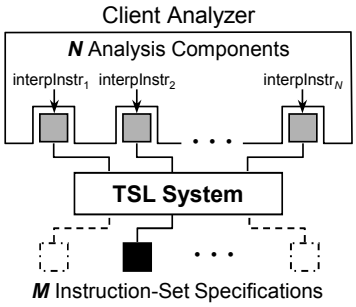

Fig. 1. The interaction between the TSL system and a client analysis tool. Each gray box represents a TSL-generated abstract-transformer generator.

$$s_1\colon x = x \oplus y;$$
$$s_2\colon y = x \oplus y;$$
$$s_3\colon x = x \oplus y;$$

Fig. 2. Code fragment that swaps two `int`s, using three $\oplus$ operations.

is shared, which allows a `flex`-generated lexer to pass tokens to a `bison`-generated parser. With ISAL and TSL, the specification of an instruction set's abstract syntax is shared, which allows ISAL to pass abstract-syntax trees to a TSL-generated instruction-set analyzer.

Many for the Price of One! As shown in Fig. 1, once one has the N analysis implementations that are the core of some client analysis tool A, one obtains a generator that can create different versions A/M_1, A/M_2, ... at the cost of writing specifications of the concrete semantics of instruction sets M_1, M_2, etc. Thus, each client analysis tool A built using abstract-transformer generators created via TSL acts as a "YACC-like" tool for generating different versions of A automatically.

2.2 Semantic Reinterpretation

The TSL system 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 basetypes, map-types, and operators (sometimes called a *semantic algebra* [Schmidt 1986]), 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*.[4]

Semantic Reinterpretation for Abstract Interpretation. The idea of exploiting such a factoring comes from the field of abstract interpretation [Cousot and Cousot 1977], where factoring-plus-reinterpretation has been proposed as a convenient tool

[4]Semantic reinterpretation is a program-generation technique, and thus we follow the terminology of the partial-evaluation literature [Jones et al. 1993], where the program on which the partial evaluator operates is called the *subject program*.

In logic and linguistics, the programming language would be called the "object language". In the compiler literature, an object program is a machine-code program produced by a compiler, and so we avoid using the term "object programs" for the programs that TSL operates on.

for formulating abstract interpretations and proving them to be sound [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993]. In particular, soundness of the *entire* abstract semantics can be established via purely *local* soundness arguments for each of the reinterpreted operators.

The following example shows the basic principles of semantic reinterpretation in the context of abstract interpretation. We use a simple language of assignments, and define the concrete semantics and an abstract sign-analysis semantics via semantic reinterpretation.

EXAMPLE 2.1. (Adapted from [Malmkjær 1993].) Consider the following fragment of a denotational semantics, which defines the meaning of assignment statements over variables that hold signed 32-bit `int` values (where $\oplus$ denotes exclusive-or):

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

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

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

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

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

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

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

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

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

Different abstract interpretations can be defined by using the same client semantics, but giving different interpretations to the basetypes, map-types, and operators of the core. For example, for sign analysis, assuming that *Int32* values are represented

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

Fig. 3. Application of the abstract transformers created by the sign-analysis reinterpretation to the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$.

in two's-complement notation, the semantic core is reinterpreted as follows:

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

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

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

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

For the code fragment shown in Fig. 2, which swaps two `int`s, sign-analysis reinterpretation creates abstract transformers that, given the initial abstract state $\sigma_0 = \{x \mapsto neg, y \mapsto pos\}$, produce the abstract states shown in Fig. 3. □

Alternatives to Semantic Reinterpretation. The mapping of a client specification to the operations of the semantic core resembles a translation to a *universal assembly language* (UAL). Thus, another approach to obtaining "systematic" reinterpretations that are similar to semantic reinterpretations—in that they can be retargeted to multiple subject languages—is to translate subject-language programs to a UAL, and then retarget the instructions of the UAL to operate on abstract values or abstract states. Semantic reinterpretation is compared with this approach in §6.2.

2.3 Technical Contributions Incorporated in the TSL Compilation Process

The specific technical contributions incorporated in the part of the TSL compiler that generates the CIR can be summarized as follows:

—*Two-Level Semantics:* In the TSL system, the notion of a *two-level* intermediate language [Nielson and Nielson 1992] is used to generate the CIR in a way that reduces the loss of precision that could otherwise come about with certain reinterpretations. To address this issue, the TSL compiler performs binding-time analysis [Jones et al. 1993] on the TSL specification to identify which values

can always be treated as concrete values, and which operations should therefore be performed in the concrete domain (i.e., should not be reinterpreted). §3.2.2 discusses more details of the two-level intermediate language along with binding-time analysis.

—*Abstract Interpretation:* From a specification, the TSL compiler generates a CIR that has the ability (i) to execute over abstract states, (ii) possibly propagate abstract states to more than one successor in a conditional expression, (iii) compare abstract states and terminate abstract execution when a fixed point is reached, and (iv) apply widening operators, if necessary, to ensure termination. §3.2.1 contains a detailed discussion of these issues.

—*Paired Semantics:* The TSL system allows easy instantiations of *reduced products* by means of *paired semantics*. The CIR can be instantiated with a *paired* semantic domain that couples two interpretations. Communication between the values carried by the two interpretations may take place in the TSL basetype and map-type operators. §3.2.3 discusses more details of paired semantics.

2.4 The Context of Our Work

TSL has primarily been applied to the creation of abstract interpreters for machine code. Machine-code analysis presents many interesting challenges. For instance, at the machine-code level, memory is one large byte-addressable array, and an analyzer must handle computed—and possibly non-aligned—addresses. It is crucial to track array accesses and updates accurately; however, the task is complicated by the fact that arithmetic and dereferencing operations are both pervasive and inextricably intermingled. For instance, if local variable `x` is at offset –12 from the activation record's frame pointer (register `ebp`), an access on `x` would be turned into an operand [`ebp`–12]. Evaluating the operand first involves pointer arithmetic ("`ebp`–12") and then dereferencing the computed address ("[·]"). On the other hand, machine-code analysis also offers new opportunities, in particular, the opportunity to track low-level, platform-specific details, such as memory-layout effects. Programmers are typically unaware of such details; however, they are often the source of exploitable security vulnerabilities.

For more discussion of the challenges and opportunities that arise in machine-code analysis, the reader is referred to [Balakrishnan and Reps 2010] and [Reps et al. 2010]. However, it is worth mentioning a couple of points here that help to illustrate the scope of the problem that TSL addresses:

—The paper presents several fragments of TSL specifications that specify the operational semantics of instruction sets, such as IA32 (also known as x86) and PowerPC (see §4). In such specifications, the subject language is modeled at the level of the instruction-set semantics. That is, the TSL specification describes how the execution of each instruction changes the execution state. Lower-level hardware operations, such as pipelining and paging, are not modeled in our specifications, although TSL is powerful enough to specify such lower-level operations.

—Given a TSL specification of an interpreter at the instruction-set level, the TSL compiler generates an analysis component that performs abstract interpretation on a per-instruction basis. It is the tool developer's responsibility to complete the implementation of the analysis by handling the higher levels of abstract inter-

pretation, such as (i) the propagation of abstract values through the control-flow graph, (ii) determining when a fixed point is reached, etc.

To help out in this task, the TSL system provides several kinds of generic execution/analysis engines that can be instantiated to create finished analyses, including (i) a worklist-based solver for abstract-value propagation over a control-flow graph (for static analysis), (ii) an instruction emulator (for dynamic analysis), and (iii) an engine for performing symbolic execution along a path (for symbolic analysis), as well as a solver for aggregate-structure-identification problems (ASI) [Ramalingam et al. 1999; Balakrishnan and Reps 2007]—a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program.

In addition, we have used TSL-generated abstract transformers with general-purpose analysis packages, such as the WALi system [WALi 2007] for weighted pushdown systems (WPDSs) [Reps et al. 2005; Bouajjani et al. 2003] and the OpenNWA system [Driscoll et al. 2012] for nested-word automata [Alur and Madhusudan 2006]. In principle, it would be easy to use TSL to drive other similar external packages, such as the Banshee system for solving set-constraint problems [Kodumal and Aiken 2005]. (See §4.1.)

—Although this paper only discusses the application of TSL to machine-code instruction sets, in principle it should be possible to extend TSL so that it can be applied to source-code languages (i.e., to create language-independent analyzers for source-level IRs), as well as bytecode. The main obstacle is that the concrete semantics of a source-code language generally uses an execution state based on a stack of variable-to-value (or variable-to-location, location-to-value) maps. For a machine-code language, the state incorporates an address-based memory model, for which the TSL language provides appropriate primitives (as well as the opportunity to redefine them). To be able to apply TSL to source-code languages, TSL would need to be extended with a way to redefine the datatype that represents the stack—i.e., to define an abstraction of the stack—as well as to redefine the set of primitives that would be provided for manipulating the stack.

3. TRANSFORMER SPECIFICATION LANGUAGE

This section presents the basic elements of the TSL system. §3.1 describes the basic elements of the TSL language and what is produced by the TSL compiler. It considers the TSL system from the perspective of instruction-set specifiers (ISS), reinterpretation developers, and tool developers. §3.2 discusses how the TSL compiler generates a CIR from a TSL specification and how the CIR is instantiated for creating analysis components. §3.2 also describes how the TSL system handles some important issues, such as recursion and conditional branches in the CIR. §3.3 discusses the leverage that the TSL system provides.

3.1 Overview of the TSL Language and its Compilation

The key principle of the TSL system is the separation of the semantics of a subject language from the analysis semantics in the development of an analysis component. As discussed in §2.2, the TSL system is based on semantic reinterpretation, which was originally proposed as a convenient *methodology* for formulating abstract interpretations [Cousot and Cousot 1977; Mycroft and Jones 1985; Jones and Mycroft 1986; Malmkjær 1993; Nielson 1989] (see §2.2). Semantic reinterpretation involves

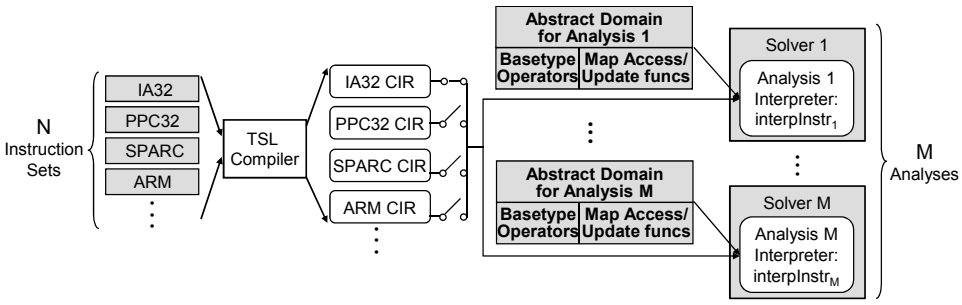

Fig. 4. A depiction of how the elements of the TSL system fit together.

refactoring the specification of the concrete semantics of a language into two parts: (i) a *client* specification, and (ii) a semantic *core*. The client is expressed in terms of the semantic core. Such an organization permits the core to be *reinterpreted* to produce an alternative semantics for the subject language.

The key insight behind the TSL system is that if a rich enough *meta-language* is provided for writing semantic specifications, the meta-language itself can serve as the core, and one thereby obtains a suitable client/core factoring for free.

As presented earlier, the TSL system has three classes of users: (i) instruction-set specification (ISS) developers, (ii) reinterpretation developers, and (iii) tool developers. Fig. 4 gives a more detailed version of Fig. 1, showing how the elements of the TSL system fit together. An ISS developer uses the TSL language to specify the concrete semantics of different instruction sets (the gray boxes on the left in Fig. 4). A reinterpretation developer redefines the types and operations of TSL's meta-language to create a component that performs abstract interpretation of machine-code instructions (the gray boxes in the middle of Fig. 4). A tool developer uses such abstract interpreters to create solvers that explore the state space of a program (the gray boxes on the right in Fig. 4); the tool developer is also in charge of orchestrating the sequence in which the M different analysis components are invoked.

3.1.1 *TSL from an Instruction-Set Specifier's Standpoint.* Fig. 5 shows part of a specification of the IA32 instruction set taken from the Intel manual [IA32]. The specification describes the syntax and the semantics of each instruction only in a semi-formal way (i.e., a mixture of English and pseudo-code).

General Purpose Registers:
EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI,EIP
Each of these registers also has 16- or 8-bit subset names.
Addressing Modes: [sreg:][offset][([base][,index][,scale])]
EFLAGS register: ZF,SF,OF,CF,AF,PF, . . .
	ADD r/m32,r32; Add r32 to r/m32
	ADD r/m16,r16; Add r16 to r/m16 . . .
	Operation: DEST ← DEST + SRC;
	Flags Affected: The OF,SF,ZF,AF,CF, and
	PF flags are set according to the result.

Fig. 5. A part of the Intel manual's specification of IA32's `add` instruction.

Our work is based on completely formal specifications that are written in TSL's meta language. TSL is a strongly typed, first-order functional language. TSL supports a fixed set of basetypes; a fixed set of arithmetic, bitwise, relational, and logi-

Type	Values	Constants
BOOL	false, true	false, true
INT64	64-bit signed integers	0d64, 1d64, 2d64, ...
INT32	32-bit signed integers	0, 1, 2, ..., 0d32, 1d32, 2d32, ...
INT16	16-bit signed integers	0d16, 1d16, 2d16, ...
INT8	8-bit signed integers	0d8, 1d8, 2d8, ...
MAP[α,β]	Maps	$[\alpha \mapsto v_\beta]$

Fig. 6. Basetype and map-type constants. $[\alpha \mapsto v_\beta]$ denotes the map $\lambda x{:}\alpha \,.\, v{:}\beta$.

cal operators; the ability to define recursive data-types, map-types, and user-defined functions; and a mechanism for deconstruction by means of pattern matching.

Basetypes. Fig. 6 shows the basetypes that TSL provides. There are two categories of primitive basetypes: *unparameterized* and *parameterized*. An unparameterized basetype is just a set of terms. For example, BOOL is a type consisting of truth values, INT32 is a type consisting of 32-bit signed whole numbers, etc. MAP[α, β] is a predefined parameterized type, with parameters α and β. Each of the following is an instance of the parameterized type MAP:

```
MAP[INT32,INT8]
MAP[INT32,BOOL]
MAP[INT32,MAP[INT8,BOOL]]
```

TSL supports arithmetic/logical operators (+, −, ∗, /, !, &&, ||, xor), bit-manipulation operators (∼, &, |, ˆ, ≪, ≫, right-rotate, left-rotate), relational operators (<, <=, >, >=, ==, !=), and a conditional-expression operator (? :). TSL also provides access/update operators for map-types.

Specifying an Instruction Set. Fig. 7(a) shows a snippet of the TSL specification that corresponds to Fig. 5. (The TSL specification has been pared down to simplify the presentation.)

Much of what an instruction-set specifier writes in a TSL specification is similar to writing an interpreter for an instruction set in first-order ML, using language features that are common to most functional languages. In particular, the TSL meta-language supports a datatype-definition mechanism for defining recursive datatypes, data-construction expressions, and data deconstruction by means of pattern matching (in the style pioneered by Burstall [1969]). For instance, Fig. 7(a) shows (i) an inductive datatype, defined by a set of constructors, that specifies the abstract syntax of the instruction set (e.g., lines 2–9), (ii) another inductive datatype for representing concrete states (e.g., lines 10–12, and (iii) the user-defined function interpInstr, which defines the concrete semantics of each instruction (e.g., lines 14–30. Lines 18–29 show a *with-expression*—a multi-branch, pattern-matching expression for data deconstruction.[5]

[5]Fig. 7(a) does not illustrate some additional features of the TSL meta-language that deserve mention:

—Patterns in with-expressions can be nested arbitrarily deeply. If pattern variables appear more than once, the sub-terms in those positions must be equal.

—The TSL compiler implements the pattern-compilation algorithm developed by Wadler [1987]

```
[1]  // User-defined abstract syntax
[2]  reg: EAX() | EBX() | . . . ;
[3]  flag: ZF() | SF() | . . . ;
[4]  operand: Indirect(reg reg INT8 INT32)
[5]              | DirectReg(reg)
[6]              | Immediate(INT32) | ...;
[7]   instruction
[8]       : MOV(operand operand)
[9]       | ADD(operand operand) | . . . ;
[10] state: State(MAP[INT32,INT8]   // memory-map
[11]              MAP[reg32,INT32]  // register-map
[12]              MAP[flag,BOOL]);  // flag-map
[13]  // User-defined functions
[14] INT32 interpOp(state S, operand op) { . . . };
[15] state updateFlag(state S, . . . ) { . . . };
[16] state updateState(state S, . . . ) { . . . };
[17] state interpInstr(instruction I, state S) {
[18]   with(I) (
[19]     MOV(dstOp, srcOp):
[20]       let srcVal = interpOp(S, srcOp);
[21]       in ( updateState( S, dstOp, srcVal ) ),
[22]     ADD(dstOp, srcOp):
[23]       let dstVal = interpOp(S, dstOp);
[24]           srcVal = interpOp(S, srcOp);
[25]           res = dstVal + srcVal;
[26]           S2 = updateFlag(S, dstVal, srcVal, res);
[27]       in ( updateState( S2, dstOp, res ) ),
[28]     . . .
[29]   );
[30] };
```

(a)

```
[1]  template <class INTERP> class CIR {
[2]    class reg { . . . };
[3]    class EAX : public reg { . . . }; . . .
[4]    class flag { . . . };
[5]    class ZF : public flag { . . . }; . . .
[6]    class operand { . . . };
[7]    class Indirect: public operand { . . . }; . . .
[8]    class instruction { . . . };
[9]    class MOV : public instruction { . . .
[10]     operand  op1; operand  op2; . . .
[11]   };
[12]   class ADD : public instruction { . . . }; . . .
[13]   class state { . . . };
[14]   class State: public state { . . . };
[15]   INTERP::INT32 interpOp(state S, operand op) { . . . };
[16]   state updateFlag(state S, . . . ) { . . . };
[17]   state updateState(state S, . . .) { . . . };
[18]   state interpInstr(instruction I, state S) {
[19]    switch(I.id) {
[20]     case ID_MOV: . . .
[21]     case ID_ADD:
[22]       operand dstOp = I.get_child1();
[23]       operand srcOp = I.get_child2();
[24]       INTERP::INT32 dstVal = interpOp(S, dstOp);
[25]       INTERP::INT32 srcVal = interpOp(S, srcOp);
[26]       INTERP::INT32 res = INTERP::Plus32(dstVal, srcVal);
[27]       state S2 = updateFlag(S, dstVal, srcVal, res);
[28]       ans = updateState( S2, dstOp, res );
[29]       break;
[30]    . . . }
[31]  }};
```

(b)

Fig. 7. (a) A part of the TSL specification of IA32 concrete semantics, which corresponds to the specification of `add` from the IA32 manual. Reserved types and function names are underlined, (b) A part of the CIR generated from (a). The CIR is simplified in this presentation.

Reserved, but User-Defined Types and Reserved Functions. Each specification must define several reserved (but user-defined) types: in Fig. 7(a), instruction (lines 7–9); state—e.g., for Intel IA32 the type state is a triple of maps (lines 10–12); as well as the reserved TSL function interpInstr (lines 17–30). These reserved types and functions form part of the API available to *analysis engines* that use the TSL-generated transformers (i.e., the instantiated CIR).

The definition of types and constructors on lines 2–9 of Fig. 7(a) is an abstract-syntax grammar for IA32. Type reg consists of nullary constructors for the names of the IA32 registers, such as EAX() and EBX(); flag consists of nullary constructors for the names of the IA32 condition codes, such as ZF() and SF(). Lines 4–6 define types and constructors to represent the various kinds of operands that IA32 supports, i.e., various sizes of immediate, direct register, and indirect memory operands. The reserved (but user-defined) type instruction consists of user-defined constructors for each instruction, such as MOV and ADD.

---

and Pettersson [1992]; a with-expression's set of patterns is checked for type mismatches, for non-exhaustiveness, and for whether any pattern is unreachable.

—The TSL meta-language supports C-style conditional expressions of the form "e_1 ? e_2 : e_3".

—The TSL meta-language supports let-expressions for locally binding a name to a value.

The type state specifies the structure of the execution state. The state for IA32 is defined on lines 10–12 of Fig. 7(a) to consist of three maps, i.e., a memory-map, a register-map, and a flag-map. The *concrete semantics* is specified by writing a function named interpInstr (see lines 17–30 of Fig. 7(a)), which maps an instruction and a state to a state. For instance, the semantics of ADD is to evaluate the two operands in the input state S, and to create a return state in which the target location holds the summation of the two values, and the flag-map holds appropriate flag values.

3.1.2 *Case Study of Instruction Sets.* In this section, we discuss the quirky characteristics of some instruction sets, and various ways these can be handled in TSL. The reader should bear in mind that an ISS developer's work is just "programming an emulator in a functional programming language"; when we sketch out two different solutions for some issue, an ISS developer could choose either solution. If the ISS developer wishes to make it possible to choose between two solutions at tool-generation time, it is often possible to support such flexibility by being disciplined in how one writes a TSL specification. That is, an ISS developer can typically establish an interface in the TSL specification, behind which they can hide the choice as to which solution is used. (The TSL compiler invokes the C preprocessor on the TSL specification, so the choice would be supported via C preprocessor "#ifdef" directives.) We would argue that such an "information-hiding" approach gives maximum flexibility to TSL users, as opposed to locking them into some limited class of solutions that we might have otherwise devised for the TSL system.

In some cases, one solution might be preferred because it could have a beneficial impact on the precision of a given reinterpretation. Part of the art of using TSL is understanding such issues; however, our experience with over twenty different reinterpretations has been that such precision issues arise very infrequently, and for the most part, the tasks of ISS development and reinterpretation development are quite well separated.

IA32. To provide compatibility with 16-bit and 8-bit versions of the instruction set, the IA32 instruction set provides overlapping register names, such as AX (the lower 16-bits of EAX), AL (the lower 8-bits of AX), and AH (the upper 8-bits of AX). There are two possible ways to specify this feature in TSL. One is to keep three separate maps, for 32-bit registers, 16-bit registers, and 8-bit registers, respectively, and specify that updates to any one of the maps affect the other two maps. Another is to keep one 32-bit map for registers, and obtain the value of a 16-bit or 8-bit register by masking the value of the 32-bit register. (The former method can yield more precise VSA results [Balakrishnan and Reps 2010, §4.7].) Similarly, a 32-bit register is updated with the value of the corresponding 16-bit or 8-bit register by masking and performing a bitwise-or.

If an ISS developer wants to have the option to choose either, the TSL specification can be written so that there is a register-storage "abstract datatype" that hides the choice of which organization is used. For instance, one would introduce the type REGMAP32, with definitions for the two alternatives, as follows:[6]

[6]In TSL, a type is called a "phylum".

```
#ifdef THREE_REG_MAPS
  REGMAP32: Regmap32(MAP[reg32,INT32]
                     MAP[reg16,INT16]
                     MAP[reg8,INT8]);
#else
  phylum MAP[reg32,INT32] REGMAP32;
#endif
```

and change lines 10–12 of Fig. 7(a) to

```
state: State(MAP[INT32,INT8]
             REGMAP32
             MAP[flag,BOOL]);
```

Finally, one would write appropriate map-access and map-update functions. For instance, the map-access functions could be written as follows:

```
#ifdef THREE_REG_MAPS
  INT32 RegValue32(REGMAP32 regs, reg32 r) {
    with(regs) (Regmap32(m32, *, *): m32(r))
  };
  INT16 RegValue16(REGMAP32 regs, reg16 r) {
    with(regs) (Regmap32(*, m16, *): m16(r))
  };
  INT8 RegValue8(REGMAP32 regs, reg8 r) {
    with(regs) (Regmap32(*, *, m8): m8(r))
  };
#else
  INT32 RegValue32(REGMAP32 regs, reg32 r) {
    regs(r)
  };
  INT16 RegValue16(REGMAP32 regs, reg16 r) {
    let reg_loc = with(r) (AX():EAX(), BX():EBX(), CX():ECX(), DX():EDX(),
                           DI():EDI(), SI():ESI(), SP():ESP(), BP():EBP(), IP():EIP());
    in ( Int32To16(regs(reg_loc)) )
  };
  INT8 RegValue8(REGMAP32 regs, reg8 r) {
    let f_lower = with(r) (AL():true, BL():true, CL():true, DL():true,
                           AH():false, BH():false, CH():false, DH():false);
    reg_loc = with(r) (AL():EAX(), BL():EBX(), CL():ECX(), DL():EDX(),
                           AH():EAX(), BH():EBX(), CH():ECX(), DH():EDX());
    in (
      f_lower ? Int32To8(regs(reg_loc)) : Int32To8((regs(reg_loc) ≫u 8) & 255)
    )
  };
#endif
```

The IA32 processor keeps condition codes in a special register, called EFLAGS. (Many other processors, such as SPARC, PowerPC, and ARM, also use a special register to store condition codes.) One way to address this feature is to declare "reg32:Eflags();", and make every flag manipulation fetch the bit value from an appropriate bit position of the value associated with Eflags in the register-map. Another way is to introduce flag names, as in Fig. 7(a), and have every manipulation of EFLAGS affect the entries in a flag-map for the individual flags (see lines 3 and 12). Again, nothing precludes making either choice, although because flag values control branching, it is desirable for an abstract domain to be able to hold onto flag values

as precisely as possible, which is often easier using an abstraction of individual Boolean values, rather than using an abstraction of an integer-valued register.

Similarly, one must choose a suitable representation of memory. The specification in Fig. 7(a) corresponds to the so-called "flat memory model", in which all available memory locations can be addressed without going through additional levels of addressing, such as memory segments.[7] Even for the flat memory model different choices are possible. In line 10 of Fig. 7(a), we use MAP[INT32,INT8], which specifies 1-byte addressing, but one could use alternative representations of memory with addressing of 2-byte or 4-byte quantities. In the latter cases, out-of-alignment accesses would have to interact with multiple map entries.

As mentioned earlier, an ISS developer's work is "just programming". If an ISS developer wants to provide the option to make different choices at tool-generation time, the best strategy is to write the TSL specification so that there is a memory-storage abstract datatype that hides the choice of which storage organization is used. For instance, TSL is already equipped with a library that has TSL functions

```
INT64 MemAccess_32_8_LE_64(MEMMAP32_8 m, INT32 addr);
INT32 MemAccess_32_8_LE_32(MEMMAP32_8 m, INT32 addr);
INT16 MemAccess_32_8_LE_16(MEMMAP32_8 m, INT32 addr);
INT8 MemAccess_32_8_LE_8 (MEMMAP32_8 m, INT32 addr);

MEMMAP32_8 MemUpdate_32_8_LE_64(MEMMAP32_8 m, INT32 addr, INT64 v);
MEMMAP32_8 MemUpdate_32_8_LE_32(MEMMAP32_8 m, INT32 addr, INT32 v);
MEMMAP32_8 MemUpdate_32_8_LE_16(MEMMAP32_8 m, INT32 addr, INT16 v);
MEMMAP32_8 MemUpdate_32_8_LE_8 (MEMMAP32_8 m, INT32 addr, INT8 v);
```

and one can hide implementation details behind different representation choices for MEMMAP32_8 and the interface functions given above.

ARM. Almost all ARM instructions contain a condition field that allows an instruction to be executed conditionally, depending on condition-code flags. This feature reduces branch overhead and compensates for the lack of a branch predictor. However, it may worsen the precision of an abstract analysis because in most instructions' specifications, the abstract values from two arms of a TSL conditional expression would be joined.

```
[1] MOVEQ(destReg, srcOprnd):
[2]   let cond = flagMap(EQ());
[3]       src = interpOperand(curState, srcOprnd);
[4]       a = regMap[destReg |-> src];
[5]       b = regMap;
[6]       answer = cond ? a : b;
[7]   in ( answer )
```

Fig. 8. An example of the specification of an ARM conditional-move instruction in TSL.

For example, MOVEQ is one of ARM's conditional instructions; if the flag EQ is true when the instruction starts executing, it executes normally; otherwise, the

[7] A segmented model could also be specified.

```
[1] reg32 : Reg(INT8) | CWP() | . . .;
[2] reg32 : OutReg(INT8) | InReg(INT8) | . . .;
[3] state: State( . . . , MAP[var32,INT32], . . . );
[4] INT32 RegAccess(MAP[var32,INT32] regmap, reg32 r) {
[5]  let cwp = regmap(CWP());
[6]    key = with(r) (
[7]     OutReg(i):
[8]       Reg(8+i+(16+cwp*16)%(NWINDOWS*16),
[9]     InReg(i): Reg(8+i+cwp*16),
[10]  . . . );
[11] in ( regmap(key) )
[12]}
```

Fig. 9. A method to handle the SPARC register window in TSL.

instruction does nothing. Fig. 8 shows the specification of the instruction in TSL. In many abstract semantics, the conditional expression "*cond* ? *a* : *b*" will be interpreted as a join of the original register map *b* and the updated map *a*, i.e., *join*(*a*,*b*). Consequently, *destReg* would receive the join of its original value and *src*, even when *cond* is known to have a definite value (TRUE or FALSE) in VSA semantics. The paired-semantics mechanism presented in §3.2.3 can help with improving the precision of analyzers by abstractly interpreting conditions. When the CIR is instantiated with a paired semantics of VSA_INTERP and DUA_INTERP, and the VSA value of *cond* is FALSE, the DUA_INTERP value for *answer* gets empty *def*- and *use*-sets because the true branch *a* is known to be unreachable according to the VSA_INTERP value of *cond* (instead of non-empty sets for *def*s and *use*s that contain all the definitions and uses in *destReg* and *srcOprnd*).

SPARC. SPARC uses register windows to reduce the overhead associated with saving registers to the stack during a conventional function call. Each window has 8 in, 8 out, 8 local, and 8 global registers. Outs become ins on a context switch, and the new context gets a new set of out and local registers. A specific platform will have some total number of registers, which are organized as a circular buffer; when the buffer becomes full, registers are spilled to the stack to free up a sufficient number for the called procedure. Fig. 9 shows a way to accommodate this feature. The syntactic register (OutReg(n) or InReg(n), defined on line 2) in an instruction is used to obtain a semantic register (Reg(m), defined on line 1, where m represents the register's global index), which is the key used for accesses on and updates to the register map. The desired index of the semantic register is computed from the index of the syntactic register, the value of CWP (the current window pointer) from the current state, and the platform-specific value NWINDOWS (lines 8–9).

Implementing register spilling and restoring is more complicated because the spilling/restoring policy is not part of the SPARC instruction-set semantics *per se*. Instead, spilling and restoring are handled by policies incorporated in appropriate trap handlers, as described by Magnusson [2011]:

> "When the SAVE instruction decrements the current window pointer (CWP) so that it coincides with the invalid window in the window invalid mask (WIM), a window overflow trap occurs. Conversely, when the

> RESTORE or RETT instructions increment the CWP to coincide with the invalid window, a window underflow trap occurs. Either trap is handled by the operating system. Generally, data is written out to memory and/or read from memory, and the WIM register suitably altered."

Thus, to implement spilling/restoring, the TSL-specified semantics of SAVE, RESTORE, and RETT would have to check for overflow/underflow and set an appropriate flag in some component of the state. Each "analysis driver" (e.g., fixed-point-finding solver, symbolic executor) would have the obligation to check whether the flag is set, and if so, start analyzing the code of the appropriate trap handler (cf. the bare-bones handlers given by Magnusson [2011]).

3.1.3 *Common Intermediate Representation (CIR).* Fig. 7(b) shows part of the common intermediate representation (CIR) generated by the TSL compiler from Fig. 7(a). (The CIR has been simplified for the presentation in the paper.)

The CIR generated for a given TSL specification is a C++ template that can be used to create multiple analysis components by instantiating the template with different semantic reinterpretations. Each generated CIR is *specific* to a given instruction-set specification, but *common* (whence the name CIR) across generated analyses. Each generated CIR is a template class that takes as input class INTERP, which is an abstract domain for an analysis (line 1 of Fig. 7(b)). The user-defined abstract syntax (lines 2–9 of Fig. 7(a)) is translated to a set of C++ abstract-syntax classes (lines 2–12 of Fig. 7(b)). The user-defined types, such as reg, operand, and instruction, are translated to abstract C++ classes, and the constructors, such as EAX(), Indirect(_,_,_,_), and ADD(_,_), are subclasses of the appropriate parent abstract C++ classes.

Each user-defined function is translated to a CIR function (lines 15–31 of Fig. 7(b)). Each TSL basetype name and basetype-operator name is prepended with the template parameter name INTERP. To instantiate the CIR, class INTERP is supplied by an analysis developer for the analysis of interest.

The TSL front-end performs *with-normalization*, which transforms all multi-level with expressions to use only one-level patterns, and then compiles the one-level pattern via the pattern-compilation algorithm developed by Wadler [1987] and Pettersson [1992]. Thus, the with expression on line 18 and the patterns on lines 19 and 22 of Fig. 7(a) are translated into switch statements in C++ (lines 19–30 in Fig. 7(b)).

The function calls for obtaining the values of the two operands (lines 23–24 in Fig. 7(a)) correspond to the C++ code on lines 22–25 in Fig. 7(b). The TSL basetype-operator + on line 25 in Fig. 7(a) is translated into a call to INTERP::Plus32, as shown on line 26 in Fig. 7(b). The function calls for updating the state (lines 26–27 in Fig. 7(a)) are translated into C++ calls (lines 27–28 in Fig. 7(b)).

§3.2 presents more details about how the CIR is generated and what kind of facilities CIR provides for creating analysis components.

3.1.4 *TSL from a Reinterpretation Developer's Standpoint.* A reinterpretation developer creates a new analysis component by defining, in C++, an *abstract domain*. This section sketches out what a reinterpretation developer has to do to define a new abstract domain in the context of TSL.

A reinterpretation developer must specify "replacements" for (i) the TSL basetypes (BOOL, INT32, INT8, etc.), (ii) the primitive operations on basetypes (Plus32, Plus8, etc.), (iii) the TSL map-types used in the specification (e.g., MEMMAP32_8, REGMAP32, FLAGMAP), and (iv) the associated map-access/update functions. These C++ definitions are used to instantiate the CIR template by passing the classes of basetypes and map-types as template parameters. Each abstract domain is also required to support a set of reserved functions, such as *join*, *meet*, and *widen*, because these operations are an additional part of the API used in the CIR template.

In essence, the reinterpretation developer's work is greatly simplified because, by reinterpreting the primitives of the TSL meta-language, reinterpretation is extended automatically in the CIR to TSL expressions and functions. One automatically obtains an abstract interpretation of the TSL language, and in particular, one obtains an abstract interpretation of the semantics of a given instruction set directly from the TSL specification of the instruction set's concrete semantics.

Table I. Parts of the definitions of the basetypes, basetype-operators, map-types, and map-access/update functions for three different analyses.

VSA	DUA	QFBV
[1] class VSA_INTERP {	[1] class DUA_INTERP {	[1] class QFBV_INTERP {
[2] // basetype	[2] // basetype	[2] // basetype
[3] typedef ValueSet32 INT32;	[3] typedef UseSet INT32;	[3] typedef QFBVTerm32 INT32;
[4] . . .	[4] . . .	[4] . . .
[5] // basetype-operators	[5] // basetype-operators	[5] // basetype-operators
[6] INT32 Plus32(INT32 a, INT32 b) {	[6] INT32 Plus32(INT32 a, INT32 b) {	[6] INT32 Plus32(INT32 a, INT32 b) {
[7] return a.addValueSet(b);	[7] return a.Union(b);	[7] return QFBVPlus32(a, b);
[8] }	[8] }	[8] }
[9] . . .	[9] . . .	[9] . . .
[10] // map-basetypes	[10] // map-basetypes	[10] // map-basetypes
[11] typedef Map<reg32,INT32>	[11] typedef Map<var32,INT32>	[11] typedef QFBVArray
[12] REGMAP32;	[12] REGMAP32;	[12] REGMAP32;
[13] . . .	[13] . . .	[13] . . .
[14] // map-access/update functions	[14] // map-access/update functions	[14] // map-access/update functions
[15] INT32 MapAccess(	[15] INT32 MapAccess(	[15] INT32 MapAccess(
[16] REGMAP32 m, reg32 k) {	[16] REGMAP32 m, reg32 k) {	[16] REGMAP32 m, reg32 k) {
[17] return m.Lookup(k);	[17] return m.Lookup(k);	[17] return QFBVArrayAccess(m,k);
[18] }	[18] }	[18] }
[19] REGMAP32	[19] REGMAP32	[19] REGMAP32
[20] MapUpdate(REGMAP32 m,	[20] MapUpdate(REGMAP32 m,	[20] MapUpdate(REGMAP32 m,
[21] reg32 k, INT32 v) {	[21] reg32 k, INT32 v) {	[21] reg32 k, INT32 v) {
[22] return m.Insert(k, v);	[22] return m.Insert(k,v);	[22] return QFBVArrayUpdate(m,k,v);
[23] }	[23] }	[23] }
[24] . . .	[24]. . .	[24] . . .
[25]};	[25]};	[25]};

Tab. I shows parts of the definitions of the basetypes, basetype-operators, map-types, and map-access/update functions for three selected analyses: (i) value-set analysis (VSA, see §4.1.1), (ii) def-use analysis (DUA, see §4.1.4), and (ii) quantifier-free bit-vector semantics (QFBV, see §4.1.5). Each interpretation defines an abstract domain. For example, line 3 of each column defines the abstract-domain class for INT32: ValueSet32, UseSet, and QFBVTerm32, respectively. To define an abstract domain and its operations, one needs to define 42 basetype operators, most of which have four variants, for 8-, 16-, 32-, and 64-bit integers,[8] as well as 12 map

[8]Interpretations are implemented in C++, which supports templates. To save work one generally implements 42 function templates, which are each instantiated four times.

Table II. Semantics of the abstract transformers created using the TSL system.

Analysis	Generated Transformers for "add ebx, eax"
VSA	λS.S[ebx $\mapsto$ S(ebx)+$^{vsa}_{32}$S(eax)] [ZF $\mapsto$ (S(ebx)+$^{vsa}_{32}$S(eax) = 0)][*more flag updates*]
DUA	[ebx $\mapsto$ {eax, ebx}, ZF $\mapsto$ {eax, ebx}, . . .]
QFBV	(ebx′ = ebx+$_{32}$eax) $\wedge$ (ZF′ $\Leftrightarrow$ (ebx+$_{32}$eax = 0)) $\wedge$ (SF′ $\Leftrightarrow$ (ebx+$_{32}$eax< 0)) $\wedge$ . . .

access/update operations and a few additional operations, such as *join*, *meet*, and *widen*.

The gray boxes in the middle of Fig. 4 represent abstract domains defined in this way. Each abstract domain is combined with the CIR (via C++ template instantiation) to create a component that performs abstract interpretation of machine-code instructions. The entry point for abstract interpretation is the reinterpreted version of the function interpInstr that was written by the ISS developer.

3.1.4.1 *Generated Transformers.* Consider the instruction "add ebx, eax", which causes the sum of the values of the 32-bit registers ebx and eax to be assigned into ebx. When Fig. 7(b) is instantiated with the three interpretations from Tab. I, the execution of lines 17–30 of Fig. 7(b) implement the three transformers that are presented (using mathematical notation) in Tab. II.

3.1.4.2 *Reinterpretation for Relational Abstract Domains.* The goal of some reinterpretations, such as affine-relation analysis, is to produce a version of interpInstr that, when applied to an instruction, creates an over-approximation of the *transition relation* for the instruction. In such situations, state is redefined to be a relational abstract-domain class whose values represent a relation between input states and output states.

However, we were initially puzzled about how the TSL basetypes should be redefined for such a relational reinterpretation, as well as how operations such as Plus32, And32, Xor32, etc. should be redefined. The literature on relational numeric abstract domains did not provide much help in figuring out how to create such a TSL reinterpretation. Most papers about relational numeric domains focus on some restricted modeling language—typically affine programs ([Cousot and Halbwachs 1978, §4], [Müller-Olm and Seidl 2004, §2], [Miné 2002, §4])—involving only assignments and tests of a restricted form. They typically describe the abstract transformers for the state transformations supported by the modeling language, but do not describe an operator-by-operator scheme for creating abstract transformers in a compositional fashion. For an assignment statement "x := e" in the original program,

—If e is a non-linear expression, it is modeled as "x := ?" or, equivalently, "havoc(x)". (That is, after "x := e" executes, x can hold any value.)

—If e is a linear expression, the statement is converted into an abstract-domain value that encodes a linear transformation, by examining the coefficient for each variable in e.

In contrast, with TSL each abstract-domain value must be constructed by evaluating an expression in the TSL meta-language. Moreover, an ISS developer often has to make use of non-linear operators. Sometimes these occur because the instruction performs a non-linear transformation of its input values, but in other situations it is

necessary to perform shifting or masking operations with respect to constant values to access certain bits of an input value, or bits of the instruction itself. Thus, there would be an unacceptable loss of precision if *every* use of a non-linear operator in an instruction's semantic definition caused the instruction's abstract semantics to be treated as a havoc transformation.

Fortunately, we were able to devise a method that can retain some degree of precision for some occurrences of non-linear operators [Elder et al. 2013, §6]. The method, which we sketch out below, was devised for use with an affine-equality domain; however, the presentation below is domain-neutral, and thus applies to other relational numeric abstract domains.

For a set of variables V, the type Rel$[V]$ denotes the set of assignments to V. When V and V' are disjoint sets of variables, the type Rel$[V; V']$ denotes the set of Rel values over variables $V \cup V'$. Rel$[V; V']$ could also be written as Rel$[V \cup V']$, but because V and V' generally denote the pre-state and post-state variables, respectively, the notation Rel$[V; V']$ emphasizes the different roles of V and V'. We extend this notation to cover singletons: if i is a single variable not in V, then the type Rel$[V; i]$ denotes the set of Rel values over the variables $V \cup \{i\}$. (Operations sometimes introduce additional temporary variables, in which case we have types like Rel$[V; i, i']$ and Rel$[V; i, i', i'']$.)

In a reinterpretation that yields abstractions of concrete transition-relations, the type state represents a relation on pre-states to post-states. For example, suppose that the goal is to tracks relationships among the values of the processor's registers. The abstraction of state would be Rel$[R; R']$, where R is the set of register names (e.g., for Intel IA32, $R \stackrel{\text{def}}{=} \{\texttt{eax}, \texttt{ebx}, \dots\}$), and R' is the same set of names, distinguished by primes ($R' \stackrel{\text{def}}{=} \{\texttt{eax'}, \texttt{ebx'}, \dots\}$).

In contrast, the abstraction of a machine-integer type, such as INT32, becomes a relation on pre-states to machine integers. Thus, for machine-integer types, we introduce a fresh variable i to hold the "current value" of a reinterpreted machine integer. Because R still refers to the pre-state registers, we write the type of a Rel-reinterpreted machine integer as Rel$[R; i]$. Although technically we are working with relations, for a Rel$[R; i]$ value it is often useful to think of R as a set of *independent variables* and i as the *dependent variable*.

Constants. The Rel reinterpretation of a constant c is the Rel$[V; i]$ value that encodes the constraint $i = c$.

Variable-Access Expressions. The Rel reinterpretation of a variable-access expression access(S, v), where S's value is a Rel state-transformer of type Rel$[V; V']$ and $v \in V$, is the Rel$[V; i]$ value obtained as follows:

(1) Extend S to be a Rel$[V; V'; i]$ value, leaving i unconstrained.

(2) Assume the constraint $i = v'$ on the extended S value.

(3) Project away V', leaving a Rel$[V; i]$ value, as desired.

Addition. Suppose that we have two Rel$[V; i]$ values x and y, and wish to compute the Rel$[V; i]$ value for the expression $x + y$. We proceed as follows:

(1) Rename y's i variable to i'; this makes y a Rel$[V; i']$ value.

(2) Extend both x and y to be Rel$[V; i, i', i'']$ values, leaving i' and i'' unconstrained in x, and i and i'' unconstrained in y.
(3) Compute $x \sqcap y$.
(4) Assume the constraint $i'' = i + i'$ on the value computed in step (3).
(5) Project away i and i', leaving a Rel$[V; i'']$ value.
(6) In the value computed in step (5), rename i'' to i, yielding a Rel$[V; i]$ value, as desired.

The vocabulary manipulations in steps (1) and (2) put the values into comparable form (i.e., Rel$[V; i, i', i'']$), and are easy to perform, as is the renaming in step (6). Adding the constraint $i'' = i + i'$ in step (4) is straightforward, assuming that the abstract domain can represent an affine constraint. The main abstract-domain operations that we rely on are meet and project in steps (3) and (5), respectively.

Non-Linear Operations. A Rel domain based on affine constraints cannot interpret most instances of non-linear operations precisely. However, when the dependent variable i of a Rel$[V; i]$ value can have only a single concrete value c, regardless of the values of the independent variables, the Rel$[V; i]$ value represents the constant c. When an operation's input Rel$[V; i]$ values each denote a constant, the operation can be performed precisely by performing it in concrete arithmetic on the identified constants. In essence, a reinterpretation can have special-case handling to identify constants and perform constant propagation.

Moreover, the notion of "constant value" can be generalized to a class of *partially-constant* values. A variable is partially constant if some bits of the dependent variable are constant across all valid assignments to the independent variables, even though overall the dependent variable might take on multiple values. For instance, we can show that i is partially constant in the states that satisfy the affine constraint $i + 12x + 8y = 3 \pmod{2^{32}}$. If we consider these values modulo 4, we would have $i + 0x + 0y = 3 \pmod 4$. Consequently, the rightmost two bits of i must both be 1, even though the leftmost thirty bits of i depend on the values of x and y.

For more information about how partially-constant values can be used to interpret some non-linear operations in a sound way, see Elder et al. [2013, §6.2].

3.1.4.3 *Usage of TSL-Generated Analysis Components.* Fig. 10 depicts how the CIR is connected to an analysis engine. The gray call-out box in Fig. 10 represents an analysis engine (or "solver") written by a tool developer to implement some exploration of the state space of a program. The white call-out box in Fig. 10 represents the CIR instantiated with an abstract domain to create a TSL-generated abstract interpreter for instructions, interpInstr$^\sharp$. The analysis engine shown in Fig. 10 uses classical worklist-based propagation of abstract states in which the TSL-generated transformer interpInstr$^\sharp$ is repeatedly invoked with an instruction and the current abstract state$^\sharp$ S. On each iteration of the main loop of the solver, changes would be propagated to successors—or predecessors, depending on the propagation direction of the analysis—if the value of new_S differs from the value of S.

§4.1 discusses more about how different kinds of analysis engines have been created, using various abstract-interpreters for machine-code instructions that reinterpretation developers have created using TSL.

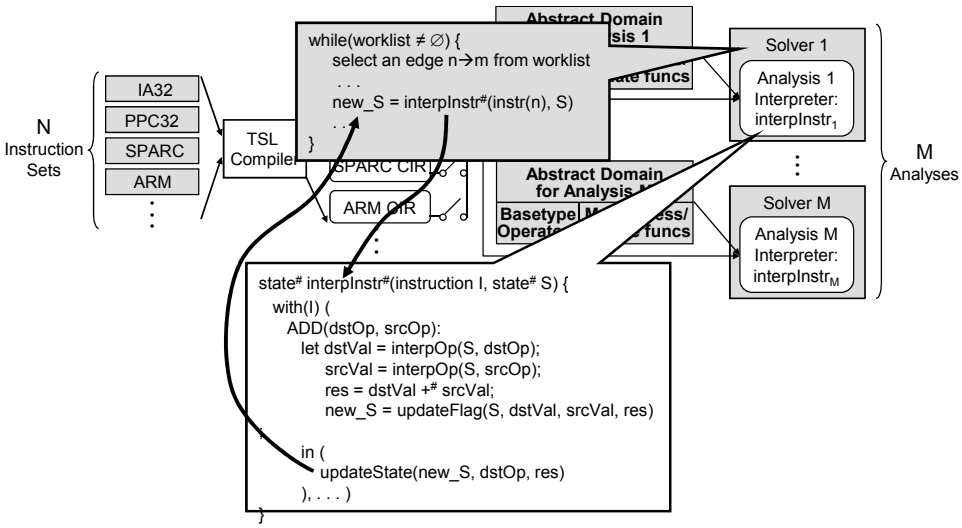

Fig. 10. How a TSL-generated abstract interpreter for instructions, interpInstr♯, is invoked by an analysis engine that performs classical worklist-based propagation of abstract states.

3.1.5 *Pragmatic Issues.*

Validation. In principle, a generator tool produces an artifact that is correct by construction. However, in practice there can be bugs both in the input specification supplied to the generator, as well as in the generator itself. Consequently, it is also important to carry out validation through testing.

D. Melski at GrammaTech, Inc. led a project to validate the TSL IA32 specification (as compiled by the TSL compiler and the C++ compiler), along with the ISAL-generated IA32 disassembler. The technique used is described by Cok [2010, §10.2], and is similar to one developed by Martignoni et al. [2012]. It uses concolic execution of the disassembler and the TSL-based emulator to generate a set of single-instruction programs and interesting input states for those programs. It then runs each instruction on the generated input twice—using the emulator and the physical processor—and checks for discrepancies in the two outputs. The technique uncovered a number of errors, including errors in the ISAL specification of the IA32 disassembler, the TSL specification of the IA32 semantics, and the TSL "reinterpretation" that implements the concrete semantics of TSL. The technique has also been adapted to test some of the abstract-domain implementations for soundness and unexpected losses of precision.

Undefined or Under-Specified Semantics. One feature of instruction sets for commercial processors is that they often have operations with under-specified semantics. For instance, in the IA32 instruction set, the "adjust flag" (AF) is set to 1 if a carry or borrow is generated from the four least-significant bits (the "low nibble"). The Intel Software Developer's Manual [Intel a; b] says that for the `and/or/xor` instructions, the value of the AF flag is undefined.

There are several possible approaches to undefined or under-specified semantics: (i) leave the value unchanged; (ii) set the value to some chosen quantity; (iii)

set the value using TSL's "random(T)" construct. In the concrete semantics, random(T) nondeterministically chooses a value of type T. In an abstract semantics, the reinterpretation of random(T) is the top value ($\top_T$) of the abstract domain that represents T values; $\top_T$ represents the set of all values of type T.

Unfortunately, these approaches are not always satisfactory: a particular implementation of a processor family generally will have some specific deterministic semantics for operations with under-specified semantics. For instance, as part of the validation project discussed above, the team at GrammaTech found that for all IA32 CPU versions that they tested, the AF flag is set to 0 by the `and`/`or`/`xor` instructions. If the goal is to create an emulator that provides bit-level compatibility with the processor, one has to resort to reverse-engineering the semantics (e.g., by testing) and then incorporating the discovered behavior into the TSL specification.

3.2 More About the Common Intermediate Representation

Given a TSL specification of an instruction set, the TSL system generates a CIR that consists of two parts: one is a list of C++ classes for the user-defined abstract-syntax grammar; the other is a list of C++ template functions for the user-defined functions, including the interface function interpInstr. The C++ functions are generated by linearizing the TSL specification, in evaluation order, into a series of C++ statements, as illustrated by Fig. 7(b). However, there are several issues—discussed below—that need to be properly handled for the resulting code to be suitable for abstract interpretation via semantic reinterpretation.

—§3.2.1 concerns the basic properties needed so that the code can be executed over an abstract domain.

—§3.2.2 discusses a technique that is needed with some generated abstract transformers to side-step a loss of precision during abstract interpretation that would otherwise occur.

—§3.2.3 presents the *paired-semantics* facility that the TSL system provides.

3.2.1 *Execution over an Abstract Domain.* There are four basic properties that the CIR code must support so that it can be executed over an abstract domain. In particular, the code generated for each transformer must be able to

(1) execute over abstract values and abstract states,

(2) possibly propagate abstract values to more than one successor in a conditional expression,

(3) compare abstract states and terminate abstract execution when a fixed point is reached, and

(4) apply widening operators, if necessary, to ensure termination.

3.2.1.1 *Conditional Expressions.* Fig. 11 shows part of the CIR that corresponds to the TSL expression "let answer = cond ? a : b". Bool3 is an abstract domain of Booleans (which consists of three values {TRUE, FALSE, MAYBE}, where MAYBE means "may be TRUE or may be FALSE"). The TSL conditional expression is translated into three if-statements (lines 3–7, lines 8–12, and lines 13–15 in Fig. 11). The body of the first if-statement is executed when the Bool3 value for cond is

```
[1]  INTERP::BOOL t0 = . . . ; // translation of a
[2]  INTERP::INT32 t1, t2, answer;
[3]  if(Bool3::possibly_true(t0.getBool3Value())) {
[4]      . . .
[5]      t1 = . . . ; // translation of a
[6]      answer = t1;
[7]  }
[8]  if(Bool3::possibly_false(t0.getBool3Value())) {
[9]      . . .
[10]     t2 = . . . ; // translation of b
[11]     answer = t2;
[12] }
[13] if(t0.getBool3Value() == Bool3::MAYBE) {
[14]     answer = t1.join(t2);
[15] }
```

Fig. 11. The translation of the conditional expression "let answer = cond ? a : b".

possibly true (i.e., either TRUE or MAYBE). Likewise, the body of the second if-statement is executed when the Bool3 value for cond is possibly false (i.e., either FALSE or MAYBE). The body of the third if-statement is executed when the Bool3 value for cond is MAYBE. Note that in the body of the third if-statement, answer is overwritten with the *join* of t1 and t2 (line 14).

The Bool3 value for the translation of a TSL BOOL-valued value is fetched by getBool3Value, which is one of the TSL interface functions that each interpretation is required to define for the type BOOL. Each analysis developer decides how to handle conditional branches by defining getBool3Value. It is always sound for getBool3Value to be defined as the constant function that always returns MAYBE. For instance, this constant function is useful when Boolean values cannot be expressed in an abstract domain, such as DUA for which the abstract domain for BOOL is a set of *use*s. For an analysis where Bool3 is itself the abstract domain for type BOOL, such as VSA, getBool3Value returns the Bool3 value from evaluating the translation of a so that either an appropriate branch or both branches can be abstractly executed.

3.2.1.2 *Comparison, Termination, and Widening.* Recursion is not often used in TSL specifications, but is useful for handling some instructions that involve iteration, such as the IA32 string-manipulation instructions (STOS, LODS, MOVS, etc., with various REP prefixes), and the PowerPC multiple-word load/store instructions (LMW, STMW, etc). For these instructions, the amount of work performed is controlled either by the value of a register, the value of one or more strings, etc. These instructions can be specified in TSL using recursion.[9] For each recursive function, the TSL system generates a function that appropriately compares abstract values and terminates the recursion if abstract values are found to be equal (i.e., the recursion has reached a fixed point). The function is also prepared to apply the widening operator that the reinterpretation developer has specified for the abstract domain in use.

For example, Fig. 12(a) shows the user-defined TSL function that handles "rep

[9]Currently, TSL supports only linear tail-recursion.

```
[1] state repMovsd(state S, INT32 count) {
[2]   count == 0
[3]    ? S
[4]    : with(S) (
[5]        State(mem, regs, flags):
[6]          let direction = flags(DF());
[7]              edi = regs(EDI());
[8]              esi = regs(ESI());
[9]              src = MemAccess_32_8_LE_32(mem, esi);
[10]             newRegs = direction
[11]                ? regs[EDI()|->edi-4][ESI()|->esi-4]
[12]                : regs[EDI()|->edi+4][ESI()|->esi+4]
[13]             newMem = MemUpdate_32_8_LE_32(
[14]                memory, edi, src);
[15]             newS = State(newMem, newRegs, flags);
[16]         in ( repMovsd(newS, count - 1) )
[17]    )
[18]};
```

(a)

```
[1] state global_S;
[2] INTERP::INT32 global_count;
[3] state global_retval;
[4] state repMovsd(
[5]   Istate S, INTERP::INT32 count) {
[6]    global_S = ⊥;
[7]    global_count = ⊥;
[8]    global_retval = ⊥;
[9]    return repMovsdAux(S, count);
[10]};
[11]state repMovsdAux(
[12]  state S, INTERP::INT32 count) {
[13]   // Widen and test for convergence
[14]   state tmp_S = global_S ∇ (global_S ⊔ S);
[15]   INTERP::INT32 tmp_count =
[16]            global_count ∇ (global_count ⊔ count);
[17]   if(tmp_S ⊑ global_S
[18]     && tmp_count ⊑ global_count) {
[19]     return global_retval;
[20]   }
[21]   S = tmp_S; global_S = tmp_S;
[22]   count = tmp_count; global_count = tmp_count;
[23]
[24]   // translation of the body of repMovsd
[25]   . . .
[26]   state newS = . . . ;
[27]   state t = repMovsdAux(newS, count - 1);
[28]   global_retval = global_retval ⊔ t;
[29]   return global_retval;
[30]};
```

(b)

Fig. 12. (a) A recursive TSL function, (b) The translation of the recursive function from (a). For simplicity, some mathematical notation is used, including ⊔ (join), ∇ (widening), ⊑ (approximation), and ⊥ (bottom).

movsd", which copies the contents of one area of memory to a second area.[10] The amount of memory to be copied is passed into the function as the argument count. Fig. 12(b) shows its translation in the CIR. A recursive function like repMovsd (Fig. 12(a)) is automatically split by the TSL compiler into two functions, repMovsd (line 4 of Fig. 12(b)) and repMovsdAux (line 11 of Fig. 12(b)). The TSL system initializes appropriate global variables global_S and global_count (lines 6–8) in repMovsd, and then calls repMovsdAux (line 9). At the beginning of repMovsdAux, it generates statements that widen each of the global variables with respect to the arguments, and test whether all of the global variables have reached a fixed point (lines 13–17). If so, repMovsdAux returns global_retval (line 19). If not, the body of repMovsdAux is analyzed again (lines 24–27). Note that at the translation of each normal return from repMovsdAux (e.g., line 28), the return value is joined into global_retval. The TSL system requires each reinterpretation developer to define the functions *join* and *widen* for the basetypes of the interpretation used in the analysis.

Although TSL recursion is translated to C++ recursion in the CIR, the presence of widening operators prevents the nesting level from becoming excessively deep. (TSL supports a NOWIDEN directive, which suppresses the compilation pattern described above. However, NOWIDEN should be used only in situations in which the ISS

[10] repMovsd is called by interpInstr, which passes in the value of register ecx, and sets ecx to 0 after repMovsd returns.

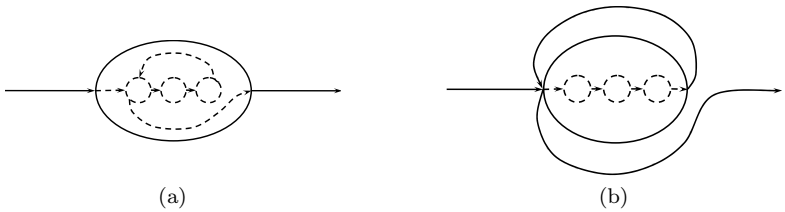

Fig. 13. Conversion of a recursively defined instruction—portrayed in (a) as a "microcode loop" over the actions denoted by the dashed circles and arrows—into (b), an explicit loop in the control-flow graph whose body is an instruction defined without using recursion. The three microcode operations in (b) correspond to the three operations in the body of the microcode loop in (a).

developer knows that the recursion depth will always be bounded.)

Recursive definitions are problematic for quantifier-free bit-vector semantics (QFBV), which is intended to translate an instruction into a formula that expresses exactly the semantics of the instruction. (See Tab. I, col. 3; Tab. II, line 4; and §4.1.5.) The QFBV-reinterpretation developer would have to define a formula-widening operation, and thus when the QFBV reinterpretation is applied to an instruction whose semantics is specified using recursion, one would only obtain a formula that over-approximates the semantics of the instruction. As discussed in [Lim et al. 2011, §6], we side-stepped this problem by changing the TSL specification so that, e.g., the IA32 string instructions have a semantics in which the instruction itself is one of its two successors. In essence, the "microcode loop" expressed via tail-recursion is converted into an explicit loop (see Fig. 13). With this approach, the "microcode loop" is materialized explicitly, and proper handling of the loop becomes the responsibility of the *tool developer*, rather than the ISS developer and the reinterpretation developer. Typically, the tool developer already has to deal with program-level loops, and so already has the means to handle the "microcode loops"—e.g., the tool would perform widening during classical worklist-based fixed-point finding, or create an appropriate path constraint during symbolic execution.

3.2.2 *Two-Level CIR.* The examples given in Fig. 7(b), Fig. 11, and Fig. 12(b), show slightly simplified versions of CIR code. The TSL system actually generates CIR code in which all the basetypes, basetype-operators, and *access/update* functions are appended with one of two predefined namespaces that define a *two-level* interpretation [Jones and Nielson 1995; Nielson and Nielson 1992]: CONCINTERP for concrete interpretation (i.e., interpretation in the concrete semantics), and ABSINTERP for abstract interpretation. Either CONCINTERP or ABSINTERP would replace the occurrences of INTERP in the example CIR shown in Fig. 7(b), Fig. 11, and Fig. 12(b).

The reason for using a two-level CIR is that the specification of an instruction set often contains some manipulations of values that should always be treated as concrete values. For example, an instruction-set specification developer could follow the approach taken in the PowerPC manual [PowerPC32] and specify variants of the conditional branch instruction (BC, BCA, BCL, BCLA) of PowerPC by interpreting some of the fields in the instruction (AA and LK) to determine which of the four variants is being executed (Fig. 14).

```
[1] // User-defined abstract-syntax grammar
[2] instruction: . . .
[3]   | BCx(BOOL BOOL INT32 BOOL BOOL)
[4]   | . . . ;
[5] // User-defined functions
[6] state interpInstr(instruction I, state S) {
[7]   . . .
[8]   BCx(BO, BI, target, AA, LK):
[9]     let . . .
[10]       cia = RegValue32(S, CIA()); // current address
[11]       new_ia = (AA ? target          // direct: BCA/BCLA
[12]                    : cia + target); // relative: BC/BCL
[13]       lr = RegValue32(S, LR());    // linkage address
[14]       new_lr =
[15]         (LK ? cia + 4 // change the link register: BCL/BCLA
[16]             : lr); // do not change the link register: BC/BCA
[17]    . . .
[18]}
```

Fig. 14. A fragment of the PowerPC specification for interpreting BCx instructions (BC, BCA, BCL, BCLA). For a given instruction, each of BO, BI, target, AA, and LK will have a specific concrete value.

```
[1] AddSubInstr(op, dstOp, srcOp): // ADD or SUB
[2]   let dstVal = interpOp(S, dstOp);
[3]       srcVal = interpOp(S, srcOp);
[4]       ans = (op == ADD() ? dstVal + srcVal
[5]                          : dstVal - srcVal); // SUB()
[6]   in ( . . . ),
[7] . . .
```

Fig. 15. An example of factoring in TSL.

Another reason that this issue arises is that most well-designed instruction sets have many regularities, and it is convenient to factor the TSL specification to take advantage of these regularities when specifying the semantics. Such factoring leads to shorter specifications, but leads to the introduction of auxiliary functions in which one of the parameters holds a constant value for a *given* instruction. Fig. 15 shows an example of factoring. The IA32 instructions `add` and `sub` both have two operands and can share the code for fetching the values of the two operands. Lines 4–5 are the instruction-specific operations; the equality expression "op == ADD()" on line 4 can be (and should be) interpreted in concrete semantics.

In both cases, the precision of an abstract transformer can sometimes be improved by interpreting subexpressions associated with the manipulation of concrete values in concrete semantics.[11] For instance, consider a TSL expression $let\ v\ =\ (b\ ?\ 1 : 2)$ that occurs in a context in which b is definitely a concrete value; v will get a precise value—either 1 or 2—when b is concretely interpreted. However, if b is

[11] For instructions whose semantics does not involve a call to a recursive TSL function, the precision of an abstract transformer can never be made worse by this technique. The widening operation incorporated into the translation of recursive functions (cf. Fig. 12) is not monotonic—i.e., more precision at one point may cause a loss of precision at a later point; thus, technically precision could be worsened in some cases. However, recursion does not play a role in the semantics of the vast majority of instructions (see §3.2.1.2).

(a)

```
[1] template <typename INTERP1, typename INTERP2>
[2] class PairedSemantics {
[3]   typedef PairedBaseType<INTERP1::INT32, INTERP2::INT32> INT32;
[4]   . . .
[5]   INT32 MemAccess_32_8_LE_32(MEMMAP32_8 mem, INT32 addr) {
[6]     return INT32(INTERP1::MemAccess_32_8_LE_32(mem.GetFirst(), addr.GetFirst()),
[7]                  INTERP2::MemAccess_32_8_LE_32(mem.GetSecond(), addr.GetSecond()));
[8]   }
[9] };
```

(b)

```
[1] typedef PairedSemantics<VSA_INTERP, DUA_INTERP> DUA;
[2] template<> DUA::INT32 DUA::MemAccess_32_8_LE_32(
[3]                                 DUA::MEMMAP32_8 mem, DUA::INT32 addr) {
[4]   DUA::INTERP1::MEMMAP32_8 memory1 = mem.GetFirst();
[5]   DUA::INTERP2::MEMMAP32_8 memory2 = mem.GetSecond();
[6]   DUA::INTERP1::INT32 addr1 = addr.GetFirst();
[7]   DUA::INTERP2::INT32 addr2 = addr.GetSecond();
[8]   DUA::INT32 answer = interact(mem1, mem2, addr1, addr2);
[9]   return answer;
[10]}
```

Fig. 16. (a) A part of the template class for paired semantics; (b) an example of C++ explicit template specialization to create a reduced product.

not expressible precisely in a given abstract domain, the conditional expression "$(b\ ?\ 1 : 2)$" will be evaluated by joining the two branches, and v will not hold a precise value. (It will hold the abstraction of $\{1, 2\}$.)

Binding-time analysis. To address the issue, we perform a kind of binding-time analysis [Jones et al. 1993] on the TSL code, the outcome of which is that expressions associated with the manipulation of concrete values in an instruction are annotated with C, and others with A. We then generate the two-level CIR by appending CONCINTERP for C values, and ABSINTERP for A values. The generated CIR is instantiated for an analysis transformer by defining ABSINTERP. The TSL translator supplies a predefined concrete interpretation for CONCINTERP.

The instruction-set-specification developer annotates the top-level user-defined (but reserved) functions, including interpInstr, with binding-time information.

$$\mathsf{EXPORT}\ <A>\ \mathsf{interpInstr}(<C>, <A>)$$

The first argument of interpInstr, of type instruction, is annotated with $<C>$, which indicates that all data extracted from an instruction is treated as *concrete*. The second argument of interpInstr, of type state, is annotated with $<A>$, which indicates that all data extracted from state is treated as *abstract*. The return type is also annotated as $<A>$.

Binding-time information is propagated through a TSL specification until a fixed point is reached, or an inconsistency is identified.

3.2.3 *Paired Semantics.* Our system allows easy instantiations of *reduced products* [Cousot and Cousot 1979] by means of *paired semantics*. The TSL system provides a template for paired semantics as shown in Fig. 16(a).

The CIR is instantiated with a *paired* semantic domain defined with two interpretations, INTERP1 and INTERP2 (each of which may itself be a paired semantic domain), as shown on line 1 of Fig. 16(b). The communication between interpretations may take place in basetype-operators or *access/update* functions; Fig. 16(b) is an example of the latter. The two components of the paired-semantics values are

```
[1] with(op) ( . . .
[2]   Indirect32(base, index, scale, disp):
[3]    let  addr = base
[4]           + index * SignExtend8To32(scale)
[5]           + disp;
[6]         m = MemUpdate_32_8_LE_32(
[7]                   mem,addr,v);
[8] . . .)
```

Fig. 17. A fragment of updateState.

deconstructed on lines 4–7 of Fig. 16(b), and the individual INTERP1 and INTERP2 components from *both* inputs can be used (as illustrated by the call to *interact* on line 8 of Fig. 16(b)) to create the paired-semantics return value, answer. Such overridings of basetype-operators and *access/update* functions are done by C++ explicit specialization of members of class templates (this is specified in C++ by "template<>"; see line 2 of Fig. 16(b)).

We also found this method of CIR instantiation to be useful to perform a form of reduced product when analyses are split into multiple phases, as in a tool like CodeSurfer/x86. CodeSurfer/x86 carries out many analysis phases, and the application of its sequence of basic analysis phases is itself iterated. On each round, CodeSurfer/x86 applies a sequence of analyses: VSA, DUA, and several others. VSA is the primary workhorse, and it is often desirable for the information acquired by VSA to influence the outcomes of other analysis phases by pairing the VSA interpretation with another interpretation.

We can use the paired-semantics mechanism to obtain desired *multi-phase interactions* among our generated analyzers—typically, by pairing the VSA interpretation with another interpretation. For instance, with DUA_INTERP alone, the information required to obtain abstract memory location(s) for addr is lost because the DUA basetype-operators (used for + and $*$ on lines 4–5 of Fig. 17) just return the union of the arguments' *use* sets. With the pairing of VSA_INTERP with DUA_INTERP (line 1 of Fig. 16(b)), DUA can use the abstract address computed for addr2 (line 7 of Fig. 16(b)) by VSA_INTERP, which uses VSA_INTERP::Plus32 and VSA_INTERP::Times32; the latter operators operate on a numeric abstract domain (rather than a set-based one).

Note that during the application of the paired semantics, VSA interpretation will be carried out on the VSA component of paired intermediate values. In some sense, this is duplicated work; however, a paired semantics is typically used only in a phase of transformer generation where the transformers are generated during a single pass over the interprocedural CFG to generate a transformer for each instruction. Thus, only a limited amount of VSA evaluation is performed (equal to what would be performed to check that the VSA solution is a fixed point).

3.3 Leverage

The TSL system provides two dimensions of parameterizability: different instruction sets and different analyses. Each instruction-set specification developer writes an instruction-set semantics, and each reinterpretation developer defines an abstract domain for a desired analysis by giving an interpretation (i.e., the implementations of TSL basetypes, basetype-operators, and *access/update* functions). Given the in-

puts from these two classes of users, the TSL system automatically generates an analysis component. Note that the work that an analysis developer performs is TSL-specific but *independent* of each language to be analyzed; from the interpretation that defines an analysis, the abstract transformers for that analysis can be generated automatically for *every* instruction set for which one has a TSL specification. Thus, to create $M \times N$ analysis components, the TSL system only requires M specifications of the concrete semantics of instruction sets, and N analysis implementations (Fig. 1), i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

The TSL system provides considerable leverage for implementing analysis tools and experimenting with new ones. New analyses are easily implemented because a clean interface is provided for defining an interpretation.

TSL as a Tool Generator. A tool generator (or tool-component generator) such as YACC [Johnson 1975] takes a declarative description of some desired behavior and automatically generates an implementation of a component that behaves in the desired way. Often the generated component consists of generated tables and code, plus some unchanging *driver* code that is used in each generated tool component. The advantage of a tool generator is that it creates correct-by-construction implementations.

For machine-code analysis, the desired components each consist of a suitable abstract interpretation of the instruction set, together with some kind of analysis driver (a solver for finding the fixed-point of a set of dataflow equations, a symbolic evaluator for performing symbolic execution, etc.). TSL is a system that takes a description of the concrete semantics of an instruction set, a description of an abstract interpretation, and creates an implementation of an abstract interpreter for the given instruction set.

TSL : concrete semantics $\times$ abstract domain $\rightarrow$ abstract semantics.

In that sense, TSL is a tool generator that, for a fixed instruction-set semantics, automatically creates different abstract interpreters for the instruction set.

The reinterpretation mechanism allows TSL to be used to implement *tool-component generators* and *tool generators*. Each implementation of an analysis component's driver (e.g., fixed-point-finding solver, symbolic executor) serves as the unchanging driver for use in different instantiations of the analysis component for different instruction sets. The TSL language becomes the specification language for retargeting that analysis component for different instruction sets:

analyzer generator = abstract-semantics generator + analysis driver.

For tools like CodeSurfer/x86 and MCVETO [Thakur et al. 2010], which incorporate multiple analysis components, we thereby obtain YACC-like tool generators for such tools:

concrete semantics of L $\rightarrow$ Tool/L.

Consistency. In addition to leverage and thoroughness, for a system like CodeSurfer/x86—which uses multiple analysis phases—automating the process of creating abstract transformers ensures *semantic consistency*; that is, because analysis implementations are generated from a *single* specification of the instruction set's concrete semantics, this guarantees that a *consistent* view of the concrete

semantics is adopted by all of the analyses used in the system.

4. APPLICATIONS

The capabilities of the TSL system have been demonstrated by writing specifications for both the IA32 and PowerPC instruction sets, and then automatically creating a variety of abstract interpreters from each of the specifications—including dynamic-analysis components, static-analysis components, and symbolic-analysis components.

In this section, we present various abstract interpreters generated using TSL, as well as summarize various program-analysis tools developed by using the TSL-generated abstract interpreters.

4.1 TSL-Generated Abstract Interpreters

As illustrated in Fig. 10, a version of the interface function interpInstr is created for each reinterpretation supplied by a reinterpretation designer. At appropriate moments, each analysis engine calls interpInstr with an instruction being processed either (i) to perform one step of abstract interpretation, or (ii) to obtain an abstract transformer. Analysis engines can be categorized as follows:

— *Worklist-based value propagation (or Transformer application)* [TA]. These perform classical worklist-based value propagation in which generated transformers are applied, and changes are propagated to successors/predecessors (depending on propagation direction). Context sensitivity in such analyses is supported by means of the call-string approach [Sharir and Pnueli 1981]. VSA uses this kind of analysis engine (§4.1.1).

— *Transformer composition* [TC]. These generally perform flow-sensitive, context-sensitive interprocedural analysis. ARA (§4.1.2) uses this kind of analysis engine.

— *Unification-based analyses* [UB]. These perform flow-insensitive interprocedural analysis. ASI (§4.1.3) uses this kind of analysis engine.

For each analysis, the CIR is instantiated with an interpretation provided by a reinterpretation developer. This mechanism provides wide flexibility in how one can couple TSL-generated components to an external package. One approach, used with VSA, is that the analysis engine (written in C++) calls interpInstr directly. In this case, the instantiated CIR serves as a *transformer evaluator*: interpInstr is prepared to receive an instruction and an abstract state, and return an abstract state. Another approach, used in both ASI and in TC analyzers, is employed when interfacing to an analysis framework that has its own input language for specifying abstract transformers. In this case, the instantiated CIR serves as an *abstract-transformer generator*: interpInstr is prepared to receive an instruction and an abstract transformer as the state parameter (often the identity function); interpInstr returns an abstract-transformer specification in the analysis component's input language.

The following subsections discuss how the CIR is instantiated for various analyses.

4.1.1 *Creation of a TA Transformer Evaluator for VSA.* VSA is a combined numeric-analysis and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses that each register and memory location holds at each program point [Balakrishnan and Reps 2004]. A

memory region is an abstract quantity that represents all runtime activation records of a procedure. To represent a set of numeric values and addresses, VSA uses *value-set*s, where a value-set is a map from memory regions to strided intervals. A strided interval consists of a lower bound lb, a stride s, and an upper bound $lb + ks$, and represents the set of numbers $\{lb,\ lb + s,\ lb + 2s,\ ...,\ lb + ks\}$ [Reps et al. 2006].

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a value-set. The abstract domain for BOOL is Bool3 ({TRUE, FALSE, MAYBE}), where MAYBE means "may be FALSE or may be TRUE". The operators on these domains are described in detail in [Reps et al. 2006].

—*The interpretation of map-types and access/update functions.* An abstract value for a memory map (MEMMAP32_8, MEMMAP64_8, etc.) is a dictionary that maps each abstract memory location (i.e., the abstraction of INT32) to a value-set. An abstract value for a register map (REGMAP32, REGMAP64, etc.) is a dictionary that maps each variable (reg32, reg64, etc.) to a value-set. An abstract value for a flag map (FLAGMAP) is a dictionary that maps a flag to a Bool3. The *access/update* functions access or update these dictionaries.

VSA uses this transformer evaluator to create an output abstract state, given an instruction and an input abstract state. For example, row 1 of Tab. II shows the generated VSA transformer for the instruction "add ebx, eax". The VSA evaluator returns a new abstract state in which ebx is updated with the sum of the values of ebx and eax from the input abstract state and the flags are updated appropriately.

4.1.2 *Creation of a TC Transformer Generator for ARA.* An affine relation is a linear-equality constraint between integer-valued variables. ARA finds affine relations that hold in the program, for a given set of variables. This analysis is used to find induction-variable relationships between registers and memory locations; these help in increasing the precision of VSA when interpreting conditional branches [Balakrishnan 2007].

As discussed in §3.1.4.2, the principle that is used to create a TC transformer generator is as follows: by interpreting the TSL expression that defines the semantics of an individual instruction using an abstract domain in which abstract-basetype values represent state-to-*value* transformations, each call to interpInstr will residuate an abstract-domain value that over-approximates the instruction's state-to-*state* transformation. To work with the ARA domain of Müller-Olm and Seidl [2005] (ARA-MOS), the CIR is instantiated so that for each instruction, the generated transformer is a set of matrices that represent affine transformations of the value in registers and memory locations of the instruction's input state.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of linear expressions in which variables are either a register or an abstract memory location—the actual representation of the domain is a set of *column*s that consist of an integer constant and an integer coefficient for each variable. This column represents an affine expression over the values that the variables hold at the beginning of the instruction. The basetype operations are defined so that only a set of linear expressions can be generated;

any operation that leads to a non-linear expression, such as Times32(eax, ebx), returns TOP, which means that no affine relationship is known to hold.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a set of matrices of size $(N+1) \times (N+1)$, where N is the number of variables. This abstraction, which is able to find all affine relationships in an affine program, was defined by Müller-Olm and Seidl [2005]. Each *access* function extracts a set of columns associated with the variable it takes as an argument, from the set of matrices for its map argument. Each *update* function creates a new set of matrices that reflects the affine transformation associated with the update to the variable in question.

For each instruction, the ARA-MOS transformer relates linear-equality relationships that hold before the instruction to those that hold after execution of the instruction.

We have also worked with an alternative abstract domain of affine relations, called ARA-KS [Elder et al. 2011; 2013] (see also §5.4 and §6.3.2). As shown by Elder et al. [2011], ARA-MOS and ARA-KS are incomparable: each domain is capable of representing properties that the other domain cannot represent. Reinterpretation for ARA-KS follows the scheme summarized in §3.1.4.2; more complete details of the reinterpretation method used for ARA-KS are given by Elder et al. [2013, §6].

4.1.3 *Creation of a UB Transformer Generator for ASI.* ASI is a unification-based, flow-insensitive algorithm to identify the structure of aggregates in a program [Balakrishnan and Reps 2007]. For each instruction, the transformer generator generates a set of ASI commands, each of which is either a command to *split* a memory region or a command to *unify* some portions of memory (and/or some registers). At analysis time, a client analyzer typically applies the transformer generator to each of the instructions in the program, and then feeds the resulting set of ASI commands to an ASI solver to refine the memory regions.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of *dataref*s, where a *dataref* is an access on specific bytes of a register or memory. The arithmetic, logical, and bit-vector operations tag *dataref*s as *non-unifiable dataref*s, which means that they will only be used to generate *split*s.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a set of *split*s and *unification*s. The *access* functions generate a set of *dataref*s associated with a memory location or register. The *update* functions create a set of *unification*s or *split*s according to the *dataref*s of the data argument.

For example, for the instruction "mov [ebx],eax", when ebx holds the abstract address AR_foo-12, where AR_foo is the memory region for the activation records of procedure *foo*, the ASI transformer generator emits one ASI *unification* command "AR_foo[-12:-9] :=: eax[0:3]".

4.1.4 *Def-Use Analysis (DUA).* *Def-Use* analysis finds the relationships between *definitions* (*defs*) and *uses* of state components (registers, flags, and memory-locations) for each instruction.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of *uses* (i.e., abstractions of the map-keys in states, such as registers, flags, and abstract memory locations), and the operators on this domain perform a set union of their arguments' sets.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a dictionary that maps each *def* to a set of *uses*. Each *access* function returns the set of *uses* associated with the key parameter. Each *update* function $update(D, k, S)$, where D is a dictionary, k is one of the state components, and S is a set of *uses*, returns an updated dictionary $D[k \mapsto (D(k) \cup S)]$ (or $D[k \mapsto S]$ if a strong update is sound).

The DUA results (e.g., row 2 of Tab. II) are used to create transformers for several additional analyses, such as GMOD analysis [Cooper and Kennedy 1988], which is an analysis to find modified variables for each function f (including variables modified by functions transitively called from f) and live-flag analysis, which is used in our version of VSA to perform trace-splitting/collapsing (see §4.1.5).

4.1.5 *Quantifier-Free Bit-Vector (QFBV) Semantics.* QFBV semantics provides a way to obtain a symbolic representation of an instruction's semantics as a logical formula in a language accepted by a modern SMT solver, such as Yices [Dutertre and de Moura 2006] or Z3 [de Moura and Bjørner 2008]—i.e., quantifier-free, first-order logic with the theories of equality, bit-vectors, arrays, and uninterpreted functions.

—*The interpretation of basetypes and basetype-operators.* An abstract value for an integer-basetype value is a set of terms, and each operator constructs a term that represents the operation. An abstract value for a BOOL value is a formula, and each BOOL-valued operator constructs a formula that represents the operation.

—*The interpretation of map-types and access/update functions.* An abstract value for a map is a dictionary that maps a storage component to a term (or a formula in the case of FLAGMAP). The *access/update* functions retrieve from and update the dictionaries, respectively.[12]

QFBV semantics is useful for a variety of purposes. One use is as auxiliary information in an abstract interpreter, such as the VSA analysis engine, to provide more precise abstract interpretation of branches in low-level code. The issue is that many instruction sets provide separate instructions for (i) setting flags (based on some condition that is tested) and (ii) branching according to the values held by flags.

To address this problem, we use a *trace-splitting/collapsing* scheme [Mauborgne and Rival 2005]. The VSA analysis engine partitions the state at each flag-setting instruction based on live-flag information (which is obtained from a backwards

[12]The purpose of using a dictionary is to represent a conjunction of equalities in a way that provides indexed access to individual equalities. That is, a dictionary $[x \mapsto \sigma(x, y), y \mapsto \tau(x, y)]$ represents the formula $x' = \sigma(x, y) \wedge y' = \tau(x, y)$. However, the equalities in such a formula are not general equalities: they are of the form "current-state variable = term over pre-state variables". For the sake of efficiency during symbolic execution, one wants to have quick access to the terms that represent the symbolic values of the current-state variables x and y (i.e., $\sigma(x, y)$ and $\tau(x, y)$, respectively).

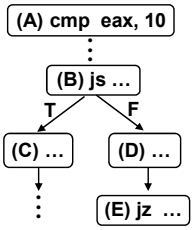

Fig. 18. An example for trace-splitting

analysis that uses the DUA transformers). A semantic reduction [Cousot and Cousot 1979] is performed on the split VSA states with respect to a formula obtained from the transformer generated by the QFBV semantics. The set of VSA states that result are propagated to appropriate successors at the branch instruction that uses the flags.

The cmp instruction (A) in Fig. 18, which is a flag-setting instruction, has `sf` and `zf` as live flags because those flags are used at the branch instructions js (B) and jz (E): js and jz jump according to `sf` and `zf`, respectively. After interpretation of (A), the state S is split into four states, S_1, S_2, S_3, and S_4, which are reduced with respect to the formulas φ_1: (eax $-$ 10 $<$ 0) associated with `sf`, and φ_2: (eax $-$ 10 == 0) associated with `zf`.

$S_1 := S[\texttt{sf}\mapsto T]\ [\texttt{zf} \mapsto T]\ [\text{eax} \mapsto \mathit{reduce}(S(\text{eax}),\ \varphi_1 \wedge \varphi_2)]$
$S_2 := S[\texttt{sf}\mapsto T]\ [\texttt{zf} \mapsto F]\ [\text{eax} \mapsto \mathit{reduce}(S(\text{eax}),\ \varphi_1 \wedge \neg\varphi_2)]$
$S_3 := S[\texttt{sf}\mapsto F]\ [\texttt{zf} \mapsto T]\ [\text{eax} \mapsto \mathit{reduce}(S(\text{eax}),\ \neg\varphi_1 \wedge \varphi_2)]$
$S_4 := S[\texttt{sf}\mapsto F]\ [\texttt{zf} \mapsto F]\ [\text{eax} \mapsto \mathit{reduce}(S(\text{eax}),\ \neg\varphi_1 \wedge \neg\varphi_2)]$

Because $\varphi_1 \wedge \varphi_2$ is not satisfiable, S_1 becomes $\bot$. State S_2 is propagated to the true branch of js (i.e., just before (C)), and S_3 and S_4 to the false branch (i.e., just before (D)). Because no flags are live just before (C), the splitting mechanism maintains just a single state, and thus all states propagated to (C)—here there is just one—are collapsed to a single abstract state. Because `zf` is still live until (E), the states S_3 and S_4 are maintained as separate abstract states at (D).

4.2 CodeSurfer/x86

TSL has been used to create a revised version of CodeSurfer/x86 in which the TSL-generated analysis components for VSA, ARA, ASI, QFBV, etc. are put together. CodeSurfer/x86 runs these analyses repeatedly, either until quiescence, or until some user-supplied bound is reached [Balakrishnan and Reps 2010, §6]. It was necessary to find a way in which the abstraction used in each analysis phase could be encoded using TSL's reinterpretation mechanism, and in several cases, including ARA (§4.1.2) and instruction generation for the stand-alone ASI solver (§4.1.3), we were forced to rethink how to implement a particular analysis. The main reason was that most of the abstract-interpretation literature implicitly or explicitly focuses on some restricted modeling language, e.g., affine programs [Cousot and Halbwachs 1978; Müller-Olm and Seidl 2004] or some special-purpose modeling language [Ramalingam et al. 1999]. As discussed in §3.1.4.2, papers typically describe the abstract transformers for the state transformations supported by the modeling language, but do not describe an operator-by-operator scheme for creating abstract transformers

in a compositional fashion. In particular, relational analyses typically are presented by describing various forms of abstract transition-relations that represent state-to-*state* transformations. While such values can be used to define the reinterpretation of the reserved TSL type state, we found that we had to define new abstractions to represent the state-to-*value* transformations needed to reinterpret TSL basetypes. (For a more complete discussion of these issues in the context of ARA, see Elder et al. [2013, §6].)

In terms of precision, the replacement analysis components created using TSL generally have the same quality, or are of better quality as the manually-written ones.

—For some analysis components (e.g., VSA) there is no qualitative difference between the original component and the replacement component. For instance, the implementation of the abstract domain of strided intervals from CodeSurfer/x86 was wrapped to implement the 42×4 operations for the TSL-based version. In both cases, the state-space-exploration component is essentially the same worklist-based algorithm for finding a fixed-point of a set of equations (incorporating widening). In the original CodeSurfer/x86, the right-hand side of each equation is the *ad hoc* VSA interpretation of an instruction; in the TSL-based version of CodeSurfer/x86, the right-hand side of each equation is the TSL-generated VSA reinterpretation of the interpInstr function of the TSL specification of the IA32 instruction set.

—For other analysis components, the TSL-based component provides essentially the same functionality, but generally in a more principled way. For instance, in the original CodeSurfer/x86, the VSA interpretation of conditional-branch instructions used an *ad hoc* method to find branch conditions [Balakrishnan and Reps 2010, §3.4.2]. In the TSL-based version of CodeSurfer/x86, that technique was replaced with the trace-splitting method discussed in §4.1.5.

§5.3 presents a more detailed comparison for an abstract domain of affine relations for modular arithmetic, called ARA-MOS [Müller-Olm and Seidl 2005]. The original CodeSurfer/x86 used a hand-coded method for creating ARA-MOS abstract transformers [Balakrishnan 2007, §7.2]. A TSL-generated component for creating ARA-MOS abstract transformers [Lim 2011, §3.3.2] was developed for use in the TSL-based version of CodeSurfer/x86.

4.3 MCVETO

Using the TSL system, we developed a model checker for machine code, called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol). MCVETO uses *directed proof generation* [Gulavani et al. 2006], which maintains two approximations of a program P's state space:

—A set T of *concrete traces*, obtained by running P with specific inputs. T under-approximates P's state space.

—A graph G, called the *abstract graph*, obtained from P via abstraction (and abstraction refinement). G over-approximates P's state space.

MCVETO finds either an input that causes a (bad) target state to be reached, or a proof that the bad state cannot be reached. (The third possibility is that MCVETO

fails to terminate.)

What distinguishes the work on MCVETO is that it addresses a large number of issues that have been ignored in previous work on software model checking, and would cause previous techniques to be unsound if applied to machine code. The contributions of our work on MCVETO can be summarized as follows:

(1) MCVETO can verify safety properties of machine code while avoiding a host of assumptions that are unsound in general, and that would be inappropriate in the machine-code context, such as reliance on symbol-table, debugging, or type information, and preprocessing steps for (a) building a precomputed, fixed, interprocedural control-flow graph (ICFG), or (b) performing points-to/alias analysis.

(2) MCVETO does not require static knowledge of the split between code vs. data, and uses a sound approach to decoding/disassembly. In particular, MCVETO does not have to be prepared to decode/disassemble collections of instructions, nested branches, loops, procedures, or the whole program all at once, which is what can confuse conventional disassemblers [Linn and Debray 2003]. Instead, MCVETO performs *trace-based disassembly*: when MCVETO runs the program on a new input to obtain a new trace, instructions are identified one at a time by decoding the current bytes of memory, starting at the current value of the program counter, using an ISAL-generated decoder. Because decoding is performed during concrete execution, each indirect jump or indirect call encountered can be resolved to a specific address. This approach, along with some other techniques described in [Thakur et al. 2010], allows MCVETO to build and maintain a (sound) abstraction of the program's state space.

    MCVETO can analyze programs with instruction aliasing[13] because it builds its two abstractions of the program's state space entirely on-the-fly. Moreover, MCVETO is capable of verifying (or detecting flaws in) self-modifying code. With self-modifying code there is no fixed association between an address and the instruction at that address, but this is handled automatically by MCVETO's mechanisms for trace-based disassembly and abstraction refinement. To the best of our knowledge, MCVETO is the first model checker to handle self-modifying code.

(3) MCVETO uses a language-independent algorithm that we developed to identify the aliasing condition relevant to a property in a given state. Unlike previous techniques [Beckman et al. 2008], it applies when static names for variables/objects are unavailable.

(4) MCVETO employs several novel techniques to refine the abstraction in use. Our techniques go beyond those that have been used in other tools that perform directed proof generation.

We were able to develop MCVETO in a language-independent way by using the TSL system to implement the analysis components needed by MCVETO—i.e., (a)

[13] Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream [Linn and Debray 2003].

an emulator for running tests, (b) a primitive for performing symbolic execution, and (c) a primitive for the pre-image operator. In addition, we developed language-independent approaches to the issues discussed above (e.g., item (3)). TSL-generated analysis components, including those for VSA, ARA, and ASI, were used for item (4).

As discussed in §3, the TSL system acts as a "YACC-like" tool for creating versions of MCVETO for different instruction sets: given an instruction-set description, a version of MCVETO is generated automatically. We created two such instantiations of MCVETO from descriptions of the IA32 and PowerPC instruction sets. The details of MCVETO can be found in the full paper ([Thakur et al. 2010]).

4.4 BCE

A substantial number of computers on the Internet have been compromised and have had software installed on them that make them part of a botnet. A typical way to analyze the behavior of a bot executable is to run it and observe its actions. To carry this out, however, one needs to identify inputs that trigger possibly malicious behaviors. Using TSL, we developed a tool for extracting botnet-command information from a bot executable.

The tool, called BCE (Botnet-Command Extractor) [Lim and Reps 2010], analyzes an executable without access to source code or symbol-table/debugging information for the executable. BCE is parameterized to take a list of library or system calls of interest (which we will refer to as "API calls"). The goals of BCE are three-fold:

(1) to identify inputs ("commands") that trigger possibly malicious behaviors,

(2) to extract information from a bot executable about botnet commands and the arguments to commands, and

(3) to provide information about the sequences of API calls controlled by each command, which can help the user understand the bot's command structure.

BCE is able to provide such information without running the bot explicitly. Instead, BCE uses directed test generation [Godefroid et al. 2005] (also known as "concolic execution"), enhanced with a new search technique that uses control-dependence information [Ferrante et al. 1987] to direct BCE's search.

As with MCVETO, the BCE implementation is structured so that it can be retargeted to different languages easily. The BCE driver only needs to be provided with implementations of concrete execution and symbolic execution for a language. We used the TSL-generated primitives for concrete execution and symbolic execution of x86 machine code.

The specific information extracted by BCE consists of information about the actual parameters of each API call, which are sometimes constant values, but more generally take the form of a symbolic expression for each actual parameter (obtained via concolic execution). The TSL-generated symbolic-analysis primitives allow BCE to obtain an accurate path constraint from the conditional branches along an explored path. A path constraint provides additional restrictions on the items that appear in the symbolic expressions—and hence provides additional information about the inputs that can trigger malicious behaviors.

Instruction Characteristics		
Kind	# instruction instances	# different opcodes
ordinary	12,734	164
lock prefix	2,048	147
rep prefix	2,143	158
repne prefix	2,141	154
full corpus	19,066	179

Fig. 19. Some of the characteristics of the corpus of 19,066 (non-privileged, non-floating point, non-mmx) instructions.

The information about inputs also includes type information (obtained by mapping type information from the prototype of an API call back to the portions of the input string on which the argument values of the API call depend). Finally, along with each command, BCE provides the sequence of API calls that are controlled by the command.

Our experiments showed that BCE's search method, in which concolic execution was enhanced by using control-dependence information, provides higher coverage of the parts of the program relevant to identifying bot commands than standard concolic execution—as well as lower overall execution time.

More details about BCE can be found in a technical report [Lim and Reps 2010].

5. EVALUATION

In this section, we present an evaluation of the costs and benefits of the TSL approach. The material discussed in this section is designed to shed light on the following questions:

—Does the use of TSL help to reduce the development time of a program-analysis tool that uses abstract interpretation?

—How does a TSL-generated method for creating abstract transformers compare with a hand-coded transformer-creation method, in terms of the running time needed to create transformers, as well as the precision of the transformers created?

—In terms of running time, space consumption, and precision, how does the TSL-based method for generating abstract transformers for a given abstract domain $\mathcal{A}$ stack up against a method for creating best abstract transformers [Cousot and Cousot 1979] for $\mathcal{A}$?

In §5.2, we present estimates of the time spent developing two different versions of CodeSurfer/x86.

In §5.3 and §5.4, we report on experiments to evaluate the precision of TSL-generated abstract transformers against both hand-written abstract transformers, as well as against best abstract transformers [Cousot and Cousot 1979]. The latter experiment measures the precision of TSL-generated abstract transformers against the limit of precision obtainable using a given abstraction.

Program name	Measures of size			
	instrs	procs	BBs	branches
finger	532	18	298	48
subst	1,093	16	609	74
label	1,167	16	573	103
chkdsk	1,468	18	787	119
convert	1,927	38	1,013	161
route	1,982	40	931	243
comp	2,377	35	1,261	224
logoff	2,470	46	1,145	306
setup	4,751	67	1,862	589

Fig. 20. Windows utility applications. The columns show the number of instructions (instrs), the number of procedures (procs), the number of basic blocks (BBs), and the number of branch instructions (branches).

5.1 Experimental Setup

For our experiments at the level of single instructions, we used two corpuses of x86 instruction instances that cover various opcodes, addressing modes, and operand sizes. The corpus used in §5.3 consists of 531 instructions; the corpus used in §5.4 consists of 19,066 instructions. The corpus of 19,066 x86 instructions was created via ISAL (see §2.1). In this case, ISAL was employed in a mode in which the input specification of the concrete syntax of the x86 instruction set is used to create a randomized instruction *generator*—instead of the more normal mode in which ISAL creates an instruction *recognizer*. By this means, we are assured that the corpus has substantial coverage of the syntactic features of the x86 instruction set (including opcodes, addressing modes, and prefixes, such as "lock", "rep", and "repne"); see Fig. 19.

Fig. 20 lists several size parameters of a set of Windows utilities (numbers of instructions, procedures, basic blocks, and branches) that we also used in our experiments. For these programs, the generated abstract transformers were used as "weights" in a weighted pushdown system (WPDS). WPDSs are a modern formalism for solving flow-sensitive, context-sensitive interprocedural dataflow-analysis problems [Reps et al. 2005; Bouajjani et al. 2003].[14] In our experiments, all WPDSs were constructed using the WALi package for WPDSs [WALi 2007]. The weight on each WPDS rule encodes the abstract transformer for a basic block of the program, including a jump or branch to a successor block.

The asymptotic cost of weight generation is linear in the size of the program: to generate the weights, each basic block in the program is visited once, and a weight is generated by the relevant method: (i) TSL-based reinterpretation of interpInstr to create a weight directly, or (ii) TSL-based reinterpretation of interpInstr to create a QFBV formula that captures the concrete semantics of the block, followed by application of an algorithm for creating the best weight from the formula [Elder et al. 2011; Thakur et al. 2012] (i.e., the best abstract transformer [Cousot and Cousot 1979]). We used EWPDS merge functions [Lal et al. 2005] to preserve

[14]Running a WPDS-based analysis to find the join-over-all-paths value for a given set of program points involves calling two operations, "post*" and "path_summary", as detailed in [Reps et al. 2005].

caller-save and callee-save registers across call sites. The post* queries used the FWPDS algorithm [Lal and Reps 2006].

For each example program, the number of WPDS rules equals the number of basic blocks plus the number of branches (see Fig. 20), together with rules for modeling procedure call and return.

Due to the high cost in §5.4 of constructing WPDSs with best-transformer weights, we ran all WPDS analyses without the code for libraries. Values are returned from x86 procedure calls in register `eax`, and thus in our experiments library functions were modeled approximately (albeit unsoundly, in general) by "`eax := ?`", where "`?`" denotes an unknown value (sometimes written as "`havoc(eax)`").

5.2 Development Time

We consider CodeSurfer/x86 to be representative of how TSL can be used to create full-fledged analysis tools; we present our experience developing two different versions of CodeSurfer/x86 as an example of the kind of leverage that TSL provides with respect to development time.

In the original implementation of CodeSurfer/x86 [Balakrishnan et al. 2005; Balakrishnan and Reps 2010], the abstract transformers were implemented in the conventional way (i.e., by hand-writing routines to generate an abstract transformer from an instruction's abstract-syntax tree).[15] The most recent incarnation of CodeSurfer/x86—a revised version whose analysis components are implemented via TSL—uses eight separate abstract-interpretation phases, each based on a different abstract-transformer generator created from the TSL specification of the IA32 instruction set by supplying an appropriate reinterpretation of the basetypes, maptypes, and operations of the TSL meta-language. We estimate that the task of hand-writing an abstract-transformer generator for the eight analysis phases used in the original CodeSurfer/x86 consumed about *twenty man-months*; in contrast, we have invested a total of about *one man-month* to write the C++ code for eight TSL interpretations that are used to generate the replacement components. To this, one should add *10–20 man-days* to write the TSL specification for IA32: the current specification for IA32 consists of 4,153 (non-comment, non-blank) lines of TSL. We conclude that TSL can greatly reduce the time to develop the abstract interpreters needed in a full-fledged analysis tool—perhaps as much as 12-fold (= 20 months/1.67 months).

Because each abstract interpretation is defined at the meta-level (i.e., by providing an interpretation for the collection of TSL primitives), an abstract-transformer generator for a given abstract interpretation can be created automatically for *each* instruction set that is specified in TSL. For instance, from the PowerPC specification (1,862 non-comment, non-blank lines, which took approximately 4 days to write), we were immediately able to generate PowerPC-specific versions of *all* of the abstract-interpretation phases that had been developed for the IA32 instruction set.

It takes approximately 1 minute (on a single core of a single-CPU, quad-core, 2.83 GHz Intel machine, running Linux 2.6.32 with 8 GB of memory, configured so a user process has 4 GB of memory) for the TSL (cross-)compiler to compile the

[15] The ideas used in the abstract transformers, and how the various abstract-interpretation phases fit together, are discussed in [Balakrishnan and Reps 2010, §5].

IA32 specification to C++. It then takes approximately 4 minutes wall-clock time (on a single core of a single-CPU, quad-core, 3.0 GHz Xeon machine with 32 GB of RAM, running 64-bit Windows 7 Enterprise with Service Pack 1) to compile the generated C++ for each CIR instantiation, using Visual Studio 2010.

5.3 Comparison to Hand-Written Analyses

This section compares the running time and precision of a TSL-generated method for creating abstract transformers against a hand-coded transformer-creation method. It discusses precision both from the standpoint of (i) the abstract transformers obtained for individual instructions, and (ii) the dataflow facts (i.e., program invariants) obtained from running an abstract interpreter using the two different sets of abstract transformers.

In this section, the comparison is performed for the abstract domain of affine relations for modular arithmetic developed by Müller-Olm and Seidl [2005] (ARA-MOS), where the variables over which affine relations are inferred are the x86 registers. The original CodeSurfer/x86 used a hand-coded method for creating ARA-MOS abstract transformers [Balakrishnan 2007, §7.2]. A TSL-generated component for creating ARA-MOS abstract transformers [Lim 2011, §3.3.2] was developed for use in the TSL-based revision of CodeSurfer/x86.

5.3.1 *Running Time.* We compared the running time needed to obtain ARA-MOS transformers via a TSL-generated method versus the hand-coded method, for a corpus of 531 instruction instances that cover various opcodes, addressing modes, and operand sizes. The experiment was run on a single core of a single-CPU, quad-core, 3.0 GHz Xeon machine with 32 GB of RAM, running Windows XP, configured so that a user process has 4 GB of memory. We measured total transformer-generation time taken for obtaining 531 ARA-MOS transformers via the two approaches. Overall, the TSL-based abstract-transformer-generation method takes about 15 times longer than the hand-coded approach.

There are several factors that contribute to the amount of the measured slowdown. First, as discussed further in §5.3.2, the TSL-generated transformers are "more thorough" than the hand-coded ones. Second, the ARA-MOS reinterpretation is similar to the reinterpretation scheme described in §3.1.4.2, which is fairly costly, and in some sense represents a worst case for TSL. Third, there are some differences between the implementations of the ARA-MOS domain used in the two systems. In particular, the version used in the TSL-based implementation of transformer generation uses a key matrix-normalization operation [Elder et al. 2011, Alg. 1] that should take roughly twice as long as the one used in the hand-coded transformer-generation routine [Müller-Olm and Seidl 2005, §2].

(The reason we performed the ARA-MOS comparison on the smaller, 531-instruction corpus is because CodeSurfer/x86 first constructs a CFG before invoking any of its analyses. As mentioned earlier, the 19,066-instruction corpus was automatically generated based on the syntax of the x86 instruction set; because the jump instructions were not generated with an eye towards creating a program, CodeSurfer/x86 builds a CFG that has many disconnected or ill-structured components, and its version of ARA-MOS analysis fails to process all 19,066 instructions. The 531-instruction corpus does contain jump instructions, but was massaged by

hand so that CodeSurfer/x86 builds a CFG in which all instructions are reachable from the entry node.)

5.3.2 *Precision per Instruction.* It is also natural to ask how the two approaches stack up from the standpoint of the precision of the transformers obtained. Due to the nature of the ARA-MOS transformers, it is easy to make an exact comparison of the relative precision of two transformers. (Four answers are possible: the first is more precise, the second is more precise, equal, or incomparable.)

On our corpus of 531 instruction instances, we found that the transformers obtained via the TSL-generated method for creating abstract transformers were equivalent to the transformers obtained using the hand-coded transformer-creation method in 328 cases and *more precise* in the remaining 203 cases (38%). For 83 of the 203 cases, the increase in precision was because in rethinking how the ARA-MOS abstraction could be encoded using TSL's reinterpretation mechanism, we discovered an easy way to extend the method of Müller-Olm and Seidl [2005] to retain some information for 8-, 16-, and 64-bit operations. (In principle, the improved approach could have been incorporated into the hand-coded version, too.)

The other 120 cases of improvement can be ascribed to "fatigue factor" on the part of the human programmer: the hand-coded versions adopted a pessimistic view and just treated certain instructions as having the abstract transformer $\lambda x.\top$, which is obviously a safe over-approximation. Because the TSL-generated transformers are based on the ARA-MOS interpretation's definitions of the TSL basetype-operators, the TSL-generated transformers were more thorough: a basetype-operator's definition in an interpretation is used in *all* places that the operator arises in the specification of the instruction set's concrete semantics.

5.3.3 *Precision in Client-Analysis Results.* Even though the TSL-based approach creates more precise abstract transformers than the approach that uses a hand-coded transformer-generation method, one might wonder whether there is actually a difference in the dataflow facts (program invariants) obtained by running an abstract interpreter using the two different sets of abstract transformers. To answer this question, we compared the precision of the ARA-MOS results obtained using a flow-sensitive, context-sensitive, interprocedural analysis of the examples from Fig. 20 using the ARA-MOS abstract transformers generated by the two methods.

For the Windows utilities listed in Fig. 20, Fig. 21 shows the result of our experiment to find one-vocabulary affine relations at blocks that end with branch instructions. The table presents the number of such points at which both methods produce equivalent (one-vocabulary) affine relations, the number at which the hand-coded ARA-MOS produces more precise results, the number at which ARA-MOS based on TSL-generated weights produces more precise results, and the number at which the two methods produce incomparable results.

ARA-MOS based on TSL-generated weights identifies more precise (one-vocabulary) affine relations in all of the examples, except subst and label. On average (computed as a geometric mean), the use of ARA-MOS weights generated by TSL led to a precision improvement at 26.3% of blocks that end with branch instructions.

Manual inspection revealed that the cases of incomparable results shown in the

Program	Precision			
	Equivalent	Hand-coded ARA-MOS better	TSL ARA-MOS better	Incomparable
finger	46	0	2	0
subst	54	*8	0	*1
label	93	0	0	*10
chkdsk	92	0	26	*1
convert	93	0	60	*8
route	189	0	35	*19
comp	213	0	4	*7
logoff	153	0	136	*17
setup	542	0	34	*13

Fig. 21. WPDS experiments for §5.3. The columns show the number of basic blocks that end with branch instruction at which both methods produce equivalent affine relations, the number at which the hand-coded ARA-MOS produced more-precise results, the number at which the TSL-generated ARA-MOS produced more-precise results, and the number at which the two methods produced incomparable results.

last column, as well as the eight cases in subst in which the hand-coded ARA-MOS transformer-generation method produced more-precise results, are all apparently due to bugs in the implementation of the hand-coded ARA-MOS transformer-generation method.

5.4 Comparison to the Best Abstract Transformer

As described in §3, for a given abstract domain $\mathcal{A}$, an over-approximating abstract transformer for an instruction is obtained by defining an over-approximating reinterpretation of each TSL basetype operator as an operation over $\mathcal{A}$. The desired set of abstract transformers are obtained by extending the reinterpretation to TSL expressions and functions, including interpInstr.

However, that method abstracts each TSL operation in isolation, and is therefore rather myopic. In some cases, one can obtain a more precise transformer by considering the semantics of an *entire* instruction (or, even better, an entire basic block or other loop-free fragment). It is known how to give a *specification* of the most-precise abstract interpretation for a given abstract domain [Cousot and Cousot 1979]: for a Galois connection defined by abstraction function α and concretization function γ, the *best abstract transformer* $\tau^{\sharp}_{\text{best}}$ for a given concrete transformer τ, defined by

$$\tau^{\sharp}_{\text{best}} \stackrel{\text{def}}{=} \alpha \circ \tau \circ \gamma, \tag{1}$$

specifies the limit of precision obtainable using the given abstraction.

Unfortunately, Eqn. (1) is non-constructive in general; however, for some abstract domains, algorithms are known for finding best abstract transformers [Graf and Saïdi 1997; Reps et al. 2004; King and Søndergaard 2010; Elder et al. 2011].

To see how a method for constructing best transformers can yield better results than TSL's operator-by-operator reinterpretation approach, consider the following example:

Example 5.1. ([Thakur et al. 2012]) The x86 instruction "`add bh,al`" adds the value of `al`, the low-order byte of 32-bit register `eax`, to `bh`, the second-to-lowest byte of 32-bit register `ebx`. The semantics of this instruction can be expressed in

quantifier-free bit-vector (QFBV) logic as follows:

$$\varphi_I \stackrel{\text{def}}{=} \texttt{ebx}' = \begin{pmatrix} (\texttt{ebx}\ \&\ \texttt{0xFFFF00FF}) \\ |\ ((\texttt{ebx} + 256 * (\texttt{eax}\ \&\ \texttt{0xFF}))\ \&\ \texttt{0xFF00}) \end{pmatrix} \wedge \texttt{eax}' = \texttt{eax},$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively. Note that the semantics of the instruction involves non-linear bit-masking operations.

Now suppose that the abstract domain is the domain of affine relations over integers mod 2^{32} [Müller-Olm and Seidl 2005; Elder et al. 2011]. For this abstract domain, the best transformer is $(2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax})$, which captures the relationship between the low-order two bytes of `ebx` and the low-order byte of `eax`. It is the best over-approximation to φ_I that can be expressed as an affine relation. In contrast, with TSL's operator-by-operator reinterpretation approach, the abstract transformer obtained from the TSL specification of "`add bh,al`" would be $(\texttt{eax}' = \texttt{eax})$, which loses all information about `ebx`. Such a loss of precision is exacerbated when considering larger loop-free blocks of instructions. □

To compare the TSL operator-by-operator approach with best abstract transformers, we carried out a study using an algorithm for obtaining best abstract transformers for an abstract domain of affine relations. For a given basic block B, the TSL QFBV reinterpretation was used to obtain a formula φ_B that captures the entire semantics of B. The formula φ_B was then used to obtain the best ARA transformer that over-approximates φ_B, using the algorithm described in [Elder et al. 2011; Thakur et al. 2012].

As in §5.3, the comparison is again based on an abstract domain of affine relations for modular arithmetic, where the variables over which affine relations are inferred are the x86 registers. However, because no algorithm is known for finding best transformers for the affine-relation domain of Müller-Olm and Seidl [2005] (ARA-MOS), we used a related affine-relation domain (ARA-KS) for which such an algorithm is known, called $\widehat{\alpha}_{\text{KS}}$ [Elder et al. 2011; Thakur et al. 2012]. $\widehat{\alpha}_{\text{KS}}$ uses an SMT solver as a subroutine, making repeated calls on the SMT solver to find satisfying assignments. In our implementation, we used Yices [Dutertre and de Moura 2006]. Given a QFBV formula φ_B for the semantics of basic block B, $\widehat{\alpha}_{\text{KS}}(\varphi_B)$ is guaranteed to obtain the best KS abstract transformer for B, unless a timeout occurs during one of the calls that $\widehat{\alpha}_{\text{KS}}$ makes on the SMT solver.

We also performed a WPDS-based analysis of the examples from Fig. 20 using three sets of ARA-KS weights (abstract transformers): (i) weights created using the TSL reinterpretation method applied to ARA-KS; (ii) weights corresponding to best ARA-KS transformers, and (iii) weights generated using the "generalized Stålmarck" algorithm [Thakur and Reps 2012], which provides another point in the design space of trade-offs between time and precision that lies somewhere between (i) and (ii).[16]

[16]To be more precise about the relationship, we used the following "chained" method for generating weights:

(1) "Reinterpretation" is the TSL ARA-KS reinterpretation method [Elder et al. 2013, §6].

(2) "Stålmarck" is the generalized-Stålmarck algorithm of Thakur and Reps [2012], *starting with the value obtained via the Reinterpretation method.* The generalized-Stålmarck algorithm successively over-approximates the best transformer from above. By starting the algorithm

Note that, except for cases in which an SMT solver timeout is reported, (ii) is guaranteed to find the most-precise basic-block transformers that are expressible in the ARA-KS abstract domain [Elder et al. 2011; Thakur et al. 2012].

The results reported in this section were obtained using a single core of a dual-CPU, quad-core, 2.40 GHz Xeon machine with 12 GB of RAM, running 64-bit Windows 7 Enterprise with Service Pack 1.

5.4.1 *Precision per Instruction.* We compared the precision and running time of the "Reinterpretation" method (TSL ARA-KS reinterpretation) versus the best-transformer method ($\widehat{\alpha}_{\mathrm{KS}}$) at the granularity of individual instructions, using the corpus of 19,066 x86 instructions.

In terms of running time, the $\widehat{\alpha}_{\mathrm{KS}}$ method took about 12.0 times longer than the ARA-KS Reinterpretation method.

With a 1-second timeout limit per Yices invocation, 170 of the 19,066 instructions had Yices timeouts (0.9%). Thus, $\widehat{\alpha}_{\mathrm{KS}}$ was guaranteed to have obtained the best KS abstract transformer on 18,896 instructions (99.1%). The experiment showed that the best-transformer method ($\widehat{\alpha}_{\mathrm{KS}}$) is strictly more precise than the ARA-KS Reinterpretation method for only about 3.2% of the instructions—i.e., for 596 out of the 18,896 (= 18,300 + 596) instances of IA32 instructions for which we knew that $\widehat{\alpha}_{\mathrm{KS}}$ had obtained the best transformer. The two methods created equal transformers for the other 18,300 instructions.[17]

We were also interested in understanding which kinds of instructions exhibited timeouts, and whether increasing the timeout limit had any impact. Fig. 22 provides information about the instruction instances that had timeouts. What stands out is the high percentage of instances of IDIV and DIV instructions that exhibit timeouts. We looked to see what distinguished the instances of IDIV and DIV without timeouts from the ones that exhibit timeouts. It appears that most instances of 32-bit IDIV/DIV instructions had a timeout, whereas most 8-bit and 16-bit instances did not.

It was difficult to get an exact measurement of the effect of changing the timeout limit because Yices is non-deterministic: when invoked multiple times on the same formula with the same timeout limit, some runs may finish and some runs may exceed the timeout limit. To get a feel for the variation in behavior due to non-determinism, we used the 170 instruction instances for which timeouts occurred

---

with the value obtained via the Reinterpretation method, the generalized-Stålmarck algorithm does not have to work its way down from $\top$; it merely continues to work its way down from the over-approximation already obtained via the TSL ARA-KS reinterpretation method. The generalized-Stålmarck algorithm is a faster algorithm than the $\widehat{\alpha}_{\mathrm{KS}}$ method, but is not guaranteed to find the best abstract transformer [Thakur and Reps 2012].

(3) $\widehat{\alpha}_{\mathrm{KS}}$ is the algorithm described in [Elder et al. 2011; Thakur et al. 2012], *starting with the value obtained via the Stålmarck method as an over-approximation* as a way to accelerate its performance. $\widehat{\alpha}_{\mathrm{KS}}$ *is* guaranteed to obtain the best abstract transformer, except for cases in which an SMT solver timeout is reported.

Thus, $\widehat{\alpha}_{\mathrm{KS}} \sqsubseteq$ Stålmarck $\sqsubseteq$ Reinterpretation is always guaranteed to hold.

[17] ARA-KS abstract transformers are values in a particular abstract domain. Two domain values a_1 and a_2 are equal if their concretizations are equal: $\gamma(a_1) = \gamma(a_2)$. Each ARA-KS transformer has a unique normalized representation [Elder et al. 2011], so equality checking is easy, and thus containment can be checked using meet and equality: $a_1 \sqsubseteq a_2$ iff $a_1 = a_1 \sqcap a_2$.

Opcode	#instances	#timeouts	%
IDIV	108	79	73.1
DIV	108	71	65.7
CMPXCHG	475	8	1.7
IMUL	564	5	0.9
XOR	720	3	0.4
CMOVA	114	2	1.8
CMOVL	114	1	0.9
MOVZX	216	1	0.5

Fig. 22. Opcodes of the 170 instruction instances that had Yices timeouts during $\widehat{\alpha}_{\mathrm{KS}}$, when a 1-second timeout limit was imposed. Column two shows the total number of instruction instances that have the given opcode in the corpus of 19,066 x86 instruction instances. Column three shows the number of instruction instances that had a timeout during $\widehat{\alpha}_{\mathrm{KS}}$. Column four gives the percentage of instruction instances that had a timeout during $\widehat{\alpha}_{\mathrm{KS}}$. (All five of the timeout cases for IMUL involved the three-argument variant of the instruction, of which there were 234 instances in the corpus of 19,066 instructions.)

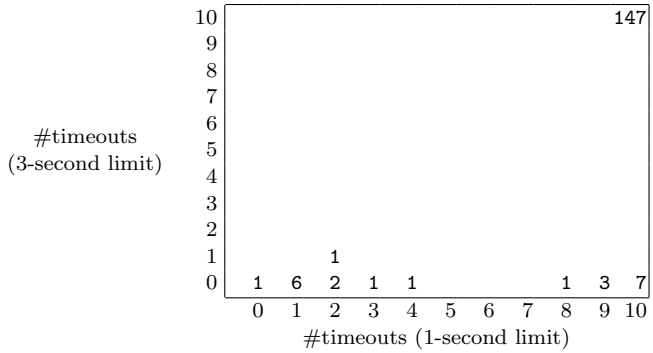

Fig. 23. Distribution of timeouts for the 170 x86 instructions that were "hard" for $\widehat{\alpha}_{\mathrm{KS}}$, based on running each example ten times with a 1-second timeout limit and ten times with a 3-second timeout limit.

when $\widehat{\alpha}_{\mathrm{KS}}$ was applied to the 19,066 instructions with a 1-second timeout limit as a sample set of "hard" instructions for $\widehat{\alpha}_{\mathrm{KS}}$. We then ran $\widehat{\alpha}_{\mathrm{KS}}$ on each example ten times with a 1-second timeout limit, and ten times with a 3-second timeout limit. Fig. 23 shows the distribution of timeouts that we obtained. 23 out of the 170 instructions (13.5%) benefited from the extra time (although one instruction still timed out on one of the ten 3-second-timeout runs). Overall, instructions appear to fall into three categories:

—Instructions that are likely to succeed on invocations of Yices with a 1-second timeout limit, and likely to succeed on invocations with a 3-second limit. (There were 12 such instructions; see the lower left-hand corner.)

—Instructions that are unlikely to succeed on invocations of Yices with a 1-second timeout limit, and likely to succeed on invocations with a 3-second limit. (There were 11 such instructions; see the lower right-hand corner.)

—Instructions that always fail with a 1-second timeout limit, and still always fail with a 3-second limit. (There were 147 such instructions; see the upper right-

Prog.	Performance (x86)										Precision		
	TSL Reinterpretation (RE)			Stålmarck (ST)			$\widehat{\alpha}_{\mathrm{KS}}$				ST $\sqsubset$	$\widehat{\alpha}_{\mathrm{KS}}$ $\sqsubset$	$\widehat{\alpha}_{\mathrm{KS}}$ $\sqsubset$
	WPDS	post*	query	WPDS	post*	query	WPDS	post*	query	t/o	RE	ST	RE
finger	3.781	0.375	0.116	18.258	0.388	0.117	133.085	0.390	0.117	5	14.6%	10.4%	25.0%
subst	5.527	0.685	0.213	23.263	0.702	0.217	207.816	0.705	0.214	3	12.2%	10.8%	17.6%
label	6.106	0.687	0.306	24.314	0.704	0.305	161.458	0.701	0.306	3	0.1%	0%	0.1%
chkdsk	7.201	0.503	0.314	44.973	0.518	0.319	477.471	0.518	0.320	12	10.9%	0%	10.9%
convert	9.688	1.579	0.476	47.642	1.465	0.439	304.772	1.476	0.437	22	30.4%	0%	30.4%
route	17.135	1.779	0.631	61.124	1.855	0.645	462.771	1.864	0.783	7	22.2%	3.7%	25.1%
comp	12.656	1.838	0.585	56.456	1.910	0.522	643.825	1.908	0.595	8	2.7%	0.5%	3.1%
logoff	21.649	2.341	0.767	89.154	2.406	0.781	606.204	2.413	0.793	25	19.3%	5.9%	23.5%
setup	38.872	1.374	1.315	235.016	1.432	1.323	1,730.410	1.454	1.438	68	4.8%	2.2%	5.8%

Fig. 24. WPDS experiments for §5.4. The columns show the times, in seconds, for ARA-KS WPDS construction, performing interprocedural dataflow analysis (running post* and "path_summary"), and finding one-vocabulary affine relations at blocks that end with branch instructions, using three methods: TSL-reinterpretation-based, Stålmarck, and $\widehat{\alpha}_{\mathrm{KS}}$; $\widehat{\alpha}_{\mathrm{KS}}$ has an additional column that reports the number of WPDS rules for which the $\widehat{\alpha}_{\mathrm{KS}}$ weight generation timed out (t/o); the last three columns show the degrees of analysis precision obtained by using (i) Stålmarck-generated ARA-KS weights versus TSL-reinterpretation-based ones, (ii) $\widehat{\alpha}_{\mathrm{KS}}$-generated ARA-KS weights versus Stålmarck-generated ones, and (iii) $\widehat{\alpha}_{\mathrm{KS}}$-generated ARA-KS weights versus TSL-reinterpretation-based ones. (The precision improvements reported in the columns labeled with "A $\sqsubset$ B" are measured as the percentage of basic blocks that (i) end with a branch instruction, and (ii) begin with a node whose inferred one-vocabulary affine relation via method A was strictly more precise than via method B.)

hand corner.)

All of the instructions in the last category were instances of IDIV and DIV. Moreover, of the 150 IDIV and DIV instructions among the 170 instructions, all but three are in the upper-right corner. (One DIV instruction is in cell (9,0); two IDIV instructions are in cell (10,0).)

5.4.2 *Precision in Client-Analysis Results.* Although the $\widehat{\alpha}_{\mathrm{KS}}$ approach does create more precise abstract transformers than TSL ARA-KS reinterpretation, that only happens for about 3.2% of the instructions. We wanted to know what effect the increased precision had on the dataflow facts (program invariants) obtained by running an abstract interpreter using the different sets of abstract transformers. To answer that question, we compared the precision of the ARA-KS results obtained from a flow-sensitive, context-sensitive, interprocedural analysis of the examples from Fig. 20, using ARA-KS abstract transformers generated by TSL ARA-KS reinterpretation, Stålmarck, and $\widehat{\alpha}_{\mathrm{KS}}$, which represent three different points in the design space of trade-offs between time and precision.

For the Windows utilities listed in Fig. 20, Fig. 24 shows the times for constructing ARA-KS abstract transformers, performing interprocedural dataflow analysis (running post* and "path_summary"), and finding one-vocabulary affine relations at blocks that end with branch instructions, using the three methods.

Column 11 of Fig. 24 shows the number of WPDS rules for which $\widehat{\alpha}_{\mathrm{KS}}$ weight-generation timed out. During WPDS construction, if the SMT solver times out, the implementation returns a sound over-approximating weight that the $\widehat{\alpha}_{\mathrm{KS}}$ algorithm has in hand. Because $\widehat{\alpha}_{\mathrm{KS}}$ starts with the weight created by the Stålmarck method, even when there is a timeout, the weight returned by $\widehat{\alpha}_{\mathrm{KS}}$ is never less precise—and sometimes more precise—than the Stålmarck weight. The number of rules is roughly equal to the number of basic blocks plus the number of branches, so a

Prog.	Performance (x86)		
	Reinterp. (RE)	Stålmarck (ST)	$\widehat{\alpha}_{\mathrm{KS}}$
finger	1,140	13,064	10,124
subst	2,296	19,096	14,452
label	2,288	17,848	14,832
chkdsk	2,268	20,264	15,452
convert	4,188	31,004	25,612
route	4,896	32,888	23,524
comp	4,568	32,692	23,952
logoff	5,828	36,604	24,660
setup	7,872	52,308	34,724

Fig. 25. Space usage in WPDS experiments for §5.4. The columns show the maximum amount of memory used, in KB, for ARA-KS WPDS construction, and running post* and "path_summary".

timeout occurred for about 0.4–2.8% of the rules (geometric mean: 1.0%).

The experiment showed that the cost of constructing transformers via the $\widehat{\alpha}_{\mathrm{KS}}$ algorithm is high: creating the ARA-KS weights via $\widehat{\alpha}_{\mathrm{KS}}$ and the Stålmarck-based method are, respectively, about 36.8 times and 4.6 times slower than creating ARA-KS weights using TSL reinterpretation (computed as the geometric mean of the construction-time ratios).

The experiment showed that the TSL ARA-KS reinterpretation method performs quite well in terms of precision, compared to the more precise methods. The precision-improvement numbers show that the $\widehat{\alpha}_{\mathrm{KS}}$ algorithm is strictly more precise than Reinterpretation at about 15.25% of the blocks that end with branch instructions, whereas Stålmarck is strictly more precise than Reinterpretation at about 12.65% of those sites (both computed as geometric means).

5.4.3 *Memory Requirements.* Fig. 25 shows the total space used during ARA-KS WPDS construction, running post*, and "path_summary". The columns of the table in Fig. 25 show the amount of memory used (in KB) while running on 64-bit Windows 7 Enterprise with Service Pack 1. The numbers were obtained by subtracting the value of PageFileUsage just before WPDS construction from the value of PeakPageFileUsage just after path_summary.

5.5 Summary and Discussion

Our evaluation of TSL shows that TSL's reinterpretation approach has a number of benefits and relatively few drawbacks, at least for the two varieties of affine-relation analysis discussed in §5.3 and §5.4 and for the other machine-code abstract interpreters used in CodeSurfer/x86.

—TSL can greatly reduce the time to develop the abstract interpreters needed in a full-fledged analysis tool—perhaps as much as 12-fold (§5.2).

—Because TSL provides a more systematic method for creating abstract transformers compared to a hand-coded transformer-creation method, the use of TSL can result in more precise abstract transformers. However, there may be a performance penalty of 15-fold in the time to create abstract transformers using the TSL-based approach. (An expensive reinterpretation, like the one used for ARA-MOS represents a kind of worst-case scenario for TSL.)

—The abstract transformers obtained via TSL's reinterpretation approach are nearly as precise as *best* abstract transformers. As discussed in §5.4.1, the best transformers are more precise for only about 3.2% of the instruction instances in our corpus of 19,066 instructions. Moreover, it takes about 12 times longer to obtain best transformers, compared to the TSL-based method.

—When best transformers generated from $\widehat{\alpha}_{\mathrm{KS}}$ and Stålmarck are used for interprocedural dataflow analysis, they allow more precise invariants to be discovered at a relatively small number of program points (15.25% and 12.65% of basic blocks that end with branch instructions, respectively, computed as a geometric mean), compared to the invariants discovered using abstract transformers obtained via TSL's reinterpretation approach. The maximum improvement was 30.4% of the basic blocks that end with branch instructions, attained by both $\widehat{\alpha}_{\mathrm{KS}}$ and Stålmarck on one example (convert).

—Analysis times proper (i.e., the running times in columns "post*" and "query") are not much different when using best transformers, compared to using transformers generated via TSL-based reinterpretation. Moreover, there can be a 36.8-fold performance penalty, with respect to the TSL-based approach, for creating best abstract transformers (i.e., during WPDS construction).

—The Stålmarck-based approach provides another point in the design space of trade-offs between time and precision. The transformers obtained using the Stålmarck-based approach produce program invariants that are more precise than those found with the TSL-based transformers, but not as precise as those found with the set of best abstract transformers. Compared to the method for obtaining best abstract transformers, the Stålmarck-based approach has a smaller penalty in running time with respect to TSL's reinterpretation approach (i.e., Stålmarck is only 4.6 times slower).

6. RELATED WORK

In this section, we discuss work on various topics that relate to TSL.

6.1 Instruction-Set Description Languages

There have been many specification languages for instruction sets and many purposes to which they have been applied. Some were designed for hardware simulation, such as cycle simulation and pipeline simulation [Pees et al. 1999; Mishra et al. 2006]. Others have been used to generate an emulator for compiler-optimization testing [Davidson and Fraser 1984; Kästner 2003]. TDL [Kästner 2003] is a hardware-description language that supports the retargeting of back-end phases, such as analyses and optimizations relevant to instruction scheduling, register assignment, and functional-unit binding. The New Jersey machine-code toolkit [Ramsey and Fernández 1997] addresses concrete syntactic issues (instruction decoding, instruction encoding, etc.).

Siewiorek et al. [1982] proposed an operational hardware specification language, called ISP (Instruction-Set Processor) notation, for describing the instructions in a processor and their semantics. The goal of ISP was to automate the generation of software, the evaluation of computer architectures, and the certification of implementations. They divided a computer system into several levels, including the

program level, which the ISP notation is designed to describe. The design of the ISP notation was based on two principles:

(1) The components of the program level are a set of memories and a set of operations. The effect of each instruction can be expressed entirely in terms of the information held in the current set of memories. The ISP notation is designed for specifying that a given operation of a processor is performed on a specific data structure that the set of memories hold.
(2) All data operations can be characterized as working on various data-types; each data-type requires distinct operations to process the values of a data-type. A processor can be completely described at the ISP level by giving its instruction set and its interpreter in terms of its operations, data-types, and memories.

TSL relies on the same principles.

While some of the existing instruction-set description *languages* would have been satisfactory for our purposes, their *runtime environments* were not satisfactory, which was what motivated us to implement our own system. In particular, to meet our goals we needed a mechanism to create abstract interpreters of instruction-set specifications. As discussed in §3.2.1, there are four issues that arise. During the abstract interpretation of each instruction, the abstract interpreter must be able to

—execute over abstract values and abstract states,
—execute both branches of a conditional expression,
—compare abstract states and terminate abstract execution when a fixed point is reached, and
—apply widening operators, if necessary, to ensure termination.

As far as we know, TSL is the first instruction-set specification language with support for such mechanisms.

Functional Languages as Instruction-Set Description Languages. Harcourt et al. used ML to specify the semantics of instruction sets [Harcourt et al. 1994]. LISAS [Cook et al. 1993] is an instruction-set-description language that was subsequently developed based on their experience using ML. Those two approaches particularly influenced the design of the TSL language.

λ-RTL. TSL shares some of the same goals as λ-RTL [Ramsey and Davidson 1999] (i.e., the ability to specify the semantics of an instruction set and to support multiple clients that make use of a single specification). The two languages were both influenced by ML, but different choices were made about what aspects of ML to retain: λ-RTL is higher-order, but without datatype constructors and recursion; TSL is first-order, but supports both datatype constructors and recursion. Recursion is not often used in specifications, but can be used to handle some instructions that involve iteration, such as the IA32 string-manipulation instructions and the PowerPC multiple-word load/store instructions (cf. §3.2.1.2). The choices made in the design and implementation of TSL were driven by the goal of being able to define multiple abstract interpretations of an instruction-sets semantics.

HOL4. A.C.J. Fox, M.O. Myreen, and M.J.C. Gordon at Cambridge University have created a HOL4 specification that simultaneously covers several versions of the

ARM instruction set (including ARMv4, ARMv5, ARMv6, and ARMv7-A) [Fox 2003; Myreen et al. 2007; Fox and Myreen 2010]. These specifications have been the subject of work on correctness models, formal verification carried out using HOL4, and formal specification and simulation using ML. They have also motivated Fox to create his own domain-specific language for specifying instruction set syntax and semantics [Fox 2012].

Fox translated the HOL4 specification of the semantics of the current ARM instruction set, ARMv7-A [ARMv7-A 2013], from HOL4 to TSL. The monadic ARMv7-A specification [Fox and Myreen 2010] was converted by proof into a form that was more amenable for export to TSL. These proofs involved (i) removing the monadic representation used in the ARMv7-A specification, and introducing let-expressions; (ii) converting bit-vector operations from the bit-widths used in the ARMv7-A specification (which includes 4-bit and 11-bit bit-vectors, for instance) to the coarser 8/16/32/64-bit bit-vector types available in TSL; (iii) replacing the HOL4 record-based state representation with state-access operations; and (iv) removing higher-order features by duplicating functions. The result was a collection of theorems representing the definitions of instruction instances. These were pretty-printed as TSL code, and extra boiler-plate was generated for the inductive datatype that specifies the abstract syntax of instructions and for a few other type declarations.

Essentially the same translation process could be applied to translate other HOL4 specifications of instruction sets to TSL; some of the aforementioned steps were automated, although a few (particularly the bit-width adjustments) involved manual guidance.

6.2 Semantic Reinterpretation

As discussed in §2, *semantic reinterpretation* involves refactoring the specification of a language's concrete semantics into a suitable form by introducing appropriate *combinators* that are subsequently redefined to create the different subject-language interpretations.

6.2.1 *Semantic Reinterpretation versus Standard Abstract Interpretation.* Semantic reinterpretation [Mycroft and Jones 1985; Jones and Mycroft 1986; Nielson 1989; Malmkjær 1993] is a form of abstract interpretation [Cousot and Cousot 1977], but differs from the way abstract interpretation is normally applied: in standard abstract interpretation, one reinterprets the constructs of each *subject language*; in contrast, with semantic reinterpretation one reinterprets combinators defined using the *meta-language*. Standard abstract interpretation helps in creating semantically sound *tools*; semantic reinterpretation helps in creating semantically sound *tool generators*. In particular, if you have M subject languages and N analyses, with semantic reinterpretation you obtain $M \times N$ analyzers by writing just $M + N$ specifications: concrete semantics for M subject languages and N reinterpretations. With the standard approach, one must write $M \times N$ abstract semantics.

The MESS compiler generator of Pleban and Lee [1987] aimed to permit the generation of realistic compilers from specifications in denotational semantics. MESS was based on an independent discovery of the principle of semantic reinterpretation: it used a semantic-definition style they called *high-level semantics*, which involves separating the semantic definition of a programming language into two distinct

specifications, called *macro-semantics* and *micro-semantics*. The macro-semantics of a language is defined by a collection of semantic functions that map syntactic phrases compositionally to terms of a semantic algebra; the micro-semantics specifies the meaning of a semantic algebra.

As originally proposed, semantic reinterpretation permits arbitrary refactoring of a semantic specification so that the desired outcome can be achieved via reinterpretation of any combinators introduced. In contrast, although it is possible to introduce combinators in TSL and reinterpret them, the primary mechanism in TSL is to reinterpret the base-types, map-types, and operators of the meta-language. TSL's approach is particularly convenient for a system to generate *multiple* analysis components from a single specification of a language's concrete semantics.

6.2.2 *Semantic Reinterpretation versus Translation to a Universal Assembly Language.* The mapping of subject-language constructs to meta-language operations that one defines as part of the semantic-reinterpretation approach resembles in some ways two other approaches to obtaining "systematic" reinterpretations of subject-language programs, namely,

(1) implementing a translator from subject-language programs to a common intermediate form (CIF) *data structure*, and then creating various interpreters that implement different abstract interpretations of the CIF node types.
(2) implementing a translator from subject-language programs to a universal assembly language (UAL), and then writing different abstract interpreters of the UAL.

An example of a CIF is the Program Intermediate Format of the SUIF compiler infrastructure [Wilson et al. 1994, §2.1]. Recent examples of UALs include Vine [Song et al. 2008, §3], REIL [Dullien and Porst 2009], and BAP [Brumley et al. 2011]. Both approaches can be used to create tools that can be applied to multiple subject languages: each CIF (UAL) abstract interpreter can be applied to the translation of a program written in any subject language L for which one has defined an L-to-CIF (L-to-UAL) translator.

Because UAL programs can be considered to be linearizations of CIF trees, we will confine ourselves to comparing the TSL approach with the UAL approach.

There are four main high-level differences between the semantic-reinterpretation approach used in TSL and the UAL approach. First, there is a difference that affects the activities of instruction-set specifiers: the UAL approach is based on a *translational semantics*, whereas the TSL approach is based on an *operational semantics*. With the UAL approach an instruction-set specifier has to write a function that walks over the abstract-syntax tree of an instruction I of instruction set *IS* and *constructs an appropriate UAL program fragment* whose meaning, when interpreted in the concrete semantics, is the desired semantics for instruction I. In contrast, with TSL one just writes an interpreter that specifies the meaning of an *IS* instruction. (Fig. 7(a), discussed in §3.1.1, shows a fragment of such an interpreter.)

The second and third differences are best explained using Fig. 26.

—The λ expression in Fig. 26(a) formulates the concept underlying the TSL approach. However, in the TSL implementation, the different λ applications are performed in a sequence of phases. Given IS, the definition of a subject lan-

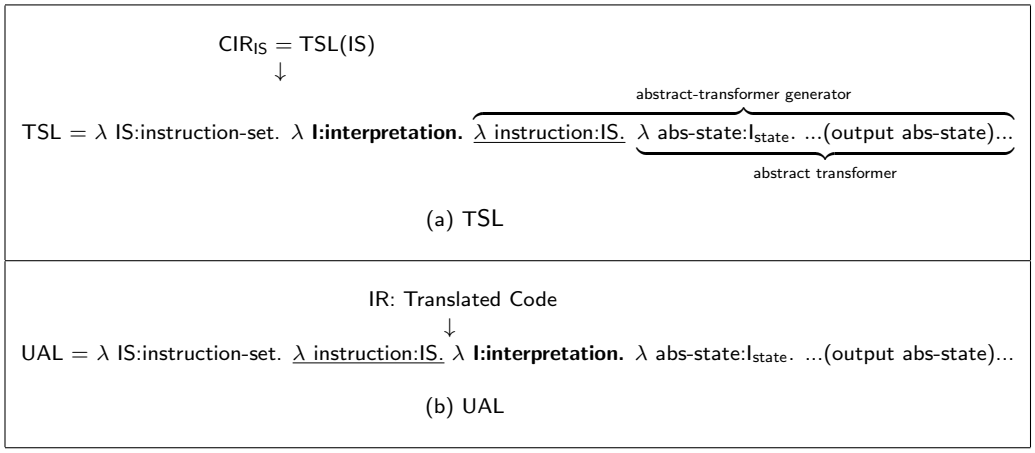

Fig. 26. A comparison of the UAL approach and TSL. (Note the use of dependent-type notation: "instruction:IS" means that instruction is an element of the language IS, and "abs-state:$\mathsf{I}_{\mathsf{state}}$" means that abs-state is a state element in the space of values defined by interpretation I.)

guage's syntax, TSL produces an explicit $\mathsf{CIR}_{\mathsf{IS}}$ artifact as a C++ template. The application of $\mathsf{CIR}_{\mathsf{IS}}$ to an interpretation I is then performed by the C++ compiler, via template instantiation.

Thus, TSL takes instruction set IS and an interpretation I, and produces $\mathsf{TSL}_{\mathsf{IS,I}}$ (e.g., $\mathsf{TSL}_{\mathsf{x86,VSA}}$, $\mathsf{TSL}_{\mathsf{x86,QFBV}}$, or $\mathsf{TSL}_{\mathsf{x86,ARA}}$). In some applications, such as ARA, $\mathsf{TSL}_{\mathsf{IS,ARA}}$ is used as an abstract-transformer generator: given an instruction instr, it produces an ARA abstract-state transformer for instr. In other applications, such as VSA, $\mathsf{TSL}_{\mathsf{IS,VSA}}$ is used to compute output abstract states; it is given an instruction and an input abstract state and returns an output abstract state.

—The UAL method can be formulated (roughly) as UAL in Fig. 26(b). Conceptually, UAL takes IS, the definition of a subject language's syntax, and instruction, and produces $\mathsf{UAL}_{\mathsf{lang,instr}}$ (e.g., $\mathsf{UAL}_{\mathsf{x86,ADD}}$). $\mathsf{UAL}_{\mathsf{lang,instr}}$ takes an interpretation and an abstract state, and produces an output abstract state.

Thus, the second—and perhaps the most enlightening—difference between the UAL and TSL approaches is the order in which the two parameters I:interpretation and instruction:IS are processed. In the TSL approach, I:interpretation is supplied before instruction:IS; with the UAL approach, they are supplied in the opposite order.

The third difference is due to the fact that, in the implementations of the UAL approach that we are aware of, the parameters IS and instruction:IS are processed first, and an intermediate representation IR consisting of the translated UAL code for instruction:IS is emitted as an explicit data object (see Fig. 26(b)). The final stages—e.g., the processing of interpretation I—involve interpreting a UAL code object. In contrast, with the semantic-reinterpretation approach used in TSL there is no explicit UAL program to be interpreted. In essence, with the semantic-reinterpretation approach as implemented in TSL, a level of interpretation is removed, and hence generated abstract interpreters should run faster.

The fourth difference has to do with the resource costs for the processing carried out by the two methods:

(1) With the UAL method, the size of the IR for a given program P is

$$\sum_{\mathsf{instr} \in P} (\text{size of translation of } \mathsf{instr}) \approx |P| \times (\text{avg. size of translated instruction}).$$

In contrast, the size of the TSL CIR is

$$\#\text{interpretations} \times |\text{OpSem}_L|,$$

where OpSem_L is the operational semantics of language L, and "#interpretations" refers to the number of interpretations in use. In other words, the size of the TSL CIR is constant, independent of the size of the subject-language program, whereas the UAL IR is roughly linear in the size of the subject-language program.

(2) The TSL CIR is a C++ template that is instantiated with a reinterpretation to generate an abstract-transformer generator. Thus, one of the disadvantages of the TSL method is that it is necessary to invoke the C++ compiler to instantiate the CIR. With the UAL method, one only has to recompile the reinterpretation itself, which should generally give faster turnaround time during development.

As discussed in [Lim et al. 2011], although the original target for TSL was the reinterpretation of the semantics of instruction sets, we were able to use TSL to reinterpret the semantics of a *logic*. That is, using the existing facilities of TSL, it was easy to define (i) the abstract syntax of the formulas of a logic, (ii) the logic's standard semantics, and (iii) an abstract interpretation of the logic by reinterpreting TSL's basetypes, map-types, and operators. In this case, we used TSL as the function

$$\mathsf{TSL} = \lambda\ \mathsf{L{:}logic}.\ \lambda\ \mathsf{I{:}semantics}.\ \lambda\ \mathsf{formula{:}L}.\ \lambda\ \mathsf{structure}.\ \ldots(\mathsf{meaning})\ldots$$

For machine-code analysis, the appropriate logic was quantifier-free bit-vector arithmetic (QFBV; see §4.1.5). Lim et al. [2011] describes the non-standard semantics for logic that we used as the second argument, and how that allowed us to create a pre-image operation for machine-code instruction sets.

Although it is no doubt possible to duplicate what we did using the UAL approach, it would not be straightforward because existing UAL-based systems appear to be less flexible in the languages that can be defined. In contrast, to apply TSL to a logic, we merely had to define a grammar for QFBV abstract-syntax trees, write an interpreter for the standard semantics of QFBV, and define the relevant reinterpretation of TSL's basetypes, map-types, and operators [Lim et al. 2011].

6.3 Systems for Generating Analyzers

Some systems for representing and analyzing programs are (mainly) targeted for a single language. For instance, SOOT [SOOT] is a powerful and flexible analysis/optimization framework that supports analysis and transformation of Java bytecode. Papers from the SOOT group describe it as a framework for Java program analysis [Lam et al. 2011], but in principle SOOT can support other languages for which front-ends exist that translate to Java bytecode. SOOT provides a universal assembly language, called Jimple, which is used as the basic intermediate form for representing programs. Other intermediate representations, such as control-

flow graphs, call-graphs, and SSA-form, are constructed from a program's Jimple representation.

WALA [WALA] is similar to SOOT, in that it emphasizes a common intermediate form (Common Abstract Syntax Trees), from which multiple additional IRs can be generated (e.g., CFGs and SSA-form). Multiple analyses can then be performed that use these IRs. Several front-ends have been written for WALA, including Java, Javascript, X10, PHP, Java bytecode, and .NET bytecode.

LLVM [Lattner and Adve 2004] is yet another multi-lingual framework similar in spirit to WALA and SOOT. LLVM support a different common intermediate form (LLVM intermediate representation). Languages supported by LLVM include Ada, C, C++, D, Fortran, and Objective-C.

One method to support the retargeting of analyses to different languages is to create a package that supports a family of program analyses that different front-ends can use to create analysis components. Examples include BDDBDDB [Whaley et al. 2005], Banshee [Kodumal and Aiken 2005], APRON [APRON], WPDS++ [WPDS++ 2004], and WALi [WALi 2007]. The writer of each client front-end needs to encode the semantics of his language by creating appropriate transformers for each statement and condition in the subject language's intermediate representation, using the package's API (or input language).

To use such packages in conjunction with TSL, a reinterpretation designer need only use the package to implement the appropriate abstract operations so as to over-approximate the semantics of the operations of the TSL meta-language. Any of the aforementioned packages could be used for creating TSL-based analyses; to date, WALi [WALi 2007] has been used for all of the TC-style analyzers (§4.1.2) that have been developed for use with TSL.

As mentioned in §1, there have been a number of past efforts to create generator tools that support abstract interpretation, including MUG2 [Wilhelm 1981], SPARE [Venkatesh 1989; Venkatesh and Fischer 1992], Steffen's work on harnessing model checking for dataflow analysis [Steffen 1991; 1993], Sharlit [Tjiang and Hennessy 1992], Z [Yi and Harrison, III 1993], PAG [Alt and Martin 1995], and OPTIMIX [Assmann 2000]. In contrast with TSL, in those systems the user specifies the *abstract* semantics of the language to be analyzed, rather than the *concrete* semantics.

Steffen [1991] [Steffen 1993] uses CTL as a specification language for dataflow-analysis algorithms. An advantage of this approach is that it is *declarative*: a specification concerns the program property under consideration, rather than specific details of the analysis algorithm or information about how the properties of interest are determined. However, Steffen's approach does not start from the concrete operational semantics of the language; instead, it starts from an abstraction of the program, which consists of a labeled transition system annotated with a set of atomic propositions. The set of atomic propositions defines the abstract domain in use. Later developments stemming from Steffen's approach include work by Schmidt [1998], Cousot and Cousot [2000], and Lacey et al. [2004].

6.3.1 *Automatic Generation of Sound Abstract Transformers.* There are two analysis systems, TVLA [Lev-Ami and Sagiv 2000; Reps et al. 2010] and RHODIUM [Scherpelz et al. 2007], in which sound analysis transformers are generated au-

tomatically from a concrete operational semantics, plus a specification of an abstraction (either via the abstraction function (TVLA) or the concretization function (RHODIUM)).

—RHODIUM uses a heuristic method for creating sound abstract transformers for parameterized predicate abstraction [Cousot 2003]. Suppose that the meaning of dataflow fact x, when expressed in logic, is some formula $\hat{\gamma}(x)$, and that the goal is to create the abstract transformer for statement S. The RHODIUM method involves two steps:
  —Create the formula $\varphi_x = \mathrm{WLP}(S, \hat{\gamma}(x))$, where WLP is the weakest-liberal-precondition operator.
  —Find a Boolean combination $\psi[\hat{\gamma}(D)]$ of pre-state dataflow facts D that under-approximates φ_x. That is, if $\psi[\hat{\gamma}(D)]$ holds in the pre-state, then φ_x must also hold in the pre-state, and hence $\hat{\gamma}(x)$ must hold in the post-state.

  The abstract transformer is a function that sets the value of x in the post-state according to whether $\psi[\hat{\gamma}(D)]$ holds in the pre-state.

—The method for automatically generating abstract transformers used in TVLA [Reps et al. 2010] is based on finite-differencing of formulas. In TVLA, the abstraction in use is defined by a set of so-called "instrumentation predicates". An instrumentation predicate captures a property that may be possessed by some components of a state and thus distinguishes them from other components of the state. In other words, the set of instrumentation predicates characterizes the distinctions among elements that are observable in the abstraction.

  The problem addressed by Reps et al. [2010] is how to establish the values of post-state instrumentation predicates after the execution of a statement S. Instrumentation predicates are defined by formulas, so the defining formula for a post-state instrumentation predicate p could always be evaluated in the post-state. However, TVLA is based on three-valued logic, in which a third truth value, denoted by 1/2, is introduced to indicate uncertainty (or the absence of information). Evaluating p's defining formula in the post-state often results in 1/2 values, even for values that were "definite" in the pre-state—i.e., either true (1) or false (0)—and could not have been affected by S.

  To overcome such loss of precision, TVLA uses an incremental-updating approach: it copies the pre-state values of an instrumentation predicate to the post-state, and only updates entries that are in the "footprint" of S (which is generally small). In essence, the role of finite differencing in the TVLA approach is to identify the effect of S on S's footprint.

HOIST [Regehr and Reid 2004] also generates abstract transformers from the concrete semantics of a machine-code language. However, the construction of an abstract transformer by HOIST is not done by processing the specification symbolically, as in TVLA and RHODIUM. Instead, HOIST accesses the CPU—either a physical CPU or an emulated one—and runs the instruction on specially selected inputs; from the results of these tests, an abstract transformer is obtained—first in the form of a BDD and then as C code. The paper on HOIST reports that it is "limited to eight-bit machines due to costs exponential in the word size of the target architecture".

The use of semantic reinterpretation in TSL as the basis for generating abstract transformers is what distinguishes our work from TVLA, RHODIUM, and HOIST. In TSL, we rely on the analysis developer to supply sound reinterpretations of the basetypes, map-types, and operators of the TSL meta-language. While this requirement places an additional burden on developers, once an analysis is developed it can be used with each instruction set specified in TSL. Moreover,

—the analyses that we support are much more efficient than those that can be created with TVLA and apply to our intended domain of application (abstract interpretation of machine code).

—some of the analyses that we use, such as ARA-MOS [Müller-Olm and Seidl 2005] and ARA-KS [Elder et al. 2011], appear to be beyond the power of the transformer-generation methods developed for use in TVLA, RHODIUM, and HOIST.

6.3.2 *Automatic Generation of Best Abstract Transformers.* The line of research on generating (or "synthesizing") best abstract transformers was initiated by Reps et al. [2004]. King and Søndergaard [2010] published a somewhat different algorithm for synthesizing best transformers that was specific to a particular domain of affine-equality constraints. Each representable set of points in N-space is the solution of a set of affine-equality constraints. The equality domain used in the experiments reported in §5.4 is essentially the King and Søndergaard domain—whence the name ARA-KS.

The version of the ARA-KS domain defined by King and Søndergaard [2010] was later improved by Elder et al. [2011], who introduced a normal form for representing ARA-KS elements, and used the normal form to give polynomial-time algorithms for "best" versions of join, meet, equality, and projection, as well as for finding best ARA-KS transformers (replacing the over-approximating methods that had been given by King and Søndergaard). In §5.4, the comparison presented is among three methods: a reinterpretation-based method for ARA-KS that fits the TSL model, and two methods for synthesizing ARA-KS transformers, one of which is the Elder et al. method for synthesizing best ARA-KS transformers. The two ends of the spectrum are the reinterpretation-based method—which finds sound ARA-KS transformers—and the Elder et al. method for synthesizing best ARA-KS transformers.

The King and Søndergaard version of the ARA-KS domain has also been used in a method for synthesizing abstract transformers for the interval domain: Brauer and King [2012] uses bit-blasting, plus a Boolean version of the King and Søndergaard domain as a heuristic to synthesize "good" interval transformers.

By now the topic of automatically generating best abstract transformers has begun to attract increasing attention. See Thakur et al. [2012, §6], Brauer and King [2012, §6], and Brauer et al. [2012, §8.1] for additional discussion of work related to the topic.

7. CONCLUSION

Although essentially all program-analysis techniques described in the literature are language-independent, analysis implementations are often tied to a particular language-specific compiler infrastructure. Unlike the situation in source-code analysis, which can be addressed by developing relatively small common intermediate representations, machine-code analysis suffers from the fact that instruction sets

typically have hundreds of instructions and a variety of architecture-specific features that are incompatible with other architectures. With future computing platforms based on multi-core architectures and transactional memory, future runtime environments using just-in-time compiling, future systems providing cloud computing and autonomic computing, plus cell phones and PDAs entering the fray, both (i) interest in establishing that security and reliability properties hold for machine code, and (ii) the variety of computing platforms to analyze will only increase.

To help address these concerns, we have developed improved infrastructure for analyzing machine code. Our work is embodied in the TSL language—a language for describing the semantics of an instruction set—as well as in the TSL run-time system, which supports the creation of a multiplicity of static-analysis, dynamic-analysis, and symbolic-analysis components.

Using TSL, we developed several applications for analyzing machine-code, including a revised version of CodeSurfer/x86 in which all analysis components are generated from a TSL specification of the IA32 instruction set. The analogous components for a CodeSurfer/ppc32 system were generated from a TSL specification of the PowerPC32 instruction set.

In addition, we were able to create mutually-consistent, correct-by-construction implementations of symbolic primitives—in particular, quantifier-free, first-order-logic formulas for (i) symbolic evaluation of a single command, (ii) pre-image with respect to a single command, and (iii) symbolic composition for a class of formulas that express state transformations. Using the symbolic-analysis primitives, together with other TSL-generated abstract-interpretation components, we developed two analysis tools—MCVETO and BCE—that use logic-based search procedures to establish properties of machine-code programs.

Our evaluation of TSL in §5 shows that TSL has a number of benefits and relatively few drawbacks.

Acknowledgments. We are grateful to our collaborators currently or formerly at Wisconsin—T. Andersen, G. Balakrishnan, E. Driscoll, M. Elder, N. Kidd, A. Lal, T. Sharma, and A. Thakur—and at GrammaTech, Inc.—T. Teitelbaum, S. Yong, D. Melski, T. Johnson, D. Gopan, A. Loginov, and B. Alliet—for their many contributions to the project. We thank A.C.J. Fox for creating a TSL specification for ARMv7-A from his HOL4 ARMv7-A specification. We thank the referees for their extensive comments on the submission, which have helped us to improve the final version of the paper.

REFERENCES

Alt, M. and Martin, F. 1995. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symp.*

Alur, R. and Madhusudan, P. 2006. Adding nesting structure to words. In *Developments in Lang. Theory.*

APRON. APRON numerical abstract domain library. apron.cri.ensmp.fr.

ARMv7-A 2013. ARMv7-A in HOL. github.com/mn200/HOL/tree/master/examples/ARM/v7.

Assmann, U. 2000. Graph rewrite systems for program optimization. *Trans. on Prog. Lang. and Syst. 22,* 4.

Balakrishnan, G. 2007. WYSINWYX: What You See Is Not What You eXecute. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1603.

Balakrishnan, G., Gruian, R., Reps, T., and Teitelbaum, T. 2005. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *Comp. Construct.*

Balakrishnan, G. and Reps, T. 2004. Analyzing memory accesses in x86 executables. In *Comp. Construct.* 5–23.

Balakrishnan, G. and Reps, T. 2007. DIVINE: DIscovering Variables IN Executables. In *Verif., Model Checking, and Abs. Interp.*

Balakrishnan, G. and Reps, T. 2010. WYSINWYX: What You See Is Not What You eXecute. *Trans. on Prog. Lang. and Syst. 32,* 6.

Beckman, N., Nori, A., Rajamani, S., and Simmons, R. 2008. Proofs from tests. In *Int. Symp. on Softw. Testing and Analysis.*

Bouajjani, A., Esparza, J., and Touili, T. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.* 62–73.

Brauer, J. and King, A. 2012. Transfer function synthesis without quantifier elimination. *Logical Methods in Comp. Sci. 8,* 3.

Brauer, J., King, A., and Kowalewski, S. 2012. Abstract interpretation of microcontroller code: Intervals meet congruences. Submitted for journal publication.

Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. 2011. BAP: A binary analysis platform. In *Computer Aided Verif.*

Burstall, R. 1969. Proving properties of programs by structural induction. *Comp. J. 12,* 1 (Feb.), 41–48.

Cok, D. 2010. Safety in numbers. www.dtic.mil/dtic/tr/fulltext/u2/a532995.pdf.

Cook, T. A., Franzon, P. D., Harcourt, E. A., and Miller, T. K. 1993. System-level specification of instruction sets. In *DAC.*

Cooper, K. and Kennedy, K. 1988. Interprocedural side-effect analysis in linear time. In *Prog. Lang. Design and Impl.* 57–66.

Cousot, P. 2003. Verification by abstract interpretation. In *Verification: Theory and Practice.*

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Princ. of Prog. Lang.* 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Princ. of Prog. Lang.* 269–282.

Cousot, P. and Cousot, R. 2000. Temporal abstract interpretation. In *Princ. of Prog. Lang.* 12–25.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear constraints among variables of a program. In *Princ. of Prog. Lang.* 84–96.

Davidson, J. W. and Fraser, C. W. 1984. Code selection through object code optimization. In *TPLS.*

de Moura, L. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Int. Conf. on Tools and Algs. for the Construction and Analysis of Systems.*

Driscoll, E., Thakur, A., and Reps, T. 2012. OpenNWA: A nested-word-automaton library (tool paper). In *Computer Aided Verif.*

Dullien, T. and Porst, S. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest.*

Dutertre, B. and de Moura, L. 2006. Yices: An SMT solver. yices.csl.sri.com.

Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T. 2011. Abstract domains of affine relations. In *Static Analysis Symp.*

Elder, M., Lim, J., Sharma, T., Andersen, T., and Reps, T. 2013. Abstract domains of affine relations. TR-1777, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. In preparation.

Ferrante, J., Ottenstein, K., and Warren, J. 1987. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst. 3,* 9, 319–349.

Fox, A. 2003. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics.* 25–40.

Fox, A. 2012. Directions in ISA specification. In *Int. Conf. on Interactive Theorem Proving*. 243–258.

Fox, A. and Myreen, M. 2010. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Int. Conf. on Interactive Theorem Proving*. 243–258.

Godefroid, P., Klarlund, N., and Sen, K. 2005. DART: Directed automated random testing. In *Prog. Lang. Design and Impl.*

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verif.* 72–83.

Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., and Rajamani, S. 2006. SYNERGY: A new algorithm for property checking. In *Found. of Softw. Eng.*

Harcourt, E., Mauney, J., and Cook, T. 1994. Functional specification and simulation of instruction set architectures. In *PLC*.

IA32. *IA-32 Intel Architecture Software Developer's Manual*. developer.intel.com/design/pentiumii/manuals/243191.htm.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*. Intel. download.intel.com/design/processor/manuals/253666.pdf.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*. Intel. download.intel.com/design/processor/manuals/253667.pdf.

Johnson, S. 1975. YACC: Yet another compiler-compiler. Tech. Rep. Comp. Sci. Tech. Rep. 32, Bell Laboratories.

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

Jones, N. and Mycroft, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Princ. of Prog. Lang.* 296–306.

Jones, N. and Nielson, F. 1995. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Vol. 4. Oxford Univ. Press, 527–636.

Kästner, D. 2003. TDL: a hardware description language for retargetable postpass optimizations and analyses. In *GPCE*.

King, A. and Søndergaard, H. 2010. Automatic abstraction for congruences. In *Verif., Model Checking, and Abs. Interp.*

Kodumal, J. and Aiken, A. 2005. Banshee: A scalable constraint-based analysis toolkit. In *Static Analysis Symp.*

Lacey, D., Jones, N., Van Wyk, E., and Frederiksen, C. 2004. Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation 17,* 3.

Lal, A. and Reps, T. 2006. Improving pushdown system model checking. In *Computer Aided Verif.*

Lal, A., Reps, T., and Balakrishnan, G. 2005. Extended weighted pushdown systems. In *Computer Aided Verif.*

Lam, P., Bodden, E., Lhotak, O., and Hendren, L. 2011. The Soot framework for Java program analysis: A retrospective. In *Cetus Users and Compiler Infrastructure Workshop*.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*.

Lev-Ami, T. and Sagiv, M. 2000. TVLA: A system for implementing static analyses. In *Static Analysis Symp.* 280–301.

Lim, J. 2011. Transformer Specification Language: A system for generating analyzers and its applications. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1689.

Lim, J., Lal, A., and Reps, T. 2009. Symbolic analysis via semantic reinterpretation. In *Spin Workshop*.

Lim, J., Lal, A., and Reps, T. 2011. Symbolic analysis via semantic reinterpretation. *Softw. Tools for Tech. Transfer 13,* 1, 61–87.

Lim, J. and Reps, T. 2008. A system for generating static analyzers for machine instructions. In *Comp. Construct.*

Lim, J. and Reps, T. 2010. BCE: Extracting botnet commands from bot executables. Tech. Rep. TR-1668, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI.

Linn, C. and Debray, S. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*.

Magnusson, P. 2011. Understanding stacks and registers in the Sparc architecture(s). "web.archive.org/web/20110610161119/http://www.sics.se/ psm/sparcstack.html", snapshot of June 10, 2011.

Malmkjær, K. 1993. Abstract interpretation of partial-evaluation algorithms. Ph.D. thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas.

Martignoni, L., McCamant, S., Poosankam, P., Song, D., and Maniatis, P. 2012. Path-exploration lifiting: Hi-fi tests for lo-fi emulators. In *Architectural Support for Prog. Lang. and Op. Syst.*

Mauborgne, L. and Rival, X. 2005. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*.

Miné, A. 2002. A few graph-based relational numerical abstract domains. In *Static Analysis Symp.* 117–132.

Mishra, P., Shrivastava, A., and Dutt, N. 2006. Architecture description language: driven software toolkit generation for architectural exploration of programmable SOCs. *TODAES*.

Müller-Olm, M. and Seidl, H. 2004. Precise interprocedural analysis through linear algebra. In *Princ. of Prog. Lang.*

Müller-Olm, M. and Seidl, H. 2005. Analysis of modular arithmetic. In *European Symp. on Programming*.

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

Myreen, M., Fox, A., and Gordon, M. 2007. Hoare logic for ARM machine code. In *Fundamentals of Softw. Eng.* 272–286.

Nielson, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comp. Sci. 69*, 117–242.

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

Pees, S., Hoffmann, A., Zivojnovic, V., and Meyr, H. 1999. LISA machine description language for cycle-accurate models of programmable DSP architectures. In *DAC*.

Pettersson, M. 1992. A term pattern-match compiler inspired by finite automata theory. In *CC*.

Pleban, U. and Lee, P. 1987. High-level semantics. In *Workshop on Mathematical Foundations of Programming Language Semantics*.

PowerPC32. The PowerPC User Instruction Set Architecture. doi.ieeecs.org/10.1109/MM.1994.363069.

Ramalingam, G., Field, J., and Tip, F. 1999. Aggregate structure identification and its application to program analysis. In *Princ. of Prog. Lang.*

Ramsey, N. and Davidson, J. 1999. Specifying instructions' semantics using λ-RTL. Unpublished manuscript.

Ramsey, N. and Fernández, M. 1997. Specifying representations of machine instructions. *Trans. on Prog. Lang. and Syst. 19,* 3, 492–524.

Regehr, J. and Reid, A. 2004. HOIST: A system for automatically deriving static analyzers for embedded systems. In *Architectural Support for Prog. Lang. and Op. Syst.*

Reps, T., Balakrishnan, G., and Lim, J. 2006. Intermediate-representation recovery from low-level code. In *Part. Eval. and Semantics-Based Prog. Manip.*

Reps, T., Lim, J., Thakur, A., Balakrishnan, G., and Lal, A. 2010. There's plenty of room at the bottom: Analyzing and verifying machine code. In *Computer Aided Verif.*

Reps, T., Sagiv, M., and Loginov, A. 2010. Finite differencing of logical formulas for static analysis. *Trans. on Prog. Lang. and Syst. 6,* 32.

Reps, T., Sagiv, M., and Yorsh, G. 2004. Symbolic implementation of the best transformer. In *Verif., Model Checking, and Abs. Interp.* 252–266.

Reps, T., Schwoon, S., Jha, S., and Melski, D. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog. 58,* 1–2 (Oct.), 206–263.

Scherpelz, E., Lerner, S., and Chambers, C. 2007. Automatic inference of optimizer flow functions from semantics meanings. In *Prog. Lang. Design and Impl.*

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

Schmidt, D. 1998. Data-flow analysis is model checking of abstract interpretations. In *Princ. of Prog. Lang.* 38–48.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall.

Siewiorek, D., Bell, G., and Newell, A. 1982. *Computer Structures: Principles and Examples.* Springer-Verlag.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. 2008. BitBlaze: A new approach to computer security via binary analysis. In *ICISS.*

SOOT. SOOT: A Java optimization framework. www.sable.mcgill.ca/soot/.

Steffen, B. 1991. Data flow analysis as model checking. In *Theor. Aspects of Comp. Softw.* Lec. Notes in Comp. Sci., vol. 526. Springer-Verlag, 346–365.

Steffen, B. 1993. Generating data flow analysis algorithms from modal specifications. *Sci. of Comp. Prog. 21,* 2, 115–139.

Thakur, A., Elder, M., and Reps, T. 2012. Bilateral algorithms for symbolic abstraction. In *Static Analysis Symp.*

Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., and Reps, T. 2010. Directed proof generation for machine code. In *Computer Aided Verif.*

Thakur, A. and Reps, T. 2012. A method for symbolic computation of abstract operations. In *Computer Aided Verif.*

Tjiang, S. and Hennessy, J. 1992. Sharlit: A tool for building optimizers. In *Prog. Lang. Design and Impl.*

Venkatesh, G. 1989. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *Prog. Lang. Design and Impl.*

Venkatesh, G. and Fischer, C. 1992. SPARE: A development environment for program analysis algorithms. *Trans. on Softw. Eng. 18,* 4.

Wadler, P. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, NJ, Chapter 5: Efficient Compilation of Pattern-Matching, 78–103.

WALA. WALA. wala.sourceforge.net/wiki/index.php/.

WALi 2007. WALi: The Weighted Automaton Library. www.cs.wisc.edu/wpis/wpds/download.php.

Whaley, J., Avots, D., Carbin, M., and Lam, M. 2005. Using Datalog with Binary Decision Diagrams for program analysis. In *Asian Symp. on Prog. Lang. and Systems.*

Wilhelm, R. 1981. Global flow analysis and optimization in MUG2 the compiler generating system. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall.

Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J.-A., Tjiang, S., Liao, S.-W., Tseng, C.-W., Hall, M., Lam, M., and Hennessy, J. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices 29,* 12.

WPDS++ 2004. WPDS++: A C++ library for Weighted Pushdown Systems. www.cs.wisc.edu/wpis/wpds/download.php.

Yi, K. and Harrison, III, W. 1993. Automatic generation and management of interprocedural program analyses. In *Princ. of Prog. Lang.*