A System for Generating Static Analyzers for Machine Instructions*

Junghee Lim[1] and Thomas Reps[1,2]

[1] Comp. Sci. Dep., Univ. of Wisconsin-Madison, WI; USA.
[2] GrammaTech, Inc.; Ithaca, NY; USA
{junghee,reps}@cs.wisc.edu

Abstract. There is growing interest in analyzing executables to look for bugs and security vulnerabilities. This paper describes the design and implementation of a language for describing the semantics of an instruction set, along with a runtime system to support the static analysis of executables written in that instruction set. The work advances the state of the art by creating multiple analysis phases from a specification of the concrete operational semantics of the language to be analyzed. By exploiting this powerful infrastructure for creating analysis components, it will be possibly for recently developed analysis techniques for analyzing executables to be applied more broadly, to executables written in a variety of instructions sets.

1 Introduction

The problem of analyzing executables to recover information about their execution properties has been receiving increased attention. However, much of this work has focused on *specialized* analyses to identify aliasing relationships [19], data dependences [7, 13], targets of indirect calls [18], values of strings [12], bounds on stack height [34], and values of parameters and return values [40]. In contrast, Balakrishnan and Reps [8, 10] developed ways to address all of these problems by means of an analysis that discovers an overapproximation of the set of states that can be reached at each point in the executable—where a *state* means *all* of the state: values of registers, flags, and the contents of memory. Moreover, their approach is able to be applied to stripped executables (i.e., neither source code nor symbol-table/debugging information is available).

Although their techniques, in principle, are language-independent, they were instantiated only for the Intel IA32 instruction set. Our motivation is to provide a systematic way of extending those analyses—and others—to instruction sets other than IA32.

The situation that we face is actually typical of much work on program analysis: although the techniques described in the literature are, in principle, language-independent, implementations are often tied to a specific language or intermediate representation (IR). This state of affairs reduces the impact that good ideas developed in one context (e.g., Java program analysis) have in other contexts (e.g., C++ analysis).

For high-level languages, the situation has been addressed by developing common intermediate languages, e.g., GCC's RTL, Microsoft's MSIL, etc. (although the academic research community has not rallied around a similar common platform). The situation is more serious for low-level instruction sets, because of (i) instruction-set evolution over time (and the desire to have backward compatibility as word size increased from 8 bits to 64 bits), which has led to instruction sets with several hundred instructions, and (ii) a variety of architecture-specific features that are incompatible with other architectures.

* Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

To address these issues, we developed a language for describing the semantics of an instruction set, along with a run-time system to support the static analysis of executables written in that instruction set. The work reported in this paper advances the state of the art by creating a system for automatically generating analysis components from a specification of the language to be analyzed. The system that we have created, called TSL (for "**T**ransformer **S**pecification **L**anguage"), has two classes of users: (1) instruction-set-specification (ISS) developers and (2) analysis developers. The former are involved in specifying the semantics of different instruction sets; the latter are involved in extending the analysis framework.

In the design of the TSL system, we were guided by the following principles:

- There should be a formal language for specifying the semantics of the language to be analyzed. Moreover, ISS developers should specify only the abstract syntax and a concrete operational semantics of the language to be analyzed—each analyzer should be automatically generated from this specification.
- 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 analysis 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).
- 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—this is the key concept that makes the analyzer-generator system language-independent.

Each of these objectives has been achieved in the TSL system. The contributions made by our work can be summarized as follows:

Transformer Specification Language. We created the TSL language for specifying the abstract syntax and concrete semantics of instruction sets, and developed mechanisms by which a multiplicity of instruction-set analyzers can be generated automatically. 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).

Support for Multiple Analysis Types. The system supports several analysis types.

- Classical worklist-based value-propagation analyses 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 [37].

[3] The translation of the concrete syntaxes to and from abstract syntax is handled by a generator tool that is separate from TSL, and will not be discussed in this paper. The relationship between the two systems is similar to that between Flex and Bison. With Flex and Bison, a Flex-generated lexer passes tokens to a Bison-generated parser. In our case, the TSL-defined abstract syntax serves as the formalism for communicating values—namely, instructions' abstract syntax trees—between the two tools.

- Transformer-composition analyses [16, 37], which are particularly useful for context-sensitive interprocedural analysis.
- Unification-based analyses for flow-insensitive interprocedural analysis.

In addition, an emulator (for the concrete semantics) is also supported.

Implemented Analyses. These mechanisms have been instantiated for a number of specific analyses that are useful for analyzing low-level code, including: value-set analysis [8, 10] (§4.1), affine-relation analysis [8, §7.2] (§4.2), aggregate structure identification [11] (§4.3), def-use analysis (for memory, registers, and flags) (§4.4), and generation of symbolic expressions for an instruction's semantics (§4.5).

Established Applicability. The capabilities of our approach have been demonstrated by writing specifications for IA32 and PowerPC. These are nearly complete specifications of the languages—not idealized subsets, as are often used in academic studies—and include such features as (1) aliasing among 8-bit, 16-bit, 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 and borrow, shifting, rotation, etc.), and (4) setting of condition codes (and their subsequent interpretation at jump instructions).

The abstract transformers for these analyses that are created from the IA32 and PowerPC32 TSL specifications have been put together to create a system that essentially duplicates CodeSurfer/x86 [9]. A similar analysis system for PowerPC is under construction. (The TSL-generated components are in place; only a few mundane infrastructure components are lacking.)

We have also experimented with sufficiently complex features of other low-level 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.

There are many specification languages for instruction sets and many purposes to which they have been applied. In our work, we needed a mechanism to create abstract interpreters of instruction-set specifications. There are (at least) four issues that arise: during the abstract interpretation of each transformer, the abstract interpreter must be able to (i) execute over abstract states, (ii) execute both branches of 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. Such a mechanism did not appear to be available in the languages that we looked at. As far as we know, TSL is the first instruction-set-specification language to support such mechanisms.

Although this paper only discusses the application of TSL to low-level instruction sets, we believe that only small extensions would be needed to be able to apply TSL to source-code languages (i.e., to create language-independent analyzers for source-level IRs). The main obstacle is that the concrete semantics of a source-code language generally uses an execution state based on nested variable-to-value (or variable-to-location, location-to-value) maps. For a low-level language, the state incorporates an address-based memory model, for which the TSL language provides appropriate primitives.

The remainder of the paper is organized as follows: §2 introduces TSL and the capabilities of the system. §3 presents how the TSL system handles some important issues, such as recursion and conditional branches in CIR. §4 explains how CIR is instantiated to create an analyzer for a specific analysis component. §5 describes quirky features

of several instruction sets, and discusses how those features are handled in TSL. §6 discusses related work.

2 Overview of the TSL System

This section provides an overview of the TSL system. We discuss how three analysis components are created automatically from a TSL specification, using a fragment of the IA32 instruction set to illustrate the process.

2.1 TSL from an ISS Developer's Standpoint

Fig. 1 shows part of a specification of the IA32 instruction set taken from the manual [1]. The specification contains information about the registers as well as the addressing modes that are supported. It also provides the specification of the ADD instruction's action, i.e., how it manipulates its operands and how it changes the state.

Fig. 1. A part of the Intel manual's specification of IA32's ADD instruction.

General Purpose registers: EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI,EIP
Each of these registers also have 16-bit 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.

```
[1]  // User-defined abstract syntax
[2]  reg32: EAX() | EBX() | . . . ;
[3]  flag: ZF() | SF() | . . . ;
[4]  operand32: Indirect32(reg32 reg32 INT8 INT32)
[5]    | DirectReg32(reg32)| Immediate32(INT32)| ...;
[6]  operand16: . . . ;
[7]  . . .
[8]  instruction
[9]    : ADD32_32(operand32 operand32)
[10]   | ADD16_16(operand16 operand16) | . . . ;
[11] var32: Reg32(reg32);
[12] var_bool: Flag(flag);
[13] state: State(MEMMAP32_8_LE  // memory-map
[14]              VAR32MAP       // register-map
[15]              VARBOOLMAP);   // flag-map
[16] // User-defined functions
[17] INT32 interpOp(state S, operand32 I) { . . . }
[18] state updateFlag(state S, . . . ) { . . . }
[19] state updateState(state S, . . . ) {
[20]   with(S) (
[21]     State(mem,regs,flags): . . .
[22] }
[23] state interpInstr(instruction I, state S) {
[24]   with(I) (
[25]     ADD32_32(dstOp, srcOp):
[26]       let dstVal = interpOp(S, dstOp);
[27]           srcVal = interpOp(S, srcOp);
[28]           res = dstVal + srcVal;
[29]           S2 = updateFlag(S, dstVal, srcVal, res);
[30]       in ( updateState( S2, dstOp, res ) ),
[31]     . . .
[32]   )
[33] }
```

Fig. 2. 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.

```
[1]  template <typename INTERP>
[2]  class CIR {
[3]    class reg32 { . . . };
[4]    class EAX: public reg32 { . . . };
[5]    . . .
[6]    class operand32 { . . . };
[7]    class Indirect32: public operand32 { . . . };
[8]    . . .
[9]    class instruction { . . . };
[10]   class ADD32_32: public instruction { . . .
[11]     enum  TSL_ID  id;
[12]     operand32  op1;
[13]     operand32  op2;
[14]   };
[15]   . . .
[16]   class state { . . . };
[17]   class State: public state { . . .
[18]     INTERP::MEMMAP32_8_LE  mapMap;
[19]     INTERP::VAR32MAP       var32Map;
[20]     INTERP::VARBOOLMAP     varBoolMap;
[21]   };
[22]   . . .
[23]   static state interpInstr(instruction I, state S) {
[24]     state ans;
[25]     switch(I.id) {
[26]       case ID_ADD32_32: {
[27]         operand32 dstOp = I.op1;
[28]         operand32 srcOp = I.op2;
[29]         INTERP::INT32 dstVal = interpOp(S, dstOp);
[30]         INTERP::INT32 srcVal = interpOp(S, srcOp);
[31]         INTERP::INT32 res = INTERP::Add(dstVal,srcVal);
[32]         state S2 = updateFlag(S, dstVal, srcVal, res);
[33]         ans = updateState(S2, dstOp, res);
[34]       } break;
[35]       . . .
[36]     }
[37]   }
[38]   return ans;
[39]};
```

Fig. 3. A part of the CIR generated from Fig. 2.

However, the specification from Fig. 1 is only semi-formal: it uses a mixture of English and pseudo-code. Our work is based on completely formal specifications, which are written in a language that we designed (TSL). TSL is a first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Fig. 2 shows the part of the TSL specification that corresponds to Fig. 1.

Much of what an ISS developer writes is similar to writing an interpreter for an instruction set in first-order ML [20]. An ISS developer specifies the abstract syntax grammar by defining the constructors for a language of instructions (lines 2–10), a concrete-state type (lines 13–15), and the concrete semantics of each instruction (lines 23–33).

TSL provides 5 basetypes: INT8, INT16, INT32, INT64, and BOOL. TSL supports arithmetic/logical operators ($+$, $-$, $*$, $/$, !, $\&\&$, $||$, xor), bit-manipulation operators ($\sim$, $\&$, $|$, $\hat{}$, $\ll$, $\gg$, right-rotate, left-rotate), relational operators ($<$, $<=$, $>$, $>=$, $==$, !=), and a conditional-expression operator ($?:$).

TSL also provides several map-basetypes: MEMMAP32_8_LE, MEMMAP32_16_LE, VAR32MAP, VAR16MAP, VAR8MAP, VARBOOLMAP, etc. MEMMAP32_8_LE maps from 32-bit values (addresses) to 8-bit values, VAR32MAP from var32 to 32-bit values, VARBOOLMAP from var_bool to Boolean values, and so forth. Tab. 1 shows the list of some of the TSL *access/update* functions. Each *access* function takes a map (e.g., MEMMAP32_8_LE, VAR32MAP, VARBOOLMAP, etc.) and an appropriate key (e.g., INT32, var32, var_bool, etc.), and returns the value that corresponds to the key. Each *update* function takes a map, a key, and a value, and returns the updated map. The *access/update* functions for MEMMAP32_8_LE implement the little-endian storage convention.

Table 1. *Access/Update* functions.

MEMMAP32_8_LE MemUpdate_32_8_LE_32(MEMMAP32_8_LE *memmap*, INT32 *key*, INT32 *v*); INT32 MemAccess_32_8_LE_32(VAR32MAP *mapmap*, INT32 *key*);
VAR32MAP Var32Update(VAR32MAP *var32Map*, var32 *key*, INT32 *v*); INT32 Var32Access(VAR32MAP *var32Map*, var32 *key*);
VARBOOLMAP VarBoolUpdate(VARBOOLMAP *varBoolMap*, var_bool *key*, BOOL *v*); BOOL VarBoolAccess(VARBOOLMAP *varBoolMap*, var_bool *key*);

Each specification must define several reserved (but user-defined) types: var64, var32, var16, var8, and var_bool, which represent storage components of 64-bit, 32-bit, 16-bit, 8-bit, and Boolean types, respectively; instruction; state; as well as the reserved function interpInstr. (These are underlined in Fig. 2.) These form part of the API available to *analysis engines* that use the TSL-generated transformers (see §4). The reserved types are used as an interface between the CIR and analysis domain implementations.

The definition of types and constructors on lines 2–10 of Fig. 2 is an abstract-syntax grammar for IA32. The definitions for var32 and var_bool wrap the user-types reg32 and flag, respectively. Type reg32 consists of nullary constructors for IA32 registers, such as EAX() and EBX(); flag consists of nullary constructors for the IA32 condition codes, such as ZF() and SF(). Lines 4–7 define types and constructors to represent the various kinds of operands that IA32 supports, i.e., various sizes of direct register, immediate, and indirect memory operands. The reserved (but user-defined) type instruction consists of user-defined constructors for each instruction, such as ADD32_32 and ADD16_16, which represent instructions with different operand sizes.

The type state specifies the structure of the execution state. The state for IA32 is defined on lines 13–15 of Fig. 2 to consist of a memory-map, a register-map, and a flag-map. The *concrete semantics* is specified by writing a function named interpInstr (see line 23 of Fig. 2), which maps an instruction and a state to a state.

2.2 Common Intermediate Representation (CIR)

Fig. 3 shows part of the TSL CIR automatically generated from Fig. 2. 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 INTERP, an abstract domain for an analysis (lines 1–2). The user-defined abstract syntax (lines 2–10 of Fig. 2) is translated to a set of C++ abstract-domain classes (lines 3–15 of Fig. 3) that contain appropriate abstract operators. The user-defined types, such as reg32, operand32, and instruction, are translated to abstract C++ classes, and the constructors, such as EAX, Indirect32, and ADD32_32, are subclasses of the parent abstract C++ class. Each user-defined function is translated to a CIR member function.

Each TSL basetype and basetype-operator is prepended with the template parameter name INTERP; INTERP is supplied for each analysis by an analysis designer. The with expression and the pattern matching on lines 24–25 of Fig. 2 are translated to switch statements in C++[4] (lines 25–36 in Fig. 3). The function calls for obtaining the values of the two operands (lines 26–27 in Fig. 2) correspond to the C++ code on lines 29–30 in Fig. 3. The TSL basetype-operator $+$ on line 28 in Fig. 2 is translated to the CIR member function Add, as shown on line 31 in Fig. 3. The function calls for updating the state (lines 29–30 in Fig. 2) are translated into C++ code (lines 32–33 in Fig. 3).

2.3 TSL from an Analysis Developer's Standpoint

The generated CIR is instantiated for an analysis by defining (in C++) an *interpretation*: a representation class for each TSL basetype, and implementations of each TSL basetype-operator and built-in function. Tab. 2 shows the implementations of primitives for three selected analyses: value-set analysis (VSA, see §4.1), quantifier-free bit-vector semantics (QFBV, see §4.5), and def-use analysis (DUA, see §4.4).

Each interpretation defines an abstract domain. For example, line 3 of each column defines the abstract-domain class for INT32: ValueSet32, QFBVTerm32, and UseSet. Each abstract domain is also required to contain a set of reserved functions, such as *join*, *meet*, and *widen*, which forms an additional part of the API available to analysis engines that use TSL-generated transformers (see §4).

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.

2.4 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. 3 is instantiated with the

[4] The TSL front end performs *with-normalization*, which transforms all (multi-level) with expressions to use only one-level patterns, using the pattern-compilation algorithm from [31, 38].

Table 2. Parts of the declarations of the basetypes, basetype-operators, and map-access/update functions for three analyses.

VSA

```
[1]  class VSA_INTERP {
[2]    // basetype
[3]    typedef ValueSet32 INT32;
[4]    ...
[5]    // basetype-operators
[6]    INT32 Add(INT32 a, INT32 b) {
[7]        return a.addValueSet(b);
[8]    }
[9]    ...
[10]   // map-basetypes
[11]   typedef Dict<var32,INT32>
[12]             VAR32MAP;
[13]   ...
[14]   // map-access/update functions
[15]   INT32 Var32Access(
[16]     VAR32MAP m, var32 k) {
[17]       return m.Lookup(k);
[18]   }
[19]   VAR32MAP
[20]   Var32Update( VAR32MAP m,
[21]     var32 k, INT32 v) {
[22]       return m.Insert(k, v);
[23]   }
[24]   ...
[25]};
```

QFBV

```
[1]  class QFBV_INTERP {
[2]    // basetype
[3]    typedef QFBVTerm32 INT32;
[4]    ...
[5]    // basetype-operators
[6]    INT32 Add(INT32 a, INT32 b) {
[7]        return QFBVPlus32(a, b);
[8]    }
[9]    ...
[10]   // map-basetypes
[11]   typedef Dict<var32,INT32>
[12]             VAR32MAP;
[13]   ...
[14]   // map-access/update functions
[15]   INT32 Var32Access(
[16]     VAR32MAP m, var32 k) {
[17]       return m.Lookup(k);
[18]   }
[19]   VAR32MAP
[20]   Var32Update( VAR32MAP m,
[21]     var32 k, INT32 v) {
[22]       return m.Insert(k, v);
[23]   }
[24]   ...
[25]};
```

DUA

```
[1]  class DUA_INTERP {
[2]    // basetype
[3]    typedef UseSet INT32;
[4]    ...
[5]    // basetype-operators
[6]    INT32 Add(INT32 a, INT32 b) {
[7]        return a.Union(b);
[8]    }
[9]    ...
[10]   // map-basetypes
[11]   typedef KillUseSet VAR32MAP;
[12]   ...
[13]   // map-access/update functions
[14]   INT32 Var32Access(
[15]     VAR32MAP m, var32 k) {
[16]       return UseSet(k);
[17]   }
[18]   VAR32MAP
[19]   Var32Update( VAR32MAP m,
[20]     var32 k, INT32 v) {
[21]       VAR32MAP  a2 =
[22]             m.Insert2Kill(k);
[23]       return a2.Insert2Use(v);
[24]   }
[25]};
```

three interpretations from Tab. 2, lines 23–33 of Fig. 2 implement the three transformers presented (using mathematical notation) in Tab. 3.

Table 3. Transformers generated by the TSL system.

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

2.5 Measures of Success

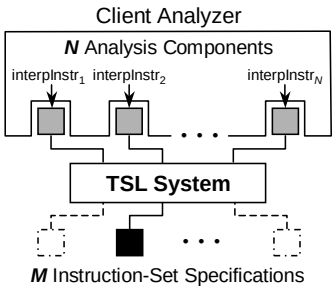


Fig. 4. The interaction between the TSL system and a client analyzer. The grey boxes represent TSL-generated analysis components.

The TSL system provides two dimensions of parameterizability: different instruction sets and different analyses. Each ISS developer specifies an instruction-set semantics, and each analysis 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 inputs from these two classes of users, the TSL system automatically generates an analysis component. 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. 4), 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. It took approximately 1 man-day to create each of the DUA and QFBV interpretations.

Another measure of success is demonstrated by our effort to use TSL to recreate the analysis components used in CodeSurfer/x86 [9]. We estimate that the task of writing transformers (for eight analysis phases used in CodeSurfer/x86) consumed about 20 man-months; in contrast, we have invested a total of about 1 man-month to write the C++ code for the set of 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 specifications for IA32 and PowerPC are, respectively, 2,834 and 1,370 (non-comment, non-blank) lines of TSL; the IA32 specification has gone through multiple revisions as the TSL system took shape; however, the PowerPC specification was written after the language stabilized, and took approximately 4 man-days.

Because each analysis is defined by providing an interpretation for the collection of *TSL* primitives, implementations of the abstract transformers for each analysis can be generated automatically for *every* instruction set for which one has a TSL specification. For instance, from the PowerPC specification, we were immediately able to generate all of the analyses that had been developed while working with the IA32 specification.

Ever since the days of the first compilers, systems that take over programming tasks previously performed manually have faced the question of how well their output performs compared to that created by human programmers. Due to the nature of the transformers used in one of the analyses that we implemented (affine-relation analysis (ARA) [28]), it was possible to write an algorithm to compare the TSL-generated ARA transformers and the hand-coded ARA transformers that were incorporated in CodeSurfer/x86. On a corpus of 542 instruction instances that covered various opcodes, addressing modes, and operand sizes, we found that the TSL-generated transformers were equivalent in 324 cases and *more precise* than the hand-coded transformers in the remaining 218 cases.[5]

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 concrete semantics, this guarantees that a *consistent* view of the concrete semantics is adopted by all of the analyses used in the system.

It takes approximately 8 seconds (on an Intel Pentium 4 with a 3.00GHz CPU and 2GB of memory, running Centos 4) for the TSL compiler to compile the IA32 specification to C++, followed by approximately 20 minutes wall-clock time (on an Intel Pentium 4 with a 1.73GHz CPU and 1.5GB of memory, running Windows XP) to compile the generated C++.

[5] Approximately 130 of the 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 always assigning an unknown value to the registers that they affected, regardless of the values of the arguments. Because the TSL-generated transformers are based on the ARA 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.

3 Generation of the Common Intermediate Representation

Given a TSL specification of an instruction set, the TSL system generates 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.

However, there are some important issues that need to be properly handled for the resulting code to be able to used to create abstract interpreters for an instruction-set specification. In particular, the code generated for each transformer must be able to: (i) execute over abstract states (§3.1), (ii) possibly propagate abstract states to more than one successor in a conditional expression (§3.2), (iii) compare abstract states and terminate abstract execution when a fixed point is reached (§3.3), and (iv) apply widening operators, if necessary, to ensure termination (§3.3). In §3.4, we discuss an additional issue that arises in CIR generation, which is important for avoiding loss of precision for some generated analyzers.

3.1 Execution Over Abstract States

As discussed in §2.2, the TSL system generates the CIR as a template class that takes as input an interpretation INTERP. For each analysis, the CIR is instantiated by an appropriate interpretation for INTERP that the analysis developer defines, as described in §2.3. (§3.4 discusses more about how the TSL system generates the CIR.)

3.2 Conditional Branch

```
[1]  INTERP::BOOL t0 = . . . ; // translation of a
[2]  INTERP::INT32 t1;
[3]  INTERP::INT32 t2;
[4]  INTERP::INT32 answer;
[5]  if(Bool3::possibly_false(t0.getBool3Value())) {
[6]     . . .
[7]     t1 = . . . ; // translation of b
[8]     answer = t1;
[9]  }
[10] if(Bool3::possibly_true(t0.getBool3Value())) {
[11]    . . .
[12]    t2 = . . . ; // translation of c
[13]    answer = t2;
[14] }
[15] if(t0.getBool3Value() == Bool3::MAYBE) {
[16]    answer = t1.join(t2);
[17] }
```

Fig. 5. The translation of the conditional branch "let answer = a ? b : c;".

Fig. 5 shows part of the CIR that corresponds to the TSL expression "let answer = a ? b : c;". Bool3 is an abstract domain of Booleans (which consists of three values {FALSE, MAYBE, TRUE}, where MAYBE means "may be FALSE or may be TRUE"). The TSL conditional expression is translated into three if-statements (lines 5–9, lines 10–14, and lines 15–17 in Fig. 5). The body of the first if-statement is executed when the Bool3 value for a is possibly false (i.e., either FALSE or MAYBE). Likewise, the body of the second if-statement is executed when the Bool3 value for a is possibly true (i.e., either TRUE or MAYBE). The body of the third if-statement is executed when the Bool3 value for a is MAYBE. Note that in the body of the third if-statement, answer is overwritten with the *join* of t1 and t2 (line 16).

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 in the abstract state so that either an appropriate branch or both branches can be abstractly executed.

3.3 Comparison, Termination, and Widening

```
[1]  state repMovsd(state S, INT32 count) {
[2]    count == 0 ? S
[3]     : with(S) (
[4]         State(memory, regs, flags):
[5]           let direction = VarBoolAccess(flags, DF());
[6]               edi = RegValue32(regs,EDI());
[7]               esi = RegValue32(regs,ESI());
[8]               src = MemAccess_32_8_LE_32(memory, esi);
[9]               newRegs = direction
[10]                 ? RegUpdate32(RegUpdate32(
[11]                       regs,EDI(),edi-4), ESI(),esi-4)
[12]                 : RegUpdate32(RegUpdate32(
[13]                       regs,EDI(),edi+4), ESI(),esi+4);
[14]              newS = State(MemUpdate_32_8_LE_32(
[15]                         memory, edi, src), newRegs, flags);
[16]          in ( repMovsd(newS, count - 1) )
[17]     )
[18]}
```

Fig. 6. A recursive TSL function.

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

Fig. 7. The translation of the recursive function from Fig. 6. For simplicity, some mathematical notation is used, including ⊔ (join), ▽ (widening), ⊑ (approximation), and ⊥ (bottom).

Recursion is not often used in TSL specifications, but is needed 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 (LMT, STMT, 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.[6] For each recursive function specified by an ISS developer, 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 analysis developer has specified for the abstract domain in use.

For example, Fig. 6 shows the user-defined TSL function that handles "rep movsd", which copies the contents of one area of memory to a second area.[7] The amount of memory to be copied is passed into the function as the argument count. Fig. 7 shows its translation into the CIR. A recursive function like repMovsd (Fig. 6) is split into two functions, repMovsd (line 4 of Fig. 7) and repMovsdAux (line 11 of Fig. 7). The TSL system initializes appropriate global variables global_S and

[6] Currently, TSL supports only tail-recursion.

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

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 fixpoint (lines 13–17). If so, repMovsdAux returns global_retval (line 18). If not, the body of repMovsdAux is analyzed again (lines 23–26). Note that at the translation of each normal return from repMovsdAux (e.g., line 27), the return value is joined into global_retval. The TSL system requires each analysis developer to define the functions *join* and *widen* for the basetypes of the interpretation used in the analysis.

3.4 Two-Level CIR

The examples given in Figs. 3, 5, and 7, 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 [29, 22]: CONC_INTERP for concrete interpretation (i.e., interpretation in the concrete semantics), and ABS_INTERP for abstract interpretation. Either CONC_INTERP or ABS_INTERP would replace the occurrences of INTERP in the example CIR shown in Figs. 3, 5, and 7.

```
[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. 8. A fragment of the PowerPC specification for interpreting BCx instructions (BC, BCA, BCL, BCLA).

```
[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. 9. An example of factoring in TSL.

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 ISS developer could follow the approach taken in the PowerPC manual [2] 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. 8).

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. 9 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—and is never made worse—by interpreting subexpressions associated with the manipulation of concrete values in concrete semantics. 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 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 $\{1, 2\}$.).

To address this issue, we perform binding-time analysis [21] 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 CONC_INTERP for C values, and ABS_INTERP for A values. The generated CIR is instantiated for an analysis transformer by defining ABS_INTERP. The TSL translator supplies a predefined concrete interpretation for CONC_INTERP.

4 Generation of Static Analyzers

In this section, we explain how various analyses are created using our system, and illustrate this process with some specific analysis examples.

As illustrated in Fig. 4, a version of the interface function interpInstr is created for each analysis. Each analysis engine calls interpInstr at appropriate moments to obtain a transformer for an instruction being processed. 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 [37]. VSA uses this kind of analysis engine (§4.1).
- *Transformer Composition [TC]*. These generally perform flow-sensitive, context-sensitive interprocedural analysis. DUA (§4.4) uses this kind of analysis engine.
- *Unification-Based Analyses [UB]*. These perform flow-insensitive interprocedural analysis.

For each analysis, the CIR is instantiated with an interpretation by an analysis developer. This mechanism provides wide flexibility in how one can couple the system 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 DUA, is used when interfacing to an analysis component that has its own input language for specifying abstract transformers. In this case, the instantiated CIR serves as a *transformer generator*: interpInstr is prepared to receive an instruction and a default abstract state[8] and return a transformer specification in the analysis component's input language.

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

[8] In the case of transformer generation for a TC analyzer, the default state is the identity function.

4.1 Creation of a TA Transformer Evaluator for VSA

VSA is a combined numeric-analysis and pointer-analysis algorithm that determines a safe approximation of the set of numeric values and addresses that each register and memory location holds at each program point [10]. 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 represents a set of numbers with a lower bound, an upper bound, and a stride [35].

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the integer basetypes is a *value-set*. The abstract domain for BOOL is Bool3 ($\{$FALSE, MAYBE, TRUE$\}$), where MAYBE means "may be FALSE or may be TRUE". The operators on these domains are described in detail in [35].

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain for memory maps (MEMMAP32_8_LE, MEMMAP32_16_LE, etc.) is a dictionary that maps each memory-location (INT32) to a *value-set*. The abstract domain for register maps (VAR32MAP, VAR16MAP, etc.) is a dictionary that maps each variable (var32, var16, etc.) to a *value-set*. The abstract domain for flag maps (VARBOOLMAP) is a dictionary that maps a var_bool 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. 3 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.

4.2 Creation of a TC Transformer Generator for ARA

An affine relation is a linear-equality constraint between integer-valued variables. ARA finds all affine relationships 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 [8, §7.2].

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 values represent transformers, each call to interpInstr will residuate a transformer for the instruction. In the case of ARA, the CIR is instantiated so that for each instruction, the generated transformer operates on an abstract domain whose values are sets of matrices that represent affine transformations on registers and memory locations of the state [28].

Interpretation of Basetypes and Basetype-Operators. The abstract domain for the integer basetypes 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 program 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 Times(eax, ebx), returns TOP, which means that no affine relationship is known to hold.

Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain of the maps for ARA is a set of matrices of size $(N+1) \times (N+1)$, where N is the number of program variables. This abstraction, which is able to find all affine relationships in an affine program, was defined by Müller-Olm and Seidl [28]. 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—e.g., *memmap* or *var32Map* in Tab. 1. 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 transformer relates linear-equality relationships that hold before the instruction to those that hold after execution of the instruction.

4.3 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 [11]. 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.

Abstract Domain for Basetypes and Basetype-Operators. The abstract domain for the basetypes is a *dataref*, which is either a memory access or a register access. The arithmetic, logical, and bit-vector operations on *dataref*s convert *dataref*s to *unassignable dataref*s, which means that they will only be used to generate *split*s.

Abstract Domain for Map-Basetypes and Access/Update Functions. The abstract domain of the maps for ASI is a set of *split*s and *unification*s. The *access* functions generate a *dataref* associated with memory location or register. The *update* functions create a set of *unification*s or *split*s according to the *dataref* 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 some procedure *foo*'s activation records, the ASI transformer generator emits the ASI *unification* command "*AR_foo*[-12:-9] :=: eax[0:3]".

4.4 Def-Use Analysis (DUA)

Def-Use analysis collects all the *definitions* and *uses* of state components (memory-locations, registers, and flags) for each instruction.

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the basetypes is a set of *uses*, and the operators on this domain perform a set-union of their arguments' sets.

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain of the maps for DUA is a tuple of sets—*defs* and *uses*. The *access/update* functions all perform set union of the sets associated with the arguments of MEMMAP32_8_LE, VAR32MAP, VARBOOLMAP, etc. (*memmap*, *var32Map*, or *varBoolMap* in Tab. 1).

The DUA results (e.g., row 3 of Tab. 3) are used to create transformers for several additional analyses, such as GMOD analysis [15], 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 (§4.5).

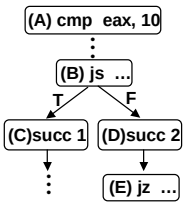
4.5 Quantifier-Free Bit-Vector (QFBV) Semantics

QFBV semantics provides a way to obtain a symbolic representation—as a formula in first-order quantifier-free bit-vector logic—of an instruction's semantics.

The Interpretation of Basetypes and Basetype-Operators. The abstract domain for the integer basetypes is a term, and each operator on it constructs a term that reflects the operation. The abstract domain for BOOL is a formula, and each operator on it constructs a formula that reflects the operation.

The Interpretation of Map-Basetypes and Access/Update Functions. The abstract domain for the state components is a dictionary that maps a storage component to a term (or a formula in the case of VARBOOLMAP). The *access/update* functions retrieve from and update the dictionaries, respectively.

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 [26]. The VSA analysis engine partitions the state at each flag-setting instruction based on live-flag information (which is obtained from an analysis that uses the DUA transformers); a semantic reduction [16] 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), which is a flag-setting instruction, has SF and ZF as live flags since 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[SF$\mapsto$T] [ZF $\mapsto$ T] [eax $\mapsto$ *reduce*(S(eax), $\varphi_1 \wedge \varphi_2$)]
S_2 := S[SF$\mapsto$T] [ZF $\mapsto$ F] [eax $\mapsto$ *reduce*(S(eax), $\varphi_1 \wedge \neg\varphi_2$)]
S_3 := S[SF$\mapsto$F] [ZF $\mapsto$ T] [eax $\mapsto$ *reduce*(S(eax), $\neg\varphi_1 \wedge \varphi_2$)]
S_4 := S[SF$\mapsto$F] [ZF $\mapsto$ F] [eax $\mapsto$ *reduce*(S(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).

4.6 Paired Semantics

Our system allows easy instantiations of *reduced product*s [16] by means of *paired semantics*. The TSL system provides a template for paired semantics as shown in Fig. 10.

```
[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_LE 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] };
```

Fig. 10. A part of the template class for paired semantics.

```
[1] typedef PairedSemantics<VSA_INTERP, DUA_INTERP> DUA;
[2] template<> DUA::INT32 DUA::MemAccess_32_8_LE_32( DUA::MEMMAP32_8_LE mem, DUA::INT32 addr) {
[3]     DUA::INTERP1::MEMMAP32_8_LE memory1 = mem.GetFirst();
[4]     DUA::INTERP2::MEMMAP32_8_LE memory2 = mem.GetSecond();
[5]     DUA::INTERP1::INT32 addr1 = addr.GetFirst();
[6]     DUA::INTERP2::INT32 addr2 = addr.GetSecond();
[7]     DUA::INT32 answer = interact(mem1, mem2, addr1, addr2);
[8]     return answer;
[9] }
```

Fig. 11. An example of C++ explicit template specialization to create a reduced product.

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. 11. The communication between interpretations may take place in basetype-operators or *access/update* functions; Fig. 11 is an example of the latter. The two components of the paired-semantics values are deconstructed on lines 3–6 of Fig. 11, and the individual INTERP1 and INTERP2 components from *both* inputs can be used (as illustrated by the call to *interact* on line 7 of Fig. 11) 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. 11).

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.

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

Fig. 12. A fragment of updateState.

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 get abstract memory location(s) for addr is lost because the DUA basetype-operators ($+$ and $*$ on line 3 of Fig. 12) just return the union of the arguments' *use* sets. With the pairing of VSA_INTERP with DUA_INTERP (line 1 of Fig. 11), DUA can use the abstract address computed for addr2 (line 6 of Fig. 11) by VSA_INTERP, which uses VSA_INTERP::Add and

VSA_INTERP::Mult; 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 (for a TC-style or UB-style evaluator), 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).

5 Instruction Sets

In this section, we discuss the quirky characteristics of some instruction sets, and various ways these can be handled in TSL.

5.1 IA32

To provide compatibility with 16-bit and 8-bit versions of the instruction set, IA32 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, 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.

Another characteristic to note is that IA32 keeps condition codes in a special register, called EFLAGS.[9] One way to specify 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 have symbolic flags, as in our examples, and have every manipulation of EFLAGS affect the individual flags.

5.2 ARM

Almost all ARM instructions contain a condition field that allows an instruction to be executed conditionally, dependent 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 = VarBoolAccess(flagMap, EQ());
[3]       src = interpOperand(curState, srcOprnd);
[4]       a = Var32Update(regMap, destReg, src);
[5]       b = regMap;
[6]       answer = cond ? a : b;
[7]   in ( answer )
```

Fig. 13. 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 instruction does nothing. Fig. 13 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

[9] Many other instruction sets, such as SPARC, PowerPC, and ARM, also use a special register to store condition codes.

and *src*, even when *cond* is known to have a definite value (TRUE or FALSE) in VSA semantics. The paired-semantics mechanism presented in §4.6 can help with improving the precision of analyzers by avoiding joins. 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*).

5.3 SPARC

```
[1]  var32 : Reg(INT8) | CWP() | . . .;
[2]  reg32 : OutReg(INT8) | InReg(INT8) | . . .;
[3]  state: State( . . . , VAR32MAP, . . . );
[4]  INT32 RegAccess(VAR32MAP regmap, reg32 r) {
[5]   let cwp = Var32Access(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 ( Var32Access(regmap, key) )
[12]}
```

Fig. 14. A method to handle the SPARC register window in TSL.

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 becames full, registers are spilled to the stack to free up a sufficient number for the called procedure. Fig. 14 shows a way to accomodate 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 with the index of the syntactic register, the value of CWP (the current window pointer) from the current state, and the platform-specific value NWINDOWS.

6 Related Work

There are 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 [30, 27]. Others have been used to generate an emulator for compiler-optimization testing [17, 23]. TDL [23] is a hardware-description language that supports the retargeting of analyses and optimizations relevant to instruction scheduling, register assignment, and functional-unit binding. The New Jersey machine-code toolkit [33] addresses concrete syntactic issues (instruction decoding, instruction encoding, etc.). Harcourt et al. used ML to specify the semantics of instruction sets [20]. LISAS [14] is an instruction-set-description language that was developed based on their experience using ML. The latter two approaches particularly influenced the design of the TSL language.

TSL shares some of the same goals as λ-RTL [32] (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 con-

structors and recursion.[10] 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-set's semantics.

Some systems for representing and analyzing programs are (mainly) targeted for a single language. For instance, SOOT [4] is a powerful and flexible analysis/optimization framework that supports analysis and transformation of Java bytecode.

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 [39], Banshee [25], the PPL [3], and WPDS++ [24]. 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 language's IR, using the package's API (or input language).

WALA [6] supports a common intermediate form (Common Abstract Syntax Tree), from which multiple additional IRs (e.g., CFGs and SSA-form) can be generated, and multiple analyses can be performed that use these IRs. Thus, this is similar to the package approach, but supports a multiplicity of analyses.

In contrast to the package approach, TSL provides a domain-specific language for instruction-set specification. With this approach, the ISS developer concentrates on specifying the concrete operational semantics of his language, using TSL, and a multiplicity of analyzers are then created automatically. Analysis developers can incorporate different analysis packages into the TSL framework by implementing appropriate abstract operations that overapproximate the semantics of a fixed set of TSL operations (that have a well-defined semantics). (Any of the aforementioned packages could be used for creating TSL-based analyses; currently, WPDS++ is used for all of the TC-style analyzers that have been developed for use with TSL so far.)

There are two analysis systems, TVLA [5] and the optimizer flow-function inference system developed by Rice et al. [36], in which sound analysis transformers are generated automatically from the concrete operational semantics, plus a specification of the abstraction (either via the abstraction function (TVLA) or the concretization function (Rice et al.)). In our system, we rely on the analysis developer to supply sound abstract operations. While this 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 (low-level code).
- Some of the analyses that we use, such as ARA [28], appear to be beyond the power of the heuristics-based transformer-generation methods developed by Rice et al.

The development of methods for creating abstract transformers from a specification of the abstraction or concretization function (à la [5] and [36]) is left for future research.

References

1. IA-32 Intel Architecture Software Developer's Manual, http://developer.intel.com/design/pentiumii/manuals/243191.htm.

[10] As discussed in §3.3, recursion is not often used in specifications, but is needed for handling some loop-interation instructions, such as the IA32 string-manipulation instructions and the PowerPC multiple-word load/store instructions.

2. The PowerPC User Instruction Set Architecture, http://doi.ieeecs.org/10.1109/mm.1994.363069.
3. PPL: The Parma polyhedra library. “http://www.cs.unipr.it/ppl/”.
4. Soot: A Java optimization framework, http://www.sable.mcgill.ca/soot/.
5. TVLA system. “http://www.cs.tau.ac.il/∼tvla/”.
6. WALA, http://wala.sourceforge.net/wiki/index.php/.
7. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *IFPP*, 2000.
8. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Univ. of Wisc., 2007.
9. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.
10. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
11. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, 2007.
12. M. Christodorescu, W. Goh, and N. Kidd. String analysis for x86 binaries. In *PASTE*, 2005.
13. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM*, 1997.
14. T. A. Cook, P. D. Franzon, E. A. Harcourt, and T. K. Miller. System-level specification of instruction sets. In *DAC*, 1993.
15. K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, 1988.
16. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
17. J. W. Davidson and C. W. Fraser. Code selection through object code optimization. In *TPLS*, 1984.
18. B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *PDPTA*, 2000.
19. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.
20. E. Harcourt, J. Mauney, and T. Cook. Functional specification and simulation of instruction set architectures. In *PLC*, 1994.
21. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
22. N. Jones and F. Nielson. Abstract interpretation: A semantics-based tool for program analysis. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 527–636. Oxford Univ. Press, 1995.
23. D. Kästner. TDL: a hardware description language for retargetable postpass optimizations and analyses. In *GPCE*, 2003.
24. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for Weighted Pushdown Systems, http://www.cs.wisc.edu/wpis/wpds++, 2004.
25. J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, 2005.
26. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.
27. P. Mishra, A. Shrivastava, and N. Dutt. Architecture description language: driven software toolkit generation for architectural exploration of programmable SOCs. *TODAES*, 2006.
28. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
29. F. Nielson and H. Nielson. *Two-Level Functional Languages*. Cambridge Univ. Press, 1992.
30. S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA machine description language for cycle-accurate models of programmable DSP architectures. In *DAC*, 1999.

31. M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In *CC*, 1992.
32. N. Ramsey and J. Davidson. Specifying instructions' semantics using λ-RTL. Unpublished manuscript, 1999.
33. N. Ramsey and M. F. Ferandez. New Jersey Machine-Code toolkit arch. spec. Technical Report TR-470-94, 1994.
34. J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *TECS*, 2005.
35. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, 2006.
36. E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantics meanings. In *PLDI*, 2007.
37. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Englewood Cliffs, NJ, 1981.
38. P. Wadler. Efficient compilation of pattern-matching. *The Impl. of Func. Prog. Lang.*, 1987.
39. J. Whaley, D. Avots, M. Carbin, and M. Lam. Using Datalog with Binary Decision Diagrams for program analysis. In *APLAS*, 2005.
40. J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *COMPSAC*, 2007.