A System for Generating Static Analyzers for Machine Instructions*

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

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

Abstract. This paper describes the design and implementation of a language for specifying 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 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.

1 Introduction

The problem of analyzing executables to recover information about their execution properties, especially for finding bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs), has been receiving increased attention. However, much of this work has focused on *specialized* analyses to identify aliasing relationships [13], data dependences [2, 8], targets of indirect calls [12], values of strings [7], bounds on stack height [20], and values of parameters and return values [24]. In contrast, Balakrishnan and Reps [3, 5] 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 can 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 retargeting those analyses—and others yet to be created—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). For high-level languages, the situation has been addressed by developing common intermediate languages, e.g., GCC's RTL, Microsoft's MSIL, etc. The situation is more serious for low-level instruction sets,

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

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.

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. Our work 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, 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 designing TSL, 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 generated automatically 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.

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). 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 (§3).

Other notable aspects of our work include

1. Support for Multiple Analysis Types.

- Classical worklist-based value-propagation analyses.
- Transformer-composition analyses [11, 22], which are particularly useful for context-sensitive interprocedural analysis, and for relational analyses.

[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.

– Unification-based analyses for flow-insensitive interprocedural analysis.

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

2. 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 [3, 5] (§3.1), def-use analysis (for memory, registers, and flags) (§3.2), aggregate structure identification [6] (§3.3), and generation of symbolic expressions for an instruction's semantics (§3.4).

3. Established Applicability. The capabilities of our approach have been demonstrated by writing specifications for IA32 and PowerPC32. These are nearly complete specifications of the 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).

The abstract transformers for these analyses that are created from the IA32 TSL specifications have been put together to create a system that essentially duplicates CodeSurfer/x86 [4]. A similar system for PowerPC32 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 for which they have been used. 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. As far as we know, TSL is the first system with an instruction-set-specification language and support for 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), 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 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 explains how CIR is instantiated to create an analyzer for a specific analysis component. §4 describes quirky features of several instruction sets, and discusses how those features are handled in TSL. §5 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 is only semi-formal: it uses a mixture of English and pseudo-code.

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

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

Our work is based on completely formal specifications, which are written in a language that we designed (TSL). TSL is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. 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 [14]. 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 (∼, &, |, ˆ, ≪, ≫, 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.

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

```
[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]     return ans;
[38]   }
[39]};
```

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

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 §3). 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-defined 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 immediate, direct register, 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.

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*);

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 lines 23–33 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-syntax classes (lines 3–15 of Fig. 3). 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 appropriate parent abstract C++ class. Each user-defined function is translated to a static CIR 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 INTERP::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 §3.1),

[4] The TSL front end performs *with-normalization*, which transforms all multi-level with expressions to use only one-level patterns, via the pattern-compilation algorithm from [18, 23].

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]};
```

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 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]};
```

def-use analysis (DUA, see §3.2), and quantifier-free bit-vector semantics (QFBV, see §3.4).

Each interpretation defines an abstract domain. For example, line 3 of each column defines the abstract-domain class for INT32: ValueSet32, UseSet, and QFBVTerm32. To define an interpretation, one needs to define 42 basetype operators, most of which have four variants, for 8-, 16-, 32-, and 64-bit integers, as well as 12 map *access/update* operations. 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 §3).

2.4 Generated Transformers

The TSL system provides considerable leverage for implementing static-analysis tools and experimenting with new ones. New static-analyses are easily implemented: each static-analysis component is created via abstract interpretation of the TSL code that defines the concrete semantics of the instruction set. In particular, all abstract interpretation is performed at the *meta-level*: an analysis designer adds a new analysis component to the TSL system by (i) redefining the TSL basetypes, and (ii) providing a set of alternative interpretations for the primitive operations on basetypes. This implicitly defines an alternative interpretation of each expression and function in an instruction-set's concrete operational

semantics, and thereby yields an abstract semantics for an instruction set from its concrete operational semantics.

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 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.DUA	[ebx $\mapsto$ {eax, ebx}, ZF $\mapsto$ {eax, ebx}, . . .]
3.QFBV	(ebx$'$ = ebx+^{32}eax) $\wedge$ (ZF$'$ $\Leftrightarrow$ (ebx+^{32}eax = 0)) $\wedge$ (SF$'$ $\Leftrightarrow$ (ebx+^{32}eax< 0)) $\wedge$ . . .

2.5 Measures of Success

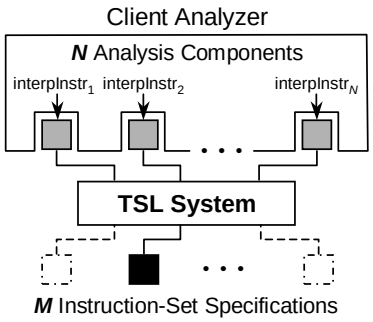


Fig. 4. The interaction between the TSL system and a client analyzer. The grey boxes represent TSL-generated analysis components.

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. 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. 4), i.e., $M + N$ inputs to obtain $M \times N$ analysis-component implementations.

One measure of success is demonstrated by our effort to use TSL to recreate the analysis components used in CodeSurfer/x86 [4]. 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 specification for IA32 consists of 2,834 (non-comment, non-blank) lines of TSL.

Because each analysis is defined at the meta-level (i.e., by providing an interpretation for the collection of TSL primitives), abstract transformers for a given analysis can be created automatically for *each* instruction set that is specified in TSL. For instance, from the PowerPC32 specification (1,370 non-comment, non-blank lines, which took approximately 4 days to write), we were immediately able to generate PowerPC32-specific versions of *all* of the analysis components that had been developed for the IA32 instruction set.

It is natural to ask how the TSL-generated analyses perform compared to their hand-coded counterparts. Due to the nature of the transformers used in one of the analyses that we implemented (affine-relation analysis (ARA) [17]), it was possible to write an algorithm to compare the TSL-generated ARA transformers with 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 (cross-)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] For 87 cases, this was because in rethinking how the ARA abstraction could be encoded using TSL mechanisms, we discovered an easy way to extend [17] to retain some information for 8-, 16-, and 64-bit operations. (In principle, these could have been incorporated into the hand-coded version, too.)

The other 131 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 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 [22]. VSA uses this kind of analysis engine (§3.1).
- *Transformer Composition* [TC]. These generally perform flow-sensitive, context-sensitive interprocedural analysis. DUA (§3.2) uses this kind of analysis engine.
- *Unification-Based Analyses* [UB]. These perform flow-insensitive interprocedural analysis. ASI (§3.3) uses this kind of analysis engine.

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 employed 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[6] and return a transformer specification in the analysis component's input language.

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

3.1 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 [5]. 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 $\{lb,\ lb + s,\ lb + 2s,\ ...,\ lb + ks\}$ [21].

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

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 [21].
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 abstract memory-location (i.e., the abstraction of 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.

3.2 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. The abstract domain for the basetypes 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-Basetypes and Access/Update Functions. The abstract domains of the maps for DUA are dictionaries that map 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. 3) are used to create transformers for several additional analyses, such as GMOD analysis [10], 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 §3.4).

3.3 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 [6]. 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. The abstract domain for the basetypes 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-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 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]".

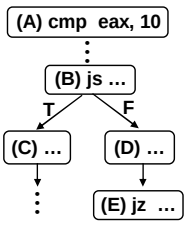
3.4 Quantifier-Free Bit-Vector (QFBV) Semantics

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 represents the operation. The abstract domain for BOOL is a formula, and each operator on it constructs a formula that represents 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 [16]. 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 [11] 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 shown above (A), 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.

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

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).

3.5 Paired Semantics

Our system allows easy instantiations of *reduced product*s [11] by means of *paired semantics*. The TSL system provides a template for paired semantics as shown in Fig. 5(a).

(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_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] };
```

(b)

```
[1] typedef PairedSemantics<VSA_INTERP, DUA_INTERP> DUA;
[2] template<> DUA::INT32 DUA::MemAccess_32_8_LE_32(
[3]                                 DUA::MEMMAP32_8_LE mem, DUA::INT32 addr) {
[4]   DUA::INTERP1::MEMMAP32_8_LE memory1 = mem.GetFirst();
[5]   DUA::INTERP2::MEMMAP32_8_LE 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. 5. (a) A part of the template class for paired semantics; (b) 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. 5(b). The communication between interpretations may take place in basetype-operators or *access/update* functions; Fig. 5(b) is an example of the latter. The two components of the paired-semantics values are deconstructed on lines 4–7 of Fig. 5(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. 5(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. 5(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.

4 Instruction Sets

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

4.1 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. (The former can yield more precise VSA results.)

Another characteristic to note is that IA32 keeps condition codes in a special register, called EFLAGS.[7] 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.

4.2 ARM

Almost all ARM instructions contain a condition field that allows an instruction to be executed conditionally, depending on condition-code flags. This feature

[7] Many other instruction sets, such as SPARC, PowerPC32, and ARM, also use a special register to store condition codes.

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(
[3]                   flagMap, EQ());
[4]       src = interpOperand(
[5]                curState, srcOprnd);
[6]       a = Var32Update(
[7]              regMap, destReg, src);
[8]       b = regMap;
[9]       answer = cond ? a : b;
[10]  in ( answer )
```

Fig. 6. 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. 6 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 §3.5 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*).

4.3 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. 7. 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. 7 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 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).

5 Related Work

There are many specification languages for instruction sets and many purposes for which they have been used, including emulation (hardware simulation) for cycle simulation, pipeline simulation, and compiler-optimization testing; retargeting of back-end phases, such as instruction scheduling, register assignment, and functional-unit binding; and concrete syntactic issues, such as instruction encoding and decoding. While some of the existing *languages* would have been satisfactory for our purposes, their *runtime components* were not satisfactory, which necessitated creating our own implementation. In particular, as mentioned in §1, we needed the runtime to (i) execute over abstract states, (ii) possibly propagate abstract states to more than one successor at a branch node, (iii) be able to compare abstract states and terminate abstract execution when a fixed point is reached, and (iv) apply widening operators, if necessary, to ensure termination.

Harcourt et al. [14] used ML to specify the semantics of instruction sets. LISAS [9] is a specification language that was developed based on their experience with ML. Their work particularly influenced the design of the TSL language.

TSL shares some of the same goals as λ-RTL [19] (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.[8] 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.

Discussion of additional work related to TSL can be found in [15].

Acknowledgements. We would like to thank Gogul Balakrishnan, Denis Gopan, and Susan Horwitz for their comments on drafts of this paper, and the anonymous referees for the helpful suggestions contained in their reviews.

References

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

[8] Recursion is not often used in specifications, but is needed for handling some loop-iteration instructions, such as the IA32 string-manipulation instructions and the PowerPC32 multiple-word load/store instructions.

2. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *IFPP*, 2000.
3. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, Univ. of Wisc., 2007.
4. G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.
5. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
6. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, 2007.
7. M. Christodorescu, W. Goh, and N. Kidd. String analysis for x86 binaries. In *PASTE*, 2005.
8. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM*, 1997.
9. T. A. Cook, P. D. Franzon, E. A. Harcourt, and T. K. Miller. System-level specification of instruction sets. In *DAC*, 1993.
10. K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, 1988.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
12. 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.
13. S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL*, 1998.
14. E. Harcourt, J. Mauney, and T. Cook. Functional specification and simulation of instruction set architectures. In *PLC*, 1994.
15. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. TR-1622, C.S. Dept., Univ. of Wisconsin, Madison, WI, Oct. 2007.
16. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.
17. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
18. M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In *CC*, 1992.
19. N. Ramsey and J. Davidson. Specifying instructions' semantics using λ-RTL. Unpublished manuscript, 1999.
20. J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *TECS*, 2005.
21. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, 2006.
22. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
23. P. Wadler. Efficient compilation of pattern-matching. *The Impl. of Func. Prog. Lang.*, 1987.
24. J. Zhang, R. Zhao, and J. Pang. Parameter and return-value analysis of binary executables. In *COMPSAC*, 2007.