A Framework Using a Common Language to Build Program Verifiers for Low-Level Languages
低級言語のプログラム検証器を構成するための共通言語を用いたフレームワーク

by

Toshihiro Yoshino
吉野 寿宏

A Master Thesis
修士論文

Submitted to
the Graduate School of Information Science and Technology
the University of Tokyo
on February 7, 2006
in Partial Fulfillment of the Requirements
for the Degree of Master of Information Science and Technology
in Computer Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Computer Science

ABSTRACT

Program verification is a technique to theoretically ensure that a program satisfies certain properties. For example, a type checker of typed assembly language (TAL) is one of such verifiers. The type checker ensures various useful properties by verifying type safety, including memory access safety which assures a program never accesses outside of the allocated memory area.

One problem of code verifiers for low-level languages (such as assembly languages and machine languages) is that it is generally hard to develop a verifier. This is mainly because low-level languages are very specific to the underlying architectures (i.e. CPUs). Therefore, each time we construct a verifier for a low-level language, we have to describe the semantics of the language and confirm the correctness of the description, which is a large burden in implementation. Also existing verifiers often do not consider portability and incorporate architecture-specific features even into the core of their verification logics.

To address this issue, we develop a general framework to verify low-level code. Our goal is to construct a common basis for developing code verifiers for low-level languages. We split a verifier into three parts: (1) a common language, (2) semantics-preserving program translators for the language and (3) a verification logic. In this framework, a program is verified after translated into the common language. If a program translator has semantics-preserving property, which is formalized in this paper, verification of the translated program is equivalent to that of the original program. Thus once a verifier is built for the common language, it is able to verify programs in the source low-level languages. For proof of concept, we implement translators from x86 and SPARC assembly to the common language, and also show semantics-preserving property of the translators.

論文要旨

プログラム検証は、プログラムがある性質を満たすことを理論的に検証する手法である。型付きアセンブリ言語 (TAL) の型検査器はそういった検証器の一つで、例えばプログラムが自分に割り当てられた範囲外のメモリを読み書きしないという、メモリアクセスの安全性などの有用な性質をプログラムに対して保証することができる。

しかし、TAL をはじめとする低級言語の検証器は、システムを作成するコストが高いという欠点がある。これは、アセンブリ言語などの低級な言語は、アーキテクチャへの依存性が高いためである。検証器を作成する際には言語の意味論を記述し、その記述の正しさを毎回確認しなければならないが、この作業にかかるコストが非常に大きい。さらにこれまでの実装では、システムの移植性を重視せず、検証器の中核部分にまでアーキテクチャ依存の部分が組み込まれてしまうことも多かった。

この問題を解決するために、低級言語コードを検証するための一般的なフレームワークを構築する。我々の目的は、低級言語のための検証器を作成するための共通基盤を構築することである。このフレームワークは、C 言語ライクな共通言語と、その言語に対する「正しい」プログラム変換器、そして検証ロジックから構成される。このフレームワークでは、まず検証対象のプログラムを共通言語に変換してから検証を行うが、プログラム変換が正しいならば、共通言語の上での検証は元のプログラムを検証することと同等である。したがって、一度共通言語の上に検証器が作成されたならば、変換元のプログラム言語におけるプログラムを検証することができるようになる。その後我々は、実際に x86 や SPARC のアセンブリ言語からの変換器を記述し、変換の正しさを示す。

Acknowledgements

First of all I would like to acknowledge the thesis supervisor, Professor Akinori Yonezawa.

Mr. Toshiyuki Maeda and Mr. Yutaka Oiwa gave me considerable advices about technical topics. Without their contribution, this thesis could not have been completed.

Also I appreciate all other supports by Yonezawa laboratory members and my friends who was actively discussing with me. Their comments were always suggestive, and I was able to refine my paper through the discussion.

I would like to thank my parents from my heart, too. They encouraged me when I was in trouble, and were always solicitous for me.

Deep gratitude for everyone who supported me to write this thesis in some ways. Thank you.

Contents

List of Figures

Chapter 1

Introduction

Program verification is a technique to theoretically ensure that a program satisfies certain properties. This technique is widely used especially in the field of security.

For example, typed assembly language (TAL), first proposed by Morrisett et al.[16, 15], is one of the systems that perform program verification. It is an assembly language extended with static type checking, like bytecode verification on Java VM[23, 13]. A TAL type checker ensures various useful properties by verifying type safety, including memory access safety which assures a program never accesses outside of the memory area allocated for the program.

Verifying low-level language programs directly has an advantage that we can reduce the trusted computing base (TCB). Conventionally, program verification is done mostly in higher-level or abstract languages, such as C, ML and lambda calculi. In this approach, a program has to be compiled after verification, thus we have to trust a compiler in addition to a verifier. On the other hand, programs in low-level languages like assembly languages and machine languages can be run on a real architecture without large modification, because an assembler is usually much simpler than compilers.

One common problem of code verifiers for low-level languages is the hardness in development. This is mainly because such languages are very specific to the underlying architectures (i.e. CPUs). There are many kinds of low-level languages and each of them has its own syntax and semantics. Thus each time we build a verifier for a low-level language, we first have to model its underlying architecture and describe the semantics of the language. These steps require substantial effort, because we then have to assure that the description is correct before we use it. Otherwise we cannot trust the verifier.

Also porting a verifier from one architecture to another generally requires a great amount of work. Existing verifiers often do not consider portability and incorporate architecture-specific features even into the core of their verification logics. Such sys-

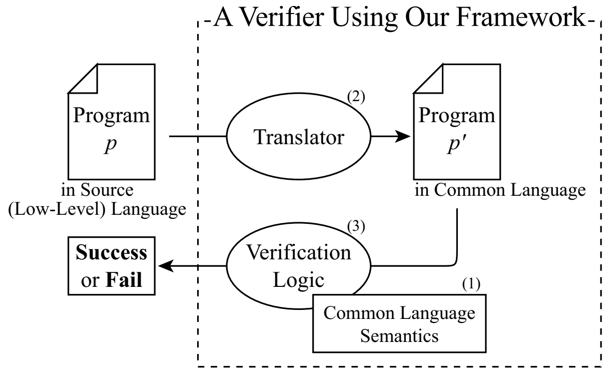


Figure 1.1: Construction of a verifier using our framework

tems, even though an implementation is given, cannot be ported easily. We will have to re-implement a large part of the system when we try to port them; this is just reinvention of a wheel and we consider this as a large burden in implementation. Thus it is important to make a verification logic explicitly architecture-independent.

In this paper, we propose a framework using a common language to build program verifiers for low-level languages, named L^{3}Cover. Our goal is to construct a common basis for code verifiers which helps the user develop a code verifier for low-level languages.

In our framework, a complete verifier is split into three parts as in Figure 1.1: (1) a common language, (2) a semantics-preserving program translator from a low-level language to the common language and (3) a verification logic for the common language. Given a program, the translator first translates it to the common language, and then the verification logic is used for the translated program. Their designs are independent from each other; this is crucial to make the system extensible. As many researches have been done on program verification logics, we do not discuss how we construct every verification logic, and we can simply apply the results of preceding researches. Thus in this research, we focused mainly on the common language and program translators.

In the rest of this paper, we explain the design and implementation of the framework for each component.

First we define the syntax and semantics of the common language ADL in Chapter 2. As program verifiers and translators are built using this semantics, ADL is exactly the core of L^{3}Cover framework. We design it as an imperative language with C-like syntax, so that it is more expressive than assembly-like languages. We implemented an interpreter of the common language using the defined semantics in order to discuss the correctness of translation. See Appendix A for the detailed information.

Next in Chapter 3, we discuss program translators. They are another important component of the system. Because a verifier is defined only for ADL, we have to translate a program to be verified into the language before using a verification logic. Therefore it is important to assure that translators preserve the semantics for any program. We discuss what is the correctness and how to build a correct, trusted translator.

For proof of concept, we implement translators from the subset of Intel x86 assembly and SPARC assembly. See Appendix B for the detailed information.

And finally in Chapter 4, we discuss verification logics. In this paper, we just model a verification logic as a blackbox function. Although assuring properties for a verifier is left for users, we define several properties which must be assured to build a *sound* verifier. Then we design an example verifier, and show that it is sound.

We use Java to implement the framework. The reasons we choose Java are the following threefold.

First, Java is an object-oriented programming language with class inheritance. Since our goal is to build a common basis for building a code verifier, the system must be extensible; it must be designed so that we can easily replace a verification logic and a program translator. We can utilize class inheritance for this purpose.

Second, Java bytecode is portable (in other words, architecture-independent). Therefore we can execute a verifier on any platforms for which a Java VM is implemented. This is a great advantage, because in many cases we want to verify a program on the same architecture as one which the program is written for.

Third, Java has plenty of class libraries to help the implementation. Fundamental data structures like list and map are already implemented as a default class library. We can avoid reinvention of a wheel by simply using these classes.

Chapter 2

ADL: The Common Language

This chapter describes the common language called ADL. ADL stands for architecture description language. The primary concern in designing the language is to be neutral to any specific architectures. We define the syntax and the semantics for the language.

2.1 Abstract Machine Model

There are many architectures today, such as Intel x86, PowerPC, ARM, SPARC, etc. However most of them do not differ much in terms of how they compute: they have several registers and memory to store data, and manipulate these data according to a program also stored in memory. They are all von Neumann architecture machines.

From this observation, we designed ADL to be an imperative programming language which manipulates the state of an abstract machine. The abstract machine has registers, memory and temporary variables to store program and data, just like those realistic architectures.

First we describe values used for computation in ADL. The language has three kinds of values as shown in Figure 2.1: integers, pointers and `junk`. This figure also expresses a value has type Value. Such types will appear when we write pseudo-code.

An integer n can express an arbitrary integer. The range is not limited for these values, because such limitations may weaken architecture-independence. Suppose we defined integers to be 32-bit values, whose range is $[-2^{31}, 2^{31}-1]$. Then it becomes

$$\begin{array}{llll} \text{Integer} & n & ::= & \cdots, -1, 0, 1, 2, \cdots \\ \text{Pointer} & m & ::= & \ell + n \\ \text{Value} & v & ::= & \texttt{junk} \mid n \mid m \qquad :: \mathsf{Value} \end{array}$$

Figure 2.1: ADL values

Byte Value	b	$\in$	$[0, 255]$	
Atomic Value	a	$::=$	$\texttt{junk} \mid b \mid m[n]$	:: Atom
Data	d	$::=$	$\langle a_0, a_1, \cdots, a_{n-1} \rangle$	:: Data
Memory Block	k	$::=$	$\langle c_0; c_1; \cdots; c_{n-1} \rangle$	(Code Block)
		$\mid$	d	(Data Block)
Register File	R	$=$	$\{\texttt{r}_1 \mapsto d_1, \texttt{r}_2 \mapsto d_2, \cdots, \texttt{r}_\texttt{N} \mapsto d_N\}$	
Memory	M	$=$	$\{\ell_1 \mapsto k_1, \ell_2 \mapsto k_2, \cdots\}$	
Temporary Variable	V	$=$	$\{A \mapsto v_1, B \mapsto v_2, \cdots\}$	
Machine State	S	$=$	(R, M, V, m)	

Figure 2.2: Storages in the abstract machine

hard, although not impossible, to express an architecture whose integers are 64-bit wide within this limitation. Moreover, the result of a computation may often overflow from an integer; for example, multiplication of two 32-bit integers produces a 64-bit integer. To handle these cases, we will need to incorporate special primitives into the language, and the definition of the language will be more and more complicated. Therefore we think it is unadvisable to limit the range of integers here.

A pointer in ADL is in fat-pointer form: a pair of a label ℓ, which identifies the memory block to be accessed, and a byte offset n from the beginning of the block. In many real machine languages[10, 11, 21], each label corresponds to an address which itself is an integer, and pointers are not distinguished from (normal) integers. However we employ this fat-pointer approach because it can simplify the semantics. Additionally, we can distinguish one memory block from another by using this formalization of pointers.

The last value, `junk`, is used to express an undetermined value. Its meaning is explained later, but roughly speaking it appears when a computation produces a value which cannot be determined without additional (or, architecture-specific) knowledge.

Next we formalize the abstract machine for ADL in Figure 2.2. It has three categories of storage as we discussed before: registers, memory and temporary variables.

Registers and memory are just as same as those used in many real architectures. Registers are distinguished by their names. The number of registers is a parameter, so that it can simulate many kinds of architectures. Memory consists of several memory blocks identified by labels. Each block can hold a fixed length of data or program code. The syntax of program code is defined later, in the next section.

As the size of these data storage is limited, a value must be encoded (and truncated if needed) into a data denoted as d in Figure 2.2 when we store it to a register or a memory block. An atomic value a expresses a one-byte-wide value, and a data d is defined as an array of atomic values. Each of three constructors of an atomic value obviously corresponds to those of a value described in Figure 2.1. Here $m[n]$ expresses the n-th byte of a pointer m in byte-array representation.

Temporary variables can, on the other hand, hold a value (defined in Figure 2.1), not a data. Typically they are used to store intermediate results of a computation in order to simplify notations.

We have just said that we need to encode a value when we store it to a register or memory. encode and decode functions are used for this purpose. encode function encodes a value into a data, and inversely decode function decodes a value from a data. They are also architecture-specific parameters. Endianness of value representation, for example, is absorbed here.

encode is almost the inverse function of decode. However an encoded data can contain more than one kinds of atomic values, and in such cases decode returns `junk`. Also an encoded data has limitation in the width of data, encode may truncate a value. The conditions that these functions must satisfy are formalized as below.

- $\mathsf{decode}\ d = \mathtt{junk}$ when d is heterogeneous or includes `junk`
- $\mathsf{encode}\ n\ \mathtt{junk} = \underbrace{\langle \mathtt{junk}, \cdots, \mathtt{junk} \rangle}_{n \text{ times}}$
- A byte array is decoded to an integer: $\mathsf{decode}\ \langle b_0, \cdots, b_{n-1} \rangle = n'$
  An integer is encoded into an array containing byte values: $\mathsf{encode}\ s\ n = \langle b_0, \cdots, b_{s-1} \rangle$
- ${}^{\forall} n.\ \mathsf{encode}\ n\ (\mathsf{decode}\ \langle b_0, \cdots, b_{n-1} \rangle) = \langle b_0, \cdots, b_{n-1} \rangle$
- A pointer is encoded into an array containing pointer values: $\mathsf{encode}\ n\ (\ell + o) = \langle \ell + o[0], \cdots, \ell + o[n-1] \rangle$[1]
- $\mathsf{decode}\ \langle \ell + o[0], \cdots, \ell + o[n-1] \rangle$ returns a pointer $\ell + o$ iff. n is equal to the size of pointer on the architecture (this is also an architecture-specific parameter). Otherwise it should return `junk`.

2.2 Language Syntax

We show the syntax of ADL program in Figure 2.3. A program in ADL is constructed by commands, like assembly languages of real architectures. There are seven primitive commands in the language including assignment, unconditional jump and conditional executions.

[1]Or alternatively it may return an array of `junk`. Since we discuss the conservative formalization, such operations can be considered as bogus.

Left Value	l	$::=$	$\mathtt{r_i}[n, n]$	(Register)
		$\mid$	$*[n]e_v$	(Memory Reference)
Expression	e_v	$::=$	v	(Literal)
		$\mid$	l	(Left value)
		$\mid$	x	(Variable)
		$\mid$	$e_v\ op_b\ e_v \mid op_u e_v$	(Arithmetic)
	op_b	$::=$	$+ \mid - \mid * \mid / \mid \% \mid \& \mid \mid \mid \hat{}$	
	op_u	$::=$	$- \mid \tilde{}$	
Boolean Expr.	e_b	$::=$	$e_v\ cmp\ e_v$	(Comparison)
		$\mid$	$e_b \wedge e_b \mid e_b \vee e_b$	(Logical Operator)
		$\mid$	$!e_b$	(Negation)
	cmp	$::=$	$== \mid != \mid \cdots$	
Command	c	$::=$	`nop`	(No Operation)
		$\mid$	`error`	(Runtime Error)
		$\mid$	$l = e_v$	(Assignment)
		$\mid$	$x = e_v$	(Variable Definition)
		$\mid$	`goto` e_v	(Jump)
		$\mid$	`if` e_b `then` c `else` c	(Conditional)
		$\mid$	`if` $e_v : kind$ `then` c `else` c	(Conditional by Kind)
	$kind$	$::=$	`junk` $\mid$ `int` $\mid$ `pointer`	

Figure 2.3: ADL syntax

Expressions in ADL are similar to those in C. We can write complex expressions using infix operators and parentheses. The notation of operators is also taken from C, thus those familiar to C will be able to read or write ADL easily. Left values are assignable values; they point either a register or a memory block. Numbers enclosed in a bracket specify the offset (for registers only) and the size of data to be accessed.

There are two kinds of expressions: one is expressions evaluated to a value (e_v). The other is boolean expressions (e_b) that are used only as a condition for conditional commands. Boolean expressions are evaluated to either true or false, but actually the language uses these values only internally.

Although ADL has only unconditional branch (`goto` command), conditional branches can also be achieved by combining `goto` command and `if` command. In several works[5, 16] on the program verification for low-level languages, there are both unconditional and several conditional branch primitives in the language as in many assembly languages. However this approach restricts the branch condition; we have to combine multiple branch instructions when we want to branch by a complex condition. Meanwhile, we can describe any conditions in one command by using `if`, and it will reduce the program size.

Actually `if` can be said as a more general mechanism than such an approach. For example, we can write also conditional assignment, which is conventionally expressed by using a conditional branch and assignment statements in different basic blocks.

2.3 Language Semantics

We define the semantics for ADL conservatively in this section. ADL consists of two levels of elements: one is expressions without a side effect, and the other is commands which modify the machine state. First we put several restrictions on the machine model. After that, we define the operational semantics for each level.

Jumps are restricted only to the top of a block This restriction clarifies that a program is structured by basic blocks. It does not weaken expressiveness of the language, because we just have to place a label for all positions of the code where a branch instruction jumps to.

Many verification algorithms for low-level languages traverse a control flow graph (CFG), whose nodes are basic blocks. By this restriction each memory block becomes a basic block, therefore it becomes easier to design a verification logic.

Memory blocks are distant Any two memory blocks are neither adjacent nor overlapped. It also means that we put no assumption on the memory layout of a program.

In standard assembly languages, accessing beyond the boundary of a memory block is not prohibited. And in such cases the contents of the next or posterior block is read, as long as a pointer points within a valid page. Although the precision of simulation goes higher by bringing this feature in, the semantics becomes much more complicated at the same time. It has also disadvantage that it requires the knowledge on the memory layout, which is not determined at the level of assembly language.

Moreover, this feature is often exploited by malicious code, and causes a serious security problem called *buffer overflow*. Therefore it is safer to check the array boundary.

Code and data are completely separated from each other A program cannot

$$\frac{}{\vdash v \Downarrow v}\ (\textsc{E-Val}) \qquad \frac{\textbf{assert}(x \in \mathit{dom}(V)) \quad V(x) = v}{(R, M, V) \vdash x \Downarrow v}\ (\textsc{E-Var})$$

$$\frac{\begin{array}{c}\textbf{assert}(\mathtt{r_i} \in \mathit{dom}(R)) \quad R(\mathtt{r_i}) = \langle a_0, \cdots, a_{n-1}\rangle \\ \textbf{assert}(0 \leqslant o < n \wedge 0 < s \leqslant n - o) \\ \mathsf{decode}\ \langle a_b, \cdots, a_{b+s-1}\rangle = v\end{array}}{(R, M, V) \vdash \mathtt{r_i}[o, s] \Downarrow v}\ (\textsc{E-Reg})$$

$$\frac{\begin{array}{c}(R, M, V) \vdash e \Downarrow v \quad \textbf{assert}(v = \ell + o \wedge \ell \in \mathit{dom}(M)) \\ M(\ell) = d \quad \textbf{assert}(d = \langle a_0, \cdots, a_{n-1}\rangle) \\ \textbf{assert}(0 \leqslant o < n \wedge 0 < s \leqslant n - o) \\ \mathsf{decode}\ \langle a_o, \cdots, a_{o+s-1}\rangle = v'\end{array}}{(R, M, V) \vdash *[s]e \Downarrow v'}\ (\textsc{E-Mem})$$

$$\frac{C \vdash e \Downarrow v \quad \mathit{op}\ v = v'}{C \vdash \mathit{op}\ e \Downarrow v'}\ (\textsc{E-UnArith})$$

$$\frac{C \vdash e_1 \Downarrow v_1 \quad C \vdash e_2 \Downarrow v_2 \quad v_1\ \mathit{op}\ v_2 = v'}{C \vdash e_1\ \mathit{op}\ e_2 \Downarrow v'}\ (\textsc{E-BinArith})$$

$$\frac{\begin{array}{c}\text{When an assertion failed in} \\ \text{evaluating } e \text{ under the context } C\end{array}}{C \vdash e \Downarrow \textbf{error}}\ (\textsc{E-ExprError})$$

Figure 2.4: ADL semantics: expressions

read from or write to a code block, and similarly the control flow cannot jump into a data block. This restriction simplifies the semantics, because we do not read or write code from a program, and thus do not need to encode nor decode program as a byte array.

In preceding researches like Foundational PCC[2] and TALT[5], program code was formalized as a byte array just like normal data. Thus we had to implement an instruction decoder function, although most programs do neither read nor write code. ADL explicitly separates them, and we do not need to model instruction fetch and decode phases any more.

About the appropriateness of these restrictions, we discuss later in Section 2.5.

2.3.1 Expressions

The semantics of expressions is shown in Figure 2.4. A triple $C \vdash e \Downarrow v$ means that an expression e evaluates to a value v under the context $C = (R, M, V)$. R, M and V

are registers, memory and temporary variables, respectively. The evaluation result v can be either an ADL value (defined in Figure 2.1) or a runtime error value denoted as **error**.

Because the contents of registers and memory blocks are encoded values, decoding occurs automatically when we use their values, as defined in E-Reg and E-Mem rules. When an expression does not satisfy all the assertions within the derivation tree for it, a runtime error occurs (by E-ExprError rule). For example, an error occurs when a program attempted to access beyond the size of a register, or when a program tried to dereference a non-pointer value.

Figure 2.5 shows the arithmetics between two values. In this formalization, $\lfloor x \rfloor$ rounds to negative infinity, i.e. it rounds a number to the largest integer less than or equal to the number. All operations except addition and subtract are valid only for integer arguments, otherwise they produce a `junk` value without causing a runtime error. Only addition and subtract are allowed to manipulate pointer values. As these rules are a bit complicated, they are shown in tables.

For bitwise operations, every number is treated as a virtually infinite bit sequence in 2's complement representation. Division and remainder for negative numbers satisfy the following rules.

- Quotient q is the largest integer less than or equal to $\frac{n}{p}$, when the sign of n and p are the same. Otherwise, q is the smallest integer greater than or equal to $\frac{n}{p}$. In other words, the quotient is rounded toward zero.
- Quotient q and remainder r must satisfy $n = pq + r$ for all n, p.

For example, $(q, r) = (-1, -2)$ for $(n, p) = (-7, 5)$, instead of $(q, r) = (-2, 3)$. On the other hand, $(q, r) = (1, -2)$ for $(n, p) = (-7, -5)$.

`junk` value represents a value which cannot be determined under the model of ADL. Suppose an expression like $(\ell_2 + 0) - (\ell_1 + 0)$. Apparently the result of this expression depends to the memory layout of a program. As we do not put any assumption for the memory layout, the value of this expression cannot be deteremined here.

Also suppose an expression like $(x + 3)\&(\tilde{}3)$. This arithmetic is well-known as alignment calculation. However, in order to perform these operations for a pointer, we have to know the characteristics of pointers and the memory layout on an architecture. By writing this, the programmer probably assumes that every memory block is placed in 4-byte alignment. This expression can be simulated by introducing this assumption, but this knowledge is obviously architecture-dependent.

The semantics for boolean expressions is shown in Figure 2.6. They are classified into two categories: (1) comparison and (2) logical expressions.

Comparison takes two expressions (e_v in Figure 2.3) and compares the result values. It can be defined only on two integers or two pointers with the same label. If two values are of different kinds or pointers with different labels, it causes a runtime error. In order to avoid an error, ADL has conditional by kind of a value in addition to (normal) conditional by comparison and logical operators. See Appendix B for several examples of the use of this command.

2.3.2 Commands

Next, the semantics for commands is shown in Figure 2.8. A triple $S \vdash c \Downarrow S'$ means that the state changes from S to S' when a command c is executed. A state is either (R, M, V, m), a quadruple of registers, memory, variables and the *next* instruction pointer, or **error**.

update function used in the figure is defined as follows. This function simply substitutes the destination data with the encoded value.

```
$\mathsf{update} \quad :: \mathsf{Data} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Value} \to \mathsf{Data}$
$\mathsf{update}\ \langle a_0, \cdots, a_{n-1}\rangle\ o\ s\ v =$
        $\mathbf{if}\ 0 \leqslant o < n \wedge 0 < s < n - o\ \mathbf{then}$
            $\langle a_0, \cdots, a_{o-1}, a'_0, \cdots, a'_{s-1}, a_{o+s}, \cdots, a_{n-1}\rangle$
        $\mathbf{else}$
            $\mathbf{error}$
    where $\mathsf{encode}\ s\ v = \langle a'_0, \cdots, a'_{s-1}\rangle$
```

Finally we define step execution relation $S \rightsquigarrow S'$ as follows. It is a binary relation $\rightsquigarrow \subseteq \mathcal{S} \times \mathcal{S}^+$, where $\mathcal{S}$ is a set of all possible states in the form (R, M, V, m) and $\mathcal{S}^+ = \mathcal{S} \cup \mathbf{error}$.

$$\frac{\begin{array}{c} m = \ell + i \quad \mathbf{assert}(\ell \in dom(M)) \quad M(\ell) = d \\ \mathbf{assert}(d = \langle c_0; c_1; \cdots; c_{n-1}\rangle \wedge 0 \leqslant i < n) \\ (R, M, V, \ell + (i+1)) \vdash c_i \Downarrow S' \end{array}}{(R, M, V, m) \rightsquigarrow S'}$$

This rule automatically extracts an instruction from memory, and performs evaluation of the instruction. Notice that we cannot advance any more if $S' = \mathbf{error} \notin \mathcal{S}$.

2.4 A Simple Example

Suppose an architecture with 32-bit machine word; there are several 32-bit registers, and the size of pointers on the architecture is also 32-bit. Below is a program which computes the sum of all integers in a linked list. An identifier prefixed by `%` denotes a register. Similarly, an identifier prefixed by `&` denotes a pointer to a label.

```
null: // the end of list

data: ...

main:   /* Start Symbol */
        %r1 = &data;
        %r2 = 0;
        goto &lp; /* Required because execution flow does NOT
                     automatically go into the next block */

lp:     /* Sum of integers in a list */
        %r2 = %r2 + *[4](%r1);
                        // *[4](...) references 4-byte value
        %r1 = *[4](%r1 + 4);
                        // take cdr part
        if %r1 - &null : int then
          if %r1 == &null then goto &ed else nop
        else
          nop;
        goto &lp;

ed:     /* Tail of the loop */
        goto &ed;       // halt
```

An equivalent program will be written like the following in C.

```
struct list {
  int car;
  struct list *cdr;
};

struct list data = ...;

int main()
{
  /* main */
  struct list *cursor = &data; /* <= r1 */
  int sum = 0; /* <= r2 */

  /* lp */
  do {
    sum += cursor->car;
    cursor = cursor->cdr;
  } while(cursor != NULL);

  /* ed */
  for(;;) ;
}
```

One important thing is that the end of each code block has `goto` command to the next block in spite that the next block is written directly adjacent to a block.

This is because we do not put any assumption in memory layout. We cannot figure out which block is next to a block in this model. Thus we have to specify adjacency of code blocks explicitly. Otherwise, execution flow falls off the end of a block, and the interpreter goes to **error** state.

For example, suppose a linked list beginning from `data` contains two integers 1 and 2.

```
data:
    .data[4] 1      // 4-byte data containing the integer 1
    .data[4] &data2

data2:
    .data[4] 2
    .data[4] &null
```

First time the interpreter reaches the label `lp`, the register `r1` points to `data`, and `r2` contains zero. After two commands are executed, `r2` changes to 1 and `r1` points to `data2`. Since `data2` and `null` have different labels, the expression `%r1 - &null` evaluates to `junk`. Thus `nop` command in the else-branch is executed, and the interpreter goes back to the label `lp`.

Execution advances similarly, but this time `r2` changes to 3 $(= 1 + 2)$ and `r1` points to `null`. Here the content of `r1` and `null` are comparable and returns an integer, thus the then-branch of the outer `if`-command is executed. The expression `%r1 - &null` obviously evaluates to 0, and also `%r1 == &null` holds. Then the `goto`-command in the then-branch of the inner `if`-command is executed, and the interpreter jumps to the label `ed`.

`ed` indicates the termination of a program. Since ADL does not have a primitive to terminate a program, we simply implement it as an infinite loop.

2.5 Discussion

As we put several restrictions on the semantics, there are several kinds of programs which cannot be expressed in this language.

One example of such programs is a program which dynamically generates the code to be executed. This feature is utilized by several programs: for example, a CPU emulator QEmu[3]. In many architectures, one large and flat memory space is shared for code and data. Thus we can generate machine code on data area, and jump to this generated code.

Most of these programs utilize this feature for the sake of performance. However, the same thing (dynamically change the execution path) can be realized by another method, such as using function pointers. Because patterns of generated code are fixed, we can implement each pattern by a function. Thus we don't think that treating code as data is compulsory.

Another example is a program with unfettered pointer arithmetics. Pointer arithmetics is tightly restricted in the defined semantics, because we did not assume anything about memory layout of a program. Only addition and subtract can be performed for pointers, whereas other operations produce `junk` value when given a pointer value. For example, the code like the following is not allowed in ADL.

```
char *p = "Hello", *q = "World";
...
p += q - p;
... /* Here p == q is expected */
```

In many real architectures, `p == q` holds after executing the third line[2], while in ADL, `q - p` evaluates to `junk`, and thus `p` contains `junk` after the third line. The important point is, though, that the evaluation results correspond naturally with those in real architectures, unless they do not evaluate to `junk`.

Actually, most of the pointer operations are used for accessing arrays and structures. Thus we think the following two operations are sufficient for these purposes.

1. Adding an integer to a pointer
2. Calculating the offset of two pointers in the same memory block

We designed ADL so that it can perform both of these operations correctly.

Anyway, it is not completeness, but soundness that we pursuit. We do not think it is necessary for the language to be able to simulate any programs which can be expressed on any architecture. We would rather be conservative than permissive when we try to ensure certain properties theoretically.

[2]According to the specification of C language[1], when subtracting a pointer from another pointer, both of them shall point elements of the same array object, or one past the last element of the array object. Therefore the result of the code above is unspecified in the specification, whereas it runs on many C compilers on many architectures as expected.

$\boxed{v_1 + v_2}$

$v_1 \setminus v_2$	`junk`	n_2	$\ell_2 + n_2$
`junk`	`junk`	`junk`	`junk`
n_1	`junk`	$n_1 + n_2$	$\ell_2 + (n_1 + n_2)$
$\ell_1 + n_1$	`junk`	$\ell_1 + (n_1 + n_2)$	`junk`

$\boxed{v_1 - v_2}$

$v_1 \setminus v_2$	`junk`	n_2	$\ell_2 + n_2$
`junk`	`junk`	`junk`	`junk`
n_1	`junk`	$n_1 - n_2$	`junk`
$\ell_1 + n_1$	`junk`	$\ell_1 + (n_1 - n_2)$	$n_1 - n_2$ where $\ell_1 = \ell_2$; `junk` otherwise

$$\boxed{v_1 * v_2} \quad \begin{cases} n_1 * n_2 & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 / v_2} \quad \begin{cases} n_1 \operatorname{div} n_2 & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 \,\%\, v_2} \quad \begin{cases} n_1 \bmod n_2 & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{-v} \quad \begin{cases} -n & \text{where } v = n \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 \,\&\, v_2} \quad \begin{cases} n_1 \text{ and } n_2 & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 \mid v_2} \quad \begin{cases} n_1 \text{ or } n_2 & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 \hat{\ } v_2} \quad \begin{cases} n_1 \text{ xor } n_2 & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{\tilde{\ } v} \quad \begin{cases} \text{not } n & \text{where } v = n \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 \ll v_2} \quad \begin{cases} n_1 \cdot 2^{n_2} & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

$$\boxed{v_1 \gg v_2} \quad \begin{cases} \lfloor \frac{n_1}{2^{n_2}} \rfloor & \text{where } v_1 = n_1, v_2 = n_2 \\ \texttt{junk} & \text{otherwise} \end{cases}$$

Figure 2.5: ADL semantics: arithmetics

$$\frac{C \vdash e_1 \Downarrow v_1 \quad C \vdash e_2 \Downarrow v_2 \qquad v_1 \lesseqgtr v_2 \quad \vdash v_1 \ \mathit{cmp} \ v_2}{C \vdash e_1 \ \mathit{cmp} \ e_2 \Downarrow \mathsf{true}} \ (\text{E-CmpTrue})$$

$$\frac{C \vdash e_1 \Downarrow v_1 \quad C \vdash e_2 \Downarrow v_2 \qquad v_1 \lesseqgtr v_2 \quad \nvdash v_1 \ \mathit{cmp} \ v_2}{C \vdash e_1 \ \mathit{cmp} \ e_2 \Downarrow \mathsf{false}} \ (\text{E-CmpFalse})$$

$$\frac{C \vdash e_1 \Downarrow v_1 \quad C \vdash e_2 \Downarrow v_2 \quad v_1 \not\lesseqgtr v_2}{C \vdash e_1 \ \mathit{cmp} \ e_2 \Downarrow \mathbf{error}} \ (\text{E-CmpError})$$

$$\frac{C \vdash e_1 \Downarrow b \quad b \neq \mathsf{true}}{C \vdash e_1 \wedge e_2 \Downarrow b} \ (\text{E-And1}) \qquad \frac{C \vdash e_1 \Downarrow \mathsf{true} \quad C \vdash e_2 \Downarrow b}{C \vdash e_1 \wedge e_2 \Downarrow b} \ (\text{E-And2})$$

$$\frac{C \vdash e_1 \Downarrow b \quad b \neq \mathsf{false}}{C \vdash e_1 \vee e_2 \Downarrow b} \ (\text{E-Or1}) \qquad \frac{C \vdash e_1 \Downarrow \mathsf{false} \quad C \vdash e_2 \Downarrow b}{C \vdash e_1 \vee e_2 \Downarrow b} \ (\text{E-Or2})$$

$$\frac{C \vdash e \Downarrow b \quad b \neq \mathbf{error}}{C \vdash \,!\,e \Downarrow \neg b} \ (\text{E-Not}) \qquad \frac{C \vdash e \Downarrow \mathbf{error}}{C \vdash \,!\,e \Downarrow \mathbf{error}} \ (\text{E-NotError})$$

$\boxed{v_1 \lesseqgtr v_2 \text{ iff.}}$

$v_1 \setminus v_2$	`junk`	n_2	$\ell_2 + n_2$
`junk`	never.		
n_1	never.	always.	never.
$\ell_1 + n_1$	never.	never.	$\ell_1 = \ell_2$

Figure 2.6: ADL semantics: boolean expressions

$$\frac{}{\vdash \texttt{junk} : \texttt{junk}} \qquad \frac{}{\vdash n : \texttt{int}} \qquad \frac{}{\vdash \ell + n : \texttt{pointer}}$$

Figure 2.7: Kind relations

$$\frac{}{S \vdash \texttt{nop} \Downarrow S}\ (\textsc{E-Nop}) \qquad \frac{}{S \vdash \texttt{error} \Downarrow \mathbf{error}}\ (\textsc{E-Error})$$

$$\frac{\begin{array}{c}\mathbf{assert}(\texttt{r}_\texttt{i} \in \mathit{dom}(R)) \quad R(\texttt{r}_\texttt{i}) = d = \langle a_0, \cdots, a_{n-1}\rangle \\ \mathbf{assert}(0 \leqslant o < n \wedge 0 < s \leqslant n - o) \quad (R, M, V) \vdash e \Downarrow v \\ \mathsf{update}\ d\ o\ s\ v = d' \quad R' = (R - \{\texttt{r}_\texttt{i}\}) \cup \{\texttt{r}_\texttt{i} \mapsto d'\}\end{array}}{(R, M, V, m) \vdash \texttt{r}_\texttt{i}[o, s] = e \Downarrow (R', M, V, m)}\ (\textsc{E-AssnReg})$$

$$\frac{\begin{array}{c}(R, M, V) \vdash e_1 \Downarrow v \quad (R, M, V) \vdash e_2 \Downarrow v' \\ \mathbf{assert}(v = \ell + o \wedge \ell \in \mathit{dom}(M)) \quad M(\ell) = d \\ \mathbf{assert}(d = \langle a_0, \cdots, a_{n-1}\rangle \wedge 0 \leqslant o < n \wedge 0 < s \leqslant n - o) \\ \mathsf{update}\ d\ o\ s\ v' = d' \quad M' = (M - \{\ell\}) \cup \{\ell \mapsto d'\}\end{array}}{(R, M, V, m) \vdash *[s]e_1 = e_2 \Downarrow (R, M', V, m)}\ (\textsc{E-AssnMem})$$

$$\frac{(R, M, V) \vdash e \Downarrow v \quad V' = (V - \{x\}) \cup \{x \mapsto v\}}{(R, M, V, m) \vdash x = e \Downarrow (R, M, V', m)}\ (\textsc{E-AssnVar})$$

$$\frac{\begin{array}{c}(R, M, V) \vdash e \Downarrow v \quad \mathbf{assert}(v = \ell + 0) \\ M(\ell) = d \quad \mathbf{assert}(d = \langle c_0; \cdots\rangle)\end{array}}{(R, M, V, m) \vdash \mathtt{goto}\, e \Downarrow (R, M, V, \ell + 0)}\ (\textsc{E-GoTo})$$

$$\frac{(R, M, V) \vdash e_b \Downarrow \mathsf{true} \quad (R, M, V, m) \vdash c_1 \Downarrow S'}{(R, M, V, m) \vdash \mathtt{if}\, e_b\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 \Downarrow S'}\ (\textsc{E-IfTrue})$$

$$\frac{(R, M, V) \vdash e_b \Downarrow \mathsf{false} \quad (R, M, V, m) \vdash c_2 \Downarrow S'}{(R, M, V, m) \vdash \mathtt{if}\, e_b\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 \Downarrow S'}\ (\textsc{E-IfFalse})$$

$$\frac{\begin{array}{c}(R, M, V) \vdash e \Downarrow v \quad \vdash v : k \\ (R, M, V, m) \vdash c_1 \Downarrow S'\end{array}}{(R, M, V, m) \vdash \mathtt{if}\, e : k\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 \Downarrow S'}\ (\textsc{E-KindTrue})$$

$$\frac{\begin{array}{c}(R, M, V) \vdash e \Downarrow v \quad \nvdash v : k \\ (R, M, V, m) \vdash c_2 \Downarrow S'\end{array}}{(R, M, V, m) \vdash \mathtt{if}\, e : k\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2 \Downarrow S'}\ (\textsc{E-KindFalse})$$

$$\frac{\exists e\, \mathbf{in}\, c.\ (R, M, V) \vdash e \Downarrow \mathbf{error}}{(R, M, V, m) \vdash c \Downarrow \mathbf{error}}\ (\textsc{E-ImpError1})$$

$$\frac{\begin{array}{c}\text{When an assertion failed in} \\ \text{executing } c \text{ from the state } S\end{array}}{S \vdash c \Downarrow \mathbf{error}}\ (\textsc{E-ImpError2})$$

Figure 2.8: ADL semantics: commands

Chapter 3

Program Translators

In this chapter, we describe program translators. We first define the model of source languages and the translation algorithm. Then we discuss what is the correctness of translation and how we can confirm it.

3.1 Modeling an Instruction

First of all, we start from modeling the source language of translation.

An instruction in many assembly languages and machine languages consists of two elements. One is an operation, which decides what kind of computation is to be done. The other is operands, which specify from where a value is loaded, and to where the result is stored.

Thus we model an operation as a code template, and operands as template parameters. A code template is a command sequence including zero or more template parameters inside. Given enough parameters, it is instantiated into concrete commands. Each operand is any of the following: an immediate value, a register reference and a memory reference.

We explain this formalization in detail with an example of `mov`(move) instruction, which is a popular assembly instruction among many architectures. Essentially it is an instruction which copies a value to another place. For instance, when we want to copy the content of a register `r1` to `r2`, we use this instruction as `mov r2, r1`[1]. We can also write `mov r4, r3` to copy from `r3` to `r4`, etc. Or in several architectures (such as x86), we can even use `mov` instruction to load from or store to memory. Obviously it is a daunting task to enumerate all the possible pairs of operands.

Actually, they all do the same thing: they just copy the content of the second

[1]Here we assume the destination operand comes before sources. However in several systems, for example GNU as, the order is reversed.

Identifier	*id*	::=	[‘a’-‘z’] [‘a’-‘z’, ‘A’-‘Z’, ‘0’-‘9’, ‘_’]*
Variables	*var*	::=	[‘A’-‘Z’] [‘a’-‘z’, ‘A’-‘Z’, ‘0’-‘9’, ‘_’]*
Terms	t	::	Term
	t	::=	v
		|	*id* ‘(’ (t (‘,’ t)*)? ‘)’
Patterns	p	::	Pattern
	p	::=	v
		|	*var*
		|	*id* ‘(’ (p (‘,’ p)*)? ‘)’

Figure 3.1: BNF syntax of terms and patterns

operand to the first one. In ADL, such an operation is written as `D = S`, when an instruction of the form `mov D, S` is given. Thus we can take this command as a template code with template variables `D` and `S`. By associating a pattern with template code, the translation rule for `mov` instruction can be defined as below. If we instantiate it with parameters `D = r2` and `S = r1`, it becomes a command corresponding to the first example.

```
mov(D, S) { D = S; }
```

This description may also be instantiated with an impossible combination of operands; for example, specifying memory to both of these two operands is invalid even in x86. However it will not be a problem, because real programs do not include such impossible instructions. The important thing about translators is that any *valid* instruction is correctly translated.

We decided to utilize Prolog-style pattern matching mechanism to implement this feature. A code template is associated with a pattern including zero or more pattern variables, and it is matched against a term. Pattern variables correspond to operands. Figure 3.1 illustrates the syntax of terms and patterns in Backus-Naur Form.

Next we define the pattern matching algorithm as in Figure 3.2. Given a pair of term and pattern, match function returns a variable binding if matching succeeded. Otherwise, **error** is returned.

We explain how it works by giving several examples.

- $a(b(c()))$ matches $a(X)$ with $\{X \mapsto b(c())\}$.
- $a(c())$ does not match $a(b())$, because the labels b and c differ.
- Neither $a()$ nor $a(b(), c())$ matches $a(X)$, because the numbers of children differ.

```
match                ::   (Term × Pattern) → (Ident → Term)
match $(t,p)$        =    match′ $\varnothing$ $(t,p)$

match′ :: (Ident → Term) → (Term × Pattern) → (Ident → Term)
match′ $V$ $(v,v')$  =    if $v = v'$ then $V$ else error
match′ $V$ $(t,X)$   =    if $X \in dom(V)$ then
                            if $t = V(X)$ then $V$ else error
                          else
                            $V \cup \{X \mapsto t\}$
match′ $V$ $(id(t_1,\cdots,t_n), id(p_1,\cdots,p_n))$ =
                          begin
                            $V' := V$;
                            for $i = 1$ to $n$ do
                              $V' :=$ match′ $V'$ $(t_i, p_i)$
                            end;
                            $V'$
                          end
match′ _ _           =    error
```

Figure 3.2: Pattern matching algorithm

Expression	e	$::=$	v	(Literal)
		$\mid$	$\mathtt{r_i}[e, e]$	(Register)
		$\mid$	$*[e]e$	(Memory Reference)
		$\mid$	x	(Variable)
		$\mid$	X	(Pattern Variable)
		$\mid$	$e\ op\ e \mid op\ e$	
Boolean Expr.	e_b	$::=$	$e\ cmp\ e$	
		$\mid$	$e_b\ lop\ e_b \mid lop\ e_b$	
Command	c	$::=$	$\mathtt{nop} \mid \mathtt{error}$	
		$\mid$	$e = e$	(Assignment)
		$\mid$	$\mathtt{goto}\, e$	(Jump)
		$\mid$	$\mathtt{if}\, e_b\, \mathtt{then}\, c\ \mathbf{else}\ c$	(Conditional)
		$\mid$	$\mathtt{if}\, e : k\, \mathtt{then}\, c\ \mathbf{else}\ c$	(Conditional by Kind)

Figure 3.3: Code template syntax

When one variable appears more than once in a pattern, corresponding subterms must have the identical form.

- $a(b(), b())$ matches $a(X, X)$ with $\{X \mapsto b()\}$.
- $a(b(c(), d()), b(c(), d()))$ also matches $a(X, X)$. In this case the variable binding is $\{X \mapsto b(c(), d())\}$.
- Neither $a(b(), c())$ nor $a(b(), b(c()))$ matches $a(X, X)$, because two children of a differ.

The syntax for code templates is shown in Figure 3.3. It is almost the same as ADL except that an expression can take a pattern variable. Operands are encapsulated into expressions e, because they can express all of three operand patterns.

However in order to allow a pattern variable to be used for the assignment destination, assignment takes two expressions in a code template. Left hand side expression must be instantiated either to a left value or a (temporary) variable, otherwise program translation will fail. The same argument also goes for offset and size of register and memory reference.

Figure 3.4 describes the instantiation rules for code templates. $\mathcal{I}_V[\![\bullet]\!]$ instantiates a command, a boolean expression or an expression with variable binding V. Here V must be a map from identifiers to *expressions*, whereas pattern matching only returns a map from identifier to term. Thus operand mapping, which translates a term into an expression, takes place here.

Expression

$$\begin{aligned}
\mathcal{I}_V[\![v]\!] &= v \\
\mathcal{I}_V[\![\mathtt{r_i}[e_1, e_2]]\!] &= \mathtt{r_i}[\mathcal{I}_V[\![e_1]\!], \mathcal{I}_V[\![e_2]\!]] \\
\mathcal{I}_V[\![*[e_1](e)]\!] &= *[\mathcal{I}_V[\![e_1]\!]](\mathcal{I}_V[\![e]\!]) \\
\mathcal{I}_V[\![X]\!] &= V(X) \\
\mathcal{I}_V[\![e_1\ op\ e_2]\!] &= \mathcal{I}_V[\![e_1]\!]\ op\ \mathcal{I}_V[\![e_2]\!] \\
\mathcal{I}_V[\![op\ e]\!] &= op\, \mathcal{I}_V[\![e]\!]
\end{aligned}$$

Boolean Expression

$$\begin{aligned}
\mathcal{I}_V[\![e_1\ cmp\ e_2]\!] &= \mathcal{I}_V[\![e_1]\!]\ cmp\ \mathcal{I}_V[\![e_2]\!] \\
\mathcal{I}_V[\![e_1\ lop\ e_2]\!] &= \mathcal{I}_V[\![e_1]\!]\ lop\ \mathcal{I}_V[\![e_2]\!] \\
\mathcal{I}_V[\![lop\ e]\!] &= lop\, \mathcal{I}_V[\![e]\!]
\end{aligned}$$

Command

$$\begin{aligned}
\mathcal{I}_V[\![\mathtt{nop}]\!] &= \mathtt{nop} \\
\mathcal{I}_V[\![\mathtt{error}]\!] &= \mathtt{error} \\
\mathcal{I}_V[\![e_1 = e_2]\!] &= \mathcal{I}_V[\![e_1]\!] = \mathcal{I}_V[\![e_2]\!] \\
\mathcal{I}_V[\![\mathtt{goto}\, e]\!] &= \mathtt{goto}\, \mathcal{I}_V[\![e]\!] \\
\mathcal{I}_V[\![\mathtt{if}\, e_b\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2]\!] &= \mathtt{if}\, \mathcal{I}_V[\![e_b]\!]\, \mathtt{then}\, \mathcal{I}_V[\![c_1]\!]\, \mathtt{else}\, \mathcal{I}_V[\![c_2]\!] \\
\mathcal{I}_V[\![\mathtt{if}\, e : k\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2]\!] &= \mathtt{if}\, \mathcal{I}_V[\![e]\!] : k\, \mathtt{then}\, \mathcal{I}_V[\![c_1]\!]\, \mathtt{else}\, \mathcal{I}_V[\![c_2]\!]
\end{aligned}$$

Command Sequence

$$\mathcal{I}_V[\![c_1; \cdots ; c_n]\!] = \mathcal{I}_V[\![c_1]\!]; \cdots ; \mathcal{I}_V[\![c_n]\!]$$

Figure 3.4: Code template instantiation rule

Finally a program translator is formalized as a pair of translation rules for operations and for operands. Each translation rule is a pair of a pattern and a template. Given a term, the translator converts it into an expression or a command sequence.

```
trans_op :: OperandRule → Term → Expression
trans_op $R\ t$    =  for$(p, e) \in R$ do
                        $M$ = match $(p, t)$;
                        if $M \neq$ error then
                          begin
                            $V = \{x \mapsto$ trans_op $R\ M(x) | x \in dom(M)\}$ ;
                            $\mathcal{I}_V[\![e]\!]$
                          end
                      end; error

trans_in :: (InstructionRule × OperandRule) → term → [Command]
trans_in $(I, O)\ t$   =  for$(p, cs) \in I$ do
                        $M$ = match $(p, t)$;
                        if $M \neq$ error then
                          begin
                            $V = \{x \mapsto$ trans_op $O\ M(x) | x \in dom(M)\}$ ;
                            $\mathcal{I}_V[\![cs]\!]$
                          end
                      end; error
```

trans_op function, operand mapping, translates a term to an expression. And trans_in function, instruction mapping, translates a term to a command sequence.

3.2 Translator Interfaces

From the model described in the previous section, we designed and implemented several classes for program translators.

The most important is `TranslatorBase` class below. It is the base class for all translators.

```
public abstract class TranslatorBase {
  // must be implemented in a subclass
  public abstract Expression translateOperand(Term term)
    throws TranslationException;

  // must be implemented in a subclass
  public abstract void translateInstruction(
    ProgramGenerator output, EventQueue input,
    Term term)
    throws TranslationException;

  protected void translateData(ProgramGenerator output,
                               Value value, int size)
    throws TranslationException { ... }

  public void performTranslation(ProgramGenerator output,
                                 EventQueue input)
    throws TranslationException { ... }

  protected TranslationContext getContext(Map/*<String, Term>*/ map)
    throws TranslationException {
    TranslationContext context = new TranslationContext();
    for(Iterator it = map.keySet().iterator(); it.hasNext(); ) {
      String key = (String) it.next();
      context.setVariable(key, translateOperand((Term) map.get(key)));
    }
    return context;
  }
};
```

Subclass implementation will look like the following.

```
public class SomeTranslator extends TranslatorBase {
  public Expression translateOperand(Term term)
    throws TranslationException {
    Map/*<String, Term>*/ map = new HashMap();
    TranslationContext context;

    // try a rule
    map.clear();
    if(pattern.matches(term, map)) {
      context = getContext(map);
      return ...;
    }
    // try another rule ...
    ...

    throw new TranslationException("Pattern match failed");
  }

  ...
};
```

`translateOperand` and `translateInstruction` implement the translation rules. They correspond to trans_op and trans_in functions described in the previous section, respectively.

Actually this implementation is more powerful than the formalization above; for example, an instruction rule can generate a label in the middle of translation. `translateInstruction` method takes `ProgramGenerator` class and `EventQueue` class in addition to a term encapsulated in `Term` class. These classes abstract programs as a stream containing three kinds of elements: instructions, data and labels.

```
public class ProgramGenerator {
  public ProgramGenerator() { }

  public Program getProgram() { ... }
  public void setProgram(Program program) { ... }

  public void startNextBlock(String labelName, int blockType)
    { ... }
  public void addCommand(Command command)
    throws TranslationException { ... }
  public void addData(Atom[] data)
    throws TranslationException { ... }

  public void endGeneration() { ... }
};

public interface EventQueue {
  public boolean isEmpty();

  public Event peek(int index)
    throws ArrayIndexOutOfBoundsException;
  public void remove(int index)
    throws ArrayIndexOutOfBoundsException;
};
```

`Event` class encapsulates an element of the source language: a term for an instruction, label name for a label or a pair of value and size for data. And `EventQueue` contains an ordered sequence of these events. Sometimes translation logic requires to look ahead of the event sequence; for example, we have to know the instruction position (i.e. label) after a function call, to where the execution flow returns from a function. This implementation allows look-ahead, and thus useful for implementing realistic translators.

`ProgramGenerator` receives a sequence of ADL program elements, and stores them into the specified program container (`Program` class). Adjacency of two code blocks is automatically inside `ProgramGenerator`, thus a translator has only to generate code corresponding to the source program without being meticulous.

`TranslationContext` is a map from name of variable to an expression. It is used to express variable binding for template instantiation.

```
public class TranslationContext {
  public Expression getVariable(String variableName)
    throws TranslationException { ... }
  public void setVariable(String variableName, Expression expr)
    { ... }
  public void removeVariable(String variableName) { ... }
};
```

Among implementation steps of a translator, writing abstract syntax trees by hand is a very large burden. To reduce the implementation cost, we built a compiler for translation rules. Using this compiler, a rule description which consists of patterns and ADL program templates is automatically compiled into Java implementation of a translator.

We used this compiler to implement translators in Appendix B. It dramatically simplified the implementation step.

3.3 Semantics-Preserving Property

In our framework, we decided to build verifiers for the common language introduced in the previous chapter, not directly for a low-level language. However what we really want to verify is programs of realistic low-level languages (such as the assembly language of an architecture). We have first to translate a target program into the common language and then verify the translated program.

Consequently, program translators take the most important role in our framework. If they do not *correctly* translate a program, verification will not be reliable; verification on the common language will be irrelevant with the properties of original program. Therefore, it is important and compulsory to check the correctness of a translator before we use it.

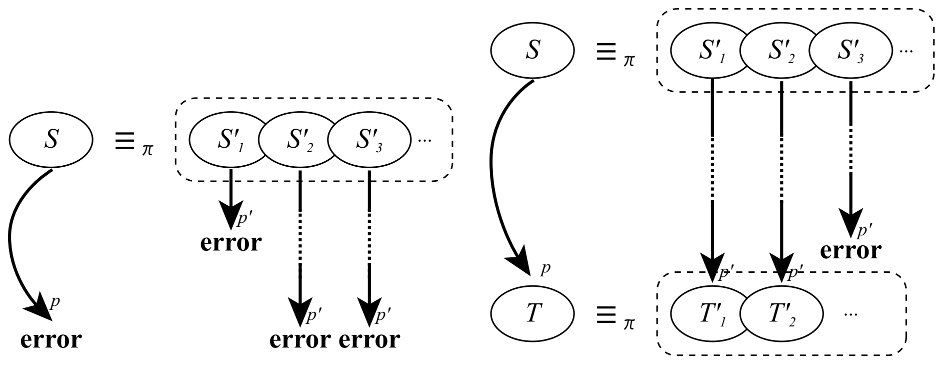
First we have to construct a mathematical model of programming languages. Here we assume the language is imperative, like C and assembly language. We model the interpreter for an imperative language as follows.

Definition 3.1. *We define an imperative language as* $(\mathcal{A}, \rightharpoonup_p)$*, where* $\rightharpoonup_p : \mathcal{A} \to \mathcal{A}^+$ *and* $\mathcal{A}^+ = \mathcal{A} \cup \mathbf{error}$*. It is a pair of the set of all states and step execution function with a program* p*. Here we assume the function* $\rightharpoonup_p$ *is total for any* p*.*

As step execution implements the execution rules, it is the mathematical model of an interpreter. The assumption of totalness is equivalent to the strong termination of step execution; in other words, each instruction in the language has to be strongly terminating. Meanwhile, the entire program do not need to terminate. In such cases, there is simply an infinite sequence of step execution without falling into **error**.

Next we discuss the correspondence between two programs in two different languages. To do this, we have to define the correspondence between states as a binary relation $\equiv_\pi \subseteq \mathcal{A} \times \mathcal{B}$ with a parameter (if needed). We write $S \equiv_\pi S'$ when the states S and S' correspond to each other under the additional information π.

Before arguing the concrete definition of this relation on certain language and ADL, we deliberate about general cases in this section. Now we define simulation relation for two programs $(\mathcal{A}, \rightharpoonup_p)$ and $(\mathcal{B}, \rightsquigarrow_{p'})$.



(a) Error is always simulated

(b) Correspondence is preserved unless an error occurs

Figure 3.5: Diagram for conservative simulation relation

Definition 3.2 (Conservative Simulation)**.** *We say a program p' conservatively simulates p, or $p' \rhd p$, iff. we can define the relation $\equiv_\pi$ which satisfies the following conditions: for all $S \in \mathcal{A}$ and $S' \in \mathcal{B}$ such that $S \equiv_\pi S'$,*

1. *If $S \rightharpoonup_p$ **error**, then $S' {\rightsquigarrow_{p'}}^* \textbf{error}$.*
2. *If $S \rightharpoonup_p T$ and $T \neq$ **error**, then there exists a sequence $S' \rightsquigarrow_{p'} T'_0 \rightsquigarrow_{p'} \cdots \rightsquigarrow_{p'} T'_n$ and the next two properties hold.*
   (a) $0 \leqslant {}^\forall i < n.\ T \not\equiv_\pi T'_i \wedge T'_i \neq \textbf{error}$.
   (b) Either $T'_n = \textbf{error}$ or $T'_n \neq \textbf{error} \wedge T \equiv_\pi T'_n$.

The concept of conservative simulation is illustrated as a diagram in Figure 3.5. Essentially it is a relation that the correspondence between states is preserved through program execution. Moreover, there is an implication that execution of p' evaluates to **error** from any state corresponding to S when p evaluates to **error** from S, thus we call the relation *conservative*.

Finally we can define the semantics-preserving property for a program translator using this relation.

Definition 3.3 (Semantics-Preserving Property)**.** *We say a program translator is semantics-preserving iff. for all program p, $p' \rhd p$ holds whenever the translator successfully translates p to p'.*

If we define step evaluation $\rightharpoonup_p$ as execution of one instruction, it usually terminates (including the cases where a runtime error occurs), and thus the relation satisfies

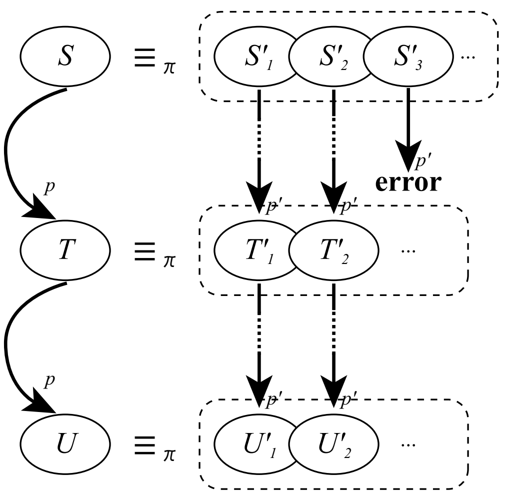


Figure 3.6: Simulation relation for composed programs

totalness. And by the next theorem, once the simulation relation holds for primitive instructions, it is preserved by program composition. Thus all we have to show for a translator is that a primitive instruction is correctly translated into another language as long as a translator can translate it.

Theorem 3.1. *Suppose two programs p and p' are given and $p' \rhd p$. For all $n \in \mathbb{N}$, $S \in \mathcal{A}$ and $S' \in \mathcal{B}$ such that $S \equiv_\pi S'$, if $S \rightharpoonup_p{}^n T$ then there exists $S' \rightsquigarrow_{p'}{}^* T'$ such that $T' = \mathbf{error} \vee S' \equiv_\pi T'$.*

Proof. By induction and the definition of conservative simulation relation. Figure 3.6 illustrates the concept. □

3.4 Mapping A Real Architecture to ADL

In this section, we describe how we can develop the correspondence between a real architecture and ADL. And then we discuss how we can confirm that the correspondence holds.

First we have to model an architecture from which we translate a program. From the observation of several architectures, we constructed a formal model of an architecture as shown in Figure 3.7. It is apparently similar to the definition of the abstract machine in ADL. It has a register file containing several registers, and memory. The difference from ADL is that it manipulates only byte values. Since only the definition of states is important, we do not discuss the execution rule here.

$$\begin{array}{lrcl}
\text{Byte Value} & b & \in & [0, 255] \quad :: \mathsf{Byte} \\
\text{Byte Array} & d & = & \langle b_0, \cdots, b_{n-1} \rangle \\
 & & & \\
\text{Register File} & R & :: & \{\mathtt{r_1} \mapsto d_1, \cdots, \mathtt{r_N} \mapsto d_N\} \\
\text{Memory} & M & :: & \mathsf{Int} \to \mathsf{Byte} \\
\text{Machine State} & S & = & (R, M, n) \in \mathcal{A}
\end{array}$$

Figure 3.7: Architecture model

Then we define the correspondence $\equiv_\pi$ between states of the architecture described above and the ADL abstract machine. In the subsequent discussion, we write the set of all ADL abstract machine states as $\mathcal{S}$, and the set of modeled architecture states as $\mathcal{A}$. The parameter for the relation is in the form $\pi = (\mathsf{layout}, \mathsf{encode}, \mathsf{codeptr})$; it is a triple of memory layout function $\mathsf{layout} :: \mathsf{Label} \to \mathsf{Int}$, value encode function defined in the previous chapter and code pointer mapping $\mathsf{codeptr} :: (\mathsf{Label} \times \mathsf{Int}) \to \mathsf{Int}$.

We start from defining the correspondence on a byte value in the source language (Byte) and an atomic value in ADL (Atom).

Definition 3.4. *We define* $b \equiv_{(\mathsf{layout},\mathsf{encode},\mathsf{codeptr})} a$ *iff.*

1. $a = \mathtt{junk}$*, or*
2. *When* $a = b'$ *(byte value),* $b = b'$*, or*
3. *When* $a = \ell + n[i]$ *(pointer),* $b = (\mathsf{encode}\ P\ ((\mathsf{layout}\ \ell) + n))[i]$

We can also define the correspondence between data (byte array) and memory using this definition.

Definition 3.5. *We say* $\langle b_0, \cdots, b_{n-1} \rangle \equiv_\pi \langle a_0, \cdots, a_{n-1} \rangle$ *iff.* $0 \leqslant {}^\forall i < n.\ b_i \equiv_\pi a_i$.

Definition 3.6. *We define memory correspondence* $M \equiv_\pi M'$ *iff.* ${}^\forall \ell \in \{\ell' \in \mathit{dom}(M') | M'(\ell') \textit{ is data}\}$. $0 \leqslant {}^\forall i < n.\ M((\mathsf{layout}\ \ell) + i) \equiv_\pi a_i$*, where* $M'(\ell) = \langle a_0, \cdots, a_{n-1} \rangle$.

And finally, we can define the correspondence relation between states.

Definition 3.7. *We define* $(R, M, n) \equiv_\pi (R', M', V', \ell + n')$ *iff.*

1. $\mathit{dom}(R) = \mathit{dom}(R')$ *and* ${}^\forall \mathtt{r} \in \mathit{dom}(R).\ R(\mathtt{r}) \equiv_\pi R'(\mathtt{r})$*, and*
2. $M \equiv_\pi M'$*, and*
3. $n = \mathsf{codeptr}\ (\ell, n')$ *where* $\pi = (\mathsf{layout}, \mathsf{encode}, \mathsf{codeptr})$.

Next we discuss the properties of the set of ADL states. We can define a partial order $\sqsubseteq$ on the set $\mathcal{S}$ as below.

Definition 3.8. *Let $d = \langle a_0, \cdots, a_{n-1} \rangle$ and $d' = \langle a'_0, \cdots, a'_{n-1} \rangle$. We say $d \sqsubseteq d'$ iff. $0 \leqslant {}^\forall i < n$.*

1. *$a'_i =$ `junk`, or*
2. *$a'_i = a_i$*

Definition 3.9. *Suppose two states $S, S' \in \mathcal{S}$ and let $S = (R, M, V, m)$ and $S' = (R', M', V', m')$. We say $S \sqsubseteq S'$ for two states $S, S' \in \mathcal{S}$, iff.*

1. *$m = m'$, and*
2. *$dom(R) = dom(R')$, ${}^\forall \mathtt{r} \in dom(R)$. $R(\mathtt{r}) \sqsubseteq R'(\mathtt{r}')$, and*
3. *$dom(M) = dom(M')$, ${}^\forall \ell \in dom(M)$. $M(\ell) \sqsubseteq M'(\ell)$*

Theorem 3.2. *$\sqsubseteq$ is a partial order.*

Proof. Reflexivity and transitivity are obvious. Antisymmetry can be proven through the case analysis. $\square$

This order compares the *abstractness* of two states. Integers and pointers are sort of concrete values, whereas `junk` corresponds to any atomic value, and so we call it an abstract value. It is apparent by the definition above that the number of `junk` included in S' is greater than or equal to that in S when $S \sqsubseteq S'$.

Here elements in $\mathcal{S}$ are classified by this partial order. We can classify them by the number of `junk` included, and we write the set of elements which contain i `junk`'s as $\mathcal{S}_i$. Then obviously $\mathcal{S}_i$ and $\mathcal{S}_j$ are disjoint for all $i \neq j$, and $\mathcal{S}$ is the sum of all classes.

$$\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_N$$

From these definitions, $\mathcal{A}$ is completely embedded into $\mathcal{S}_0$. We say a set B is completely embedded into another set A when there exists a total, surjective function from A to B.

Lemma 3.3. *Suppose a total, surjective function $f : A \to B$ can be defined on two sets A and B. Then A can be splitted into several equivalence classes, and when we write the set of these classes as $A/\sim$, we can define a bijection between $A/\sim$ and B.*

Proof. First we define the relation $a \sim a'$ for two elements $a, a' \in A$.

$$a \sim a' \Leftrightarrow f(a) = f(a')$$

This relation is obviously reflexive, transitive and symmetric. Thus it is an equivalence relation.

We then define a map $g : A/\sim \to B$ as follows. g is obviously surjective.

$$g([a]) = f(a)$$

By the properties of equivalence classes, $a \neq a' \Rightarrow [a] \cap [a'] = \emptyset$. If we assume $a \neq a'$ and $g([a]) = g'([a'])$, then $f(a) = f(a')$ holds from the definition of g. This is a contradiction, and thus $g([a]) \neq g'([a'])$ must hold. Therefore g is also injective, and this implies g is bijective. □

Theorem 3.4. *A total, surjective function $h_\pi : \mathcal{S}_0 \to \mathcal{A}$ can be constructed with a parameter π, by defining $h_\pi(S') = S$ where $S \equiv_\pi S'$.*

Proof. First we show that exactly one such element $S \in \mathcal{A}$ exists for all $S' \in \mathcal{S}_0$. Assume two states $S_1, S_2 \in \mathcal{A}$ satisfy $S_1 \equiv_\pi S'$ and $S_2 \equiv_\pi S'$. Since $S' \in \mathcal{S}_0$, S' does not contain `junk`. Thus from the definition of $\equiv_\pi$, a value contained in S must be fixed. Therefore $S_1 = S_2$ must hold, and h_π is definitely a total function.

Then we prove h_π is surjective by reduction to absurdity. Assume h_π is not surjective, thus there is a certain S such that $\nexists S' \in \mathcal{S}_0.\ h(S') = S$. However we can construct a state $S' \in \mathcal{S}$ corresponding to S by mapping atomic values to bytes with the same values. Obviously $S \equiv_\pi S'$ for any π, and this leads to contradiction. □

From now on, we refer to the function defined in Theorem 3.4 when we write h_π or simply h. π is abbreviated when it is apparent from the context. And also by Lemma 3.3 and Theorem 3.4, there exists an inverse function $h_\pi^{-1} : \mathcal{A} \to 2^{\mathcal{S}_0}$ such that h_π^{-1} is total and injective. We refer to this function when we write h_π^{-1} or h^{-1}.

The next lemmas are obvious from the definition of $\sqsubseteq$ and $\equiv_\pi$.

Lemma 3.5. *$S' \sqsubseteq S \Rightarrow h_\pi(S') \equiv_\pi S$, for all $S \in \mathcal{S}$, $S' \in \mathcal{S}_0$ and π.*

Lemma 3.6. *For all $S' \in \mathcal{A}$ and $S \in \mathcal{S}$, $S' \equiv_\pi S \Rightarrow {}^\exists S'' \in h_\pi^{-1}(S')$ s.t. $S'' \sqsubseteq S$.*

Now we go on to discuss how we can prove the correctness of translation. According to Theorem 3.1 in the previous section, we have only to prove correctness for each instruction patterns, because we define $\rightharpoonup$ as execution of one instruction in a real architecture. Consequently, discussing correctness just about programs containing only one instruction is sufficient.

Then we can define the termination of an execution for such programs. What we have to show becomes the preservation of correspondence $\equiv_\pi$ after termination. From now on we define $\rightsquigarrow^*$ as execution until the control flow reaches the end of a program.

Suppose two programs p and p' are given, where p is defined on an architecture A and p' on ADL. Then we define simulation relation only on $\mathcal{S}_0$ and $\mathcal{A}$.

Definition 3.10 (ADL Simulation). *For any memory map π and a state $S' \in \mathcal{S}_0$, let $h_\pi(S') \rightharpoonup_p T$ and $S' \rightsquigarrow_{p'}^* T'$. We write $p' \blacktriangleright p$, or p' conservatively simulates p, iff. the next two hold.*

1. *If $T =$ **error**, then $T' =$ **error**.*

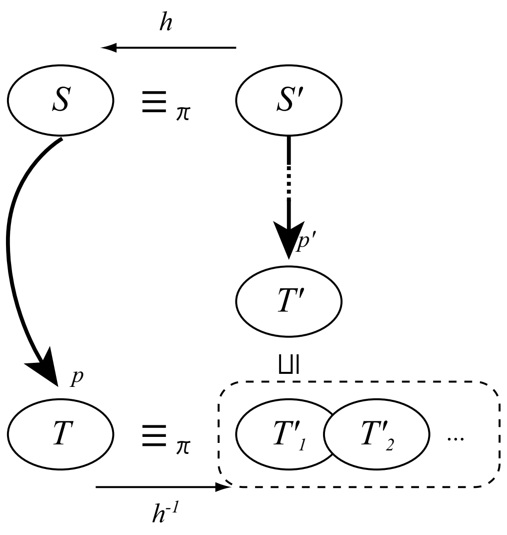


Figure 3.8: Conservative simulation diagram in ADL

2. *Otherwise,* $^{\forall}T'' \in h^{-1}(T).\ T'' \sqsubseteq T'$

The concept of this relation is shown as a diagram in Figure 3.8.

Definition 3.11 (Homomorphic program)**.** *We say a program in ADL is homomorphic (about the order* $\sqsubseteq$*), iff. the next two hold if* $S \sqsubseteq S'$*.*

1. *If* $S \rightsquigarrow$ **error***, then* $S' \rightsquigarrow$ **error***.*

2. *If* $S \rightsquigarrow T$ *and* $T \neq$ **error**, $T' \neq$ **error** $\Rightarrow T \sqsubseteq T'$ *where* $S' \rightsquigarrow T'$*.*

Definition 3.12. *We say* $v \sqsubseteq v'$ *for two values* v *and* v' *iff. either* $v' =$ `junk` *or* $v = v'$*.*

Theorem 3.7. *A program in ADL is homomorphic, if it utilizes conditional by kind command only in the following forms.*

1. `if` e : `junk` `then` `error` `else` c *or* `if` e : k `then` c `else` `error`*.*

2. `if` e : `junk` `then` $l =$ `junk` `else` c *or* `if` e : k `then` c `else` $l =$ `junk`*, where* $def[\![c]\!] \subseteq \{l\}$*. Here* l *is either a left value or a temporary variable.*

where $k \neq$ `junk`*.*

Proof. We show each command is homomorphic by case analysis. It is obvious from the semantics except for conditional by kind. Thus we assume the command is conditional by kind of the forms above.

Since $S \sqsubseteq S'$, $v \sqsubseteq v'$ where $S \vdash e \Downarrow v$ and $S' \vdash e \Downarrow v'$. If $v = v'$, the result state is equal, so we assume $v \neq v'$. Then from the definition of order, $v \neq$ `junk` $\wedge\ v' =$ `junk`.

$$\begin{array}{rcl}
def[\![\texttt{nop}]\!] & = & \varnothing \\
def[\![\texttt{error}]\!] & = & \varnothing \\
def[\![l = e]\!] & = & \{l\} \\
def[\![x = e]\!] & = & \{x\} \\
def[\![\texttt{goto}\, e]\!] & = & \varnothing \\
def[\![\texttt{if}\, e_b\, \texttt{then}\, c_1\, \texttt{else}\, c_2]\!] & = & def[\![c_1]\!] \cup def[\![c_2]\!] \\
def[\![\texttt{if}\, e : k\, \texttt{then}\, c_1\, \texttt{else}\, c_2]\!] & = & def[\![c_1]\!] \cup def[\![c_2]\!]
\end{array}$$

Figure 3.9: Possibly substituted values

Case 1: $\texttt{if}\, e : \texttt{junk}\, \texttt{then}\, \texttt{error}\, \texttt{else}\, c$ or $\texttt{if}\, e : k\, \texttt{then}\, c\, \texttt{else}\, \texttt{error}$.
In this case, $S' \rightsquigarrow$ **error**.

Case 2: $\texttt{if}\, e : \texttt{junk}\, \texttt{then}\, l = \texttt{junk}\, \texttt{else}\, c$ or $\texttt{if}\, e : k\, \texttt{then}\, c\, \texttt{else}\, l = \texttt{junk}$.
The storage referenced by l will contain `junk` after executing under the state S'. Then obviously $T \sqsubseteq T'$ by the definition of order.

□

Proposition 3.1. *If p' is homomorphic, $p' \blacktriangleright p \Rightarrow p' \rhd p$.*

Proof. We take an arbitrary $S \in \mathcal{S}$ and $S' \in \mathcal{A}$ such that $S' \equiv_\pi S$. Then we show (general) conservative simulation relation holds by these assumptions implies the condition for it.

First, because $\mathcal{A}$ is completely embedded into $\mathcal{S}_0$, there are functions $h : \mathcal{S}_0 \to \mathcal{A}$ and $h^{-1} : \mathcal{A} \to 2^{\mathcal{S}_0}$ by Lemma 3.3 and Theorem 3.4. And by Lemma 3.6, there exists $S'' \in h^{-1}(S')$ such that $S'' \sqsubseteq S$.

1. Error is always simulated.
We assume $S' \rightharpoonup_p$ **error**. Then from the definitions of ADL simulation and homomorphic property, $S'' \rightsquigarrow_{p'}{}^*$**error** and thus $S \rightsquigarrow_{p'}{}^*$**error**.

2. Correspondence is preserved unless an error occurs.
Here we assume $T' \neq$ **error** and $T \neq$ **error**. What we have to show is $T' \equiv_\pi T$ where $S' \rightharpoonup_p T'$ and $S \rightsquigarrow_{p'}{}^* T$.

From the definition of ADL simulation, $T'' \neq$ **error** and $U \sqsubseteq T''$ for all $U \in h^{-1}(T')$ where $S'' \rightsquigarrow_{p'}{}^* T''$. Since we assume p' is homomorphic, $T'' \sqsubseteq T$ holds. As $\sqsubseteq$ satisfies transitivity, ${}^\forall U \in h^{-1}(T')$. $U \sqsubseteq T$. By Lemma 3.5, it implies $T' \equiv_\pi T$. □

Here what we have to confirm for a program translator is as follows.

1. It generates homomorphic programs only.
2. It translates an instruction correctly, unless it fails.

Homomorphic property is easy to confirm, because we have only to check the use of conditional by kind. This is implied by Theorem 3.7. And also by Theorem 3.1, program composition preserves the correspondence. Thus we just have to show the correctness for each instruction pattern.

Unfortunately, the correctness is very difficult to prove automatically, because our target (the thing which defines the semantics of a low-level language) is often black-boxed as hardware, not a program. We may need to take so-called *brute force* approach to try every possible input state to a program. It is theoretically possible, as the domain of inputs is finite, but not practically feasible in many cases. However we will be able to get certain clues by an empirical approach to check several states for each instruction pattern.

In the area of compiler certification, many researches[6, 8, 9, 14, 17, 18] have already been done to prove automatically the correctness of program translation. Most of these researches discuss the correctness on certain logical framework, thus they implicitly require the description of the target language. However if we describe an architecture, there is also a need to show the correctness of description. Or inversely, even though we trust the specification, we yet have to confirm whether the underlying architecture (hardware) correctly implements the specification. Therefore, in spite of their contributions, the problem is deep-seated and still alive.

Chapter 4

Program Verifiers

This chapter discusses program verifiers. First we deliberate about theoretical topics of program verifiers and translators. We define the soundness and show that semantics-preserving program translators preserve this property. Then we define several interfaces for implementing a verifier. Finally we give the description of a simple verifier as an example, and show that it is sound.

4.1 Mathematical Properties of Verifiers

A verifier restricts the possible states for each point of execution by certain logic. Therefore we model it as an oracle.

Definition 4.1. *We define a verifier for ADL as an oracle* $V : \mathcal{S} \rightarrow \mathsf{Bool}$*, which takes an interpreter state and returns whether it is legitimate or not.*

Although it is the developer's responsibility to show certain properties for a verifier, here we define several important properties for program verifiers here.

Definition 4.2 (Progress)**.** *For all* S *such that* $V(S)$*,* $S \rightsquigarrow T$ *and* $T \neq \textbf{error}$*.*[1]

Definition 4.3 (Preservation)**.** *For all* S *such that* $V(S)$ *and* $S \rightsquigarrow T$ *where* $T \neq$ **error***, then* $V(T)$*.*

Definition 4.4 (Soundness of a Verifier)**.** *We say a verifier is sound iff. a program satisfies progress and preservation whenever verification succeeded for it.*

Soundness property means that the set $\{S \in \mathcal{S} | V(S)\}$ is closed about program execution relation $\rightsquigarrow$. When such a set cannot be constructed for a program, we say the program failed the verification.

[1]Notice that the ADL state S contains a program p inside. Thus V is parameterized implicitly by a program p.

If a verifier V is sound and a program translator used is semantics-preserving, then we can define a sound verifier V_π for the source architecture of translation.

Definition 4.5. *We construct a verifier $V_\pi : \mathcal{A} \rightarrow \mathsf{Bool}$ as follows.*

$$V_\pi(S) \quad \Leftrightarrow \quad {}^\exists S' \in \mathcal{S} \text{ such that } S \equiv_\pi S' \wedge V(S')$$

Proposition 4.1. *A verifier V_π is sound for any π, if a verifier V is sound and a program translator used is semantics-preserving.*

Proof. By the definition, there exists $S' \in \mathcal{S}$ such that $S \equiv_\pi S'$. And from the soundness, **error** is not reachable from the state S. Then there exists $T \in \mathcal{A}$ such that $S \rightharpoondown T$ and $T \neq$ **error**, since the program translator is semantics-preserving.

Also it implies $T \equiv_\pi T'$ where $S' \rightsquigarrow^* T'$. And from the soundness of the verifier V, T' satisfies $V(T')$, and therefore $V_\pi(T)$ holds. $\square$

4.2 Verifier Interfaces

In our framework, a verification logic is hidden, or *blackboxed*, from the framework, so that we can easily replace the logic one from another. We model a verification logic as a function that takes a program and additional information (if needed), and returns a verification result, either success or failure.

When a program is verified, many verification logics require certain information in addition to the program. For instance, a type checker requires type information for each component of the program. In ADL, this additional information is typically associated with each memory block.

Thus we designed the interface for verifiers and their factories as follows.

```
public interface Verifier {
  public MachineParameter getMachineParameter();

  public void verify(Program program, Signature signature)
    throws VerificationException;
};

public interface VerifierFactory {
  public Verifier createVerifier(MachineParameter machine);
};
```

`verify` method implements a verification logic. It is called from the framework with a translated program and additional information, which we call a verification signature. If the verification failed by certain reason, it throws `VerificationException`. This exception may contain information about the reason why the verification failed. Unless an exception is thrown, the verification succeeded for the program.

Because many verifiers also require information about the underlying architecture, `MachineParameter` has to be given to create a verifier. A factory takes `MachineParameter` as an argument to create a verifier object.

`Signature` class below describes a verification signature. It is a (partial) function from labels to annotations, each of which expresses information for one memory block. `Annotation` class is the base class for every annotation.

```
public class Signature {
  public Signature() { ... }

  public Annotation getAnnotation(String labelName)
    throws VerificationException { ... }
  public void addAnnotation(String labelName, Annotation annotation)
    throws VerificationException { ... }
};

// Base class for annotation
public class Annotation {};
```

4.3 An Example Verifier

In this section, we construct a very simple verifier as an example, and show that it satisfies the soundness property defined before.

4.3.1 Type-Level Abstract Interpreter

$$
\begin{array}{llcll}
\text{Data Types} & \tau_d & :: & \mathsf{DataType} & \\
 & \tau_d & ::= & \mathsf{junk} \mid \mathsf{byte} \mid \tau_d\ \mathsf{ptr} & \\
 & & \mid & \epsilon & \text{(Empty Type)} \\
 & & \mid & \tau_d \times \tau_d & \text{(Product)} \\
 & & \mid & \mathsf{code}(\Gamma) & \text{(Code Type)} \\
\text{Value Types} & \tau_v & :: & \mathsf{ValueType} & \\
 & \tau_v & ::= & \mathsf{junk} \mid \mathsf{int} \mid \tau_d\ \mathsf{ptr} & \\
 & & & & \\
\text{Type Context} & \Sigma & = & (\Phi, \Lambda, \Delta) = (\Gamma, \Lambda) & \\
 & \Gamma & = & (\Phi, \Delta) & \\
 & \Phi & = & \{\mathtt{r}_1 \mapsto \tau_d, \cdots, \mathtt{r}_\mathtt{N} \mapsto \tau_d\} & \text{(Register Types)} \\
 & \Lambda & = & \{\ell_1 \mapsto \tau_d, \cdots, \ell_{N'} \mapsto \tau_d\} & \text{(Memory Types)} \\
 & \Delta & = & \{x_1 \mapsto \tau_v, \cdots, x_{N''} \mapsto \tau_v\} & \text{(Variable Types)}
\end{array}
$$

Figure 4.1: Type-level abstract interpreter: types

First of all, we give a description of types in Figure 4.1. There are two levels of types: data types denoted as τ_d correspond to a data in a register and a memory block, and value types denoted as τ_v an ADL value.

A type context Γ in a code type $\mathsf{code}(\Gamma)$ is a precondition for the code associated with the type. It expresses constraints that the machine state must satisfy to execute the code. Type contexts are for describing types of register and temporary variables. Meanwhile, types of memory blocks are described by a verification signature and fixed.

Product type is used to express that two types are concatenated in the order. Associativity obviously holds for product types.

$$\tau_1 \times (\tau_2 \times \tau_3) = (\tau_1 \times \tau_2) \times \tau_3$$

Also it is obvious that an empty type ϵ is two-sided identity for product operator. Therefore product types form a monoid.

$$\begin{aligned} \tau \times \epsilon &= \tau \\ \epsilon \times \tau &= \tau \end{aligned}$$

To simplify the notation, we define *exponentiation* of a type as described below, which corresponds to a n-element array of type τ.

$$\tau^n = \overbrace{\tau \times \tau \times \cdots \times \tau}^{n \text{ times}}$$

We define type size function $\sigma :: \mathsf{DataType} \to \mathsf{Int}$ as follows. It is used to calculate the width of a data type in the subsequent formalization.

$$\begin{aligned} \sigma(\epsilon) &= 0 \\ \sigma(\mathsf{junk}) &= 1 \\ \sigma(\mathsf{byte}) &= 1 \\ \sigma(\tau \ \mathsf{ptr}) &= P \\ \sigma(\tau_1 \times \tau_2) &= \sigma(\tau_1) + \sigma(\tau_2) \end{aligned}$$

where P is an architecture-specific parameter which represents the size of pointers on the underlying architecture. Notice that we define the function only on data types, and code type does not appear in the equations above.

Now we go on to describe typing rules. We start from showing those for values and data as in Figure 4.2 and 4.3. A triple $\Lambda \vdash v : \tau_v$ means that a value v has type τ_v under the memory context Λ. Similarly a triple $\Lambda \vdash d : \tau_d$ for data d and type τ_d. Memory context is required because it may be referenced by a pointer value.

Next we define typing rules for expressions, boolean expressions and commands. They all look similar to the evaluation rules described in Section 2.3. Figure 4.4 and 4.5

$$\frac{}{\vdash \mathtt{junk} : \mathsf{junk}} \qquad \frac{}{\vdash n : \mathsf{int}}$$

$$\frac{\Lambda(\ell) = \tau_0 \times \tau \quad \sigma(\tau_0) = n}{\Lambda \vdash \ell + n : \tau\ \mathsf{ptr}} \qquad \frac{\Lambda(\ell) = \mathsf{code}(\Gamma)}{\Lambda \vdash \ell + 0 : \mathsf{code}(\Gamma)\ \mathsf{ptr}}$$

Figure 4.2: Typing rules for values

$$\frac{}{\vdash \langle\rangle : \epsilon} \qquad \frac{}{\vdash \langle \mathtt{junk} \rangle : \mathsf{junk}} \qquad \frac{}{\vdash \langle b \rangle : \mathsf{byte}}$$

$$\frac{\Lambda(\ell) = \mathsf{code}(\Gamma)}{\Lambda \vdash \langle \ell + 0[0], \cdots, \ell + 0[P-1] \rangle : \mathsf{code}(\Gamma)\ \mathsf{ptr}}$$

$$\frac{m = \ell + n \quad \Lambda(\ell) = \tau_0 \times \tau \quad \sigma(\tau_0) = n}{\Lambda \vdash \langle m[0], \cdots, m[P-1] \rangle : \tau\ \mathsf{ptr}} \qquad \frac{\Lambda \vdash d_1 : \tau_1 \quad \Lambda \vdash d_2 : \tau_2}{\Lambda \vdash d_1 \mathbin{+\!\!+} d_2 : \tau_1 \times \tau_2}$$

Figure 4.3: Typing rules for data

illustrate the typing rules for expressions and boolean expressions. Expressions that do not match these rules are considered invalid, and verification fails for a program containing such an expression.

Figure 4.6 defines the typing rules for commands. A triple $\Sigma \vdash c \Downarrow \Sigma'$ represents that when a command c is executed under the type context Σ, and if the control flow reaches the next command, then the type context becomes Σ'. There is a case where $\Sigma' = \bot$, and it means that the control flow never reaches there.

Two functions δ and ϵ defined in Figure 4.7 correspond to decode and encode functions. These functions abstract the value encode/decode phases which take place in evaluating expressions and commands.

A function υ below returns the least upper bound for two type contexts using the subtype relation defined next.

$$\upsilon(\Sigma_1, \Sigma_2) = \mathrm{lub}\,\{\Sigma_1, \Sigma_2\}$$

The subtype relation for type contexts is similar to that for record types. We define the rule as follows.

$$\frac{}{\{x_1 \mapsto \tau_1, \cdots, x_{n+m} \mapsto \tau_{n+m}\} <: \{x_1 \mapsto \tau_1, \cdots, x_n \mapsto \tau_n\}}$$

Here $<:$ is obviously a partial order; reflexivity, transitivity and antisymmetry are obvious from this rule. Therefore $(\mathfrak{S}, <:)$ forms a poset where $\mathfrak{S}$ is the set of type contexts. $\bot$ is the minimum element, i.e. $\bot <: \Sigma$ holds for all Σ, and no state is typed as $\bot$. Since $\mathfrak{S}$ also has the $<:$-maximal element $\top = (\{\}, \{\}, \{\})$, $(\mathfrak{S}, <:)$ is a lattice, and the least upper bound always exists.

$$\frac{\Lambda \vdash v : \tau}{(\Phi, \Lambda, \Delta) \vdash v : \tau}\ (\textsc{T-Val}) \qquad \frac{\Delta(x) = \tau}{(\Phi, \Lambda, \Delta) \vdash x : \tau}\ (\textsc{T-Var})$$

$$\frac{\Phi(\mathtt{r_i}) = \tau_0 \times \tau_1 \times \tau_2 \quad \sigma(\tau_0) = o \wedge \sigma(\tau_1) = s \quad \tau = \delta(\tau_1)}{(\Phi, \Lambda, \Delta) \vdash \mathtt{r_i}[o, s] : \tau}\ (\textsc{T-Reg})$$

$$\frac{\Sigma \vdash e : (\tau_1 \times \tau_2)\ \mathsf{ptr} \quad \sigma(\tau_1) = s \quad \tau = \delta(\tau_1)}{\Sigma \vdash *[s](e) : \tau}\ (\textsc{T-Mem})$$

$$\frac{\Sigma \vdash e : \mathsf{int}}{\Sigma \vdash \mathit{op}\, e : \mathsf{int}}\ (\textsc{T-UnArith}) \qquad \frac{\Sigma \vdash e_1 : \mathsf{int} \quad \Sigma \vdash e_2 : \mathsf{int}}{\Sigma \vdash e_1\ \mathit{op}\ e_2 : \mathsf{int}}\ (\textsc{T-BinArith})$$

$$\frac{\Sigma \vdash e : (\tau_1 \times \tau_2)\ \mathsf{ptr} \quad \sigma(\tau_1) = n}{\Sigma \vdash e + n : \tau_2\ \mathsf{ptr}}\ (\textsc{T-AddrAdd})$$

Figure 4.4: Typing rules for expressions

$$\frac{\Sigma \vdash e_1 : \mathsf{int} \quad \Sigma \vdash e_2 : \mathsf{int}}{\Sigma \vdash e_1\ \mathit{cmp}\ e_2 : \mathsf{bool}}\ (\textsc{T-Cmp})$$

$$\frac{\Sigma \vdash e : \mathsf{bool}}{\Sigma \vdash \mathit{lop}\, e : \mathsf{bool}}\ (\textsc{T-UnLog}) \qquad \frac{\Sigma \vdash e_1 : \mathsf{bool} \quad \Sigma \vdash e_2 : \mathsf{bool}}{\Sigma \vdash e_1\ \mathit{lop}\ e_2 : \mathsf{bool}}\ (\textsc{T-BinLog})$$

Figure 4.5: Typing rules for boolean expressions

4.3.2 Using Type-Level Abstract Interpreter as a Verification Logic

We can define a verification logic using the type-level abstract interpreter described in the previous subsection.

A typing rule for code blocks is shown in Figure 4.8. This rule indicates that there is a sequence of type contexts $\{\Sigma_0, \Sigma_1, \cdots, \Sigma_n\}$. Since typing rules of the type-level abstract interpreter are all deterministic, such a sequence is statically determined by giving the first type context and a command sequence. The last condition $\Sigma_1 = \bot \vee \Sigma_2 = \bot \vee \cdots \vee \Sigma_n = \bot$ means that the control flow will never fall off the end of this code block.

Figure 4.9 illustrates the typing judgement for the entire program. If this judgement is constructed without any inconsistency, we say a program is well-typed, or it passed the verification.

When a program is well-typed, a sequence $\{\Sigma_0, \Sigma_1, \cdots, \Sigma_n\}$ is assigned to each code block. We refer to this sequence by writing $\iota(M, \Lambda, \ell)$ in the subsequent discussion.

$$\frac{}{\Sigma \vdash \texttt{nop} \Downarrow \Sigma}\ (\text{T-Nop})$$

$$\frac{(\Phi, \Lambda, \Delta) \vdash e : \tau \quad \Delta' = (\Delta - \{x\}) \cup \{x \mapsto \tau\}}{(\Phi, \Lambda, \Delta) \vdash x = e \Downarrow (\Phi, \Lambda, \Delta')}\ (\text{T-AssnVar})$$

$$\frac{\begin{array}{c}(\Phi, \Lambda, \Delta) \vdash e : \tau \quad \Phi(\texttt{r}_\texttt{i}) = \tau_1 \times \tau_2 \times \tau_3 \\ \sigma(\tau_1) = o \wedge \sigma(\tau_2) = s \\ \Phi' = (\Phi - \{\texttt{r}_\texttt{i}\}) \cup \{x \mapsto \tau_1 \times \epsilon(\tau, s) \times \tau_3\}\end{array}}{(\Phi, \Lambda, \Delta) \vdash \texttt{r}_\texttt{i}[o, s] = e \Downarrow (\Phi', \Lambda, \Delta)}\ (\text{T-AssnReg})$$

$$\frac{\begin{array}{c}\Sigma \vdash e_1 : (\tau_1 \times \tau_2)\ \mathsf{ptr} \quad \Sigma \vdash e_2 : \tau \\ \sigma(\tau_1) = s \quad \epsilon(\tau, s) = \tau_1\end{array}}{\Sigma \vdash *[s](e_1) = e_2 \Downarrow \Sigma}\ (\text{T-AssnMem})$$

$$\frac{(\Gamma, \Lambda) \vdash e : \mathsf{code}(\Gamma')\ \mathsf{ptr} \quad \Gamma <: \Gamma'}{(\Gamma, \Lambda) \vdash \texttt{goto}\, e \Downarrow \bot}\ (\text{T-GoTo})$$

$$\frac{\begin{array}{c}\Sigma \vdash e : \mathsf{bool} \quad \Sigma \vdash c_1 \Downarrow \Sigma_1 \quad \Sigma \vdash c_2 \Downarrow \Sigma_2 \\ \Sigma' = \upsilon(\Sigma_1, \Sigma_2)\end{array}}{\Sigma \vdash \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow \Sigma'}\ (\text{T-IfThenElse})$$

$$\frac{\Sigma \vdash e : \tau \quad \vdash \tau : k \quad \Sigma \vdash c_1 \Downarrow \Sigma'}{\Sigma \vdash \texttt{if}\, e : k\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow \Sigma'}\ (\text{T-IfKindTrue})$$

$$\frac{\Sigma \vdash e : \tau \quad \not\vdash \tau : k \quad \Sigma \vdash c_2 \Downarrow \Sigma'}{\Sigma \vdash \texttt{if}\, e : k\, \texttt{then}\, c_1\, \texttt{else}\, c_2 \Downarrow \Sigma'}\ (\text{T-IfKindFalse})$$

$$\frac{}{\vdash \mathsf{junk} : \texttt{junk}} \qquad \frac{}{\vdash \mathsf{int} : \texttt{int}} \qquad \frac{}{\vdash \tau\ \mathsf{ptr} : \texttt{pointer}}$$

Figure 4.6: Typing rules for commands

4.3.3 Associating with the Value Interpreter

In order to show certain properties for a verification logic, we have to associate verification information with an (value-level) interpreter state.

Here we assume a program is well-typed ($\vdash M : \Lambda$, as defined in the previous subsection), and discuss the correspondence between type contexts and ADL abstract machine states.

Typing judgements for abstract machine states are shown in Figure 4.10. Since memory has already been proven to be well-typed on checking the program, we do not need to define the judgement here. Thus if registers and temporary variables are well-typed, we say the *state* is well-typed.

Type decode function δ :: $\mathsf{DataType} \rightarrow \mathsf{ValueType}$

$$\begin{aligned}\delta(\mathsf{byte}^n) &= \mathsf{int}\\ \delta(\tau\ \mathsf{ptr}) &= \tau\ \mathsf{ptr}\\ \delta(\_) &= \mathsf{junk}\end{aligned}$$

Type encode function ϵ :: $\mathsf{ValueType} \rightarrow \mathsf{DataType}$

$$\begin{aligned}\epsilon(\mathsf{int}, n) &= \mathsf{byte}^n\\ \epsilon(\tau\ \mathsf{ptr}, n) &= \begin{cases}\tau\ \mathsf{ptr} & n = P\\ \text{fail} & \text{otherwise}\end{cases}\\ \epsilon(\_, n) &= \mathsf{junk}^n\end{aligned}$$

Figure 4.7: Type encode/decode functions

$$\frac{\begin{array}{c}\Sigma_0 = (\Gamma, \Lambda)\\ \Sigma_0 \vdash c_0 \Downarrow \Sigma_1\\ \Sigma_1 \neq \bot \Rightarrow \Sigma_1 \vdash c_1 \Downarrow \Sigma_2\\ (\Sigma_1 \neq \bot \wedge \Sigma_2 \neq \bot) \Rightarrow \Sigma_2 \vdash c_2 \Downarrow \Sigma_3\\ \vdots\\ \Sigma_1 = \bot \vee \Sigma_2 = \bot \vee \cdots \vee \Sigma_n = \bot\end{array}}{\Lambda \vdash \langle c_0; c_1; \cdots; c_{n-1}\rangle : \mathsf{code}(\Gamma)}$$

Figure 4.8: Typine rule for code blocks

$$\frac{dom(M) \supseteq dom(\Lambda) \quad {}^{\forall}\ell \in dom(\Lambda).\ \Lambda \vdash M(\ell) : \Lambda(\ell)}{\vdash M : \Lambda}$$

Figure 4.9: Typing judgement for the entire program

$$\frac{dom(R) \supseteq dom(\Phi) \quad {}^{\forall}\mathtt{r} \in dom(\Phi).\ \Lambda \vdash R(\mathtt{r}) : \Phi(\mathtt{r})}{\Lambda \vdash R : \Phi}$$

$$\frac{dom(V) \supseteq dom(\Delta) \quad {}^{\forall}x \in dom(\Delta).\ \Lambda \vdash V(x) : \Delta(x)}{\Lambda \vdash V : \Delta}$$

$$\frac{\Lambda \vdash R : \Phi \quad \Lambda \vdash V : \Delta \quad \vdash M : \Lambda}{\vdash (R, M, V, m) : (\Phi, \Lambda, \Delta)}$$

Figure 4.10: Judgements for abstract machine states

4.3.4 Proof of Soundness

In this subsection, we show the soundness for the type system that we have designed hitherto.

First we have to say the correctness of type functions. These properties are obvious by their definitions.

Lemma 4.1 (Type Size Function)**.** *If* $\Lambda \vdash d : \tau$*, then* $\sigma(\tau) = |d|$.

Lemma 4.2 (Type Decode Function)**.** *If* $\Lambda \vdash d : \tau$*, then* $\Lambda \vdash \mathsf{decode}\ d : \delta(\tau)$.

Lemma 4.3 (Type Encode Function)**.** *If* $\Lambda \vdash v : \tau$*, then* ${}^\forall n.\ \Lambda \vdash \mathsf{encode}\ n\ v : \epsilon(\tau, n)$.

The next lemma is the inverse of the typing rule of data concatenation.

Lemma 4.4 (Data Splitting)**.** *If* $\Lambda \vdash d_1 +\!\!+ d_2 : \tau_1 \times \tau_2$ *and* $\sigma(\tau_1) = |d_1| \wedge \sigma(\tau_2) = |d_2|$, *then* $\Lambda \vdash d_1 : \tau_1$ *and* $\Lambda \vdash d_2 : \tau_2$ *hold.*

Proof. Obvious by case analysis for the construction of type τ_1. □

Then we prove several lemmas which guarantee the abstract interpretation corresponds to the (value-level) interpretation. We need a lemma for each interpretation unit, i.e. language element.

Lemma 4.5. *If* $\Gamma, \Lambda \vdash e : \tau$ *and* $\vdash C : \Gamma, \Lambda$*, then* $v \neq$ **error** *and* $\Lambda \vdash v : \tau$ *where* $C \vdash e \Downarrow v$.

Proof. By induction over the typing tree of an expression. Let $\Gamma = (\Phi, \Delta)$ and $C = (R, M, V)$.

Case 1: $e = v$

Trivial.

Case 2: $e = x$

From the definitions, $v = V(x)$ and $\tau = \Delta(x)$. Then $\Lambda \vdash v : \tau$ by well-typedness.

Case 3: $e = \mathtt{r_i}[o, s]$

From the definitions, $\tau = \delta(\tau_1)$ and $v = \mathsf{decode}\ \langle a_o, \cdots, a_{o+s-1}\rangle$ where $\Phi(\mathtt{r_i}) = \tau_0 \times \tau_1 \times \tau_2$, $R(\mathtt{r_i}) = \langle a_0, \cdots, a_{n-1}\rangle$ and $\sigma(\tau_0) = o \wedge \sigma(\tau_1) = s$. By Lemma 4.1, $\sigma(\Phi(\mathtt{r_i})) = \sigma(\tau_0) + \sigma(\tau_1) + \sigma(\tau_2) = n$. Since the type size function returns natural number only, obviously $o = \sigma(\tau_1) \geqslant 0$, $s = \sigma(\tau_2) \geqslant 0$ and $\sigma(\tau_3) \geqslant 0$. And $s > 0$ by induction hypothesis, which implies $0 \leqslant o < n \wedge 0 < s \leqslant n - o$.

By Lemma 4.4, $\Lambda \vdash \langle a_o, \cdots, a_{o+s-1}\rangle : \tau_1$ holds. Therefore $\Lambda \vdash v : \tau$ by Lemma 4.2

Case 4: $e = *[s](e')$

By induction hypothesis, $C \vdash e' \Downarrow \ell + o$ and $\Lambda \vdash \ell + o : (\tau_1 \times \tau_2)$ ptr where $\sigma(\tau_1) = s$. Then by the typing rules, $\Lambda \vdash \langle a_o, \cdots, a_{o+s-1}, a_{o+s}, \cdots, \rangle : \tau_1 \times \tau_2$ where $M(\ell) = \langle a_0, \cdots \rangle$. By Lemma 4.4, $\Lambda \vdash \langle a_o, \cdots, a_{o+s-1} \rangle : \tau_1$ holds. Therefore $\Lambda \vdash v : \tau$ where $v =$ decode $\langle a_o, \cdots, a_{o+s-1} \rangle$ and $\tau = \delta(\tau_1)$ by Lemma 4.2.

Case 5: $e = op\, e'$

From the definitions, $v = op\, v'$ and $\tau =$ int where $C \vdash e' \Downarrow v'$. By induction hypothesis, $\Lambda \vdash v'$: int, thus the arithmetic operation returns an integer. Obviously $\Lambda \vdash v : \tau$.

Case 6: $e = e_1\ op\ e_2$

(a) When T-BinArith rule is used, $v = v_1\ op\ v_2$ and $\tau =$ int where $C \vdash e_1 \Downarrow v_1$ and $C \vdash e_2 \Downarrow v_2$. By induction hypothesis, both v_1 and v_2 are integers, and the result is also an integer. Therefore $\vdash v$: int.

(b) When T-AddrAdd rule is used, $C \vdash e_1 \Downarrow \ell + o$, $e_2 = n$, $\Lambda \vdash \ell + o : (\tau_1 \times \tau_2)$ ptr and $\sigma(\tau_1) = n$ by induction hypothesis. From the typing rule, $\Lambda \vdash \langle a_o, \cdots \rangle : \tau_1 \times \tau_2$ where $M(\ell) = \langle a_0, \cdots, \rangle$. By Lemma 4.4, $\Lambda \vdash \langle a_{o+n}, \cdots \rangle : \tau_2$. Therefore $\Lambda \vdash v = \ell + (o + n) : \tau = \tau_2$ ptr.

□

Lemma 4.6. *A well-typed boolean expression will never fail an assertion.*

Proof. By induction over the typing tree of an expression.

Case 1: If an expression is comparison, both of two operands are integer by Lemma 4.5. Then comparison is always possible.

Case 2: If an expression is logical expression, obvious by induction hypothesis.

□

The next lemma discusses the correspondence for command execution.

Lemma 4.7. *Assume* $\vdash (R, M, V) : \Sigma$, $\Sigma \vdash c \Downarrow \Sigma'$. *If* $(R, M, V, m) \vdash c \Downarrow (R', M', V', m)$, *then* $\Sigma' \neq \bot$ *and* $\vdash (R', M', V') : \Sigma'$.

Proof. By induction over the typing tree of c.

Case 1: $c =$ nop

From the definitions, $(R', M', V') = (R, M, V)$ and $\Sigma' = \Sigma$.

Case 2: $c = \texttt{error}$

This command is not typed.

Case 3: $c = x = e$

From the assumption, an expression e is well-typed. Thus by Lemma 4.5, $(R, M, V) \vdash e \Downarrow v$ and $\Lambda \vdash v : \tau$. From the definitions, $V' = (V - \{x\}) \cup \{x \mapsto v\}$ and $\Delta' = (\Delta - \{x\}) \cup \{x \mapsto \tau\}$. Then obviously $\Lambda \vdash V' : \Delta'$, which implies $\vdash (R', M', V') : \Sigma'$.

Case 4: $c = \texttt{r}_\texttt{i}[o, s] = e$

Similar to the argument on referencing a register in an expression. See the proof of Lemma 4.5.

Case 5: $c = *[s](e') = e$

Similar to the argument on referencing a memory block in an expression. See the proof of Lemma 4.5.

Case 6: $c = \texttt{goto}\, e$

This command modifies the instruction pointer, and therefore we do not need to discuss here.

Case 7: $c = \texttt{if}\, e\, \texttt{then}\, c_1\, \texttt{else}\, c_2$

From the assumption, a boolean expression e is well-typed, thus by Lemma 4.6, e will never evaluate to **error**. By induction hypothesis, $\Sigma \vdash c_1 \Downarrow \Sigma_1$ and $\Sigma \vdash c_1 \Downarrow \Sigma_2$.

Since $\Sigma' = \upsilon(\Sigma_1, \Sigma_2)$ and υ computes the least upper bound, $\Sigma_1 <: \Sigma'$ and $\Sigma_2 <: \Sigma'$. (R', M', V') is either the state after c_1 or c_2, it obviously satisfies $\vdash (R', M', V') : \Sigma'$.

Case 8: $c = \texttt{if}\, e : k\, \texttt{then}\, c_1\, \texttt{else}\, c_2$

From the assumption, an expression e is well-typed. By Lemma 4.5, $(R, M, V) \vdash e \Downarrow v$ and $\Lambda \vdash v : \tau$.

(a) Assume $\vdash \tau : k$. In this case c_1 is executed, and by induction hypothesis $\vdash (R', M', V') : \Sigma'$.

(b) Otherwise, c_2 is executed. The rest is similar to the argument in the first case.

□

Finally we can prove soundness by using these lemmas.

Theorem 4.8 (Progress). *Assume $\vdash M : \Lambda$, $\vdash S : \Sigma_i$ where $S = (R, M, V, \ell + i)$ and $\{\Sigma_0, \Sigma_1, \cdots\} = \iota(M, \Lambda, \ell)$. Then $S \rightsquigarrow T$ and $T \neq$* **error**.

Proof. Let $c = M(\ell)[i]$. We show that the evaluation $S \vdash c \Downarrow T$ will never return $T = \mathbf{error}$ by induction over the typing tree of a command c.

Case 1: `nop`
Trivial.

Case 2: `error`
From the well-typedness, this case is impossible.

Case 3: $x = e$
From the well-typedness, $x \in dom(\Delta) \subseteq dom(V)$ where $\Sigma_i = (\Phi, \Lambda, \Delta)$. By Lemma 4.5, e never evaluates to **error**. Thus the execution does not fail.

Case 4: $\mathtt{r_i}[o, s] = e$
Similar to the argument on referencing a register in an expression.

Case 5: $*[s](e') = e$
Similar to the argument on referencing a memory block in an expression.

Case 6: $\mathtt{goto}\, e$
By the assumption, $\Sigma \vdash e : \mathsf{code}(\Gamma')\ \mathtt{ptr}$. And by Lemma 4.5, $(R, M, V) \vdash e \Downarrow \ell + n$. Since address calculation is impossible for code pointers, n must be 0. From the well-typedness, a block identified by ℓ must contain program code. Then all the assertion is satisfied.

Case 7: $\mathtt{if}\, e\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2$
By Lemma 4.6, e evaluates to either true or false. And by the induction hypothesis, $T_1 \neq \mathbf{error}$ and $T_2 \neq \mathbf{error}$ where $S \vdash c_1 \Downarrow T_1$ and $S \vdash c_2 \Downarrow T_2$. Therefore $T \neq \mathbf{error}$.

Case 8: $\mathtt{if}\, e : k\, \mathtt{then}\, c_1\, \mathtt{else}\, c_2$
By Lemma 4.5, $v \neq \mathbf{error}$ and $\Lambda \vdash v : \tau$ where $(R, M, V) \vdash e \Downarrow v$ and $\Sigma \vdash e : \tau$.

(a) If $\vdash \tau : k$, c_1 is executed. By the induction hypothesis, $S \vdash c_1 \Downarrow T$ and $T \neq \mathbf{error}$.

(b) Otherwise, c_2 is executed. By the induction hypothesis, $S \vdash c_2 \Downarrow T$ and $T \neq \mathbf{error}$.

□

Theorem 4.9 (Preservation). *Assume* $\vdash M : \Lambda$, $\vdash S : \Sigma_i$ *where* $S = (R, M, V, \ell + i)$ *and* $\{\Sigma_0, \Sigma_1, \cdots\} = \iota(M, \Lambda, \ell)$. *If* $S \rightsquigarrow T$ *and* $T \neq \mathbf{error}$, *then either:*

1. $T = (R', M', V', \ell' + 0)$ *and* $\vdash T : (\Gamma', \Lambda)$ *where* $\Lambda(\ell') = \mathsf{code}(\Gamma')$.

2. $T = (R', M', V', \ell + (i+1))$*,* $\Sigma_{i+1} \neq \bot$ *and* $\vdash T : \Sigma_{i+1}$*.*

Proof. By case analysis of the command executed.

Case 1: Assume the control flow jumps to another memory block. In ADL, at most one side effect occurs in an command, thus $(R', M', V') = (R, M, V)$. From the well-typedness, $\vdash (R, M, V) : (\Gamma, \Lambda)$ and $\Gamma <: \Gamma'$ where $\Lambda(\ell') = \mathsf{code}(\Gamma')$.

Case 2: Otherwise, the control flow falls to the next instruction whose address is $\ell + (i+1)$. From the definition, $S \vdash c \Downarrow T$ where $c = M(\ell)[i]$. Then obviously by Lemma 4.7, $\Sigma_{i+1} \neq \bot$ and $\vdash T : \Sigma_{i+1}$.

$\square$

Proposition 4.2. *The type system described in this section is sound.*

Proof. Obvious by Theorem 4.8 and 4.9. We just define $V(S)$ where $S = (R, M, V, \ell+i)$ like this:

$$V(S) \quad \Leftrightarrow \quad \vdash M : \Lambda \wedge \vdash S : \iota(M, \Lambda, \ell)[i]$$

$\square$

Chapter 5

Related Work

The approach to split the implementation of a verifier using a common language is also taken in Foundational TAL by Crary[5]. Inspired by Appel's Foundational PCC[2], he built a type system for an assembly-like language TALT, and described it in a logical framework Twelf. Then he proved by a theorem prover that the type system satisfies several properties including GC safety.

His primary concern was to reduce trusted computing base (TCB), and to realize this he constructed a system to verify the program safety on a logical framework. However the correspondence between the real assembly languages or machine languages and TALT has not yet been discussed in his paper.

Many researches have also been done on proving that the semantics is preserved in a program translation, especially in the field of compiler optimizations.

Lerner et al.[12] discussed the correctness of compiler optimizations with a language named Cobalt. Cobalt is a domain-specific language designed for implementing optimizations as guarded rewrite rules. They presented a strategy for automatically proving the soundness of optimizations and analyses using Cobalt. Necula[17] built a translation validation infrastructure (TVI) for the GNU C compiler. He designed an intermediate language IL similar to the language that the GNU C compiler uses, and constructed a framework to check two IL programs have the same semantics.

Rinard[20] tried to build a credible compiler. It produces a machine-checkable proof which demonstrates the correctness of compilation, in addition to compiled code. The user does not need to trust the compiler, just verifies the proof and knows that the compilation is correctly performed.

However it is difficult to extract the semantics for a low-level language, because it is often implemented as hardware. These researches assume that the semantics of the language is given on certain logical framework. Even if we describe the semantics, it is still required to show the correctness of the description (or alternatively, the correctness of the *hardware*). Thus our work is also aimed to develop a formal foundation for verifying the correctness of a program translator when the semantics is blackboxed.

CPU emulators, such as Bochs[19], PearPC[7] and QEmu[3], are similar to our research in that both they and our framework translate low-level programs to certain common language. They are using C or C++ as a common language to achieve architecture-independence[1]. Their correctness is discussed only informally, by the observation that many realistic applications seem to run correctly.

We would be able to modify their code to produce C- or C++-code from a program on a target architecture, though, we think neither C nor C++ is appropriate as a common language which we build a verifier for. This is mainly because the semantics of these languages are too complicated, and the semantics is not completely fixed. Additionally, we can cast an integer to a pointer anytime, and we can also perform unfettered pointer arithmetics. Some of these operations are bogus, and to avoid such bogus operations, we restricted the pointer arithmetics in ADL, and we think it is reasonable. We think ADL is a small but enough expressive, safe subset of C, and it is more approproate than C to build a verifier for.

[1]Actually the semantics is a bit different among many platforms. One example is the size of `int`. Also, the semantics of C has many unspecified operations.

Chapter 6

Conclusion

We presented a framework using a common language to build program verifiers for low-level languages. It is aimed to construct a common basis to help developers of a verifier for low-level languages, such as assembly language and machine language. In particular, we focused on the design of a common language and program translators.

First we designed the common language ADL, presented the syntax and semantics and implemented an interpreter to confirm the correctness of translation. The user of this framework can build a verification logic or a program translator using the defined semantics.

Then we formalized program translators and discussed their properties. We showed what properties must be proven and how we can confirm the correctness of a translator. For proof of concept, we implemented translators from the subset of Intel x86 and SPARC assembly languages.

Finally we modeled verification logics as blackboxed oracles, and showed the soundness is preserved through program translation using semantics-preserving program translators. We defined the soundness property required for a verifier. As an example, we constructed a very simple type system and showed that it satisfies the soundness.

Recent researches in the area of program verification and compiler certification have shown considerable progress. However there still seems to be a large gap between the target language of verifiers and the language in which most of the realistic programs are written. Our work is, therefore, aimed to fill this gap.

We believe that our framework helps the developers to build code verifiers for low-level languages.

Chapter 7

Future Work

In this paper, we checked a partial correctness of implemented translators by our hand, using the ADL interpreter and the GNU gdb debugger on the real architectures. This required a large amount of work, and thus we feel that we will need to construct a testing framework. As an ADL interpreter is already implemented as a part of the framework, and there are several multi-architectural CPU emulators like QEmu[3], we would be able to build a system that automatically performs tests.

Or alternatively, we may as well trust the specification of assembly languages and that the hardware correctly implements the specification. If so, we would be able to apply the result of translation validation researches here.

Currently ADL does not support dynamic allocation of memory. Thus we have to statically prepare every memory block used in a program with enough sizes beforehand. This limitation may weaken the expressiveness of the system, because there is a possibility that the size of a memory block actually allocated in a real architecture may not be equal to the modeled program. Or we can handle this by implementing a heap manager in ADL, however it spoils the *separatedness* between memory blocks.

In order to implement dynamic memory allocation, we have to incorporate a special instruction like `malloc` (as in typed assembly languages[16, 15, 5]), or more generally we have to extend an ADL interpreter so that it can call external native subroutines. Both extensions will be straightforward, however at the same time we will need to describe mathematical models of every external handler to do that.

Similarly in current implementation of ADL, the stack is handled just as other memory blocks. We may be able to allocate a large memory area of fixed length also for the stack, and use it from one side. If it grows too deep, the stack pointer falls off the stack area at certain time and overflow can be detected. Although this behavior is popular among many realistic platforms, the stack should virtually be an infinite structure, which is automatically extended and shrunk when needed, for building a theoretical model.

Also $\mathsf{L}^3\mathsf{Cover}$ framework does not consider concurrent, multi-threaded programs at this time. As interference between two or more concurrent threads is the most important feature in these programs, we will have to model this behavior correctly.

Threads in concurrent programs communicate each other using shared memory areas. As the timing is critical to the way reading from and writing to these memory areas are serialized, we must simulate randomized behavior here. If we naively try to enumerate all the possibilities, the number of states increases according to the exponential order. Therefore a substantial effort is required to treat such programs[4, 22].

Appendix A

Implementation of an Interpreter

A.1 Program Containers

First we have to define classes to represent a program in ADL. We implemented one class for each language element as shown in Figure A.1.

The complete program (code and data) is stored in `Program` class which corresponds to memory. As a program is also stored in memory in this model, it is designed to be a container for memory blocks. It contains a map from labels (`java.lang.String`) to memory blocks (`MemoryData`).

We used `java.math.BigInteger` class for representing integers. We can construct an instance of `java.math.BigInteger` class from 2's complement representation of an integer, which is popular among many architectures. In this formalization we just have to concern the byte ordering. And it is encapsulated into `Int` class.

Memory blocks and registers are distinguished by names. We simply used `java.lang.String` class for this purpose.

A.2 Interpreter Classes

There are several architecture-specific parameters to an interpreter.

- Names and sizes of registers
- Size of pointers
- encode and decode functions which convert values and data

Class name	Corresponding language element
`Int`	Integer n
`Pointer`	Fat pointer m
`Atom`	Atomic Value a
`Atom[]`	Data d
`Value`	Values v
`LeftValue`	Left Values l
`Expression`	Expression e_v
`BooleanExpression`	Boolean Expression e_b
`Command`	Command c
`Command[]`	Command Sequence c*
`MemoryData`	Memory Data k
`Program`	Memory M

Figure A.1: Classes for representing an ADL program

We designed `MachineParameter` and `ValueCodec` interfaces to encapsulate these information.

`ValueCodec` interface implements `encode` and `decode` functions, and `MachineParameter` interface contains the rest of above. `MachineParameter` also has a method to return `ValueCodec`, thus it is the complete object to represent all architecture-specific parameters.

```
public interface ValueCodec {
  public Atom[] encode(Value value, int size);
  public Value  decode(Atom[] data);
};

public interface MachineParameter {
  public String[] getRegisterNames();
  public int getRegisterSize(String registerName);

  public int getPointerSize();

  public ValueCodec getValueCodec();
};
```

Given a `MachineParameter`, we can construct a concrete interpreter class. Thus we employed factory pattern here.

```
public interface ExecutionEngine {
  public void initialize(Program program,
                         InstructionPointer programCounter)
    throws ExecutionException;

  public InstructionPointer getProgramCounter();
  public void setProgramCounter(InstructionPointer programCounter);
  public ExecutionContext getContext();

  public ExecutionEngine step() throws ExecutionException;
};

public interface ExecutionEngineFactory {
  public ExecutionEngine createExecutionEngine(
    MachineParameter parameter
  )
    throws ClassCastException;
};
```

`step` method in `ExecutionEngine` class implements step execution $S \rightsquigarrow S'$, where S is represented by an instance of `ExecutionContext` interface and the instruction pointer. When $S' = \textbf{error}$, it throws an `ExecutionException`. If an exception is thrown, we cannot continue execution any more, and the results for the subsequent calls to interpreter methods are unspecified.

The current execution context can be obtained through calling `getContext` method. It returns an instance of `ExecutionContext` interface. This class contains three kinds of storages: registers, memory and temporary variables.

```
public interface ExecutionContext {
  public Register getRegister(String registerName)
    throws ExecutionException;

  public MemoryBlock getMemoryBlock(String labelName)
    throws ExecutionException;

  public Value getVariable(String variableName)
    throws ExecutionException;
  public void defineVariable(String variableName, Value value)
    throws ExecutionException;
  public void forgetVariable(String variableName)
    throws ExecutionException;
};
```

`ExecutionContext` is created in the call to `initialize` method of `ExecutionEngine` class. Since it may contain implementation-specific information in addition to the states described above, it is defined as an interface. Concrete classes for `ExecutionEngine` and `ExecutionContext` must correspond to each other.

Instruction pointer is encapsulated into an `InstructionPointer` class. It is a pair of label and instruction index, just like `Pointer` class.

```
public class InstructionPointer {
  public InstructionPointer(String labelName, int index) { ... }
  public InstructionPointer(String labelName) { this(labelName, 0); }

  public String getLabelName() { ... }
  public int getIndex() { ... }
};
```

A.3 Usage of the Interpreter

To use the interpreter described above, first obtain an instance of `ExecutionEngineFactory` interface. Then call `createExecutionEngine` method with an instance of `MachineParameter` which contains information about the architecture to be simulated. This method returns an instance of `ExecutionEngine` interface, which is the core of the interpreter.

After that, call `initialize` with a `Program` class to be executed and `InstructionPointer` class which contains the first instruction pointer. Here the execution engine class is initialized and we can start the interpretation. Calling `step` method advances execution by one instruction.

We can easily inspect the machine state by calling `getContext` method and `getInstructionPointer` method between calls to `step` method.

Thus, top level code will look like the following.

```
// prepare required information
ExecutionEngineFactory eef  = ...;
MachineParameter       mp   = ...;
Program                prog = ...;

// initialize the engine
// and start execution from a block labeled "_start"
ExecutionEngine interp = eef.createExecutionEngine(mp);
interp.initialize(prog, new InstructionPointer("_start"));

try {
  while(true) {
    // do some inspection here
    // ...

    // and step one instruction
    interp.step();
  }
} catch(ExecutionException exn) {
  // an error occurred
  ...
}
```

Appendix B

Implementation of Program Translators

B.1 Program Translator for x86

Intel x86 architecture[11] is the most popular architecture in the personal computer market. x86 is one of CISC architectures, on which complex instructions and addressing modes are implemented.

B.1.1 Modeling x86 Architecture

x86 architecture, or also known as IA-32 architecture, has 8 general purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP. They are 32-bit each and we can freely use them as operands to instructions. Also x86 has 8 floating-point registers, but as a prototype implementation, we do not handle floating-point instructions here.

In addition to these registers, x86 has a flag register EFLAGS. It consists of many flags, but typical programs do not need most of them. Here we just focus on ZF (zero flag, set when the result is zero), CF (carry flag), OF (overflow flag) and SF (sign flag, set when the result is negative). We prepare one-byte-wide register for each of these flags.

In x86, address is actually 48-bit long; 32 bits come from the computation result and 16 bits from one of segment registers. Essentially a segment register is used to explicitly separate code, data and the stack. However they are not separated well; in many OSes, flat memory model is employed. Therefore we just ignore segment registers in implementing a translator.

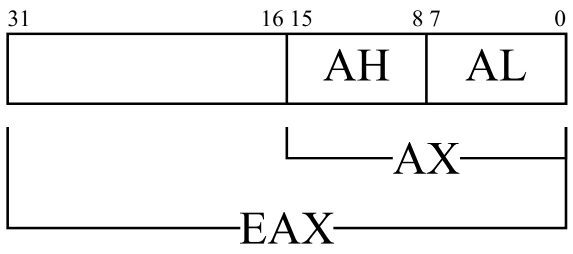


Figure B.1: A register in x86 architecture

B.1.2 Addressing Modes

One feature of x86 is the abundance of addressing modes. Addressing modes are patterns of operands. An immediate value, a register or even reference to a memory cell can be described in the architecture.

For registers, we can specify a part of a register instead of referencing the entire values contained in a register (see Figure B.1 for detailed information). As x86 is a little-endian architecture, LSB comes to the first of byte array, and MSB to the last. Thus AX is 2-byte data starting from the index 0 of EAX register. Similarly AL and AH can be expressed as a byte data at the index 0 and 1, respectively. For example, we can express this feature as follows with the syntax of ADL.

```
#operand {
    eax = %eax;
    ax  = %eax[0, 2];
    al  = %eax[0, 1];
    ah  = %eax[1, 1];
...
}
```

In x86, memory references are not restricted to use with a special load/store instruction. We have to include them into the operand mapping.

```
#operand {
    mem(Base, Size) =
        *[Size](Base);
    mem(Base, Offset, Size) =
        *[Size](Base + Offset);
    mem(Base, Offset, Index, Scale, Size) =
        *[Size](Base + Offset + Index * Scale);
}
```

B.1.3 Instructions

`mov` instruction

This instruction copies the content of one operand to another. As it does not affect any flags, we have only to write assignment.

```
mov(D, S) {
    D = S;
}
```

`add` instruction

This instruction works like `+=` operator in C; it adds the content of the second operand to the first. And simultaneously it modifies flag registers appropriately.

```
add(D, S)
    // calculate results
    $s      = (1 << sizeof(D) * 8) - 1; // all 1's with the operand size
    $_cftmp = (D & $s) + (S & $s);
    $_oftmp = (D ^ S) & (($s >> 1) + 1);
    D = D + S;

    // ZF
    if D : int then
      if D == 0 then %_zf = 1 else %_zf = 0
    else if (D - &null) : int then // = if comparable to &null
      if (D - &null) & $s == 0 then %_zf = 1 else %_zf = 0
    else
      %_zf = junk;
    // SF
    if D : int then
      if D < 0 then %_sf = 1 else %_sf = 0
    else if (D - &null) : int then // = if comparable to &null
      if (D - &null) & (($s >> 1) + 1) != 0 then %_sf = 1 else %_sf = 0
    else
      %_sf = junk;
    // CF
    // - set if carry to the 32nd bit as unsigned values
    if $_cftmp : int then
      if $_cftmp >= ($s + 1) then %_cf = 1 else %_cf = 0
    else
      %_cf = junk;
    // OF
    // - set if signed result overflow
    //   = original two input but output has the same sign
    if $_oftmp : int then /* both D and S has int */
      if $_oftmp == 0 && ((D ^ S) & (($s >> 1) + 1)) != 0 then
        %_of = 1
      else
        %_of = 0
    else
      %_of = junk;
}
```

Computation rules for flags are conservatively implemented. When a pointer is given as an operand, flags become `junk` except the cases where `null` appears. `null` is the only label whose pointer is statically determined, and we assume its value is 0.

Since two pointers with different labels are not comparable, we have to use conditional by kind when comparing pointers. As shown in Section 2.3, subtracting a pointer from another returns an integer if they are of the same label, otherwise `junk` is returned. This operation can be done without generating a runtime error, conditional by kind of this result is appropriate for comparing pointers.

`jmp`, `j`*cc* instructions

`jmp` instruction is unconditional jump. It simply corresponds to `goto` command.

```
jmp(OP) {
    goto OP;
}
```

Conditional jumps looks flag registers. For example, `jz` instruction below branches when the result of the antemortem arithmetic operation is equal to 0.

```
jz(OP) {
    if %_zf : int then
      if %_zf != 0 then goto OP else nop
    else
      error; // undecidable !!
}
```

As we have shown in `add` instruction, there is a possibility that flag registers contain `junk`. In such cases, we cannot decide which branch to take; selecting one branch statically may lead to the discordance on simulation relation. Therefore we have to give up simulation and generate a runtime error.

`push`, `pop` instructions

These instructions are used to operate on the stack. `push` inserts data to the top of the stack, and `pop` retrieves data from the stack.

In x86, stack pointer is always ESP register. This register always points the current top of the stack. When we push a data to the stack, first the stack pointer is subtracted by the size of the data, and then the data is stored.

```
push(D, Size) {
    %esp = %esp - Size;
    *[Size](%esp) = D;
}

pop(D) {
    D = *[sizeof(D)](%esp);
    %esp = %esp + sizeof(D);
}
```

Size is needed for `push` because it may take an immediate value. Meanwhile, `pop` does not require operand size.

`call` instruction

This instruction is a good example which looks ahead of the input sequence.

`call` instruction is used to make a function call. It first pushes the address after the instruction to the stack (return address), and then jumps to the specified address. And when we return from a function, return address is popped from the stack. Thus we have to know the instruction position (i.e. a label) which comes directly after the instruction.

We assume each `call` instruction has a label adjacent to it. If there is no label after a `call` instruction, a translator throws an exception to indicate that the program translation failed.

```
call(OP) {
    // Obtain the address of the next instruction (i.e. where to return)
    <% Event nev = input.peek(0);
       if(nev.getType() != Event.TYPE_LABEL) throw new TranslationException();
       Expression _retaddr =
         new Expression(Value.Pointer(new Pointer(nev.getLabelName()))); %>

    // Push it to the stack
    %esp = %esp - 4;
    *[4](%esp) = <%= _retaddr %>;
    // And jump to the function
    goto OP;
}

ret() {
    $retaddr = *[4](%esp);
    %esp = %esp + 4;
    goto $retaddr;
}
```

Blocks enclosed in `<% ... %>` and `<%= ... %>` are Java code, like those in Java Server Pages (JSP).

B.2 Program Translator for SPARC

SPARC architecture[21] is one of RISC architectures. In this section, we introduce how we can write translator rules for several architecture-specific features.

B.2.1 Handling a Zero Register

Zero register is a register whose value is always zero. SPARC also has `g0` register as a zero register, like many RISC architectures. It always reads the value 0, and writing to it is discarded.

We do not need a storage for the register `g0`. In operand mapping, we just return the constant zero.

```
#operand {
    g0 = 0;
    g1 = %g1;
...
}
```

Writing rule is described in the instruction mapping as follows.

```
#instruction {
    mov(S, g0) { }
    mov(S, D)  { D = S; }

    add(S1, S2, g0) { }
    add(S1, S2, D)  { ... }
...
}
```

We leverage template priority to implement the zero register. Because translators simply generate the code corresponding to the first pattern that matched to the given term, the rule written anterior has a priority.

B.2.2 Handling Delayed Branch

Delayed branch is to delay completion of branching by several instructions. It is popular among RISC architectures, and often incorporated for pipeline efficiency. For example in SPARC, branch is always delayed for one instruction. It means that an instruction after a branch instruction is always executed before branch is completed.

A simple solution is to transpose a branch instruction and delayed slot. However this is not the complete solution; it cannot handle the cases where the instruction in the delayed slot modifies the operand to the branch instruction. Thus first we have to save the branch target to somewhere. Temporary variables are appropriate for this purpose.

The rule below is for `ba` instruction (branch always).

```
ba(OP) {
    // first determine where to go
    $__next = OP;

    // execute delayed slot
    <% Event dslot = input.peek(0); input.pop();
       if(dslot.getType() != Event.TYPE_INSTRUCTION)
         throw new TranslationException();
       if(isBranch(dslot.getInstruction()))
         throw new TranslationException();
       translateInstruction(output, input, dslot.getInstruction()); %>

    // and branch
    goto $__next;
}
```

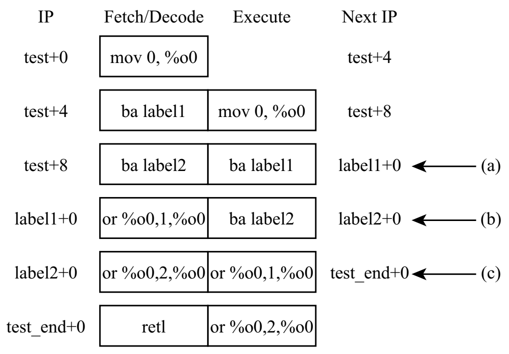


Figure B.2: Processor behavior for complex delayed branch

In this case we need to use Java code inside the command template. In order to achieve this, we take JSP-like solution to escape from ADL program.

We assume a branch instruction never comes to the delayed slot in the rule above. This is because a branch instruction in the delayed slot leads to difficulty. The assembly code below is an example of such a complex branch.

```
        .global test
test:
        mov     0, %o0
        ba      label1
        ba      label2
        or      %o0, 4, %o0
        ba      test_end
        nop
label1:
        or      %o0, 1, %o0
        ba      test_end
        nop
label2:
        or      %o0, 2, %o0

test_end:
        retl
        nop
```

Figure B.2 shows the processor behavior for this program.

First branch for `label1` is taken, but the next instruction, branch to `label2`, is still fetched because of delayed branch ((a) in the figure). Then the execution goes to the next instruction, and hereat an instruction is fetched from `label1` ((b) in the figure). Since the instruction currently executed is also a branch, the next instruction pointer becomes `label2`.

After that, the first instruction of the block named `label1` is executed. The instruction next executed is fetched from `label2` at this moment ((c) in the figure). And finally the register `o0` contains the value 3.

The behavior like this is difficult to model. Especially when the branch target is dynamically determined (for example, it is passed using a function pointer), it is hard to simulate using statically generated code. However, in most cases delayed slot is filled using a non-branch instruction. Thus we decided to reject such programs.

References

[1] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C.* American National Standards Institute, 1430 Broadway, New York, NT 10018, USA, 1999.

[2] Andrew W. Appel. Foundational Proof-Carrying Code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pp. 247–256. IEEE Computer Society Press, June 2001.

[3] Fabrice Bellard. QEmu. `http://fabrice.bellard.free.fr/qemu/`.

[4] A. T. Chamillard. Improving Static Analysis Accuracy on Concurrent Ada Progra: Complexity Results and Empirical Findings. Technical Report UM-CS-1995-049, Amherst, MA, USA, 1995.

[5] Karl Crary. Toward a Foundational Typed Assembly Language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 198–212, New York, NY, USA, 2003. ACM Press.

[6] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, Vol. 28, No. 6, pp. 2–2, 2003.

[7] Sebastian Biallas et al. PearPC: PowerPC Architecture Emulator. `http://pearpc.sourceforge.net/`.

[8] Susan L. Gerhart. Correctness-preserving program transformations. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 54–66, New York, NY, USA, 1975. ACM Press.

[9] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his etirement from his professorship at the University of Kiel)*, pp. 201–230, London, UK, 1999. Springer-Verlag.

[10] IBM Corporation. *PowerPC Microprocessor User's Manual.*

[11] Intel Corporation. *IA-32 Intel®Architecture Software Developer's Manual.*

[12] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 220–231, New York, NY, USA, 2003. ACM Press.

[13] Xavier Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, Vol. 30, No. 3-4, pp. 235–269, 2003.

[14] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 42–54, New York, NY, USA, 2006. ACM Press.

[15] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35, May 1999.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3, pp. 528–569, May 1999.

[17] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 83–94, New York, NY, USA, 2000. ACM Press.

[18] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. *Lecture Notes in Computer Science*, Vol. 1384, pp. 151–166, 1998.

[19] The Bochs Project. Bochs: The Open Source IA-32 Emulation Project. `http://bochs.sourceforge.net/`.

[20] Martin Rinard. Credible Compilation. Technical Report MIT/LCS/TR-776, Massachusetts Institute of Technology, 1999.

[21] Sun Microsystems, Inc. *UltraSPARC Processors Documentation.*

[22] Richard N. Taylor. Complexity of Analyzing the Synchronization Structure of Concurrent Programs. *Acta Informatica*, Vol. 19(1), pp. 57–84, April 1983.

[23] Frank Yellin. Low Level Security in Java. `http://java.sun.com/sfaq/verifier.html`.