A Declarative Approach to Generating Machine Code Tools

Ben L. Titzer, Jonathan K. Lee, and Jens Palsberg

UCLA Computer Science Department
University of California, Los Angeles, USA
{titzer,jkenl,palsberg}@ucla.edu

Abstract. Tools that analyze, manipulate, and simulate machine code across many instruction architectures are increasingly important, especially in domains such as sensor networks, where multiple sensor architectures with different processors are rapidly being explored. Sensor network applications increasingly require on-the-fly updates, which can entail inserting new machine code that may overwrite running code. Dynamic code update presents a major challenge for sensor network simulators that provide a high degree of visibility of the execution through nonintrusive precision instrumentation. In this paper we present the design of Isildur, a declarative instruction set description language, and a compiler that generates machine code tools from Isildur specifications. We can easily integrate generated tools such as interpreters and disassemblers into a sensor network simulator that handles dynamic code update.

1 Introduction

Background. Machine code tools such as simulators, disassemblers, and analyzers are fundamental in many phases of software engineering, including development, testing, and tuning. Applications, operating systems, and compilers all benefit from the availability of quality machine code tools. In particular, simulators are of special importance since they allow behavioral tests and detailed performance measurements for new applications and operating systems, especially on architectures where the hardware is difficult or impossible to instrument. With instrumentation capabilities [1–11], simulators can allow measurements of complex dynamic execution characteristics without being intrusive into applications' source or binary code. However, developing new machine code tools often involves dealing with low-level details of instruction encodings and execution behavior; such code is tedious to write and difficult to verify. In this paper we present a declarative approach to specifying *what* an instruction set is and *what* generated tools should do, without specifying *how* generated tools should be implemented.

Our approach is motivated by our need to develop flexible, instrumentable, and fast simulators for sensor network hardware architectures. The sensor network domain is marked by the rapid exploration of many competing sensor architectures with different processors. For example, three of the most popular

platforms, the Mica2, the Telos, and the Stargate use three different processor architectures (AVR, MSP430, and ARM, respectively). In previous work, we developed the Avrora [12] simulator, which simulates the Mica2 sensor node platform and provides numerous instrumentation capabilities [13]. Importantly, Avrora can simulate sensor networks which use dynamic code update to re-task nodes [14]. Internally, Avrora uses pre-decoded instructions for performance and instrumentation, but the overwriting of existing code may invalidate any pre-decoded instructions that are being used internally in the simulator. Avrora therefore requires a disassembler that translates binary machine code into the internal representation dynamically as the program executes new code. Construction of such a system requires significant engineering effort, which has led us to explore tools and languages for automatically generating significant portions of the simulator and generating the disassembler entirely.

Our Results. We have designed Isildur, a declarative instruction set description language, and a compiler that generates machine code tools from Isildur specifications. We can easily integrate generated tools such as interpreters and disassemblers into a sensor network simulator that handles dynamic code update. An Isildur specification has three parts that describe the structure, behavior, and encoding of instructions, which allow a programmer to describe an instruction set architecture at a higher level than previous languages, separating the low-level implementation details from the conceptual model. So far have written instruction set descriptions for two RISC architectures, AVR and MIPS, and one CISC architecture, MSP430.

Isildur has a static type system and a consistency checker that enable us to find bit-level bugs early, including problems with arithmetic overflows and sign and zero extensions. The type system exposes the bit-width of all expressions and introduces types and operators that simplify expressing bit-level operations such as accessing and updating individual bits and bit ranges of registers. This allows static checking for common programming errors relating to bit manipulations and guarantees that the generated simulator code computes the intended results, regardless of the bit-width of the machine on which the simulator runs. The consistency checker verifies that the encoding specifications satisfy three key requirements that we call operand encoding completeness, word alignment, and nonambiguity. In the appendix we present Isildur's syntax, operational semantics, and type system, along with a proof of type soundness. We have automatically verified the type soundness result for a core subset of Isildur using the Coq proof assistant (http://coq.inria.fr).

Isildur's approach to specifying the encodings of instructions is superior to other description languages that force the user to specify the bit fields that encode the instruction and operands and the order in which fields are matched. In Isildur the programmer need only specify encoding patterns, and the disassembler generator automatically checks for consistency and generates an efficient algorithm for disassembly. CISC architectures that have numerous addressing modes for each instruction operand have classically presented a significant challenge to automatic matching techniques; our system includes a novel decoder

reduction algorithm that allows for the most general case but reduces redundancy to yield a space and time efficient matching algorithm automatically. The generated disassembler hides all the encoding details of an instructions behind an interface that accepts a sequence of bytes or words and produces a single instruction in the intermediate format. This simplicity allows the disassembler to be integrated into a flexible simulator in a straightforward way in order to support updatable code.

The tools generated from Isildur specifications are already being used in deployment. In addition to our own work on programming languages and compilers for sensor networks, the Avrora simulator, which relies on Isildur tools, has been the vehicle for the sensor network experiments reported in published papers from University of Tuebingen, Germany [15], University of Utah [16], and UCLA [14]. At UCLA, Mani Srivastava [17] has used Avrora as the basis for class projects in multiple instances of his two graduate courses on Embedded and Real-Time Systems and on Mobile and Wireless Networked Computing Systems. Most of Srivastava's class projects use SOS [14] which the students run with Avrora, taking advantage of Avrora's ability to handle updatable code.

Running Example. Figure 1 introduces an Isildur specification that will serve as a running example throughout the rest of the paper. The example specifies an instruction set with three instructions, two addressing modes, sixteen 16-bit registers, and 64K memory. Lines 2-5 declare an enumerated type representing the names of the registers. Lines 6-8 declare global state that is manipulated in executing instructions. Operand types, which represent parameters to instructions, are declared in lines 9-17; in particular, the GPR operand type represents a register operand to an instruction, while the other types simply specify integer immediates and addresses. Addressing modes are declared in 18-23, and represent a combination of operands and their encodings that can be reused by multiple instructions. An addressing mode set is declared in line 24, which allows an instruction to implement multiple addressing modes (that is, have polymorphic operand types). Three instructions are declared in lines 25-43; the familiar three-address "add" instruction, a load immediate instruction, and a call instruction that accepts an absolute address as the operand.

Related Work. An Isildur encoding specification serves a purpose similar Ramsey and Fernandez's notion of SLED specification [18]. From a SLED specification one can automatically generate the core parts of assemblers, disassemblers, and retargeting tools. Ramsey and Fernandez [19] also presented a testing-based approach to checking that a SLED specification is consistent with an existing assembler. A major advantage of Isildur compared to SLED is that the compiler performs several static consistency checks that help find bugs in instruction encodings, which does not rely on the availability of an external assembler. We expect that our disassembler generator will be helpful in building retargeting tools and binary translators [20, 21].

An Isildur behavioral specification is written in a typed language, similarly to Ramsey and Davidson's CSDL language, which is written in the typed language

λ-RTL [22]. Both type systems can express the bit width of values, but Isildur's type system goes further by calculating more precise bounds on the number of bits needed to represent a given value. Our type system also supports a wider range of complex data types, but in contrast to the λ-RTL type system, our compiler does not support type inference.

Another simulator, SimpleScalar [9], also has a fast implementation allowing dynamic code updates and provides a way for users to specify their own architectures. The specification is implemented through C preprocessor macros which can be messy and error-prone. There is no notion of operands and addressing modes, and thus instructions with multiple addressing modes must be implemented individually. Finally, the decoding specification is obscured by the complexity of the macro system and the manual disambiguation of encodings by placing them into groups. SimpleScalar may not be able to handle complicated encodings efficiently.

Rest of the Paper. Section 2 gives details of the Isildur language and Section 3 discusses how to generate machine code tools from Isildur specifications.

2 Isildur: A Typed Instruction Set Description Language

Structural Specification. The structural specification of an instruction set declares the names of instructions, the types of operands that they accept, and the addressing modes that each implements. The structural specification serves as the backbone of generated machine code tools, which are often most interested in the structure of the instructions and their operands, and not details such as the exact execution behavior of each instruction, or the exact bit pattern to which the instruction corresponds in machine code. Isildur provides a suite of specification primitives for declaring these entities that serve as the skeleton around which the other parts of the specification are centered.

Isildur allows the programmer to declare new enumeration types that are useful for defining entities such as a register file. The example declares (in lines 2–5) a type REG that gives symbolic names to the 16 general purpose registers.

An operand type declaration introduces a new type for operands to instructions. For example, an operand type may represent a register, an immediate, a code address, or a memory address. In the example, lines 9, 10, 11 and 14 declare operand types. Simple operands may represent an integer, a boolean, an address, or an enumeration, for direct use in a behavioral specification, while complex operand types (not shown) might represent a register-plus-index operand.

Instructions are declared in Isildur (lines 25, 32, and 38) with a unique name and a list of named operands with their types. In the example, the programmer declares three instructions; for example, the `ldi` instruction takes a destination register and an immediate constant. Within the instruction declaration, the programmer can include a behavioral specification and an encoding specification (see below).

Many instructions may accept the same combination of operands. For example, in RISC architectures arithmetic operations often use two input registers

```
01: architecture u16 {
02:     enum REG {
03:         r0=0, r1=1, r2=2, r3=3, r4=4, r5=5, r6=6, r7=7, r8=8,
04:         r9=9, r10=10, r11=11, r12=12, r13=13, r14=14, r15=15
05:     }
06:     global pc: +int.16;
07:     global registers: map<REG, +int.16>;
08:     global ram: map<+int.16, +int.16>;
09:     operand-type IMMB[8]: +int.8;
10:     operand-type ADDR[14]: +int.14;
11:     operand-type IMMW[16]: +int.16 {
12:         read: +int.16 { return this: +int.16; }
13:     }
14:     operand-type GPR[4]: REG {
15:         read: +int.16 { return registers[this]; }
16:         write: +int.16 { registers[this] = value; }
17:     }
18:     addr-mode RR_IMM dest: GPR, ra: GPR, rb: IMMW {
19:         encoding = { opcode[2:0], 0b1, dest, ra, 0b0000, rb }
20:     }
21:     addr-mode RRR dest: GPR, ra: GPR, rb: GPR {
22:         encoding = { opcode[2:0], 0b0, dest, ra, rb }
23:     }
24:     addr-set OP3 { RR_IMM, RRR }
25:     instruction "add": OP3 {
26:         property opcode: +int.3 = 0b010;
27:         execute {
28:             local result: +int.17 = read(ra) + read(rb);
29:             write(dest, result: +int.16);
30:         }
31:     }
32:     instruction "ldi" dest: GPR, imm: IMMB {
33:         encoding = { 0b1111, dest, imm } // 16 bits total
34:         execute {
35:             write(dest, imm: +int.8);
36:         }
37:     }
38:     instruction "call" addr: ADDR {
39:         encoding = { 0b00, addr } // 16 bits total
40:         execute {
41:             pc[15:2] = addr;
42:         }
43:     }
44: }
```

Fig. 1. An Isildur specification

and a single output register. To reduce redundancy in instruction declarations, an addressing mode can be declared that defines a particular combination of operands and an encoding specification. Thus, addressing modes serve as kind of classification or templating mechanism for instructions. This example uses addressing modes declared on lines 18 and 21 to define addressing modes corresponding to three operand addressing and register and immediate addressing.

Addressing modes also serve another important purpose. For CISC architectures where an instruction may implement multiple addressing modes, Isildur provides a construct called an *addressing mode set* (line 24) that represents a set of addressing modes. This construct unifies a collection of addressing modes such that each operand in the addressing set is a union of all the operands of the same name from each addressing mode. An instruction that implements the addressing mode set therefore implements all of the declared addressing modes; the operands to that instruction are therefore polymorphic.

The structural specification primitives illustrate the declarative nature of Isildur: rather than describing the implementation of the data structures that implement the instruction set, as one would in a general purpose programming language, the programmer declares only the structure. The Isildur compiler can then generate a Java implementation of that architecture in which each element of the structural specification corresponds to a Java class.

Behavioral Specification and Type System. A behavioral specification defines the semantics of an instruction. These are written in the execute blocks of the instruction declaration (lines 27-30, 34-36, and 40-42). The operands defined in the addressing mode of an instruction are used in the scope of the execute body to access the desired fields to avoid direct access to the instruction word and to generalize the instruction instead of specifying different implementations for seperate addressing modes.

Writing a behavioral specification of an instruction set often entails working with bit sized quantities, which is normally performed through masks and shifts. Depending on the usage, bits may be interpreted as signed or unsigned numbers or indexes into register files, memories, etc. Arithmetic operations may cause overflows which will require a larger bit representation. Language constructs such as bit indexing (not shown), which specifies an individual bit, and bit ranges (line 41), which statically specifies a block of bits, allow individual bits to be specified and manipulated instead of mask and shifts that obscure the important bits.

Isildur has a static type system to aid in identifying bit level errors in complex bit manipulations. Integer types retain sign and bit width information which gives the programmer fine grained control. Other languages like Java already have this information implicitly built into their primitive types, but they are very coarse in their granularity and may require many manipulations to transform the bits to have the same intended value. Our type system makes truncation explicit so that bits are not implicitly lost in assignments. We perform checks on assignment to make ensure that the destination is of a certain bit width to ensure that no bit information is lost due to truncations. Operations such

as addition and multiplication yield an increased bit width to represent the increased number of bits required to adequately represent that value. On line 28, the read operations both yield 16 bit integers which when added may require a 17th bit. The variable result is declared as 17 bits to hold the resulting sum. For bit range operations, we ensure that the desired bits exists in the target value. However, bit indexing operations are similar to array accesses and we cannot statically determine whether the index is in bounds and thus may cause a runtime error which will be handled external to the specification.

Maps (lines 7 and 8) are associative arrays that are implemented by external code. This is often used in the case of a register file or memory. On line 8, `ram` is a 16 bit address space which could be implemented as a flat array or a multi-level array. Maps types are parameterized by their key and value types and our type system checks that the usage of the map is consistent. In a similar vein, external subroutines (not shown) may be declared to provide functionality that may be better expressed in the target implementation.

Encoding Specification. An encoding specification describes how instructions and their operands are represented in binary format. Computers represent machine code in a compact binary encoding that represents the instruction, its addressing mode, and its operands as patterns of bits. Machine code tools such as assemblers and disassemblers must deal directly with these bit patterns, and are traditionally written with tedious bit masks and shifts that are notoriously difficult to get right. Isildur's declarative approach to this problem removes all need for such tools to deal directly with bit patterns. Instead, the programmer simply annotates the structural specification of instructions with an encoding specification that is used by the Isildur compiler to automatically generate efficient tools that retrieve the instructions and operands from bit patterns.

Unlike previous languages for describing instruction sets, Isildur does not force the programmer to differentiate between opcode fields nor specify the order in which fields of the bit patterns are to be matched. Instead, the programmer specifies only encoding patterns and the compiler automatically derives an efficient disambiguation routine.

Each simple operand type (lines 9, 10, 11, 14) declares the number of bits required to encode it (e.g. X[n]: T indicates the operand type X requires n bits and is of base type T). For integer immediate operands and addresses (lines 9, 10, 11) the encoding is implicitly defined as the two's complement binary representation of the integer value. For operand types whose base type is an enum (line 14), the encoding of the operand into bits is given by the values declared in the enum type declaration.

For instructions and addressing modes, the user can specify an encoding (lines 19, 22, 33, 39) that consists of a list of bit fields. Each bit field in the list has a size and can be an expression such as a binary constant, an operand, or a variable. The individual bits of an operand or variable may not necessarily be contiguous; they can be split up and spread throughout the encoding, which occurs in some architectures such as the AVR [23].

The bits in the encoding list are arranged into a bit pattern using the left-to-right logical bit ordering. Concerns such as word size and endianness of the machine are handled separately; all tools that operate on bit patterns can assume a logical bit ordering for patterns. This simple specification of a list of bit patterns is in contrast to other systems which require the programmer to specify special opcode fields and the order in which opcode fields must be compared by the disassembler.

3 Generating Machine Code Tools

The Isildur compiler generates Java source code from a given description file. The tools generated include a disassembler, an interpreter, and a collection of Java classes that implement a intermediate representation of instructions. The Avrora framework provides a number of simulator building blocks such as instrumentable memories, binary loaders, and device I/O registers. These building blocks, along with the generated tools, can be used by the programmer to put together a fast and instrumentable simulator quickly. One author, without any prior experience with the Avrora framework, was able to completely integrate a MIPS interpreter generated by the Isildur compiler into Avrora in just a few days, complete with instrumentation capability and binary program loading via the ELF format.

3.1 Automatic Interpreter Generation

The core of any instruction-level simulator is the interpreter which executes instructions and maintains the state of the machine. Often an interpreter must manipulate many bit-level quantities with tricky masks and shifts. Isildur's behavioral specification provides a richer language that includes bit and bit range operators, simplifying such tedious code and reducing opportunities to make mistakes.

After type checking and consistency checking, the Isildur compiler translates the behavioral specification (that is, the instruction execute blocks) into source code in the implementation language such as Java. Most general purpose languages' type systems are less rich than those provided by Isildur, which complicates code generation. For example, Java provides the coarse-grained primitive types of `byte`, `short`, `int`, and `long`. Further, Java has implicit promotion of subexpressions to `int` and `long`. In general, when compiling an Isildur specification to a target language or machine, we would like to preserve the same semantics regardless of the bit-width of the machine.

A straightforward code generator that ignores the bitwidth types of expressions and promotes expressions to the largest integer type available may be both inefficient and may not always compute the same results on all machines. For example, consider a signed 32-bit integer and a bit-range assignment that updates the sign bit. If the implementation machine is 32 bits, then the update will

give the correct value without explicit sign extension, but on a 64-bit machine, the update of the sign bit requires a sign extension to give the same value.

The Isildur compiler addresses this problem by using the types of subexpressions to drive code generation. A normal code generator can be thought of as a function of type $L_1 \rightarrow L_2$ that converts expressions in language L_1 into expressions in language L_2. The Isildur compiler's code generator can be thought of as a function of type $L_1 \times \text{Type} \times \text{int} \rightarrow L_2$ which accepts, in addition to an L_1 expression, an expected type and an expected shift position, which it uses to avoid unnecessary masks, shifts, and sign extensions that would otherwise be generated by a naïve code generator. For complex bit range updates, the code generator produces Java code that would match what would be written by hand. This optimization therefore allows efficient code generation while allowing programmers to specify their intent with a declarative programming construct.

3.2 Automatic Disassembler Generation

Disassemblers are an important component in a number of machine code manipulation tools like simulators and debuggers. In writing such tools, it is inconvenient and error-prone to refer to the binary encodings of instructions directly because binary formats often contain numerous bit fields and can be frustratingly complex for some instruction architectures. The structural specification, and the intermediate form abstracts away encoding details, but requires a disassembler to perform translation from binary.

The encoding specification of an Isildur program specifies the bit patterns that represent instructions, operands and addressing modes. The *disassembler generator* subsystem uses these encoding specifications to automatically generate a disassembler that other tools use to translate raw bytes and words into instructions in the intermediate form. For example, the disassembler is used in program loaders that parse a file in a binary format such as ELF [24] and produce a complete program representation. The disassembler is also be used in implementing simulators that support dynamically updatable code, which detect when instruction memory is updated and invalidate previously decoded versions of instructions either immediately, or upon first execution [25].

Consistency Checks. Isildur's expressive encoding specification language reduces the complexity of specifying instruction encodings, but programmers tend to make a number of simple mistakes such as specifying the wrong binary constant for an opcode, omitting a field, or misaligning an instruction with one more or one fewer bits than the word size. To help catch these problems, the Isildur compiler performs a number of consistency checks on the specification, including *operand encoding completeness*, *word alignment*, and *nonambiguity*.

The first check, operand encoding completeness, requires that each instruction with operands has every bit of every operand's encoding in the instruction's encoding. For example, if instruction uses an operand with 5 encoding bits, then all 5 bits must be present somewhere in the encoding for the instruction. This

ensures that the disassembler will be able to locate all bits of the operand, load them from the instruction word, and decode them into the correct operand value.

The second check, word alignment, simply checks that every instruction encoding is aligned on the natural bit-width of the target architecture (usually 8, 16, or 32 bits). This catches specification errors where a binary constant is too small or too large, a field is omitted, etc.

Nonambiguity, the third check, is the most comprehensive because all instruction encodings in the Isildur file are considered together. The Isildur compiler generates all possible instruction/addressing mode encoding specifications and verifies that at most one instruction/addressing mode encoding applies for any input.

Decoder Construction. The core task of the disassembler generator is to discover an efficient algorithm for matching binary input to the correct instruction and addressing mode patterns. The first step is to enumerate the instruction/addressing mode possibilities into a list of bit patterns, where each bit pattern is a vector of bit values (0, 1, or _) and a label (instruction × addrmode). The bit values 0 and 1 are called *concrete* and values of _ may represent a bit of any value. These unknown bit values can correspond to unused bits, or bits that encode operands to the instruction. For the purposes of decoder construction, their values do not matter. Note that these bit patterns may not necessarily be all the same length (for example, the MSP430 has 16, 32, and 48-bit instructions).

The next step is to derive a systematic routine called a *decoder* to disambiguate the list of bit patterns. A decoder is a function that given a concrete bit pattern p, determines which pattern in the list matches p. The disassembler uses the decoder routine to match the instruction and addressing mode. The bits that encode the operands can then be extracted and translated into the internal operand format. Finally, the intermediate representation of the instruction can be constructed with the operands.

A natural and efficient approach to constructing a decoder is to build a binary tree. The algorithm starts by selecting a bit that is concrete in all bit patterns and then divides the list into two sub-lists; patterns with the value 0 for the selected bit are in one list, patterns with 1 in the other list. The algorithm then sets the decision bit to _ for all patterns and then applies itself to the sub-lists recursively. The binary tree that results is similar to a BDD [26, 27] in which the order of bit comparisons may differ along different paths. The binary tree approach has a number of benefits that are easy to show:

1. the number of nodes in the tree is at most $2n - 1$, where n is the number of bit patterns,
2. each bit in the concrete pattern is touched only once on any path,
3. unmatchable concrete bit patterns correspond to nodes with only one child,
4. each bit pattern at a leaf node has all concrete bits matched, and
5. an unambiguous list of patterns has exactly one bit pattern at each leaf node.

The last property of the binary tree approach is particularly helpful in describing instruction sets, because detecting *ambiguous* lists of bit patterns is

needed for many instruction set architectures. A list of bit patterns is ambiguous if there exists some concrete bit pattern p that matches more than one bit pattern in the list. Sometimes ambiguity is benign, for example, "clear register A" is often encoded as "xor register A with register A", or may be encoded as "move zero into register A". More importantly, in our experience, ambiguous encodings often signal a programmer has made a mistake describing the encodings that needs to be corrected. Detecting ambiguity while building the binary tree is trivial; in the step selecting a decision bit for a list of patterns, if there is no single bit that is concrete in all the patterns, then the list is ambiguous.

The Isildur compiler extends the binary tree decoder approach to match multiple contiguous bits at a time. This reduces the height of the tree and therefore the number of comparisons. A simple lookup table approach implements multi-way branches It is easy to show that properties 1, 2, 4, and 5 of the binary tree approach are retained, and property 3 has a natural extension. For simplicity, the discussion focuses only on binary decoder tree implementations, although the Isildur compiler internally uses multi-way trees.

Resolving Ambiguity. Some instruction set architectures have inherent ambiguities. While this does not arise often in RISC architectures, many CISC architectures with complex formats can have some instructions encode to the exact same concrete bits. Consider this example:

```
1____ = X        1_1__ = X
10___ = Y        10___ = Y
```

Two types of ambiguities are possible. In the first, Y is *more specific* than X, and could be automatically matched as a special case of X. In the second, neither X nor Y is more specific; this case must be resolved by adding priority manually.

Isildur provides two mechanisms to combat encoding ambiguities: (1) the `pseudo` modifier for instructions and (2) encoding priorities. The `pseudo` modifier for an instruction declaration instructs the disassembler generator to simply ignore the instruction. The result is that the disassembler with never decode bit patterns into that particular instruction, but will always prefer the remaining encodings. This is a reasonable solution for many instructions, especially instructions that are simply an alias for another instruction, such as a "jump if zero $\equiv$ jump if equal", or for shorthand instructions such as "clear register" being encoded as "xor register with itself".

The second mechanism the language provides is an optional *encoding priority* for each encoding declaration. The priority is simply an integer value representing a priority level. Bit patterns with higher priority are matched first, before considering patterns of lower priority. The problem of ambiguity is then reduced to ambiguity among patterns with the same priority level, as ambiguities among patterns of different priority levels always resolve to the higher priority pattern.

Priority levels can be supported in the binary tree decoder scheme in one of two ways. The simplest way is to simply divide the complete list of patterns into separate lists for each priority level. The original algorithm can then be run for each set independently, generating one decoder for each priority level. The

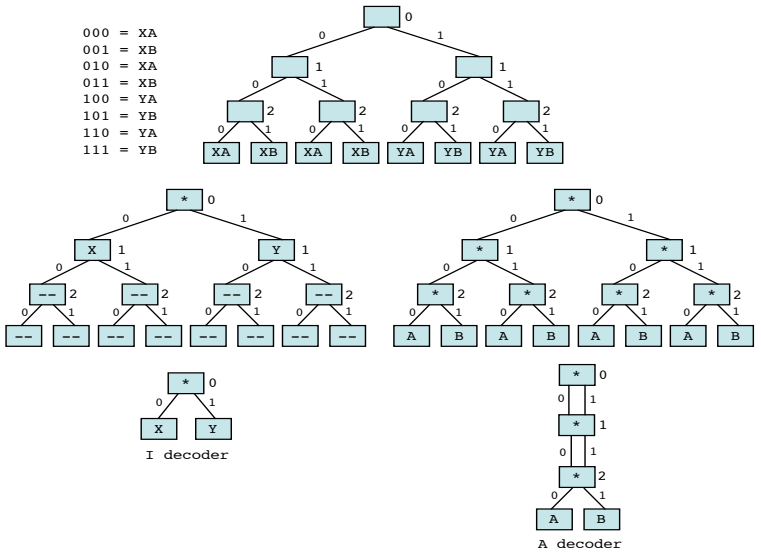

Fig. 2. Decoder tree reduction example

complete decoder routine then simply applies the decoders for each priority from highest to lowest until the first match is found. This works well for architectures with only a few simple ambiguities with one or two priority levels, but the more priority levels, the more redundant matching work is performed. In our experience, this approach was sufficient for AVR but not MSP430, which required 6 priority levels for full disambiguation.

The second approach to supporting ambiguity directly modifies the decoder tree construction algorithm. Instead of segregating patterns by priority level, all patterns are considered together as before. To select a decision bit, the new algorithm need only choose a bit that is concrete in all of the patterns of the highest priority level, instead of a bit that is concrete in all priority levels. If the algorithm cannot find such a bit, then the patterns within the highest priority level are ambiguous. After selecting the decision bit, the list is then split into two sub-lists recursively, except that some lower priority patterns may have _ for the value of the decision bit. These patterns are replicated and put into both sub-lists. In the worst case, this can lead to expansion exponential in the length of the longest pattern. The selection of the decision bit becomes more important for efficiency in this case. Our compiler uses a simple heuristic that selects the eligible bit position that is concrete in the most number of patterns in the list; this works very well in practice.

Optimization. Tree-based decoders are generally quite small, and the number of nodes guaranteed to be proportional to the number of bit patterns. However, in the naïve approach of generating one bit pattern per instruction/addressing mode combination, up to $O(n^2)$ bit patterns can be generated. For RISC architectures, most instructions have just one addressing mode, but for CISC archi-

tectures, there can be thousands of bit patterns. For example, in the MSP430 description, 44 instructions give rise to more than 1400 bit patterns. Is there a way to reduce the size of the decoder implementation for such architectures?

On first glance, it appears that some of the bits in a pattern encode the addressing mode and some of the bits encode the instruction. If this were true, then we could have one decoder for the instruction bits and one decoder for the addressing mode bits. Unfortunately, we found dependencies between decoding the addressing mode and instruction type because some instructions only implement certain addressing modes, and some addressing modes only apply to certain instructions. For some architectures, neither the instruction decoder nor the addressing mode decoder can be made unambiguous independently.

Our solution is to recognize that the general case requires a combined instruction/addressing mode decoder for full disambiguation, but the size of the decoder can be reduced by finding redundant paths. Our factoring algorithm automatically reduces a combined decoder tree into two reduced decoder graphs, one to match the instruction type, and one to match the addressing mode, without introducing ambiguity.

Our factoring algorithm starts with the combined decoder tree D generated by applying the naïve $O(n^2)$ algorithm. The first step is to replicate D to produce two identical decoders: I to match instructions, and A to match addressing modes. Each decoder tree will be reduced independently while preserving the order of bit comparisons in every path. Figure 2 shows an example.

The key to the factoring algorithm is to give each node a smart label that is used by a general BDD-like reduction algorithm. Consider first the decoder I. Each node is given a label that marks it as one of three types: a node whose successors correspond to multiple different instructions: "*"; a node that denotes a single instruction i: "i", and a node that is a successor of a node labeled with some "i": "–". The labeling algorithm for the tree works from the leaves of the tree up; it is simple enough that it is left as an exercise for the reader. After labeling, each node in I is marked either an instruction name or the special "–" and "*" labels. The next step, decoder reduction, will use these labels to reduce the tree.

The factoring algorithm first considers the nodes with the label "–". These nodes can only occur as children to a node labeled with an instruction. Further, all children of a node labeled "–" are also labeled with "–". Because these nodes contribute no more information about disambiguating the instruction type, they can simply be removed.

Now, we would like to collapse redundant nodes in the tree to reduce its overall size. First, after removing nodes with label "–", all leaf nodes are now labeled with an instruction. Intuitively, this tree can be reduced by collapsing leaves with the same label into one node. This is fine for nodes that are leaves, but for internal nodes in the tree, we must only collapse nodes when it will not alter the order of bit comparisons on any path. For this, we need check three things when potentially collapsing two nodes, (1) they have the same label, (2) they agree on which bit to compare next, (3) the first two properties apply

recursively to the left children and right children, respectively. Using the last three criteria, we can implement a depth-first factoring algorithm that visits each node once, and with constant-time node identity checks using a hashing scheme, this algorithm runs in linear time. This factoring algorithm is a straightforward extension of the classical BDD reduction algorithm.

The same reduction algorithm, can be applied to A, only this time we label the nodes in A with addressing mode names rather than instruction types. In this way, A is reduced to a simpler graph (different from I) that disambiguates addressing modes, but the order of its bit comparisons are not altered from the original decoder D.

After reducing decoders I and A with the factoring algorithm, we are left with decoders I and A that are much smaller than the original D. These two decoder graphs can be used to build a single decoder routine by running them in a lock-step fashion. The two graphs originated from the same decoder tree and the factoring process preserved the ordering of bit comparisons, so when decoding starts with the bit at the root in I, the same bit is being compared in A. For each step, the routine moves to the next node in I depending on the value of the bit. Ordering preservation implies there will always be a corresponding move in the graph for A. The two new nodes will agree on which bit to compare next, since order was preserved for all paths in both graphs. The resulting decoding algorithm is simple; run both decoders in lock-step until both either reach a leaf node or one signals an error. I will match the instruction type; A will match the addressing mode.

The reduction algorithm can reduce the size of both graphs by as much as a quadratic factor. For the MSP430, the original single dimension decoder tree has 1990 nodes from 1400 instruction/addressing mode bit patterns; after copying and factoring automatically, the I and A decoder graphs have a total of 169 nodes. For comparison, the AVR, which has 101 instructions, has 160 nodes in its decoder implementation.

4 Conclusion

We have presented a declarative approach to building simulators that support updatable code and have used our approach to build a sensor network simulator for Mica2-based sensor networks, as well as simulators for the MIPS and MSP430 architectures. We automatically generate three central pieces of the simulator, namely the Java classes that represent instructions and operands, an interpreter for the instruction set, and a disassembler. The Isildur language and compiler help simulator designers develop more robust simulators and machine code tools by offering a declarative language, a static type system, and a suite of consistency checks that find pernicious low-level bugs and allow the compiler to generate fast, elegant, and type correct Java code.

A Isildur's Syntax, Semantics, and Type System

In this appendix we describe the Isildur's behavioral specification in detail. We will present the syntax, operational semantics, and type system, along with a proof of type soundness. We have automatically verified the type soundness result for a core subset of Isildur using the Coq proof assistant (http://coq.inria.fr).

A.1 Syntax

$$
\begin{array}{lllll}
e & = e_1\ bop\ e_2 & \mid uop\ e & \mid e_1.e_2 & \\
 & \mid e[n_1 : n_2] & \mid e_1[e_2] & \mid e\textbf{:}T & \\
 & \mid \textbf{read.}T(e) & \mid f(e_1, e_2, \ldots, e_n) & \mid x & \\
 & \mid b & \mid n & & \\
lv & = x & \mid lv[e] & & \\
s & = \textbf{if}\ (e)\ s & \mid \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2 & \mid lv = e; & \\
 & \mid lv[n_1 : n_2] = e; & \mid \textbf{return}\ e; & \mid \textbf{write.}T(x, e); & \\
 & \mid f(e_1, e_2, \ldots, e_n); & \mid \textbf{local}\ x\textbf{:}T = e; & \mid sl & \\
sl & = \{\} & \mid \{s\} & \mid \{s\ sl\} & \\
bop & = + & \mid - & \mid * & \mid / \\
 & \mid \% & \mid \& & \mid \mid & \mid \hat{} \\
 & \mid \textbf{and} & \mid \textbf{or} & \mid \textbf{xor} & \mid << \\
 & \mid >> & \mid < & \mid \leq & \mid \geq \\
 & \mid > & \mid == & \mid != & \\
uop & = + & \mid \tilde{} & \mid - & \mid !
\end{array}
$$

While statements lists can are normally written $\{s_1\ s_2\ \ldots s_n\}$, internally this can be represented as $\{s_1\ \{s_2\ \{\ldots\ \{s_n\}\}\}\}$ which simplifies the typing rules and operational semantics.

A.2 Type System

There is only one possible nontrivial subtyping relation. This involves enums and enum-subsets. An enum-subset is a subtype of the parent enum. While there is technically no subtype relationship between any integer type, there is an assignability relationship where an integer is assignable to a different integer type if either condition is true given $T_1 = g_1\,\textbf{int}.z_1, T_2 = g_2\,\textbf{int}.z_2$:

- If $g_1 = g_2, z_1 \leq z_2$ then T_1 is assignable to T_2.
- If $g_1 = +, z_1 < z_2$ then T_1 is assignable to T_2.

In our type system we have the following types:

- **boolean** - Boolean type.
- $g\,\textbf{int}.z$ - Integer types can be unsigned or signed $(+, -)$, g, and have a specific bit size, z.
- $\textbf{Enum}(ID, [x_1, x_2, \ldots, x_n])$ - Enum types have the name of their type as well as a list of member enum values. Enum types may have non trivial subtypes if another enum type is declare to be a enum-subset.

- **SOperand**(ID, z, T) - Simple operand types consists of the name of the type, the size in bits of the operand, and the base type in which the operand may be converted to.
- **COperand**$(ID, [x_i \mapsto O_i])$ - Compound operand types contains a map of suboperand names to their operand types, as well as the name of operand type.
- **map**$\langle T_1, T_2 \rangle$ - Map types consists of a key type and a value type.

Additionally we use some psuedo functions that for typing rules:

- $isAssignable(T_1, T_2)$ - This function returns true if T_2 is assignable to T_1. More specifically, any type is assignable to the exact same type; any subtype of another type is assignable; and any integer type is assignable if it follows the above rules.
- $lookupEnumType(ID)$ - returns the enum type for the given ID.
- $lookupSubroutine(f)$ - returns a list of types: the return type and the type of each argument, in order.
- $readMethodExists(O, T)$ - returns true if there exists a read method for type T in operand type O.
- $writeMethodExists(O, T)$ - returns true if there exists a write method for type T in operand type O.
- $g_1 \sqcup g_2$ - we use this notation on sign notation to mean that if g_1 and g_2 are unsigned, then $g_1 \sqcup g_2$ is unsigned, otherwise the value is signed.

Type rules come in two forms. For expressions we have $\Gamma, \Psi \vdash e : T$ which says that given an environment Γ and a store type Ψ, it is provable that expression e has type T. The enviroment Γ is a map from variable names to types and the notation $\Gamma(x)$ retrieves the type for variable x. The store type Ψ is a map from locations to types, with a similary notation $\Psi(l)$ having the type of location l.

The other form of typing rule is for statements denoted as $\Gamma, \Psi, R \vdash s$ which says that given an environment Γ, a store type Ψ, and a return type R, it is provable that statement s type checks.

Values:

$$\Gamma, \Psi \vdash b : \mathbf{boolean} \tag{1}$$

$$\Gamma, \Psi \vdash int(n, g, z) : g\,\mathbf{int}.z \tag{2}$$

$$\Gamma, \Psi \vdash operand(n, \mathbf{SOperand}(ID, z, T)) : \tag{3}$$

$$\Gamma, \Psi \vdash operand([x_i \mapsto l_i], \mathbf{COperand}(ID, [x_i \mapsto O_i])) : \mathbf{COperand}(ID, [x_i \mapsto O_i]) \tag{4}$$

$$\Gamma, \Psi \vdash enum(n, \mathbf{Enum}(ID, [x_1, x_2, \ldots, x_n])) : \mathbf{Enum}(ID, [x_1, x_2, \ldots, x_n]) \tag{5}$$

$$\Gamma, \Psi \vdash map(T_1, T_2) : \mathbf{map}\langle T_1, T_2 \rangle \tag{6}$$

Addition:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \qquad g_1 \sqcup g_2 = g_3 \\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2 \; max(z_1, z_2) + 1 = z_3\end{array}}{\Gamma, \Psi \vdash e_1 + e_2 : g_3\,\mathbf{int}.z_3} \tag{7}$$

Subtraction:

$$\frac{\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \; \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2 \; max(z_1, z_2) + 1 = z_3}{\Gamma, \Psi \vdash e_1 - e_2 : -\,\mathbf{int}.z_3} \tag{8}$$

Multiplication:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \; g_1 \sqcup g_2 = g_3 \\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2 \; z_1 + z_2 = z_3\end{array}}{\Gamma, \Psi \vdash e_1 \; * \; e_2 : g_3\,\mathbf{int}.z_3} \tag{9}$$

Division, Modulus:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \; op \in \{/, \%\} \\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2 \; g_1 \sqcup g_2 = g_3\end{array}}{\Gamma, \Psi \vdash e_1 \; \mathbf{op} \; e_2 : g_3\,\mathbf{int}.z_1} \tag{10}$$

Bitwise And, Or, Xor:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \quad op \in \{\&, |, \hat{}\} \\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2 \; max(z_1, z_2) = z_3\end{array}}{\Gamma, \Psi \vdash e_1 \; \mathbf{op} \; e_2 : +\,\mathbf{int}.z_3} \tag{11}$$

Logical And, Or, Xor:

$$\frac{\Gamma, \Psi \vdash e_1 : \mathbf{boolean} \; \Gamma, \Psi \vdash e_2 : \mathbf{boolean} \; op \in \{\mathbf{and}, \mathbf{or}, \mathbf{xor}\}}{\Gamma, \Psi \vdash e_1 \; \mathbf{op} \; e_2 : \mathbf{boolean}} \tag{12}$$

Left, Right Shift:

$$\frac{\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2\ op \in \{<<, >>\}}{\Gamma, \Psi \vdash e_1\ \mathbf{op}\ e_2 : g_1\,\mathbf{int}.z_1} \tag{13}$$

Less Than, Less Than or Equals, Greater Than or Equals, Greater Than:

$$\frac{\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2\ op \in \{<, \leq, \geq, >\}}{\Gamma, \Psi \vdash e_1\ \mathbf{op}\ e_2 : \mathbf{boolean}} \tag{14}$$

Equality, Inequality:

$$\frac{\begin{array}{ll}\Gamma, \Psi \vdash e_1 : T_1 & op \in \{==, !=\} \\ \Gamma, \Psi \vdash e_2 : T_2\ T_1 \neq \mathbf{map}\langle T_3, T_4\rangle, T_2 \neq \mathbf{map}\langle T_5, T_6\rangle & \end{array}}{\Gamma, \Psi \vdash e_1\ \mathbf{op}\ e_2 : \mathbf{boolean}} \tag{15}$$

Complement, Unsign:

$$\frac{\Gamma, \Psi \vdash e : g\,\mathbf{int}.z\ op \in \{\tilde{}, +\}}{\Gamma, \Psi \vdash \mathbf{op}\ e : g\,\mathbf{int}.z} \tag{16}$$

Negate:

$$\frac{\Gamma, \Psi \vdash e : g\,\mathbf{int}.z\ z + 1 = z'}{\Gamma, \Psi \vdash -e : g\,\mathbf{int}.z'} \tag{17}$$

Not:

$$\frac{\Gamma, \Psi \vdash e : \mathbf{boolean}}{\Gamma, \Psi \vdash !e : \mathbf{boolean}} \tag{18}$$

Enum:

$$\frac{getEnumType(x) = \mathbf{Enum}(ID, [x_1, x_2, \ldots, x_n])}{\Gamma, \Psi \vdash ID.x_i : \mathbf{Enum}(ID, [x_1, x_2, \ldots, x_n])} \tag{19}$$

Suboperand:

$$\frac{\Gamma, \Psi \vdash x : O\ O = \mathbf{COperand}(ID, [x_i \mapsto O_i])}{\Gamma, \Psi \vdash x.x_i : O_i} \tag{20}$$

Fixed Range:

$$\frac{\begin{array}{ll}max(n_1, n_2) = hiBit & min(n_1, n_2) = loBit \\ \Gamma, \Psi \vdash e : g\,\mathbf{int}.z & 0 \leq loBit < hiBit < z\end{array}}{\Gamma, \Psi \vdash e[n_1 : n_2] : +\,\mathbf{int}.(hiBit - loBit + 1)} \tag{21}$$

Bit Index:

$$\frac{\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1\ \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2}{\Gamma, \Psi \vdash e_1[e_2] : \mathbf{boolean}} \tag{22}$$

Map Index:

$$\frac{\Gamma, \Psi \vdash e_1 : \mathbf{map}\langle T_1, T_2\rangle\ \Gamma, \Psi \vdash e_2 : T_1'\ isAssignable(T_1, T_1')}{\Gamma, \Psi \vdash e_1[e_2] : T_2} \tag{23}$$

Conversion:

$$\frac{\Gamma, \Psi \vdash e : T}{\Gamma, \Psi \vdash e{:}T : T} \tag{24}$$

$$\frac{\Gamma, \Psi \vdash e : \textbf{boolean}}{\Gamma, \Psi \vdash e{:}g\,\textbf{int}.z : g\,\textbf{int}.z} \tag{25}$$

$$\frac{\Gamma, \Psi \vdash e : \textbf{SOperand}(ID, z, T)}{\Gamma, \Psi \vdash e{:}T : T} \tag{26}$$

$$\frac{\Gamma, \Psi \vdash e : \textbf{Enum}(ID, [x_1, x_2, \ldots, x_n])}{\Gamma, \Psi \vdash e{:}g\,\textbf{int}.z : g\,\textbf{int}.z} \tag{27}$$

Read Method:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash x : O \\ readMethodExists(O, T)\end{array}}{\Gamma, \Psi \vdash \textbf{read.}T(x) : T} \tag{28}$$

Subroutine Call (expression):

$$\frac{\begin{array}{c}lookupSubroutine(f) = \{T_r, T_1, T_2, \ldots, T_n\} \\ \Gamma, \Psi \vdash e_i : T_i' \\ isAssignable(T_i, T_i') \\ T_r \neq \textbf{void}\end{array}}{\Gamma, \Psi \vdash f(e_1, e_2, \ldots, e_n) : T_r} \tag{29}$$

Variable Reference:

$$\frac{\Gamma(x) = T}{\Gamma, \Psi \vdash x : T} \tag{30}$$

Location Reference:

$$\frac{\Psi(l) = T}{\Gamma, \Psi \vdash l : T} \tag{31}$$

Boolean Literal:

$$\Gamma, \Psi \vdash b : \textbf{boolean} \tag{32}$$

Integer Literal:

$$\frac{\begin{array}{c}g = \begin{cases} + & n \geq 0 \\ - & n < 0 \end{cases} \\ z = \begin{cases} 1 & n = 0 \\ \lceil \log_2 n \rceil & n > 0 \\ \lceil \log_2 -n \rceil + 1 & n < 0 \end{cases}\end{array}}{\Gamma, \Psi \vdash n : g\,\textbf{int}.z} \tag{33}$$

Statement Expression:

$$\frac{\Gamma, \Psi, T \vdash s}{\Gamma, \Psi \vdash \lceil s \rfloor : T} \tag{34}$$

If:

$$\frac{\Gamma, \Psi \vdash e : \textbf{boolean}\ \Gamma, \Psi, \Gamma \vdash Ms}{\Gamma, \Psi, M \vdash \textbf{if}\ (e)\ s} \tag{35}$$

If-Else:

$$\frac{\Gamma, \Psi \vdash e : \textbf{boolean}\ \Gamma, \Psi, \Gamma \vdash Ms_1\ \Gamma, \Psi, \Gamma \vdash Ms_2}{\Gamma, \Psi, M \vdash \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2} \tag{36}$$

Assignment:

$$\frac{\Gamma, \Psi \vdash lv : T\ \Gamma, \Psi \vdash e : T'\ isAssignable(T, T')}{\Gamma, \Psi, M \vdash lv = e;} \tag{37}$$

Bit Index Assignment:

$$\frac{\Gamma, \Psi \vdash lv : g\,\textbf{int}.z\ \Gamma, \Psi \vdash e_1 : g_1\,\textbf{int}.z_1\ \Gamma, \Psi \vdash e_2 : \textbf{boolean}}{\Gamma, \Psi, M \vdash lv[e_1] = e_2;} \tag{38}$$

Map Index Assignment:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash lv : \textbf{map}\langle T_1, T_2\rangle \qquad \Gamma, \Psi \vdash e_1 : T_1' \qquad \Gamma, \Psi \vdash e_2 : T_2' \\ isAssignable(T_1, T_1') \quad isAssignable(T_2, T_2')\end{array}}{\Gamma, \Psi, M \vdash lv[e_1] = e_2;} \tag{39}$$

Bit Range Assignment:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash lv : g\,\textbf{int}.z \qquad \Gamma, \Psi \vdash e : g'\,\textbf{int}.z' \\ max(n_1, n_2) = hiBit \quad min(n_1, n_2) = loBit \\ 0 \leq loBit < hiBit < z\ z' \leq (hiBit - loBit)\end{array}}{\Gamma, \Psi, M \vdash l[n_1 : n_2] = e;} \tag{40}$$

Return:

$$\frac{\Gamma, \Psi \vdash e : M'\ isAssignable(M, M')}{\Gamma, \Psi, M \vdash \textbf{return}\ e;} \tag{41}$$

Write Method:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash x : OwriteMethodExists(O, T) \\ \Gamma, \Psi \vdash e : T' \qquad\qquad isAssignable(T, T')\end{array}}{\Gamma, \Psi, M \vdash \textbf{write.}T(x, e);} \tag{42}$$

Subroutine Call (Statement):

$$\frac{\begin{array}{c}lookupSubroutine(f) = \{\textbf{void}, T_1, T_2, \ldots, T_n\} \\ \Gamma, \Psi \vdash e_i : T_i' \\ isAssignable(T_i, T_i')\end{array}}{\Gamma, \Psi, M \vdash f(e_1, e_2, \ldots, e_n);} \tag{43}$$

Local Variable Declaration:

$$\frac{x \notin \Gamma\ \Gamma, \Psi \vdash e : T'\ isAssignable(T, T')}{\Gamma, \Psi, M \vdash \textbf{local}\ x\textbf{:}T = e;} \tag{44}$$

$$\frac{\begin{array}{c}x \notin \Gamma \qquad\qquad \Gamma, \Psi \vdash e : T' \\ \Gamma, \Psi, \{x \mapsto T\}E \vdash Msl\ isAssignable(T, T')\end{array}}{\Gamma, \Psi, M \vdash \{\textbf{local}\ x\textbf{:}T = e;\ sl\}} \tag{45}$$

Block:

$$\Gamma, \Psi, M \vdash \{\} \tag{46}$$

$$\frac{\Gamma, \Psi, M \vdash s}{\Gamma, \Psi, M \vdash \{s\}} \tag{47}$$

$$\frac{\Gamma, \Psi, M \vdash s \; \Gamma, \Psi, M \vdash sl}{\Gamma, \Psi, M \vdash \{s \; sl\}} \tag{48}$$

Skip:

$$\Gamma, \Psi, M \vdash \textbf{skip;} \tag{49}$$

Error:

$$\Gamma, \Psi \vdash error : T \tag{50}$$

$$\Gamma, \Psi, M \vdash error \tag{51}$$

Store Type:

$$\frac{\Psi(l_i) = T_i \quad \Gamma, \Psi \vdash v_i : T_i}{\vdash \{l_i \mapsto v_i\} : \Psi} \tag{52}$$

To type check program states we have the additional rule:

$$\frac{\vdash \sigma : \Psi \quad \emptyset, \Psi, T \vdash s}{\vdash \langle s, \sigma \rangle} \tag{53}$$

$$\frac{\vdash \sigma : \Psi \quad \emptyset, \Psi \vdash e : T}{\vdash (e, \sigma) : T} \tag{54}$$

A.3 Operational Semantics

We have two forms of steps for our operational semantics. Expressions stepping is denoted as $(e, \sigma) \rightarrow (e', \sigma')$ where e is an expression and σ is the store. Similarly, we have statement steps which look like $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$, where s is a statement and σ is the store.

Some of the actions performed in a step have been reduced to a function as well as some notational items. They are:

- σ - the store which maps locations to values.
- l - locations in the store.
- r - a subset of locations in the store which only points to map values.
- m - a map value.
- $lookupSubroutine(f)$ - contains a list of variable names and a statement for the body of the subroutine. All the global variables have already been substituted in all statements. If this function returns *external* it means that the actual implementation of the subroutine exists outside of the language and is assumed to work properly. The lookupSubroutine method implies that the body of the method also type checks.

- $lookupReadMethod(O, T)$ - retrieves the body of a read method for an operand and type pair.
- $lookupWriteMethod(O, T)$ - retrieves the body of a write method for an operand and type pair.
- $getEnumValue(x_1, x_2)$ - retrieves the enum value for the enum type x_1 and the specific value x_2.
- $zeroExtend(n, z)$ - takes the bit reperesntation and expands its size to z bits with the new bits set to zero.
- $truncate(n, z)$ - takes the bit representation and returns the lower $z - 1$ bits. The sign may change as a result.
- $newLocation(\sigma)$ - returns a fresh location from the store.
- $f(\sigma, v_1, v_2, \ldots, v_n)$ - some subroutines are external and need to be handled outside of the description language. This represents the external code that can alter the state, return a value, or return an error.
- $map(T_1, T_2)(v)$ - like external functions maps also are handled outside of the language they either return some location or an error.

The following are values in our language:

- b - A boolean value; it can be **true** or **false**.
- $int(n, g, z)$ - Integer values consists of three values, the numeric value of the integer, a sign to indicate how what the bit representation is, and a bit size indicating how many bits are in the bit representation.
- $enum(n, T)$ - An enum value consists of the numeric value as well as the type of the enum itself.
- $operand(n, T)$ - A simple operand value consisting of the numeric value of the operand and the type of the operand.
- $operand([x_i \mapsto l_i], T)$ - A compound operand value which is a map of sub-operand names to store locations and the type of the operand.
- $map(T_1, T_2)$ - A map value where it takes values of type T_1 and returns values of T_2.

Binary Operation:

$$\frac{(e_1,\sigma) \rightarrow (e_1',\sigma')\ op \in \left\{ \begin{array}{ccccc} +, & -, & *, & /, & \%, \\ \&, & |, & \hat{}, & \textbf{and}, & \textbf{or}, \\ \textbf{xor}, & <<, & >>, & <, & \leq, \\ \geq, & >, & ==, & != & \end{array} \right\}}{(e_1\ \textbf{op}\ e_2,\sigma) \rightarrow (e_1'\ \textbf{op}\ e_2,\sigma')} \tag{55}$$

$$\frac{(e,\sigma) \rightarrow (e',\sigma')\ op \in \left\{ \begin{array}{ccccc} +, & -, & *, & /, & \%, \\ \&, & |, & \hat{}, & \textbf{and}, & \textbf{or}, \\ \textbf{xor}, & <<, & >>, & <, & \leq, \\ \geq, & >, & ==, & != & \end{array} \right\}}{(v\ \textbf{op}\ e,\sigma) \rightarrow (v\ \textbf{op}\ e',\sigma')} \tag{56}$$

Addition:

$$\frac{n_1 + n_2 = n_3\ max(z_1,z_2)+1 = z_3\ g_1 \sqcup g_2 = g_3}{(int(n_1,g_1,z_1) + int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,g_3,z_3),\sigma)} \tag{57}$$

Subtraction:

$$\frac{n_1 - n_2 = n_3\ max(z_1,z_2)+1 = z_3}{(int(n_1,g_1,z_1) - int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,-,z_3),\sigma)} \tag{58}$$

Multiplication:

$$\frac{n_1 \cdot n_2 = n_3\ z_1 + z_2 = z_3\ g_1 \sqcup g_2 = g_3}{(int(n_1,g_1,z_1)\ *\ int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,g_3,z_3),\sigma)} \tag{59}$$

Division:

$$\frac{n_1/n_2 = n_3\ g_1 \sqcup g_2 = g_3}{(int(n_1,g_1,z_1)\ /\ int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,g_3,z_1),\sigma)} \tag{60}$$

Modulus:

$$\frac{n_1 \bmod n_2 = n_3\ g_1 \sqcup g_2 = g_3}{(int(n_1,g_1,z_1)\ \%\ int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,g_3,z_1),\sigma)} \tag{61}$$

Bitwise And:

$$\frac{\begin{array}{cc} max(z_1,z_2) = z_3 & zeroExtend(n_1,z_3) = n_1' \\ zeroExtend(n_2) = n_2' & n_1'\ \&\ n_2' = n_3 \end{array}}{(int(n_1,g_1,z_1)\ \%\ int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,g_1,z_3),\sigma)} \tag{62}$$

Bitwise Or:

$$\frac{\begin{array}{cc} max(z_1,z_2) = z_3 & zeroExtend(n_1,z_3) = n_1' \\ zeroExtend(n_2) = n_2' & n_1'\ |\ n_2' = n_3 \end{array}}{(int(n_1,g_1,z_1)\ |\ int(n_2,g_2,z_2),\sigma) \rightarrow (int(n_3,g_1,z_3),\sigma)} \tag{63}$$

Bitwise Xor:

$$\frac{\begin{array}{cc} max(z_1, z_2) = z_3 & zeroExtend(n_1, z_3) = n_1' \\ zeroExtend(n_2) = n_2' & n_1' \ \hat{} \ n_2' = n_3 \end{array}}{(int(n_1, g_1, z_1) \ \hat{} \ int(n_2, g_2, z_2), \sigma) \rightarrow (int(n_3, g_1, z_3), \sigma)} \tag{64}$$

Logical And:

$$\frac{b_1 \wedge b_2 = b_3}{(b_1 \ \textbf{and} \ b_2, \sigma) \rightarrow (b_3, \sigma)} \tag{65}$$

Logical Or:

$$\frac{b_1 \vee b_2 = b_3}{(b_1 \ \textbf{or} \ b_2, \sigma) \rightarrow (b_3, \sigma)} \tag{66}$$

Logical Xor:

$$\frac{b_1 \oplus b_2 = b_3}{(b_1 \ \textbf{xor} \ b_2, \sigma) \rightarrow (b_3, \sigma)} \tag{67}$$

Shift Left:

$$\frac{truncate(n_1 * 2^{n_2}, z_1) = n_3}{(int(n_1, g_1, z_1) << int(n_2, g_2, z_2), \sigma) \rightarrow (int(n_3, g_1, z_1), \sigma)} \tag{68}$$

Shift Right:

$$\frac{truncate(n_1 / 2^{n_2}, z_1) = n_3}{(int(n_1, g_1, z_1) >> int(n_2, g_2, z_2), \sigma) \rightarrow (int(n_3, g_1, z_1), \sigma)} \tag{69}$$

Integer Comparison:

$$\frac{(n_1 \ \textbf{op} \ n_2) = b \ op \in \{<, \leq, \geq, >\}}{(int(n_1, g_1, z_1) \ \textbf{op} \ int(n_2, g_2, z_2), \sigma) \rightarrow (b, \sigma)} \tag{70}$$

Integer Equality:

$$\frac{(n_1 = n_2) = b}{(int(n_1, g_1, z_1) == int(n_2, g_2, z_2), \sigma) \rightarrow (b, \sigma)} \tag{71}$$

Integer Inequality:

$$\frac{(n_1 \neq n_2) = b}{(int(n_1, g_1, z_1) \ != int(n_2, g_2, z_2), \sigma) \rightarrow (b, \sigma)} \tag{72}$$

Simple Operand Equality:

$$\frac{(n_1 = n_2) = b}{(operand(n_1, T) == operand(n_2, T), \sigma) \rightarrow (b, \sigma)} \tag{73}$$

Simple Operand Inequality:

$$\frac{(n_1 \neq n_2) = b}{(operand(n_1, T) \ != operand(n_2, T), \sigma) \rightarrow (b, \sigma)} \tag{74}$$

Enum Equality:

$$\frac{(n_1 = n_2) = b}{(enum(n_1, T) == enum(n_2, T), \sigma) \rightarrow (b, \sigma)} \tag{75}$$

Enum Inequality:

$$\frac{(n_1 \neq n_2) = b}{(enum(n_1, T) == enum(n_2, T), \sigma) \rightarrow (b, \sigma)} \tag{76}$$

Equality:

$$\frac{(v_1 = v_2) = b}{(v_1 == v_2, \sigma) \rightarrow (b, \sigma)} \tag{77}$$

Inequality:

$$\frac{(v_1 \neq v_2) = b_3}{(v_1 \; != v_2, \sigma) \rightarrow (b, \sigma)} \tag{78}$$

Unary Operations:

$$\frac{(e, \sigma) \rightarrow (e', \sigma') \; op \in \{+, -, !, \tilde{}\}}{(\textbf{op} \; e, \sigma) \rightarrow (\textbf{op} \; e', \sigma')} \tag{79}$$

Unsign:

$$\frac{|n| = n'}{(+int(n, g, z), \sigma) \rightarrow (int(n', +, z), \sigma)} \tag{80}$$

Complement:

$$\frac{\text{each bit of } n \text{ flipped} = n'}{(-int(n, g, z), \sigma) \rightarrow (int(n', g, z), \sigma)} \tag{81}$$

Negate:

$$\frac{-n = n'}{(-int(n, g, z), \sigma) \rightarrow (int(n', +, z), \sigma)} \tag{82}$$

Not:

$$\frac{\neg b = b'}{(!b, \sigma) \rightarrow (b', \sigma)} \tag{83}$$

Enum:

$$\frac{getEnumValue(x_1, x_2) = enum(x_1, n)}{(x_1.x_2, \sigma) \rightarrow (enum(x_1, n), \sigma)} \tag{84}$$

Suboperand:

$$(operand([x_i \mapsto l_i], O).x_i, \sigma) \rightarrow (l_i, \sigma) \tag{85}$$

Index:

$$\frac{(e_1, \sigma) \rightarrow (e_1', \sigma')}{(e_1[e_2], \sigma) \rightarrow (e_1'[e_2], \sigma')} \tag{86}$$

$$\frac{(e, \sigma) \rightarrow (e', \sigma')}{(v[e], \sigma) \rightarrow (v[e'], \sigma')} \tag{87}$$

Bit Index:

$$\frac{0 \leq n_2 < z_1 \text{ the } n_2\text{th bit of } n_1 = 1}{(int(n_1, g_1, z_1)[int(n_2, g_2, z_2)], \sigma) \rightarrow (\textbf{true}, \sigma)} \tag{88}$$

$$\frac{0 \leq n_2 < z_1 \text{ the } n_2\text{th of } n_1 = 0}{(int(n_1, g_1, z_1)[int(n_2, g_2, z_2)], \sigma) \rightarrow (\textbf{false}, \sigma)} \tag{89}$$

Map Index:

$$\frac{\sigma(r) = map(T_1, T_2)\ map(T_1, T_2)(v) = l}{(r[v], \sigma) \rightarrow (l, \sigma)} \tag{90}$$

Bit Range:

$$\frac{(e, \sigma) \rightarrow (e', \sigma')}{(e[n_1 : n_2], \sigma) \rightarrow (e'[n_1 : n_2], \sigma')} \tag{91}$$

$$\frac{\begin{array}{cc} max(n_1, n_2) = hiBit & min(n_1, n_2) = loBit \\ (hiBit - loBit + 1) = z' \text{ the bits from } hiBit \text{ to } loBit \text{ of } n = n' & \end{array}}{(int(n, g, z)[n_1 : n_2], \sigma) \rightarrow (int(n', +, z'), \sigma)} \tag{92}$$

Conversion:

$$\frac{(e, \sigma) \rightarrow (e', \sigma')}{(e\textbf{:}T, \sigma) \rightarrow (e'\textbf{:}T, \sigma')} \tag{93}$$

Self Conversions:

$$(b\textbf{:boolean}, \sigma) \rightarrow (b, \sigma) \tag{94}$$

$$(operand(n, T)\textbf{:}T, \sigma) \rightarrow (operand(n, T), \sigma) \tag{95}$$

$$(operand([x_i \mapsto l_i], T)\textbf{:}T, \sigma) \rightarrow (operand([x_i \mapsto l_i], T), \sigma) \tag{96}$$

$$(enum(ID, n)\textbf{:Enum}(ID, [x_1, x_2, \ldots, x_n]), \sigma) \rightarrow (enum(ID, n), \sigma) \tag{97}$$

Enum Conversions:

$$\frac{g = \begin{cases} + & n \geq 0 \\ - & n < 0 \end{cases} \quad z = \begin{cases} 1 & n = 0 \\ \lceil \log_2 n \rceil & n > 0 \\ \lceil \log_2 -n \rceil + 1 & n < 0 \end{cases}}{(enum(ID, n)\textbf{:}g\,\textbf{int}.z, \sigma) \rightarrow (int(n, g', z')\textbf{:}g\,\textbf{int}.z, \sigma)} \tag{98}$$

Operand Conversions:

$$\frac{(n \neq 0) = b}{(operand(n, T)\textbf{:boolean}, \sigma) \rightarrow (b, \sigma)} \tag{99}$$

$$(operand(n, \mathbf{SOperand}(ID, z, g_1\,\mathbf{int}.z_1))\mathbf{:}T, \sigma) \rightarrow (int(n, g_1, z_1)\mathbf{:}T, \sigma) \quad (100)$$

Boolean to Integer:

$$\frac{n = (b)?1 : 0}{(b\mathbf{:}g\,\mathbf{int}.z, \sigma) \rightarrow (int(n, g, z), \sigma)} \quad (101)$$

Integer to Boolean:

$$\frac{(n \neq 0) = b}{(int(n, g, z)\mathbf{:boolean}, \sigma) \rightarrow (b, \sigma)} \quad (102)$$

Integer to Integer:

$$\frac{z_1 \leq z_2}{(int(n, g, z_1)\mathbf{:}g\,\mathbf{int}.z_2, \sigma) \rightarrow (int(n, g, z_2), \sigma)} \quad (103)$$

$$\frac{z_1 \leq z_2 \; n' = \begin{cases} n & n \geq 0 \\ n + 2^{z_1} & n < 0 \end{cases}}{(int(n, -, z_1)\mathbf{:+}\;\mathbf{int}.z_2, \sigma) \rightarrow (int(n', +, z_2), \sigma)} \quad (104)$$

$$\frac{z_1 \leq z_2}{(int(n, +, z_1)\mathbf{:-}\;\mathbf{int}.z_2, \sigma) \rightarrow (int(n', -, z_2), \sigma)} \quad (105)$$

$$\frac{z_1 > z_2 \; truncate(n, z_2) = n'}{(int(n, +, z_1)\mathbf{:+}\;\mathbf{int}.z_2, \sigma) \rightarrow (int(n', +, z_2), \sigma)} \quad (106)$$

Read Method:

$$\frac{(e, \sigma) \rightarrow (e', \sigma')}{(\mathbf{read.}T(e), \sigma) \rightarrow (\mathbf{read.}T(e'), \sigma')} \quad (107)$$

$$\frac{\begin{array}{c} lookupReadMethod(O, T) = s \\ l = newLocation(\sigma) \\ \{l \mapsto operand(n, O)\}\sigma = \sigma' \end{array}}{(\mathbf{read.}T(operand(n, O)), \sigma) \rightarrow (\lceil s[this := l]\rfloor, \sigma')} \quad (108)$$

$$\frac{\begin{array}{c} lookupReadMethod(O, T) = s \\ l = newLocation(\sigma) \\ \{l \mapsto operand([x_i \mapsto l_i], O)\}\sigma = \sigma' \end{array}}{(\mathbf{read.}T(operand([x_i \mapsto l_i], O)), \sigma) \rightarrow (\lceil s[this := l, x_i := l_i]\rfloor, \sigma')} \quad (109)$$

Subroutine Call (Expression):

$$\frac{(e_1, \sigma) \rightarrow (e_1', \sigma')}{(f(e_1, e_2, \ldots, e_n), \sigma) \rightarrow (f(e_1', e_2, \ldots, e_n), \sigma')} \quad (110)$$

$$\frac{(e_2, \sigma) \to (e_2', \sigma')}{(f(v_1, e_2, \ldots, e_n), \sigma) \to (f(v_1, e_2', \ldots, e_n), \sigma')} \tag{111}$$

$$\frac{(e_n, \sigma) \to (e_n', \sigma')}{(f(v_1, v_2, \ldots, e_n), \sigma) \to (f(v_1, v_2, \ldots, e_n'), \sigma')} \tag{112}$$

$$\frac{\begin{array}{c} lookupSubroutine(f) = [x_1, x_2, \ldots, x_n, s] \\ l_i = newLocation(\sigma) \\ \{l_i \mapsto v_i\}\sigma = \sigma' \end{array}}{(f(v_1, v_2, \ldots, v_n), \sigma) \to (\lceil s[x_i := l_i] \rfloor, \sigma')} \tag{113}$$

External Subroutine Call (Expression):

$$\frac{\begin{array}{c} lookupSubroutine(f) = external \\ f(\sigma, v_1, v_2, \ldots, v_n) = (v_r, \sigma') \end{array}}{(f(v_1, v_2, \ldots, v_n), \sigma) \to (v_r, \sigma')} \tag{114}$$

Location (variable) reference:

$$\frac{\sigma(l) = v}{(l, \sigma) \to (v, \sigma)} \tag{115}$$

Statement Expression:

$$\frac{\langle s, \sigma \rangle \to \langle s', \sigma' \rangle}{(\lceil s \rfloor, \sigma) \to (\lceil s' \rfloor, \sigma')} \tag{116}$$

$$(\lceil \textbf{return } v; \rfloor, \sigma) \to (v, \sigma) \tag{117}$$

If:

$$\frac{(e, \sigma) \to (e', \sigma')}{\langle \textbf{if } (e) \ s, \sigma \rangle \to \langle \textbf{if } (e') \ s, \sigma' \rangle} \tag{118}$$

$$\langle \textbf{if } (\textbf{true}) \ s, \sigma \rangle \to \langle s, \sigma \rangle \tag{119}$$

$$\langle \textbf{if } (\textbf{false}) \ s, \sigma \rangle \to \langle \textbf{skip;}, \sigma \rangle \tag{120}$$

If-Else:

$$\frac{(e, \sigma) \to (e', \sigma')}{\langle \textbf{if } (e) \ s_1 \textbf{ else } s_2, \sigma \rangle \to \langle \textbf{if } (e') \ s_1 \textbf{ else } s_2, \sigma' \rangle} \tag{121}$$

$$\langle \textbf{if } (\textbf{true}) \ s_1 \textbf{ else } s_2, \sigma \rangle \to \langle s_1, \sigma \rangle \tag{122}$$

$$\langle \textbf{if } (\textbf{false}) \ s_1 \textbf{ else } s_2, \sigma \rangle \to \langle s_2, \sigma \rangle \tag{123}$$

Assignment:

$$\frac{(e_1, \sigma) \to (e_1', \sigma')}{\langle e_1 = e_2;, \sigma \rangle \to \langle e_1' = e_2;, \sigma' \rangle} \tag{124}$$

$$\frac{(e, \sigma) \to (e', \sigma')}{\langle l = e;, \sigma\rangle \to \langle l = e';, \sigma'\rangle} \tag{125}$$

$$\langle l = v;, \sigma\rangle \to \langle \textbf{skip;}, \{l \mapsto v\}\sigma\rangle \tag{126}$$

Integer Assignment: (Implicit widening)

$$\frac{\sigma(l) = int(n, z, g)}{\langle l = int(n', z', g');, \sigma\rangle \to \langle \textbf{skip;}, \{l \mapsto int(n', z, g)\}\sigma\rangle} \tag{127}$$

Bit Assignment:

$$\frac{(e_1, \sigma) \to (e_1', \sigma')}{\langle l[e_1] = e_2;, \sigma\rangle \to \langle l[e_1'] = e_2;, \sigma'\rangle} \tag{128}$$

$$\frac{(e, \sigma) \to (e', \sigma')}{\langle l[int(n, g, z)] = e;, \sigma\rangle \to \langle l[int(n, g, z)] = e';, \sigma'\rangle} \tag{129}$$

$$\frac{\begin{array}{c}\sigma(l) = int(n_2, g_2, z_2) \qquad\qquad 0 \le n_1 < z_2 \\ \text{the } n_1\text{th bit of } n_2 \text{ set to ((b) ? 1 : 0)} = n_2'\end{array}}{\langle l[int(n_1, g_1, z_1)] = \textbf{true};, \sigma\rangle \to \langle \textbf{skip;}, \{l \mapsto int(n_2', g_2, z_2)\}\sigma\rangle} \tag{130}$$

Bit Range Assignment:

$$\frac{(e, \sigma) \to (e', \sigma')}{\langle l[n_1 : n_2] = e;, \sigma\rangle \to \langle l[n_1 : n_2] = e';, \sigma'\rangle} \tag{131}$$

$$\frac{\begin{array}{c}\sigma(l) = int(n_4, g_4, z_4) \\ max(n_1, n_2) = hiBit \\ min(n_1, n_2) = loBit \\ signExtend(n_3, hiBit - loBit + 1) = n_3' \\ \text{the bits of } n_4 \text{ from } hiBit \text{ to } loBit \text{ replaced by } n_3 = n_4'\end{array}}{\langle l[n_1 : n_2] = int(n_3, g_3, z_3);, \sigma\rangle \to \langle \textbf{skip;}, \{l \mapsto int(n_4', g_4, z_4)\sigma\rangle} \tag{132}$$

Return:

$$\frac{(e, \sigma) \to (e', \sigma')}{(\textbf{return } e;, \sigma) \to (\textbf{return } e';, \sigma')} \tag{133}$$

Write Method:

$$\frac{(e_1, \sigma) \to (e_1', \sigma')}{\langle \textbf{write.}T(e_1, e_2);, \sigma\rangle \to \langle \textbf{write.}T(e_1', e_2);, \sigma'\rangle} \tag{134}$$

$$\frac{(e, \sigma) \to (e', \sigma')}{\langle \textbf{write.}T(o, e);, \sigma\rangle \to \langle \textbf{write.}T(o, e');, \sigma'\rangle} \tag{135}$$

$$\frac{\begin{array}{c} lookupWriteMethod(O,T) = s \\ l_t = newLocation(\sigma) \\ l_v = newLocation(\sigma) \\ s[this := l_t, value := l_v] = s' \\ \{l_t \mapsto operand(n,O)\}\sigma = \sigma' \end{array}}{\langle \textbf{write.}T(operand(n,O),v);,\sigma\rangle \rightarrow \langle s',\sigma'\rangle} \tag{136}$$

$$\frac{\begin{array}{c} lookupWriteMethod(O,T) = s \\ l_t = newLocation(\sigma) \\ l_v = newLocation(\sigma) \\ s[this := l_t, value := l_v, x_i \mapsto l_i] = s' \\ \{l_t \mapsto operand([x_i \mapsto l_i],O), l_v \mapsto v\}\sigma = \sigma' \end{array}}{\langle \textbf{write.}T(operand([x_i \mapsto l_i],O),v);,\sigma\rangle \rightarrow \langle s',\sigma'\rangle} \tag{137}$$

Subroutine Call (Statement):

$$\frac{(e_1,\sigma) \rightarrow (e_1',\sigma')}{(f(e_1,e_2,\ldots,e_n);,\sigma) \rightarrow (f(e_1',e_2,\ldots,e_n);,\sigma')} \tag{138}$$

$$\frac{(e_2,\sigma) \rightarrow (e_2',\sigma')}{(f(v_1,e_2,\ldots,e_n);,\sigma) \rightarrow (f(v_1,e_2',\ldots,e_n);,\sigma')} \tag{139}$$

$$\frac{(e_n,\sigma) \rightarrow (e_n',\sigma')}{(f(v_1,v_2,\ldots,e_n);,\sigma) \rightarrow (f(v_1,v_2,\ldots,e_n');,\sigma')} \tag{140}$$

$$\frac{\begin{array}{c} lookupSubroutine(f) = \{x_1,x_2,\ldots,x_n,s\} \\ l_i = newLocation(\sigma) \\ \{x_i \mapsto v_i\}\sigma = \sigma' \end{array}}{\langle f(v_1,v_2,\ldots,v_n);,\sigma\rangle \rightarrow \langle s[x_i := l_i],\sigma'\rangle} \tag{141}$$

External Subroutine Call (Statement):

$$\frac{\begin{array}{c} lookupSubroutine(f) = external \\ f(\sigma,v_1,v_2,\ldots,v_n) = \sigma' \end{array}}{\langle f(v_1,v_2,\ldots,v_n);,\sigma\rangle \rightarrow \langle \textbf{skip;},\sigma'\rangle} \tag{142}$$

Local Declaration:

$$\frac{(e,\sigma) \rightarrow (e',\sigma')}{\langle \textbf{local } x\textbf{:}T = e;,\sigma\rangle \rightarrow \langle \textbf{local } x\textbf{:}T = e';,\sigma'\rangle} \tag{143}$$

$$\langle \textbf{local } x\textbf{:}T = v;,\sigma\rangle \rightarrow \langle \textbf{skip;},\sigma\rangle \tag{144}$$

$$\frac{l = newLocation(\sigma)}{\langle\{\textbf{local } x\textbf{:}T = v;\ sl\}\},\sigma\rangle \rightarrow \langle sl[x := l],\{l \mapsto v\}\sigma\rangle} \tag{145}$$

Block:

$$\langle \{\}, \sigma \rangle \rightarrow \langle \textbf{skip;}, \sigma \rangle \tag{146}$$

$$\langle \{s\}, \sigma \rangle \rightarrow \langle s, \sigma \rangle \tag{147}$$

$$\frac{\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle}{\langle \{s\ sl\}, \sigma \rangle \rightarrow \langle \{s'\ sl\}, \sigma' \rangle} \tag{148}$$

$$\langle \{\textbf{skip;}\ sl\}, \sigma \rangle \rightarrow \langle sl, \sigma \rangle \tag{149}$$

Error Generation & propogation:

$$(int(n_1, g_1, z_1) int(0, g_2, z_2)\ /\ , \sigma) \rightarrow (error, \sigma) \tag{150}$$

$$(\lceil \textbf{skip;} \rfloor, \sigma) \rightarrow (error, \sigma) \tag{151}$$

$$\frac{\sigma(r) = m\ m(v) = error}{(r[v], \sigma) \rightarrow (error, \sigma)} \tag{152}$$

$$\frac{\begin{array}{c} lookupSubroutine(f) = external \\ f(\sigma, v_1, v_2, \ldots, v_n) = error \end{array}}{(f(v_1, v_2, \ldots, v_n), \sigma) \rightarrow (error, \sigma)} \tag{153}$$

$$\frac{\begin{array}{c} lookupSubroutine(f) = external \\ f(\sigma, v_1, v_2, \ldots, v_n) = error \end{array}}{\langle f(v_1, v_2, \ldots, v_n); , \sigma \rangle \rightarrow \langle error, \sigma \rangle} \tag{154}$$

$$\frac{\neg(0 \leq n_2 < z_1)}{(int(n_1, g_1, z_1)[int(n_2, g_2, z_2)], \sigma) \rightarrow (error, \sigma)} \tag{155}$$

There are also rules for propogating errors, however we omit them for brevity.

A.4 Type Soundness

Lemma 1 (Substitution).
If $\{x \mapsto T_1\}\Gamma, \Psi, M \vdash s$ *where* $\Psi(l) = T_1$ *then* $\Gamma, \Psi, M \vdash s[x := l]$.
If $\{x \mapsto T_1\}\Gamma, \Psi \vdash e : T_2$ *where* $\Psi(l) = T_1$ *then* $\Gamma, \Psi \vdash e[x := l] : T_2$.

Proof: We proceed by induction n the structure of s or e.

– Rule (7): We have $e \equiv e_1 + e_2$, and $(e_1 + e_2)[x := v] \equiv e_1[x := v] + e_2[x := v]$. The last step in the derivation of $\{x \mapsto T_1\}\Gamma, \Psi \vdash e : T_2$ is of the form:

$$\frac{\begin{array}{c} \{x \mapsto T_1\}\Gamma, \Psi \vdash e_1 : g_1\ \textbf{int}.z_1 \\ \{x \mapsto T_1\}\Gamma, \Psi \vdash e_2 : g_2\ \textbf{int}.z_2 \\ g_1 \sqcup g_2 = g_3 \\ max(z_1, z_2) + 1 = z_3 \end{array}}{\{x \mapsto T_1\}\Gamma, \Psi \vdash e_1 + e_2 : g_3\ \textbf{int}.z_3}$$

From the induction hypothesis we have $\Gamma, \Psi \vdash e_1[x := l] : g_1\,\mathbf{int}.z_1$ and $\Gamma, \Psi \vdash e_2[x := l] : g_2\,\mathbf{int}.z_2$, and from Rule (7), we get $\Gamma, \Psi \vdash e_1[x := l] + e_2[x := l] : g_3\,\mathbf{int}.z_3$.

- Rule (8) - (18): These rules are similar to the Addition Rule and follow the same reasoning.
- Rule (19): We have $e \equiv x_1.x_2$, where $x_1.x_2$ is an enum value, then x does not appear in e and $e[x := l]$ has no effect on the expression.
- Rule (20): We have $e \equiv x.x_i$ where e is not an enum identifier. The last step of the derivation is of the form:

$$\frac{\{x \mapsto T_1\}\Gamma, \Psi \vdash x : O \; O = \mathbf{COperand}(ID, [x_i \mapsto O_i])}{\{x \mapsto T_1\}\Gamma, \Psi \vdash x.x_i : O_i}$$

  The substitution only applies to x and not x_i. By induction we have $tExprx[x := l]O$ which still satisfies the conditions for $\Gamma, \Psi \vdash x.x_i[x := l] : O_i$.
- Rule (21): We have $e \equiv e_1[n_1 : n_2]$ and $(e_1[i_1 : i_2])[x := l] \equiv (e_1)[x := l][n_1 : n_2]$ where the last step of the derivation is of the form:

$$\frac{\begin{array}{c}\{x \mapsto T_1\}\Gamma, \Psi \vdash e_1 : g\,\mathbf{int}.z \\ max(n_1, n_2) = hiBit \qquad min(n_1, n_2) = loBit \\ 0 \leq loBit < hiBit < z\end{array}}{\{x \mapsto T_1\}\Gamma, \Psi \vdash e_1[n_1 : n_2] : +\,\mathbf{int}.(hiBit - loBit + 1)}$$

  By induction we have $\Gamma, \Psi \vdash e_1[x := l] : g\,\mathbf{int}.z$. Because the substitution has no effect on n_1, n_2, the other two conditions hold and thus $\Gamma, \Psi \vdash (e_1[n_1 : n_2])[x := l] : +\,\mathbf{int}.(hiBit - loBit + 1)$.
- Rule (22): We have $e \equiv e_1[e_2]$ and $(e_1[e_2])[x := l] \equiv (e_1)[x := l][(e_2)[x := l]]$. The last step of the derivation is of the form:

$$\frac{\{x \mapsto T_1\}\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \; \{x \mapsto T_1\}\Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2}{\{x \mapsto T_1\}\Gamma, \Psi \vdash e_1[e_2] : \mathbf{boolean}}$$

  By induction we have that $\Gamma, \Psi \vdash e_1[x := l] : g_1\,\mathbf{int}.z_1$ and $\Gamma, \Psi \vdash e_2 : x := l]g_2\,\mathbf{int}.z_3$, which is sufficient for $\Gamma, \Psi \vdash (e_1[e_2])[x := l] : \mathbf{boolean}$.
- Rule (23): We have $e \equiv e_1[e_2]$ and $(e_1[e_2])[x := l] \equiv (e_1)[x := l][(e_2)[x := l]]$. The last step of the derivation is of the form:

$$\frac{\begin{array}{c}\{x \mapsto T\}\Gamma, \Psi \vdash e_1 : \mathbf{map}\langle T_1, T_2\rangle \\ \{x \mapsto T\}\Gamma, \Psi \vdash e_2 : T_1' \\ isAssignable(T_1, T_1')\end{array}}{\{x \mapsto T\}\Gamma, \Psi \vdash e_1[e_2] : T_2}$$

  By induction we have $\Gamma, \Psi \vdash e_1[x := l] : \mathbf{map}\langle T_1, T_2\rangle$ and $\Gamma, \Psi \vdash e_2[x := l] : T_1'$. Since the types remain the same, the isAssignable relation still holds and thus $\Gamma, \Psi \vdash e_1[e_2][x := l] : T_2$.
- Rule (24): We have $e \equiv e\mathbf{:}T$ and $(e\mathbf{:}T)[x := l] \equiv (e[x := T])\mathbf{:}T$. The last derivation step is of the form:

$$\frac{\{x \mapsto T_1\}\Gamma, \Psi \vdash e : T}{\{x \mapsto T_1\}\Gamma, \Psi \vdash e\mathbf{:}T : T}$$

By induction we have that $\Gamma, \Psi \vdash e[x := l] : T$ and thus $\Gamma, \Psi \vdash e\mathbf{:}T[x := l] : T$.

- Rule (25) - (27): These versions of the conversion typing rule follow under the same arguments as Rule (24).
- Rule (28): We have $e \equiv \mathbf{read.}T(x)$ and $(\mathbf{read.}T(y))[x := l] \equiv \mathbf{read.}T(y[x := l])$. The last step of the derivation is of the form:

$$\frac{\{x \mapsto T_1\}\Gamma, \Psi \vdash y : O \quad readMethodExists(O, T)}{\{x \mapsto T_1\}\Gamma, \Psi \vdash \mathbf{read.}T(y) : T}$$

  By induction we have that $\Gamma, \Psi \vdash y : O$. If O satisfied the readMethodExists condition before the substitution, it will still be satisfied and thus $\Gamma, \Psi \vdash \mathbf{read.}T(y)[x := l] : T$.
- Rule (29): We have $e \equiv f(e_1, e_2, \ldots, e_n)$ and $f(e_i)[x := l] \equiv f(e_i[x := l])$. The last step of the derivation is of the form:

$$\frac{lookupSubroutine(f) = \{T_r, T_1, T_2, \ldots, T_n\} \quad \{x \mapsto T_x\}\Gamma, \Psi \vdash e_i : T_i' \quad isAssignable(T_i, T_i') \quad T_r \neq \mathbf{void}}{\{x \mapsto T_x\}\Gamma, \Psi \vdash f(e_1, e_2, \ldots, e_n) : T_r}$$

  By induction $\Gamma, \Psi \vdash e_i[x := l] : T_i'$ for each of the e_i. The substitution has no affect on the lookupSubroutine condition and thus returns the same set of types. Each of the T_i's are also the same as before so the isAssignable relation still holds. The final condition regarding the return type not being void also remains true. Thus $\Gamma, \Psi \vdash f(e_1, e_2, \ldots, e_n)[x := l] : T_r$.
- Rule (30): We have $e \equiv y$. There are two cases where $x \equiv y$ and $x \not\equiv y$. If $x \equiv y$, then we also have that $T_1 = T_2$ by Rule (30). Additionally we have that $e[x := l] \equiv l$, which by Rule (31), $\Gamma, \Psi \vdash l : T_1$ because $\Psi(l) = T_1$, and then $\Gamma, \Psi \vdash l : T_2$ because $T_1 = T_2$. If $x \not\equiv y$, we have that $\Gamma, \Psi \vdash y : T_2$ as the substitution yielded the same expression as before.
- Rule (31): We have $e \equiv l'$. Because $l'[x := l] = l'$, it still retains the type before the substitution.
- Rule (32) - (33): Substitution yields the same initial expression and thus has the same type as before substitution.
- Rule (34): We have that $e \equiv \lceil s \rfloor$ and $\lceil s \rfloor[x := l] \equiv \lceil s[x := l] \rfloor$. The last step of the derivation is of the form:

$$\frac{\{x \mapsto T_x\}\Gamma, \Psi, T \vdash s}{\{x \mapsto T_x\}\Gamma, \Psi \vdash \lceil s \rfloor : T}$$

  By induction we have that $\Gamma, \Psi, T \vdash s[x := l]$ which means that $\Gamma, \Psi \vdash \lceil s \rfloor[x := l] : T$.
- Rule (35): We have $s \equiv \mathbf{if}\ (e_1)\ s_1$ and $(\mathbf{if}\ (e_1)\ s_1)[x := l] \equiv \mathbf{if}\ ((e_1)[x := l])\ s_1[x := l]$. From the hypothesis we have the $\{x := T_1\}\Gamma, \Psi \vdash e_1 : \mathbf{boolean}$ from Rule (35), then by induction we have $\Gamma, \Psi \vdash e_1[x := l] : \mathbf{boolean}$.

Additionally we have that $\{x := T_1\}\Gamma, \Psi, M \vdash s_1$, and also by induction $\Gamma, \Psi, M \vdash s_1[x := l]$. With these facts and Rule (35) we have that $\Gamma, \Psi, m \vdash$ (**if** (e_1) $s_1)[x := T_1]$.

- Rule (36): This rule is similar to Rule(35) and falls under similar reasoning.
- Rule (37): We have $s \equiv lv = e_1;$ and $(lv = e_1;)[x := l] \equiv lv[x := l] = e_1[x := l];$. The lv term is a subset of expressions so we may use the inductive hypothesis to get that $\Gamma, \Psi \vdash lv[x := l] : T_2$. Similarly by induction, we have that $\Gamma, \Psi \vdash e_1[x := l] : T_2'$. Since the types haven't changed, the isAssignable condition still holds and thus $\Gamma, \Psi, M \vdash lv = e_1; [x := l]$.
- Rule (38) - (39): These rules follow from both the assignement rule (37) and the repsective index rules (22), (23) for similar reasoning.
- Rule(40): Follows under similar reasoning for Rule (21 and (37).
- Rule (41): This rule is also similar to the assignment rule and follows similar reasoning.
- Rule (42): This rule is similar to the read method rule and falls under similar reasoning.
- Rule (43): This rule is similar to the expression subroutine call and follows under similar reasoning.
- Rule (44): We have $s \equiv$ **local** y**:**$T = e;$ and (**local** y**:**$T = e;)[x := l] \equiv$ **local** y**:**$T = e[x := l];$. The last step of the derivation is of the form:

$$\frac{y \notin \Gamma \ \{x \mapsto T_x\}\Gamma, \Psi \vdash e : T' \ isAssignable(T, T')}{\{x \mapsto T_x\}\Gamma, \Psi, M \vdash \textbf{local } y\textbf{:}T = e;}$$

  By induction we have that $\Gamma, \Psi \vdash e[x := l] : T'$, which still maintains the other conditions. The $y \notin \Gamma$ condition means future subsequent substitutions do not interfere with the current substitution. Since all the preconditions still hold, we also have $\Gamma, \Psi, M \vdash$ **local** y**:**$T = e; [x := l]$.
- Rule (45): This rule follows under similar reasoning to the other local declaration rule.
- Rule (46): There is nothing to substitute; an empty block always type checks.
- Rule (47): We have $s \equiv \{s\}$ and $(\{s\})[x := l] \equiv \{s[x := l]\}$. The last step of the derivation is of the form:

$$\frac{\{x \mapsto T\}\Gamma, \Psi, M \vdash s}{\{x \mapsto T\}\Gamma, \Psi, M \vdash \{s\}}$$

  By induction we have that $\Gamma, \Psi, M \vdash s[x := l]$ and thus we have $\Gamma, \Psi, M \vdash \{s\}[x := l]$
- Rule (48): Similar to the above rule, but also invokes induction to apply substitution to sl.
- Rule (49),(50),(51): Substitution has no effect on these expressions and statements and always type check.

Theorem 1 (Type Preservation). *If* $\vdash \langle s, \sigma\rangle$ *and there exists an* s' *and* σ' *such that* $\langle s, \sigma\rangle \rightarrow \langle s', \sigma'\rangle$*, then* $\vdash \langle s', \sigma'\rangle$*. If* $\vdash (e, \sigma) : T$ *and there exists an* e' *and* σ' *such that* $(e, \sigma) \rightarrow (e', \sigma')$*, then* $\vdash (e', \sigma') : T$*.*

Proof: We proceed by induction on the structure of the derivation of $\Gamma, \Psi \vdash e : T$ or $\Gamma, \Psi, M \vdash s$. There are now 46 subcases depending on which one of the Rules (1) - (46) was the last one used in the derivation of $\Gamma, \Psi \vdash e : T$ or $\Gamma, \Psi, M \vdash s$. Since we already have that $\vdash (e, \sigma) : T$, we also have that $\emptyset, \Psi \vdash e : T$, where e has all variable references removed by Rule (53). Similarly if we have that $\vdash \langle s, \sigma \rangle$ we also have $\emptyset, \Psi, T \vdash s$ for some T from Rule (54). These facts will apply every case that relies on the inductive hypothesis so we've skipped that step for each case analysis.

- Rule (7): We have $e \equiv e_1 + e_2$. There are three subcases depending on which one of Rules (55), (56), or (57) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (55),(56): In each case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (7).
  - Rule (57): We have $e \equiv int(n_1, g_1, z_1) + int(n_2, g_2, z_2)$ and $e' \equiv int(n_3, g_3, z_3)$ where $n_3 = n_1 + n_2$, $g_3 = g_1 \sqcup g_2$, and $z_3 = max(z_1, z_2) + 1$. The last part of the derivation of $\Gamma, \Psi \vdash e : T$ is of the form $\Gamma, \Psi \vdash int(n_1, g_1, z_1) + int(n_2, g_2, z_2) : g_3\, \mathbf{int}.z_3$ and from Rule (2) we also have $\Gamma, \Psi \vdash int(n_3, g_3, z_3) : g_3\, \mathbf{int}.z_3$.
- Rule (8): Similar to Rule (57). We have $e \equiv e_1 - e_2$. There are three subcases depending on which one of Rules (55), (56), or (58) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (55),(56): In each case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (8).
  - Rule (58): We have $e \equiv int(n_1, g_1, z_1) - int(n_2, g_2, z_2)$ and $e' \equiv int(n_3, +, z_3)$ where $n_3 = n_1 - n_2$, and $z_3 = max(z_1, z_2) + 1$. The last part of the derivation of $\Gamma, \Psi \vdash e : T$ is of the form $\Gamma, \Psi \vdash int(n_1, g_1, z_1) - int(n_2, g_2, z_2) : +\,\mathbf{int}.z_3$, and from Rule (2) we have $\Gamma, \Psi \vdash int(n_3, +, z_3) : +\,\mathbf{int}.z_3$.
- Rule (9) - (10): These rules are similar to Rules (7) and (8).
- Rule (11): We have $e \equiv e_1\ \mathbf{op}\ e_2$ where $op \in \{\&, |, \hat{}\}$. There are 5 sub cases depending on which one of Rule (55), (56), (62), (63), or (64) was the last one used in the derivation $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (55),(56): In each case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (11).
  - Rule (62),(63),(64): We have $e \equiv int(n_1, g_1, z_1)\ \mathbf{op}\ int(n_2, g_2, z_2)$ where $op \in \{\&, |, \hat{}\}$ and $e' \equiv int(n_3, +, z_3)$ where $n_3 = n_1\ \mathbf{op}\ n_2$ and $z_3 = max(z_1, z_2)$. The last part of the derivation of $\Gamma, \Psi \vdash e : T$ is of the form $\Gamma, \Psi \vdash e_1\ \mathbf{op}\ e_2 : +\,\mathbf{int}.z_3$, and from Rule (2) we have $\Gamma, \Psi \vdash int(n_3, +, z_3) : +\,\mathbf{int}.z_3$.
- Rule (12): We have $e \equiv e_1\ \mathbf{op}\ e_2$ where $op \in \{\mathbf{and}, \mathbf{or}, \mathbf{xor}\}$. There are 5 sub cases depending on which one of Rule (55), (56), (65), (66), or (67) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (55),(56): In each case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (12).

- Rule (65),(66),(67): We have $e \equiv b_1$ **op** b_2 where $op \in \{\mathbf{and}, \mathbf{or}, \mathbf{xor}\}$ and $e' \equiv b_3$ where $b_3 = b_1$ **op** b_2. The last part of the derivation of $\Gamma, \Psi \vdash e : T$ is of the form $\Gamma, \Psi \vdash e_1$ **op** e_2 : **boolean**, and from Rule (32), we have $\Gamma, \Psi \vdash b_3$: **boolean**.

– Rule (13): We have $e \equiv e_1$ **op** e_2 where $op \in \{<<, >>\}$. We proceed on a case analysis of the various steps that may be used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (55),(56): Similar to the other expressions that utilize the binop steps, we have by induction that $\Gamma, \Psi \vdash e' : T$ and Rule (13).
  - Rule (68): We have $e \equiv int(n_1, g_1, z_1) << int(n_2, g_2, z_2)$ and $e' \equiv int(n_3, g_1, z_1)$. By Rule (2) we have $\Gamma, \Psi \vdash int(n_3, g_1, z_1) : g_1$ **int**.z_1.
  - Rule (69): The right shift step follows the same arguments as the left shift step.

– Rule (14): We have $e \equiv e_1$ **op** e_2 where $op \in \{<, \leq, \geq, >\}$. We have three cases to analyze base on which rule we proceed in $(e, \sigma) \rightarrow (e', \sigma')$.

– Rule (15) - (18): These rules are similar to the other binary rules.
  - Rule (55),(56): By induction and Rule (14) we have that $\Gamma, \Psi \vdash e' : T$.
  - Rule (70): We have $e \equiv int(n_1, g_1, z_1)$ **op** $int(n_2, g_2, z_2)$ where $op \in \{< , \leq, ge, >\}$ and $e' \equiv b$. From rule (14), $\Gamma, \Psi \vdash e$: **boolean** which matches $\Gamma, \Psi \vdash b$: **boolean** from Rule (32).

– Rule (22): We have $e \equiv e_1[e_2]$. There are five subcases depending on which one of Rule (86), (87), (88), (89), or (155) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (86),(87): In each case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (22).
  - Rule (88), (89): We have $e \equiv int(n_1, g_1, z_1)[int(n_2, g_2, z_2)]$ where $0 \leq n_2 < z_1$ and $e' \equiv b$ is whether the n_2th bit of n_1 is either 0 or 1. The last part of the derivation of $\Gamma, \Psi \vdash e : T$ is of the form $\Gamma, \Psi \vdash e_1[e_2]$: **boolean**, and from Rule (32), we have $\Gamma, \Psi \vdash b$: **boolean**.
  - Rule (155): Error expressions always have the appropriate type for the surrounding context.

– Rule (23): We have $e \equiv e_1[e_2]$. There are three subcases depending on which one of Rule (86), (87), (90), or (152) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (86),(87): In each case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (23).
  - Rule (90): We have $e \equiv map(T_1, T_2)[v]$ and $e' \equiv v'$ where v' is of type T_2 for the following reasons. There is only one ways to insert values into a map through means within the language, Rule (126), where l results from Rule (90). Since the language does not allow access to any of the references the only way to get map locations are through map indexing operations. Under these conditions, which are covered by Rule (39) for type checking map assignments, the assignment to maps only contain values of type T_2. The other condition is external to our language, but using the Java type system also enforces that if the map index operation returns a value, it is of type T_2.

- Rule (152): Error expressions always have the appropriate type for the surrounding context.

– Rule (21): We have $e \equiv e_1[n_1 : n_2]$. There are two subcase depending on which one of Rule (91) or (92) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (91): In this case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (21).
  - Rule (92): We have $e \equiv e_1[n_1 : n_2]$ and $e' \equiv int(n, +, z)$, where $z = |n_1 - n_2| + 1$. The last part of the derivation of $\Gamma, \Psi \vdash e : T$ is of the form $\Gamma, \Psi \vdash e_1[n_1 : n_2] : +\mathbf{int}.z$, and from Rule (2) we have $\Gamma, \Psi \vdash int(n, +, z) :$.

– Rule (24 - 27): All typing rules have the resultant type as the cast type specified and the each of the conversion semantics Rules (94 - 106) step to a value of the designated type and thus are trivially true. In the case of Rule (93), the induction hypothesis and the same rule used to type the expression in the previous step may be used to preserve the type.
  - Rule (93): In this case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (27).
  - Rule (98): We have $e \equiv e{:}T$ where $T \equiv g\,\mathbf{int}.z, e \equiv enum(ID, n)$ and $e' \equiv int(n, g, z)$ and thus we have $\Gamma, \Psi \vdash e' : int(g, z,)$ by Rule (2).

– Rule (29): We have $e \equiv f(e_1, e_2, \ldots, e_n)$. There are five possible subcases depending on which one of Rule (110), (111), (112), (113), or (114) was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (110),(111), (112): Under the induction hypothesis and Rule (29) each has the same type.
  - Rule (113): We have $e \equiv f(v_1, v_2, \ldots, v_n)$. By the induction hypothesis we have $\Gamma, \Psi \vdash e : T$ and Rule (29) we have that the body of the subroutine type checks under the implication of the *lookupSubroutine*() condition. We have $e' \equiv \lceil s[x_i := l_i] \rfloor$ where $s[x_i := l_i]$ type checks under in accordance with Lemma 1. Given that $s[x_i := l_i]$ type checks, then $\Gamma, \Psi \vdash \lceil s[x_i := l_i] \rfloor : T$ as e'.
  - Rule (114): We rely on the Java type system to ensure that values of the correct form are returned. External subroutines may also return error in which that also type checks by the error rule.

– Rule (30): Variables do not appear in the semantics; they are replaced by locations through the substitution process.

– Rule (31): We have $e \equiv l$. We may only step by Rule (115). By the induction hypothesis, we have that l has some type T. We have $e' \equiv v$ that also has some type T.

– Rule (34): We have $e \equiv \lceil s \rfloor$. There are three possible subcases depending on which conversion Rule was the last one used in the derivation of $(e, \sigma) \rightarrow (e', \sigma')$.
  - Rule (116): In this case, $\Gamma, \Psi \vdash e : T$ from the induction hypothesis and Rule (34).

  - Rule (117): We have $e \equiv \lceil \textbf{return } v; \rfloor$ and $e' \equiv v$. By Rule (34) and (41) we have that v of type T which conforms to $\Gamma, \Psi \vdash e : T$.
  - Rule (151): Error expressions always type check.
- Rule (35): We have $s \equiv \textbf{if } (e_1)\ s_1$. There are three cases depending on which one of Rule (118), (119), or (120) was the last one used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
  - Rule (118): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (35) we have $\Gamma, \Psi, M \vdash \textbf{if } (e_1')\ s$.
  - Rule (119): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (35) we have $\Gamma, \Psi, M \vdash s_1$.
  - Rule (120): The skip statement always type checks, Rule (49).
- Rule (36): We have $s \equiv \textbf{if } (e_1)\ s_1\ \textbf{else } s_2$. There are three cases depending on which Rule (121), (122), or (123) was the last one used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
  - Rule (121): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (36) we have $\Gamma, \Psi, M \vdash \textbf{if } (e_1')\ s_1\ \textbf{else } s_2$.
  - Rule (122): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (36) we have $\Gamma, \Psi, M \vdash s_1$.
  - Rule (123): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (36) we have $\Gamma, \Psi, M \vdash s_2$.
- Rule (37): We have $s \equiv lv = e_1;$. There are three cases depending on which Rule (124), (125), or (126) was the last one used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$. There are other possibilities, but this an general assignment rule, and as such, the cases are only for generalized steps.
  - Rule (124): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (37), we have $\Gamma, \Psi, M \vdash lv' = e_1;$.
  - Rule (125): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (37), we have $\Gamma, \Psi, M \vdash l = e_1';$.
  - Rule (126): We have $s' \equiv \textbf{skip;}$. Skip statements always type check by Rule (49).
- Rule (38): We have $s \equiv lv[e_1] = e_2;$. There are five cases depending on which Rule (124), (128), (129), (130), or (155) was the last rule used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
  - Rule (124): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (38) we have $\Gamma, \Psi, M \vdash lv'[e_1] = e_2;$. This step results Rule (124) and (86).
  - Rule (128): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (38) we have $\Gamma, \Psi, M \vdash lv[e_1'] = e_2;$.
  - Rule (129): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (38) we have $\Gamma, \Psi, M \vdash lv[e_1] = e_2';$.
  - Rule (129): We have $s' \equiv \textbf{skip;}$ and skip statements always type check according to Rule (49).
  - Rule (155): We have $s' \equiv \mathit{error}$ and error statements also always type check according to Rule (51).

- Rule (39): The map assignment rule is only a special case of the general assignement rule. It uses the same general assignment rules to step and use similar reasoning for each case.
- Rule (40): We have $s \equiv lv[n_1 : n_2] = e_1;$. There are three cases depending on which of Rule (124), (131), or (132) was the last used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
  - Rule (124): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (40) we have $\Gamma, \Psi, M \vdash lv'[n_1 : n_2] = e_1;$. This is similar to the corresponding bit index assignment case.
  - Rule (131): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (40) we have $\Gamma, \Psi, M \vdash lv[n_1 : n_2] = e_1';$.
  - Rule (132): We have $s' \equiv$ **skip;** and skip statements always type check by Rule (49).
- Rule (41): We have $s \equiv$ **return** $e;$. There is only one possible s' such that $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ which is denoted by Rule (133). By the induction hypothesis and Rule (41) we have have $\Gamma, \Psi, M \vdash s'$. The base stepping case for return statements occurs in the statement expression construct.
- Rule (42): We have $s \equiv$ **write.**$T(e_1, e_2);$. There are four casesdepending on which one of Rule (134), (135), (136), or (137) was the last one used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
  - Rule (134): By induction hypothesis $\Gamma, \Psi, M \vdash s$ and Rule (42) we have $\Gamma, \Psi, M \vdash$ **write.**$T(e_1', e_2);$.
  - Rule (135): By induction hypothesis $\Gamma, \Psi, M \vdash s$ and Rule (42) we have $\Gamma, \Psi, M \vdash$ **write.**$T(e_1, e_2');$.
  - Rule (136): We have $s \equiv$ **write.**$T(operand(n, O), v);$. The $lookupWriteMethod()$ conditon implies that given that v conforms to T, the body s also type checks. Thus s' type checks.
  - Rule (137): Similar argument as above.
- Rule (43): We have $s \equiv f(e_1, e_2, \ldots, e_n);$. There are 5 cases depending on which rule was last used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
  - Rule (138): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (43) we have $\Gamma, \Psi, M \vdash f(e_1', e_2, \ldots, e_n);$.
  - Rule (139): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (43) we have $\Gamma, \Psi, M \vdash f(e_1, e_2', \ldots, e_n);$.
  - Rule (140): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (43) we have $\Gamma, \Psi, M \vdash f(e_1, e_2, \ldots, e_n');$.
  - Rule (141): We have $s \equiv f(v_1, v_2, \ldots, v_n);$ and under the definition of *lookupSubroutine* we have that the body of f given that the arguments are of the correct form. Under these conditions, the body of function $s[x_i := l]$ as s' also type checks.
  - Rule (142): We have s' as **skip;** which always type checks.
- Rule (44): We have $s \equiv$ **local** x**:**$T = e;$. There are two cases depending on whether Rule (143) or (144) was the last used in the derivation of $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.

  - Rule (143): By induction hypothesis, $\Gamma, \Psi, M \vdash s$ and Rule (44) we have $\Gamma, \Psi, M \vdash \textbf{local } x{:}T = e';$.
  - Rule (144): We have $s' \equiv \textbf{skip;}$ which always type checks according to Rule (49).
- Rule (45): We have $s \equiv \{\textbf{local } x{:}T = v;\ sl\}$. The cases where applying steps to $\{\textbf{local } x{:}T = e;\ sl\}$ are handled by the cases for (44) and (48). There is only one step possible by Rule (145).
- Rule (46): We have $s \equiv \{\}$. There is only one possible step by Rule (146), where $s' \equiv \textbf{skip;}$ which also typechecks by Rule (49).
- Rule (47): We have $s \equiv \{s_1\}$. There is only one possible step by Rule (147), where $s' \equiv s_1$. This type checks because the Rule (47) implies that if $\Gamma, \Psi, M \vdash \{s_1\}$ then $\Gamma, \Psi, M \vdash s_1$.
- Rule (48): We have $s \equiv \{s\ sl\}$. There are two possible steps by Rule (148) and (149). If we step using Rule (148) then by the the induction hypothesis and Rule (48), s' also type checks. If we step by Rule (149), $s' \equiv sl$ which type checks from the original invocation of Rule (48).

Lemma 2 (Value Type).

- *If* $\Gamma, \Psi \vdash v : \textbf{\textit{boolean}}$ *then* v *is of the form* b.
- *If* $\Gamma, \Psi \vdash v : g\ \textbf{\textit{int}}.z$ *then* v *is of the form* $int(n, g, z)$.
- *If* $\Gamma, \Psi \vdash v : \textbf{\textit{SOperand}}(ID, z, T)$ *then* v *is of the form* $operand(n, \textbf{\textit{SOperand}}(ID, z, T))$.
- *If* $\Gamma, \Psi \vdash v : \textbf{\textit{COperand}}(ID, [x_i \mapsto O_i])$ *then* v *is of the form* $operand([x_i \mapsto l_i], \textbf{\textit{COperand}}(ID, [x_i \mapsto O_i]))$.
- *If* $\Gamma, \Psi \vdash v : \textbf{\textit{Enum}}(ID, [x_1, x_2, \ldots, x_n])$ *then* v *is of the form* $enum(n, \textbf{\textit{Enum}}(ID, [x_1, x_2, \ldots, x_n]))$.
- *If* $\Gamma, \Psi \vdash v : \textbf{\textit{map}}\langle T_1, T_2\rangle$ *then* v *is of the form* $map(T_1, T_2)$.

Proof: It comes immediately from Rules (1, 2, 3, 4, 5, and 6).

Lemma 3 (Location Type).

- *If* $\Gamma, \Psi \vdash l : \textbf{\textit{boolean}}$ *and* $\vdash \sigma : \Psi$ *then* $\sigma(l)$ *is of the form* b.
- *If* $\Gamma, \Psi \vdash l : g\ \textbf{\textit{int}}.z$ *and* $\vdash \sigma : \Psi$ *then* $\sigma(l)$ *is of the form* $int(n, g, z)$.
- *If* $\Gamma, \Psi \vdash l : \textbf{\textit{SOperand}}(ID, z, T)$ *and* $\vdash \sigma : \Psi$ *then* $\sigma(l)$ *is of the form* $operand(n, \textbf{\textit{SOperand}}(ID, z, T))$.
- *If* $\Gamma, \Psi \vdash l : \textbf{\textit{COperand}}(ID, [x_i \mapsto O_i])$ *and* $\vdash \sigma : \Psi$ *then* $\sigma(l)$ *is of the form* $operand([x_i \mapsto l_i], \textbf{\textit{COperand}}(ID, [x_i \mapsto O_i]))$.
- *If* $\Gamma, \Psi \vdash l : \textbf{\textit{Enum}}(ID, [x_1, x_2, \ldots, x_n])$ *and* $\vdash \sigma : \Psi$ *then* $\sigma(l)$ *is of the form* $enum(n, \textbf{\textit{Enum}}(ID, [x_1, x_2, \ldots, x_n]))$.
- *If* $\Gamma, \Psi \vdash l : \textbf{\textit{map}}\langle T_1, T_2\rangle$ *and* $\vdash \sigma : \Psi$ *then* $\sigma(l)$ *is of the form* $map(T_1, T_2)$.

Proof: It comes immediately from Rule (52) and Lemma 2.

Theorem 2 (Progress). *If* $\vdash \langle s, \sigma \rangle$ *then either of the following is true:*
1) $s \equiv error$
2) $s \equiv$ ***skip;***
3) There exists an s' *and* σ' *such that* $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$.
Similarly for expressions, If $\vdash (e, \sigma)$ *then either of the following is true:*
1) $e \equiv error$
2) $e \equiv v$
3) $e \equiv l$
4) There exists an e' *and* σ' *such that* $(e, \sigma) \rightarrow (e', \sigma')$.

*Proof:*We proceed by induction on the structure of th derivation of $\Gamma, \Psi \vdash e : T$ or $\Gamma, \Psi, M \vdash s$. There are now n subcases depnding on which one of Rules (1) - (n) was the last used in the derivation of $\Gamma, \Psi \vdash e : T$ or $\Gamma, \Psi, M \vdash s$. For convenience we have already remove the program state notation, where the store σ is shown to have type Ψ and used for the current context in typing e, s.

- Rule (7): We have $e \equiv e_1 + e_2$. We have that e is closed, so e_1, e_2 is also closed. The last step in the derivation of $\Gamma, \Psi \vdash e : T$ is of the form:

$$\frac{\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \qquad g_1 \sqcup g_2 = g_3 \quad \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2 \; max(z_1, z_2) + 1 = z_3}{\Gamma, \Psi \vdash e_1 + e_2 : g_3\,\mathbf{int}.z_3}$$

  From the induction hypothesis we have that e_1 is either a value, or there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ and we have that e_2 is also either a value, or there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$. We proceed by case analysis.
  - If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$, then $(e_1 + e_2, \sigma) \rightarrow (e_1' + e_2, \sigma')$ by Rule (55).
  - If e_1 is a value and there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$, then $(e_1 + e_2, \sigma) \rightarrow (e_1 + e_2', \sigma')$ by Rule (56).
  - If e_1 is a value and e_2 is a value, then from $\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1, \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2$ and Lemma 2 we have that e_1, e_2 are of the form $int(n_1, g_1, z_1), int(n_2, g_2, z_2)$ and may step $(e_1 + e_2, \sigma) \rightarrow (int(n_3, g_3, z_3), \sigma)$ where $n_3 = n_1 + n_2$, $g_3 = g_1 \sqcup g_2$, and $z_3 = max(z_1, z_2) + 1$, by Rule (57).
- Rule (8) - (21): These rules fall under similar reasoning as Rule (7).
- Rule (22): We have $e \equiv e_1[e_2]$. We have that e is closed, so e_1, e_2 are also closed. The last step in the derivation of $\Gamma, \Psi \vdash e : T$ is of the form:

$$\frac{\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1 \; \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2}{\Gamma, \Psi \vdash e_1[e_2] : \mathbf{boolean}}$$

  From the induction hypothesis we have that e_1 is either a value, or there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ and e_2 is either a value, or there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$. We proceed by case analysis.
  - If e_1 is a value and e_2 is a value then from $\Gamma, \Psi \vdash e_1 : g_1\,\mathbf{int}.z_1, \Gamma, \Psi \vdash e_2 : g_2\,\mathbf{int}.z_2$ and Lemma 2 we have that e_1, e_2 ae of the form $int(n_1, g_1, z_1), int(n_2, g_2, z_2)$

and may step $(e_1[e_2], \sigma) \to (b, \sigma)$ using Rule (88) or (89). Additionally we may step $(e_1[e_2], \sigma) \to (error, \sigma)$ if n_2 is not a valid index using Rule (155).
  - If there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$, then $(e_1[e_2], \sigma) \to (e_1'[e_2], \sigma')$ by Rule (86).
  - If e_1 is a value and there exists e_2' such that $(e_2, \sigma) \to (e_2', \sigma')$, then $(e_1[e_2], \sigma) \to (e_1[e_2'], \sigma')$ by Rule (87).
- Rule (23): We have $e \equiv e_1[e_2]$. We have that e is closed, so e_1, e_2 are also closed. The last step in the derivation of $\Gamma, \Psi \vdash e : T$ is of the form:

$$\frac{\Gamma, \Psi \vdash e_1 : \mathbf{map}\langle T_1, T_2\rangle \ \Gamma, \Psi \vdash e_2 : T_1' \ isAssignable(T_1, T_1')}{\Gamma, \Psi \vdash e_1[e_2] : T_2}$$

From the induction hypothesis we have that e_1 is either a value, or there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$ and e_2 is either a value, or there exists e_2' such that $(e_2, \sigma) \to (e_2', \sigma')$. We proceed by case analysis.
  - If there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$ then we step $(e_1[e_2], \sigma) \to (e_1'[e_2], \sigma')$ by Rule (86).
  - If e_1 is a value and there exists e_2' such that $(e_2, \sigma) \to (e_2', \sigma')$, then we may step $(e_1[e_2], \sigma) \to (e_1[e_2'], \sigma')$ by Rule (87).
  - If e_1, e_2 are both values, then e_1 is of the form $map(T_1, T_2)$ by (2) and e_2 is a value of type T_1' such that $isAssignable(T_1, T_1')$. We may step $(e_1[e_2], \sigma) \to (l, \sigma')$ by Rule (90). We may additionally step $(e_1[e_2], \sigma) \to (error, \sigma)$ by Rule (152).
- Rule (24) - (27): Each of these rules are handled by the appropriate conversion steps. The various conversion cases (Rules (94) - (106)) handle these steps, in addition to the induction case where it is handled by Rule (93).
- Rule (28): Similar to the write rule; see Rule (42) below.
- Rule (29): See the case for Rule (43) below.
- Rule (34): We have $e \equiv \lceil s_1 \rfloor$. The last step in the derivation of $\Gamma, \Psi \vdash e : T$ is of the form:

$$\frac{\Gamma, \Psi, T \vdash s}{\Gamma, \Psi \vdash \lceil s \rfloor : T}$$

From the induction hypothesis we have that s is either **skip;**, or there exists s_1' such that $\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle$. We proceed by case analysis.
  - If s is **skip;**, then we proceed $(\lceil s_1 \rfloor, \sigma) \to (error, \sigma)$ by Rule (151).
  - If there exists s_1' such that $\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle$, then may step $(\lceil s_1 \rfloor, \sigma) \to (\lceil s_1' \rfloor, \sigma')$.
  - There is a special case if $s_1 \equiv$ **return** v; in which we may proceed $(\lceil$**return** $v; \rfloor, \sigma) \to (v, \sigma)$, by Rule (117).
- Rule (35): We have $s \equiv$ **if** (e_1) s_1. We have that e, s are closed, so e_1, s_1 are also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\Gamma, \Psi \vdash e : \mathbf{boolean} \ \Gamma, \Psi, M \vdash s}{\Gamma, \Psi, M \vdash \mathbf{if}\ (e)\ s}$$

From the induction hypothesis we have that e_1 is either a value, or there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$. We proceed by case analysis.

- If e_1 is a value, then from $\Gamma, \Psi \vdash e_1$: **boolean** and Lemma 2 we have that e_1 is either **true** or $vFalse$. If $e_1 \equiv$ **true** then we may step $\langle$**if** (e_1) $s_1, \sigma\rangle \rightarrow \langle s_1, \sigma\rangle$ by Rule (119). If $e_1 \equiv$ **false** then we may step $\langle$**if** (e_1) $s_1, \sigma\rangle \rightarrow \langle$**skip;**$, \sigma\rangle$ by Rule (120).
- If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$, then $\langle$**if** (e_1) $s_1, \sigma\rangle \rightarrow \langle$**if** (e_1') $s_1, \sigma'\rangle$ by Rule (118).

– Rule (36): We have $s \equiv$ **if** (e_1) s_1 **else** s_2. We have that s, e are closed, so e_1 is also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\Gamma, \Psi \vdash e : \textbf{boolean}\ \Gamma, \Psi, M \vdash s_1\ \Gamma, \Psi, M \vdash s_2}{\Gamma, \Psi, M \vdash \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2}$$

From the induction hypothesis we have that e_1 is either a value, or there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$. We proceed by case analysis.

- If e_1 is a value, then from $\Gamma, \Psi \vdash e_1$: **boolean** and Lemma 2 we have that e_1 is either **true** or **false**. If $e_1 \equiv$ **true** then we may step $\langle$**if** (e_1) s_1 **else** $s_2, \sigma\rangle \rightarrow \langle s_1, \sigma\rangle$ by Rule (122). If $e_1 \equiv$ **false** then we may step $\langle$**if** (e_1) s_1 **else** $s_2, \sigma\rangle \rightarrow \langle s_2, \sigma\rangle$ by Rule (123).
- If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$, then $\langle$**if** (e_1) s_1 **else** $s_2, \sigma\rangle \rightarrow \langle$**if** (e_1') s_1 **else** $s_2, \sigma'\rangle$ by Rule (121).

– Rule (37): We have $s \equiv e_1 = e_2;$. We have that s, e is closed, so e_1, e_2 are also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\Gamma, \Psi \vdash lv : T\ \Gamma, \Psi \vdash e : T'\ isAssignable(T, T')}{\Gamma, \Psi, M \vdash lv = e;}$$

From the induction hypothesis we have that e_1 is a value, or there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ and e_2 is a value, or there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$. We proceed by case analysis.

- If e_1 is a value is impossible. The grammar disallows expressions that generate value as an assignable. Additionally, if we have a location, we do not step to reveal the value of the location by Rule (125), (126).
- If e_1 is a location, and there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$, then we step $\langle e_1 = e_2;, \sigma\rangle \rightarrow \langle e_1 = e_2';, \sigma'\rangle$ by Rule (125).
- If e_1 is a location and e_2 is a value then we step $\langle e_1 = e_2;, \sigma\rangle \rightarrow \langle$**skip;**$, \{l \mapsto v\}\sigma\rangle$ by Rule (126).
- If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ then we step $\langle e_1 = e_2;, \sigma\rangle \rightarrow \langle e_1' = e_2;, \sigma'\rangle$ by Rule (124).

– Rule (38): We have $e \equiv e_1[e_2] = e_3;$. We have that s, e is closed, so e_1, e_2, e_3 are also closed. The last step of the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\Gamma, \Psi \vdash lv : g\,\textbf{int}.z\ \Gamma, \Psi \vdash e_1 : g_1\,\textbf{int}.z_1\ \Gamma, \Psi \vdash e_2 : \textbf{boolean}}{\Gamma, \Psi, M \vdash lv[e_1] = e_2;}$$

From the induction hypothesis we have that e_1, e_2, e_3 may either be a location, a value, or the exists a e_i' such that $(e_i, \sigma) \rightarrow (e_i', \sigma')$. We proceed by case analysis.

- If e_1 is a value is an impossible case. The grammar disallows expressions that generate a value as an assignable. Additionally if we have a location, we will not step to the value but use Rules (128), (129), and (130).
- If e_1 is a location, e_2 is a value, and e_3 is a value, then e_2 is of the form $int(n_i, g_i, z_i)$ by Rule (38) and Lemma 2 and e_3 has the form b by similar reasons. The location e_1 containse a value of the form $int(n_1, g_1, z_1)$ by the typing rule and Lemma 3. We may proceed using Rule (130) or (155).
- If e_1 is a location and there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$, then we my proceed using Rule (129).
- If e_1 is a location, e_2 is a value, and there exists e_3' such that $(e_3, \sigma) \rightarrow (e_3', \sigma')$, then e_2 is of the form $int(n, g, z)$ by Rule (38) and Lemma 2. We may proceed using the Rule (130).
- If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ then we proceed using Rule (124). This Rule is general enough to handle this case.

– Rule (39): This case is a special case of the assignment rule.
– Rule (40): We have $e \equiv e_1[n_1 : n_2] = e_2;$. We have that s, e is closed, so e_1, e_2 are also closed. The last step of the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\begin{array}{c}\Gamma, \Psi \vdash lv : g\,\mathbf{int}.z \qquad \Gamma, \Psi \vdash e : g'\,\mathbf{int}.z' \\ max(n_1, n_2) = hiBit \quad min(n_1, n_2) = loBit \\ 0 \leq loBit < hiBit < z\ z' \leq (hiBit - loBit)\end{array}}{\Gamma, \Psi, M \vdash l[n_1 : n_2] = e;}$$

From the induction hypothesis we have that e_1 is either a location, a value, or there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$, and similarly, so would e_2. We proceed by case analysis.

- If e_1 is a value is an impossible case. The grammar disallows expressions that generate a value as an assignable. Additionally if we have a location, we will not step to the value but use Rules (125), (131), and (132).
- If e_1 is a location and e_2 is a value, then by Lemma 2 and Rule (40) we have that e_2 is of the form $g\,\mathbf{int}.z$ as well as e_1 containing a value of the form $g'\,\mathbf{int}.z'$ by Lemma 3 and Rule (40). To fulfill the clause regarding the new value in the location l, this is ensured to have a well defined replacement by the assigning bit size condition in Rule (40). Thus by Rule (132) we proceed $\langle l[n_1 : n_2] = int(g, z, ;), \sigma\rangle \rightarrow \langle \mathbf{skip;}, \sigma'\rangle$. By the indices range condition, we ensure that there exists bits to replace for the stepping condition. By the size restrictions on the assigning integer we ensure that there will be no truncation in the bits.
- If e_1 is a location and there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$, then we step using Rule (131).
- If there exists a e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ then we proceed using Rule (124). This rule is general enough to capture this particular construct.

– Rule (41): We have $s \equiv \mathbf{return}\ e_1;$. We have that s, e are closed, so e_1 is also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\Gamma, \Psi \vdash e : M'\ isAssignable(M, M')}{\Gamma, \Psi, M \vdash \mathbf{return}\ e;}$$

From the induction hypothesis we have that e_1 is either a value, or there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$. We proceed by case analysis.

- There is no rule to step if e_1 is a value. Returns are handled in the expression statement context described above. Statements may only appear in four places: instruction execute bodies, read methods, write methods, and subroutines and are the only contexts in which return statements may be written. Statement expressions are an internal construct that transplants statements from read methods and subroutines that are used in an expression context so they do not apply in regards to appearance or return statements. In execute bodies, write bodies, and subroutines that have a void return type, we have the return type M context set to **void**. There is no way to generate a void value in this language so return statements in those contexts do not type check; see Rule (41). For read methods, they step using a statement expression, see Rule (108), (109) and thus any return statement will be handled by Rule (117). For subroutines with a read value, they may not be called in a statement context by Rule (43) and only used in an expressional context which wraps the body in a statement expression (113). Thus this case is not an issue as it is handled elsewhere.
- If there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$, then we may step $\langle \textbf{return } e_1;, \sigma \rangle \to \langle \textbf{return } e_1';, \sigma' \rangle$ using Rule (133).

– Rule (42): We have $s \equiv \textbf{write.}T(x, e_1);$. We have that s, e are closed, so e_1 is also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\begin{array}{cc} E(x) = O & writeMethodExists(O, T) \\ \Gamma, \Psi \vdash e : T' & isAssignable(T, T') \end{array}}{\Gamma, \Psi, M \vdash \textbf{write.}T(x, e);}$$

From the induction hypothesis we have that e_1 is either a value or there exists e_1' such that $(e_1, \sigma) \to (e_1', \sigma')$. We proceed by case analysis.

- If x is a value then it may either be of the form $operand(n, O)$ or $operand([x_i \mapsto l_i], O)$ by Rule (42) and Lemma 2 and e_1 is a value, it will have a form dictated by its type.

– Rule (43): We have $s \equiv f(e_1, e_2, \ldots, e_n);$. We have that s, e are closed, so $e_1, e_2, \ldots, e_n$ are also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{\begin{array}{c} lookupSubroutine(f) = \{\textbf{void}, T_1, T_2, \ldots, T_n\} \\ \Gamma, \Psi \vdash e_i : T_i' \\ isAssignable(T_i, T_i') \end{array}}{\Gamma, \Psi, M \vdash f(e_1, e_2, \ldots, e_n);}$$

From the induction hypothesis we have that e_i is either a value, or there exists e_i' such that $(e_i, \sigma) \to (e_i', \sigma')$. We proceed by case analysis.

- If $e_1, e_2, \ldots, e_n$ are values, then we may proceed using either Rule (141) or (142) depending on where the implementation resides. The $lookupSubroutine()$ condition in the typing rule guarantees that such a function exists. In

the case of Rule (141), we proceed with $\langle f(v_1, v_2, \ldots, v_n);, \sigma\rangle \rightarrow \langle s, \sigma'\rangle$. For Rule (142), we proceed with $\langle f(v_1, v_2, \ldots, v_n);, \sigma\rangle \rightarrow \langle \textbf{skip;}, \sigma'\rangle$.
  - If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$, then $\langle f(e_1, e_2, \ldots, e_n);, \sigma\rangle \rightarrow \langle f(e_1', e_2, \ldots, e_n);, \sigma'\rangle$ by Rule (138).
  - If e_1 is a value and there exists e_2' such that $(e_2, \sigma) \rightarrow (e_2', \sigma')$, then $\langle f(e_1, e_2, \ldots, e_n);, \sigma\rangle \rightarrow \langle f(e_1, e_2', \ldots, e_n);, \sigma'\rangle$ by Rule (139).
  - The remaining arguments proceed in a similar fashion to the two cases above.
- Rule (44): We have $s \equiv \textbf{local } x\textbf{:}T = e_1;$. We have s, e are closed, so e_1 is also closed. The last step in the derivation of $\Gamma, \Psi, M \vdash s$ is of the form:

$$\frac{x \notin E \qquad \Gamma, \Psi \vdash e : T' \qquad \{x \mapsto T\}\Gamma, \Psi, M \vdash sl \; isAssignable(T, T')}{\Gamma, \Psi, M \vdash \textbf{local } x\textbf{:}T = e;}$$

  From the induction hypothesis we have that e_1 is either a value or there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$. We proceed by case analysis.
  - If there exists e_1' such that $(e_1, \sigma) \rightarrow (e_1', \sigma')$ then we may step $\langle \textbf{local } x\textbf{:}T = e_1;, \sigma\rangle \rightarrow \langle \textbf{local } x\textbf{:}T = e_1';, \sigma'\rangle$ using Rule (143).
  - If e_1 is a value then we may proceed $\langle \textbf{local } x\textbf{:}T = v;, \sigma\rangle \rightarrow \langle \textbf{skip;}, \sigma\rangle$ using Rule (145).
- Rule (45): We have $s \equiv \textbf{local } x\textbf{:}T = e_1;\ sl$. This follows from the case above, except when e_1 is a value. We proceed by the following step $\langle \textbf{local } x\textbf{:}T = v;\ sl, \sigma\rangle \rightarrow \langle sl[x := l, \{l \mapsto v\}\sigma\rangle$ using Rule (145).
- Rule (46): We have $s \equiv \{\}$. There are no substatements or expressions, but steps to **skip;** by Rule (146).
- Rule (47): We have $s \equiv \{s_1\}$. This always steps to s_1 by Rule (147).
- Rule (48): We have $s \equiv \{s_1\ sl\}$. From the induction hypothesis we have that s_1 is either **skip;** or there exists s_1' such that $\langle s_1, \sigma\rangle \rightarrow \langle s_1', \sigma'\rangle$. We proceed by case analysis.
  - If $s_1 \equiv \textbf{skip;}$ then we step $\langle \{s_1\ sl\}, \sigma\rangle \rightarrow \langle sl, \sigma\rangle$ by Rule (149).
  - If there exists s_1' such that $\langle s_1, \sigma\rangle \rightarrow \langle s_1', \sigma'\rangle$ then we step $\langle \{s_1\ sl\}, \sigma\rangle \rightarrow \langle \{s_1'\ sl\}, \sigma'\rangle$ by Rule (148).

Theorem 3 (Soundness). *A well-typed program cannot go wrong.*

Proof: Suppose we have a well-typed program s and σ such that $\vdash \langle s, \sigma\rangle$. Suppose this program were able to go wrong – that there exists a stuck state $\langle s', \sigma'\rangle$ such that $\langle s, \sigma\rangle \rightarrow * \langle s', \sigma'\rangle$. From Theorem 1 we have that $\vdash \langle s', \sigma'\rangle$. From Theorem 2 we have that $\vdash \langle s', \sigma'\rangle$ may take a step or has completed execution, which is a contradiction and thus $\langle s', \sigma'\rangle$ cannot exist and $\langle s, \sigma\rangle$ cannot go wrong.

A.5 Type Soundness verified using Coq

We have automatically verified the type soundness result for a core subset of Isildur using the Coq proof assistant (http://coq.inria.fr). The core language focuses on the bit width aspect of the type system and consists of constructs which

utilize bit width information in the typing rules, including some arithmetic and bit manipulation operations and assignments. The only two types in the core language are integers and booleans.

The syntax of the core language is given by following grammar:

$$
\begin{array}{lll}
e & = e_1\ bop\ e_2 & \mid e[n_1 : n_2] \mid lv \\
 & \mid b & \mid n \\
lv & = x & \mid lv[e] \\
s & = \textbf{if}\ (e)\ s_1\ \textbf{else}\ s_2 & \mid lv = e; \quad \mid lv[n_1 : n_2] = e; \\
 & \mid \textbf{local}\ x{:}T = e; & \mid sl \\
sl & = \{\} & \mid \{s\} \quad \mid \{s\ sl\} \\
bop & = + & \mid *
\end{array}
$$

For the core language we have proved the following lemma in Coq:

```
Lemma soundness :
 forall (n : nat) (s : stmt) (sigma : Map value) (Psi : Map type),
    tc_store sigma Psi ->
    tc_stmt nil Psi s ->
    (exists s' : stmt, exists sigma' : Map value,
       execute s sigma s' sigma' n).
```

The Lemma corresponds to Theorem 3. The execute function steps through the program n times; if a step results in a value or runtime error at the ith iteration, the $(i+1)$th iteration will be the same value or runtime error. The function takes as its parameters a start state, consisting of a statement and a store, an ending state, and the number steps between these two states. It results in true if it is possible to take n steps to get from the start state to the end state. Thus, the lemma states that if a statement type checks with a specific store, then there is an ending state for any number of steps and therefore the statement cannot go wrong.

References

1. Cmelik, B., Keppel, D.: Shade: a fast instruction-set simulator for execution profiling. In: Proceedings of SIGMETRICS'94, Conference on Measurement and Modeling of Computer Systems. (1994) 128–137
2. Jeffery, C.L., Griswold, R.: A framework for execution monitoring in Icon. Software – Practice & Experience **24**(11) (1994) 1025–1049
3. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of PLDI'97, ACM SIGPLAN Conference on Programming Language Design and Implementation. (1997) 85–96
4. Hollingsworth, J.K., Miller, B.P., Goncalves, M.J.R., Naim, O., Xu, Z., Zheng, L.: Mdl: A language and compiler for dynamic program instrumentation. In: Proceedings of PACT'97, Conference on Parallel Architectures and Compilation Techniques. (1997) 201–213
5. Templer, K., Jeffery, C.L.: A configurable automatic instrumentation tool for ANSI C. In: Proceedings of ASE'98, Automated Software Engineering. (1998) 249–259

6. Tamches, A., Miller, B.: Fine-grained dynamic instrumentation of commodity operating system kernels. In: Proceedings of OSDI'99, Symposium on Operating Systems Design and Implementation. (1999) 117–130
7. Auguston, M.: Assertion checker for the C programming language based on computations over event traces. In: Proceedings of AADEBUG'99, International Workshop on Automated Debugging. (2000)
8. Moore, R.J.: A universal dynamic trace for linux and other operating systems. In: Proceedings of USENIX Annual Technical Conference, FREENIX Track. (2001) 297–308
9. Austin, T.M., Larson, E., Ernst, D.: Simplescalar: An infrastructure for computer system modeling. IEEE Computer **35**(2) (2002) 59–67
10. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of USENIX Annual Technical Conference, General Track. (2004) 15–28
11. Adl-Tabatabai, A.R., Hudson, R.L., Serrano, M.J., Subramoney, S.: Prefetch injection based on hardware monitoring and object metadata. In: Proceedings of PLDI'04, ACM SIGPLAN Conference on Programming Language Design and Implementation. (2004) 267–276
12. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: Proceedings of IPSN'05, Fourth International Conference on Information Processing in Sensor Networks. (2005) 477–482
13. Titzer, B.L., Palsberg, J.: Nonintrusive precision instrumentation of microcontroller software. In: Proceedings of LCTES'05, Conference on Languages, Compilers and Tools for Embedded Systems. (2005) 59–68
14. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: Proceedings of Mobisys'05, International Conference on Mobile Systems, Applications, and Services. (2005) 163–176
15. Landsiedel, O., Wehrle, K., Goettz, S.: Accurate prediction of power consumption in sensor networks. In: Proceedings of EmNetS-II, Second IEEE Workshop on Embedded Networked Sensors. (2005)
16. Regehr, J.: Random testing of interrupt-driven software. In: ACM International Conference On Embedded Software. (2005) 290–298
17. Srivastava, M.: Personal communication. UCLA (2006)
18. Ramsey, N., Fernandez, M.F.: Specifying representations of machine instructions. ACM Transactions on Programming Languages and Systems **19**(3) (1997) 492–524
19. Fernandez, M.F., Ramsey, N.: Automatic checking of instruction specifications. In: In Proceedings of ICSE'97, International Conference on Software Engineering. (1997) 326–336
20. Cifuentes, C., Emmerik, M.V., Ramsey, N.: The design of a resourceable and retargetable binary translator. In: In Proceedings of WCRE'99, Working Conference on Reverse Engineering. (1999) 280–291
21. Probst, M., Krall, A., Scholz, B.: Register liveness analysis for optimizing dynamic binary translation. In: In Proceedings of WCRE'02, Working Conference on Reverse Engineering. (2002) 35–44
22. Ramsey, N., Davidson, J.W.: Machine descriptions to build tools for embedded systems. In: In Proceedings of LCTES'98, Languages, Compilers, and Tools for Embedded Systems. (1998) 176–192
23. ATMEL: Avr 8-bit RISC. http://www.atmel.com/products/AVR/ (2006)
24. Committee, T.I.S.: Tool interface standard (tis) executable and linking format (elf). http://x86.ddj.com/ftp/manuals/tools/elf.pdf (1995)

25. Reshadi, M., Mishra, P., Dutt, N.D.: Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In: Proceedings of DAC'03, Design Automation Conference. (2003) 758–763
26. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal **38** (1959) 985–999
27. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27**(6) (1978) 509–516