NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE STUDIES

UNDERGRADUATE THESIS

Relational Representation of the LLVM Intermediate Language

Eirini I. Psallida

Supervisors: Yannis Smaragdakis, Associate Professor NKUA
Georgios Balatsouras, PhD Student NKUA

ATHENS

JANUARY 2014

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΠΤΥΧΙΑΚΕΣ ΣΠΟΥΔΕΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεσιακή Αναπαράσταση της Ενδιάμεσης Γλώσσας του LLVM

Ειρήνη Ι. Ψαλλίδα

Επιβλέποντες: Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ
Γεώργιος Μπαλατσούρας, Διδακτορικός φοιτητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΑΝΟΥΑΡΙΟΣ 2014

UNDERGRADUATE THESIS

Relational Representation of the LLVM Intermediate Language

Eirini I. Psallida

R.N.: 1115200700272

Supervisor: Yannis Smaragdakis, Associate Professor NKUA
Georgios Balatsouras, PhD Student NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεσιακή Αναπαράσταση της ενδιάμεσης γλώσσας του LLVM

Ειρήνη Ι. Ψαλλίδα

Α.Μ.: 1115200700272

Επιβλέπων: Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ
Γεώργιος Μπαλατσούρας, Διδακτορικός φοιτητής ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Περιγράφουμε τη σχεσιακή αναπαράσταση της ενδιάμεσης γλώσσας του LLVM, γνωστή ως LLVM IR. Η υλοποίηση μας παράγει σχέσεις από ένα πρόγραμμα εισόδου σε ενδιάμεση μορφή LLVM. Κάθε σχέση αποθηκεύεται σαν πίνακας βάσης δεδομένων σε ένα περιβάλλον εργασίας Datalog. Αναπαριστούμε το σύστημα τύπων καθώς και το σύνολο εντολών της γλώσσας του LLVM. Υποστηρίζουμε επίσης τους περιορισμούς της γλώσσας προσδιορίζοντάς τους με χρήση της προγραμματιστικής γλώσσας Datalog.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μεταγλωτιστές, Γλώσσες Προγραμματισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: σχεσιακή αναπαράσταση, ενδιάμεση αναπαράσταση, σύστημα τύπων, σύνολο εντολών, LLVM, Datalog

ABSTRACT

We describe the relational representation of the LLVM intermediate language, known as the LLVM IR. Our implementation produces the relation contents of an input program in the LLVM intermediate form. Each relation is stored as a database table into a Datalog workspace. We represent both the type system and the instruction set of the LLVM language. We also support the restrictions of the language specifying the corresponding constraints using the Datalog programming language.

SUBJECT AREA: Compilers, Programming Languages

KEYWORDS: relational representation, intermediate representation, type system, instruction set, LLVM, Datalog

To my sister, Ariadni

Acknowledgements

I would like to thank my supervisor, Yannis Smaragdakis for giving me the chance to work on this project and for his guidance and support during these months.

I would also like to thank Georgios Balatsouras for his contribution and help, providing advices and solutions to the problems we faced in order to complete this project.

Table of Contents

PROLOGUE.....14

1.INTRODUCTION.....15

2. THE LLVM COMPILER INFRASTRUCTURE.....17

2.1 Description.....17

2.2 Overview of the LLVMIR.....17

2.2.1 Type System.....17

2.2.2 Instruction Set.....21

3. RELATIONAL REPRESENTATION OF THE LLVM IR.....36

3.1 Background.....36

3.2 Representation of the LLVM Type System.....37

3.3 Representation of the LLVM Instruction Set.....45

CONCLUSION.....55

ABBREVIATIONS.....56

REFERENCES.....57

LIST OF TABLES

Table 1: Examples of integer types.....18
Table 2: Array type syntax.....19
Table 3: Examples of array types.....19
Table 4: Function type syntax.....19
Table 5: Examples of function types.....19
Table 6: Pointer type syntax.....20
Table 7: Examples of pointer types.....20
Table 8: Identified structure typ syntax.....20
Table 9: Literal strucure type syntax.....20
Table 10: Examples of stucture types.....20
Table 11: Opaque structure type syntax.....21
Table 12: Vector type syntax.....21
Table 13: Examples of vector types.....21
Table 14: Binary operation syntax.....22
Table 15: Bitwise binary operator syntax.....23
Table 16: 'ret' instruction first form.....24
Table 17: 'ret' instruction second form.....24
Table 18: 'br' instruction first form.....24
Table 19: 'br' instruction second form.....24
Table 20: 'switch' instruction syntax.....24
Table 21: 'indirectbr' instruction syntax.....24
Table 22: 'resume' instruction syntax.....25
Table 23: 'invoke' instruction syntax.....25
Table 24: 'alloca' instruction syntax.....25
Table 25: 'load' instruction basic syntax.....26
Table 26: 'store' instruction basic syntax.....26
Table 27: 'atomiccmpxchg' instruction syntax.....27
Table 28: 'atomicrmw' instruction syntax.....27
Table 29: 'getelementptr' intruction syntax.....27
Table 30: 'getelementptr' instruction syntax.....27
Table 31: 'extractvalue' instruction syntax.....28
Table 32: 'insertvalue' instruction syntax.....28
Table 33: 'extractelement' instruction syntax.....28
Table 34: 'insertelement' instruction syntax.....28
Table 35: 'shufflevector' instruction syntax.....29

Table 36: Conversion operator syntax....29
Table 37: 'icmp' instruction syntax....31
Table 38: 'fcmp' instruction syntax....31
Table 39: 'phi' instruction syntax....31
Table 40: 'call' instruction syntax....31
Table 41: 'va_arg' instruction syntax....32
Table 42: 'landingpad' instruction first form....32
Table 43: 'landingpad' instruction second form....32
Table 44: Catch close syntax....32
Table 45: Filter close syntax....33
Table 46: LLVM identifiers....36
Table 47: Function definition basic syntax....34
Table 48: Examples of constants....34
Table 49: The entity type 'instruction'....37
Table 50: Entity type declaration with refmode....37
Table 51: The entity 'type'....37
Table 52: Primitive type declaration....38
Table 53: Derived type declaration....38
Table 54: The primitive type hierarchy....38
Table 55: Standard primitive types....38
Table 56: The derived type hierarchy....39
Table 57: 'vector_type:size' and 'vector_type:component' predicates....39
Table 58: The 'vector_type:size' table....39
Table 59: The 'vector_type:component' table....40
Table 60: Mandatory role constraints for the vector type....40
Table 61: Constraint for the size of vector....40
Table 62: Constraint for the component type of vector....40
Table 63: The 'pointer_type:integer' predicate....41
Table 64: The 'pointer_type:fp' predicate....41
Table 65: Function type predicates....41
Table 66: 'function_type:return' table....41
Table 67: 'function:type:params' table....42
Table 68: 'function:type:nparams' table....42
Table 69: Constraint for the return type of function type....42
Table 70: The entity 'function'....42
Table 71: Function predicates....42
Table 72: Constraint for the function....43

Table 73: Variable and immediate entities....43
Table 74: 'variable:type' and 'immediate:type' predicates....43
Table 75: The entity 'operand'....43
Table 76: The constructor for the entity 'variable'....44
Table 77: The constructor for the entity 'immediate'....44
Table 78: The 'operand:type' from variable....44
Table 79: The 'operand:type' from immediate....44
Table 80: The 'operand:as_variable' predicate....44
Table 81: The 'operand:as_immediate' predicate....44
Table 82: The entity 'instruction....45
Table 83: The 'instruction:to' predicate....45
Table 84: The 'instruction:type' predicate....45
Table 85: The 'instruction:next' predicate....45
Table 86: The 'instruction:function' predicate....45
Table 87: LLVM instructions....46
Table 88: The 'add_instruction:first_operand' predicate....46
Table 89: The 'add_instruction:second_operand' predicate....46
Table 90: Constraints for the 'add' instruction....46
Table 91: Constraint for the 'add' instruction type....47
Table 92: Constraints for the 'add' instruction and 'instruction' type....47
Table 93: The 'trunc' instruction....47
Table 94: The 'trunc_instruction:from_type' predicate....47
Table 95: Constraints for the 'trunc' instruction....48
Table 96: Constraints for the 'trunc' instruction type....48
Table 97: Constraint for the resulting value type of 'trunc'....48
Table 98: The 'vector:compatible' idb predicate....48
Table 99: The 'ret' instruction....49
Table 100: Rules that specify if a function is well formed....49
Table 101: The 'function:wellformed' idb predicate....50
Table 102: The 'invoke' instruction....50
Table 103: The direct and indirect 'invoke' instruction form....50
Table 104: Idb predicates for the 'invoke' instruction....50
Table 105: The 'invoke_instruction:arg' predicate....50
Table 106: The 'invoke_instruction:signature' idb predicate....51
Table 107: The 'invoke' instruction labels....51
Table 108: Constraints for the 'invoke' instruction labels....51
Table 109: Constraint for the 'invoke' instruction return value type....51

Table 110: Constraint for the 'invoke' instruction return type....52
Table 111: The 'extractvalue' instruction....52
Table 112: Predicates for the indices of the 'extractvalue' instruction....52
Table 113: Rules to compute the resulting type for each dereference in the 'extractvalue' instruction....53
Table 114: The 'constant:toInt' predicate....53
Table 115: The 'extractvalue_instruction:interm_type' idb predicate....53
Table 116: The 'extractvalue_instruction:value_type' idb predicate....53
Table 117: The 'extractvalue_instruction:value' predicate....53
Table 118: The 'extractvalue_instruction:base_type' idb predicate....54
Table 119: Constraint for the base type of the 'extractvalue' instruction....54

PROLOGUE

This project has been developed as my undergraduate thesis since May of 2013 in the University of Athens at the department of Informatics and Telecommunications.

1. Introduction

LLVM was introduced in 2000 at the University of Illinois as a research project by Chris Lattner and Vikram Adve and it was originally developed to provide compilation techniques for both static and dynamic programming languages. LLVM has now become an umbrella project comprising a collection of reusable toolchain technologies (e.g., compilers, optimizers, etc) all written in C++. A variety of programming languages (e.g., Python, Ruby, Scala) nowadays use LLVM as their front end compiler.

The LLVM intermediate language, known as the LLVM IR, consists one of the strongest points of the LLVM compiler system. It is a low-level language with strictly defined semantics supporting a language-independent type system and instruction set. The type system consists of primitive types (e.g., integer, floating point, etc) and derived types (e.g., function, pointer, structure, etc). The instruction set consists of several different categories of instructions (e.g., terminator, binary, conversion, memory, etc) depending on their use (e.g., control flow, computations, memory management, etc).

We describe how we represent the LLVM IR as relations, stored as database tables. Our implementation produces the relation contents of an input program in LLVM intermediate form. This constitutes the pre-processing step for a *points-to (or pointer) analysis*. After this step, new relations can be derived from the existing ones using the Datalog programming language. Relations, equivalently *predicates*, are the main Datalog data type. The relations that are produced from our implementation are called *EDB (extensional database)* as they represent facts that are explicitly entered into the database. The derived relations are called *IDB (intentional database).*

A Datalog program is a set of clauses. There are three kinds of clauses: *facts*, *constraints* and *rules*. *Constraints* are used to indicate the restrictions that facts must specify. *Rules* are used to derive new facts. IDB relations are computed with *rules*. We declare *entity types* to represent the LLVM IR components (e.g., instructions, functions, types, etc). We use Datalog to represent the type system, the instruction set as well as the restrictions specified by the LLVM language.

In chapter 2 we describe the LLVM IR syntax, the type system and its components and the several instructions as well as their use. In chapter 3 we describe in more detail the modeling of the LLVM IR. We see several examples of the type system and the instruction set representation as well as examples of constraints.

1.1 Motivation

Such a relational representation enables a host of program analyses over LLVM programs. Static program analysis in Datalog is an important recent trend [8, 9]. The declarativeness of Datalog and its ability to define recursive relations makes it ideal for specifying complex program analysis algorithms and, as a consequence, Datalog has been extensively used for both low level [1, 2, 5] and for high level [6, 7] analyses.

Points-to (or pointer) analysis [3, 4] is one of the most fundamental static program analyses. It computes a static approximation of all the data that a pointer variable or expression can reference during program run-time. In order to do points-to analysis, we need to have reachability information. This means that we have to know the truly possible variable assignment actions. In Datalog we can easily specify a mutually recursive definition of a reachability and points-to analysis.

2. The LLVM Compiler Infrastructure

2.1 Description

LLVM is an open source umbrella project, introduced in 2000 as a research project by Chris Lattner and Vikram Adve at the University of Illinois. It comprises a collection of reusable libraries and toolchain components (e.g., compilers, debuggers, optimizers, etc) written in C++. The name "LLVM" used to stand for “Low Level Virtual Machine”, but today (since it became meaningless) is just a brand for the project. The 2012 ACM Software System Award is given to Vikram Adve, Evan Cheng and Chris Lattner for LLVM.

One of the tools that LLVM is known for is the clang compiler, a C, C++, Objective C and Objective C++ front-end for the LLVM compiler. Clang converts the source into the LLVM intermediate representation code.

The initial goal of LLVM was to provide a modern, SSA-based compilation strategy which would support static as well as dynamic compilation of arbitrary languages, but its main use nowadays is to implement a wide variety of programming languages (e.g., Java, .NET, Python, Ruby, Haskell). LLVM was also used to create a lot of new products, such as the OpenCL GPU programming language and runtime.

2.2 Overview of the LLVM IR

The LLVM assembly language [10] is one of the strongest points of the LLVM compiler system. It is a low-level language, with strictly defined semantics, known as the LLVM intermediate representation *(LLVM IR)*. It supports a language-independent type system and instruction set, which constitute the most important features of the IR.

2.2.1 Type System

LLVM has a strong type system that allows a number of optimizations to be performed directly on the IR. The type system consists of *primitive types* and *derived types*.

Primitive Types

The *primitive types* are the basic building components of the LLVM system consisting of the following types:

- integer
- floating point
- label
- void
- metadata

- x86mmx

The *integer* type has the syntax iN, where N specifies its bit width.

Table 1: Examples of integer types

i1	a single-bit integer (the boolean type in C and C++)
i32	a 32-bit integer
i64	a 64-bit integer

The *floating point* type includes the subtypes:

- half 16-bit floating point value
- float 32-bit floating point value
- double 64-bit floating point value
- fp128 128-bit floating point value
- x86_fp80 80-bit floating point value
- ppc_fp128 128-bit floating point value (two 64-bits)

The *x86mmx* type represents a value held in an MMX register on an x86 machine. Its only allowed uses are in parameters and return values as well as in load, store and bitcast instructions.

Derived Types

Derived types are a powerful feature in LLVM. They allow the representation of other useful types using as element type other *primitive* or *derived* types. *Derived types* consist of the following types:

- array
- function
- pointer
- structure
- opaque structures
- vector

Array Type

The *array type* has the following syntax.

Table 2: Array type syntax

[<# elements> x <element_type>]

The next table shows some examples of the array type.

Table 3: Examples of array types

[5 x i32]	Array of 5 32-bit integer values.
[10 x [2 x float]]	A multidimensional array, 10x2 of float values.
[2 x { i8, double }]	Array of 2 elements of structure type containing one 8-bit integer and a double value.

Function Type

The *function type* represents a function signature and has the following syntax.

Table 4: Function type syntax

<returntype> (<argument list>)

The next table shows some examples of the function type.

Table 5: Examples of function types

i32 (float, i32) *	Pointer to a function that takes a float and an i32 value, returning an i32.
{i32, i32} (i32*, ...)	A vararg function that takes at least one pointer to i32, returning a structure that contains two i32 values.

Pointer Type

The *pointer type* specifies locations and refers to objects in memory. It may have an optional address space attribute. It has the following syntax.

Table 6: Pointer type syntax

<type> *

The next table shows some examples of the pointer type.

Table 7: Examples of pointer types

[2 x half] *	Pointer to array of 2 half values.
float addrspace(3) *	Pointer to a float value that resides in address space #3.
i32 (float, i32) *	Pointer to a function that takes a float and an i32, returning an i32.

Structure Type

The *structure type* is a group of data members together in memory. A structure can either be "*literal*" or "*identified*". An *identified* structure type has the following syntax.

Table 8: Identified structure type syntax

%T = type { <type list> }	%T is the name of the structure type.

A *literal* structure has the following syntax.

Table 9: Literal structure type syntax

{ <type list> }

The next table shows some examples of the structure type.

Table 10: Examples of structure types

%struct.S = type {i32, i8*}	An identified structure type, where the first element is an i32 and the second a pointer to an i8.
{i8, i8}	A literal structure of 2 i8 values.
{i32, float (i32*)}	A literal structure where the first element is an i32 and the second a function taking a pointer to an i32 value, returning a float.

Opaque Structure Types

Opaque structure types represent named structures with no body specified.
They have the following syntax.

Table 11: Opaque structure type syntax

%T = type opaque

Vector Type

The *vector type* is used when multiple data are processed simultaneously by a single instruction (SIMD). It has the following syntax.

Table 12: Vector type syntax

< <# elements> x <element_type> >

The next table shows some examples of the vector type.

Table 13: Examples of vector types

< 2 x i64* >	Vector of 2 pointers to i64.
< 5 x float >	Vector of 5 float values.
< 10 x i32 >	Vector of 10 i32 values.

Instructions produce values that can have any type other than void and function types.

2.2.2 Instruction Set

The entire LLVM instruction set consists of 56 instructions that are classified in the following categories:

- Binary operations
- Bitwise binary operations
- Terminator instructions
- Memory Access and Addressing Operations
- Vector Operations
- Aggregate Operations
- Conversion Operations
- Other Operations

Most of the instructions, including binary, bitwise binary and conversion operations, are

three-address instructions. They take one or two operands and produce a single result. An operand can be a variable or a constant value. LLVM supports a Static Single Assignment form (SSA), which means that each variable (a typed register) is assigned exactly once. For that reason LLVM provides an infinite set of typed virtual registers that hold the produced values of the instructions.

Binary Operations

Binary operators are responsible for most of the computations in a program. They take two operands of the same type and produce a value that has the same type as its operands. The type of the resulting value can be integer, floating point or vector type in the case of multiple data. Binary operators have the following syntax.

Table 14: Binary operation syntax

<result_value> = opcode_name <type> <operand 1>, <operand 2>

The *opcode_name* must be one of the following instructions:

1. add
2. sub
3. mul
4. udiv
5. sdiv
6. urem
7. srem
8. fadd
9. fsub
10. fmul
11. fdiv
12. frem

The type <type> in the instructions 1 – 7 must be integer or vector of integer values and in the instructions 8 – 12 floating point or vector of floating point values.

Bitwise Binary Operators

Bitwise binary operators perform several forms of bit-twiddling in a program. Like binary operators, they take two operands of the same type and produce a value that has the same type as its operands. The type of the resulting value can be integer or vector of integer values. Bitwise binary operators have the following syntax.

Table 15: Bitwise binary operator syntax

<result_value> = opcode_name <type> <operand 1>, <operand 2>

The *opcode_name* must be one of the following instructions:

- shl
- lshr
- ashr
- and
- or
- xor

The 'shl' instruction returns the first operand shifted to the left a specified number of bits.

The 'lshr' instruction (logical shift right) returns the first operand shifted to the right a specified number of bits, and filled with zero.

The 'ashr' instruction (arithmetic shift right) returns the first operand shifted to the right a specified number of bits, with sign extension.

The 'and' instruction returns the bitwise logical and of its two operands.

The 'or' instruction returns the bitwise logical inclusive or of its two operands.

The 'xor' instruction returns the bitwise logical exclusive or of its two operands.

Terminator Instructions

Every basic block must end with a terminator instruction in order to be well-formed. Terminator instructions cause control flow and they indicate which block should be executed after the current is finished. Terminator instructions consist of the following instructions:

- ret
- br
- switch
- indirectbr
- resume
- unreachable
- invoke

The 'ret' instruction returns control flow from a function back to the caller.
It has two forms.

Table 16: 'ret' instruction first form

ret void

Table 17: 'ret' instruction second form

ret <type> <value>

The first form has no return value. It returns from a function that has a void return type. The other form takes a single operand, a typed value, and returns from a function that has a non-void return type.
The 'br' instruction transfers control flow to a different basic block in the current function.
It has two forms.

Table 18: 'br' instruction first form

br i1 <condition>, label <true>, label <false>

Table 19: 'br' instruction second form

br label <destination>

The first form is a conditional br instruction and if the <condition> is evaluated to be true control flows to label <true>, otherwise to label <false>. The other form is an unconditional br instruction and control flows to its single label <destination>.

The 'switch' instruction can transfer control flow to one of several different destinations.
It has the following syntax.

Table 20: 'switch' instruction syntax

switch <integer_type> <value>, label <default_destination> [<integer_type> <constant_value>, label <destination> ...]

It takes three arguments, a typed value, a label that indicates the default destination and a table of pairs of constants and labels.

The 'indirectbr' instruction has the following syntax.

Table 21: 'indirectbr' instruction syntax

indirectbr <type>* <address>, [label <destination1>, label <destination2>, ...]

It causes control flow to the label that is specified by the <address> argument, which is always of pointer type.

The 'resume' instruction has the following syntax.

Table 22: 'resume' instruction syntax

```
resume <type> <value>
```

It takes a single argument and has no successors (there are no basic blocks to be executed after the current one is finished).

The 'unreachable' instruction takes no arguments and indicates that a certain part of the code cannot be reached.
The 'invoke' instruction is the only terminator instruction that produces a value.
It has the following syntax.

Table 23: 'invoke' instruction syntax

```
<result_value> = invoke <pointer_type> <function_value>(<function arguments>)
to label <normal> unwind label <exception>
```

The 'invoke' instruction represents high-level exceptions directly in LLVM. It transfers control flow to a function specified by the <function_value> argument. The 'invoke' instruction can be direct, providing a pointer to a function name, or indirect, providing a function pointer variable. It is also possible for control flow to be transferred to either the <normal> or the <exception> label. The 'normal' label is reached when the called function executes a ‘ret‘ instruction. The 'exception' label is reached when a callee returns via the 'resume' instruction or some other exception handling mechanism.

Memory Access and Addressing Operations

The following instructions are responsible for the memory management in LLVM:

- alloca
- load
- store
- atomic instructions
- getelementptr

The 'alloca' instruction is used to allocate memory on the stack frame of the current function. After the function's execution the allocated memory is automatically released.
It has the following syntax.

Table 24: 'alloca' instruction syntax

```
<result_value> = alloca <allocated_type> [, <type> <num_elements>] [, align
<alignment>]
```

It allocates space equal to sizeof (<allocated_type>) * <num_elements>. If <num_elements> is omitted, it defaults to one. The type of the resulting value is a pointer to the

<allocated_type> argument. The 'alloca' instruction has also an optional alignment, which is a constant value.

The 'load' instruction reads from memory and it has the following basic syntax.

Table 25: 'load' instruction basic syntax

<result_value> = load <type>* <ptr_value>

It takes a single operand that specifies the location of memory from which to load.

The 'store' instruction writes to memory and it doesn't produce any value. It has the following basic syntax.

Table 26: 'store' instruction basic syntax

store <type> <value>, <type>* <pointer_value>

The <value> operand indicates the value that will be stored and the <pointer_value> operand the memory location to store it. The type of the second operand must be a pointer to the type of the first operand.

If 'load' and 'store' are marked as atomic, they take some extra arguments as shown below in the atomic instructions.

Atomic Instructions

This category includes the following instructions:

- atomic load
- atomic store
- fence
- atomiccmpxchg
- atomicrmw

Atomic instructions provide guarantees in the case of threads and signals. They take an 'ordering' and an optional 'singlethread' argument. The ordering parameter (e.g., unordered, monotonic, acquire, etc) indicates the synchronization with other atomic instructions. If an atomic instruction is marked as singlethread, it only synchronizes with other operations running in the same thread (for example, in signal handlers).
The 'atomiccmpxchg' instruction (atomic-compare-and-exchange) modifies the memory atomically. It loads a value in memory and compares it to a given value. If they are equal, a new value is stored into the memory. 'atomiccmpxchg' has the following syntax.

Table 27: 'atomiccmpxchg' instruction syntax

```
<result_value> = atomiccmpxchg <type>* <ptr_value>, <ty> <compare_value>, <ty> <new_value> [singlethread] <ordering>
```

The <ptr_value> specifies the memory address from which to load. If the value at the address specified by the <ptr_value> and the <compare_value> are equal, then the <new_value> is stored into the memory.
The 'atomicrmw' instruction (atomic-read-modify-write) modifies the memory atomically. It takes an operation argument, an address whose value to modify and an argument to the operation. 'atomicrmw' has the following syntax.

Table 28: 'atomicrmw' instruction syntax

```
atomicrmw <operation> <type>* <ptr>, <type> <value> [singlethread] <ordering>
```

The <operation> argument is a keyword that defines the operation that will be performed (e.g., add, sub, xchg, and, or, max, etc). The <ptr> operand specifies the memory location that will be modified according to the operation and the <value> operand is the operation's argument.

The 'getelementptr' instruction makes address computations only and, unlike load and store instructions, it dereferences nothing. It has the following syntax.

Table 29: 'getelementptr' intruction syntax

```
<result_value> = getelementptr <type>* <ptr_value> {, <integer_type> <idx>}*
```

The operand <ptr_value> is always a value of a pointer type and is the base for the computations. Subsequent types can be arrays, vectors, and structs. (Note that subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing the calculation.) The rest of the arguments indicate which of the elements of the object are indexed. It returns a pointer to the specified element. In the case where the first operand is a vector of pointers the 'getelementptr' instruction has the following syntax.

Table 30: 'getelementptr' instruction syntax

```
<result_value> = getelementptr <vector> <ptr_value>, <vector index type><idx>
```

Aggregate Operations

Arrays and structures are considered aggregate types. The aggregate operations include the following instructions:

- extractvalue
- insertvalue

The 'extractvalue' instruction has the following syntax.

Table 31: 'extractvalue' instruction syntax

<result_value> = extractvalue <aggregate_type> <value>, <index>{, <index>}*

The <result_value> holds the extracted value of the aggregate from the position specified by the indices. All the indices must be constant values.

The 'insertvalue' instruction has the following syntax.

Table 32: 'insertvalue' instruction syntax

<result_value> = insertvalue <aggregate_type> <value>, <type> <element>, <idx> {, <idx>}*

It inserts an element in the aggregate at the position specified by the constant indices <idx>. The <result_value> holds an aggregate with type <aggregate_type>. Its value is the same as <value> except that the value at the position specified by the indices is that of <element>.

Vector Operations

Vector operations include the following instructions:

- extractelement
- insertelement
- shufflevector

The 'extractelement' instruction has the following syntax.

Table 33: 'extractelement' instruction syntax

<result_value> = extractelement < n x <type> > <value>, i32 <position>

It extracts an element from a <value> of vector type at a specified position. The <result_value> holds the extracted element with type <type>.

The 'insertelement' instruction has the following syntax.

Table 34: 'insertelement' instruction syntax

<result_value> = insertelement < n x <type> > <value>, <ty> <element>, i32 <position>

It inserts an element into a vector at a specified position. The <result_value> holds the changed vector which is of the same type as the <value> operand.

The 'shufflevector' instruction has the following syntax.

Table 35: 'shufflevector' instruction syntax

```
<result_value> = shufflevector < n x <type> > <vector1>, < n x <type> >
<vector2>,<m x i32> <shuffle_mask>
```

It takes two vector operands of the same type and makes a permutation of their elements. The <result_value> is a vector whose length is the same as the <shuffle_mask> and whose element type is the same as the element type of the first two operands. The <shuffle_mask> argument must be a vector of constant values.

Conversion Operations

Conversion operations have the following syntax.

Table 36: Conversion operator syntax

```
<result_value> = opcode_name <type> <value> to <type2>
```

Conversion operations take a single operand and a type. They perform several bit conversions on the operand <value> and return a value with type <type2>.

The *opcode_name* must be one of the following instructions:

- trunc
- zext
- sext
- fptrunc
- fpext
- fptoui
- fptosi
- uitofp
- sitofp
- ptrtoint
- inttoptr
- bitcast

The 'trunc' instruction truncates the high order bits in <value> and converts the remaining to <type2>. Both types must be integers or vectors with the same number of integer values.

The 'zext' instruction fills the high order bits of the <value> with zero until it reaches the size of <type2>. Both types must be integers or vectors with the same number of integer values.
The 'sext' instruction copies the highest order bit of the <value> until it reaches the bit size of <type2>. Both types must be integers or vectors with the same number of integer values.

The 'fptrunc' instruction truncates a <value> from a larger floating point type to a smaller floating point type.

The 'fpext' instruction extends the <value> from a smaller floating point type to a larger floating point type.

The 'fptoui' instruction converts the floating point <value> into the nearest (rounding towards zero) unsigned integer value.
The 'fptosi' instruction converts the floating point <value> into the nearest (rounding towards zero) signed integer value.

The 'uitofp' instruction interprets the <value> as an unsigned integer and converts it to the corresponding floating point value.
The 'sitofp' instruction interprets the <value> as a signed integer and converts it to the corresponding floating point value.

The 'ptrtoint' instruction converts the pointer or a vector of pointers <value> to the integer or vector of integers <type2>. If the <value> is smaller than <type2>, then a zero extension is done, otherwise a truncation is done. If they have the same size, then nothing is done other than a type change.

The 'inttoptr' instruction converts an integer <value> to a pointer type, <type2>. If the <value> is smaller than <type2>, then a zero extension is done, otherwise a truncation is done. If they have the same size, then nothing is done other than a type change.

The ‘bitcast‘ instruction converts the <value> to type <type2> without changing any bits.

Other Operations

This category includes the following instructions:

- icmp
- fcmp
- select
- phi
- call
- va_arg
- landingpad

The 'icmp' instruction compares two operands and returns a value that has boolean (i1) or of vector of boolean values type. It has the following syntax.

Table 37: 'icmp' instruction syntax

```
<result_value> = icmp <condition> <type> <operand1>, <operand2>
```

The <condition> argument is a keyword that indicates the kind of comparison that will be performed (e.g., eq, ne, ugt, uge, ult, etc). The <type> must be integer, or pointer, or integer vector type.

The 'fcmp' instruction is used like the 'icmp' instruction. The only difference is that the <type> must be floating point or vector of floating point values type.

The 'select' instruction has the following syntax.

Table 38: 'fcmp' instruction syntax

```
<result_value> = select sel_type <condition>, <type> <value1>, <type><value2>
```

It selects <value1> or <value2> according to the <condition> argument. If the <condition> operand is evaluated to 1, the first operand is returned, otherwise the second. The operands <value1> and <value2> must have identical types.
The 'phi' instruction corresponds to the SSA φ - node. It has the following syntax.

Table 39: 'phi' instruction syntax

```
<result_value> = phi <type> [ <value0>, <label0> ], ...
```

The first argument <type> defines the type of the incoming values. The second argument is a list of pairs of values and labels and each pair corresponds to one of the predecessor blocks of the current basic block. The phi instruction returns one of the incoming values depending on which basic block control flow came from. Phi instructions must appear at the beginning of a basic block and there must be no non-phi instructions between a phi and the start of a block.

The 'call' instruction is used to perform a function call. Its basic syntax is the following.

Table 40: 'call' instruction syntax

```
<result_value> = call <type> [<function_type>*] <function_value>(<function
arguments>)
```

The argument <function_value> contains a pointer to the function being invoked. The argument <type> corresponds to the call instruction's type and is also the function's return type. The argument <function_type>* is a pointer to the function's signature type. Extra arguments may be defined (e.g., calling convention, parameter attributes, function attributes, etc) that provide information to the optimizer.

The 'va_arg' instruction implements the va_arg macro in C. It has the following syntax.

Table 41: 'va_arg' instruction syntax

```
<result_value> = va_arg <va_list*> <arg_list>, <argument_type>
```

It loads an argument from the list <arg_list> that has <arg_type> type. The <arg_list> points then to the next argument.

The 'landingpad' instruction is used to represent exception handling in LLVM. It specifies the basic block where the exception lands and corresponds to the 'catch' code found in a try/catch sequence. The 'landingpad' instruction has two forms:

Table 42: 'landingpad' instruction first form

```
<result_value> = landingpad <result_type> personality <type> <pers_func>
                    <clause>+
```

Table 43: 'landingpad' instruction second form

```
<result_value> = landingpad <result_type> personality <type> <pers_func>
                    cleanup <clause>*
```

The <result_value> has the the type <result_type>. The <pers_func> argument refers to the personality function that is used for this try/catch sequence. The personallity function (e.g. __gxx_personality_v0 in C++) defines an exception handling behavior and receives the context of the exception, an exception structure containing the exception object type and value, and a reference to the exception table for the current function.

The 'landingpad' instruction must have either the cleanup flag or at least one <clause> argument. There are two types of clauses, the catch and the filter clause. They have the following syntax.

Table 44: Catch close syntax

```
<clause> := catch <type> <value>
```

Table 45: Filter close syntax

<clause> := filter <array constant type> <array value>

The argument of the filter clause must have an array type of constant values.

Intrinsic Functions

LLVM supports several intrinsic functions that constitute an extension mechanism for the LLVM IR. Their name must always start with the prefix “llvm.” Intrinsic functions cannot have a body specified so they appear as function declarations in the LLVM code. They can only be used in a call or invoke instruction.

LLVM Identifiers

LLVM provides local and global identifiers. Local identifiers comprise register names and named types (e.g., structures) and global identifiers comprise functions and global variables. Local identifiers must start with the prefix character '%' and global ones with '@'.

The next table shows some examples of LLVM identifiers.

Table 46: LLVM identifiers

%x, @main	Named values with their prefix.
%0, @10	Unnamed (unsigned numeric) values with their prefix.

Module

The Module is a translation unit of an input program and constitutes the top level structure in the LLVM IR. It contains global variables and functions.

Global Variable

Global variables are accessed through pointers, as they define memory locations. A global variable has several arguments and attributes (e.g., address space, linkage type, visibility style, alignment, etc), which define the variable.

Function

A function, like global variables, is accessed by a pointer to its memory location.
The basic syntax of a function definition is the following.

Table 47: Function definition basic syntax

define <return_type> @<function_name> ([argument_list]) { ... }

A function definition contains a list of basic blocks and each block contains a sequence of LLVM instructions, ending in exactly one terminator instruction. A function (definition and declaration) might have several attributes (e.g., parameter attributes, linkage type, calling convention, alignment, etc), which define the function.

Basic Block

Each basic block starts with a label, contains a list of instructions and must end with a 'terminator' instruction (e.g., ret, indirectbr, switch, etc). The first block in a function is always executed immediately on entrance to the function and cannot have any predecessor basic blocks. For that reason the first block can never have 'phi instructions.

Constants

Constants constitute one more kind for LLVM identifiers. LLVM supports several types of constants (e.g., boolean, integer, array, vector, etc) both of primitive and derived types. Constants with void or x86mmx type are not specified.

The next table shows some examples of constants.

Table 48: Examples of constants

true, false	boolean constants of i1 type
4, 15	integer constants
null	a constant of pointer type
[i32, 1, i32 2, i32 3]	a constant array of type [3 x i32]
{ i8* null, i32 2, float 3.0 }	a constant structure of type { i8*, i32, float }

Constant Expressions

Constant expressions are used as constants involving other constants. They perform the following LLVM operations on constants:

- binary operations
- bitwise operations
- conversion operations
- getelementptr
- icmp

- fcmp
- select
- aggregate operations
- vector operations

The restrictions on operands are the same as those with the corresponding instructions.

3. Relational Representation of the LLVM IR

In this chapter we describe how we represent the LLVM intermediate form of programs as relations, typically stored as database tables. Our implementation takes a program in LLVM IR format as input and produces the relation contents *(EDB predicates)*. This constitutes the pre-processing step for a pointer analysis. After this step, new relations *(IDB predicates)* can be derived from the existing ones using the Datalog language. The declarativeness of Datalog and its ability to define recursive relations makes it ideal for specifying complex program analysis algorithms.

3.1 Background

Datalog is a declarative programming language originally introduced in the database domain. Syntactically it is a subset of Prolog but it does not permit functions. It appeared in the mid-1970's and the term “Datalog”, which suggests “data combined with logic”, was originally coined by David Maier. It is based on first-order logic and provides support for deductive inferences, including recursive rules. A Datalog program is a set of clauses. A clause comprises a head and/or a body, both of which contain atoms. An atom consists of a predicate name followed by a list of arguments, each of which corresponds to one of the predicate's roles. In logic, predicates are either properties that may be held by individual values or relationships that may apply to multiple values. There are three kinds of clauses: *facts*, *constraints* and *rules*. Unlike Prolog, Datalog programs are guaranteed to terminate.

EDB and IDB Predicates

Relations are the main Datalog data type. Each relation is a predicate stored as a database table. The facts of a predicate correspond to table rows and each row comprises a tuple of data elements. All rows in a given table must have the same number of data elements. Hence, a column in a table consists of all of the data elements occupying the corresponding position in the facts of the predicate. Moreover, all of the elements of a column must be of the same type.

The predicates that are produced from our implementation are called *EDB (extensional database)* as they hold the facts that are explicitly entered into the database. Further information can be computed with *IDB predicates*. I*DB (intentional database)* predicates are computed with rules. A rule is used to derive new facts from existing ones. It consists of two parts separated by a left arrow. The left part of the arrow is the head and the right the body of the rule. The rule specifies that if the body is true then the head will also be true and a new fact can then be derived.

In our implementation we use LogiQL [11], which is a Datalog dialect by LogicBlox Inc. It supports standard Datalog concepts, such as predicates and inference rules, as well as types and constraints. We follow the convention of capitalizing the first letter of variable names, while writing predicate names in lower case concatenating two different words with the '_'

character.

Entity Type and Refmode Declarations

Apart from the built-in types (e.g., string, int, etc) new types can be defined, called *entity* types. *Entity* type declarations specify the kind of entities and their representation. A declaration consists of two parts separated by a right arrow. The left part consists of a predicate name giving the name of the entity type. For example, the next table shows how the 'instruction' entity type could be declared.

Table 49: The entity type 'instruction'

```
instruction(Insn) -> .
```

In this example there is no information given about how instruction entities are represented. The right part of the arrow consists only of a period, which signifies the end of the declaration. *Refmodes* can be used in cases as this to identify the individual elements of the entity. A *refmode* predicate constitutes the primary key that identifies the entity and it is declared at the same time as the entity.

Table 50: Entity type declaration with refmode

```
instruction(Insn), instruction:id(Insn:Id) -> string(Id).
```

In the above example 'instruction' is an entity type and 'instruction:id' is a refmode predicate for it. This declaration specifies that instructions are identified by their id and this id is represented as a string.

Constraints

We can express restrictions on our data using constraints. A constraint has the same syntax as a type declaration, two parts separated by a right arrow. The left part of the arrow indicates the predicate being constrained and the right part indicates the restrictions that facts must satisfy. Constraints are checked at runtime to ensure that they are in no way violated by the input data. We will see several examples of constraints in the modeling of the LLVM IR, below.

3.2 Representation of the LLVM Type System

We declare the entity 'type' with refmode as follows.

Table 51: The entity 'type'

```
type(Type), type:id(Type:Id) -> string(Id).
```

'type' is an entity type and 'type:id' is a refmode predicate for it. This declaration specifies that

types are identified by their id and this id is represented as a string.

As mentioned previously, the LLVM type system consists of primitive and derived types. These two kind of types are declared as subtypes of 'type' as follows.

Table 52: Primitive type declaration

```
primitive_type(Type) -> type(Type).
```

Table 53: Derived type declaration

```
derived_type(Type) -> type(Type).
```

Both primitive and derived types are entities identified by their id 'Type'.

The next table shows how the primitive type hierarchy can be declared.

Table 54: The primitive type hierarchy

```
integer_type(Type)    -> primitive_type(Type).

fp_type(Type) -> primitive_type(Type).

void_type[ ] = Type   -> primitive_type(Type).

label_type[ ] = Type  -> primitive_type(Type).

metadata_type[ ] = Type  -> primitive_type(Type).
```

Integer and *floating point* types are considered primitives types. We also construct some standard types, as they are used often in several of our declarations, as the next table shows.

Table 55: Standard primitive types

```
void_type[ ]  = Type <- primitive_type(Type), type:id(Type:"void").

label_type[ ] = Type <- primitive_type(Type), type:id(Type:"label").

metadata_type[ ] = Type <- primitive_type(Type), type:id(Type:"metadata").
```

The 'void_type', 'label_type' and 'metadata_type' predicates are functional predicates. Functional predicates have at most one value for a certain key. For instance, the 'void_type' predicate that has an empty key can have only one value, which must be a primitive type and have the string value “void”.

The next table shows how the derived type hierarchy can be declared.

Table 56: The derived type hierarchy

```
function_type(Type) -> derived_type(Type).

vector_type(Type)  -> derived_type(Type).

pointer_type(Type) -> derived_type(Type).

aggregate_type(Type) -> derived_type(Type).

array_type(Type) -> aggregate_type(Type).

struct_type(Type) -> aggregate_type(Type).
```

Arrays and *structures* are considered aggregate types. We also declare a number of functional predicates to represent the structure of derived types (e.g., component type, number of elements, address space, etc). For example, the *vector* and the *function* type can be represented as follows.

Vector type

The *vector type* requires a size (number of elements) and a component type. We declare the following predicates describing the vector type.

Table 57: 'vector_type:size' and 'vector_type:component' predicates

```
vector_type:size[Type] = Size -> vector_type(Type), int[64](Size).

vector_type:component[Type] = Comp -> vector_type(Type), type(Comp).
```

In the 'vector_type:size' predicate the key Type is of type 'vector_type' and its value isArial *Size*, which is a 64-bit integer. In the 'vector_type:component' predicate the key *Type* is of type 'vector_type' and has a value *Comp*, which is of type 'type'.

For example, if we have the vector type *< 5 x i32 >* we get the following tables.

Table 58: The 'vector_type:size' table

vector type id	size
< 5 x i32 >	5

Table 59: The 'vector_type:component' table

vector type id	component type id
< 5 x i32 >	i32

Constraints

A vector type has some restrictions that it must satisfy and which can be expressed using constraints. For example:

1. A vector type must have a size and a component type.
2. The size must be larger than 0.
3. The component type can be any integer or floating point type,

or a pointer to these types.

The first restriction can be declared by the following two mandatory role constraints.

Table 60: Mandatory role constraints for the vector type

```
vector_type(Type) -> vector_type:component[Type] = _.

vector_type(Type) -> vector_type:size[Type] = _.
```

The above constraints declare that the role of having a component type and a size is mandatory for each vector type. We use the anonymous variable as we don't care what the size or the component is, only that they must exist.

The second restriction can be expressed as follows.

Table 61: Constraint for the size of vector

```
vector_type:size[_] = Size -> Size > 0.
```

In this constraint the anonymous variable is used as we don't care what the vector type is, only that it must have size larger than 0.

The third restriction can be expressed as follows.

Table 62: Constraint for the component type of vector

```
vector_type:component[_] = Comp ->
   integer_type(Comp);
   fp_type(Comp);
   pointer_type:integer(Comp);
   pointer_type:fp(Comp).
```

The 'pointer_type:integer' and pointer_type:fp' predicates are IDB predicates and have been declared as follows.

Table 63: The 'pointer_type:integer' predicate

```
pointer_type:integer(Type) <-
   pointer_type:component[Type] = Comp, integer_type(Comp).
```

The 'pointer_type:integer' predicate specifies that if 'Type' is of pointer type and has an integer type as its component then 'Type' is a pointer to integer.

Table 64: The 'pointer_type:fp' predicate

```
pointer_type:fp(Type) <- pointer_type:component[Type] = Comp, fp_type(Comp).
```

The 'pointer_type:fp' predicate specifies that if 'Type' is of pointer type and has a floating type as its component then 'Type' is a pointer to floating point.

Function type

A *function type* consists of a return type and a list of formal parameters. We declare the following predicates describing the function type.

Table 65: Function type predicates

```
function_type:return[Type] = Ret -> function_type(Type), type(Ret).

function_type:params[Type, Index] = Arg ->
   function_type(Type), int[64](Index), type(Arg).

function_type:nparams[Type] = Total ->  function_type(Type), int[64](Total).
```

The predicate 'function_type:return' defines that the return type of a function type is of type 'type'. The predicate 'function_type:params' defines both Type and Index variables as its key. It indicates the type of an argument at the specified index for the function type Type. The index is represented as a 64-bit integer. The predicate 'function_type:nparams' defines the number of arguments of the function type 'Type'.

For instance, if we have the function type *i32 (i32, i8**) we get the following tables.

Table 66: 'function_type:return' table

function type id	return type id
i32 (i32, i8*)	i32

Table 67: 'function:type:params' table

function type id	index	parameter type id
i32 (i32, i8*)	0	i32
i32 (i32, i8*)	1	i8*

Table 68: 'function:type:nparams' table

function type id	number of parameters
i32 (i32, i8*)	2

Constraints

A *function type* must also satisfy the following constraint:

- The return type is any type other than label and metadata.

Table 69: Constraint for the return type of function type

```
function_type:return[_] = Ret -> !metadata_type[ ] = Ret, !label_type[] = Ret.
```

Function

The next table shows how we declare the entity 'function' with refmode.

Table 70: The entity 'function'

```
function(Func), function:id(Func:Ref) -> string(Ref).
```

An LLVM function consists of a return type, a name, an argument list and a variety of optional attributes. We mention here some of the predicates we have declared.

Table 71: Function predicates

```
1. function:name[Func] = Name -> function(Func), string(Name).

2. function:type[Func] = Type -> function(Func), function_type(Type).

3. function:param[Func, Index] = Param->
      function(Func), int[64](Index), variable(Param).

4. function:nparams[Func] = Total -> function(Func), int[64](Total).
```

The first predicate holds the name of a function, which is a string value. The second holds the function type, which constitutes the function's signature. The third predicate indicates the

function formal parameters at a specified index. The last predicate holds the number of formal parameters in the function.

Constraints

One of the constraints that must be satisfied is that the number of formal parameters must match those declared in the function type. This can be declared by the following rule that computes the number of parameters in a function from the corresponding function type.

Table 72: Constraint for the function

```
function:nparams[Func] = Total <-
   function:type[Func] = Type, function_type:nparams[Type] = Total.
```

Instruction Operands

As mentioned previously, an instruction operand can be either a variable or a constant value that has a type. We declare the variable and the immediate entity types with refmode, as follows.

Table 73: Variable and immediate entities

```
variable(Var), variable:id(Var:Id) -> string(Id).

immediate(Imm), immediate:value(Imm:Id) -> string(Id).
```

Both variable and constant entities are represented as string values. We also declare two functional predicates to hold their type.

Table 74: 'variable:type' and 'immediate:type' predicates

```
variable:type[Var] = Type -> variable(Var), type(Type).

immediate:type[Imm] = Type -> immediate(Imm), type(Type).
```

We declare the entity 'operand'.

Table 75: The entity 'operand'

```
operand(Operand) -> .
```

Operands cannot be constructed directly from primitive values. Instead they are used to unify the concepts of variables and constants. For this reason we declare the 'operand' as a constructor of the variable and constant entities.

Table 76: The constructor for the entity 'variable'

```
operand:by_variable[Var] = Operand  -> variable(Var), operand(Operand).
lang:constructor(`operand:by_variable).
```

Table 77: The constructor for the entity 'immediate'

```
operand:by_immediate[Imm] = Operand -> immediate(Imm), operand(Operand).
lang:constructor(`operand:by_immediate).
```

The operand's type can be computed by the following IDB predicates.

Table 78: The 'operand:type' from variable

```
operand:type[Operand] = Type <-
  operand:by_variable[Var] = Operand,
  variable:type[Var] = Type.
```

Table 79: The 'operand:type' from immediate

```
operand:type[Operand] = Type <-
  operand:by_immediate[Imm] = Operand,
  immediate:type[Imm] = Type.
```

If the operand is constructed by a variable then its type is the variable's type. If it is constructed by an immediate then it has the same type as the corresponding immediate.

We also declare the following IDB predicates that compute whether an operand is a variable or an immediate value.

Table 80: The 'operand:as_variable' predicate

```
operand:as_variable[Operand] = Var <- operand:by_variable[Var] = Operand.
```

Table 81: The 'operand:as_immediate' predicate

```
operand:as_immediate[Operand] = Imm <- operand:by_immediate[Imm] = Operand.
```

3.3 Representation of the LLVM Instruction Set

The next table shows how we declare the entity 'instruction' with refmode.

Table 82: The entity 'instruction

```
instruction(Insn), instruction:id(Insn:Id) -> string(Id).
```

We represent all the useful information about the 'instruction' entity. For example:

1. Apart from instructions with void type, every instruction assigns its result to a variable.

   Table 83: The 'instruction:to' predicate

   ```
   instruction:to[Insn] = Var -> instruction(Insn), variable(Var).
   ```

   We can also compute the resulting type of an instruction with the following IDB predicate.

   Table 84: The 'instruction:type' predicate

   ```
   instruction:type[Insn] = Type <-
     instruction:to[Insn] = Var,variable:type[Var] = Type.
   ```

2. The sequence of instructions in a file is specified by the following predicate that given an instruction returns the next instruction (if it is not the last one).

   Table 85: The 'instruction:next' predicate

   ```
   instruction:next[Insn] = Next -> instruction(Insn), instruction(Next).
   ```

3. Every instruction has a function that contains it.

   Table 86: The 'instruction:function' predicate

   ```
   instruction:function[Insn] = Func -> instruction(Insn), function(Func).
   ```

We declare then the several LLVM instructions as subtypes of the entity type 'instruction' as shown in the table below.

Table 87: LLVM instructions

```
add_instruction(Insn) -> instruction(Insn).

bitcast_instruction(Insn) -> instruction(Insn).

load_instruction(Insn) -> instruction(Insn).

getelementptr_instruction(Insn) -> instruction(Insn).
```

Each instruction is identified by its id 'Insn'. The naming convention we follow for the predicate names of the instructions, is the concatenation of the instruction's opcode name with the suffix '_instruction'. We represent some of the LLVM instructions as follows.

Binary Operations

Binary and *bitwise binary* operators (add, sub, mul, and, or etc) require two operands of the same type and produce a resulting value of the same type as well. For example, the 'add' instruction can be represented as follows:

We declare the following predicates that hold the two operands of the 'add' instruction.

Table 88: The 'add_instruction:first_operand' predicate

```
add_instruction:first_operand[Insn] = Left ->
  add_instruction(Insn), operand(Left).
```

Table 89: The 'add_instruction:second_operand' predicate

```
add_instruction:second_operand[Insn] = Right ->
  add_instruction(Insn), operand(Right).
```

The 'add' instruction must also satisfy the following constraints:

1. Every 'add 'instruction must have two operands.

Table 90: Constraints for the 'add' instruction

```
add_instruction(Insn) -> add_instruction:first_operand[Insn] = _.

add_instruction(Insn) -> add_instruction:second_operand[Insn] = _.
```

2. The two operands must be integer or vector-of-integer. Both operands and the resulting value must have identical types.

Table 91: Constraint for the 'add' instruction type

```
add_instruction(Insn), instruction:type[Insn] = Type ->
  integer_type(Type); vector_type:integer(Type).
```

The 'instruction:type' predicate holds the type of the resulting value, which must be integer or vector of integer values type. We declare then that both operands must have the same type as the type of the 'instruction:type' predicate.

Table 92: Constraints for the 'add' instruction and 'instruction' type

```
instruction:type[Insn] = Type, add_instruction:first_operand[Insn] = Left ->
  operand:type[Left] = Type.

instruction:type[Insn] = Type, add_instruction:second_operand[Insn] = Right ->
  operand:type[Right] = Type.
```

Conversion Operations

Conversion operations (trunc, zext, sitofp, bitcast, etc) take a single operand and a type and produce a resulting value of the same type as the type argument. For example, the 'trunc' instruction can be represented as below.

Table 93: The 'trunc' instruction

```
trunc_instruction:from[Insn] = Value ->
  trunc_instruction(Insn), operand(Value).

trunc_instruction:to_type[Insn] = Type ->
  trunc_instruction(Insn), type(Type).
```

The above predicates hold the operand and the type argument of the 'trunc' instruction. We also compute the operand's type with the following rule:

Table 94: The 'trunc_instruction:from_type' predicate

```
trunc_instruction:from_type[Insn] = Type <-
  trunc_instruction:from[Insn] = Value,
  operand:type[Value] = Type.
```

The 'trunc' instruction must also satisfy a number of constraints:

1. Every 'trunc' instruction must have an operand and a type.

Table 95: Constraints for the 'trunc' instruction

```
trunc_instruction(Insn) -> trunc_instruction:from[Insn] = _.

trunc_instruction(Insn) -> trunc_instruction:to_type[Insn] = _.
```

2. The operand, the type argument and the resulting value must be integers or vectors (of the same number of integer values).

Table 96: Constraints for the 'trunc' instruction type

```
trunc_instruction:from_type[_] = Type ->
  integer_type(Type); vector_type:integer(Type).

trunc_instruction:to_type[_] = Type ->
  integer_type(Type); vector_type:integer(Type).

trunc_instruction:to_type[Insn] = Type  ->
  instruction:type[Insn] = Type.
```

The first two predicates specify that the operand and the argument type are of integer or of vector of integers type. The last predicate specifies that the instruction type (the resulting value type) must be the same as the argument type.

Table 97: Constraint for the resulting value type of 'trunc'

```
trunc_instruction(Insn),
  trunc_instruction:from_type[Insn] = From,
  trunc_instruction:to_type[Insn] = To
 ->
  vector:compatible(From, To).
```

The 'vector:compatible' IDB predicate derives the fact that two vectors are compatible if they have the same size.

Table 98: The 'vector:compatible' idb predicate

```
vector:compatible(Type1, Type2) <-
  vector_type:size[Type1] = Size,
  vector_type:size[Type2] = Size.
```

Terminator Instructions

Terminator instructions (ret, br, switch, invoke, etc) produce control flow and no values. The only exception is the 'invoke' instruction. We show below how we represent the 'ret' and the 'invoke' instructions.

The 'ret' instruction optionally accepts an operand, the return value. We declare the following predicates that form the 'ret' instruction.

Table 99: The 'ret' instruction

```
ret_instruction:void(Insn) -> ret_instruction(Insn).

ret_instruction:value[Insn] = Value ->
  ret_instruction(Insn), operand(Value).
```

The 'ret' instruction specifies also whether a function is well formed. A function is ill formed:

1. if it it has a non-void return type and contains a ‘ret‘ instruction with no return value or
2. if it has a return value with a type that does not match its type, or
3. if it has a void return type and contains a ‘ret‘ instruction with a return value.

We declare the following three rules to specify that a function is ill formed as shown in the table below.

Table 100: Rules that specify if a function is well formed

```
1. function:illformed(Func) <-
  void_type[ ] = Void,
  function:type[Func] = Type,
  function_type:return[Type] != Void,
  ret_instruction:void(Insn),
  instruction:function[Insn] = Func.

2. function:illformed(Func) <-
  void_type[ ] = Void,
  function:type[Func] = Type,
  function_type:return[Type] = Ret,
  Ret != Void,
  ret_instruction[Insn] = Value,
  operand:type[Value] != Ret,
  instruction:function[Insn] = Func.

3. function:illformed(Func) <-
  void_type[ ] = Void,
  function:type[Func] = Type,
  function_type:return[Type] = Void,
  ret_instruction(Insn),
  !ret_instruction:void(Insn),
  instruction:function[Insn] = Func.
```

The following rule can now derive the fact that a function is well formed if it is not ill formed (i.e., none of the above three rules is true).

Table 101: The 'function:wellformed' idb predicate

```
function:wellformed(Func) <- function(Func), !function:illformed(Func).
```

The 'invoke' instruction takes an operand that specifies the function where control transfers.

Table 102: The 'invoke' instruction

```
invoke_instruction:function[Insn] = Func ->
  invoke_instruction(Insn), operand(Func).
```

We specify two forms of the 'invoke' instruction, direct and indirect. In direct invocations the function operand is an immediate value and in indirect invocations a variable. The operand's type is the function's signature. We declare these two forms as subtypes of the 'invoke' instruction as below:

Table 103: The direct and indirect 'invoke' instruction form

```
direct_invoke_instruction(Insn) -> invoke_instruction(Insn).

indirect_invoke_instruction(Insn) -> invoke_instruction(Insn).
```

We also declare the following IDB predicates that compute whether an 'invoke' instruction is direct or indirect.

Table 104: Idb predicates for the 'invoke' instruction

```
direct_invoke_instruction(Insn) <-
  invoke_instruction:function[Insn] = Func,
  operand:as_immediate[Func] = _.

indirect_invoke_instruction(Insn) <-
  invoke_instruction:function[Insn] = Func,
  operand:as_variable[Func] = _.
```

The 'invoke' instruction takes a list with the function's actual arguments.

Table 105: The 'invoke_instruction:arg' predicate

```
invoke_instruction:arg[Insn, Index] = Arg ->
  invoke_instruction(Insn), int[64](Index), operand(Arg).
```

We also compute the function signature and its return type with the following IDB predicates as shown n the table below.

Table 106: The 'invoke_instruction:signature' idb predicate

```
invoke_instruction:signature[Insn] = Type <-
  invoke_instruction:function[Insn] = Func,
  operand:type[Func] = Type.

invoke_instruction:return_type[Insn] = Type <-
  invoke_instruction:signature[Insn] = Signature,
  function_type:return[Signature] = Type.
```

The 'invoke' instruction takes also two labels, a 'normal' and an 'exceptional label.

Table 107: The 'invoke' instruction labels

```
invoke_instruction:normal_label[Insn] = Normal ->
  invoke_instruction(Insn), variable:label(Normal).

invoke_instruction:exception_label[Insn] = Exception ->
  invoke_instruction(Insn), variable:label(Exception).
```

The 'invoke' instruction must satisfy a number of constraints. We mention some of them below:

1. Every 'invoke' instruction must have a 'normal' and an 'exceptional' label'.

Table 108: Constraints for the 'invoke' instruction labels

```
invoke_instruction:normal_label[_] = Normal ->
  variable:label(Normal).

invoke_instruction:exceptional_label[_] = Exceptional ->
  variable:label(Exceptional).
```

2. The type of the 'invoke' instruction s also the type of the return value. Functions that return no value are marked void.

Table 109: Constraint for the 'invoke' instruction return value type

```
invoke_instruction:return_type[Insn] = Type, void_type[] != Type ->
  instruction:type[Insn] = Type.
```

As mentioned earlier, the 'instruction:type' predicate holds the type of the resulting value for each instruction. In this case, the 'invoke_instruction:return_type' predicate specifies that if

the return type of the function's signature is not void then it must be the same as the type of the 'invoke' instruction. This can also be declared as below:

Table 110: Constraint for the 'invoke' instruction return type

```
invoke_instruction:return_type[Insn] = Type, void_type[ ] = Type ->
  !instruction:type[Insn] = _.
```

If the return type is void, then the 'invoke' instruction must return no type.
Aggregate Operations

As mentioned previously, arrays and structures are considered aggregate types. Aggregate operations comprise the instructions 'insertvalue' and 'extractvalue'. We describe how we represent the 'extractvalue' instruction.

The 'extractvalue' instruction extracts the value of a member field from an aggregate value. It takes an operand that represents the aggregate value.

Table 111: The 'extractvalue' instruction

```
extractvalue_instruction:base[Insn] = Aggregate ->
  extractvalue_instruction(Insn), operand(Aggregate).
```

The 'extractvalue' instruction takes also a number of constant indices that specify which value to extract.

Table 112: Predicates for the indices of the 'extractvalue' instruction

```
extractvalue_instruction:index[Insn, Index] = Idx ->
  extractvalue_instruction(Insn), int[64](Index), immediate(Idx).

extractvalue_instruction:nindices[Insn] = Total ->
  extractvalue_instruction(Insn), int[64](Total).
```

We also declare the 'extractvalue_instruction:nindices' predicate that holds the total number of the indices.

The result is the value at the position in the aggregate specified by the indices. We can get the resulting type for each dereference with the following rules as they compute the type of the value of the aggregate at the specified index.

At position 0 the type is the same as the type of the operand argument. When the aggregate is of array type we compute the resulting type for each dereference by its component type.

Table 113: Rules to compute the resulting type for each dereference in the 'extractvalue' instruction

```
extractvalue_instruction:interm_type[Insn, 0] = Type <-
  extractvalue_instruction:base[Insn] = Aggregate,
  operand:type[Aggregate] = Type.

extractvalue_instruction:interm_type[Insn, Index + 1] = Comp <-
  extractvalue_instruction:interm_type[Insn, Index] = Type,
  array_type:component[Type] = Comp.
```

In order to determine the exact field of the struct type that we're trying to extract, we must be able to compute the exact integer values that would be produced if we were to evaluate the indexing constants, so that we can match against the fields of the operand argument. We declare the following predicate that represents a map of [constant -> int]. For each constant index, it returns its integer value.

Table 114: The 'constant:toInt' predicate

```
constant:toInt[Constant] = Value -> immediate(Constant), int[64](Value).
```

Now we can compute the resulting type for each dereference by the following rule.

Table 115: The 'extractvalue_instruction:interm_type' idb predicate

```
extractvalue_instruction:interm_type[Insn, Index + 1] = Comp <-
  extractvalue_instruction:interm_type[Insn, Index] = Type,
  extractvalue_instruction:index[Insn, Index] = Constant,
  constant:toInt[Constant] = Value,
  struct_type:field[Type, Value] = Comp.
```

The following rule computes the type of the 'extractvalue' instruction, which is the type of the value at the position specified by the indices.

Table 116: The 'extractvalue_instruction:value_type' idb predicate

```
extractvalue_instruction:value_type[Insn] = Type <-
  extractvalue_instruction:nindices[Insn] = Total,
  extractvalue_instruction:interm_type[Insn, Total] = Type.
```

This type must be the same as the type of the 'instruction:type' predicate for a given id.

Table 117: The 'extractvalue_instruction:value' predicate

```
extractvalue_instruction:value_type[Insn] = Type ->
  instruction:type[Insn] = Type.
```

Another constraint that must be satisfied is that the operand must be a value of struct or array type.

Table 118: The 'extractvalue_instruction:base_type' idb predicate

```
extractvalue_instruction:base_type[Insn] = Type <-
  extractvalue_instruction:base[Insn] = Aggregate,
  operand:type[Aggregate] = Type.
```

We compute the type of the operand value as above and then we declare the constraint:

Table 119: Constraint for the base type of the 'extractvalue' instruction

```
extractvalue_instruction:base_type[_] = Type -> aggregate_type(Type).
```

We represent the rest of the instructions with the same principles as described above.

Conclusion

In this project we introduced a relational representation for the LLVM IR, a low level language with strictly defined semantics. We presented the type system and the instruction set of the language and how they can be represented relationally into a Datalog workspace.

Abbreviations

SSA	Static Single Assignment
SIMD	Single Instruction Multiple Data
ACM	Association for Computing Machinery
IR	Intermediate Representation
EDB	Extensional Database
IDB	Intentional Database

References

[1] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In PODS ’05: Proc. of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 1–12, New York, NY, USA, 2005. ACM.

[2] T. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, Applications of Logic Databases, pages 163–196. Kluwer Academic Publishers, 1994.

[3] Y. Smaragdakis, M. Bravenboer, “Using Datalog for Fast and Easy Program Analysis”, Datalog Reloaded:Datalog 2.0 post-workshop proceedings, 2010

[4] M. Bravenboer, Y. Smaragdakis, “Strictly Declarative Specification of Sophisticated Points-to Analyses”, OOPSLA, 2009

[5] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In PLDI ’04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation, pages 131–144, New York, NY, USA, 2004. ACM.

[6] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In ICSE ’08: Proc. of the 30th int. conf. on Software engineering, pages 391–400, New York, NY, USA, 2008. ACM.

[7] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with Datalog. In Proc. European Conf. on Object-Oriented Programming (ECOOP), pages 2–27. Spinger, 2006.

[8] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In K. Yi, editor, APLAS, volume 3780 of Lecture Notes in Computer Science, pages 97–118. Springer, 2005.

[9] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In PLDI ’04: Proc. of the ACM SIGPLAN 2004 conf. On Programming language design and implementation, pages 131–144, New York, NY, USA, 2004. ACM.

[10] http://llvm.org/docs/LangRef.html [Accesed 20/01/2014]

[11] http://www.logicblox.com/technology.html [Accesed 20/01/2014]