GBIL: Generic Binary Instrumentation Language

Language Reference Manual

By: Andrew Calvano

COMS W4115 Fall 2015 CVN

Table of Contents

1.) Introduction

GBIL (Generic Binary Instrumentation Language) is a domain specific language for binary instrumentation. Its purpose is to lower the development time required to develop a binary instrumentation tool to analyze compiled software. It allows a developer to specify an analysis to perform during a program's execution.

Previously, a developer would have to learn a very specific and often low-level library to specify the same analysis. Expressing an analysis in the GBIL language not only makes it easier for the developer to get their tools up and running faster, it also allows for targeting multiple instrumentation platforms such as Intel PIN, DyanmoRIO, Dyninst, debuggers, or even emulator plug ins.

This document describes the details of the GBIL language and will serve as a reference manual for a GBIL developer.

2.) Lexical Conventions

Tokens

Whitespace

Whitespace characters in the GBIL language include the newline character '\n', the carriage return character '\r', the tab character '\t' and the space character ' '.

Whitespace is unimportant except to distinguish between token boundaries and has no semantic meaning.

Comments

Comments in the GBIL language exist so that the developer can annotate his or her code for reference at a later time. Comments begin with the lexeme “/*” and end with the lexeme “*/”.

Comments can span multiple lines. Nested comments are not allowed.

Literals

There are four types of literals in the GBIL language: string literals, uint literals, sint literals, and float literals.

It is important to note that there are no map, list, or file literals at this time.

String Literals

String literals are a sequence of ASCII characters enclosed by double quotes. A string literal looks like the following “string literal”.

The regular expression for a string literal is the following.

\”[ASCII Character Set]*\”

Note that the string literal can be empty. A string literal has type string and can be used in all operations expecting a string type operand.

uint and sint Literals

uint and sint literals represent instances of the uint and sint types.

uint and sint literals take the exact lexical form with the exception that an sint literal can optionally have a “-” token in front of it to signify whether it is negative or positive.

These lexemes consist of one or more digits in either hexidecimal or decimal form.

There regular expressions for describing and uint and sint literal is the following:

uint

0x[0-9A-Fa-f]+ | [0-9]+

sint

[-]?0x[0-9A-Fa-f]+ | [-]?[0-9]+

float Literals

float literals take the exact form as they do as described in “The C Programming Language 2nd” edition by Kernighan and Ritchie and as described in Homework 2.

Identifiers

Identifiers can be used to name variables. Identifiers can consist of alphabetic and numeric characters as well as the underscore character but can not start with a numeric character.

The regular expression for identifiers is the following:

[A-Za-z_][0-9A-Za-z_]+

The only restriction on identifiers is that an identifier can not match any of the keywords.

Special Identifiers

Special identifiers are used to reference context specific attributes of an instrumentation scope.

For example, a special identifier can be used to access the mnemonic of a disassembled assembly instruction in the instruction instrument handler or the address of a given register at the time the handler is triggered.

Special Identifiers take exactly the same form as identifiers with the exception that they must start with the “$” character.

Examples of some special identifiers include:

$bb_begin, $eax, $mnem, and $op0_type.

A complete list of built in special identifiers paired with their corresponding instrumentation scopes is listed below:

Across all instrumentation scopes the registers of the target architecture can be accessed as the following:

$eax, $ebx, $ecx, $eip, etc.

These will return the value of the register as a uint at the time the instrumentation handler is executed.

Instruction Instrumentation Scope

$mnem – returns the mnemonic for the instruction as a string. e.g. “add”

$text – returns the disassemble instruction and its operands as a string. e.g. “add ecx, ebx”

$op_count – returns the number of operands as a uint.

$opX_type – returns the type of the operand as a string. e.g. “register”, “memory”, “constant”. X is the index of the operand.

$opX_addr – return the address of a memory operand, can not be used with any other operand type.

e.g. [ebx + ecx*4] will return 0x8048000 if ebx + ecx*4 evaluates to 0x8048000.

$opX_mem_text – return the memory expression as a string of a memory operands

e.g. [ebx + ecx*4] will return “[ebx +ecx*4]”

$opX_mem_op_count – return the number of operands in the memory operand expression.

e.g. [ebx + ecx] will return 2.

$opX_mem_opX_type – return the type of an operand in a memory expression.

$opX_mem_opX_val – returns the value of an operand in a memory expression.

$opX_mem_scale – returns the scale of a memory expression.

e.g. [ebx + ecx*4] will return 4.

$opX_val – returns the value of the operand as a string e.g. if it is a register it will return the value of the register, if it is memory it will return the value at that memory location.

Basic Block Instrumentation Scope

$num_instrs – returns the number of instructions in the basic block.

$instrs – returns a list of instruction objects. Instruction objects can not be explicitly created by a developer they must be returned from a special variable access. A list of instructions can be passed to a utilty function for further processing or can iterated over in the handler in which it was accessed.

$last_instr – returns an instruction object representing the last instruction in the basic block. This is useful for collecting things like jmp or ret target addresses.

Function Instrumentation Scope

$instrs – returns list of instruction objects contained in the function boundaries

$basic_blocks – returns a list of basic block objects contained in the function. Basic block objects are like the instruction objects. They can not be defined by the used only collected from the context of an instrumentation routine.

$num_blocks – returns the number of basic blocks

$num_instrs – returns the number of instructions.

Operators

Operators are used in expressions to perform some specified behavior and either return the result or perform some side effect.

There are two types of operators in the GBIL language: polymorphic operators and operators that can be perform only on one specific type.

The operator lexemes consist of the following:

“*” - Multiplication binary operator or the unary dereference operator.

“+” - Addition operators

“-” - Subtraction operator

“/” - Division operator

“&&” - Logical And operator

“||” - Logical Or operator

“!” - Unary logical not operator

“=” - Assignment operator

“!=” - Inquality operator

“==” - Equality operator

“<” - Less than operator

“<=” - Less than or equal operator

“>” - Greater than operator

“>=” - Greater than or equal operator

“in” - List or map membership operator

“not” - Can be used with “in” operator to negate its evaluation.

“[]” - Map access operator

“.” - Dot operator

Keywords

Type Keywords

Type keywords can be used to declare a variable type or to specify a type of an argument to a utility function.

The type keywords are the following:

“uint”, “sint”, “string”, “float”,
“string”, “file”, “list”, “map”
“bool”, “instruction”, “basic_block”, “function”

Function Type Keywords

A function type keyword is used in a function definition. There are multiple types of functions in the GBIL language that correspond to the context and time at which the defined function executes.

The function type keywords are:

“@init”, “@term”, “@instruction”, “@basic_block”, “@function”, and “@utility”

Control Flow Statement Keywords

There are keywords in the GBIL language which are used in the bodies of function to express control flow.

These keywords are:

“if”, “else”, “for”, and “while”, “return”, “break”, “continue”

Built in Function Keywords

There are built in function that exist in the GBIL language that operate on some specific types

The built in function are identified as follows:

“len” - used to take the length of a list or a map

“append” - used to append an object to a list

“remove” - used to remove an object from a list

“pop” - pop an element off the top of the list.

“write” - writes a string to a file

“read” - reads from a file and returns a string representing the file's contents.

Other Tokens

There are also some miscellaneous tokens that will be described here that have no been mentioned previously.

Parenthesis “(“ and “)”

The parenthesis tokens can be used in a unary * dereference expression to separate an expression to evaluate to an address to retrieve memory from.

Parenthesis tokens are also used in utility function definitions to contain a list of parameters to that function.

Semicolon “;”

The semicolon token is used to separate statements.

Brackets “{“ and “}”

Brackets are used to enclose the body of a function and to define bodies for control flow statements.

3.) Syntax

Program Structure

The structure of a GBIL program consists of an optional list of global variable declarations preceding a set of function definitions. Each function definition consists of a list of optional local variable declarations preceding a list of statements.

Each function type denotes the context and/or execution order for that specific type. For example, the @instruction type corresponds to an instrumentation routine for an instruction encounter during the program's execution, the @basic_block to a basic block encounter, and the @function to a function encounter.

There can exist multiple instrumentation routines in a GBIL program. Each instrumentation routine must be preceded by a logical predicate that decides if the routine is to be executed based on the context of the event. The predicate has access to the same context and special variable as the function type it is preceding and must return a boolean value. If multiple routines' predicates evaluate to true then all will be executed (order is in the order defined in the GBIL program).

For each instrumentation routine a definition can be placed both before and after the function type keyword. The definition above the function type keyword corresponds to the actions to take before the instruction/basic block/function is executed, while the definition afterwards corresponds to the actions to take after the instruction/basic block/function has executed.

There can be an arbitrary number of utility functions in the GBIL program that can be called from any of the instrumentation routines, the initialization routine, or the termination routine.

There can only be one initialization routine and one termination routine for any GBIL program.

Variable Declarations

Variables in the GBIL language can not be declared and defined at the same time and can not be intermingled with statements. Variables must be declared first and then defined in a statement following the declaration.

This means something like: uint test = 3; is not valid. You would have to do express something like: uint test; test = 3;.

For the non-polymorphic types: uint, sint, float, string, file, bool, instruction, basic_block, and function, a variable declaration consists of a type name followed by an identifier. A non polymorphic type variable can be declared as in the following:

type_name identifier;

For polymorphic types: list and map. The variable declaration is specific to the type.

A list variable declaration takes the following form:

list<type> identifier;

The list variable declaration can contain other polymorphic types such as list or map. If this is the case the declaration looks like the following:

list<list<type>> identifier; or list<map<type,type>> identifier;

A map variable declaration takes the following form:

map<type, type> identifier;

Either type can be a polymorphic type so you can have maps of maps to lists and a number of recursive type declarations.

map<list<uint>,map<list<uint>,sint>> identifier;

is a valid declaration.

Function Definitions

There are multiple types of functions in the GBIL language and each has its own definition syntax.

@init and @term function definitions

The @init and @term function types correspond to functions that execute upon initialization and termination. These functions take no arguments and have a single definition consisting of a list of variable declarations and a list of statements.

A template for @init and @term

```
@init
{
        variable declaration list

        statement list
}
```

@utility

The @utility function type is used to separate logic and make more complex tasks easier to understand. An @utility function can be called from the context of any of the other function types including an @utility type. This also means that @utility functions support recursion.

@utility functions are also the only functions that return values and accept arguments. There is no limit on the number of parameters used in an @utility definition. Its return type can be any of the built in types including polymorphic types.

@utility functions are also the only function types that an identifier can be attached to.

The syntax for an @utility function is as follows:

```
return_type @utility identifier(arg_type_0 identifier, …, arg_type_n identifier)
{
        variable_declaration_list;

        statement_list;
}
```

The return_type and argument list is optional. If these are not present the syntax can be something like:

“@utility identifier(arg1, arg2,argn)” or “return_type @utility identifier()” or simply “@utility identifier”

@instruction, @basic_block, and @function

@instruction, @basic_block, and @function types correspond to handlers for instrumentation events.

Each of these function types have access to a set of special variables which are representative of attributes that are commonly queried from within these instrumentation scopes. For example, an @instruction function type can access special variables that return information about the encountered instruction.

Each instrumentation function type has the following structure:

```
if predicate
{
}
@type
{
}
```

The purpose of the predicate is to decide at the instrumentation event whether the handler is executed or not. If the predicate evaluates to True then the handler will execute,if False it will not. The predicate has access to the same special variables as the rest of the handler routine.

The block of code before the type keyword corresponds to code that executes as soon as the event is triggered and before the function, basic_block, or instruction has executed. The block of code after the type keyword corresponds to code that should execute after the instruction, basic block, or function has finished executing. Each handler block consists of a list of variable declaration and a list of statements.

Expressions

Expressions are the building blocks to express things to be done in the language. Expressions allow the developer to specify things such as arithmetic and side-effect operations such as variable assignment, output, and data structure manipulation.

Add Expressions

expr + expr

Add expressions perform addition on two operands and return the result. Add expressions support the uint, sint, string, and float types. Therefore, the left hand and right hand side of the add expression must evaluate to one of these types and both the left hand and right hand side must match if either is a string type. Otherwise, uint and float, sint and float, and float and float can all be operated on. Uint and sint can not be evaluated together.

Minus Expressions

expr – expr

Minus expressions perform subtraction on two operands and return the result. Minus expressions support the uint, sint, and float types. Both the left hand side and right side must evaluate to one of these types. The rules for which types are valid to operate on are the same as add expressions. The type returned will be the type of the left hand side.

Multiplication Expressions

*expr * expr*

Multiplication expressions perform multiplication on two operands and return the result. Multiplication expressions support the uint, sint, and float types as operands. The rules for which types are valid to operate on are the same as add expressions. The type returned will be the type of the left hand side.

Division Expressions

expr / expr

Division expressions perform division on two operands and return the result. Division expressions support the uint, sint, and float types as operands. The rules for which types are valid to operate on are the same as add expressions. The type returned will be the type of the left hand side.

Less Than Expressions

expr < expr

Less Than expressions operate on two operands and returns a boolean result. If the left hand side is strictly less than the right hand side the boolean value returned is True otherwise the value returned is False.

Less Than expressions support the following types: uint, sint, and float. Uint and float, and sint and float can be compared with each other but uint and sint can not be.

Less Than or Equal Expressions

expr <= expr

Less Than or Equal expressions operate on two operands and return a boolean result. If the left hand side is less than or equal to the right hand side True is returned otherwise False is returned.

Less Than or Equal expressions support the following types: uint, sint, and float. The rules for the operand types are the same as the less than expression.

Greater Than Expressions

expr > expr

Greater Than expressions operate on two operands and return a boolean result. If the left hand side is strictly greater than the right hand side, the boolean value returned is True otherwise it is False. The rules for operand types are the same as less than expressions.

Greater Than or Equal Expressions

expr >= expr

Greater Than Or Equal expressions operate on two operands and return a boolean result. If the left hand side is strictly greater than or equal to the right hand side, the boolean value returned is True otherwise it is False. The rules for operand types are the same as less than expressions.

Logical And Expressions

expr && expr

Logical And Expressions operate on two boolean and operands and return a boolean value. The boolean value returned is the same as the logical and from boolean algebra.

Logical Or Expressions

expr || expr

Logical Or Expressions operate on two boolean and operands and return a boolean value. The boolean value returned is the same as the logical or from boolean algebra.

Logical Not Expressions

! expr

Logical Not Expressions operate on a single operand that evaluates to a boolean type. The value returned is the same as the logical not from boolean algebra.

Logical Equality Expressions

expr == expr

Logical Equality Expressions operate on two boolean operands (returned from left hand side and right hand side expressions) and returns a boolean value of True if the left hand side boolean value equals the right hand side boolean value or False if they are different.

Logical Inequality Expressions

expr != expr

Logical Inequality Expressions are exactly the opposite of the Logical Equality Expressions.

Dereference Expressions

**(expr)*

Dereference expressions evaluate the interior expression to a uint and returns the memory contents at the uint address in the instrumented process. Dereference expressions can only be present in an @function, @basic_block, @instruction, or an @utility function with a function, instruction, or basic_block type as an argument. This is because the target process must currently be instrumented in order to query its address space for the address.

If the address is invalid, an exception will occur in the compiled tool.

In Expressions

expr in expr

The in expression is used to check if a value is in a list or a map and will return a boolean value of True if present or False if it is not present. The right hand side of the in expression must be a list or map type. The left hand side must be the type of the list or the type of the map's key. This expression evaluates to true if the value on the left hand side is present in the list or is a key in the map of the right hand side.

Not In Expressions

expr not in expr

The not in expression is the same as the in expression excepts that it will return False if the left hand side is in the right hand side and True if it is not.

Dot Expressions

identifer.identifer

The dot expression can be used to call a built in method from a map or list type or to access an attribute of a function, basic_block, or instruction object that would normally be accessed as a special variable.

Map [] Expressions

identifer[expr]

The map[] expr is used to access a value corresponding to a key in a map object. The expr inside of the brackets must be the same type as the key type of the map. The value returned is the value type.

If the value returned by the expr is not in the map, an exception will occurs in the compiled tool. Because of this, a developer should use the “in” or “not in” expression to test for the key before access.

Function Call Expressions

func_args = type_name identifier

| type_name identifier, func_args

identifier(func_args)

Function call expressions are used to call utility functions. The value return is of the type

specified as the return type. If not return type is present in the utility function definition then nothing is returned. The function arguments must be in the same order and of the same types as specified in the @utility function definition.

Literal Expressions

identifer
| *uint_literal*
| *sint_literal*
| *float_literal*
| *string_literal*

Literal expressions simply return the value of the literal or the value contained in the identifier. The type returned is the type of the literal or the type of the identifier. The identifier must have been declared.

Predicates

Predicates combine expressions together with the requirement that the resulting evaluation produces a boolean value. Predicates are used with instrumentation handlers to decide during the program's execution whether or not the described instrumentation handler should be executed. The predicate construct exists to allow the developer to filter the instrumentation routines they want to execute based on the context of the instrumentation event.

Predicates take the following form:

if expr

where expr must return a boolean value of True or False.

Statements

Statements act as “sentences” in the GBIL language they put together expressions and possibly other statements to state what the program must do in different situations.

General Statements

expr ;

A general statement is simply an expression followed by a semicolon. The general statement is mostly used to perform a side-effect such as manipulating data structures, or outputing text to a file.

Assignment Statements

identifier = expr ;
| **(expr) = expr ;*
| *sidentifer = expr ;*

Assignment statements exist to write the return value of an expression to a variable or to memory in the instrumented process.

The "*identifer = expr ;*" form writes the return value of the right hand side expression to the variable denoted by the identifier on the left hand side.

The "**(expr) = expr ;*" form exists to evaluate the expression on the left hand side dereference the address in the instrumented process and write the return value of the right hand side of the expression to this location.

The "*sidentifer = expr ;*" form will write the return value of the right hand side expression to the sidentifer variable on the left hand side. This is used to write a value to a register. e.g. "*$eax = 0x5;*"

If Else Statements

if (expr) { statement_list; } else { statement_list; }

The if else statement exists to allow the developer to branch control flow if a specified condition evaluates to true or false. The expr argument to the if-else statement must be an expression that evaluates to a boolean value. If the expression has a boolean value of true the first code block is executed, otherwise the second code block is executed.

The if-else statements can be nested. This is how you would have to implement if-else-if style constructs.

For Statements

for (assign_stmt; expr; expr) { statement_list; }

The for loop statement is an iteration construct that allows the developer to specify how many times a given loop should execute based on an initial condition a conditional expression and an update expression.

It works mostly the same way a for loop works in the C language.

While Statements

while (expr) { statement_list; }

The while loop statement is another iteration tool. This works exactly the same way as the

while loop from the C language.

Semantics

Types

bool

A type representing boolean values. Its values can be either True or False.

uint

A type representing unsigned integers.

sint

A type representing signed integers.

string

A type representing strings.

file

The file type corresponds to a file handle on the underlying operating system. The name of the file on disk corresponds to the identifier of the file variable, the directory where it is stored will be the current directory where the compiled tool is running. File identifier support two methods “read” and “write”. Read takes no arguments and returns a string representing the file's contents. The “write” routine takes a string argument that it will write to the file and an additional argument that can be “w” or “a” that tells it whether it should write over the file or append to the file.

instruction

The instruction type corresponds to the context for an instruction level instrumentation routine. This is a special type whose instances are returned from the $instrs special variable from the basic block and function level instrumentation contexts. An instance of this type can be passed to a utility function that expects an instruction type as one of its argument. Instances of this type are also not mutable and can not be changed once they are instantiated.

The special variables for the instrumentation routine are essentially short hand for accessing a field of an instance of this object. To access a field you usually access from a special variable in the instrumentation routine you would use the “.” operator.

For example, to access the $mnem special variable that returns the mnemonic as text from an instance of the instruction type you would write:

instr.mnem

basic_block

The basic_block type is very similar to the instruction type. It has a different set of fields that area accessible via the “.” operator because it represents a different type of instrumentation context. It has the same type of fields as you do special variables inside of the basic block handler context.

function

The function type is similar to the instruction and basic_block types and corresponds to a function instrumentation scope. The “.” operator allows you to access all of the function specific attributes you would be able to access in a function instrumentation scope.

list

The list type is a container type. A list object can be initiated by supplying the type of what the list is to contain and an identifier. All members contained in the list object must be of the same type. The list type is recursive meaning we can have a list of lists.

There are a few built in functions for the list type. These are the “len”, “remove”, “append”, and “pop” functions. To call one of these functions you must use the “.” operator with an identifier that has the list type. The “len” function returns the number of elements contained in the list. The remove function takes an argument of the same type as the list and removes all elements in the list with that type. The “append” built in function takes an argument with the same type as the list and pushes the argument to the end of the list. The “pop” function will remove the last element on the list and return it.

Map

The map type is also a container type. It contains a list of key value pairs. When a map object is declared the type of the key and the type of the value is declared. If a key is used that is not of the correct type a compilation error will occur. Map's can be recursive and can contain another container type such as a list type.

Maps have a built in operator [], that allows a developer to specify a key and retrieve the corresponding value from the map. The “in” and “not in” expressions are also supported with the map container type. These behave the same way as they do with lists except they can only be used to test against a key's presence in the map.

Scope

Scoping rules are straightforward. Local declarations override global declarations. Global declarations are carried over to all parts of the program. Local declarations are only in scope for the block and sub-blocks from the statement lists in which they correspond.

Semantic Errors

This section will describe potential errors that are not caught by the lexer or parser, and how the GBIL language handles them. There are numerous cases that correspond to a program that is syntactically correct yet semantically invalid. In the case where a semantically invalid program is not detected at compile time, an uncaught exception in the compiled tool will occur. Exceptions are not built into the GBIL language at this time. Some example of errors that the GBIL compiler can not catch at compile time are described below.

If a special variable that corresponds to a specific type of argument in an instruction handler such as $opX_mem_text when the X operand in the instruction is not memory, this will not be caught at compile time. Instead, there will be a check in the compiled code for this type of error and if the operand is not a memory operand then an exception will occur. Because of this it is beneficial for the GBIL developer to put in checks for the type of the operand before using accessing a type specific field such as $opX_mem_text.

Another situation that can occurs it the querying of an address that is not present in the target process address space. This will be handled as described above.

Errors such as using operators on unsupported types will be caught at compile time through the use of a type checking system. Errors will be reported to the user.