FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

DEPARTMENT OF COMPUTER SCIENCE

Bachelor thesis

Static Assertion Checking Optimization in a Janus-like programming language

Jonas Wolpers Reholt <jsf130@alumni.ku.dk>

Supervisor
Robert Glück <glueck@di.ku.dk>

Abstract

Contents

1 Introduction

Reversible programming languages allow the programmer to write programs, that can be run both forward and backwards. Getting this ability into a language impacts the basic structure such that assignments, conditional operations, and loops must be handled in another way than traditional languages do. The reversibility of the programs does however have the benefits of:

- Lowering side-channel attack, as it does not constantly delete memory to make room for new information [FIND CITATION], thereby creating programs that generates a more constant stream of heat.
- Removing the theoretical lower limit of heat generation [FIND CITATION], making way for lowering the energy usage of computers (This does however require computers build for reversibility).
- Models the world of physics more precise, as physics in itself is reversible [FIND CITATION].
- Only needing to write one program, when functionality such as zip/unzip, that has a natural inversion that you want. This allows the programmer to write one program, prove correctness of one program, and then ship ”two” programs.

This project focuses on the last item for two reasons: 1) that the majority of computers today are not reversible, and 2) that translating from a reversible language to an irreversible generates certain overhead, as i.e. `if`-statements need both an entering condition and an exiting assertion for reversibility; meaning the program needs to check an extra assertion each time an `if`-statement is run. This assertion is ends up taking approximately 7 instruction, containing a conditional jump removing linearity from the code, when translating to `x86-64` assembly code. E.g. the simple dummy function:

```
    void f(int a)
    {
        assert(a == 0);
    }
```

Gets the assertion translated, using gcc version 11.2, into:

```
    ...
    cmp     DWORD PTR [rbp-4], 0
    je      .L3
    mov     ecx, OFFSET FLAT:.LC0
    mov     edx, 5
    mov     esi, OFFSET FLAT:.LC1
    mov     edi, OFFSET FLAT:.LC2
    call    __assert_fail
.L3:
    ...
```

Meaning 7 instructions could be optimized away, herein including a conditional jump, that requires jump prediction to run efficiently.

1.1 Boundaries

The focus of this project is on finding whether a theorem prover is available for making static program analyse in compile time, checking whether these extra assertions can be removed. Hence there will be no other focus on optimization in the code generation

Also as this is a bachelor thesis the focus will be on freely available theorem provers.

2 Janus-like Language Definition

This project alters and extents the syntax for `Janus` to make it better match modern programming languages. This ”new” language will be an imperative reversible language like `Janus`, and henceforth will be noted as `JAPA` (short for Janus alike programming alternative) in this report. But before altering the syntax and functionality of `Janus`, a more formal characterizations of reversibility must be made. Reversible computation refers to the use of logically reversible transformations [1], and require two things: Local reversibility and global reversibility.

Local reversibility

Definition 2.1. For a local process to be reversible information from step to step must be preserved, meaning no information is destroyed [2].

Global reversibility

Definition 2.2. A global process can only be reversible if the mapping from a start state to a final state is bijective [2].

This means that every construct of `JAPA` must preserve information over time, so the program knows where to start of when running in either direction, and that there must be a ”one-to-one” transformation from every state. e.g. consider the assignment operator `var = 10`. This statement has no way of undoing itself, as we do not know what `var` was before this assignment, meaning it goes against definition 2.1. It also removes the bijective property of the program, as all previous states of `var` leads to the same final state where `var == 10`.

The definition for logical reversibility stated above does allow for all programs to be reversible, as one could simply save a copy of the initial state and final state, as this would make the program bijective, as the final program becomes a tuple $(state_{initial}, state_{final})$, making the program transformation $\{state_{initial}\} \rightarrow \{state_{initial}, state_{final}\}$. These program will however not live up to Landauer's Principle, if every individual step is not reversible. It is therefore imminent that all construct of `LR` are reversible in them self.

2.1 Data Types

So far `JAPA` only support the same data types as `Janus`, which is integers and arrays of integers, where integers are 32 bit positive integers. The constraint with extending the type system to e.g. floats is, that the data type is imprecise, meaning no perfect inverse is ensured to exist.

Together with arrays come a set of library functions, which at the moment contains:

- `length(x)` gives the length of the array x.

2.2 Declaration of Variables

There are essentially two types of variables: Global and local. Because a global variable is always in scope, it preserves the information at all time (assuming all operations done on it are reversible). Hence global variable declaration can simply be done by:

$$\begin{array}{lcl} \texttt{<id>} & \xleftrightarrow{\text{Inverse of}} & \text{no inverse} \\ \texttt{<id>[]} & & \end{array}$$

Local variables are mote tricky, as they will leave the scope, meaning we cannot know the value of the variable when running the program backwards, if the variable have left scope. Hence there needs to be some operation stating what the value of the variable is when it is last used, so the program step of leaving the scope of a local variable also becomes reversible. This can be done in the following way:

$$\texttt{<id>} \xleftrightarrow{\text{Inverse of}} \texttt{delocal <id> == <expr}$$

It is important to note, that declaring a local, does not open a scope, meaning when a variable y is shadowed by a local x, it is not brought into the light by deallocation of x.

2.3 Modification Arithmetics

To modify a variable there needs to be a inverse function of the operator, such that the modification can be reversed. As the only datatype for `JAPA` is integers, the only operators with an inverse (the function is bijective) is addition, subtraction, and exclusive or. Multiplication and division cannot be used, as they are not each others inverse when operating on natural numbers. Hence the operation becomes:

```
<id> += <expr1>
<id> ^= <expr2>
```

$\xleftrightarrow{\text{Inverse of}}$

```
<id> -= <expr1
<id> ^= <expr2>
```

Where ^ indicates exclusive or.

2.4 Arithmetics

Performing arithmetics on expressions of the language does not require, that the arithmetic function is bijective, as it is only the state transformations that need be bijective, e.g. in `a += 5 · 6`, the expression `5 · 6` is not the "altering" function, and inverse of the whole statement would simply be subtracting `5 · 6` from `a`.

The only thing one needs to remember is, that a variable name cannot occur at both side if the statement is to be reversible. Having this constraint ensures forward and backward determinism [FIND CITATION].

2.5 If Statements

The `if`-statements are similar to those of `Janus`:

```
if (<expr1>) {
 <stmts>
} fi <expr2>
else {
 <stmts>
}
```

$\xleftrightarrow{\text{Inverse of}}$

```
if (<expr2>) {
<stmts>^{-1}
} fi <expr1>
else {
 <stmts>^{-1}
}
```

Where the `else` part can be omitted.

2.6 Loops

`JAPA` will support two different kind of loops: a `for`- and a `from`-loop. The `from`-loop is identical to the one from `Janus`: The `for`-loop is closer to `C++` syntax:

```
for <id> = <expr1> {
 <stmts>
} <incr>; until (<id> == <expr2>)
```

$\xleftrightarrow{\text{Inverse of}}$

```
for <id> = <expr2>; <inc>^{-1} {
 <stmts>^{-1}
} until (<id> == <expr1>)
```

Where `inc` is a modification operation typically on <id>, a^{-1} indicated that a is reversed, and having `<inc>` above the body executes it before the body and if it is after the body, it is executed after.

Moving the `<inc>` operation when reversing the loop is a necessity to make sure that `<id>` always have to correct value when going into an iteration. If the operation was not moved, the loop would be unaligned with its reversion. E.g. if the loop iterated over a list, it might stop when `<id>` is equal to the length of the list. If we reversed without moving the `<inc>` operation, the first iteration of the loop would access the index of the list length, which would be out of bounds.

2.7 Procedures

Procedures in `JAPA` does not have a return value; every outcome is exposed through side effect, by letting arguments be pass by reference. Return values are omitted because it complicates preserving reversibility, as the program needs a way to know where to enter and exit the procedure. This can be done by labeling all exits and possible enter areas with a condition, so it would require the programmer to know the exact state of the program at all possible exit areas. I have stayed at keeping procedure free of return values, for simplicity.

In regard to arguments, they can either be a constant value or a previous declared variable. This extinction needs to be taken, to ensure reversibility, as a procedure has no way of knowing, whether a given argument is a constant or variable, unless stated in the procedure definition. If variable was passed by value without being constant, it would be the same as creating a local variable at the start of the procedure; and given that the procedure input can change for every call, keeping track of the deallocation condition becomes complex, and I therefore force variables to be either pass by reference or a constant. Hence defining a procedure can be done as:

```
    procedure <id>(<args>) {
        <stmts>
    }
```

Where `<args>` is a comma separated list of arguments written `<type> <id>` or `<type> const <id>`, where `const` indicates that the given argument is passed by value, and the value cannot be altered in the procedure.

The reversibility of a procedure is straightforward now that all underlying statements are reversible. It is simply done by:

$$\texttt{call <id>(<args>)} \quad \xleftrightarrow{\text{Inverse of}} \quad \texttt{uncall <id>(<args>)}$$

Where `uncall` simply reverse the underlying statements.

2.8 Formal Definition of Syntax

The language `JAPA` can be formally written in Bacus-Naur form as:

```
<program> := <v-decls> <p-decls>
<v-decls> := <v-decl> <v-decls> | ""
<p-decls> := <p-decl> <p-decls> | ""

<v-decl>  := <type> <id> | <type> <id> "[" <const> "]"
<p-decl>  := "procedure" <id> "(" <f-args> ")" "{" <stmts> "}"

<f-args>  := <f-args'> | ""
<f-args'> := <type> <id> "," <f-args'> | <type> <id> "[" "]" "," <f-args'>
           | <type> <id> | <type> <id> "[" "]"
<a-args>  := <a-args'> | ""
<a-args'> := <id> "," <a-args'> | <constant> "," <a-args'> | <id> "[" <expr> "]" "," <a-args'>
           | <id> | <id> "[" <expr> "]" | <constant>

<type>    := "int" | "bool"

<stmts>   := <stmt> <stmts> | ""
<stmt>    := "local" <type> <id> "=" <expr>
           | "dealloc" <type> <id> "=" <expr>
           | <id> <mod-op> "=" <expr>
           | <id> "[" <expr> "]" <mod-op> "=" <expr>
           | <id> "<=>" <id>
           | "if" "(" <expr> ")" "{" <stmts> "}" "fi" "(" <expr> ")" "else" "{" <stmts> "}"
           | "if" "(" <expr> ")" "{" <stmts> "}" "fi"
```

```
            | "for" <id> "=" <expr> "{" <stmts> "}"
              <id> <mod-op> "=" <expr> "," "untill" "(" <id> "=" <expr> ")"
            | "for" <id> "=" <expr> "," <id> <mod-op> "=" <expr> "{" <stmts> "}"
              "untill" "(" <id> "=" <expr> ")"
            | "call" <id> "(" <a-args> ")"
            | "uncall" <id> "(" <a-args> ")"

<expr>      := <constant> | <id> | <id> "[" <expr> "]" | <expr> <bin-op> <expr>
            | "length" "(" <id> ")"

<bin-op>    := "+" | "-" | "^" | "*" | "/" | "%" | "&" | "&&" | "|" | "||" | ">"
            | ">=" | "<" | "<=" | "!=" | "=="

<mod-op>    := "+" | "-" | "^"
```

3 Theorem Prover

3.1 Theorem Prover Versus Other Alternatives

3.2 Choosing Theorem Prover

4 Compiler Structure

4.1 Lexer

4.2 Parser

4.3 Optimization

Because the compiler translates directly from the source language into an abstract syntax tree, and then into `C++`, the optimization must be done on the abstract syntax tree, as it is the easiest data structure to work on of the three. Doing the optimization directly on the source language itself, saves the computation of translating to some intermediate language, but does make this optimizer local to the source language.

This optimization involves four steps:

1. Discovering language constructs, that introduce assertions on translation.
2. Locating and gathering information for the prove.
3. Translating subpart of abstract syntax tree into `z3`.
4. Deciding on whether to optimize the assertion or not, based on the answer from `z3`.

4.3.1 Language Constructs Introducing Asseritons

The first point can be discovered by consulting the language specification introduced in section 2. Here it is imminent, that the following construct introduce assertions for the translated code:

- Local declaration as they require deallocation at some point for reversibility.
- If statement as the program needs to know whether the if-path was chosen or not when reversing computation.
- from statement as the first boolean expression following `from` can only be true before the loop runs, and the expression after `until` can only be true after all iterations are complete.

Which means that when running through the abstract syntax tree, the compiler must pause translation when the above constructs are found, and then the optimizer module must be called to tell the translator, whether to include assertions or not.

4.3.2 Gathering Information for Z3

Point two has two main obstacles: 1) to know how much information `z3` needs for the proof, and 2) granting the optimizer access to this appropriate subtree.

For 1) a simply approach could be to give the tree representing the current procedure to the optimizer. This could give too much information in the sense that only part of this subtree is actually needed to perform the validation check. e.g. in the small program below the only thing needed for the validation is the two last lines. However, this would require backtracking the tree while keeping a list of ”unassigned” variables, until this list is empty, so the first method is used in this optimizer. For future work, it would be a good idea to check whether the other approach would be faster. The below piece of code also reveal two other problem in regards to information: Global variables and function calls.

Getting the value of a global variable requires that the language be interpreted on the go, making the question a matter of a table lookup; there is however no interpretation going on in this compiler, limiting the optimizer into using unspecified variables for globals.

Function calls hide information that might be needed when performing validation for a procedure. To mitigate this, aggressive inlining is used. This does however pose the question of recursive procedures. I will leave this question for now, and return to it below, when addressing loops.

```
int c

procedure g()
{
    c += 5
}

procedure f(int a)
{
    call g()
    a += c
    local b = 1
    delocal b == 1
}
```

4.3.3 Translating to Z3

As the compiler is implemented in `Haskell` these bindings for `z3` in `Haskell` is used [3]. The optimizer module is therefore a translator from the abstract syntax tree into these bindings, and can be seen in the Appendices.

Validating whether an assertion can be removed with `z3`, can be done by creating a model of the code, that ends up inverting the boolean expression being asserted, and then checking for satisfiability. If this is not satisfiable, then the assertion can never be false, and can safely be removed.

Because of constraints in `z3` the following language constructs of `JAPA` is problematic:

- Moderating statements.
- Conditional statements.
- Loops.
- Function calls.
- Reversibility.

For solving the difficulties of these language constructs, I have used the approach outlined in the talk [4] at Compose 2016.

Moderating Statements

The problem with this construct is that all variables in `z3` are immutable. This can be addressed by creating new fresh variables, and asserting these fresh variables to the moderated value e.g.

```
x += 5  =>  (declare-const x1 Int)
            (assert (= x1 (+ x 5)))
```

Conditional statements

If the assertion being validated is not associated to a specific conditional, the program will not know which path is going to be taken at run time. Therefore both paths must be constructed, and then the `ite` function from `z3` can be used to determine the path. e.g.

```
local int x = 5         (define-fun x () Int 5)
if (x < 5)              (declare-const x1 Int)
{                   =>  (assert (= x1 (- x 5)))
    x -= 5              (declare-const x2 Int)
} fi (x == 0)           (assert (= x2 (ite (< x 5) x x1)))
delocal int x == 5      (assert not (= x2 5))
```

Loops

Because `z3` does not have a construction for loops, loops can only be validated by unrolling them. The question of how far to unroll loops is however rather complex. e.g. what should be done with non-terminating loops, and how to detect these? Also how far should loops be unrolled? One important factor is, that the optimizer must be conservative, so to not change the runtime behavior. Which means if it's not possible to analyze how many times to unroll, the optimization is blocked by loops.

When it comes to `for`-loops the analysis is relatively simple, as the constructs both tell a starting value, what happens to this variable at each iteration, and when to stop. So as long as the local variable being declared in the beginning, is the same as the one being deallocated after the loop, and the same being moderated at each iteration, it is straight forward. However if one of these are not the case, the analysis becomes harder.

The unrolling of a simple `for`-loop can be done by transforming it into a series of `if`-statements that can be represented in `z3`:

```
local Int bit = 1                   local int bit = 1
for int i = 0                       local int i = 0
{                                   if (!(i == 10))
    local int z = bit           =>  {
    bit += z                            local int z = bit
    delocal int z = bit / 2             bit += z
} i += 1; untill (int i = 10)           delocal int z = bit / 2
                                        i += 1
                                        if (!(i == 10))
                                        {
                                            ...
                                        } fi (i == 10)
                                    } fi (i == 10)
                                    delocal int i = 10
                                    delocal int bit = 2048
```

Function Calls

To model the functions correctly in `z3` inlining must be done, as all changes done by a function is through side effects. After inlining the function, it can be modeled using the other strategies outlined above.

Reversibility

Because the translation is from a reversible language to a irreversible language, every procedure will be translated into two: One going forward, and one backwards. It is necessary to perform the validation check individually for these two outcome procedures, as one way being valid, does not imply the other is. E.g. in the following code, the assertion can be removed in the forward run, but not the backwards, as giving an uneven number to `double` backwards will result in information loss due to integer division.

```
    procedure double(int bit)
    {
        local int z = bit
        bit += z
        delocal int z = bit / 2
    }
```

4.4 Translation to C++

5 Testing of Compiler

6 Benchmark of Compiled Code

7 Packing Compiler

7.1 Compiler Package and Usage

7.2 Web Interface for Compiler

8 Reflection Over Project

9 Conclusion

10 References

[1] L. Muehlhauser, “Mike frank on reversible computing,” *Machine Intelligence Research institite*, 2014.

[2] E. Rose, “Arrow: A modern reversible programming language,” 2015.

[3] I. Abal and D. Castro, “z3: Bindings for the z3 theorem prover.” `https://hackage.haskell.org/package/z3`. Accessed: 18/02-2022.

[4] T. Jelvis, “Analyzing programs with z3,” 2016.

11 Appendices