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>

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.

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 Abstract

3 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 `JL` (short for Janus-like) 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 [**?**], and require two things: Local reversibility and global reversibility.

Local reversibility

Definition 3.1. For a local process to be reversible information from step to step must be preserved, meaning no information is destroyed [**?**].

Global reversibility

Definition 3.2. A global process can only be reversible if the mapping from a start state to a final state is bijective [**?**].

This means that every construct of `JL` 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 3.1. 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.

3.1 Data types

3.2 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:

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

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:

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

3.3 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 `JL` 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:

$$\begin{array}{l} \texttt{<id> += <expr1>} \\ \texttt{<id> \^{}= <expr2>} \end{array} \xleftrightarrow{\text{Inverse of}} \begin{array}{l} \texttt{<id> -= <expr1} \\ \texttt{<id> \^{}= <expr2>} \end{array}$$

Where ^ indicates exclusive or.

3.4 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].

3.5 Loops

`JL` will support two different kind of loops: a `for`- and a `from`-loop. The `from`-loop is identical to the one from `Janus`:

```
from <expr1> {      Inverse of    from <expr2> {
 <stmts>            <-------->     <stmts>^{-1}
} until (<expr2>)                 } until (<expr1>)
```

The `for`-loop is closer to `C++` syntax:

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

3.6 If statements

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

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

Where the `else` part can be omitted.

3.7 Procedures

Now where all building blocks for procedures are reversible, it is straight forward to reverse a procedure itself:

4 Theorem prover

5 Compiler structure

5.1 Lexer

5.2 Parser

5.3 Optimization

5.4 Translation to `C++`

6 Testing of compiler

7 Benchmark of compiled code

8 Packing compiler

8.1 Compiler package and usage

8.2 Web interface for compiler

9 Reflection over project

10 Conclusion