UNIVERSITY OF CALGARY

Floey, an Intermediate Language for Optimizing Compilers

by

Ning Tang

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

July, 2008

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Floey, an Intermediate Language for Optimizing Compilers” submitted by Ning Tang in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. J. Robin Cockett
Department of Computer Science

Dr. Rob Simmonds
Department of Computer Science

Dr. Laurence E. Turner
Department of Electrical & Computer Engineering

Date

Abstract

In modern optimizing compilers, linear human-readable text representation of a program is first transformed into an abstract syntax tree that represents the structure of that program. Abstract syntax tree is then transformed into **intermediate representation** (IR), based on which compiler optimizations are accomplished. The optimized IR is sent to the code generator and finally translated into assembly or machine code. Research on IRs has been focused on how they can be designed to facilitate compiler optimizations or more effective code generation on specific architecture.

This thesis presents a mid-level intermediate language, called Floey. In a Floey program, control flowgraphs are separated into different tree-like structures called control expressions. Different control expressions are connected by entries.

On Floey, a machine independent optimization, called the reduction algorithm, is implemented. By comparing the reduction algorithm to various conventional optimizations, we argue that not only Floey facilities compiler optimization design, it also provides a cleaner and uniform perspective on compiler optimizations in general.

Acknowledgements

It is a great pleasure to thank the many people who made this thesis possible.

I wish to thank my supervisor, Dr. Robin Cockett. He is not only a wonderful mentor, working with me day and night all these years, but also a good friend, from whom I have so much to learn beyond the contents of this thesis.

Thank you to my proof-readers, Xiuzhan Guo and Brian Redmond.

Thank you to Pieter Hofstra for helping me on Category Theory.

Thank you to Brett Giles and Sean Nichols for helping me on LaTeX and Haskell.

I would also like to thank all my friends in Calgary for being there with me through out the good times and the difficult ones.

My special thanks go Ying Wang, my Baobao, for being the motivation and inspiration of my life.

Lastly, and most importantly, I wish to thank my parents, Riming Tang and Meixiu Liu, for their selfless support and the sacrifices they have made since the day I was born.

Dedicated to my wonderful parents, Riming Tang and Meixiu Liu, and to my lovely girlfriend Ying Wang.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis introduces a language, called Floey, which is an intermediate representation designed to facilitate the implementation of compiler optimizations. The Floey language is essentially a flow diagram language for reducible flow graphs. However, it is organized around extended blocks and, thus, Floey expressions are trees, on which optimization techniques are generally simpler to implement. An optimization technique for Floey programs, the reduction algorithm, is presented. The implementation of the reduction algorithm on Floey programs provides a different and more uniform perspective on compiler optimizations which we hope may be of benefit to the design of optimizing compilers in general.

Section 1.1 gives a general introduction to the structure of optimizing compilers and the role of intermediate representations in compiler designs. Section 1.2 introduces the concept of control flow graph. The limitations of the control flow graph based approaches and the motivations of our work are discussed in section 1.3. The general organizations of the thesis is present in section 1.4.

1.1 Intermediate representation in optimizing compilers

In modern optimizing compilers [Aho et al., 2007; Appel, 1998; Muchnick, 1997], linear human-readable text representation of a program is first transformed into an abstract syntax tree that represents the structure of that program. The syntactic and semantic checking are both done in this phase. These steps are generally categorized as the “front end” of the

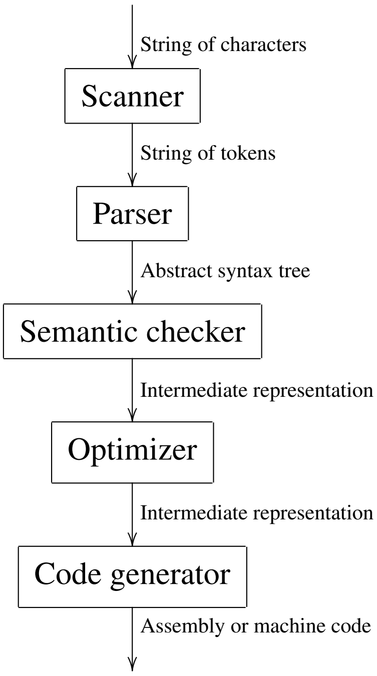


Figure 1.1: Structure of modern optimizing compilers

compiling process. In the output of the "front end", the abstract syntax tree is transformed into **intermediate representation** (IR), based on which compiler optimizations are accomplished. The optimized IR is then sent to the code generator, regarded as the back end of compilers, and is finally translated into assembly or machine code. A general structure of optimizing compilers is given in figure 1.1 (more detailed discussions in [Muchnick, 1997]).

From figure 1.1, we can see that IRs are the data structures that optimizers work on. There is an obvious advantage of this design: IRs ease the task of implementing compilers with multiple front ends and back ends, as all the front ends have a uniform target, while all the back ends have a uniform source. No matter how many front ends and back ends a compiler has, each optimization is only implemented once on a specific IR. Moreover, a compiler can use different IRs for different optimizations it performs. For example in the compilers of GNU Compiler Collection (GCC), there are three different IRs: Register Transfer Language (RTL), GENERIC and GIMPLE [Novillo, 2003, 2004], each of these are useful for the implementation of specific compiler optimizations. All the transformations between different IRs are done in the optimizer, which can be seen as a black box from other parts of compiler.

Intermediate representations are often vaguely categorized into three abstract levels: high-level, mid-level and low-level. The higher the level is, the closer the IR is to the source program. The lower the level is, the more architecture dependent and more closely it resembles machine code (or assembly). High-level and mid-level IRs often facilitates machine independent optimizations, for example common sub-expression elimination [Cocke, J., 1970; Ullman, 1972] and dead code elimination [Aho et al., 2007]. While on low-level IRs, instruction level optimizations are easier to implement, for example the instruction selection [Aho and Johnson, 1976] and register allocation [Briggs, P., 1992].

Research on IRs has been focused on how they can be designed to facilitate compiler optimizations or more effective code generation on specific architecture [Click and Paleczny, 1995; Wells, 2004]. In this thesis, we introduce a mid-level intermediate language, called Floey, and present a machine independent optimization on Floey, called the reduction algorithm. We wish to show how Floey can be used to assist in the implementation of compiler optimizations, and how it may provide a cleaner organization for compiler optimization designs in general.

1.2 Control flow graphs

Most compiler optimizations rely on control flow information, which is often expressed as Control Flow Graph (CFG). A **control flow graph** is a directed flow graph with program statements as nodes and program execution flows as arrows. Two artificial nodes, called “begin” and “end”, are often added into a CFG indicating the begin and the end of the program. Figure 1.2 shows a typical CFG of a computer program. Every node other than the “begin” and the “end” represents a statement in the program. The statements are linked by directed arrows as execution flow. For instance, after executing S_1, S_2 will be executed (assuming S_1 terminates). After S_3, either S_4 or S_6 will be executed (again assuming S_3 terminates).

There is a special kind of CFGs, called reducible CFG, that attract the most attention in programming language research, as these CFGs have various properties that facilitate optimization [Hecht and Ullman, 1974]. There are several equivalent definitions of reducible CFG. Here we will use the definition based on collapsing (see [Aho et al., 2007; Hecht, 1977] for other definitions). A control flow graph is **reducible** if and only if repeated application of the following two actions yields a CFG with only one node [Hecht, 1977], otherwise it is called an irreducible or nonreducible CFG.

- T$_1$: Let G be a flow graph and let (w,w) be an arrow of G. T$_1$ is the removal of this arrow.

- T$_2$: Let G be a flow graph, and let y not be the initial node and have a single predecessor, x. T$_2$ is the replacement of x, y and (x,y) by a single node z. Predecessors of x become predecessors of z. Successors of x or y become successors of z.

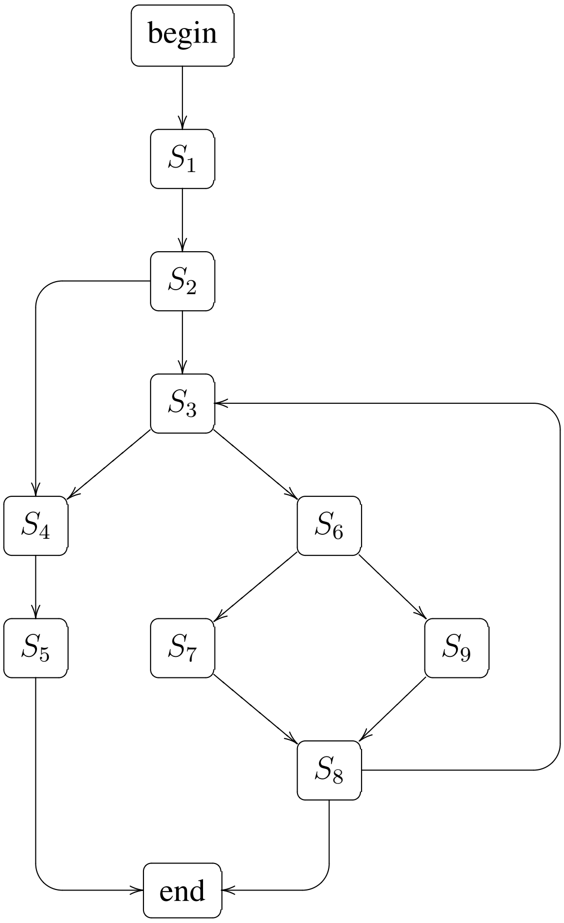


Figure 1.2: Control Flow Graph

The CFG in figure 1.2 is reducible, while the ones in table 1.1 are irreducible. Although all irreducible CFGs can be converted into reducible one, the transformation may cause possible code size explosion (see [Hecht, 1977]). “Structured” programming languages allow only programs with reducible CFGs. Common control-flow constructs, such as **if-then-else**, **while do**, and **repeat-until** generate reducible CFGs; it is the arbitrary **goto** that may generate irreducible CFGs. All CFGs in Floey are reducible.

There are three important structures in CFG that concern the design of Floey: basic

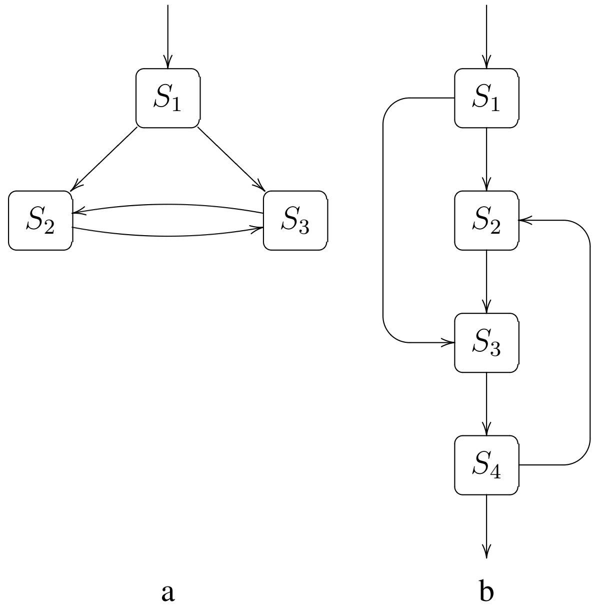


Table 1.1: Irreducible control flow graphs

blocks, extended basic blocks, and loops.

- A **basic block** is a sequence of consecutive statements in which execution flow enters at the beginning and leaves at the end without possibility of branching except at the end [Aho et al., 2007]. In other words, once the first statement in a basic block is executed, all other statements in that basic block will always be executed in order. The sequences $\{S_1, S_2\}$,$\{S_3\}$,$\{S_4, S_5\}$,$\{S_6\}$,$\{S_7\}$,$\{S_8\}$ and $\{S_9\}$ in figure 1.2 are basic blocks.

- An **extended basic block** is a connected collection of nodes in CFG such that all nodes except for the first one have only one predecessor, i.e. a tree structure of statements. The first node either has multiple predecessors or has the “begin” node as its predecessor. $\{S_1, S_2\}$, $\{S_3, S_6, S_7, S_9\}$, $\{S_4, S_5\}$ and $\{S_8\}$ are the extended basic blocks in 1.2.

- A **loop** is a strongly connected component in a CFG. A **strongly connected component** in a directed graph is a set of nodes such that every node can reach all other nodes in the set. In reducible CFGs, any loop has a loop header that dominates all other nodes in the loop. A node x is said to **dominate** node y only if every path from the "begin" to y includes x. $\{S_3, S_6, S_7, S_8, S_9\}$ is the only loop in 1.2, with S_3 as the loop header.

1.3 Limitations and motivations

Although Control Flow Graphs are widely used in the design of compiler optimization algorithms, we believe there are some limitations to this approach, which not only affect the efficiency of the optimization algorithms but make the implementation difficult as well.

- Many aggressive compiler optimizations heavily modify the structure of CFG. As shown in table 1.2, the loop invariant code motion [Neel and Amirchahy, 1975] creates a loop pre-header (S_5) above the original loop. This code manipulation modifies the basic block structure, as the basic block of S_1 should now include S_5. Changes like this in the CFG require frequent reanalysis of the structure, which is costly and difficult to manage.

- Most local level optimizations are based on the basic blocks, as the simple structure makes the design and implementation easier. However, we believe that the extended basic blocks deserve more attention in compiler optimization designs. Since it is often the joint points, the nodes with more than one predecessors, in CFGs that

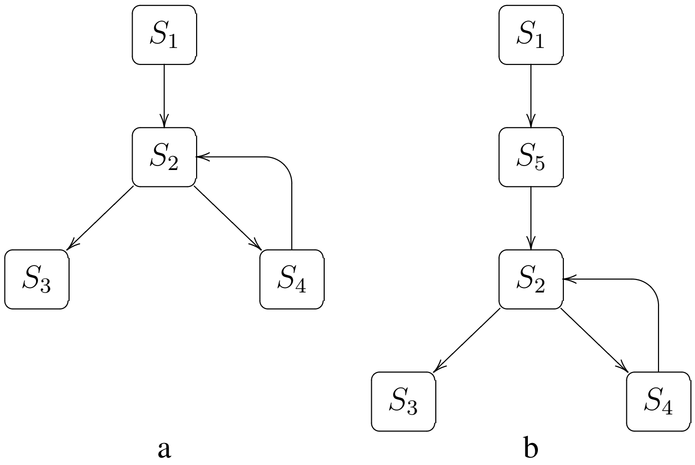


Table 1.2: Loop invariant code motion modifies basic block structures

introduce subtleties and not the branching structure (of extended basic blocks). Most basic block optimizations can be easily extended to extended basic blocks.

To overcome these limitations of CFSs, a new intermediate representation, called Floey (chapter 2), is introduced in this thesis. Floey programs are divided into basic structures called control expressions. Each of these control expressions represents an extended block (possibly with loops). Different control expressions in a CFG are linked together by the entries. Floey allows compiler optimizations to be implemented on control expression, which is a significantly larger unit than a basic block. On the global level, entries provide uniformed interfaces between control expressions which makes global optimizations easier to design and to implement (although this beyond the scope of this thesis).

An optimization technique for Floey programs, called the reduction algorithm (chapter 5), is presented in this thesis. In Floey, all programs are first converted in control normal form (chapter 4), which is a canonical form for Floey program. The normalized programs are then optimized by the reduction algorithm. The algorithm first eliminates all repeated computations in control expression. It then tries to delay every statement in a control expression in order to remove unnecessary ones. In the last step, invariants in loops are eliminated. We believe the reduction algorithm has several advantages over conventional optimization techniques as it gives a uniform structure to the optimizing process. From the design and implementation of the reduction algorithm, we also argue that the structure of Floey facilitates these optimizations (chapter 6).

1.4 Organization

The thesis is organized as follows. Chapter 2 provides a detailed introduction to the Floey intermediate language. The type system of Floey is introduced in chapter 3. Chapter 4 discusses the control normal form of Floey programs. An optimization technique implemented in Floey, called the reduction algorithm, is introduced in chapter 5. Chapter 6 compares the optimizations on Floey to the conventional approaches and concludes the thesis.

Chapter 2

Introduction to Floey

This chapter presents an overview of Floey language. The structure, constructs and syntax of Floey programs are illustrated with examples. The chapter is organized as follows. Section 2.1 gives an overview of the Floey language, while each of the rest sections introduces one construct in Floey.

2.1 Floey overview

Floey is a typed intermediate representation language designed to facilitate compiler optimizations. Floey programs are well-structured flow diagrams which have been broken down into extended basic blocks and loops. This allows standard local compiler optimizations on basic blocks, extended basic blocks and loops to be applied more easily.

```
Floey : main : mainFun
        FloeyDefs
FloeyDefs : FloeyData FloeyDefs
          | FloeyFun FloeyDefs
          | FloeyControl FloeyDefs
```

Figure 2.1: Floey program structure

Figure 2.1 gives the structure of a Floey program, which begins with a main function specification followed by a sequence of FloeyDefs. The function specified by mainFun is the starting point of the program. Each of the FloeyDefs can be either a data declaration, FloeyData, a function definition, FloeyFun, or a control operation definition, FloeyControl.

Each function or control operation definition can also contain local FloeyDefs together with a well-structured flow diagram, expressed as a control expression with a series of entries. Control expressions are the basic units of Floey programs, each of which is essentially a tree structure with loops.

The program shown in figure 2.2 demonstrates Floey program structure. At the top level, there is one data declaration and one function definition, calGrade, which is the main function of the program. The function takes in the mid-term, final-term and bonus question score of a student and calculates his or her final grade. In calGrade, there are two local function and control operation definitions. getFinal calculates the final score based on the mid-term and final-term score. The result of getFinal is assigned to variable final and is used in the control expression from line 29 to line 33. If a student correctly answered the bonus question, he or she has ten bonus score added to the final score. That score is passed to entry printGrade as a actual parameter. Based on the output of getGrade, a letter grade will be returned if a student has passed. Otherwise, the function returns an integer, which stands for the score in short for passing the course.

The following sections will explain in detail the different Floey constructs.

2.2 Data declarations

Floey has various built-in data types, such as: the boolean type, Bool, the integer type, Int, the real number type, Real, and the character type, Char. These types can be used in Floey programs without declaration. Moreover, programmers can declare new data types as global or local types.

A data declaration consists of the specification of a type constructor and its term con-

```
main: calGrade

--data declarations
data PF = Passed (Char)
        | Failed (Int)

fun calGrade(x,y:Real,bonus:Bool):PF
{
    --a control operation definition
    def getGrade(score:Real)[a,b,c,d(Int)]
    {
        begin
        case score>=85 of
        [True.exit a
        |False.
            case score>=70 of
            [True.exit b
            |False.
                case score>=60 of
                [True.exit c
                |False.exit d(60-score)]]]
    }
    --a function definition
    fun getFinal(a,b:Real):Real
    {
        begin return (a*0.4+b*0.6)/2
    }
    --a control expression
    begin
    final <- getFinal(x,y).
    case bonus of
    [True.
        final <- final + 10.
        exit printGrade(final)
    |False.exit printGrade(final)]
    --an entry
    entry printGrade(x:Real)
    {
        begin
        getGrade(x) of
        [a.return Passed('A')
        |b.return Passed('B')
        |c.return Passed('C')
        |d(diff).return Failed(diff)]
    }
}
```

Figure 2.2: A typical Floey program

structors. Each type constructor and term constructor starts with an upper case letter. The syntax for data declaration, FloeyData, is as follows. Here T is the type constructor, which

```
FloeyData :
data T $a_1$... $a_{n_1}$ = $Cons_1$($T_{11}$,...,$T_{1m_1}$)
                      ...
                      | $Cons_{n_2}$($T_{n_21}$,...,$T_{n_2m_{n_2}}$)
```

Figure 2.3: Floey data declaration

is the identifier of that type. Cons_1 to Cons_{n_2} are the term constructors. a_1 to a_{n_1} are the type arguments for T, each of which starts with lower case letters indicating a polymorphic type. Each term constructor Cons_i can also have term arguments: T_i1 to $\text{T}_i m_i$, each of which can either be a built-in type, a user defined type of a polymorphic type from the type arguments a_1 to a_{n_1}. For example: In figure 2.4, List is the new type constructor being

```
data List a = Cons(a, (List a))
            | Nil
```

Figure 2.4: List of polymorphic type “a”

declared. a is a polymorphic type argument. Cons and Nil are the term constructors with their argument type specified. Nil is a unit type with no argument, while Cons has two arguments, one of which is a recursive use of type List a. When List a is used in Floey programs, the type a can be specified. For example: List Int, List (List a).

The data type PF in figure 2.2 (line 3 to line 5) gives an example of data declaration. Some more examples of data declarations:

```
--Pre-defined boolean type
data Bool = True
          | False
```

```
--List of boolean
data BList = BCons (Bool, BList)
           | BNil
--Tree of list of boolean
data BListTree = Branch (BListTree, BListTree)
               | Leaf (BList)
```

2.3 Functions

A function in Floey takes in a list of arguments and returns a value of certain type. Each function can have local data declarations, which are only visible inside the function. One can also define local functions or control operations inside a function. At the end of each function, there must be a control expression with a series of entries, which constitute the control flow graph of that function.

A function definition begins with the keyword **fun** followed by an identifier, a list of arguments in braces and a return type separated by colons. The identifier must start with a lower case letter. An empty parameter list is permitted: and braces can be omitted in that case. The structure of a function definition, FloeyFun, is as follows. where id is the

```
FloeyFun :
fun id(a_1:T_1,...,a_{n_1}:T_{n_1}):T
{
    FloeyDefs
    begin
    E
    entry e_1(...){...}
    ...
    entry e_{n_6}(...){...}
}
```

Figure 2.5: Floey function definition

identifier of the function. Each a_i is an argument of type T_i. The function returns a value of type T.

In Floey, basic operations on integer, real numbers and boolean values are built-in. These operations are also functions and can be used without definition.

Both calGrade and getFinal in figure 2.2 are examples of function definition. Another example is as follows.

```
1 fun gcd(a,b:Int):Int
2 {
3     begin
4     case b==0 ->
5     [ True.return a
6     | False.return (gcd(b,a mod b))]
7 }
```

Figure 2.6: A recursive function to calculate the GCD

As a function returns a single type, function calls can be used as (sub-)expressions. For example final <−getFinal(x,y) (line 28 in figure 2.2) is an assignment from expression of a function call to a variable. Line 4 in figure 2.6 uses b==0, an equal operation, as the expression in a case statement.

2.4 Control operations

In Floey, a control operation definition has a very similar structure to that of a function definition. The difference is that instead of returning a result of a single type, a control operation may return one of the branch specifications that are specified in its definition.

A **branch specification** is a branch name together with a list of arguments of certain types. Branch names start with lower case letters, for instance d(Int) in figure 2.2. Branch specifications are used as labels to which the program execution can jump. In a control operation definition, an exit statement (see section 2.6.1) can return to any one of the branch specifications of that control operation thereby completing its execution. In control statements (see section 2.6.3), where control operations are used, branch specifications are bound to the control expressions which must be executed should the branch be returned from the control operation.

The structure of a control operation definition, FloeyControl, is as follows.

```
FloeyControl :
def id($a_1$:$T_1$,...,$a_{n_1}$:$T_{n_1}$)[$s_1$,...,$s_{n_2}$]
{
    FloeyDefs
    begin
    $E$
    entry $e_1(\ldots)\{\ldots\}$
    ...
    entry $e_{n_7}(\ldots)\{\ldots\}$
}
```

Figure 2.7: Floey control operation definition

Here **def** is the keyword for control operation definition followed by id, an identifier which must start with a lower case letter. Each a_i is an argument of type T_i and s_1 to s_n is the list of branch specifications this control operation may return. The body of the control operation, enclosed between a pair of curly brackets, is similar to that of a function. getGrade (line 10 to line 22 in figure 2.2) is an example of control operation definition.

Note that since a control operation can return a variant type, the result of a control operation cannot be assigned to variables and therefore cannot be used as functions. Control operations can be used in control statements (see section 2.6.3), for example on line 40 to 44 in figure 2.2.

2.5 Entries

Entries are program fragments that are used to link control expressions. In Floey, control flow can jump from one control expression to another either by calling a control operation or a function, or by exiting to an entry using exit statements (see section 2.6.1). Entries are the general interface for the non-cyclic joint points of the control flow graphs.

The declaration of an entry starts with the keyword **entry** followed by an identifier, which starts with a lower case letter, and binding list with expected types. The syntax for entries is as follows.

```
entry en_id($a_1$:$T_1$,...,$a_{n_1}$:$T_{n_1}$)
{
    $E$
    entry $en_1$(...){...}
    ...
    entry $en_{n_2}$(...){...}
}
```

Here en_id is the identifier of the entry. Each a_i is a binding of type T_i. Variable values are passed through entries via bindings. E is the control expression contained in the entry. en_1 to en_{n_2} are the subsidiary entries. printGrade (line 37 to line 45 in figure 2.2) is an example of entry. Following is another example.

Both functions in table 2.1 return the larger integer of the two inputs. The program in 2.1.b uses an entry to capture the joint point in that execution flow and potentially avoids code duplication. It is a simple example but does demonstrate how entries can be used as the joint points for the control flow graph.

Note that cyclic entry calls are not allowed in Floey. Instead, such structures must be

```
fun max(x,y:Int):Int
{
  begin
  case x<y ->
  [True.return y
  |False.return x
  ]
}
```

a

```
fun max(x,y:Int):Int
{
  begin
  case x<y ->
  [True.exit foo(y)
  |False.exit foo(x)
  ]
  entry foo(a:Int)
  {return a}
}
```

b

Table 2.1: An example of Floey entries

```
entry gcd(a,b:Int)
{
    b==0 ->
    [True.return a
    |False.
        t <- b.
        b <- a mod b.
        a <- t.
        exit gcd(a,b)]
}
```

Figure 2.8: Illegal cyclic entry calls

expressed as loop expressions (see section 2.6.6). The example in figure 2.5 presents an illegal cyclic entry call. The function performs an iteration without using loop expression. As the statement **exit** gcd(a,b) cyclically jumps back to the header of the entry it is in.

Direct or indirect cyclic entry calls in Floey are prohibited and are picked up in the semantic checking and reported as errors. The reason for this design is to clearly distinguish loop headers from other joint points in Floey programs. In Floey, entries are designed to contain control expressions, which are extended basic blocks with loops (see section 2.6). Cyclic entries are looping structures, and so are included as loop expressions. This (together with entry normalization, see section 4.3) ensures entries are only used for the purpose for which they were designed.

2.6 Control expressions

Control expressions are the basic units of Floey program. Each of them describes the flow graph of a program fragment. A series of these control expressions, enclosed in different entries, control operations, or functions describe complete flow graphs of the program.

Line 13 to line 22, line 30 to 35 in figure 2.2 are both examples of control expression. The following example shows a control expression with a loop in it.

```
loop gcd_loop(a<-x,b<-y)
{
    case b==0 ->
    [True.return a
    |False.
        t <- b.
        b <- a mod b.
        a <- t.
        exit gcd_loop(a,b)]
}
```

Figure 2.9: A function to calculate the GCD using iteration

The rest of this section will explain in detail the six constructs of control expressions: exit statement (section 2.6.1), return statement (section 2.6.2), control statement (section 2.6.3), case statement (section 2.6.4), assignment statement (section 2.6.5) and loop expression (section 2.6.6).

2.6.1 Exit statements

Exit statements terminate the execution a control expression and redirect the program to an entry, a loop expression (see section 2.6.6) or a branch specification of the enclosing control operation definition. Exit statements are the leaves of the tree-like structure of control expressions. An exit statement alone is a control expression. The syntax of an exit statement is:

```
exit id(e1,...,en)
```

An exit statement starts with the keyword **exit**, followed by the identifier id of the entry, loop expression or branch specification and a list of expressions as parameters, (e_1,...,e_n), enclosed by a pair of braces (braces can be omitted when the expression list is empty). Type checking ensures these expressions match the expected argument types.

In Floey, an **expression** is either:

- a variable: x, y
- a constant: 12, True, 'A'
- a function, applied to expressions: gcd(x,y), getFinal(a,b)+10, Cons(10,Nil)

An example of an exit statement in figure 2.2 is **exit** printGrade(final). Here printGrade is the entry identifier and final is the expression passed to the entry. In the entry interface, **entry** printGrade(x:Real), the parameter x is expected to be a real number. If final matches the type Real, x will be initialized to that value and can be used in the control expression in printGrade.

exit gcd_loop(a,b) (in figure 2.9) is another example of an exit statement. Here gcd_loop is a loop identifier, while (a,b) is the list of expressions passed to the loop. Execution will be redirected to the loop header **loop** gcd_loop(a<−x,b<−y).

Note that if there are entries or loops with the same identifier in different scopes, the entry or loop in the closest scope to the exit statement will be called. For example in the following example, the exit statement on line 7 exits to the entry on line 8 and 9. True will be returned as the result.

```
1  fun foo():Bool
2  {
3      begin
4      exit a
5      entry a
6      {
7          exit b
8          entry b
9          {return True}
10     }
11     entry b
12     {return False
13     }
14 }
```

2.6.2 Return statements

Return statements terminate the execution of control expressions and return result of the current function redirecting the program to where the function is called. Return statements are leaves of control expression structure within functions. A return statement alone is a control expression. The syntax of a return statement is as follows.

```
return e
```

Here e is the expression that is returned. **return** Passed('A') and **return** (a*0.4+b*0.6)/2 in figure 2.2 are both examples of return statements.

2.6.3 Control statements

In Floey, a control statement is a branching statement which controls the execution flow redirecting it to one of its branches. A control statement consists of a control operation call and a series of branches that bind subsidiary control expressions. Based on the result of the control operation called, which is a branch call, program control flow selects that branch with the induced branch bindings and continue with the subsidiary control expression. The syntax of a control statement is as follows.

$$
\begin{array}{l}
\mathrm{con\_op(e_1,\ldots,e_{n_1})}\ \mathbf{of}\\
[\mathrm{b_1(v_{11},\ldots,v_{1m_1})}.E_1\\
|\ldots\\
|\mathrm{b_{n_2}(v_{n_21},\ldots,v_{n_2m_{n_2}})}.E_{n_2}]
\end{array}
$$

Here con_op is the control operation applied to expressions $\mathrm{e_1,\ldots,e_{n_1}}$. Each b_i is a branch name, where variables $\mathrm{v}_{i1},\ldots,\mathrm{v}_{im_i}$ are bound in the control expression E_i by the binding operation, ".". Branches in the same control statement are enclosed by a pair of square brackets and separated by slashes, "|".

Consider the following example of control statement from figure 2.2. The control operation getGrade returns grades if this student passed. Otherwise, grade d will be returned together with, diff, the extra score required to pass the course. diff is then bound in its subsidiary control expression, **return** Fail (diff), where it can be used.

```
getGrade(x) of
[a.return Passed('A')
|b.return Passed('B')
|c.return Passed('C')
|d(diff).return Failed(diff)]
```

2.6.4 Case statements

Case statements are special control statements that have the “case” control operation. The “case” control operation takes in one expression as argument and returns a constructor of the type of that expression. Similarly to control statements, case statements are also the branching structures in Floey program. A case statement directs program execution flow to one of its branches, based on the returned constructor. The syntax for a case statement is as follows.

```
case exp of
[$\mathrm{Cons}_1(\mathrm{v}_{11},\ldots,\mathrm{v}_{1m_1}).E_1$
...
|$\mathrm{Cons}_n(\mathrm{v}_{n1},\ldots,\mathrm{v}_{nm_n}).E_n$]
```

Here **case** is the keyword for the “case” control operation. It is followed by an expression, exp, and a series of branches that bind subsidiary control expressions. Each Cons_i is a type constructor with its arguments v_{i1} to v_{im}. These variables will be bound in E_i, the control expressions follows, by the binding operations, written as “.”. Branches in the same control statement are enclosed by a pair of square brackets and separated by slashes, “|”.

In the following example, the function first returns the first element of a list of integers. The control expression in the function body is a case statement that has two branches. If x is Nil, an empty list, 0 is returned. Otherwise, the constructor argument i will be returned.

```
fun first(x:List Int):Int
{
    begin
    case x of
    [Cons(i,j).return i
    |Nil.return 0
    ]
}
```

2.6.5 Assignment statements

The syntax for assignment statements is as follows.

```
v <- exp.
E
```

Here v is a variable and exp is an expression. v is assigned to the value of exp, by the assignment sign, "<–", and is bound in the control expression E that follows, by the binding operator, ".".

Above is a standard definition of assignments. However, internally in Floey, assignments are viewed from a different perspective, as they are essentially special unary decisions with no branching and one branch variable. Thus in Floey, assignments are represented as case statements with a special "#Assign" constructor. The above assignment is translated into the following statement.

```
case exp of
[#Assign(v).E]
```

Here the branch variable, v, can be of any type that exp returns and is bound in E. This translation simplifies the internal data structure and more importantly makes the i implementation of optimizations in chapter 4 and 5 more uniform.

2.6.6 Loop expressions

Loop expressions are the only legitimate "looping" structure in Floey. They have similar structure to entries. The syntax for loop expressions is as follows.

```
loop id(y_1<-x_1,...,y_{n_1}<-x_{n_1})
{
    E
    entry en_1(...){...}
    ...
    entry en_{n_2}(...){...}
}
```

Here **loop** id(y_1<−x_1,...,y_{n_1}<−x_{n_1}) is the loop header, which starts with the keyword "**loop**", followed by the identifier of the loop id and a list of loop bindings. The loop identifiers are used in exit statements whenever control flow jumps back to the loop header. Enclosed in a pair of curly brackets is the loop body, a control expression E together with a series of subsidiary entries **entry**$_1$ to **entry**$_{n_2}$. The control expression in figure 2.9 is an example of loop expression.

Each loop binding is a variable paired up with an expression separated by the assignment symbol, such as x<−exp. Here x is a **bound variable** of the loop and exp is called an **initialization expression**. When the loop is executed for the first time, x is initially assigned to exp. Thus the initialization works just like an assignment on first iteration. When the execution flow is redirected back to the loop header by exit statements in that loop body, the bound variables are updated to the values of the arguments in exit statements.

The flow diagrams which correspond to legal Floey programs must be reducible. This means loop headers dominate their loop bodies. In other words, control flow can only enter a loop through the loop header. Any attempt to jump from outside of a loop to a subsidiary entry to that loop will cause semantic checking stage to report an error.

Chapter 3

Type system of Floey

A type theoretic description of Floey is provided in this chapter. The chapter starts with the introduction of the static type system for Floey in section 3.1. Examples of proofs in this type system are given to illustrate how the type system works on real Floey programs. Section 3.2 discusses the equalities which govern the manipulation of Floey control expressions.

3.1 Floey type system

This section presents a static type system for Floey, which guided the design of the Floey type checker. The system consists of a set of inference rules, which are also called judgements. Each inference rule has a premise and a conclusion. The following is a typical inference rule:

$$\frac{\Gamma \vdash t : T}{\Gamma \models \mathsf{return}\ t : \{\mathsf{return}(T)\}}\ \text{return}$$

Here $\Gamma \vdash t : T$ is the premise and $\Gamma \models \mathsf{return}\ t : \{\mathsf{return}(T)\}$ is the conclusion. Γ is the context that consists of typed variables. Note that we introduced two types of sequent: $\vdash$ is used in sequents for expressions, and $\models$ is used in sequents for control expressions. We also use $\Gamma \vdash t_1, \ldots, t_n$ as the short hand for $\Gamma \vdash t_1, \ldots, \Gamma \vdash t_n$.

The inference rules are categories into three groups: inference rules for expressions, inference rules for control expressions, and the ones for definitions of functions and control operations. The following subsections explain the inference rules in detail.

3.1.1 The inference rules for expressions

The rules for Floey expressions are given in table 3.1.

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad f : \mathcal{F}(T_1, \ldots, T_n \to T)}{\Gamma \vdash f(t_1, \ldots, t_n) : T}\ \text{expression}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma, y : T \vdash t' : T'}{\Gamma \vdash t'[t/y]}\ \text{cut}$$

Table 3.1: The inference rules for expressions

The basic rule for an expression is as follows.

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad f : \mathcal{F}(T_1, \ldots, T_n \to T)}{\Gamma \vdash f(t_1, \ldots, t_n) : T}\ \text{expression}$$

Here $\mathcal{F}$ is the declarations of all functions (see section 3.1.3) that are visible at the point where this rule is applied. If the terms, $t_1, \ldots, t_n$, match the expected types for arguments, $T_1, \ldots, T_n$, of function f. We can construct of new expression, that is the function f applied to terms $t_1, \ldots, t_n$, which returns type T.

In the Floey type checker, the arguments of a function are first type checked. All resulting types will be then used to match the types of the function arguments. If all these type checking steps succeed, the whole expression results in the return type of that function. Otherwise, type checking fails and type errors will be output.

Moreover, all expressions in Floey should satisfy the following cut rule.

$$\frac{\Gamma \vdash t : T \quad \Gamma, y : T \vdash t' : T'}{\Gamma \vdash t'[t/y]}\ \text{cut}$$

3.1.2 The inference rules for control expressions

The inference rules for constructing Floey control expressions are given in table 3.2. Each of the first five rules focuses on one Floey construct, while the last one includes entries into the type system

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad b(T_1, \ldots, T_n)}{\Gamma \models \mathsf{exit}\ b(t_1, \ldots, t_n) : \{b\}}\ \text{exit} \qquad \frac{\Gamma \vdash t : T}{\Gamma \models \mathsf{return}\ t : \{\mathsf{return}(T)\}}\ \text{return}$$

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad \Gamma, \tilde{x_i} : \Delta_i \models e_i : \mathcal{B}_i \quad g : \mathcal{G}(\tilde{T} \to \{b_i(\Delta_i)\}_{i \in I})}{\Gamma \models g(t_1, \ldots, t_n) \to [b_i(\tilde{x_i}).e_i]_{i \in I} : \bigcup \mathcal{B}_i}\ \text{control}$$

$$\frac{\Gamma \vdash t :: C\ a_1\ \ldots\ a_n \quad \Gamma, \tilde{x_i} : \Delta_i \models e_i : \mathcal{B}_i \quad \mathsf{data}\ C\ a_1\ \ldots\ a_n\ = \{C_i(\Delta_i)\}_{i \in I}}{\Gamma \models \mathsf{case}\ t \to [C_i(\tilde{x_i}).e_i]_{i \in I} : \bigcup \mathcal{B}_i}\ \text{case}$$

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad \Gamma \models e : \mathcal{B} \bigcup \{\mathrm{id}(T_1, \ldots, T_n)\}}{\Gamma \models \mathsf{loop}\ \mathsf{id}(x_1 \leftarrow t_1 : T_1, \ldots, x_n \leftarrow t_n : T_n)\{e\} : \mathcal{B}}\ \text{loop expression}$$

$$\frac{\tilde{y} : \Gamma \models e : \mathcal{B} \bigcup \{b(\Delta)\} \quad \tilde{x} : \Delta \models e' : \mathcal{B}'}{\tilde{y} : \Gamma \models e\ \mathsf{entry}\ b(\tilde{x})\{e'\} : \mathcal{B} \bigcup \mathcal{B}'}\ \text{control expression cut rule}$$

$$\frac{\Gamma \vdash t : X \quad \Gamma, x : X \models e : \mathcal{B}}{\Gamma \models e[t/x] : \mathcal{B}}\ \text{expression cut rule}$$

Table 3.2: The inference rules for control expressions

- Exit statement

  The type of an exit statement is given by the following rule.

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad b(T_1, \ldots, T_n)}{\Gamma \models \mathsf{exit}\ b(t_1, \ldots, t_n) : \{b\}}\ \text{exit}$$

The symbol b is either an entry, a loop or a branch specification. The type checker type-checks all expressions, $t_1, \ldots, t_n$, that are passed to the exit statement. Not only should these expression be correctly typed, but also their types should match the expected argument types, $T_1, \ldots, T_n$. If all these checks succeed, the singleton set consisting of the branch identifier $\{\mathsf{b}\}$ will be returned: this is the type of the exit statement. The following is a proof of the statement on line 35 from figure 2.2.

$$\frac{\Gamma \vdash \text{ final :Real} \quad \text{printGrade } : \mathcal{B}(\text{x:Real})}{\Gamma \models \mathsf{exit}\ \text{printGrade(final :Real)} : \{\text{printGrade}\}}\ \text{exit}$$

- Return statement

The type of a return statement is given by the following rule.

$$\frac{\Gamma \vdash t : T}{\Gamma \models \mathsf{return}\ t : \{\mathsf{return}(T)\}}\ \text{return}$$

The type checker first type-checks the expression $\mathtt{t}$. A type $\mathtt{T}$ will be returned if $\mathtt{t}$ is successfully checked. And the return statement has the type $\{\mathsf{return}(T)\}$. The following is a proof of the statement on line 27 in figure 2.2.

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \text{a : Real} \quad \Gamma \vdash \text{0.4 : Real}}{\Gamma \vdash \text{a}{*}\text{0.4 : Real}} \quad \dfrac{\Gamma \vdash \text{b : Real} \quad \Gamma \vdash \text{0.6 : Real}}{\Gamma \vdash \text{b}{*}\text{0.6 : Real}}}{\Gamma \vdash \text{(a}{*}\text{0.4+b}{*}\text{0.6) : Real}} \quad \Gamma \vdash \text{2 : Real}}{\dfrac{\Gamma \vdash \text{(a}{*}\text{0.4+b}{*}\text{0.6)/2 : Real}}{\Gamma \models \mathsf{return}\ \text{(a}{*}\text{0.4+b}{*}\text{0.6)/2} : \{\mathsf{return}(\text{Real})\}}\ \text{return}}$$

- Control statement

The type of a control statement is given by the following rule.

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad \Gamma, \tilde{x_i} : \Delta_i \models e_i : \mathcal{B}_i \quad g : \mathcal{G}(\tilde{T} \to \{b_i(\Delta_i)\}_{i \in I})}{\Gamma \models g(t_1, \ldots, t_n) \to [b_i(\tilde{x_i}).e_i]_{i \in I} : \bigcup \mathcal{B}_i}\ \text{control}$$

Here g is a control operation that can be found in the control operation declarations, represented by $\mathcal{G}$. The Floey type checker type-checks the expressions, which are applied to the control operation g. These expressions, $t_1, \ldots, t_n$, are expected to match the argument types, $T_1, \ldots, T_n$. In each branch, branch variables, $\tilde{x_i}$, should be of the expected types, Δ_i, and are bound to the control expression in that branch, e_i. The type of a control statement is the union of the types of the control expressions in its branches.

- Case statement

  The type of a case statement is as follows.

$$\frac{\Gamma \vdash t :: C\ a_1\ \ldots\ a_n \quad \Gamma, \tilde{x_i} : \Delta_i \models e_i : \mathcal{B}_i \quad \mathsf{data}\ C\ a_1\ \ldots\ a_n\ = \{C_i(\Delta_i)\}_{i \in I}}{\Gamma \models \mathsf{case}\ t \rightarrow [C_i(\tilde{x_i}).e_i]_{i \in I} : \bigcup \mathcal{B}_i}\ \text{case}$$

  Here C is a data type that is already declared. The expression t as well as the branch constructors C_i should match the type of C. If these type-checking steps succeed, the branch variables, $\tilde{x_i}$, of the expected types Δ, are bound to the control expression, e_i. The type of a case statement is the union of the types of the control expressions in its branches.

- Loop expression

  The type of a loop expression is given in the following rule.

$$\frac{\Gamma \vdash t_1 : T_1, \ldots, t_n : T_n \quad \Gamma \models e : \mathcal{B} \bigcup \{\mathrm{id}(T_1, \ldots, T_n)\}}{\Gamma \models \mathsf{loop}\ \mathsf{id}(x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n)\{e\} : \mathcal{B}}\ \text{loop expression}$$

  The initialization expressions in the bindings, $t_1, \ldots, t_n$, are type-checked. The type of a loop expression is the type of the control expression in its loop body, without the loop label.

- Entry

  In Floey, entries are designed as the interfaces for control expressions (section 2.5). In the type system, entries are presented by the following cut rule.

  $$\frac{\tilde{y} : \Gamma \models e : \mathcal{B} \bigcup \{b(\Delta)\} \quad \tilde{x} : \Delta \models e' : \mathcal{B}'}{\tilde{y} : \Gamma \models e \ \mathsf{entry}\ b(\tilde{x})\{e'\} : \mathcal{B} \bigcup \mathcal{B}'} \text{ control expression cut rule}$$

  When a control expression exits to an entry, the entry can be included in the control expression by applying this cut rule. The type of the resulting structure is the union of the type of the control expression, without the entry label, and the type of the control expression inside the entry.

- Expression cut

  Using the expression cut rule below, one can substitute a variable by another one of the same type in a control expression.

  $$\frac{\Gamma \vdash t : X \quad \Gamma, x : X \models e : \mathcal{B}}{\Gamma \models e[t/x] : \mathcal{B}} \text{ expression cut rule}$$

3.1.3 The inference rules for functions and control operations

Type checking functions and control operations are more complicated in Floey, as functions and control operations in a scope are allowed to be defined in arbitrary order, which means that use of a function or a control operation can happen before it is defined. To resolve this issue, we separate the declarations of functions and control operations (i.e. the introduced symbol and its type) from their body. When a function or control operation call is encountered, its declaration is used to type check uses. The type checking of its body, on the other hand, may be delayed to the point when the declarations used in the body are available to the type checker.

$$\frac{}{\{\}|\{\} \leadsto \{\}|\{\}}\ \text{empty axiom}$$

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D}}{f : \tilde{T} \to T, \mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D}}\ \text{fun weakening}$$

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D}}{\mathcal{F}|g : \tilde{T} \to \mathcal{B}, \mathcal{G} \leadsto \mathcal{O}|\mathcal{D}}\ \text{control weakening}$$

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D} \quad \Gamma \vdash_{\mathcal{F},\mathcal{G}} t : T}{\mathcal{F}|\mathcal{G} \leadsto \{\Gamma \vdash t : T\} \cup \mathcal{O}|\mathcal{D}}\ \text{discharging expression obligation}$$

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D} \quad \Gamma \models_{\mathcal{F},\mathcal{G}} e : \mathcal{B}}{\mathcal{F}|\mathcal{G} \leadsto \{\Gamma \models e : \mathcal{B}\} \cup \mathcal{O}|\mathcal{D}}\ \text{discharging control expression obligation}$$

$$\frac{f : \tilde{T} \to T, \mathcal{F}|\mathcal{G} \leadsto \{\Gamma \models e : \{\mathsf{return}(T)\}\} \cup \mathcal{O}|\mathcal{D}}{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathsf{fun}\ f(\tilde{T}) : T\{e\} \cup \mathcal{D}}\ \text{function definition}$$

$$\frac{\mathcal{F}|g : \tilde{T} \to \mathcal{B}, \mathcal{G} \leadsto \{\Gamma \models e : \mathcal{B}\} \cup \mathcal{O}|\mathcal{D}}{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathsf{def}\ g(\tilde{T})[\mathcal{B}]\{e\} \cup \mathcal{D}}\ \text{control definition}$$

Table 3.3: The inference rules for functions and control operations

The inference rules for type checking functions and control operations in Floey are given in table 3.3. These rules actually just model type checking in a single scope: as local function and control operation definitions are allowed in Floey, the Floey type checker, in practice, also have to handle these scoping issue. A typical predicate for type checking functions and control operations is as follows.

$$\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D}$$

Here $\mathcal{F}$ and $\mathcal{G}$ are sets of function and control operation declarations, which may or may not have been type checked. $\mathcal{O}$ is the set of (type checking) obligations where functions and control operations are possibly used. $\mathcal{D}$ is the set of definitions of functions and control operations.

Type checking begins with $\mathcal{F}$ and $\mathcal{G}$ consisting of only the pre-defined primitive functions and control operations, and empty $\mathcal{O}$ and $\mathcal{D}$. When type checking in a scope, all the function and control operation definitions in the scope are first collected and added into $\mathcal{D}$. Using two rules below, definitions can be removed from $\mathcal{D}$. Their interfaces are added into $\mathcal{F}$ or $\mathcal{G}$, and the obligations formed by their bodies of are added into $\mathcal{O}$.

$$\frac{f : \tilde{T} \to T, \mathcal{F}|\mathcal{G} \leadsto \{\Gamma \models e : \{\mathsf{return}(T)\}\} \cup \mathcal{O}|\mathcal{D}}{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathsf{fun}\ f(\tilde{T}) : T\{e\} \cup \mathcal{D}} \text{ function definition}$$

$$\frac{\mathcal{F}|g : \tilde{T} \to \mathcal{B}, \mathcal{G} \leadsto \{\Gamma \models e : \mathcal{B}\} \cup \mathcal{O}|\mathcal{D}}{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathsf{def}\ g(\tilde{T})[\mathcal{B}]\{e\} \cup \mathcal{D}} \text{ control definition}$$

At this stage, $\mathcal{D}$ is empty and $\mathcal{F}$ and $\mathcal{G}$ contain all the declarations in the scope. The obligations in $\mathcal{O}$ need to be type checked based on the information held in $\mathcal{F}$ and $\mathcal{G}$. The following two discharging rules allow one to remove obligations from $\mathcal{O}$ and to type check them by applying the inference rules in section 3.1.1 and 3.1.2 .

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D} \quad \Gamma \vdash_{\mathcal{F},\mathcal{G}} t : T}{\mathcal{F}|\mathcal{G} \leadsto \{\Gamma \vdash t : T\} \cup \mathcal{O}|\mathcal{D}} \text{ discharging expression obligation}$$

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D} \quad \Gamma \models_{\mathcal{F},\mathcal{G}} e : \mathcal{B}}{\mathcal{F}|\mathcal{G} \leadsto \{\Gamma \models e : \mathcal{B}\} \cup \mathcal{O}|\mathcal{D}} \text{ discharging control expression obligation}$$

Once all the obligations have been removed from $\mathcal{O}$, one can apply the following weakening rules to $\mathcal{F}$ and $\mathcal{G}$, in order to delete the declarations in them. As the rules allows declarations which do not occur in $\mathcal{O}$ to be deleted from $\mathcal{F}$ and $\mathcal{G}$.

$$\frac{\mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D}}{f : \tilde{T} \to T, \mathcal{F}|\mathcal{G} \leadsto \mathcal{O}|\mathcal{D}} \text{ fun weakening}$$

$$\frac{\mathcal{F}|\mathcal{G} \rightsquigarrow \mathcal{O}|\mathcal{D}}{\mathcal{F}|g : \tilde{T} \rightarrow \mathcal{B}, \mathcal{G} \rightsquigarrow \mathcal{O}|\mathcal{D}} \text{ control weakening}$$

Should all the type checking steps succeed, $\mathcal{F}$, $\mathcal{G}$, $\mathcal{O}$ and $\mathcal{D}$ will all end up empty, as shown in the empty axiom below.

$$\frac{}{\{\}|\{\} \rightsquigarrow \{\}|\{\}} \text{ empty axiom}$$

3.2 Equalities

This section discusses the equalities that hold for Floey control expressions. These equalities are used to govern the transformation steps discussed in chapter 4 and 5, ensuring the code modifications are valid.

The section is organized as follows. Section 3.2.1 introduces the basic equality of control expressions and its properties. Section 3.2.2 discusses the three equalities of decisions: idempotence, transposition and repetition. The important consequences of decision equalities are give in section 3.2.3. While section 3.2.4 focuses on loop equalities: loop binding weakening, infinite loops, loop unrolling, loop motion and loop splitting, the consequences of which are discussed in section 3.2.5.

3.2.1 Basic equalities

An equality of control expressions has the form:

$$e_1 =_\Gamma e_2 : \mathcal{B}$$

where Γ is the context of the equality and $\mathcal{B}$ is the type of the two control expressions. $\Gamma \vdash e_1 : \mathcal{B}$ must be a valid control inference rule as must $\Gamma \vdash e_2 : \mathcal{B}$.

The following are the three basic properties of equality: reflexivity, symmetry and transitivity.

$$\frac{\Gamma \vdash e : \mathcal{B}}{e =_\Gamma e : \mathcal{B}} \text{ reflexive} \quad \frac{e_1 =_\Gamma e_2 : \mathcal{B}}{e_2 =_\Gamma e_1 : \mathcal{B}} \text{ symmetric}$$

$$\frac{e_1 =_\Gamma e_2 : \mathcal{B}, e_2 =_\Gamma e_3 : \mathcal{B}}{e_1 =_\Gamma e_3 : \mathcal{B}} \text{ transitive}$$

Other than the above basic properties, the following substitution property should also hold for equalities. It says that after substituting two equal variables, t_1 and t_2, for x in control expression e_1 and e_2, the equality between the two control expressions still hold.

$$\frac{e_1 =_{x:X,\Gamma} e_2 : \mathcal{B} \quad t_1 =_\Gamma t_2 : X}{e_1[t_1/x] =_\Gamma e_2[t_2/x] : \mathcal{B}} \text{ substitution}$$

3.2.2 Decision equalities

This section introduces the equalities of decisions (without loops) and how they are used in the optimization algorithms. The equalities are: assignment expansion, idempotence, transposition and repetition.

- Assignment expansion

$$x := s \mathrel{.} q(\tilde{t}) \to [b_i(\tilde{y_i}).e]_{i\in I} \;=_\Gamma\; q(\tilde{t}[s/x]) \to [b_i(\tilde{y_i}).e]_{i\in I} \tag{3.1}$$

The equality states that an assignment from s to x can be removed by substituting x for s in the decision that follows. Note that this is not a general equality and is only valid under the condition that x is only used in the terms $\tilde{t}$. The equality is only used to justify the code manipulation in expression expansion (section 4.1.2), in which the above condition is always true.

- Idempotence

$$q(\tilde{x}) \rightarrow [b_i(\tilde{x_i}).e]_{i\in I} \;=_\Gamma\; e|_{q(\tilde{x})} \tag{3.2}$$

The equality states that if the control expressions in each branch of a decision is the same, and the binding $b_i(\tilde{x_i})$ does not capture any variable in e as all variables used in e come from the context Γ. Then the decision can be replaced by e provided the control operation $q(\tilde{x})$ terminates. In Floey, the only case that a control operation or a function does not terminate is when it contains an infinite looping.

This equality is used extensively in the reduction algorithm (section 5.2) to remove computational steps (in idempotent reduction, see section 5.2.4).

- Transposition

$$\begin{aligned} & q_1(\tilde{x}) \rightarrow [b_i(\tilde{x_i}).q_2(\tilde{y}) \rightarrow [b_j(\tilde{y_{ij}}).e_{ij}[\tilde{x_i}/\tilde{x_{ij}}]]_{j\in J}]_{i\in I} \\ =_\Gamma\; & q_2(\tilde{y}) \rightarrow [b_j(\tilde{y_j}).q_1(\tilde{x}) \rightarrow [b_i(\tilde{x_{ij}}).e_{ij}[\tilde{y_j}/\tilde{y_{ij}}]]_{i\in I}]_{j\in J} \end{aligned} \tag{3.3}$$

This equality says that if the same decision $q_2(\tilde{y})$ occurs in every branch of another decision, and all the variables $\tilde{y}$ used in that decision come from the context Γ, rather than from binding $b_i(\tilde{x_i})$, they can be exchanged.

- Repetition

$$\begin{aligned} & q(\tilde{x}) \rightarrow [b_1(\tilde{x_1}).e_1 \mid \ldots \mid b_i(\tilde{x_i}).q(\tilde{x}) \rightarrow [b_j(\tilde{x'_j}).e'_j]_{1\leq j\leq n} \mid \ldots \mid b_n(\tilde{x_n}).e_n] \\ =_\Gamma\; & q(\tilde{x}) \rightarrow [b_1(\tilde{x_1}).e_1 \mid \ldots \mid b_i(\tilde{x_i}).e'_i[\tilde{x_i}/\tilde{x'_i}] \mid \ldots \mid b_n(\tilde{x_n}).e_n] \end{aligned} \tag{3.4}$$

This equality says that whenever two nested decisions have the same control operation with same binding for their parameters, they return the same branch constructor. The inner decision can be then omitted by selecting the control expression below that branch constructor, where an appropriate substitution $[\tilde{x}_i/\widetilde{x'_i}]$ of bound variables is required.

The repetition equality is fundamental to the repeat reduction algorithm (section 5.1), although in Floey, additional constructs must be considered (such as case statements, assignment statements and data types).

3.2.3 Consequences of decision equalities

This section discusses some important consequences of the decision equalities introduced in section 3.2.2. These consequences are used to justify code manipulations in various optimizations as well as to prove some consequences discussed later in section 3.2.5.

- Pulling up semi-essentials

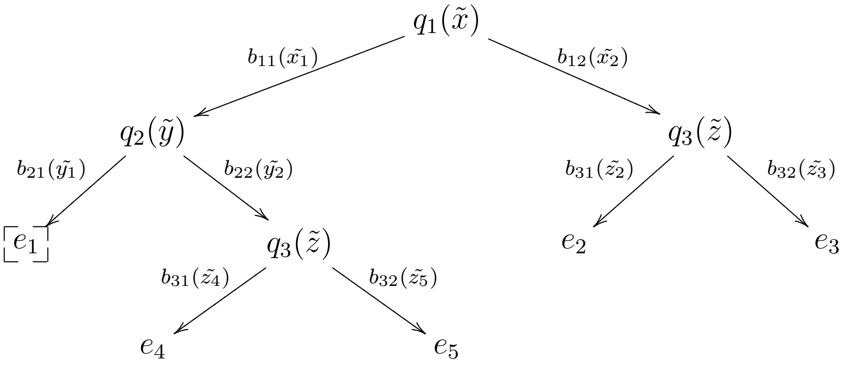
  Transposition equality gives the important property that semi-essentials in control expressions without loops can be pulled up to the root. A **semi-essential** decision in control expression e, or a semi-essential in short, is a decision q that occurs in every path from the root to the exits and doesn't use any bound variable produced by other decisions.

  When pulling up semi-essentials to the root of control expression, transposition is used to exchange decisions, until the semi-essential reaches the root.

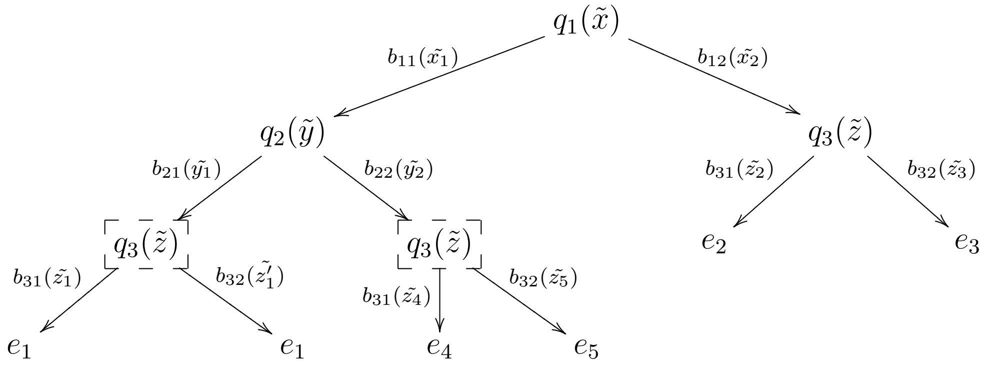
- Pulling up arbitrary decisions

When pulling up an arbitrary decision, which is not a semi-essential, to the root of a control expression, idempotence equality is also needed as shown in the following example.

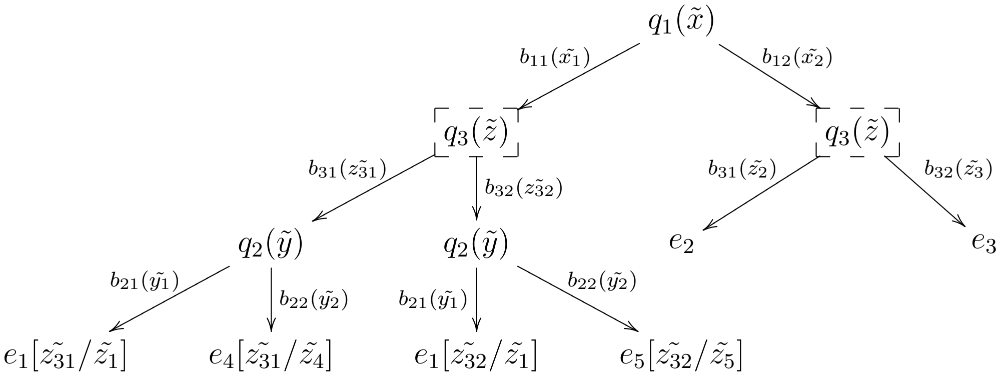
This is the orginal control expression where $q_3(\tilde{z})$ is not a semi-essential.



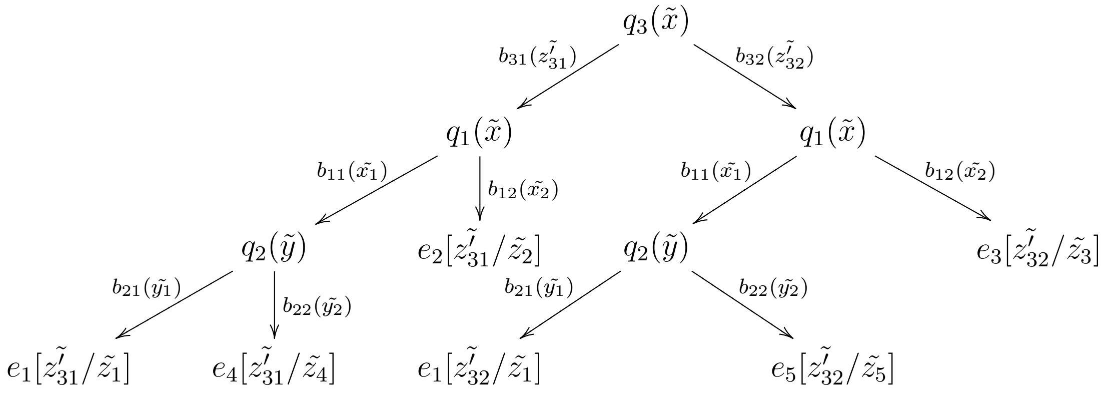
The first step of pulling up $q_3(\tilde{z})$ is to introduce it above e_1 using idempotence equality (3.2). Note that the idempotence used here is an equality only if $q_3(\tilde{z})$ terminates. However, as this consequence is only used in justifying repeat reduction in general form (see the consequence deep repetition below), where $q_3(\tilde{z})$ is a repetition and has occurred before, in this case the termination of $q_3(\tilde{z})$ introduced by idempotence doesn't have any effect. The equality thus holds.



Transpose $q_3(\tilde{z})$ above $q_2(\tilde{y})$, using transposition (3.3).



Transpose $q_3(\tilde{z})$ to the root, using transposition (3.3).

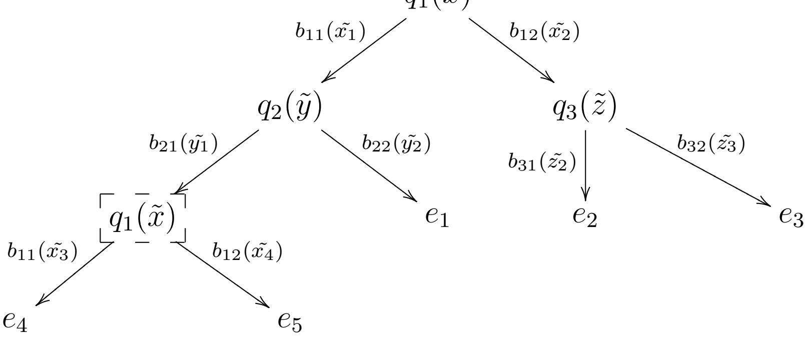

$q_3(\tilde{x})$
$b_{31}(\tilde{z'_{31}})$
$b_{32}(\tilde{z'_{32}})$
$q_1(\tilde{x})$
$q_1(\tilde{x})$
$b_{11}(\tilde{x_1})$
$b_{12}(\tilde{x_2})$
$b_{11}(\tilde{x_1})$
$b_{12}(\tilde{x_2})$
$q_2(\tilde{y})$
$e_2[\tilde{z'_{31}}/\tilde{z_2}]$
$q_2(\tilde{y})$
$e_3[\tilde{z'_{32}}/\tilde{z_3}]$
$b_{21}(\tilde{y_1})$
$b_{22}(\tilde{y_2})$
$b_{21}(\tilde{y_1})$
$b_{22}(\tilde{y_2})$
$e_1[\tilde{z'_{31}}/\tilde{z_1}]$
$e_4[\tilde{z'_{31}}/\tilde{z_4}]$
$e_1[\tilde{z'_{32}}/\tilde{z_1}]$
$e_5[\tilde{z'_{32}}/\tilde{z_5}]$

The above demonstrates the sequence of manipulations needed to pull an arbitrary decision to the root of a control expression.

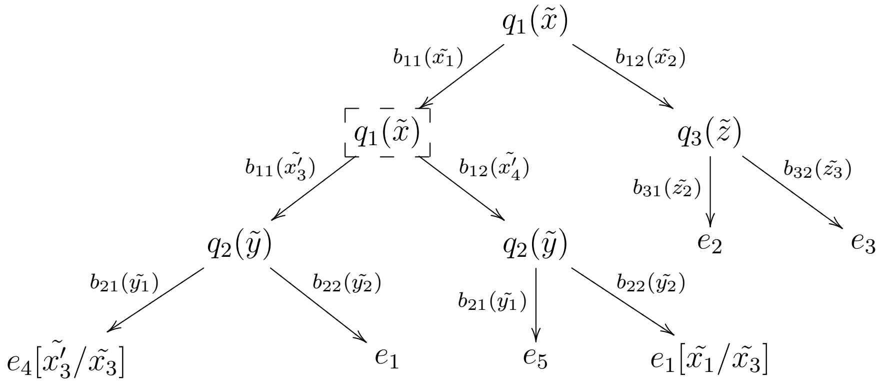
- Deep repetition

The general form of repeat reduction allows decisions to be separated by many intermediate decisions. Using the consequence of pulling up arbitrary decisions, a deep repetition can be first reduced to an immediate repetition. The repetitive decision is then eliminated as shown in the following example.

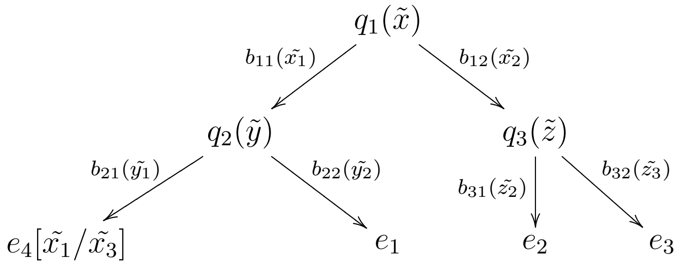
This is the original control expression, where $q_1(\tilde{x})$ is a deep repetition:



Pull $q_1(\tilde{x})$ above $q_2(\tilde{y})$, using the consequence of pulling up arbitrary decisions.



Apply repetition equality (3.4) to $q_1(\tilde{x})$.



The above demonstrates the sequence of code manipulations needed to verify deep repeat reduction.

3.2.4 Loop equalities

This section introduces the five equalities of control expression of loops which are used in this thesis: loop binding weakening, infinite loops, loop unrolling, loop motion and loop splitting.

- Loop binding weakening

$$\begin{aligned}&\mathsf{loop\ id}(x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n, z \leftarrow s)\{e[\mathsf{exit\ id}(y_1, \ldots, y_n, z)/\mathsf{exit\ id}(y_1, \ldots, y_n)]\}\\ =_\Gamma\ &\mathsf{loop\ id}(x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n)\{e[s/z]\} \qquad (3.5)\end{aligned}$$

A loop binding is unchanged if in all exit statements jumping to the loop header the same variable is returned. This equality states that if a loop binding is neither changed or used in the loop, the binding can be eliminated from loop header with necessary variable substitution in loop body.

This equality is used in loop binding reduction, where all unchanged or unused loop bindings are removed (section 4.4.4).

- Infinite loops

$$\mathsf{loop\ id}(x_i \leftarrow t_i)_{i\in I}\{e\} \ =_\Gamma\ \mathsf{loop\ id}()\{\mathsf{exit\ id}()\} : \emptyset \qquad (3.6)$$

A loop expression can be identified as an infinite loop, if all branches of in the loop body jump back to the loop header, that is there is no exit pointing to outside of the loop. Such infinite loops can be identified in type checking as the loop body has the type of $\emptyset$. All the loop bindings of an infinite loops can be removed with the loop body being replace by a simple exit statement jumping back to the loop header.

This equality verifies the code modification of infinite invariant code removal (section 4.4.2) in loop normalization.

- Loop unrolling

$$\begin{aligned}&\mathsf{loop\ id}(x_i \leftarrow t_i)_{i\in I}\{e\} \qquad (3.7)\\ =_\Gamma\ &e[t_i/x_i]_{i\in I}[\mathsf{loop\ id}(x_i \leftarrow w_i)_{i\in I}\{e\}/\mathsf{exit\ id}(w_i)_{i\in I}]\end{aligned}$$

The equality says that the first iteration of a loop can be unrolled by pulling the loop body out of loop and replacing each exit statement that jumps to the loop header with the actual loop expression. Free bindings substitute for loop bindings in e. As loop expressions start at the second iteration as the original one, necessary free binding substitution $[w_i/t_i]_{i \in I}$ is applied to address that.

- Loop motion

$$\begin{aligned} & \mathsf{loop\ id}(x_i \leftarrow t_i)_{i \in I}\{q(\tilde{x}) \rightarrow [b_j(y_j).e_j]_{j \in J}\} \\ =_\Gamma\ & q(\tilde{x}) \rightarrow [b_j(y_j).\mathsf{loop\ id}(x_i \leftarrow t_i)_{i \in I}\{e_j\}]_{j \in J} \end{aligned} \tag{3.8}$$

The equality says that if the first decision in the loop body is not dependent on loop header as all variable $\tilde{x}$ come from context Γ, it can be moved out of the loop.

- Loop splitting

$$\begin{aligned} & \mathsf{loop\ id}(x_i \leftarrow t_i)_{i \in I}\{e[\mathsf{exit\ id}(\tilde{y})/\mathsf{exit\ id'}(\tilde{y})]\} \\ =_\Gamma\ & \mathsf{loop\ id}(x_i \leftarrow t_i)_{i \in I}\{\mathsf{loop\ id'}(x_i \leftarrow x_i)_{i \in I}\{e\}\} \end{aligned} \tag{3.9}$$

This equality states an arbitrary way of splitting a loop expression into a nested looping structure of two headers. The newly generated outer loop has the same header of the original one. The free loop bindings of the inner loop all come from the local loop bindings of the original loop.

3.2.5 Consequences of loop equalities

This section focuses on several important consequences of the loop equalities. These consequences are used to verify various code manipulations used in loop normalization (section 4.4), loop reduction (section 5.2.6), and invariant code removal (section 5.3).

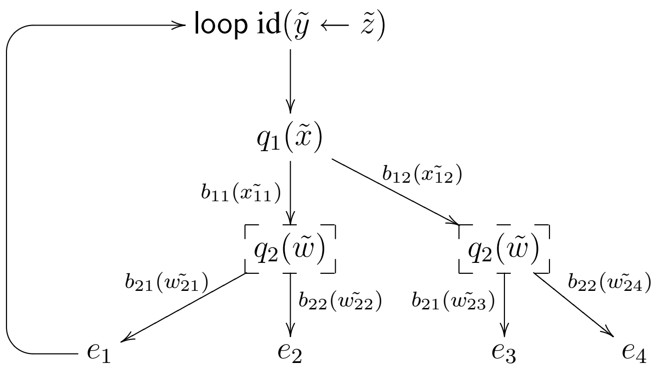
- Unlooping

  Unlooping algorithm removes a loop expression, which has no exit statement jumping back to the loop header, with its loop body (see section 4.4.3). The elimination such loop expression is just a special case of loop unrolling, when the substitution $[\textsf{loop id}(x_i \leftarrow w_i)_{i\in I}\{e\}/\textsf{exit id}(w_i)_{i\in I}]$ never takes place and the loop expression is removed.

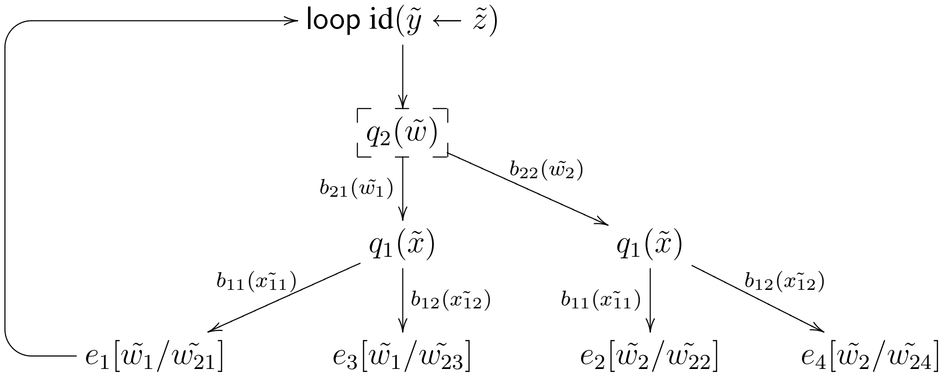
- Pulling out semi-essentials from loops

  The optimization that pulls semi-essentials out of loops (section 5.3.2) can be verified by using a sequence of transpositions (3.3) and a loop motion (3.8). All semi-essentials in a loop can be first pulled to the root in loop body (given by transposition). They can be then be pulled out of loop expressions (given by loop motion). The following is an example of this manipulation.

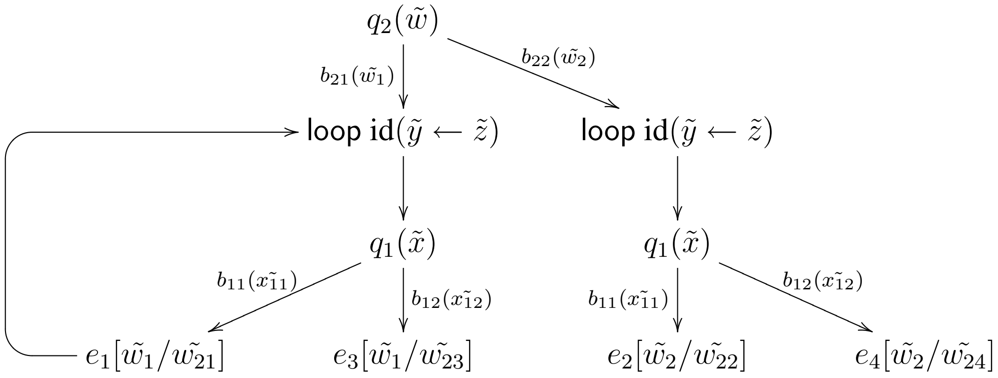
  This is the original control expression, where $q_2(\tilde{w})$ is a semi-essential in the loop.



  Transpose (3.3) $q_2(\tilde{w})$ to the root of the loop body.



Move $q_2(\tilde{w})$ out of the loop (given by loop motion 3.8).


$q_2(\tilde{w})$
$b_{21}(\tilde{w_1})$
$b_{22}(\tilde{w_2})$
loop id($\tilde{y} \leftarrow \tilde{z}$)
loop id($\tilde{y} \leftarrow \tilde{z}$)
$q_1(\tilde{x})$
$q_1(\tilde{x})$
$b_{11}(\tilde{x_{11}})$
$b_{12}(\tilde{x_{12}})$
$b_{11}(\tilde{x_{11}})$
$b_{12}(\tilde{x_{12}})$
$e_1[\tilde{w_1}/\tilde{w_{21}}]$
$e_3[\tilde{w_1}/\tilde{w_{23}}]$
$e_2[\tilde{w_2}/\tilde{w_{22}}]$
$e_4[\tilde{w_2}/\tilde{w_{24}}]$

- Deep repetition for loops

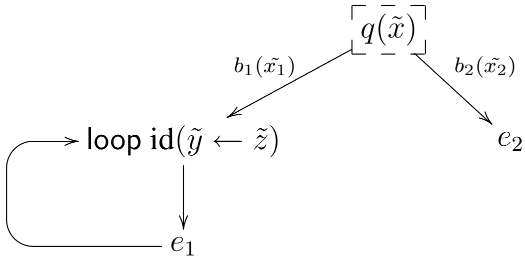
  The above together with deep repetition, can be used to justify repeat reduction through a loop. If a decision in a loop is a repetition of another one outside the loop, it must be an invariant decision of the loop and can be pulled to the top of the loop body then loop motion can be used to pull it out of the loop. The repetition can then be eliminated by deep repetition.

  This is used in the repeat reduction to eliminate deep repetitions in loops (section 5.1).

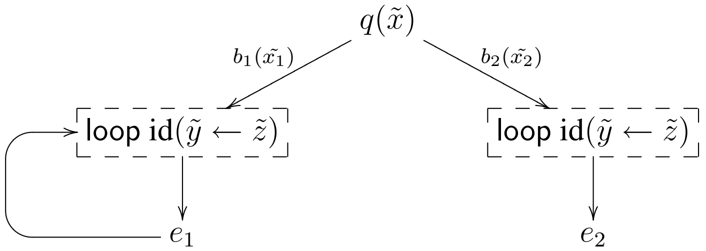
- Pushing decisions into loops

  The code modification of pushing an invariant decision **into** a loop, which is used in the reduction algorithm (section 5.2), can be justified by using loop motion equality (3.8) and unlooping as shown in the following example.

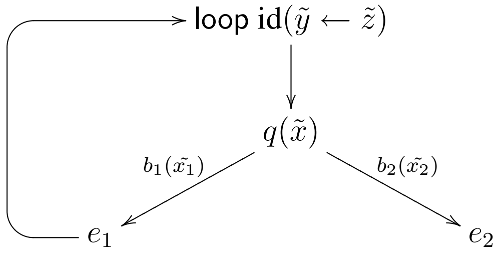
  $q(\tilde{x})$ is the invariant decision to be pushed into the loop.



  Apply unlooping on e_2.


$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
loop id$(\tilde{y} \leftarrow \tilde{z})$
loop id$(\tilde{y} \leftarrow \tilde{z})$
e_1
e_2

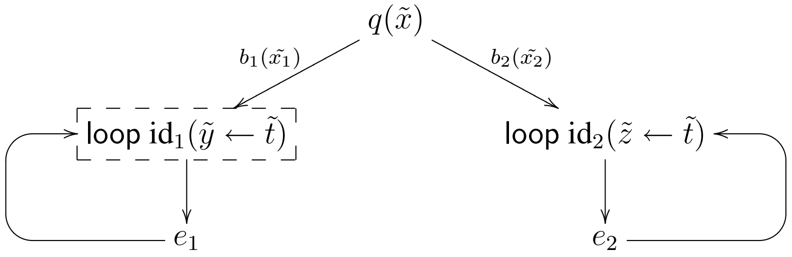
  Transpose $q(\tilde{x})$ below the loops by using loop motion (3.8).



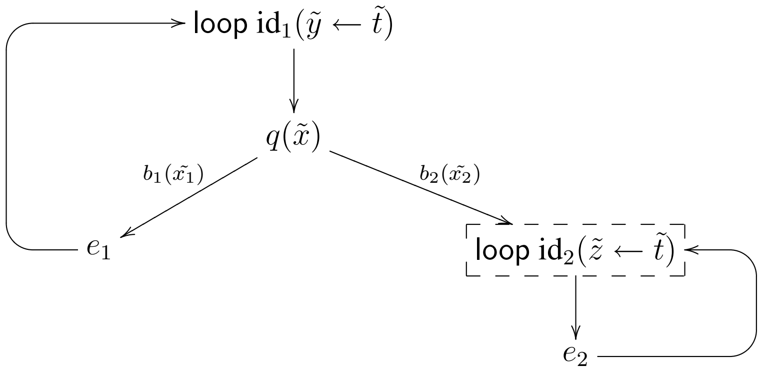
- Binding renaming

In loop reduction (5.2.6), independent loop headers are pulled up in the hope that new semi-essentials can be found. During the course of identifying these so called potential semi-essentials (see section 5.2.6 for definition), loop bound variables that are initialized the same need to be renamed th same. The binding renaming equality justifies this manipulation.

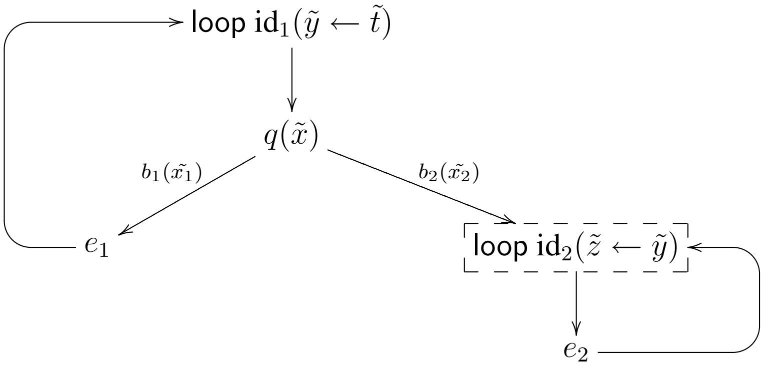
This is the original control expression, with two independent loops id$_1$ and id$_2$.


$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
loop id$_1(\tilde{y} \leftarrow \tilde{t})$
loop id$_2(\tilde{z} \leftarrow \tilde{t})$
e_1
e_2

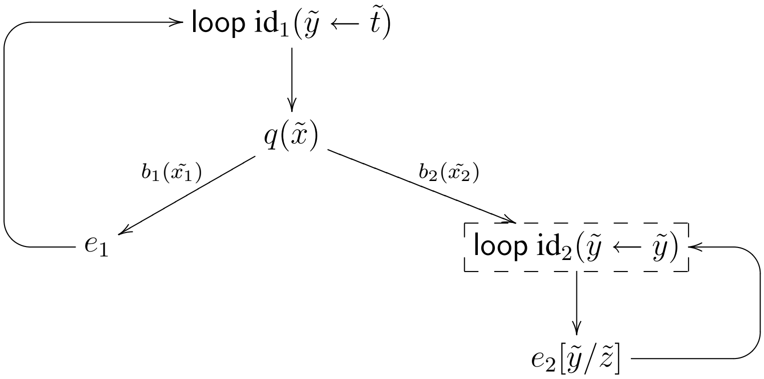
The loop header of id$_1$ is first pulled up to the root, given by the consequence of pushing decisions into loops.


loop id$_1(\tilde{y} \leftarrow \tilde{t})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1
loop id$_2(\tilde{z} \leftarrow \tilde{t})$
e_2

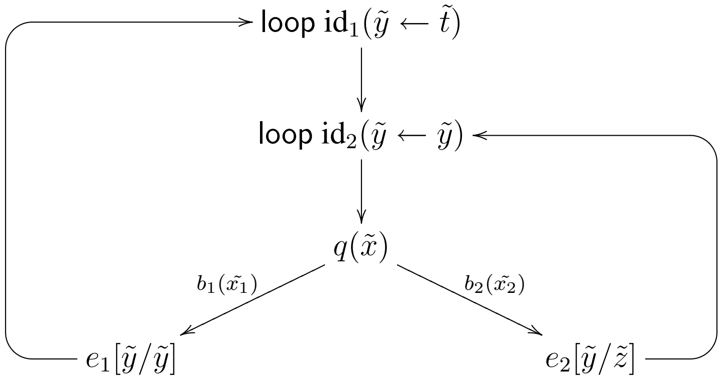
In the loop header of id$_2$, t is substituted by y, as y is initialized to t in loop id$_1$. Also since two loops are independent, when execution flow enters loop id$_2$, it will never be directed to loop id_1, the bound variable of id_1 is only initialized once.


loop $\mathrm{id}_1(\tilde{y} \leftarrow \tilde{t})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1
loop $\mathrm{id}_2(\tilde{z} \leftarrow \tilde{y})$
e_2

Bound variable substitution is applied in loop id_2.


loop $\mathrm{id}_1(\tilde{y} \leftarrow \tilde{t})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1
loop $\mathrm{id}_2(\tilde{y} \leftarrow \tilde{y})$
$e_2[\tilde{y}/\tilde{z}]$

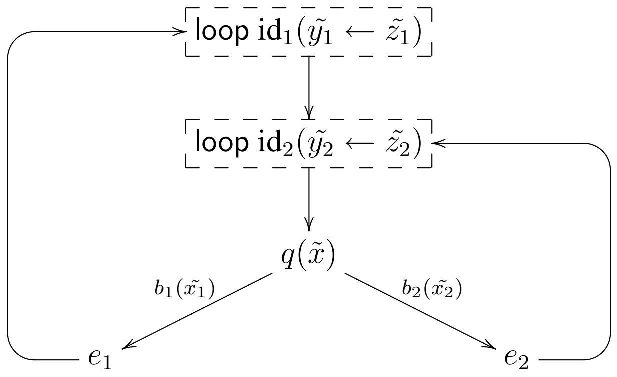
Finally, the loop header of id_2 is pulled up. As variable y is also used in e_1, substitution of y to y is applied but makes no real difference to e_1.


loop $\mathrm{id}_1(\tilde{y} \leftarrow \tilde{t})$
loop $\mathrm{id}_2(\tilde{y} \leftarrow \tilde{y})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
$e_1[\tilde{y}/\tilde{y}]$
$e_2[\tilde{y}/\tilde{z}]$

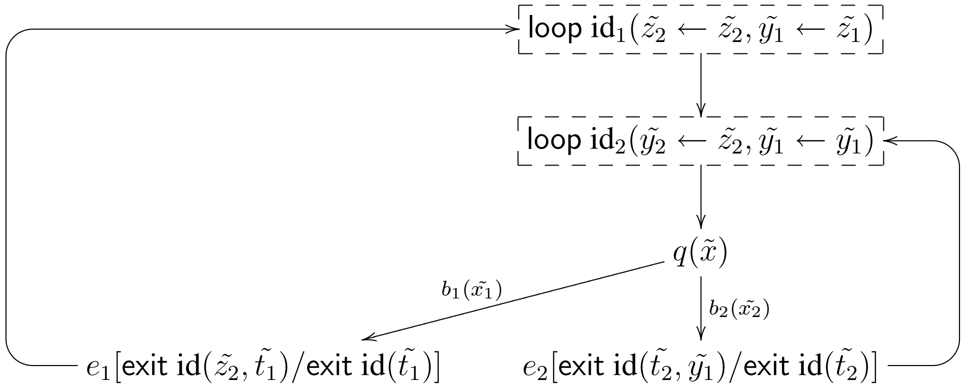
- Loop flipping

In invariant code removal, nested loop headers are flipped in order to pull out more semi-essentials from loops (section 5.3.4). This loop flipping algorithm uses the loop binding weakening (3.5) and loop splitting (3.9).

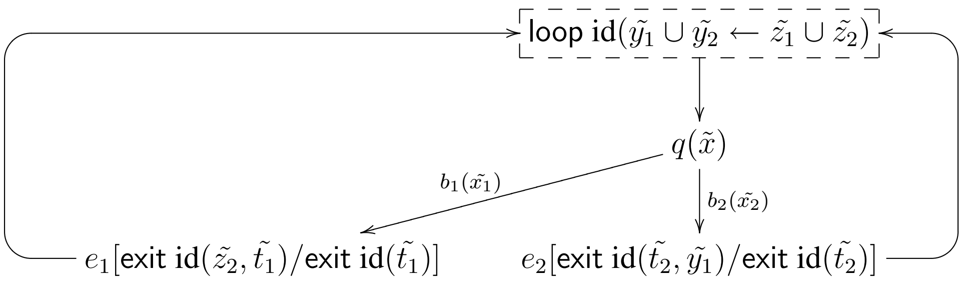
When flipping two loop headers, the bindings should be independent, that is in the following example, $\tilde{y_1} \cap \tilde{z_2} = \emptyset$ and $\tilde{y_1} \cap \tilde{y_2} = \emptyset$.


loop $\text{id}_1(\tilde{y_1} \leftarrow \tilde{z_1})$
loop $\text{id}_2(\tilde{y_2} \leftarrow \tilde{z_2})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1
e_2

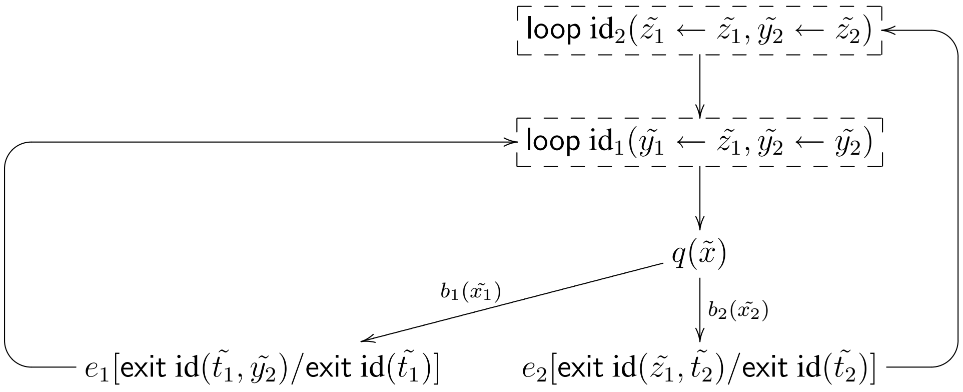
Weaken the loop bindings (3.5).


loop $\text{id}_1(\tilde{z_2} \leftarrow \tilde{z_2}, \tilde{y_1} \leftarrow \tilde{z_1})$
loop $\text{id}_2(\tilde{y_2} \leftarrow \tilde{z_2}, \tilde{y_1} \leftarrow \tilde{y_1})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
$e_1[\text{exit id}(\tilde{z_2}, \tilde{t_1})/\text{exit id}(\tilde{t_1})]$
$e_2[\text{exit id}(\tilde{t_2}, \tilde{y_1})/\text{exit id}(\tilde{t_2})]$

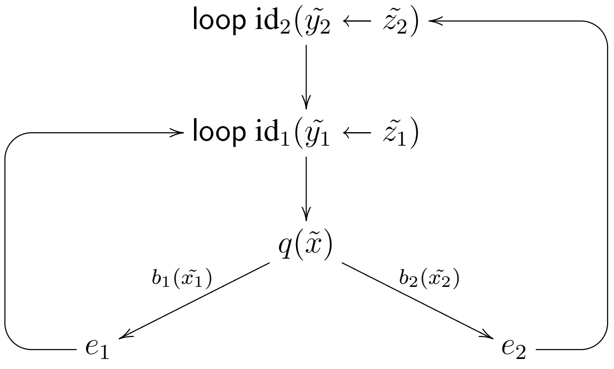
Amalgamate the loops by using loop splitting (3.9).


loop id$(\tilde{y_1} \cup \tilde{y_2} \leftarrow \tilde{z_1} \cup \tilde{z_2})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1[exit id$(\tilde{z_2}, \tilde{t_1})$/exit id$(\tilde{t_1})$]
e_2[exit id$(\tilde{t_2}, \tilde{y_1})$/exit id$(\tilde{t_2})$]

Split the loop again (3.9)


loop id$_2(\tilde{z_1} \leftarrow \tilde{z_1}, \tilde{y_2} \leftarrow \tilde{z_2})$
loop id$_1(\tilde{y_1} \leftarrow \tilde{z_1}, \tilde{y_2} \leftarrow \tilde{y_2})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1[exit id$(\tilde{t_1}, \tilde{y_2})$/exit id$(\tilde{t_1})$]
e_2[exit id$(\tilde{z_1}, \tilde{t_2})$/exit id$(\tilde{t_2})$]

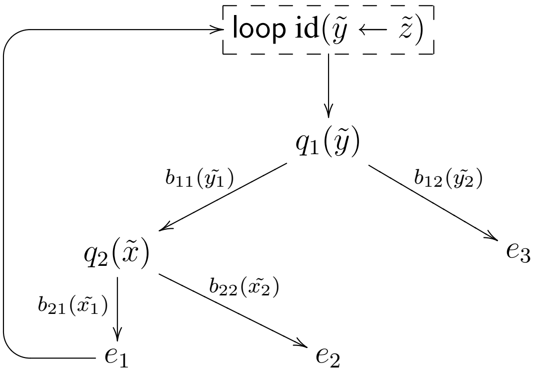
Remove the unchanged bindings (3.5).


loop id$_2(\tilde{y_2} \leftarrow \tilde{z_2})$
loop id$_1(\tilde{y_1} \leftarrow \tilde{z_1})$
$q(\tilde{x})$
$b_1(\tilde{x_1})$
$b_2(\tilde{x_2})$
e_1
e_2

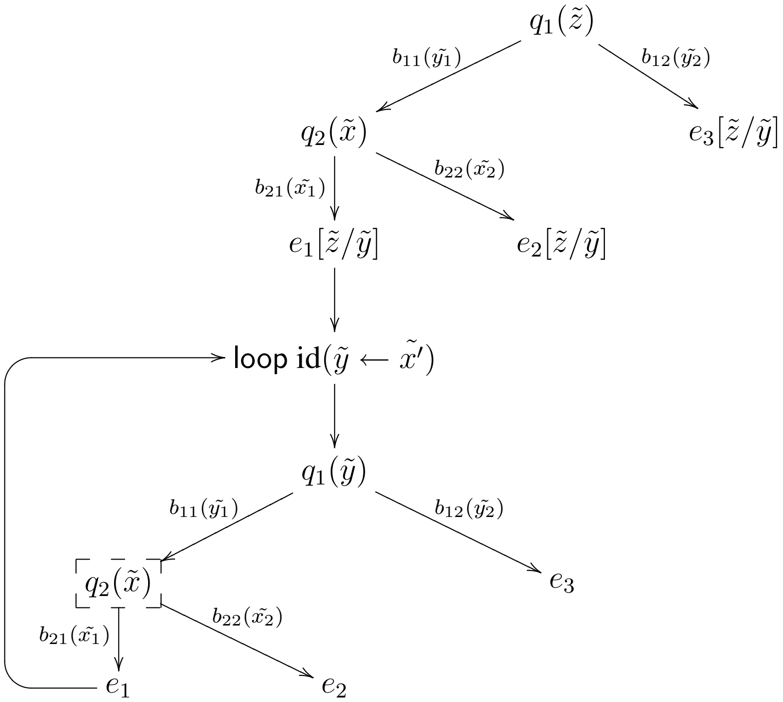
- Pulling out arbitrary invariant decision

The algorithm of pulling out an arbitrary invariant decision (section 5.3.3) can be verified by the unrolling equality (3.7) and deep repetition for loops, as shown in the following example.

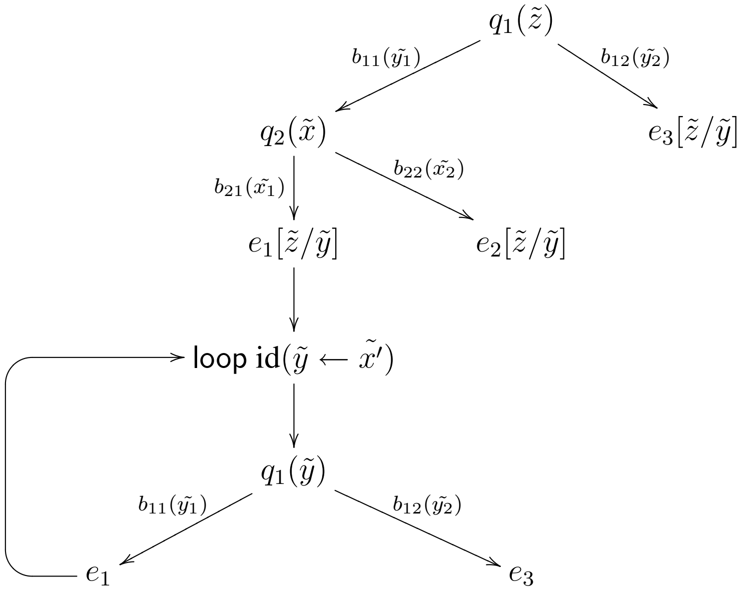
In this loop expression, the decision $q_2(\tilde{x})$ is a loop semi-essential but not a semi-essential.



Unroll the loop expression (3.7).



Eliminate the deep repetition $q_2(\tilde{x})$ (given by deep repetition for loops) of in the loop.


$q_1(\tilde{z})$
$b_{11}(\tilde{y_1})$
$b_{12}(\tilde{y_2})$
$q_2(\tilde{x})$
$e_3[\tilde{z}/\tilde{y}]$
$b_{21}(\tilde{x_1})$
$b_{22}(\tilde{x_2})$
$e_1[\tilde{z}/\tilde{y}]$
$e_2[\tilde{z}/\tilde{y}]$
$\mathsf{loop\ id}(\tilde{y} \leftarrow \tilde{x'})$
$q_1(\tilde{y})$
$b_{11}(\tilde{y_1})$
$b_{12}(\tilde{y_2})$
e_1
e_3

Chapter 4

Control normal form for Floey programs

This chapter discusses a normal form for Floey programs, called control normal form, which is designed to facilitate the optimizations of chapter 5. On programs in control normal form, implementing compiler optimizations becomes easier as a number of book-keeping issues concerned with variable scopes and naming have been brought to heel. The optimizing algorithms described in the next chapter are all based on the programs being in control normal form.

The Floey control normal form includes the following aspects:

- normalized expressions (section 4.1): constants are folded and expression are expanded to atomic operations
- normalized variables (section 4.2): every variable is uniquely defined and variable to variable and constant to variable assignments are removed
- normalized entries (section 4.3): every entry is uniquely named and put in the highest possible scope level in topological order; uniquely used entries are collected and unreachable ones are discarded; unused bindings are removed from entries
- normalized loops (section 4.4): every loop is uniquely named; infinite loops and loops that do not have any cyclic calls are identified and removed; unused or unchanged loop bindings are removed

The rest of this chapter will discuss how to convert Floey programs into the control normal form.

4.1 Normalizing expressions

This section discusses the basic normalizations that will facilitate the optimizations on expressions, such as common sub-expression elimination [Cocke, J., 1970], constant and copy propagation [Callahan et al., 1986, 2004; Vanbroekhoven et al., 2003; Wegman and Zadeck, 1991] and more generally the repeat reduction algorithm (section 5.1).

4.1.1 Constant folding

A constant expression is an expression with constant operands, which can be evaluated at compile time. Constant folding [Muchnick, 1997] replaces these expressions in a program by evaluating them to their constant values. For example the expression in the return statement, **return** (3+4*7), will be folded and converted to **return** 31.

In Floey, constant folding is implemented in the expression expansion phase (section 4.1.2). An expression is first constant folded and then split into atomic assignment instructions.

By moving such evaluations into compile time, the efficiency of resulting code at run time is improved. Furthermore, folding constant expressions in assignment generates constant assignments, which allows further constant propagation (section 4.2.2).

4.1.2 Expression expansion

Expression expansion splits complex expressions into multiple assignment instructions. Each added instruction is an “atomic” operation with an assignment to a temporary variable. The result of expression expansion facilitates common sub-expression elimination [Cocke, J., 1970] and more generally the repeat reduction algorithm (section 5.1).

return ((a*0.4+b*0.6)/2.0)	t_1 <- a * 0.4. t_2 <- b * 0.6. t_3 <- t_1 + t_2. t_4 <- t_3 / 2.0. **return** t_4
a	b

Table 4.1: Expression expansion

The example in table 4.1 demonstrates the effect of expression expansion on a return statement from figure 2.2. The return statement is expanded to a series of atomic assignment statements followed by a return statement acting on a variable. New temporaries are introduced by expression expansion and uniquely named in order to avoid name clashes. The expression expansion algorithm first traverses the whole control expression to find the largest existing subscript i associated with t. On the second traversal, it expands expressions and subscripts newly generated temporaries starting from $i + 1$.

4.1.3 Expression collection

Expression collection is not a normalization, but rather a process that reverses expression expansion and remove unnecessary assignments in order to generate more readable Floey programs. It collects assignments to variables which are used only once, back into the expression where they are used. For example, applying expression collection to the expanded code in table 4.1.b will result in the single return statement in 4.1.a.

Expression collection not only improves the readability of code, but also facilitates instruction selection [Appel, 1998] or other tree based algorithms on expressions [Sethi and Ullman, 1970]. Given larger expressions, the instruction selection phase has an better basis from which to generate efficient assembly code. Furthermore, if a variable is used only once, storing its value in a register brings no gain to the code as it does not avoid any re-computation but may increase the register pressure.

With no other code modification applied, every temporary introduced by the expression expansion is used strictly once. Therefore, if expression collection is applied right after expression expansion, it reverses the effect of expression expansion, cleaning up those temporaries introduced by expression expansion. However, the application of other optimizations may prevent this reversion, and some new temporaries can be left in resulting code.

The following example shows how the application of common sub-expression elimination (see section 5.1 on repeat reduction) may prevent expression collection from removing all temporaries introduced by expression expansion. In table 4.1.3.c, after the repeat reduction, t_1 is used three times and should be kept in the result, as it avoids repeat computations and therefore improves the code efficiency.

4.2 Normalizing variables

This section discusses the normalizations into static single assignment form. This arranges that each variable has a unique place of definition. At the same time we discuss constant and copy propagation.

Original	Expression expansion	Repeat reduction	Expression collection

case 3∗z>x **of**
[True.
  return 3∗z/2
|False.
  return 3∗z+10
]
	t_1<−3∗z.
	t_2<−t_1>x.
	case t_2 **of**
	[True.
	  t_3<−3∗z.
	  t_4<−t_3/2.
	  return t_4
	|False.
	  t_5<−3∗z.
	  t_6<−t_5+10.
	  return t_6
	]
		t_1<−3∗z.
		t_2<−t_1>x.
		case t_2 **of**
		[True.
		  t_4<−t_1/2.
		  return t_4
		|False.
		  t_6<−t_1+10.
		  return t_6
		]
			t_1<−3∗z.
			case t_1>x **of**
			[True.
			  return t_1/2
			|False.
			  return t_1+10
			]

a	b	c	d

Table 4.2: Expression collection

4.2.1 Static single assignment form

Static single assignment form (SSA) [Cytron et al., 1991], is an intermediate representation that is used in many optimizing compilers, such as GCC [Novillo, 2003, 2004] and SUIF [Stanford SUIF Compiler Group, 1994]. The essential property of SSA form is that whenever a variable is defined it is given a unique name. This means that if two variables have the same name they must also have the same value. Converting program into SSA form makes the bookkeeping issues for variable much simpler and improves the efficiency of optimization algorithms. Many compiler optimization algorithms are SSA form based [Chow et al., 1997; Gerlek et al., 1995; Sastry and Ju, 1998; VanDrunen and Hosking, 2004]. In Floey, the implementations of repeat reduction (section 5.1) and the reduction algorithm (section 5.2) also benefit from the simplicity of SSA form.

In Floey, the conversion to SSA form is accomplished by simply renaming each variable in every definition, including loop bindings, uniquely (see section 6.2.2). Similar to section 4.1.2, the unique variable names are generated by subscripting them with unique numbers. The following example in table 4.3 shows how a control expression is converted to SSA form.

```
fun foo(i:Int):Int             fun foo(i:Int):Int
{                              {
  begin                          begin
  x <- i * 2.                    x_1 <- i * 2.
  x <- x - 10.                   x_2 <- x_1 - 10.
  loop lp(x<-x)                  loop lp(x_3<-x_2)
  {                              {
    case x > 0 of                  case x_3 > 0 of
    [True.                         [True.
      y <- x * 4.                    y_4 <- x_3 * 4.
      return y                       return y_4
    |False.                        |False.
      y <- x * (-1).                 y_5 <- x_3 * (-1).
      exit lp(y)]                    exit lp(y_5)]
  }                              }
}                              }
```

Table 4.3: Static Single Assignment form

Floey programs must be normalized before they are passed to the optimization algorithms in the next chapter. Therefore in the later sections, we shall generally assume that every variable has a unique place of definition (i.e. the program is in SSA form).

To maintain the readability of the resulting code, it is desirable to keep the variable names as close to the original ones as possible. To this end, after optimizations are done, the Floey compiler reverses the variable renaming process and converts the SSA program back to original form, as long as it does not cause name clash. The reversal is accomplished by removing the subscripts.

For the purpose of readability of this document, we tend to avoid using SSA form in most examples but to manually name all variables differently, so no variable reassignment can occur.

4.2.2 Copy and constant propagation

There are two optimizing processes that can be applied in the normalization phase to eliminate redundant variable assignments: copy propagation and constant propagation [Callahan et al., 1986, 2004; Vanbroekhoven et al., 2003; Wegman and Zadeck, 1991]. If the expression on the right hand side of an assignment statement is a variable or a constant, the statement is called a copy assignment or a constant assignment respectively. In the copy/ constant propagation process, the variable x in a copy or constant assignment, x<−y, is substituted by y in the control expression that follows. As Floey code is first converted into SSA form prior to copy/ constant propagation, no variable re-assignment can happen to x or y (if it is a variable). This means the substitution will continue till the end of the control expression. x is then never used and the assignment can be thus removed.

```
1  fun foo(a:Real):Real
2  {
3      begin
4      case f of
5      [True.
6          x <- 3 / a.
7          y <- x.
8          return y - 7
9      |False.
10         x <- 9.
11         y <- x.
12         return y * a / x
13     ]
14 }
```

a

```
fun foo(a:Real):Real
{
    begin
    case f of
    [True.
        x <- 3 / a.
        return x - 7
    |False.return 9 * a / 9
    ]
}
```

b

Table 4.4: Copy and constant propagation

For example, the copy assignment on line 7 in table 4.4.a is removed, as it is a copy assignment. x is substituted for y in the statement: **return** y−7. The constant assignment on line 10 is removed, and x is substituted by integer 9. This substitution makes the assignment on line 11 another constant assignment and so can be removed.

4.3 Normalizing the entry structure

The entry structure normalization is a preparatory process for the reduction algorithm (section 5.2). This process moves entries to the highest possible scope level and puts them in topologically sorted order. At the same time, it collects uniquely used entries and removes the unreachable ones. The resulting Floey code has normalized and simpler entry structures, which allows a much simpler implementation of further normalizations or optimizations.

4.3.1 Entry renaming

In order to avoid name clashes on entries during code manipulation, especially during entry promotion (section 4.3.3), the Floey compiler first renames all entries uniquely. Similar to the approach used in section 4.2.1, the entry renaming is accomplished by adding uniquely numbered subscripts, after which the entry identifiers in exit statements are updated accordingly.

Entry renaming is applied in the beginning of entry normalization and is reversed after optimizations. The reversal is accomplished by removing the added subscripts and updating exit statements accordingly, as long as it does not cause entry name clashed on local scope level.

Again, for the readability of this document, we shall not show the entry renaming process in most examples by arranging to start with that each entry is uniquely named.

4.3.2 Local entries

An entry B is local to an entry or loop A if it can re-direct the execution back to A. Notice that since cyclic entry calls are prohibited in Floey, an entry cannot be local to another entry. Local entries thus are the entries in a loop which (indirectly) exit back to the loop header. For example, `en1` in table 4.5 is a local entry to loop `lp`, as it has an exit statement jumping to the header of loop `lp`. On the other hand, `en2` is a non-local entry to loop `lp`, as it does not directly or indirectly jump back to the loop header.

Identifying such local entries can be accomplished by the use of the dependency graph of that loop. In each function, control operation, entry or loop, a dependency graph of subsidiary entries, $\mathrm{x}_1,\ldots,\mathrm{x}_n$, can be generated. Each entry x_i is represented as a node in the graph. An arrow from x_i to x_j is present if and only if there is an exit statement in entry x_i jumping to entry x_j. The root of the dependency graph is the control expression that transitively calls these entries.

All local entries much directly or indirectly call the root. Therefore in the reversed dependency graph, local entries are those nodes that are reachable from the root. Thus, a traversal from the root in the reversed dependency graph identifies all local entries. The rest of the subsidiary entries are then non-local entries to that loop.

4.3.3 Entry promotion

Entry promotion recursively moves entries to the highest possible level and sorts them in the topological order. The resulting Floey code has a flattened and topologically sorted entry structure.

The entry promotion algorithm starts from the deepest level of the entry hierarchy and recursively works towards higher levels. On each level, the algorithm first identifies all

```
main: promote
fun promote(x:Int):Int
{
  begin
  loop lp(y<-x)
  {
    exit en1(y)
    entry en1(a:Int)
    {
      case a<100 of
      [True.exit lp(a*2)
      |False.exit en2(a,1)
      ]
    }
    entry en2(x,y:Int)
    {exit out1(x)}
  }
  entry out1(x:Int)
  {exit out3(x)}
  entry out2
  {exit out3(0)}
  entry out3(x:Int)
  {return x}
}
```

a

```
main: promote
fun promote(x:Int)
{
  begin
  loop lp(y<-x)
  {
    exit en1(y)
    entry en1(a:Int)
    {
      case a<100 of
      [True.exit l(a*2)
      |False.exit en2(a)
      ]
    }
  }
  entry out3(x:Int)
  {return x}
  entry out1(x:Int)
  {exit out3(x)}
  entry en2(x:Int)
  {exit out1(x)}
}
```

b

Table 4.5: Entry promotion

local subsidiary entries (see section 4.3.2) and moves the rest to one higher level. After that, entry promotion topologically sorts the remaining subsidiary entries on that level. When pulling an entry to the upper level, entry renaming (see section 4.3.1) ensures no entry name clash occurs.

As discussed in section 4.3.2, an entry cannot be local to another entry. As the result, after entry promotion, an entry will be either moved to the top level of a function or control operation, or it will be stuck in a loop to which it is local. There is therefore no nested entry structure left in the resulting code. For example, in table 4.5.a, entry en1 is a local entry to loop lp and cannot be promoted. While en2 is a non-local entry to lp and is moved up to the level of function promote.

As topologically sorted entry structure is desirable for the implementation of reduction algorithm (section 5.2). Given the dependency graph, the topological sorting algorithm based on depth first search [Corman et al., 1990] is used to sort all subsidiary entries in the reserve order that they are executed. During the topological sorting of entries, unreachable entries, which are the unreachable nodes from the root in the dependency graph, can also be eliminated. As the topological sorting algorithm is based on depth first search, by the end of the sorting, unreachable entries will not have been traversed and can be discarded. Table 4.5.b illustrates the effect of topological sorting: entry out3, out1 and en are sorted in the new order as en is executed before out1 and out3. Entry out2 is unreachable and is thus removed from the result. The process of eliminating unreachable entries is similar to the compiler optimization of eliminating unreachable code [Aho et al., 2007] (see section 6.2.4 for more discussion).

4.3.4 Entry collection

Entry collection collects the entries that are only called once by exit statements. The dependency graph discussed in section 4.3.2 can be used to recognize such entries: an entry with only one incoming arrow in the dependency graph is uniquely used. After a uniquely called entry en is identified, entry collection replaces the exit statement that exits to en with the control expression in en. As entry collection is applied after entry promotion is done, there are no subsidiary entries left in en.

After entry collection, all entries left in Floey programs are used in the way that they are designed to: they are the interface for joint points in control flow graph. The resulting Floey programs have fewer entries and larger control expressions, which not only reduces the overhead of entry calling but, more importantly, it improves the effect of optimizations on the resulting code as most Floey program optimizations (chapter 5) are designed to act locally on control expressions rather than globally across entries.

Entry collection is similar to the conventional inline expansion [Chang and Hwu, 1989], which replaces function calls by the code of that routine. By only collecting uniquely used entries, entry collection will not result in a code size explosion. For the simplicity, in the examples presented we generally skip entry collection.

4.3.5 Entry binding reduction

If a local binding is never used in a entry, it should be removed from the binding list. Binding reduction in an entry not only reduce the use of variables in that entry, it may also affect the variable liveness of the exits that jump to it, as the exit statements to that entry may also then be reduced. It is desirable to apply the entry binding reduction in the topological sorted order after entry promotion (section 4.3.3), as variable reduction in an entry can precipitate variable reduction in entries which exit to that entry.

Given a topologically sorted list of entries, in the reserve execution order, entry binding reduction is applied to each entry in the order. In each entry, two traversals are performed. In the first traversal, variable use information is collected and is used to identify and remove the unused bindings from entry interface. Such binding modifications are recorded in a list of entry identifiers paired with their updated bindings. In the second traversal, whenever an exit statement is encountered, for which the bindings of the entry it is jumping to has been changed, the parameters of the removed bindings in the entry will be removed from the exit statement. Entry binding reduction stops at the root of the dependency graph (the control expression in the local level).

For example in table 4.5.b, the local binding y is never used and thus removed. The exit statement in entry en1 that exits to en2 is thus updated.

4.4 Normalizing loops

Since loop expressions are special entry structures that allow cyclic calls, entry structure normalizations also applies to loop expressions. In this section, we discuss three loop normalizations that handle the special cyclic feature of loops expressions and further normalize them. Loop normalizations are applied after loop expressions are first entry normalized.

The scenarios discussed in the following sections can be present in the original source code but more often they are generated by optimization steps, such as repeat reduction (section 5.1).

4.4.1 Loop renaming

In the reduction algorithm (see section 5.2.6), independent loop headers are pulled up to the root of control expressions. In order to avoid loop name clashes, the first step of loop normalization is to rename each loop in a control expression uniquely. Similar algorithm to entry renaming (section 4.3.1) is applied to accomplish this task.

4.4.2 Infinite loop reduction

A loop expression with no return statement or exit statement jumping to outside the loop is an infinite loop, as the only way an iteration can end is by jumping outside the loop. Loop normalization traverses the loop body, through subsidiary entries, to identify infinite loops. Assuming Floey programs do not have side effect - which is the assumption in this thesis - an infinite loop is equivalent to non-termination. Such loop expressions can be reduced by removing all bindings and subsidiary entries. The control expression in the loop body is replaced by a simple exit statement jumping back to the loop header with no actual parameters.

This process not only reduce the code size of infinite loops, it also provides extra possibilities for other optimizations. The infinite loop in table 4.6.a for example, is normalized as shown in 4.6.b. As variable x on line 5 is never used after loop normalization, the assignment can be eliminated in idempotent reduction (section 5.2.4).

```
main: foo
fun foo(a:Int):Int
{
  begin
  x<-a*2.
  loop lp(y<-x) {
    case y<0 of
    [True.exit lp(y+1)
    |False.exit lp(y-1)
    ]
  }
}
```

a

```
1 main: foo
2 fun foo(a:Int):Int
3 {
4   begin
5   x<-a*2.
6   loop lp() {
7     exit lp
8   }
9 }
```

b

Table 4.6: Reducing infinite loops

4.4.3 Unlooping

A loop expression whose body has no exit statement jumping back to the loop header is not a looping structure and the loop header can be removed. Such loop expressions can be identified easily after entry promotion (section 4.3.3), when all the resulting subsidiary entries left in a loop expression are reachable local entries. In case there is any subsidiary entry left, the loop expression must be a looping structure. Otherwise, loop reduction traverses the control expression in loop body and identifies those exit statements that jump directly back to the loop header. If no such exit statement exists, the loop expression is not a looping structure and can be removed. The loop expression is replaced by the control expression in the loop body with loop bound variables substituted with their initialization expressions.

```
main: foo
fun foo(a:Int):Int
{
  begin
  x<-a*2.
  loop lp(y<-x) {
    case y<0 of
    [True.return -y
    |False.return y
    ]
  }
}
```

a

```
main: foo
fun foo(a:Int):Int
{
  begin
  x<-a*2.
  case x<0 of
  [True.return -x
  |False.return x
  ]
}
```

b

Table 4.7: Unlooping

The loop lp in table 4.7 doesn't have any exit statements jumping back to its header. It is replaced by the control expression in its loop body with y being substituted with x.

4.4.4 Loop binding reduction

Loop binding reduction is designed to remove all unused or unchanged bindings. A minimized list of binding allows loop reduction to employ more effective loop invariant code motion (discussed in section 5.3.2) to further optimize the loop.

Similarly to the ones in entries, unused bindings are those local bindings that are never used anywhere in a loop. These bindings can simply be removed from the binding list without modifying the loop body. Unchanged bindings are the local bindings that are used in a loop but their values are never altered during iterations. That is, in every exit statement jumping back to the loop header, the actual parameter of a local binding is its bound variable or initialization expression. Removing unchanged bindings from loops requires that the bound variables are substituted by their initialization expressions, as these variables can be used in the loop. A first traversal of the loop body, through subsidiary entries, can identify all unused and unchanged loop variables. If all local bindings are used or changed, the original loop expression is returned unchanged. Otherwise, unused and unchanged variables are removed accordingly. In the second traversal, exit statements that jump back to the loop header should be modified, as the loop interface has been changed. Actual parameters in respective positions will be removed. Such modifications may produce unused bindings in subsidiary entries, which requires entry binding reduction being applied again to these them.

```
loop lp(x'<-x,y'<-y) {
  case x'<0 of
  [True.
    exit return x'
  |False.
    exit lp(x',y'-1)
  ]
}
```

a

```
loop lp() {
  case x<0 of
  [True.return x
  |False.exit lp
  ]
}
```

b

Table 4.8: Removing unused or unchanged loop variables

In the example shown in table 4.8, loop variable x is unchanged and y is unused. Both bindings are removed with necessary variable substitution been applied. For another example on loop binding reduction, see table 5.15.

4.4.5 Loop amalgamation

In Floey, two directly nested loop expressions, that is their loop headers are next to each other, can be amalgamated if the following conditions are satisfied. Bound variables of the outer loop are assigned, one on one, to bound variables of the inner loop, and none of them are used in the loop body.

Original

```
fun foo(x,y:Bool):Int
{
  begin
  loop id1(t1<-y)
  {
    loop id2(t2<-t1)
    {
      case t2 of
      [True.return 1
      |False.
        case x of
        [True.
          exit id1(t2+1)
        |False.
          exit id2(t2*2)
        ]
      ]
    }
  }
}
```

a

Reduced

```
fun foo(x,y:Bool):Int
{
  begin
  loop id2(t2<-y)
  {
    case t2 of
    [True.return 1
    |False.
      case x of
      [True.
        exit id2(t2+1)
      |False.
        exit id2(t2*2)
      ]
    ]
  }
}
```

b

Table 4.9: Loop amalgamation

For example, here in table 4.9.a, t_1 is assigned to t_2 in the binding of loop id_2 and is never used in the loop body. Loop id_1 is then amalgamated with id_2. Program results in a simpler form.

Chapter 5

The Reduction Algorithm for Floey Programs

The purpose of this chapter is to describe the reduction algorithm for Floey programs. The algorithm has three main components: repeat reduction, reduction, and invariant code removal. The organization of these algorithms follow the organization of the decision tree optimization [Cockett and Herrera, 1990]. The new aspects concern how variables and loops are integrated into this algorithm.

Repeat reduction (section 5.1) looks for repeated calculations and, thus, not only includes common subexpression elimination but also the removal of conditional statements whose outcome is already known. The reduction algorithm (section 5.2) then more aggressively tries to remove code which unnecessarily lengthens some execution paths. Finally invariant code removal (section 5.3) tries to remove invariant code from the loops. Throughout this chapter, we assume Floey programs have already been converted into control normal form (see chapter 4).

5.1 Repeat reduction

A basic step in decision tree reduction [Cockett and Herrera, 1990] and code optimization is to eliminate common subexpressions and repeated decisions. For Floey the repeat reduction algorithm removes both the re-computation of expressions and of control operations. Thus common sub-expression and branches which will not be executed are removed at this stage.

The repeat reduction algorithm for Floey programs will be explained progressively in the following sections. Section 5.1.1 gives an overview of repeat reduction on non-looping control expression constructs, the details of which are discussed from section 5.1.2 to section 5.1.5. Finally repeat reduction for loop expressions is discussed in section 5.1.6.

5.1.1 Overview of repeat reduction algorithm

```
1 repeat_reduce : List(Choice) × Cexp -> Cexp
2 repeat_reduce chs (exit b($\tilde{t}$)) = exit b($\tilde{t}$)
3 repeat_reduce chs (q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$) =
4   if (q($\tilde{t}$),b$_j$($\tilde{v}$)) ∈ chs then repeat_reduce chs cexp$_j$[$\tilde{v}/\tilde{v_j}$]
5   else q($\tilde{t}$){b$_i$($\tilde{v_i}$).(repeat_reduce (q($\tilde{t}$),b$_i$($\tilde{v_i}$)):chs cexp$_i$)}$_{i\in I}$
```

Figure 5.1: Repeat reduction algorithm

The repeat reduction algorithm on non-looping programs is given in figure 5.1. Repeat reduction on loop expression is similar but involves local entries, which is discussed in section 5.1.6. In figure 5.1, **exit** b($\tilde{t}$) stands for an exit statement or a return statement, as they are treated the same in the repeat reduction. On line 3, q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ stands for a decision in Floey (control statements, case statements and assignments). Here q($\tilde{t}$) is the control operation in that decision (note that case statements have the special "case" control operations, while assignments are treated internally as unary case statements, see section 2.6), which is followed by the branches. In each branch i, cexp$_i$ is the control expression that follows the branch binding, b$_i$($\tilde{v_i}$).

The repeat reduction algorithm produces a repeat reduced control expression from a control expression and an execution history. The execution history is held as a list of choices. Each **choice** is a control operation call paired up with a branch binding. A **control operation call** is a control operation applied to its arguments. The arguments are expressions in general, however, in control normalized Floey code, these arguments can only be variables or constants (see expression expansion in section 4.1.2). Every decision in Floey programs starts with a control operation call, based on the result of which the execution is directed to one of the branches that follow. The list of choices, thus, holds the history of control operations and their branch bindings, which indicate the path the execution has chosen through these control operations. Assuming control operations and functions in Floey are all pure (with no side-effect), repeated uses of a control operation call should return the same branch binding, which will be selected by repeat reduction. Program efficiency can gain from avoiding re-computations of such repetitions. Program size can also be reduced as the branches that cannot be reached are removed.

Repeat reduction algorithm does a recursive traversal of control expressions, during which the execution history is passed and updated. Whenever a decision with a repeated control operation call is encountered, the branch with the same branch binding as the one found in the execution history is selected.

Following sections discuss the implementations of the algorithm on different Floey control expression constructs. Examples are also provided to demonstrate the effects of repeat reduction.

5.1.2 Exit and return statements

As Floey programs have been control normalized, the expressions in exit or return statements must either be constants or variables. There can be no repetition of computation in these statements as there is no computation in them. Exit and return statements are left unchanged by repeat reduction (line 2 in figure 5.1).

5.1.3 Control statements

When a control statement is encountered, the repeat reduction searches for its control operation call, q($\tilde{t}$), in the execution history. In case a choice, (q($\tilde{t}$),b$_j$($\tilde{v}$)), is found, the control statement is then replaced with cexp$_j[\tilde{v}/\tilde{v_j}]$ (line 4 in figure 5.1). That is the control expression which follows b$_j$($\tilde{v_j}$) with the variables $\tilde{v_j}$ substituted by $\tilde{v}$. The traversal then continue on to the resulting control expression.

Otherwise, if the control operation is not found in the history, the traversal moves on to each control expression in the branches of that control statement, with a new choice (q($\tilde{t}$),b$_i$($\tilde{v_i}$)) added for the traversal on branch i.

```
1  main: repeat
2  def foo(x:Bool)[a(Int),b(Int)]
3  {
4      begin
5      case x of
6      [True.exit a(1)
7      |False.exit b(0)]
8  }
9  fun repeat(x:Bool):Int
10 {
11     begin
12     foo(x) of
13     [a(y).
14         foo(x) of
15         [a(z).return z
16         |b(z). return 0
17         ]
18     |b(y).
19         foo(x) of
20         [a(z).return 0
21         |b(z).return z
22         ]
23     ]
24 }
```

a

```
main: repeat
def foo(x:Bool)[a(Int),b(Int)]
{
    begin
    case x of
    [True.exit a(1)
    |False.exit b(0)]
}
fun repeat(x:Bool):Int
{
    begin
    foo(x) of
    [a(y).return y
    |b(y).return y
  ]
}
```

b

Table 5.1: Repeat reduction on control statements

The example in table 5.1 demonstrates the effect of repeat reduction on control statements. On line 14 and 19 in table 5.1.a, the result of foo(x) can be determined by its previous occurrence. As foo(x) on line 14 is in the branch of a(y), the choice (foo(x), a(y)) must be found in the history. Repeat reduction then selects the branch a(z), substitutes y for z. A similar repetition appears on line 19.

5.1.4 Case statements

As the control expression is in control normal form, when a case statement is encountered, its argument is either a constant or a variable. Whenever an argument in a case statement is a constant constructor or a variable which can be found in the history, repeat reduction is able to predict the branch that the execution will take. In the former case, the branch that starts with the same constructor is selected. In the latter case, repeat reduction algorithm searches in the history for the assigned value of the variable, which should be in the special form: (**case**(exp),#Assign(v)). Here exp is the expression that the variable v is assigned to. #Assign is the special constructor used in assignments (see section 2.6.5). If such choice is found and the expression exp is a constructor, the branch with that constructor will be selected by repeat reduction with necessary variable substitutions. The traversal then moves on to the resulting control expression.

Otherwise, if none of the above conditions is satisfied, the traversal continues on to each control expression in the branches of that case statement. In the traversal on each branch i, a new choice, (**case**($b_i(\tilde{v}_i)$),#Assign(v)), is added to the execution history.

The example in table 5.2 demonstrates the effect of repeat reduction on case statements. When the traversal comes to the case statement on line 8 in 5.2.a, a choice (**case**(SS(a)),#Assign(x)) is found in the history. The branch that starts with SS(b) is then

```
1  main: repeat
2  data SF a = SS(a)
3           | FF
4  fun repeat(x:SF Int):Int {
5      begin
6      case x of
7      [SS(a).
8          case x of
9          [SS(b).return b
10         |FF.return 0
11         ]
12     |FF.
13         case SF(1) of
14         [SS(d).return d
15         |FF.return 0
16         ]
17     ]
18 }
```

a

```
main: repeat
data SF a = SS(a)
         | FF
fun repeat(x:SF Int):Int {
    begin
    case x of
    [SS(a).return a
    |FF.return 1
    ]
}
```

b

Table 5.2: Repeat reduction on case statements

selected, b is substituted by a in **return** b. The expression SS(1) on line 13 is a constant constructor. Repeat reduction selects the branch with SS(d) and substitutes d with 1 in **return** d.

5.1.5 Assignment statements

As the result of expression expansion (see section 4.1.2), all calculations on expressions are moved into assignment statements. It is, thus, on the assignments that common subexpressions are detected.

An assignment, internally, is stored as a special case statement with unary branch: **case**(exp){#Assign(v).cexp}. When an assignment is encountered, the repeat reduction looks in the history for a choice in the form: (**case**(exp),#Assign(t)). The existence of such a choice in the history means that the expression exp is already assigned to another variable

Original

```
fun comsub(a:Int):Int
{
  begin
  x<-5*a.
  x<-x+1.
  y<-5*a.
  z<-6*(y+2).
  return (y+2)/4
}
```

a

Control normal form

```
1  fun comsub(a:Int):Int
2  {
3    begin
4    x1<-5*a.
5    x2<-x1+1.
6    y3<-5*a.
7    t1<-y3+2.
8    z4<-6*t1.
9    t3<-y3+2.
10   t4<-t3/4.
11   return t4
12 }
```

b

Repeat reduction

```
fun comsub(a:Int):Int
{
  begin
  x1<-5*a.
  x2<-x1+1.
  t1<-x1+2.
  z4<-6*t1.
  t4<-t1/4.
  return t4
}
```

c

Table 5.3: Common sub-expression elimination

v. To avoid re-computation, the assignment is replaced by cexp[t/v], the control expression that follows with t substituted for v. If the expression in an assignment is not found the history, repeat reduction adds a new choice into the history: (**case**(exp),#Assign(v)) and moves on to control expression cexp. In table 5.3.b, since the expression 5 $*$ 8 on line 5 is already computed, x_1 is substituted for y_3 in the control expression that follows. Due to the application of expression expansion, the re-computation common sub-expressions y_3+2 on line 6 and line 8 is recognized and avoided. The result is shown in 5.3.c.

Example in table 5.4 demonstrates the effect of repeat reduction in a control expression with both assignments and case statements. When processing the case statement on line 7 in 5.4.a, a choice (SS(a),#Assign(x)) is found in the history and used to select the branch with binding SS(b). The expression on line 13 results to SS(a) after variable substitution and is later replaced by x. When the traversal comes to the assignment on line 11, a choice (SS(c),y) is found in the history, which means the assignment is removed with z being substituted to y in **return** z.

```
main: repeat
data SF a = SS(a)
          | FF
fun repeat(a:Int,y:SF Int):List {
  begin
  x<-SS(a).
  case x of
  [SS(b).
    case y of
    [SS(c).
      z<-SS(c).
      return z
    |FF.return SS(b)
    ]
  |FF.return FF
  ]
}
```

a

```
main: repeat
data SF a = SS(a)
          | FF
fun repeat(a:Int,y:SF Int):List {
  begin
  x<-SS(a).
  case y of
  [SS(c).
    return y
  |FF.
    return x
  ]
}
```

b

Table 5.4: Repeat reduction on case and assignment statements

5.1.6 Loop expressions

```
repeat_reduce : List(Choice) × Cexp -> Cexp
repeat_reduce chs (loop id (bindings) {body})
  = loop id (bindings) {repeat_body chs body}

repeat_body : List(Choice) × (Cexp, List(Entry)) -> (Cexp, List(Entry))
repeat_body chs (cexp, {entry id_i (bindings_i) {body_i}}_{i∈I})
  = (repeat_reduce chs cexp,
    {entry id_i (bindings_i) {(repeat_body [] body_i)}}_{i∈I})
```

Figure 5.2: Repeat reduction algorithm on loop expressions

When a loop expression is encountered, repeat reduction goes into the loop body and repeat reduces it. In figure 5.2, repeat_body is used to accomplish that.

repeat_body applies repeat reduction on the control expression and on the subsidiary entries in the loop body. The execution history is passed to repeat_reduce on the control expression, while repeat_body on entries receive empty list as input. That is because history information cannot be simply passed into entries, because for an entry, there can be more than one exit statement that jumps to it.

5.2 The reduction algorithm

The reduction algorithm in its original form is a decision tree optimization technique [Cockett and Herrera, 1990], which here is applied to achieve compiler optimizations. Each case statement or control statement in Floey is a decision with multiple choices, while assignment statements can be seen as decisions with only one choice. It is therefore natural to apply decision tree manipulations to Floey programs, although the manipulations are now constrained by variable bindings and must be extended to handle loops.

The organization of this section is as follows: section 5.2.1 gives an overview of the reduction algorithm, the main components of which are discussed through section 5.2.2 to section 5.2.5. Finally, section 5.2.6 presents the loop reduction algorithm, an extended version of the reduction algorithm that is designed to further reduce loops.

5.2.1 Overview of reduction algorithm

In essence, the reduction algorithm optimize a control expression by removing unnecessary decisions. As a result, program performance can be improved as some execution paths may be reduced. The algorithm can be broken down to four major components: identifying semi-essentials in a control expression (section 5.2.2), pulling up semi-essentials (section 5.2.3), idempotent reduction (section 5.2.4), and elimination factoring (section 5.2.5). It recursively pushes decisions in control expression down, by identifying and pulling up semi-essential decisions. When a decision is pushed down until it is immediately above leaves (exit or return statements), it has the opportunity to be idempotent reduced. This reduces the length of an execution path and, thus, gives potential improvement in efficiency. If any statement is successfully idempotent reduced after the "pulling-up" of a semi-essential, elimination factoring will keep the resulting control expression. Otherwise, the control expression rolls back to the previous form before that semi-essential was pulled up.

```
1  reduce : Cexp -> Cexp
2  reduce cexp = reduction (repeat_reduce cexp)
3
4  reduction : Cexp -> Cexp
5  reduction (exit b($\tilde{t}$)) = exit b($\tilde{t}$)
6  reduction (q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$) = if v then t
7    else q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp'$_i$}$_{i\in I}$
8    where cexp'$_i$ = reduction cexp$_i$
9           (v,t) = elimination_factor q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp'$_i$}$_{i\in I}$
10 reduction (loop id (bindings) {body})
11   = loop id (bindings) {reduction_body body}
12
13 reduction_body : (Cexp, List(Entry)) -> (Cexp, List(Entry))
14 reduction_body (cexp, {entry id$_i$ (bindings$_i$) {body$_i$}}$_{i\in I}$)
15   = (reduction cexp, {entry id$_i$ (bindings$_i$) {reduction_body body$_i$}}$_{i\in I}$)
```

Figure 5.3: The reduction algorithm

The pseudo code of the reduction algorithm is given in figure 5.3. As a preliminary step of the reduction algorithm, repeat reduction is first applied to the control expression. The resulting control expression is then sent to the procedure reduction, which produces a reduced control expression. The algorithm does a traversal on the given control expression. **exit** b($\tilde{t}$) on line 5 stands for leaf statements of the control expression, which are left unchanged. When a loop expression is encountered (line 10), the loop header is unchanged. The traversal moves on to the loop body and reduces it. Whenever a decision is encountered (line 6), each control expression in its branches is first reduced (line 8). The elimination factor algorithm is then used trying to reduce the decision as a whole (line 9). If it succeeded, the reduced decision is returned. Otherwise, the decision with reduced branches is returned.

The following subsections will explain the four major components of the reduction algorithm in details.

5.2.2 Identify semi-essentials

Identifying semi-essentials (see section 3.2.3 for the definition) in a control expression is an important step in the reduction algorithm, so that these semi-essentials can be pulled up to the root (section 5.2.3), which may allow potential idempotent reductions (section 5.2.4).

Semi-essentials in decision trees are the decisions that occur in every path from the root to the leaves. When identifying semi-essentials in a control expression, however, one needs to ensure that the semi-essentials do not use any branch variables bound by other decisions.

```
get_semi_essentials : List(String) $\times$ Cexp -> List(Struct)
get_semi_essentials vars (**exit** b($\tilde{t}$)) = $\emptyset$
get_semi_essentials vars q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$
    = defn $\bigcup$ ($\bigcap${get_semi_essentials (vars ++ $\tilde{v_i}$) cexp$_i$}$_{i\in I}$)
  **where** defn = **if** $\tilde{t}\bigcap$vars==$\emptyset$ **then** q($\tilde{t}$){b$_i$($\tilde{v_i}$)}$_{i\in I}$ **else** $\emptyset$
get_semi_essentials vars (**loop** id (local$_i$ $\leftarrow$free$_i$)$_{i\in I}$ {cexp entries})
  = get_semi_essentials (vars ++ (local$_i$)$_{i\in I}$) cexp
```

Figure 5.4: Identify semi-essentials from a control expression

The algorithm of identifying semi-essentials in a control expression is given in figure 5.2.2. The procedure get_semi_essentials returns a list of structures of semi-essentials from a list of strings together with a control expression. A **structure** is a control operation call associated with its branch constructors. The returning list of structures can then be used to pull semi-essentials up and reconstruct the control expression (see figure 5.7 and figure 5.3.1). The algorithm does a depth first traversal on a control expression, during which the list of strings is used for collecting defined variables.

On line 2, **exit** b($\tilde{t}$) stands for the leaf statements. As none of them can be a semi-essential, an empty list is return.

When a decision is encountered (line 3), the algorithm first checks the variables used in the decision. If it does not use any variable that is defined earlier in the traversal, its structure is added into the list. For a multiple decision, the semi-essentials beneath it is the intersection of semi-essentials in each branch.

Original

```
main: pull
data TF = T(Int)
    | F(Int)
fun pull(x,y:TF):Int
{
  begin
  case x of
  [T(a).
    case y of
    [T(a).return a
    |F(a).return a
    ]
  |F(a).
    case y of
    [T(a).return a
    |F(a).return a
    ]
  ]
}
```

a

Control normal form

```
main: pull
data TF = T(Int)
    | F(Int)
fun pull(x,y:TF):Int
{
  begin
  case x of
  [T(a1).
    case y of
    [T(a2).return a2
    |F(a3).return a3
    ]
  |F(a4).
    case y of
    [T(a5).return a5
    |F(a6).return a6
    ]
  ]
}
```

b

Table 5.5: Semi-essentials with different branch variables

It is worth noting that in the intersection calculation, different branch variables in the structures do not affect their equivalence. Consider the example in table 5.5. After control normalization, all variable are renamed differently in 5.5.b. Despite of having different branch variables, structures (y{T(a_2),F(a_3)}) and (y{T(a_5),F(a_6)}) are considered to be equal in the intersection and the case statements on line 9 and 14 are considered to be semi-essentials.

The looping structure in Floey requires careful consideration. When a loop expression in encountered (line 6), the traversal moves into the loop body looking for semi-essentials. Although loop headers can also be semi-essential to a control expression, it may not be desirable to pull loop headers to the root of that control expression. As such code manipulation may result in including more code in loops, critical parts of any program, and may in general affect the program performance. We therefore avoid identifying loop headers as semi-essentials. However, in section 5.2.6, a more aggressive approach called loop reduction is presented. It includes pulling loop headers up for potential optimizations and thus requires extra effort (see section 5.3) to clean up invariant code introduced into the loops.

5.2.3 Pulling up decisions

Once all semi-essentials in a control expression are identified, given by the consequence shown in section 3.2.3, they can be pulled up to the root of the control expression without changing the output of the program, This "pulling-up" manipulation is used both in the reduction algorithm (section 5.2.1) and in invariant code removal (section 5.3).

Figure 5.5 presents the algorithm of pulling up a semi-essential in a control expression. Given a semi-essential in a control expression, the pull_up procedure reconstructs a new control expression, in which that semi-essential is moved to the root. The algorithm consists of two basic steps: copying the semi-essential to the root and then removing the

```
1  pull_up : Struct × Cexp -> Cexp
2  pull_up q($\tilde{t}$){b$_i$($\tilde{v_i}$)}$_{i\in I}$ cexp
3    = q($\tilde{t}$){b$_i$($\tilde{v_i}$).(branch_reduce (q($\tilde{t}$),b$_i$($\tilde{v_i}$)) cexp)}$_{i\in I}$
4
5  branch_reduce : Choice × Cexp -> Cexp
6  branch_reduce (q($\tilde{t}$),b($\tilde{v}$)) (exit b'($\tilde{v'}$)) = exit b'($\tilde{v'}$)
7  branch_reduce (q($\tilde{t}$),b($\tilde{v}$)) q'($\tilde{t'}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ =
8    if q($\tilde{t}$)==q'($\tilde{t'}$) and b$_k$==b then branch_reduce (q($\tilde{t}$),b($\tilde{v}$)) cexp$_k$[$\tilde{v}/\tilde{v_k}$]
9    else q'($\tilde{t'}$){b$_i$($\tilde{v_i}$).(branch_reduce (q($\tilde{t}$),b($\tilde{v}$)) cexp$_i$)}$_{i\in I}$
10 branch_reduce (q($\tilde{t}$),b($\tilde{v}$)) (loop id (bindings) {cexp entries})
11   = loop id (bindings) {(branch_reduce (q($\tilde{t}$),b($\tilde{v}$)) cexp) entries}
```

Figure 5.5: Pulling up semi-essentials

repetition introduced in the control expression.

The first step is accomplished by attaching the original control expression to each of the branches of the given structure (see line 3). Shown in table 5.6.a is the result of such manipulation from the program in table 5.5.b. The structure (**case** y,{T(a_2),F(a_3)}) has been pulled up to the root.

In the second step, a procedure called branch reduce is designed to remove all the repetitions introduced by the previous step. As shown in table 5.6.a, case statements on line 11, 16, 24 and 29 are repeated decisions. Although an application of repeat reduction can eliminate such repetitions, it is unnecessary. Since the input control expression is already repeat reduced, the only repetitions introduced by the pulling up process are the repetitions of the semi-essential being pulled up. Such repetitions is eliminated by branch reduction, a simplified version of repeat reduction, which only aims at reducing repeated computation of one specific decision. As shown in table 5.6.b, repeated branches on case statements have been eliminated. Necessary variable substitutions are also done in branch reduction to handle the issue of different branch variables in semi-essentials. In table 5.6.b, a_5 is substituted for a_2, and a_6 is substituted for a_3 in the return statements.

Pulling up

```
main: pull
data TF = T(Int)
     | F(Int)
fun pull(x,y:TF):Int
{
  begin
  case y of
  [T(a2).
    case x of
    [T(a1).
      case y of
      [T(a2).return a2
      |F(a3).return a3
      ]
    |F(a4).
      case y of
      [T(a5).return a5
      |F(a6).return a6
      ]
    ]
  |F(a3).
    case x of
    [T(a1).
      case y of
      [T(a2).return a2
      |F(a3).return a3
      ]
    |F(a4).
      case y of
      [T(a5).return a5
      |F(a6).return a6
      ]
    ]
  ]
}
```

a

Branch reduction

```
data TF = T(Int)
     | F(Int)
fun semi(x,y:TF):Int
{
  begin
  case y of
  [T(a2).
    case x of
    [T(a1).return a2
    |F(a4).return a2
    ]
  |F(a3).
    case x of
    [T(a1).return a3
    |F(a4).return a3
      ]
    ]
  ]
}
```

b

Table 5.6: Pulling up semi-essentials

5.2.4 Idempotent reduction

Given by the idempotence equality stated in section 3.2.2, a decision should be removed if all the branches that follow it are the same and never use the branch variables produced by the decision, because no matter which branch is executed the control expression outputs the same result. This optimization is called **idempotent reduction**, which is the basic step in the reduction algorithm where improvements are achieved. Because the reduction algorithm repeatedly pushes decisions down by pulling up semi-essentials, idempotent only applies when a decision is pushed immediately above leaves.

```
1 idem_reduce : Cexp -> (Bool,Cexp)
2 idem_reduce (exit b($\tilde{t}$)) = (False, exit b($\tilde{t}$))
3 idem_reduce q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ =
4   if $\forall j,k \in I$ cexp$_j$==cexp$_k$==(exit b($\tilde{v}$)) and $\tilde{v}\bigcap\tilde{v_j} == \tilde{v}\bigcap\tilde{v_k} == \emptyset$
5     then (True, cexp$_1$)
6     else (False, q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$)
```

Figure 5.6: Idempotent reduction algorithm

The algorithm of idempotent reduction is given in figure 5.6. idem_reduce returns the resulting control expression together with a boolean value, which indicates the success of idempotent reduction.

On line 2 and line 4, **exit** b($\tilde{v}$) stands for leaf statements, as they are treated the same in the idempotent reduction. All leaf statements (exit and return statements) are left unchanged (line 2).

When a decision is encountered (line 3), it will be idempotent reduced only if all the following conditions satisfy: the control expressions that follow it are the same leaves and do not use any of the branch variables produced by the decision. Table 5.7 gives an example of an idempotent reduction on a case statement. As case statement is followed

```
main: idem
fun idem(x:Int):Int
{
  begin
  case x>0 of
  [True.return x
  |False.return x
  ]
}
```

a

```
main: idem
fun idem(x:Int):Int
{
  begin
  return x
}
```

b

Table 5.7: Idempotent reduction on multiple decisions

by the same return statements, it is removed. Table 5.8 illustrates the effect of idempotent reduction on unary decisions. The assignment is first removed, which makes the control statement idempotent and is thus removed.

5.2.5 Elimination factoring

Elimination factoring is the core of the reduction algorithm. The idea of elimination factoring is to pull semi-essentials up to the root of control expression and detect successful idempotent reductions. Given any successful idempotent reduction, the resulting control expression is optimized and thus should be kept. The algorithm is given in figure 5.7.

Elimination factoring takes in a control expression and produces a control expression together with a boolean value, which indicates whether an idempotent reduction has been possible. In the reduction algorithm (figure 5.3), this returned boolean value is checked, and if it is true, the returned control expression will be kept in the reduction algorithm. Otherwise the original control expression is returned.

On leaf statements, no idempotent reduction is possible and the original control expression is returned with a false boolean value.

```
main: idem
def foo()[a(Int)]
{
  begin
  return a(1)
}
fun idem():Int
{
  begin
  foo() of
  [a(v).
    x<-3*5.
    return 1
  ]
}
```

a

```
main: idem
def foo()[a(Int)]
{
  begin
  return a(1)
}
fun idem():Int
{
  begin
  return 1
}
```

b

Table 5.8: Idempotent reduction on unary decisions

Given a decision applied to leaf values, if idempotent reduction applies the resulting control expression is returned with a true boolean value. Otherwise, the decision cannot be idempotent reduced. However, it may still be reduced if it contains semi-essentials, that is the get_semi_essentials procedure (section 5.2.2) returns a non-empty list of structures.

By applying pull_up procedure (section 5.2.3), the first semi-essential is pulled to the root of the newly generated control expression. The selection of which semi-essential to be pulled up in this step does not matter. Other semi-essentials will be pulled up by the recursions of elimination factoring (see line 10). The order of these semi-essentials being pulled up makes no difference to the final optimizing effect [Cockett and Herrera, 1990].

In the new control expression, elimination factoring recursively applies to each branch. If any f_i is successfully reduced, the new control expression is returned with the branches being replaced by the elimination factored arguments.

The example in table 5.9 illustrates the effect of the reduction algorithm on two assign-

```
elimination_factor : Cexp -> (Bool,Cexp)
elimination_factor (exit b($\tilde{v}$)) = (False, exit b($\tilde{v}$))
elimination_factor q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ =
  case idem_reduce q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ of
    (True, cexp) → (True, cexp)
    (False, cexp) → case get_semi_essentials [] cexp of
      [] → (False, cexp)
      (q'($\tilde{t'}$){b$_i$($\tilde{v_i}$)}$_{i\in I}$):rest → do
        q'($\tilde{t'}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ ← pull_up (q'($\tilde{t'}$){b$_i$($\tilde{v_i}$)}$_{i\in I}$) cexp
        {(f$_i$,cexp'$_i$)}$_{i\in I}$ ← {elimination_factor cexp$_i$}$_{i\in I}$
        if any {f$_i$}$_{i\in I}$ then (True, q'($\tilde{t'}$){b$_i$($\tilde{v_i}$).cexp'$_i$}$_{i\in I}$)
        else (False, cexp)
```

Figure 5.7: The elimination factor algorithm

ments. During the recursive traversal of the reduction algorithm, the assignment, z<−x∗2 is first pushed down. As the variable z is used in both return statements, no idempotent reduction can be applied and the control expression rolls back to the original form. The assignment, y<−x+1 is then pushed further down. When it reaches the return statements (5.9.b), idempotent reduction applies and removes it from one branch, where the variable y is not used. The reduced control expression is returned (5.9.c).

Table 5.10 gives another example of the reduction algorithm. The original program in 5.10.a cannot be optimized by the reduction algorithm. However, after variable normalization (copy propagation in particular), the re-computation of expression x+1 is recognized and removed by repeat reduction (5.10.b). The result is then further optimized by the idempotent reduction (5.10.c). This example shows how normalization and repeat reduction work as the preliminary steps for the reduction algorithm.

```
main: ex1
fun ex1(x:Int):Int {
  begin
  y<-x+1.
  z<-x*2.
  case x<0 of
  [True.
    return y*z
  |False.
    return z
  ]
}
```

a

```
main: ex1
fun ex1(x:Int):Int {
  begin
  z<-x*2.
  case x<0 of
  [True.
    y<-x+1.
    return y*z
  |False.
    y<-x+1.
    return z
  ]
}
```

b

```
main: ex1
fun ex1(x:Int):Int {
  begin
  z<-x*2.
  case x<0 of
  [True.
    y<-x+1.
    return y*z
  |False.
    return z
  ]
}
```

c

Table 5.9: The reduction algorithm on assignments

Original

```
main: ex2
fun ex2(x:Int):Int {
  begin
  y<-x.
  z<-y+1.
  case z<0 of
  [True.
    w<-x+1.
    return w
  |False.
    return z
  ]
}
```

a

Normalized and repeat reduced

```
main: ex2
fun ex2(x:Int):Int {
  begin
  z<-x+1.
  case z<0 of
  [True.
    return z
  |False.
    return z
  ]
}
```

b

Reduced

```
main: ex2
fun ex2(x:Int):Int {
  begin
  z<-x+1.
  return z
}
```

c

Table 5.10: Another example of the reduction algorithm

5.2.6 Loop reduction

In general, pushing code into loops is not desirable as larger loop bodies may reduce program efficiency. However, in the reduction algorithm, pushing decisions into loops (by pulling up loop headers) can potentially make gains, as potentially more reductions can be found on larger loop bodies.

Original

```
fun foo(x:Int):Int
{
  begin
  y<-x*2.
  loop id(t<-x)
  {
    case t<0 of
    [True.
      case x>0 of
      [True.return x
      |False.return y
      ]
    |False.
      exit id(t+1)
    ]
  }
}
```

a

Pulling up loop header

```
fun foo(x:Int):Int
{
  begin
  loop id(t<-x)
  {
    y<-x*2.
    case t<0 of
    [True.
      case x>0 of
      [True.return x
      |False.return y
      ]
    |False.
      exit id(t+1)
    ]
  }
}
```

b

Reduced

```
fun foo(x:Int):Int
{
  begin
  loop id(t<-x)
  {
    case t<0 of
    [True.
      case x>0 of
      [True.return x
      |False.
        y<-x*2.
        return y
      ]
    |False.
      exit id(t+1)
    ]
  }
}
```

c

Table 5.11: Pushing unary decisions into loops

The example in table 5.11 shows how pushing an unary decision into loop enable for further optimizations. The assignment y<−x∗2 is pushed into the loop expression and becomes the root of the loop body in 5.11.b. After applying the reduction algorithm on the body, the assignment is moved to the branch above the only use of variable y. The program performance may improve as y<−x∗2 is not involved in the iterations and is only executed if y is used, while in the original code the assignment is always executed once.

Original

```
fun foo(x,y:Bool):Int
{
  begin
  case x of
  [True.
    loop id1(t1<-y)
    {
      case t1 of
      [True.return 1
      |False.
        exit id1(t1+1)
      ]
    }
  |False.
    loop id2(t2<-y)
    {
      case t2 of
      [True.return 1
      |False.
        exit id2(t2*2)
      ]
    }
  ]
}
```

a

Pulling up loop headers

```
fun foo(x,y:Bool):Int
{
  begin
  loop id1(t1<-y)
  {
    loop id2(t2<-y)
    {
      case x of
      [True.
        case t1 of
        [True.return 1
        |False.
          exit id1(t1+1)
        ]
      |False.
        case t2 of
        [True.return 1
        |False.
          exit id2(t2*2)
        ]
      ]
    }
  }
}
```

b

Reduced

```
fun foo(x,y:Bool):Int
{
  begin
  loop id1(t1<-y)
  {
    loop id2(t1<-t1)
    {
      case t1 of
      [True.return 1
      |False.
        case x of
        [True.
          exit id1(t1+1)
        |False.
          exit id2(t1*2)
        ]
      ]
    }
  }
}
```

c

Table 5.12: Pushing multiple decisions into loops

Table 5.12 gives another example, in which program performance benefits from pushing a multiple decision into loops. The case statement on the expression x is pushed into two independent loops below it (see 5.12.b). The enlarged loop body is further optimized in 5.12.c, after the variable substitution of t_1 to t_2. The case statement disappears on one branch due to idempotent reduction (section 5.2.4).

It is worth noting that although pushing decision into loops may allow more thorough optimizations, it has one drawback: the result may have more invariant decisions in loops. As shown in table 5.13, the assignment, y<−x+2, is pushed into the loop and disappears in

Original

```
fun foo(x:Int):Int
{
  begin
  y<-x*2.
  loop id(t<-x)
  {
    case t<0 of
    [True.
      case x>0 of
      [True.return x
      |False.
        exit id(y+t)
      ]
    |False.
      exit id(t+1)
    ]
  }
}
```

a

Pulling up loop header

```
fun foo(x:Int):Int
{
  begin
  loop id(t<-x)
  {
    y<-x*2.
    case t<0 of
    [True.
      case x>0 of
      [True.return x
      |False.
        exit id(y+t)
      ]
    |False.
      exit id(t+1)
    ]
  }
}
```

b

Reduced

```
fun foo(x:Int):Int
{
  begin
  loop id(t<-x)
  {
    case t<0 of
    [True.
      case x>0 of
      [True.return x
      |False.
        y<-x*2.
        exit id(y+t)
      ]
    |False.
      exit id(t+1)
    ]
  }
}
```

c

Table 5.13: Invariant decision left in loop

one branch. It is, however, left in another branch and becomes loop invariant, which may adversely affect the program performance. This problem is resolved in a subsequent step by removing all loop invariant code (see section 5.3.3). Therefore, in the end, pushing decisions into loops, when some gain can be made will be beneficial.

In the reduction algorithm as discussed so far, these optimizations are not secured as decisions are never pushed through loop headers. In this section, we present a more aggressive version of the reduction algorithm, called the loop reduction algorithm, which does secure these gains. The loop reduction algorithm first uses all the techniques in the reduction algorithm to reduce a control expression. On top of that, it pushes decisions into loops in order to make further optimization, just as in the examples shown in table 5.11 and 5.12. However, there is an extra subtlety involved: further optimizing the enlarged loop bodies may often require identifying the decisions which we call potential semi-essentials. **Potential semi-essentials** are the decisions that become semi-essentials, when certain loop variables are identified. For example in table 5.12.b, after the identification of t_1 and t_2, the case statements on t_1 in the loop become semi-essentials and can be pulled up to the root.

```
elimination_factor' : Cexp -> (Bool,Cexp)
elimination_factor' cexp = (v,cexp')
  where (bs,v,cexp') = loop_reduce [] cexp
```

Figure 5.8: The modified elimination factor algorithm

In the loop reduction algorithm, the algorithms discussed from section 5.2.1 to section 5.2.4 can be reused, as all the adjustments made are in the elimination factoring. The modified elimination factoring is present in figure 5.8. Here elimination_factor' keeps the same interface as elimination_factor (see figure 5.7), so that it can be used to replace elimination factoring in the reduction algorithm (see figure 5.3). The loop reduction procedure, loop_reduce, is used to optimize the input control expression. The control expression that results from loop reduction will be returned with a boolean value, which indicates loop reduction has actually occured. If a gain is made, the resulting control expression will be kept, otherwise the original control expression is returned unchanged.

The loop reduction algorithm is present in figure 5.9. The function takes in a list of bindings, with a control expression. Both List (Bind) in the input and output are used to hold the loop bindings. If there is any gain made by the loop reduction, the resulting control expression will be returned with a true boolean value. Otherwise, the unchanged control expression is returned with a false value.

The loop reduction algorithm from line 1 to 11 is similar to elimination factoring (see

```
1  loop_reduce : List(Bind) × Cexp -> (List(Bind),Bool,Cexp)
2  loop_reduce bs (exit b($\tilde{v}$)) = (bs,False,exit b($\tilde{v}$))
3  loop_reduce bs$_0$ q($\tilde{t}$){b$_k$($\tilde{v_k}$).cexp$_k$}$_{k\in K}$ =
4    case idem_reduce q($\tilde{t}$){b$_k$($\tilde{v_k}$).cexp$_k$}$_{k\in K}$ of
5      (True,cexp) → (bs$_0$,True,cexp)
6      (False,cexp) → case get_semi_essentials [] cexp of
7        (q'($\tilde{t'}$){b$_i$($\tilde{v_i}$)}$_{i\in I}$):rest → do
8          q'($\tilde{t'}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$ ← pull_up (q'($\tilde{t'}$){b$_i$($\tilde{v_i}$)}$_{i\in I}$) cexp
9          {(bs$_i$,f$_i$,cexp'$_i$)}$_{i\in I}$ ← {loop_reduce bs$_{i-1}$ cexp$_i$}$_{i\in I}$
10         if any {f$_i$}$_{i\in I}$ then (bs$_i$,True,q'($\tilde{t'}$){b$_i$($\tilde{v_i}$).cexp'$_i$}$_{i\in I}$)
11         else (bs$_0$,False,cexp)
12       [] → case (pull_loop cexp) of
13         cexp' → loop_reduce bs$_0$ cexp;
14         cexp → case (get_potential_semi bs$_0$ cexp) of
15           [] → (bs$_0$,False,cexp)
16           [(defi$_i$,subs$_i$)]$_{i\in I}$ → do
17             for_each (defi$_i$,subs$_i$) do
18               (bs$_i$,v$_i$,cexp$_i$) ← loop_reduce bs$_0$ (pull_up defi$_i$ (cexp[subs$_i$]))
19               if v$_i$ then return (bs$_i$,True,cexp$_i$)
20             return (bs$_0$,False,cexp)
21 loop_reduce bs (loop id(bindings)){(cexp,entries)} =
22 if v then (bs',True,(loop id(updateBS bindings bs')){(cexp',entries)})
23 else (bs,False,(loop id(bindings){(cexp,entries)}))
24   where (bs',v,cexp') = loop_reduce (bs ++ bindings) cexp
```

Figure 5.9: Loop reduction algorithm

figure 5.7). An idempotent reduction is first applied. If it fails, a semi-essential (if any) is pulled up to the root of current control expression. Loop reduction continues recursively to the branches of the newly generated control expression.

It is from line 12, in case that there is no semi-essential found, that the loop reduction makes further optimizations. Function, pull_loop, is used to pull up all loop headers which are independent of the branch variables above them, to the root of the control expression. By doing so, more decisions are included in the loops, which may allow further optimization of the loop bodies. If any loop header is pulled up, loop reduction continues on in the newly constructed control expression. Otherwise, the algorithm tries to identify and pull

Pulled up loops

```
fun foo(x,y:Bool):Int
{
  begin
  loop id1(y1<-y,y2<-y)
  {
    loop id2(y3<-y,y4<-y)
    {
      case x of
      [True.
        case y1 of
        [True.
          case y2 of
          [True.return 1
          |False.return 2
          ]
        |False.
          case y2 of
          [True.return 3
          |False.exit id1(y2,y1)
          ]
        ]
      |False.
        case y3 of
        [True.
          case y4 of
          [True.return 4
          |False.return 3
          ]
        |False.
          case y4 of
          [True.return 2
          |False.exit id2(y4,y3)
          ]
        ]
      ]
    }
  }
}
```

a

Possible reductions

```
fun foo(x,y:Bool):Int
{
  begin
  loop id1(y1<-y,y2<-y)
  {
    loop id2(y2<-y,y1<-y)
    {
      case y1 of
      [True.
        case y2 of
        [True.
          case x of
          [True.return 1
          |False.return 4
          ]
        |False.
          case x of
          [True.return 2
          |False.return 2
          ]
        ]
      |False.
        case y2 of
        [True.
          case x of
          [True.return 3
          |False.return 3
          ]
        |False.
          case x of
          [True.exit id1(y2,y1)
          |False.exit id2(y1,y2)
          ]
        ]
      ]
    }
  }
}
```

b

Table 5.14: Potential semi-essentials

up the potential semi-essentials in the control expression. The function, get_potential_semi , is used to collect the structures of these potential semi-essentials, together with the necessary substitutions for each one. Loop bindings are added into List(Bind) when loop reduction goes pass a loop header (see line 24). When a reduced loop body is returned, the bindings of that loop is updated according to the returned List(Bind) (see line 22). For example in table 5.12, after the case statements being identified as potential semi-essentials, the variable t_2 is substituted by t_1 in the List(Bind). And the binding of loop id_2 is updated accordingly.

Note that different combinations of variable substitutions may provide different opportunities for reductions. Incorrect selection of variable substitution can block further reductions. As shown in table 5.14.a, there are two possible combinations of substitutions: t_1 for t_3 and t_2 for t_4, or t_1 for t_4 and t_2 for t_3. Only the latter one will lead to the result in 5.14.b and enable further idempotent reductions. To address this issue, the loop reduction algorithm tries every element in the list that get_potential_semi returns, until a reduction is made. This is perhaps not the optimal solution, but it is guaranteed to find a reduction if there is any in all possibilities.

5.3 Invariant code removal

In this section, we present an algorithm that focuses on removing all invariant code from loops. An **invariant variable** of a loop is a variable which is not altered during the iterations of the loop. Any decision that is based on invariant variables of a loop or constants is called an **invariant decision** and is referred to as “invariant code” of that loop. Invariant decisions may be present in original source programs but often will be introduced by the loop reduction algorithm (see section 5.2.6).

The section is organized as follows. Section 5.3.1 gives an overview of the invariant code removal algorithm, the three phases of which is discussed in section 5.3.2, 5.3.3 and 5.3.4.

5.3.1 Overview of invariant code removal

As loops are critical parts of any program, removing invariant code from loops can significantly improve program performance. As the outcomes of the invariant decisions of a loop will not change during the iterations, it is thus desirable to move these decisions out of the loop in order to avoid repeated computation on them. In Floey, the invariant code removal algorithm is designed to remove all invariant code from loop expressions.

```
invariant_code_removal : Cexp -> Cexp
invariant_code_removal (exit b($\tilde{v}$)) = exit b($\tilde{v}$)
invariant_code_removal (q($\tilde{t}$){b$_i$($\tilde{v_i}$).cexp$_i$}$_{i\in I}$) =
  q($\tilde{t}$){b$_i$($\tilde{v_i}$).invariant_code_removal cexp$_i$}$_{i\in I}$
invariant_code_removal (loop id binding (cexp, entries)) = do
  cexp' ← invariant_code_removal cexp
  case cexp' of
    loop id' binding' (cexp',entries') → do
      if canFlip binding binding' then do
        flipped ← loop id binding (cexp',entries++entries')
        case invariant_code_removal flipped of
          loop id _ _ → return newloop
          flipped' → return loop id' binding' (flipped',[])
      else return newloop
    _ → return newloop
  where newloop = do
          exp'' ← pull_out_semi (loop id binding (cexp',entries))
          return (pull_all_invariants cexp'')
```

Figure 5.10: Invariant code removal algorithm

The pseudo code for the invariant code removal algorithm is given in figure 5.3.1. It returns a control expression with all invariant code removed from loop expressions. The algorithm does a traversal on a control expression. When a loop expression is encountered, the invariant decisions in it are pulled out in three steps: all semi-essential decisions are first pulled out (line 17), other invariant decisions are then pulled out (line 18), and finally nested loop headers are flipped if necessary (line 8 to 14).

```
main: invariant_removal
def foo(x,y,z:Int)[a,b,c,d]
{
  begin
  exit a
}
fun invariant_removal(t:Int):Int
{
  begin
  loop lp_a(x<-t)
  {
    loop lp_b(y<-t, z<-t)
    {
      loop lp_c(t'<-t)
      {
        v <- w + 1.
        u <- t * 2.
        foo(v,x,z) of
        [a.return 1
        |b.exit lp_a(v)
        |c.exit lp_b(v*u,z)
        |d.
          t''<-t'*10.
          exit lp_c(t'')
        ]
      }
    }
  }
}
```

a

```
main: invariant_removal
def foo(x,y,z:Int)[a,b,c,d]
{
  begin
  exit a
}
fun loop_repeat(t:Int):Int
{
  begin
  v <- w + 1.
  loop lp_a(x<-t)
  {
    foo(v,x,t) of
    [a.return 1
    |b.exit lp_a(v)
    |c.loop lp_b{exit lp_b}
    |d.loop lp_c{exit lp_c}
    ]
  }
}
```

b

Table 5.15: Invariant code removal

By pulling up loop invariant, loop expressions are progressively pushed down as their bodies become smaller. As shown in 5.15, all three loop expressions are pushed down and producing smaller bodies.

Note that in control normal form, all subsidiary entries of a loop expression are reachable local entries and topologically sorted with minimal binding list. We can also assume that local entries of loop expressions have first been optimized. The process of optimizing entries is omitted for simplicity.

The following sections discuss the three steps of invariant code removal in details.

5.3.2 Pulling out semi-essentials from loop

This section discusses a technique to pull out all semi-essentials (see section 3.2.3 for definition) from a loop expression. This is the first step of the invariant code removal algorithm (see line 17 in figure 5.3.1).

```
pull_all_semi : Cexp -> Cexp
pull_all_semi cexp = do
  semis ← get_semi_essentials [] cexp
  pull_up_semis semi cexp

pull_up_semis : List(Definication) × Cexp -> Cexp
pull_up_semis (x:xs) cexp =
  if (x:xs) == [] then cexp
  else pull_up_semis xs (pull_up x cexp)
```

Figure 5.11: Pulling out semi-essentials from loop

Given the transition equality (section 3.2.2), semi-essentials of a loop expression can be pulled up to the root (above the loop header) without changing the meaning of the program. In particular these semi-essentials are independent of the loop and are thus invariant decisions. Figure 5.3.2 gives the algorithm of pulling out all semi-essentials from a loop expression. It first identifies all semi-essentials in the loop expression by calling get_semi_essentials (see figure 5.2.2). These semi-essentials are then pulled up to the root by the function, pull_up_semis.

Original

```
main: foo
fun foo(x,y:Int):Int
{
  begin
  loop lp(a<-x,b<-y)
  {
    case x<b of
    [True.
      c<-a+1.
      exit lp(c,b)
    |False.
      return x
    ]
  }
}
```

a

Pulled

```
main: foo
fun foo(x,y:Int):Int
{
  begin
  case x<y of
  [True.
    loop lp1(a<-x)
    {
      c<-a+1.
      exit lp1(c)
    }
  |False.
    loop lp2(a<-x)
    {
      return x
    }
  ]
}
```

b

Reduced

```
main: loop
fun foo(x,y:Int):Int
{
  begin
  case x<y of
  [True.
    loop lp1(a<-x)
    {
      c<-a+1.
      exit lp1(c)
    }
  |False.
    return x
  ]
}
```

c

Table 5.16: Pulling out semi-essentials from loops

Pulling an unary decision out of a loop will not change the program structure significantly, however, pulling out a branching decision will duplicate the loop structure down each branch. However, each loop is smaller in size, as the branch reduction in the procedure pull_up (section 5.2.3) guarantees that no repetition is added in the result. Table 5.16 gives another example of pulling out branching decisions. After pulling out the case statement, shown in 5.16.b, despite the number of loop expressions being duplicated, the loop bodies in each loop expression are reduced by the branch reduction. In 5.16, loop expression lp$_1$ only has the loop body in the True branch, while lp$_2$ has only the other branch. In extreme cases, some resulting loops may not even have exit statement in the loop body. Such loop expressions are removed altogether by the unlooping normalization (see section

4.4.3). For instance the loop lp_2 is removed in this manner in table 5.16.c.

5.3.3 Pulling out arbitrary loop invariant decisions

Although all the semi-essentials of a loop expression are invariant code of that loop, not all invariant code is semi-essential. This section introduces an algorithm that removes the invariant decisions which are not semi-essential. The algorithm is used in the invariant code removal algorithm, after all semi-essentials are removed from a loop expression (see line 18 in figure 5.3.1).

```
1  pull_all_invariants : Cexp -> Cexp
2  pull_all_invariants (exit b($\tilde{v}$)) = exit b($\tilde{v}$)
3  pull_all_invariants ($q(\tilde{t})\{b_i(\tilde{v_i}).cexp_i\}_{i\in I}$) =
4    $q(\tilde{t})\{b_i(\tilde{v_i})$.pull_all_invariants $cexp_i\}_{i\in I}$
5  pull_all_invariants (loop id binding (cexp, entries)) = unroll cexp []
6    where unroll : Cexp × List(Choice) -> Cexp
7          unroll (loop id binding (cexp, entries)) chs =
8            loop id binding (cexp, entries)
9          unroll $q(\tilde{t})\{b_i(\tilde{v_i}).cexp_i\}_{i\in I}$ chs =
10           $q(\tilde{t})\{b_i(\tilde{v_i})$.(unroll $cexp_i$ chs')$\}_{i\in I}$
11             where chs' = if isInvariant then (($q(\tilde{t})$,$b_i(\tilde{v_i})$):chs) else chs
12         unroll (exit b($\tilde{v}$)) chs =
13           if b == id && chs != [] then do
14             lp = (repeat_reduce chs (loop id binding[$\tilde{v}/\tilde{e}$] (cexp, entries)))
15             return (pull_all_invariants lp)
16           else = exit b($\tilde{v}$)
```

Figure 5.12: Pulling out all loop invariant code

The pseudo code of this algorithm is given in figure 5.3.3. It traverses the loop body to remove loop invariant decisions. Whenever a loop expression is encountered, the procedure unloop is used to unroll some iterations of the loop. unloop traverses the loop body looking for invariant decisions. If a loop expression is encountered, it is left unchanged (line 7). As the invariant code removal algorithm recursively reduces deeper loops first, all inner loops are already optimized. When a decision is encountered, the unloop algorithm goes on in each of its branches (line 9). If the decision is invariant, corresponding choice will be added into the list chs. The list of choices holds all choices made on invariant decisions along the traversal. Whenever an exit statement back to the loop header is found, it is replaced by the loop under the condition that there are some invariant decisions above it (line 13). That is, the list of choices is not empty. Repeat reduction is applied to eliminate repetitions (held in the list of choices) in the newly generated loop, lp. Finally, pull_all_invariants is then recursively called to optimize the resulting lp.

Original

```
fun foo(x:Int):Int
{
  begin
  loop id(t<-x)
  {
    case t<0 of
    [True.return x
    |False.
      y<-x*2.
      exit id(t+y)
    ]
  }
}
```

a

Unrolled

```
fun foo(x:Int):Int
{
  begin
  case x<0 of
  [True.return x
  |False.
    y<-x*2.
    loop id'(t<-x+y)
    {
      case t<0 of
      [True.return x
      |False.
        y<-x*2.
        exit id'(t+y)
      ]
    }
  ]
}
```

b

Repeat reduced

```
fun foo(x:Int):Int
{
  begin
  case x<0 of
  [True.return x
  |False.
    y<-x*2.
    loop id'(t<-x+y)
    {
      case t<0 of
      [True.return x
      |False.
        exit id'(t+y)
      ]
    }
  ]
}
```

c

Table 5.17: Pulling out arbitrary invariant decisions

The example in table 5.17 shows how an invariant decision is pulled out of a loop. The code in 5.17.a results from the loop reduction (section 5.2.6) that pushes y<−x*2 into the loop expression. The assignment is left in one branch as an invariant decision but is not semi-essential. The loop is first unrolled and becomes dead after **exit** id(t+y) is replaced (shown in 5.17.b). The repetition of y<−x∗2 is eliminated by repeat reduction (shown in pull invariant 1.c). In the result, the assignment is not included in any iteration and will only execute in one branch of the case statement.

5.3.4 Flipping loop headers

In the invariant code removal algorithm, if both of the techniques introduced in section 5.3.2 and 5.3.3 fail, it is still possible to pull out invariant code. In nested loop expressions, a decision may use some bound variables defined in the binding of inner loops, which blocks its opportunity of being pulled out of the outer loops. For example in table 5.18.a, the assignment on line 14 uses variables y and z that are defined in the bindings of lp_b and lp_c, which prevents it from being pulled out. By design, however, it can be pulled out of lp_a, as it does not use any bound variables defined in that loop header.

This problem is resolved in invariant code removal by flipping nested loop headers (line 8 to 14 in figure 5.3.1). Two nested loop headers may be flipped if none of the bound variables defined in the outer loop are used as the initialization expressions in the inner loop. After flipping the loop headers, invariant code removal is applied to the newly generated inner loop body. If the flipping process makes some gain, that is some decisions are pulled out of loops, the resulting Floey program will be returned. Otherwise, the looping structure is rolled back to its original form. As shown in table 5.18.b, the assignment w <− y∗z is pulled out from loop lp_a, after two flips of loop headers.

When flipping loop headers, local entries in these loops need to be reorganized. Local entries in the outer level loop will be pushed deeper with its loop header. In the example of table 5.18.b, entry en1 is pushed with lp_a. Entries in deeper level are attached to the

```
main: flip_loop
def foo(x,y,z:Int)[a,b,c,d,e] {
  begin
  exit a
}
fun flip_loop(t:Int) {
  begin
  loop lp_a(x<-t)
  {
    loop lp_b(y<-t)
    {
      loop lp_c(z<-t)
      {
        w<-y*z.
        foo(x,y,z) of
        [a.
          exit lp_a(w)
        |b.
          exit lp_b(w)
        |c.
          exit lp_c(w)
        |d.
          exit en1(w)
        |e.
          exit en2(w)
        ]
        entry en2(x:Int)
        {
          exit lp_c(x)
        }
      }
    }
    entry en1(x:Int)
    {
      case x<0 of
      [True.exit lp_a(x)
      |False.exit en2(x)
      ]
    }
  }
}
```

a

```
main: flip_loop
def foo(x,y,z:Int)[a,b,c,d,e] {
  begin
  exit a
}
fun flip_loop(t:Int) {
  begin
  loop lp_b(y<-t)
  {
    loop lp_c(z<-t)
    {
      w<-y*z.
      loop lp_a(x<-t)
      {
        foo(x,y,z) of
        [a.
          exit lp_a(w)
        |b.
          exit lp_b(w)
        |c.
          exit lp_c(w)
        |d.
          exit en1(w)
        |e.
          exit en2(w)
        ]
        entry en1(x:Int)
        {
          case x<0 of
          [True.exit lp_a(x)
          |False.exit en2(x)
          ]
        }
        entry en2(x:Int)
        {
          exit lp_c(x)
        }
      }
    }
  }
}
```

b

Table 5.18: Flipping loop headers

end of entry list on their levels. The reason is that entries on deeper levels may call the outer level entries. This design maintains the topological order of entries. In table 5.18.b, entry en2 remains in the deepest level and is placed after entry en1.

Chapter 6

Conclusion

In this chapter, we conclude the main contributions of the thesis by reviewing the design of Floey and by comparing the optimizations in Floey to conventional approaches. The comparison shows how Floey's structure facilitates the design of optimizations, but also shows the limitations of some optimizations discussed in this thesis.

The chapter starts by reviewing the Floey language and its contributions to compiler designs in section 6.1. Section 6.2 compares basic optimizations used in control normalization to conventional algorithms, while the reduction algorithm is discussed in section 6.3. Finally, section 6.4 focuses on loop related optimizations.

6.1 Floey intermediate language

As a fundamental area in Computer Science research, compiler construction, especially compiler optimizations, have been studied extensively for decades [Allen and Cocke, 1971; Bacon et al., 1994; Padua and Wolfe, 1986] and have been a popular topic for Computer Science texts [Aho et al., 2007; Appel, 1998; Corman et al., 1990; Ellis, 1986; Muchnick, 1997]. Although a lot of great work has been accomplished, we feel that the organization of the compiler optimizations is not very clear. For example in the classic text [Aho et al., 2007], compiler optimizations are categorized into those based on basic blocks, those that are machine independent or those that are at the instruction level. However, how these different optimizations are related is not explained. It is the same in [Muchnick, 1997], where optimizations are categorized by their functionality, each optimization is explained in isolation so that the extent to which they are complete is also not clear. In fact, these are the recognized sources on optimization, neither provides a complete or systematic coverage.

In this thesis, a language for mid-level intermediate representation, called Floey, is introduced (chapter 2). In Floey, programs are organized around control expressions which are extended basic blocks with loops. Different control expressions are linked together by the structure called entries. The Floey language is not only designed to facilitate machine independent compiler optimizations, more importantly, we hope that from the design of Floey and the reduction of programs therein, a clearer perspective on compiler optimization emerges. The reduction algorithm and the invariant code removal have been implemented (chapter 5). These optimizations have a similar effect to at least four different conventional optimizations, which are rarely viewed as being related.

It is worth mentioning here the scope of our optimizations. As a project decision, we decided to focus our design on control expressions, but not through different entries. Our optimizations thus have smaller scope than the conventional global approaches that handle entire control flow graph. However, entries in Floey provide uniform interfaces between control expression, which allows one to extend our work on control expressions in order to optimize through entries and therefore in the control flow graphs. This is one of the future work of our project.

The following sections compare the optimizations applied to Floey to the conventional approaches. These comparisons show the advantages of designing and implementing optimization algorithms on Floey, as well as some limitations in our work.

6.2 Optimizations in control normalization

This section relates the basic optimizations used in the control normalization phase (chapter 4) to the conventional optimizations.

6.2.1 Constant folding

Constant folding used in the expression normalization (section 4.1.1) is the conventional constant folding algorithm [Allen and Cocke, 1971]. It evaluates the operations on constants and replaces them with their constant results.

6.2.2 Static Single Assignment form

Static single assignment form (SSA) [Cytron et al., 1991], is an intermediate representation that is used in many optimizing compilers, such as GCC [Novillo, 2003, 2004] and SUIF [Stanford SUIF Compiler Group, 1994]. The essential property of SSA form is that whenever a variable is defined it is given a unique name. This means that if two variables have the same name they must also have the same value. Converting program into SSA form makes the bookkeeping issues for variable much simpler and improves the efficiency of optimization algorithms.

In general, the difficulty and the key step of converting a program viewed as a flow graph to SSA form [Cytron and Gershbein, 1993; Cytron et al., 1991] is the placement of ϕ function [Bilardi and Pingali, 1999, 2003; Sreedhar and Gao, 1995]. ϕ functions are pseudo-assignments that are introduced at the join points, the nodes with multiple predecessors in the program control flow graph. The conversion of a Floey program to SSA form on the other hand, is much simpler as control expressions are tree-like structures that do not have joint points except at loop headers, where all loop variables are reassigned in a binding list. When converting Floey control expressions into SSA form, there is therefore no need for ϕ functions. The conversion is accomplished by simply renaming each variable in every definition, including loop bindings, uniquely.

6.2.3 Copy and constant propagations

Copy and constant propagations [Callahan et al., 1986, 2004; Vanbroekhoven et al., 2003; Wegman and Zadeck, 1991] are the code transformations that, given an assignment of a variable or a constant to a variable: x<−e, replace the later uses of x with e until the variable is redefined. When the definition of x is no longer used in the code, the assignment can be removed. Conventional copy and constant propagation need to handle the redefinitions of variables and to determine whether x is dead. Joint points in Control Flow Graph also requires extra analysis.

Implementing copy and constant propagations in Floey is much simpler, as Floey code is first converted to control normal form, which includes Static Single Assignment form (SSA). Value of variables cannot be altered after their definitions as there are no variable reassignments of variables in SSA form. In Floey, variable substitutions for copy and constant propagation always continues till the leaves of control expressions. The assignment x<−e is then never used after the substitutions and so can always be removed. The tree-like structure of control expression also simplifies the implementation as both propagations can be implemented on a tree traversal.

The scope of copy and constant propagation in Floey, however, is smaller than conventional means as it is only applied to control expressions instead to the whole program flow graph.

6.2.4 Unreachable code elimination

Unreachable code is the parts of a program that will never be executed. In a Control Flow Graph (CFG), unreachable code are the unreachable nodes from the "begin" node. The problem of recognizing unreachable code of a program then becomes the problem of identifying unreachable nodes in the CFG of that program. By removing unreachable code, programs result in smaller size, which require less resources. This also improves the efficiency of control flow analysis. The optimization is usually called unreachable code elimination [Aho et al., 2007].

In Floey, a CFG represents a function or control operation, which is broken down into a control expressions and various entries. Each control expression, in control normal form (section 4), is a tree-like structure with loops which may have reachable local entries. Thus a control expression cannot contain unreachable code. It is the existence of the unreachable entries in a function or a control operation that introduces unreachable code into Floey. Those are the entries that are never directly or indirectly called by the control expression in that function or control operation.

Unreachable code elimination is accomplished in the entry promotion phase (section 4.3.3), when entries are moved to the highest possible level and topologically sorted. At the same time, all unused entries are identified and discarded. By applying entry promotion to every entry in a CFG, all unreachable code in that CFG is eliminated.

6.3 Optimizations in the repeat reduction and the reduction algorithm

This section compares the optimizations achieved in the repeat reduction algorithm (section 5.1) and the reduction algorithm (section 5.2) to three conventional optimizations:

common sub-expression elimination [Cocke, J., 1970; Ullman, 1972], if simplification [Muchnick, 1997] and partial dead code elimination [Bodík and Gupta, 1997; Briggs and Cooper, 1994; Dhamdhere, 1991; Feigen et al., 1994; Knoop et al., 1994].

6.3.1 Common sub-expression elimination

Common sub-expression elimination [Cocke, J., 1970; Ullman, 1972] removes the repetitions of computations on common sub-expressions. There are two versions of common sub-expression elimination: a local version that applies to only basic blocks and global version that applies to whole control flow graph.

In Floey, programs are first control normalized (chapter 4), in which complex expressions are split into atomic instructions. Repeat reduction then eliminates repetitive computations on common sub-expressions. In the control normal form, since there is no variable reassignment, the implementation of repeat reduction is much easier. Only one auxiliary list is used to hold the history of variables with the expressions that they are assigned to.

As repeat reduction is applied to control expressions, which are extended basic blocks with loops, its scope is larger than the local common sub-expression elimination but it does not perform global common sub-expression elimination.

6.3.2 If simplification

Conditionals have long been considered as expensive operations, as modern architectures are extremely sensitive to control flow changes [Yang et al., 2002]. In repeat reduction, repeated decisions are reduced and removed with their unreachable branches. This is similar to a branch optimization called “if simplification” [Muchnick, 1997], which optimizes if-then-else conditional statements in the following four cases.

1. If either branch is empty, remove the corresponding branch and change the if-then-else construct to a if-then construct.
2. If both branches are empty, remove the entire if statement.
3. If the condition is a constant, evaluate the condition and remove the unreachable branch. If it is the only branch, remove the entire if statement.
4. If common sub-expressions occur in conditions of a if statement and a subsequent dependent if statement, evaluate the latter condition when possible and replace the if statement with the branch in the corresponding arm.

The first two cases simply cannot happen in Floey as no empty branch is allowed. Repeat reduction (section 5.1) and idempotent reduction (section 5.2.4) cover the last two cases in “if simplification”. Repeat reduction uses constant conditions or common sub-expression condition to predict the direction of control flow on case and control statements. While idempotent reduction eliminates those case or control statements whose branches are the same.

One advantage of repeat reduction over “if simplification” is that it handles multiple decisions on data types. “If simplification” only handles binary conditionals on boolean conditions.

6.3.3 Partial dead code elimination

In conventional compiler texts, if a variable is defined but never used, it is said to be dead. The assignment to these variables is then **dead code** and thus can be removed. The optimization of removing this code is called dead code elimination [Aho et al., 2007].

When the definitions of the variables are only used in some branches of the Control Flow Graph, they are are called **partially dead**. For example in table 6.1.a, variable y is dead as it is never used, while x is only partially dead as it is used in one of the two branches. Therefore, unlike the assignment to y that can be immediately removed (shown in table 6.1.b), assignment to x can only be removed from the branches that x is not used.

```
x<-a+1.
y<-a*3.
case a<0 of
[True. return x
|False. return a
]
```

a

```
case a<0 of
[True.
  x<-a+1.
  return x
|False. return a
]
```

b

Table 6.1: Partial dead code elimination

Identifying and eliminating partial dead code has been an active topic in compiler optimization research. Various algorithms have been developed for partial dead code elimination [Bodík and Gupta, 1997; Briggs and Cooper, 1994; Dhamdhere, 1991; Feigen et al., 1994; Knoop et al., 1992, 1994], many of which require modification of program control flow structure.

One of the main objectives of the reduction algorithm (section 5.2) is to eliminate partial dead code in control expressions. It is accomplished by progressively pushing every decision further down in the control expression, by pulling up semi-essentials (section 5.2.3). Whenever a decision is dead on one branch, it is eliminated by idempotent reduction (section 5.2.4). Table 5.9 and 6.1 are both examples of how (partial) dead code is eliminated by the reduction algorithm. Unlike the conventional approaches that focus only on assignments, the reduction algorithm also moves and eliminates case and control statements, as in Floey, these statements are all decisions. For example, the so called "anticipated expression" in lazy code motion algorithm [Knoop et al., 1992] is the idea of semi-essential but only on expressions. While pushing statements in the programs (even into loops) are also applied by [Briggs and Cooper, 1994; Knoop et al., 1994], but only to assignments. We believe the new perspective of Floey does give an edge to optimization designed on it.

6.4 Loop related optimizations

As loops are the critical parts of most programs, any improvement to the loop execution time can lead to significant benefit to program efficiency. To this end, many loop optimization algorithms have been developed to reduce the size of loops and thus to reduce the complexity in loops. In this section, we compare the loop related optimizations in Floey to five conventional techniques.

Promoting non-local entries out of loops (section 4.3.3) and the reduction algorithm in loops (section 5.2) have similar effects to loop dead elimination [Behera and Kumar, 2005]. In invariant code removal (section 5.3), the process of pulling semi-essentials out of loops (section 5.3.2) is similar to two conventional optimizations: loop invariant code motion [Neel and Amirchahy, 1975] and loop unswitching [Allen and Cocke, 1971]. While flipping loop headers is similar to loop interchange [Allen and Kennedy, 1984], although it is for different purposes. The more aggressive restructuring modification that unrolls loops in invariant code removal is similar to loop unrolling but for different purposes too [Dongarra and Jinds, 1979; Ellis, 1986].

6.4.1 Loop dead elimination

```
main: loopdead
fun loopdead(x:Int):Int {
  begin
  loop lp(y<-x)
  {
    z<-y*2.
    case y>0 of
    [True. exit en(z)
    |False. exit lp(y+10)
    ]
    entry en(x:Int)
    {
      return x
    }
  }
}
```

a

```
main: loopdead
fun loopdead(x:Int):Int {
  begin
  loop lp(y<-x)
  {
    case y>0 of
    [True.
      z<-y*2.
      exit en(z)
    |False. exit lp(y+10)
    ]
  }
  entry en(x:Int)
  {
    return x
  }
}
```

b

Table 6.2: Loop dead elimination

Loop dead variables are the variables which are defined in a loop but never used in the iterations of that loop. The values of loop dead variables may be altered during the iterations but only the last one is used (outside the iteration). The optimization of pushing the assignments to loop dead variables out of loop is called loop dead elimination [Behera and Kumar, 2005; Shaw and Kumar, 2005].

The only case of loop dead code in Floey is the partially dead code which is only used in the branches that are exiting the loop expression. For example in table 6.2.a, z is a loop dead variable which is altered and used, but only in the non-local entry en. The reduction algorithm in the loop body eliminates this partially dead code. As shown in table 6.2.b, the assignment is pushed down to the branch exiting to entry en. This way, the assignment avoids occurring in loop iterations.

```
main: loopdead
fun loopdead(x:Int):Int {
  begin
  loop lp(y<-x)
  {
    z<-y*2.
    exit en(y,z)
    entry en(y,z:Int)
    {
      case y>0 of
      [True.return z
      |False.exit lp(y+10)
      ]
    }
  }
}
```

Table 6.3: Loop dead elimination

However, as the reduction algorithm applies only to control expressions, loop dead variables which could be eliminated through entry points can still be left in resulting code. For instance the variable z in table 6.3 is loop dead variable but is not optimized by the reduction algorithm. This can be resolved should one extends the reduction algorithm through entries, as discussed in section 6.1.

6.4.2 Loop invariant code motion

The optimization that removes loop invariant code (see section 5.3 for definition) from loops is called loop invariant code motion (also called loop invariant elimination) [Neel and Amirchahy, 1975]. In this algorithm, all assignments to invariant variables of the loop are moved into a manually created basic block called the “pre-header” that is inserted right above the loop header. This design avoids dramatic changes of the control flow structure during the process. However, loop invariant code motion does not guarantee a gain in program performance. As shown in table 6.4.a, if y>0 is true, the assignment z<−x+1 is never executed in the original code. While in 6.4.b, the assignment is always executed once, which might reduce the efficiency of the resulting code.

```
main: foo
fun foo(x:Int):Int {
  begin
  loop lp(y<-x)
  {
    case y>0 of
    [True.return 1
    |False.
      z<-x+1.
      exit lp(z)
    ]
  }
}
```

a

```
main: foo
fun foo(x:Int):Int {
  begin
  z<-x+1.
  loop lp(y<-x)
  {
    case y>0 of
    [True.return 1
    |False.exit lp(z)
    ]
  }
}
```

b

Table 6.4: Loop invariant code motion

In Floey, invariant code removal pulls out semi-essentials from loop expressions (section 5.3.2). When the semi-essentials are assignments, the optimization is similar to loop invariant code motion. It moves all semi-essential assignments onto the top, above the loop expression. Since semi-essentials in a loop expression are the invariant code that occur in every path, pulling out only semi-essentials guarantees no added computation in the resulting code.

The scope of Floey's invariant code removal is smaller than loop invariant code motion as it only applies to control expressions. But it has an advantage over invariant code motion: it removes decisions which are not semi-essential (see section 6.4.4) from loops with the guarantee that program efficiency is not decreasing.

6.4.3 Loop unswitching

Loop unswitching [Allen and Cocke, 1971] moves loop invariant conditional statements out of loops, saving the overhead of the conditional branching in the loop, reducing the loop size, and possibly increasing the potential parallelization of the program [Bacon et al., 1994]. In invariant code removal (section 5.3), pulling out semi-essentials which are multiple decisions (case statements and control statements) from loop expressions (section 5.3.2), is similar to the loop unswitching.

Note that in conventional compiler optimization texts, loop unswitching is distinguished from loop invariant code motion, as the conditionals and assignments are generally considers as different constructs. And loop unswitching modifies program control flow graph structure more aggressively as it produces smaller loops (see table 5.15.b). However, Floey provides us the platform, in which both optimizations can be seen as being the same (section 5.3.2); conditionals and assignments in Floey are both the same construct as they are decisions. Pulling up unary decisions out of loop thus is just a special case of pulling out branching decisions. It is conceptually useful to see these optimization as being the same as thus it helps in the organization of optimizations.

6.4.4 Loop unrolling

Loop unrolling (also called loop unwinding) is the optimization that reduces the number of iterations of a loop by duplicating the iteration body [Dongarra and Jinds, 1979; Ellis, 1986]. It may reduce the control overhead and can possibly increase the instruction parallelism.

In invariant code removal, the code manipulation of pulling out arbitrary invariant decisions (section 5.3.3) applies a similar idea to loop unrolling, but for different purposes.

It unrolls only the first iteration of loops, in order to remove invariant decisions from iterations. By doing so, loop strength is reduced at the expense of possible code size increase due to the duplication of loop bodies.

6.4.5 Loop interchange

Loop interchange [Allen and Kennedy, 1984; Wolfe, 1989] is the optimization that rearrange the order of nested loops so as to enhance code performance in parallel or vector machines.

In invariant code removal, loop headers are also flipped but only for the purpose of pulling out more semi-essentials from loop expressions (section 5.3.4). Because a semi-essential may be blocked by one loop header as it uses some of the bound variables in the bindings of that header, but is not blocked by other ones. By flipping the order of nested loop headers, the semi-essentials can be pulled out from the loops in which they are independent.

Appendix A

Floey Grammar

A.1 Floey Header: FloeyHeader

```
FloeyImport : Import FloeyImport |
Import : import file
```

A Floey file starts optionally by specifying other Floey files it imports. Floey compiler will recursively collect all the imported files and tokenize them. The import information will be removed once it is processed. Resulting tokens will be combined together and sent to the Floey parser. The Floey parser therefore expects the token list of a complete and legitimate Floey program.

A.2 Floey program: Floey

```
Floey : FloeyMain FloeyDefs
FloeyMain : mainfun ':' lower |
```

A Floey program starts by specifying the function at which the execution begins. This is specified by FloeyMain and can take the following form.

```
mainfunction: helloworld
```

When no main function is specified, the Floey file cannot be compiled into stand-alone executable but can only be imported as a library.

A.3 Floey Definitions: FloeyDefs

```
FloeyDefs : FloeyData FloeyDefs
          | FloeyArray FloeyDefs
          | FloeyControl FloeyDefs
          | FloeyFun FloeyDefs
          |
```

The main body of a Floey program consists of a list of Floey definitions, as defined by FloeyDefs. Each Floey definition, FloeyDef, is either a data declaration, FloeyData, an array declaration, FloeyArray, a control operation definition, FloeyControl, or a function definition, FloeyFun.

A.3.1 Data declaration: FloeyData

```
FloeyData : data upper ParaList '=' ConsList
ParaList : lower ParaList |
```

A Floey data declaration, FloeyData, starts with the key word **data**, followed by the name and parameters of the type. The first letter of type name has to be in upper case, while type variables start with lower a case letter. Type variables are used in polymorphic data declarations. (e.g. List a, Tree a b)

```
ConsList : Cons MoreConsList
MoreConsList : '|' Cons MoreConsList |
Cons : upper TypeList
TypeList : '(' TypeID MoreTypeList ')' | '(' ')' |
MoreTypeList : ',' TypeID MoreTypeList h|
TypeID : upper moreType | lower
moreTypeArg : TypeArg moreTypeArg |
TypeArg : '(' TypeID ')' | upper | lower
```

After the “=” is the constructor list of that data declaration, ConsList. Constructors are separated by "|". Each constructor consists of a name that begins with an upper case letter, followed by a list of types, TypeList. The list of types specify the arguments types of that constructor. For example the declaration of a list of polymorphic type is as follows.

```
data List a = Nil
            | Cons (a, List a)
```

A.3.2 Array declaration: FloeyArray

```
FloeyArray : array lower Dims ':' TypeID
Dims : Dim MoreDims
MoreDims : Dim MoreDims |
Dim : '[' int ']'
```

An array declaration FloeyArray consists of the key word **array**, the name that starts with a lower case letter, the dimensions, Dims, and the type of array elements, TypeID. For example a two dimensional array of lists of integer is declared as follows.

```
array b[10][15]:List Int
```

A.3.3 Control operation definition: FloeyControl

```
FloeyControl : def lower ArgList BranchSpecList '{' ProcBody '}'
```

A control operation definition consists of the keyword **def**, a name which starts with a lower case letter, an argument list, ArgList, a branch specification list, BranchSpecList, and the body of the control operation, ProcBody.

```
ArgList : '(' Arg MoreArgList ')' | '(' ')' |
MoreArgList : ',' Arg MoreArgList |
Arg: lower MoreVarList ':' TypeID
```

An argument list, ArgList, consists a list of variable-type pairs. For example an argument x of boolean and an argument y of the type: list of integer, is declared as follows.

```
(x:Bool,y:List Int)
```

However, if two or more consecutive arguments are of the same type, the type needs to be input only once after the last variable. For example three arguments of list of boolean: (x,y,z:List Bool)

```
BranchSpecList : '[' BranchSpec MoreBranchSpecList ']' | '[' ']' |
MoreBranchSpecList : ',' BranchSpec MoreBranchSpecList |
BranchSpec : lower TypeList
```

A control operation returns a list of branch specifications enclosed by a pair of square brackets, as defined by BranchSpecList. The syntax of a branch specification is very similar to that of a constructor. The only difference is that a name for a branch specification starts with a lower case letter. The different branch specifications are separated by commas.

```
ProcBody : FloeyDefs ConExp_Entry
```

The main procedure body, ProcBody, consists of a list of local Floey definitions, FloeyDefs, and a control expression with a list of entries, ConExp_Entry. ConExp_Entry is discussed in the next section.

A.3.4 Function definition: FloeyFun

```
FloeyFun : fun lower ArgList ':' TypeID '{' ProcBody '}'
```

A Floey function, as defined by FloeyFun, differs from a Floey control operation, FloeyControl, in that it returns a value of some type, indicated by TypeID, rather than a branch specification.

A.4 Entry and control expression: ConExp_Entries

```
ConExp_Entries : main ConExp Entries | Entries
```

In Floey, each main code section, ConExp_Entry, consists with a control expression and a list of entries. ConExp_Entry starts with the keyword **main**, followed by the control expression, ConExp, and the entry list, Entries. In case the control expression is empty, the keyword **main** should be omitted.

A.4.1 Entry: Entry

```
Entries : Entry Entries |
Entry : entry lower ArgList '{' ConExp_Entry '}'
```

The entry list, as defined by Entries, is a (possibly empty) list of entries. Each entry, as defined by Entry, starts with the keyword **entry** followed by the the entry name, a argument list, ArgList, and a ConExp_Entries enclosed my curly brackets. An entry name starts with a lower case letter.

A.4.2 Control expression: ConExp

```
ConExp : exit lower ExpList
       | return Exp
       | case Exp of Branches
       | lower ExpList of Branches
       | lower "<-" Exp '.' ConExp
       | loop lower BindingList '{' ConExp_Entry '}'
       |
```

A control expression, ConExp, is either an exit statement, a return statement, a case statement, a control statement, an assignment, or a loop expression.

ExpList

```
ExpList : '(' Exp MoreExpList ')' | '(' ')' |
MoreExpList : ',' Exp MoreExpList |
```

An ExpList is a comma separated list of expressions enclosed by a pair of braces. In case the list is empty, the braces can be omitted as well.

Branches

```
Branches: '[' lower VarList '.' ConExp Restbranches ']'
        | '[' upper VarList '.' ConExp Restbranches ']'
Restbranches: '|' lower VarList '.' ConExp Restbranches
            | '|' upper VarList '.' ConExp Restbranches
            |
```

Branches of case statements or control statements are enclosed in square brackets. Different branches are separated by "|".

VarList

```
VarList : '(' lower MoreVarList ')' | '(' ')' |
MoreVarList : ',' lower MoreVarList |
```

In each branch, VarList is the list of variables that are bound to the following control expression, ConExp. Please note that, the Floey parser does not distinguish branches of case statements from those of control statements. That is left to the semantic checking phase. The reason for this design is that in semantic checking, the compiler would be able to provide more useful feedback to assist debugging.

BindingList

```
BindingList : '(' Binding MoreBindingList ')' | '(' ')' |
MoreBindingList : ',' Binding MoreBindingList |
Binding : lower "<−" lower
```

A binding list, BindingList, is a list of variable pairs separated by commas. Each variable pair, separated by "<−", is a binding: bound variable on the left and free variable on the right.

A.5 Expression: Exp

```
Exp : Exp '+' Exp | Exp '−' Exp | Exp '∗' Exp | Exp '/' Exp
    | Exp mod Exp | Exp "&&" Exp | Exp "||" Exp | Exp xor Exp
    | Exp "==" Exp | Exp "!=" Exp | Exp '<' Exp | Exp '>' Exp
    | Exp "<=" Exp | Exp ">=" Exp | '−' '(' Exp ')'
    | lower FunArgList | upper ExpList | '(' Exp ')' | Term
    | lower Indices
Term : '−' Atom | Atom
Atom : int | real | lower | char
FunArgList : '(' Exp MoreExpList ')' | '(' ')'
Indices : Index MoreIndices
MoreIndices: Index MoreIndices |
Index : '[' Exp ']'
```

An expression, as defined by Exp, can simply be an integer, a real number, a variable, a character, or an array element. It can also be any function or constructor applied to other expressions. Please note any identifier for variables, arrays or functions starts with a lower case letter, while an identifier for constructor starts with an upper case letter.

Appendix B

Source Code

B.1 Data structures used in the Floey compiler

```
{-Floey program-}
data Floey = Floey String [TypeDec] [ArrayDec] [Def]
    deriving (Eq)
{-function or control operation definition-}
data Def = Def String [Arg] [Spec] Proc
    deriving Eq
type Arg = (String, TypeID) --argument
type Spec = (String, [TypeID]) --branch specification
type Proc = ([TypeDec], [ArrayDec], [Def], Body)
type Body = (ConExp, [Entry])
{-data declarations-}
data TypeDec = TypeDec TypeID [Cons]     --data type declaration
    deriving Eq
data TypeID = TypeID String [TypeID]     --type identification
    deriving (Eq,Ord)
data Cons = Cons String [TypeID]     --constructor
data ConsArg = ArgLeaf String
             | ArgBranch Cons
    deriving (Show,Eq)
{-array declarations-}
data ArrayDec = ArrayDec String TypeID [Int]
```

```
    deriving Eq
{-entry-}
data Entry = Entry String [Arg] Body
    deriving (Show, Eq)
{-control expression-}
data ConExp = FLOEYcase Exp [((String, [String]), ConExp)]
            | FLOEYcontrol ConOp [((String, [String]), ConExp)]
            | FLOEYexit String [Exp]
            | FLOEYreturn Exp
            | FLOEYloop String [Binding] Body
            deriving Eq
type Binding = (Exp, Exp) --loop binding
data ConOp = ConOp String [Exp]
    deriving (Ord, Eq)
{-expression-}
data Exp = BinE String Exp Exp
    | FunE String [Exp]
    | ConsE String [Exp]
    | ElemE String [Exp]
    | BraceE Exp
    | NegE Exp
    | IntE Int
    | RealE Double
    | CharE Char
    | VarE String
    deriving (Ord, Eq)
```

B.2 Normalizations

```
{-expression expansion-}
expandNode :: ConExp -> Int -> (ConExp, Int)
expandNode et num =
    if simAssign then (FLOEYcase (head exps) simbr, simn)
    else case et of
      FLOEYloop _ _ _ ->
        (FLOEYloop s bindings (newetree',entries),newnum')
      _ -> if null exps then (et, num) else (newet, newn')
        where (FLOEYloop s bindings (etree,entries)) = et
              (newetree',newnum') = expandNode etree num
              --id = getConExpID et
              simAssign = sim' et
                where sim' (FLOEYcase q args) =
                            length args == 1 &&
                            (fst $ fst $ head $ args) ==
                            "#Assign" && isSemiFinal q
                      sim' _ = False
              (simbr,simn) = updateBranches et num
              leaveOne = isAssign et
              exps = getConExpExp et
              ((addedexps, newexps), newnum) =
                foldl (\x y -> updatelist x y) (([], []), num) exps
                where updatelist ((a, b), n) ex =
                        if null exs
                            then ((a, b ++ ex : []), n)
                            else ((a ++ exs, b ++ nexp:[]), newn)
                        where (exs, newn) = expandExp ex n True
                              nexp = if leaveOne then fst $ last exs
```

```
                                      else VarE (tmpPrefix ++ show (newn - 1))
              (newbr,newn') = updateBranches et newnum
              newet' = case et of
                        (FLOEYcase q _) -> FLOEYcase q newbr
                        (FLOEYcontrol q _) -> FLOEYcontrol q newbr
                        _ -> et
              newet = fst $ foldl (\(x,i) y ->
                if snd y < 0 then (x,i)
                else (FLOEYcase (fst y)
                     [((”#Assign”, [tmpPrefix ++
                        show (snd y)]), x)],i-1))
                        (updateConExpExp newet'
                        newexps,-1) $
                        (if leaveOne
                        then tail $ reverse addedexps
                        else reverse addedexps)

expandExp :: Exp -> Int -> Bool -> ([(Exp, Int)], Int)
expandExp ex num flag = if isFinal ex then ([], num)
    else case ex of
        (FunE s explist) ->
            (expandedExps ++ [(FunE s newlist,
            if flag then newnum else -1)],
            newnum+(if flag then 1 else 0))
            where (newlist, expandedExps, newnum) =
                    foldl (\(a, b, n) x ->
                      if isFinal x then (a ++ [x], b, n)
                      else let (ex',n') =
                    expandExp x n True in
                    ((a ++ [(VarE (tmpPrefix ++
                    (show $ (n') - 1)))],
                    b ++ ex', n'))) ([], [], num) explist
    (ConsE s explist) ->
      (expandedExps ++ [(ConsE s newlist,
      if flag then newnum else -1)],
      newnum+(if flag then 1 else 0))
        where (newlist, expandedExps, newnum) =
                foldl (\(a, b, n) x ->
                  if isFinal x then (a ++ [x], b, n)
                  else let (ex',n') =
                  expandExp x n True in
                  ((a ++ [(VarE (tmpPrefix ++
                  (show $ (n') - 1)))],
                  b ++ ex', n'))) ([], [], num) explist
    (ElemE s explist) ->
      (expandedExps ++ [(ElemE s newlist,
      if flag then newnum else -1)],
      newnum+(if flag then 1 else 0))
        where (newlist, expandedExps, newnum) =
                foldl (\(a, b, n) x ->
                  if isFinal x then (a ++ [x], b, n)
                  else let (ex',n') =
                    expandExp x n True in
                    ((a ++ [(VarE (tmpPrefix ++
                    (show $ (n') - 1)))],
                    b ++ ex', n'))) ([], [], num) explist
        (NegE exp) -> (newlist, newnum+1)
            where (explist, newnum) = expandExp exp num True
                  newlist =
                    explist ++ if isFinal exp then [(NegE exp,num)]
                    else [(NegE $
                         VarE (tmpPrefix ++ (show $ newnum-1)), newnum)]
        (BraceE exp) -> expandExp exp num True
        (BinE s exp1 exp2) ->
        (leftexp ++ rightexp ++ (thisexp : []), num2 + 1)
            where (leftexp, num1) = expandExp exp1 num True
                  (rightexp, num2) = expandExp exp2 num1 True
                  newexp1 = if num1 == num then exp1
                            else VarE $ tmpPrefix ++ show (num1 - 1)
                  newexp2 = if num2 == num1 then exp2
                            else VarE $ tmpPrefix ++ show (num2 - 1)
                  thisexp = (BinE s newexp1 newexp2, num2)

{-expression collection-}
expCollect :: ConExp -> ConExp
expCollect et =
    collect et
    where (useNO,subLst) = useCount et M.empty M.empty
          reLst = foldl (\l (s,i) ->
            M.insert s (needReplace s i) l) M.empty (M.toList useNO)
          collect :: ConExp -> ConExp
          collect (FLOEYexit s ex) =
            FLOEYexit s (map (\x -> varReplace x reLst subLst) ex)
          collect (FLOEYreturn ex) =
      FLOEYreturn (varReplace ex reLst subLst)
     collect (FLOEYcase q args) =
      case getAssignPro (FLOEYcase q args) of
       SS s -> case M.lookup s reLst of
             Just True -> collect (snd $ head args)
             _ -> FLOEYcase newq
                 (map (\(p,e) ->
                 (p,collect e)) args)
       _ -> FLOEYcase newq (map (\(p,e) -> (p,collect e)) args)
      where newq = varReplace q reLst subLst
     collect (FLOEYloop s bs (et,entries)) =
      FLOEYloop s newbs (collect et,
      map (expCollect_Entry) entries)
      where newbs = map (\(a,b) ->
              (a, varReplace b reLst subLst)) bs
     collect et = et

{-SSA form-}
diffName :: ConExp -> Int -> (ConExp,Int)
diffName et i = diffName' et i
 where diffName' :: ConExp -> Int -> (ConExp,Int)
    diffName' (FLOEYcase q args) n = (FLOEYcase q args',n')
     where (args',n') = foldl (\(newargs,num) b->
               diffOneBranch newargs num b) ([],n) args
        diffOneBranch bs i ((b,p),et) =
          (bs++[((b,p'),et')],i'')
         where (p',i') = foldl (\(nv,ni) v ->
                   (nv++[diffVar v ni],ni+1)) ([],i) p
```

```
                (et',i'') = diffName' (simReplace
                            (map (VarE) p) (map (VarE) p') et) i'
diffName' (FLOEYcontrol q args) n = (FLOEYcontrol q args',n')
  where (args',n') = foldl (\(newargs,num) b->
                     diffOneBranch newargs num b) ([],n) args
        diffOneBranch :: [Branch] -> Int -> Branch -> ([Branch],Int)
        diffOneBranch bs i ((b,p),et) = (bs++[((b,p'),et')],i'')
          where (p',i') =
                  foldl (\(nv,ni) v ->
                  (nv++[diffVar v ni],ni+1)) ([],i) p
                (et',i'') =
                  diffName'
                  (simReplace (map (VarE) p) (map (VarE) p') et) i'
diffName' (FLOEYloop s bins (et,entries)) n =
  (FLOEYloop s newbins (newet,entries'),n''')
    where (inten,exten) = unzip bins
          (newinten,n') =
            foldl (\(nin,nn) (VarE v) ->
            (nin++[VarE $ diffVar v nn],nn+1)) ([],n) inten
          (newet,n'') = diffName' (simReplace (inten) (newinten) et) n'
          newbins = zip newinten exten
          (entries',n''') =
            foldl (\(newen,newi) en ->
            let (en',newi') = diffName_Entry en newi in
            (newen++[en'],newi'))
            ([],n'') entries
diffName' (FLOEYreturn e) n = (FLOEYreturn e,n)
diffName' (FLOEYexit s exs) n = (FLOEYexit s exs,n)
```

```
{-entry promotion-}
promote_inTree :: ConExp -> (ConExp,[Entry])
promote_inTree (FLOEYexit s ex) = (FLOEYexit s ex,[])
promote_inTree (FLOEYreturn ex) = (FLOEYreturn ex,[])
promote_inTree (FLOEYcase q args) = (FLOEYcase q newargs,ens)
    where (newargs,ens) = foldl (\(n,e) (p,et) -> let (newet,ens) =
            promote_inTree et in (n++[(p,newet)],e++ens)) ([],[]) args
promote_inTree (FLOEYcontrol q args) = (FLOEYcontrol q newargs,ens)
    where (newargs,ens) = foldl (\(n,e) (p,et) -> let (newet,ens) =
            promote_inTree et in (n++[(p,newet)],e++ens)) ([],[]) args
promote_inTree (FLOEYloop s bs (et,entries)) =
  (FLOEYloop s bs (newet,l),f)
    where (newet,added) = promote_inTree et
          newentries =
            foldl (\newen this ->
            newen ++ promote_inEntry this) [] entries
          (locals,frees) =
            classifyEntry (FLOEYloop s bs (newet,newentries++added))
          l = foldl (\l this -> l++[(findEn this)]) [] locals
          f = foldl (\f this -> f++[(findEn this)]) [] frees
          findEn :: String -> Entry
          findEn s =
            case find (\(Entry x _ _) -> s == x) (newentries++added) of
                        Just en -> en

{-loop binding reduction and dead loop elim-}
loopbs_reduce :: ConExp -> ConExp
loopbs_reduce (FLOEYloop s bs (et,entries)) =
    if alive || (not $ null entries) then FLOEYloop s newbs newbody
    else et       --kills a loop when no loop calls
    where (inten,exten) = unzip bs
          (used,changed,alive) =
            loop' (expandBody (et,entries) []) []
            (map (\_->False) inten) False
          todolst =
            -- 0: do nothing, 1: unused, 2: unchanged
            map (\(VarE x,c) ->
                if x `notElem` used then 1
                else if c then 0 else 2) (zip inten changed)
          todo' = fst $ foldl (\(l,n) t ->
                    if t==0 then (l,n+1)
                    else (l++[n],n+1)) ([],0) todolst
          unchanged = filter (\(b,c)->c==2) (zip bs todolst)
          et' = foldl (\tree ((VarE i,e),_) ->
                FLOEYcase e [(("#Assign",[i]),tree)]) et unchanged
          newbs = if null todo' then bs
                  else fst $ unzip $ filter (\(a,b)->b==0) (zip bs todolst)
          newbody = if null todo' then (et,entries) else (et',entries)
          -- used and changed variables in loop
          loop' (FLOEYreturn ex) u c alive = (u',c,alive)
            where u' = union (getVar (E ex)) u
          loop' (FLOEYexit l ex) u c alive = (u',c',alive')
            where u' = union (foldl (\vl exp ->
                        vl ++ (getVar (E exp))) [] ex) u
                  c' = if l == s
            then map (\(ch,(v,v'),e)->
                 if ch then ch
                 else (v/=e && v'/=e)) (zip3 c bs ex)
            else c
         alive' = alive || l == s
     loop' (FLOEYcase q args) u c alive =
      foldl (\(ud,cd,al) (p,et) ->
      loop' et ud cd al) (u++(getVar (E q)),c,alive) args
     loop' (FLOEYcontrol con args) u c alive =
      foldl (\(ud,cd,al) (p,et) ->
      loop' et ud cd al) (u++(getVar (C con)),c,alive) args
     loop' (FLOEYloop l' b' (et',entries')) u c alive =
      loop' et' u' c alive
      where u' = foldl (\used (i,VarE e) -> union used [e]) u b'
```

B.3 Repeat reduction

```
repeat_reduce :: ConExp -> EntryArgLst -> ConExp
repeat_reduce t arglst = branchs_reduce_arglst [] arglst $ normalTree t

branchs_reduce_arglst :: Choices -> EntryArgLst -> ConExp -> ConExp
branchs_reduce_arglst ch arglst et = branchs_reduce ch et where
  branchs_reduce :: Choices -> ConExp -> ConExp
  branchs_reduce ch (FLOEYexit s e) =
    case M.lookup s arglst of
      Just bl ->
        FLOEYexit s
        (fst $ unzip $ filter (\(a,b)->b) $ zip e bl)
      _ -> FLOEYexit s e
```

```
branchs_reduce ch (FLOEYreturn e) = FLOEYreturn e
branchs_reduce ch (FLOEYcase q (((br,pro),et):rest))
  | isAssign (FLOEYcase q (((br,pro),et):rest)) =
        if (VarE (head pro) == q) then branchs_reduce ch et
        {-const/copy propagation-}
        else if isConst q || isVar q
             then branchs_reduce ch $
                simReplace (map (VarE) pro) [q] et
        else case get_choice ch (E q) of
            SS (br',vl') ->
                if br' == "#Assign"
                then branchs_reduce ch $
                    simReplace (map (VarE) pro) (map (VarE) vl') et
                else branchs_reduce ch $
                    simReplace (map (VarE) pro)
                    [ConsE br' (map (VarE) vl')] et
            FF -> recursive_Reduce (FLOEYcase q (((br,pro),et):rest)) ch
  | otherwise =
        if getExpType q == "ConsE" then
            let (ConsE cons exs) = q in
                case get_branch (cons,map (\x->"") exs)
                (((br,pro),et):rest) of
                    SS ((_,thisv),thiset) ->
                        branchs_reduce ch $
                        simReplace (map (VarE) thisv) (exs) thiset
        else if isVar q then let (VarE s) = q in
            case get_var_exp ch s of
                SS ex -> case ex of
```

```
                   (ConsE cid exs) ->
                        case get_branch (cid,map (\x->””) exs)
                        (((br,pro),et):rest) of
                           SS ((thisb,thisv),thiset) ->
                               branchs_reduce ch newet' where
                               newet' = simReplace
                                   (map (VarE) thisv) (exs) thiset
                           _ -> recursive_Reduce
                               (FLOEYcase q (((br,pro),et):rest)) ch
                _ -> recursive_Reduce
                   (FLOEYcase q (((br,pro),et):rest)) ch
        else case get_choice ch (E q) of
        SS (br',vl') ->
            case get_branch (br',vl') (((br,pro),et):rest) of
                   SS ((thisb,thisv),thiset) ->
                        branchs_reduce ch newet'
                     where newet' = simReplace
                        (map (VarE) thisv) (map (VarE) vl') thiset
        FF -> recursive_Reduce (FLOEYcase q (((br,pro),et):rest)) ch
branchs_reduce ch (FLOEYcontrol q (((br,pro),et):rest)) =
      case get_choice ch (C q) of
        SS (br',vl') ->
            case get_branch (br',vl') (((br,pro),et):rest) of
                   SS ((thisb,thisv),thiset) ->
                        branchs_reduce ch newet'
                     where newet' = simReplace
                        (map (VarE) thisv) (map (VarE) vl') thiset
        FF -> recursive_Reduce (FLOEYcontrol q (((br,pro),et):rest)) ch
```

```
branchs_reduce ch (FLOEYloop s bs (et,entries)) =
  FLOEYloop s bs (branchs_reduce ch et,entries)
recursive_Reduce :: ConExp -> Choices -> ConExp
recursive_Reduce (FLOEYcase q br) ch = FLOEYcase q newbr
    where newbr = foldl (\nb b -> update' nb b) [] br
        where update' nb' ((b',p'),et') = (nb'++[((b',p'),et'')])
            where et'' = if getExpType q == ”VarE” then
                             if null p'
                                 then branchs_reduce
                                     ch (simReplace [q] [ConsE b' []] et')
                                 else let (VarE s) = q in
                                     branchs_reduce (addChoice
                                     (E (ConsE b' (map (VarE) p')),
                                     (”#Assign”,[s])) ch) et'
                             else branchs_reduce
                                 (addChoice ((E q),(b',p')) ch) et'
recursive_Reduce (FLOEYcontrol q br) ch = FLOEYcontrol q newbr
    where newbr = foldl (\nb b -> update' nb b) [] br
            where update' nb' ((b',p'),et') = (nb'++[((b',p'),et'')])
                   where et'' = branchs_reduce
                    (addChoice ((C q),(b',p')) ch) et'
```

B.4 The reduction algorithm

```
{-get the semi-essentials in a control expression-}
get_semi_essential :: ConExp -> SF Definition
get_semi_essential cexp = case get_semi_essential_all cexp [] of
       SS ds -> SS $ head ds
       FF -> FF
```

```
get_semi_essential_all :: ConExp -> [String] -> SF [Definition]
get_semi_essential_all et vl = case et of
    FLOEYcase _ args -> (case intersection $ map (\((b,p),t) ->
                          get_semi_essentials t $ nub (p++vl)) args of
                                  [] -> FF
                                  (d') -> SS d')
    FLOEYcontrol _ args -> (case intersection $ map (\((b,p),t) ->
                          get_semi_essentials t $ nub (p++vl)) args of
                                  [] -> FF
                                  (d') -> SS d')
    FLOEYloop _ bs (et',_) -> (case get_semi_essentials et'
                                  (union vl (map (\(VarE i,e)->i) bs)) of
                                  [] -> FF
                                  (d') -> SS d')
    FLOEYexit _ _ -> FF
    FLOEYreturn _ -> FF
    where
      get_semi_essentials :: ConExp -> [String] -> [Definition]
      get_semi_essentials (FLOEYexit _ _) _ = []
      get_semi_essentials (FLOEYreturn _) _ = []
      --a decision can't move pass the definitions of its used variables
      get_semi_essentials (FLOEYcase q args) vl
            = if null $ intersect (getVar (E q)) vl
              then add (DF ((E q),map (\(a,b)->a) args)) se
              else se
            where se = intersection (map (\((b,p),t) ->
                          get_semi_essentials t (union (nub p) vl)) args)
```

```
                  newused = filter (\x -> x `notElem` vl) $ nub $ getVar (E q)
      --a decision can't move pass the definitions of its used variables
      get_semi_essentials (FLOEYcontrol c args) vl
            = if null $ intersect (getVar (C c)) vl
              then add (DF ((C c),map (\(a,b)->a) args)) se
              else se
            where se = intersection (map (\((b,p),t) ->
                    get_semi_essentials t (union (nub p) vl)) args)
                  newused = filter (\x -> x `notElem` vl) $ nub $ getVar (C c)
      get_semi_essentials (FLOEYloop _ bs' (et',_)) vl =
        get_semi_essentials et' $ union vl (map (\(VarE i,e)->i) bs')
      add :: Definition -> [Definition] -> [Definition]
      add q [] = [q]
      add q (q':rest)
               | q == q' = (q':rest)
               | q < q' = q:(q':rest)
               | otherwise = q':(add q rest)
      intersection :: [[Definition]] -> [Definition]
      intersection [qs] = qs
      intersection (qs:rest) = intersect qs (intersection rest)
      intersection [] = []

{-idempotent reduction-}
idem_reduce :: Decision -> [Product] -> [Branch] -> SF ConExp
idem_reduce _ _ ((_,FLOEYcase _ _):_) = FF
idem_reduce _ _ ((_,FLOEYcontrol _ _):_) = FF
idem_reduce q pds (((b,pro),FLOEYloop s bs (et,entries)):rest)
    | null rest = if any (\x->x)
```

```
                (areAlive (map (VarE) pro) $
                FLOEYloop s bs (et,entries))
               then FF
               else SS $ FLOEYloop s bs (et,entries)
   | otherwise = FF
idem_reduce q pds (((b,pro),FLOEYexit s x):[]) =
      if null $ intersect def used then SS (FLOEYexit s x)
      else FF
    where def = foldl (\x p->p:x) [] pro
         used = foldl (\u ex-> union u $ getVar (E ex)) [] x
idem_reduce q pds (((b,pro),FLOEYexit s x):rest) =
       do (s,x) <- all_same (remove_branch (b,pro) pds) rest s x
          return (FLOEYexit s x)
    where all_same [] [] s x = SS (s,x)
         all_same pds ((pd,FLOEYexit s' x'):rest) s x
            | s == s' && x == x' = all_same
             (remove_branch pd pds) rest s x
            | otherwise = FF
         all_same _ _ _ _ = FF
idem_reduce q pds (((b,pro),FLOEYreturn x):[]) =
      if null $ intersect def used then SS (FLOEYreturn x)
      else FF
    where def = foldl (\x p->p:x) [] pro
         used = getVar (E x)
idem_reduce q pds (((b,pro),FLOEYreturn x):rest) =
       do x <- all_same (remove_branch (b,pro) pds) rest x
          return (FLOEYreturn x)
    where all_same :: [Product] -> [Branch] -> Exp -> SF Exp
```

```
           all_same [] [] x = SS x
           all_same pds ((pd,FLOEYreturn x'):rest) x
               | x == x' = all_same (remove_branch pd pds) rest x
               | otherwise = FF
           all_same _ _ _ = FF

{- pulling up decisions -}
pull_up :: Decision -> [Branch] -> SF (Decision,[Branch])
pull_up q args =
    case q of
        E exp -> case get_semi_essential (FLOEYcase exp args) of
                    SS (DF (E q',products')) ->
                        SS (E q',map (\b ->
                        (b,branch_reduce ((E q'),b)
                        (FLOEYcase exp args) [])) products')
                    SS (DF (C c',products')) ->
                        SS (C c',map (\b ->
                        (b,branch_reduce ((C c'),b)
                        (FLOEYcase exp args) [])) products')
                    FF -> FF
        C con -> case get_semi_essential (FLOEYcontrol con args) of
                    SS (DF (E q',products')) ->
                        SS (E q',map (\b -> (b,branch_reduce
                        ((E q'),b) (FLOEYcontrol con args) [])) products')
                    SS (DF (C c',products')) ->
                        SS (C c',map (\b -> (b,branch_reduce
                        ((C c'),b) (FLOEYcontrol con args) [])) products')
                    FF -> FF
```

```
{-elimination factor-}
elimination_factor :: Decision -> [Branch] -> (Bool,ConExp)
elimination_factor q args =
    case idem_reduce q (map (\(p,e)->p) args) args of
       SS t -> (True,t)
       FF -> (case pull_up q args of
                     FF -> case q of
                            E exp -> (False,FLOEYcase exp args)
                            C conop -> (False,FLOEYcontrol conop args)
                     SS (q',args') -> elim q' args')
 where
   elim q' args' = case q' of
                    E exp -> (v,FLOEYcase exp args'')
                    C conop -> (v,FLOEYcontrol conop args'')
    where
      (v,args'') = elimination_factors args'
      elimination_factors [] = (False,[])
      elimination_factors ((b,t):rest) = (v' || v'', (b,t'):rest')
            where (v',rest') = elimination_factors rest
                  (v'',t') = case t of
                               FLOEYexit s e -> (False,FLOEYexit s e)
                               FLOEYreturn e -> (False,FLOEYreturn e)
                               FLOEYcase thisq thisargs ->
                                 elimination_factor (E thisq) thisargs
                               FLOEYcontrol thisc thisargs ->
                                 elimination_factor (C thisc) thisargs
```

```
{-master function of the reduction algorithm-}
reduce :: ConExp -> EntryArgLst -> ConExp
reduce t arglst = expCollect $ reduction (repeat_reduce t arglst) [] where
    reduction :: ConExp -> [Choice] -> ConExp
    reduction (FLOEYexit s x) _ = FLOEYexit s x
    reduction (FLOEYreturn x) _ = FLOEYreturn x
    reduction (FLOEYcase q args) ch = if v then t' else FLOEYcase q args'
        where args' = map (\(b,t) ->
                (b,reduction t (addChoice ((E q),b) ch))) args
              (v,t') = elimination_factor (E q) args'
    reduction (FLOEYcontrol c args) ch = if v then t' else FLOEYcontrol c args'
        where args' = map (\(b,t) ->
                (b,reduction t (addChoice ((C c),b) ch))) args
              (v,t') = elimination_factor (C c) args'
    reduction (FLOEYloop s bs (et,entries)) ch =
        case newinner of
            (FLOEYloop s' bs' (et',entries')) ->
                if null $ intersect (fst $ unzip bs) (snd $ unzip $ bs')
                then case flipped of
                        (FLOEYloop s'' _ _) ->
                            if s'' == s then newloop
                            else branchs_reduce_arglst ch arglst $
                                FLOEYloop s' bs' (flipped,[])
                        _ -> branchs_reduce_arglst ch arglst $
                                FLOEYloop s' bs' (flipped,[])
                else newloop
                    where flipped = reduction
                        (FLOEYloop s bs (et',entries++entries')) ch
            _ -> newloop
        where newinner = reduction et ch
              newloop = branchs_reduce_arglst ch arglst
                             (FLOEYloop s bs (newinner,entries))

{- reduction algorithm (incluing loop reduction) -}
reduce' :: ConExp -> EntryArgLst -> ConExp
reduce' t arglst =
    expCollect $ reduction (repeat_reduce t arglst) [] where
    reduction :: ConExp -> [Choice] -> ConExp
    reduction (FLOEYexit s x) _ = FLOEYexit s x
    reduction (FLOEYreturn x) _ = FLOEYreturn x
    reduction (FLOEYcase q args) ch =
        if v then loop_invariant t' False
        else FLOEYcase q args'
        where args' = map (\(b,t) ->
                         (b,reduction t (addChoice ((E q),b) ch))) args
              (v,t') = elimination_factor' (FLOEYcase q args')
    reduction (FLOEYcontrol c args) ch =
        if v then loop_invariant t' False
        else FLOEYcontrol c args'
        where args' = map (\(b,t) ->
                         (b,reduction t (addChoice ((C c),b) ch))) args
              (v,t') = elimination_factor' (FLOEYcontrol c args')
    reduction (FLOEYloop s bs (et,entries)) ch =
        if v then loop'
        else (FLOEYloop s bs (et,entries))
        where et' = reduction et ch
```

```
              (v,loop') = elimination_factor'
                            (FLOEYloop s bs (et',entries))

elimination_factor' :: ConExp -> (Bool,ConExp)
elimination_factor' cexp =
    let (bs,s,v,cexp') = loop_reduce [] [] cexp in
        (v,cexp')

updateSub :: ConExp -> [Sub] -> ConExp
updateSub cexp subs = let (to,from) = unzip subs in simReplace from to cexp

{- loop reduction -}
loop_reduce :: [BS] -> [Sub] -> ConExp -> ([BS],[Sub],Bool,ConExp)
loop_reduce bs subs (FLOEYexit s exs) =
    (bs,subs,False,updateSub (FLOEYexit s exs) subs)
loop_reduce bs subs (FLOEYreturn ex) =
    (bs,subs,False,updateSub (FLOEYreturn ex) subs)
loop_reduce bs subs (FLOEYcase oex oargs) =
    let (FLOEYcase ex args) = updateSub (FLOEYcase oex oargs) subs in
    case idem_reduce (E ex) (map (\(p,e)->p) args) args of
       SS t -> (bs,subs,True,t)
       FF -> case pull_up (E ex) args of
            SS (d',args') ->
                let (bs',subs',v,et) =
                    foldl (\(b,s,f,e) ((br,pro),et)-> let
                             (b',s',v',et') = loop_reduce b s et in
                             (b',s',v' || f,e++[et'])
                             ) (bs,subs,False,[]) args' in
                if v then case d' of
                 (E ex') -> (bs',subs',True,updateSub
                            (FLOEYcase ex'
                            (map (\(((b,p),t),t')->((b,p),t'))
                            (zip args' et))) subs')
                 (C q') -> (bs',subs',True,updateSub
                            (FLOEYcontrol q'
                            (map (\(((b,p),t),t')->((b,p),t'))
                            (zip args' et))) subs')
                else (bs,subs,False,FLOEYcase ex args)
            FF -> case pull_header (FLOEYcase ex args) of
                    SS cexp -> loop_reduce bs subs cexp
                    FF -> case getP args bs of
                            [] -> (bs,subs,False,FLOEYcase ex args)
                            pts ->
                                foldl (\(b,s,v,c) (d,ss) ->
                                if v then (b,s,v,c) else
                                        let (to,from) = unzip ss
                                            c' = simReplace from to c
                                            b' = updateLocalB bs ss
                                            (b'',s',v',c'')=
                                                loop_reduce b'
                                                (nub $ s++ss) c' in
                                            if v' then (b'',s',v',c'')
                                            else (b,s,v,c)
                                      ) (bs,subs,False,FLOEYcase ex args) pts
loop_reduce bs subs (FLOEYcontrol oq oargs) =
    let (FLOEYcontrol q args) = updateSub (FLOEYcontrol oq oargs) subs in
```

```
case idem_reduce (C q) (map (\(p,e)->p) args) args of
   SS t -> (bs,subs,True,t)
   FF -> case pull_up (C q) args of
        SS (d',args') ->
            let (bs',subs',v,et) = foldl (\(b,s,f,e) ((br,pro),et)-> let
                                        (b',s',v',et') =
                                        loop_reduce b s et in
                                        (b',s',v' || f,e++[et'])
                                    ) (bs,subs,False,[]) args' in
            if v then case d' of
                    (E ex') -> (bs',subs',True,updateSub
                        (FLOEYcase ex' (map (\(((b,p),t),t')->
                        ((b,p),t')) (zip args' et))) subs')
                    (C q') -> (bs',subs',True,updateSub
                    (FLOEYcontrol q' (map (\(((b,p),t),t')->
                    ((b,p),t')) (zip args' et))) subs')
            else (bs,subs,False,FLOEYcontrol q args)
        FF -> case pull_header (FLOEYcontrol q args) of
                SS cexp -> loop_reduce bs subs cexp
                FF -> case getP args bs of
                        [] -> (bs,subs,False,FLOEYcontrol q args)
                        pts ->
                            foldl (\(b,s,v,c) (d,ss) ->
                                    if v then (b,s,v,c) else
                                    let (to,from) = unzip ss
                                        c' = simReplace from to c
                                        b' = updateLocalB bs ss
                                        (b'',s',v',c'')=
                                                 loop_reduce b'
                                                 (nub $ s++ss) c' in
                                             if v' then (b'',s',v',c'')
                                             else (b,s,v,c)
                                       ) (bs,subs,False,
                                       FLOEYcontrol q args) pts
loop_reduce bs subs (FLOEYloop id binds (cexp,entries)) =
    if v then (bs',subs',True,FLOEYloop id binds' (cexp',entries))
    else (bs,subs,False,FLOEYloop id binds (cexp,entries))
    where (bs',subs',v,cexp') = loop_reduce ((id,binds):bs) subs cexp
          binds' = case (getBS bs' id) of
                    SS b -> b

{- get potential semis -}
type Protential = (Definition,[Sub])

validSubs :: [Sub] -> Bool
validSubs subs = let (x,y) = unzip subs in
                 nub x == x && nub y == y

checkDS :: [Protential] -> [Definition] -> [BS] -> [Protential]
checkDS [] _ _ = []
checkDS _ [] _ = []
checkDS ps ds bs = foldl (\p d -> p ++ (checkDS' ps d bs)) [] ds

checkDS' :: [Protential] -> Definition -> [BS] -> [Protential]
checkDS' [] d bs = []
checkDS' ((d',subs'):ps) d bs = let ps'=checkDS' ps d bs in
```

```
    case checkD d' d bs of
        FF -> ps'
        SS subs -> let newsubs = nub $ subs' ++ subs in
                    (d',newsubs):ps' else ps'


checkD :: Definition -> Definition -> [BS] -> SF [Sub]
checkD (DF (q1, pd1)) (DF (q2, pd2)) bs
    | length pd1 == length pd2 &&
        (all (\(x,y)->x==y) (zip (fst $ unzip pd1) (fst $ unzip pd2))) =
        case equalM q1 q2 of
            FF -> FF
            SS subs -> if validSubs subs then SS subs else FF
    | otherwise = FF
    where equalM :: Decision -> Decision -> SF [Sub]
          equalM (E _) (C _) = FF
          equalM (C _) (E _) = FF
          equalM (E ex1) (E ex2) = case exM ex1 ex2 of
                                     SS subs ->
                                        if null subs then FF
                                        else SS $ nub subs
                                     FF -> FF
          equalM (C (ConOp s1 ex1)) (C (ConOp s2 ex2))
             | s1 == s2  = case foldl (\f (e1,e2) ->
                                          case f of
                                            FF -> FF
                                            SS sub ->
                                              case exM e1 e2 of
                                                  FF -> FF
```

```
                                 SS sub' -> SS $ sub++sub'
                       ) (SS []) (zip ex1 ex2) of
                  SS subs ->
                   if null subs
                   then FF else SS subs
                  FF -> FF
  | otherwise = FF
exM :: Exp -> Exp -> SF [Sub]
exM (VarE x1) (VarE x2) =
  if x1 == x2 then SS []
  else case getFree bs x1 of
         SS y -> if getFree bs x2 == (SS y)
                 then SS [(VarE x1,VarE x2)]
                 else FF
         FF -> FF
exM (FunE s1 ex1) (FunE s2 ex2) =
  if s1/=s2 then FF
  else foldl (\f (e1,e2) -> case f of
           FF -> FF
           SS sub -> case exM e1 e2 of
             FF -> FF
             SS sub' -> SS $ sub++sub'
           ) (SS []) (zip ex1 ex2)
exM (ConsE s1 ex1) (ConsE s2 ex2) =
  if s1/=s2 then FF
  else foldl (\f (e1,e2) -> case f of
           FF -> FF
           SS sub -> case exM e1 e2 of
```

```
             FF -> FF
             SS sub' -> SS $ sub++sub'
           ) (SS []) (zip ex1 ex2)
exM (ElemE s1 ex1) (ElemE s2 ex2) =
  if s1/=s2 then FF
  else foldl (\f (e1,e2) -> case f of
           FF -> FF
           SS sub -> case exM e1 e2 of
             FF -> FF
             SS sub' -> SS $ sub++sub'
           ) (SS []) (zip ex1 ex2)
exM (BinE s1 ex1 ex2) (BinE s2 ex3 ex4) =
  if s1/=s2 then FF
  else foldl (\f (e1,e2) -> case f of
           FF -> FF
           SS sub -> case exM e1 e2 of
             FF -> FF
             SS sub' -> SS $ sub++sub'
           ) (SS []) [(ex1,ex3),(ex2,ex4)]
exM (BraceE ex1) (BraceE ex2) = exM ex1 ex2
exM (NegE ex1) (NegE ex2) = exM ex1 ex2
exM (IntE i1) (IntE i2) = if i1==i2 then SS [] else FF
exM (RealE i1) (RealE i2) = if i1==i2 then SS [] else FF
exM (CharE i1) (CharE i2) = if i1==i2 then SS [] else FF
exM ex1 ex2 = FF
```

B.5 Invariant code removal

```
{-pulling out semi-essentials from loops-}
```

```
loop_semi_invariant :: ConExp -> ConExp
loop_semi_invariant (FLOEYexit s exs) = FLOEYexit s exs
loop_semi_invariant (FLOEYreturn ex) = FLOEYreturn ex
loop_semi_invariant (FLOEYcase ex args) =
    FLOEYcase ex (map (\((b,p),t)->
    ((b,p),loop_semi_invariant t)) args)
loop_semi_invariant (FLOEYcontrol q args) =
    FLOEYcontrol q (map (\((b,p),t)->
    ((b,p),loop_semi_invariant t)) args)
loop_semi_invariant (FLOEYloop id bs (t,ens)) =
    let t' = loop_semi_invariant t in
    case pull_up_inloop (FLOEYloop id bs (t',ens)) of
        FF -> (FLOEYloop id bs (t',ens))
        SS et -> loop_semi_invariant et

{-pulling out all invariants from loops-}
loop_semi_invariant :: ConExp -> ConExp
loop_invariant :: ConExp -> Bool -> ConExp
loop_invariant cexp f
    | f = let maxSub = (getMaxSub cexp) + 1 in
          fst $ loop_all_invariant cexp [] maxSub
    | otherwise = loop_semi_invariant cexp

loop_all_invariant :: ConExp -> [ConExp] -> Int -> (ConExp,Int)
loop_all_invariant (FLOEYexit s exs) _ i = (FLOEYexit s exs,i)
loop_all_invariant (FLOEYreturn ex) _ i = (FLOEYreturn ex,i)
loop_all_invariant (FLOEYcase ex args) ls i =
    let (args',i') = foldl (\(a',i') ((b,p),t) ->
                           let (t',i'') = loop_all_invariant t ls i' in
                           (a'++[((b,p),t')],i'')
                           ) ([],i) args
    in (FLOEYcase ex args',i')
loop_all_invariant (FLOEYloop id bs (t,ens)) ls i =
    case pull_up_inloop (FLOEYloop id bs (t',ens)) of
        SS et -> loop_all_invariant et ls' i
        FF -> let (t'',i'') =
            reLoop t' (map (\(VarE x,y)->x) bs) [] i' in
            (branchs_reduce_arglst []
            M.empty $ FLOEYloop id bs (t'',ens),i'')
    where (t',i') = loop_all_invariant t ((FLOEYloop id bs (t,ens)):ls) i
          ls' = (FLOEYloop id bs (t',ens)):ls
          reLoop :: ConExp -> [String] -> Choices -> Int -> (ConExp,Int)
          reLoop (FLOEYexit s exs) vl chs i
            | null chs = (FLOEYexit s exs,i)
            | otherwise =
                case find (\(FLOEYloop id bs (t,ens)) -> id == s) ls' of
                    Just t ->
                        let (t'',i'') = diffName t' i'
                            (t',i') =
                                diffLoop
                                (loopbs_reduce $
                                branchs_reduce_arglst chs M.empty $
                                updateHeader t) i M.empty
                        in (t'',i'')
            where updateHeader (FLOEYloop id bs (t,ens)) =
                  FLOEYloop id
        (map (\((a,b),y)->(a,y)) (zip bs exs)) (t,ens)
reLoop (FLOEYreturn ex) _ _ i = (FLOEYreturn ex,i)
reLoop (FLOEYcase ex args) vl chs i =
  let f = null $ intersect (getVar (E ex)) vl
      (args',i') = foldl (\(a',i) ((b,p),t)->
                  let chs' =
                     if f then (E ex, (b,p)):chs
                     else chs
                      (t',i') =
                      reLoop t (nub $ p++vl) chs' i
                  in (a'++[((b,p),t')],i')
                  ) ([],i) args
  in (FLOEYcase ex args',i')
reLoop (FLOEYcontrol q args) vl chs i =
  let f = null $ intersect (getVar (C q)) vl
      (args',i') = foldl (\(a',i) ((b,p),t)->
                         let chs' =
                             if f then (C q, (b,p)):chs
                             else chs
                             (t',i') =
                             reLoop t (nub $ p++vl) chs' i
                         in (a'++[((b,p),t')],i')
                         ) ([],i) args
  in (FLOEYcontrol q args',i')
reLoop (FLOEYloop id bs (t,ens)) _ _ i =
  (FLOEYloop id bs (t,ens),i)
```

Bibliography

A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23 (3):488–501, 1976. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321958.321970.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, editors. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007. ISBN 0-321-48681-1. URL `http://www.loc.gov/catdir/toc/ecip0618/2006024333.html`.

F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971.

J. R. Allen and K. Kennedy. Automatic loop interchange. In *SIGPLAN84*, June 1984.

Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN 0-521-58274-1.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/197405.197406.

Chandan Kumar Behera and Pawan Kumar. An improved algorithm for loop dead optimization. *SIGPLAN Notices*, 40(11):18–28, 2005. URL `http://doi.acm.org/10.1145/1107541.1107545`.

G. Bilardi and K. Pingali. The static single assignment form and its computation, 1999. URL `citeseer.ist.psu.edu/bilardi99static.html`.

Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/765568.765573.

Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 159–170, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258930.

Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 159–170, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178257.

Briggs, P. *Register Allocation via Graph Coloring*. PhD thesis, Dept. of Computer Science, Rice Univ., 1992.

David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Not.*, 21(7):152–161, 1986. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/13310.13327.

David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Not.*, 39(4):155–166, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989412.

P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. *SIGPLAN Not.*, 24(7):246–257, 1989. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/74818.74840.

Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 273–286, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258940.

Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *Intermediate Representations Workshop*, pages 35–49, 1995.

Cocke, J. Global common subexpression elimination. *SIGPLAN Notices*, 5:20–24, 1970.

J. R. B. Cockett and J. A. Herrera. Decision tree reduction. *J. ACM*, 37(4):815–842, 1990. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/96559.96576.

Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. ISBN 0-262-03141-8.

Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in ssa form. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 36–45, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. doi: http://doi.acm.org/10.1145/155090.155094.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/115372.115320.

D. M. Dhamdhere. Practical adaption of the global optimization algorithm of morel and renvoise. *ACM Trans. Program. Lang. Syst.*, 13(2):291–294, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/103135.214520.

J. J. Dongarra and A. R. Jinds. Unrolling loops in Fortran. *Software—Practice and Experience*, 9(3):219–226, March 1979.

John R. Ellis. *Bulldog: a compiler for VLSI architectures*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-05034-X.

Lawrence Feigen, David Klappholz, Robert Casazza, and Xing Xue. The revival transformation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 421–434, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: http://doi.acm.org/10.1145/174675.178043.

Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/200994.201003.

M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3): 367–375, 1974. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321832.321835.

Matthew S. Hecht. *Flow Analysis of Computer Programs*. New York: North Holland, 1977.

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *PLDI*, pages 224–234, 1992.

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 147–158, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: http://doi.acm.org/10.1145/178243.178256.

S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

D. Neel and M. Amirchahy. Removal of invariant statements from nested-loops in a single effective compiler pass. In *Proceedings of the conference on Programming languages and compilers for parallel and vector machines*, pages 87–96, 1975.

Diego Novillo. Tree SSA: A new optimization infrastructure for GCC. In Andrew J. Hutton, Stephanie Donovan, and C. Craig Ross, editors, *Proceedings of the GCC Developers Summit May 25–27, 2003, Ottawa, Ontario Canada*, pages 181–193, 2003.

Diego Novillo. Design and implementation of Tree SSA. In Andrew J. Hutton, Stephanie Donovan, and C. Craig Ross, editors, *Proceedings of the GCC Developers Summit ,June 2nd-4th, 2004, Ottawa, Ontario, Canada*, pages 119–130, 2004.

David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/7902.7904.

A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on ssa form. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 15–25, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: http://doi.acm.org/10.1145/277650.277656.

Sethi and Ullman. The generation of optimal code for arithmetic expressions. *JACM: Journal of the ACM*, 17, 1970.

Sachin Shaw and Pawan Kumar. Loop-dead optimization. *SIGPLAN Not.*, 40(2):33–40, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1052659.1052665.

Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ;-nodes. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 62–73, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: http://doi.acm.org/10.1145/199448.199464.

Stanford SUIF Compiler Group. SUIF: A parallelizing & optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

Ullman. A fast algorithm for the elimination of common subexpressions. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1972.

Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Henk Corporaal, and Francky Catthoor. Advanced copy propagation for arrays. *SIGPLAN Not.*, 38(7):24–33, 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/780731.780736.

Thomas VanDrunen and Antony L. Hosking. Value-based partial redundancy elimination. In *13th International Conference on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2004.

Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/103135.103136.

Paul Damian Wells. A universal intermediate representation for massively parallel software development. *SIGPLAN Notices*, 39(5):48–57, 2004. URL http://doi.acm.org/10.1145/997140.997145.

M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM. ISBN 0-89791-341-8. doi: http://doi.acm.org/10.1145/76263.76337.

Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Efficient and effective branch reordering using profile data. *ACM Trans. Program. Lang. Syst.*, 24(6):667–697, 2002. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/586088.586091.