Exception Analysis in MLton

Master thesis in computer science

Alexander Bjerremand Hansen — 20062813

Supervisor: Anders Møller & Jan Midtgaard 2011.09.15

Datalogisk Institut
Aarhus Universitet

Abstract

This thesis examines the possibility of further optimizing the code generated by the industrial strength Standard ML compiler MLton. By studying the code generated by the compiler we identify dead code and propose two analyses that will detect and remove this dead code. The first improvement will remove general dead exceptions handlers and the second will remove overflow checks on integer arithmetic where overflow cannot happen. We extract a BNF of the Static Single-Assignment form used internally in MLton and use this BNF to formulate two analyses by use of constraint rules. The first analysis collects sets of exception constructors possibly represented by each variable and the second is a flow-sensitive integer interval analysis. Both analyses are implemented in Standard ML and integrated into the MLton compiler as additional optimization passes. Whereas the first analysis is unsuccessful in optimizing real-world programs, the second analysis is able to remove 22% of the overflow checks in the MLton benchmark suite. The binary size of the programs are decreased overall (0%-10%) and the run time of arithmetic heavy programs are decreased significantly - in some cases up to 27%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Definition of problem

Program optimization is an important aspect of a compiler. The Standard ML compiler MLton is a whole program optimizing compiler that focuses on generating fast and small binary executables. This thesis explores the possibility of performing even better optimizations in the MLton compiler by identifying and removing dead code induced by unused exception handlers.

Our thesis is twofold: that MLton, the industrial strength compiler developed over the last 14 years, does not produce optimal code regarding exception handlers, and that we are able to identify and remove dead exception handlers using standard static analysis tools.

By using static analysis we expect to identify and remove unused exception handlers in real-world programs and thereby reduce the size and run time of the generated binaries.

1.2 Method and overview

The method we will use to identify the dead exception handlers is manual inspection. By studying the intermediate static single-assignment (SSA) code generated by the MLton compiler we are able to identify dead exception handlers that can be removed. One aspect is to detect the dead code by manual inspection another aspect is providing the compiler with enough information to detect the dead code as well. We explore what information the compiler needs and how it can use this information to detect and remove the dead code.

The thesis is split into two different analyses; the first analysis explores the possibility of detecting general exception handlers that will not be taken.

By removing handlers for exceptions that are not thrown the resulting binaries are expected to decrease in size. The second analysis is based on a thorough study of the intermediate SSA code of real-world programs where a key observation is made; by tracking the possible integer values for each variable in a *flow-sensitive* analysis the compiler will be able to remove overflow checks on integer arithmetic at compile time. The Standard ML specification [Milner et al., 1997] requires all integer arithmetic to check for overflow and act accordingly, so this is the default behavior in the SSA code. By not checking for overflow where we know it cannot occur we can reduce the size of the binaries and hopefully reduce the run time of arithmetic heavy Standard ML code and still adhere to the specification.

Each improvement starts with a couple of examples that show some dead code and discusses how this might be detected by the compiler. After the examples we introduce a formal notation for capturing the behavior of the needed analysis leading to an actual implementation. The implementation is done in Standard ML and integrated into the existing MLton compiler as additional optimization phases. The focus of the implementation is to measure how well the analysis is at detecting and removing dead code. The success criteria is the effect it has on the resulting binaries regarding size and run time.

The first chapter after the introduction gives an overview of the MLton compiler and the intermediate language in SSA form we are working with. This chapter also introduces a BNF of the SSA intermediate language that will be used throughout the rest of the thesis. Chapter three presents the first improvement starting with examples followed by a description of an existing analysis that will be expanded on. The transformation that the compiler will perform based on the collected information is then discussed together with a description of the actual implementation. Lastly the improvement is benchmarked and we discuss how well the improvement works and how it relates to our hypotheses.

The fourth chapter presents the second improvement with a structure almost identical to the first improvement. The report ends with a chapter covering related work in both exception analysis and interval analysis. We will conclude on the findings of this thesis and discuss the results obtained from the experiments.

Chapter 2

The MLton compiler

```
MLton is an open-source, whole-program, optimizing
Standard ML compiler[1]
```

During the compilation of ML programs several intermediate languages are used internally by the MLton compiler. These languages primarily serve as steps in the translation from ML source code to machine code where each translation performs one or more transformations on the code from the previous layer. An overview of the MLton compiler is shown in figure 2.1, where each of the boxes is an intermediate language and the arrows between them are the translations.

The input to the compiler is the ML source code where `MLLex` and `MLYacc` are used for lexing and parsing, respectively, resulting in an abstract syntax tree (AST). On the AST the elaborate pass performs type inference and type checking and then defunctorization eliminates all module-level constructs giving a higher-order, polymorphic language with nested case-patterns called CoreML.

The CoreML language is then translated into an explicitly-typed language called XML (eXplicitly-typed ML) by annotating each expression with its type. This translation also performs match compilation which flattens case-patterns and adds default cases to make a match exhaustive. Furthermore this pass lifts all datatype declarations to the top level.

XML is then translated into a simply-typed language called SXML. This is done by monomorphisation that removes all uses of polymorphism by duplicating each polymorphic value and datatype declaration for each type it is used with. Higher-order functions are then removed by using

[1]http://mlton.org

closure conversion (defunctionalization) on the SXML code resulting in a simply-typed, first-order language called SSA (Static Single-Assignment).

From SSA the code is translated three times before ending as machine code. These translations are not described here since they are on a lower level and deal with register allocation, stack-sizes, etc.

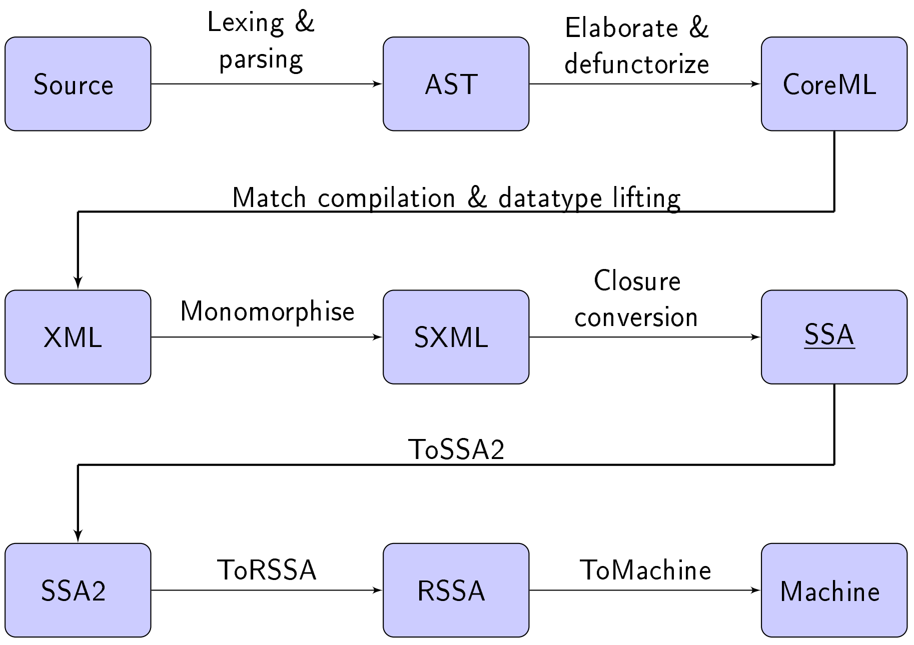

Figure 2.1: MLton compiler overview.

The different properties of the intermediate languages make some static analyses easier on one intermediate language than others. One of the first analyses performed is type checking since the translations later in the process are defined on well-typed programs only. Each intermediate step has one or more optimization passes that are performed before further translation is performed.

As described, one of the translation steps of the compiler is a closure conversion with defunctionalization that produces code in SSA form. This intermediate form is interesting since it allows for easy optimization and it is here that the MLton compiler performs most of the optimizations.

2.1 Static Single-Assignment form

Definition 1. *In static single-assignment form a variable has only one definition in the program text.*

SSA form is a common intermediate representation used for optimization in modern compilers [Appel, 1998, chap. 19]. As the *single-assignment* part of the name suggests, each variable has only one assignment/definition in the program text. The *static* part emphasizes that the single-assignment part is a static property and not a dynamic one since an assignment can be placed in a function that is called several times.

Dataflow analysis is made easier when each variable is reduced to a single let-binding that dominates its uses. Furthermore the space required to represent a def-use chain data structure is linear in the size of the program in most real-world applications[Appel, 1998, page 427].

A simplified grammar for the SSA form in MLton is shown in figure 2.2. The simplification consists of removing constructs that are not relevant for our purposes.

A program in SSA form in MLton consists of 3 major parts; collections of *datatype declarations*, *global statements*, and *functions*. One of the functions is explicitly marked as a "main" function and is the entry for the executable binary. Each function consists of a number of basic blocks that each consists of a number of statements ending with a control transfer.

An interesting observation is that arithmetic occurs in a construct with an explicit `Overflow` handler attached and that the exception is implicitly raised by the primitive arithmetic operations. Since the construct that performs possibly overflowing arithmetic is a control transfer the control-flow on the primitive `Overflow` is explicit. The handler for the exception will then explicitly raise the `Overflow` exception with the `raise` control transfer. Another way to perform arithmetic in SSA is to use primitive function definitions that take the operands as parameter and returns the result of the calculation.

As described in the compilation process of MLton and shown in the BNF many of the higher-level constructs in ML have been eliminated at this point in the compilation, e.g., polymorphic values and datatype declarations, higher-order functions, structures, functors. This makes SSA a relatively simple intermediate language that is simply-typed and first-order with complete type information on variables and functions.

$$
\begin{array}{rl}
\text{Program} & P ::= \; D^* \; F^* \\
\text{Function} & F ::= \; \texttt{fun f}(x^*) = \texttt{l}() \; B^* \\
\text{Block} & B ::= \; \texttt{l}(x^*) \; S \\
\text{Block statements} & S ::= D \; S \\
 & \quad \mid \texttt{return } x \\
 & \quad \mid \texttt{raise } x \\
 & \quad \mid \texttt{f}(V^*) \; \texttt{NonTail \{cont=l,handler=}H\texttt{\}} \\
 & \quad \mid \texttt{f}(V^*) \; \texttt{Tail} \\
 & \quad \mid \texttt{l}(V^*) \\
 & \quad \mid \texttt{l}(A) \; \texttt{Overflow => l} \\
 & \quad \mid \texttt{case } x \texttt{ of } \; (V \texttt{ => l})^* \; \texttt{|} \; \texttt{\_ => l} \\
\text{Definition} & D ::= \; x \leftarrow V \mid x \leftarrow \texttt{p}(x^*) \mid x \leftarrow \texttt{c}(x^*) \\
\text{Arithmetic} & A ::= x \; \texttt{op} \; x \\
\text{Handler} & H ::= \texttt{l} \mid \texttt{Caller} \\
\text{Value} & V ::= x \mid v \\
\end{array}
$$

$$
\begin{array}{rl}
\text{where} & x \in \text{variables}, \; v \in \text{literal value} \\
 & \texttt{l} \in \text{labels}, \; \texttt{p} \in \text{primitive function} \\
 & \texttt{c} \in \text{datatype constructor}, \; \texttt{f} \in \text{function label} \\
 & \texttt{op} \in \{+, -, *, /\} \\
\end{array}
$$

Figure 2.2: Simplified BNF of the SSA form in MLton.

To give an example of the generated SSA code in MLton we translate the simple program in Standard ML that calculates Fibonacci numbers in listing 2.1. This simple example uses recursion, pattern matching, and has some simple integer arithmetic. A reachable call to the `fib` function is omitted from the SML example but is necessary when compiling with MLton or else `fib` will be marked as unused and removed by an optimization pass.

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

Listing 2.1: Fibonacci in SML

Compiling the code above with MLton and saving the intermediate SSA output produces the snippet in listing 2.2 (the comments are mine).

```
fun fib_0 (x_1219: word32): = L_1408 ()
  L_1408 ()
    case x_1219 of      (* pattern match on arg *)
      0x0 => L_4218 | 0x1 => L_4217 | _ => L_1409
  L_4218 ()
    return global_2
  L_4217 ()
    return global_1
  L_1409 ()                         (* third case *)
    L_1411 (x_1219 - global_1) Overflow => L_4216 ()
  L_4216 ()
    raise (global_28)
  L_1411 (x_1220: word32)     (* first rec call *)
    fib_0 (x_1220) NonTail {cont = L_1413, handler =
        Caller}
  L_1413 (x_1221: word32)
    L_1415 (x_1219 - global_5) Overflow => L_4216 ()
  L_1415 (x_1222: word32)    (* second rec call *)
    fib_0 (x_1222) NonTail {cont = L_1417, handler =
        Caller}
  L_1417 (x_1223: word32)
    L_4047 (x_1221 + x_1223) Overflow => L_4216 ()
  L_4047 (x_1224: word32)
    return x_1224
```

Listing 2.2: Fibonacci snippet in SSA

This is only a snippet of the SSA output. Since MLton is a whole-program compiler pieces of the standard library that has not been optimized away are included in the output together with a designated `main` function that calls `fib_0`.

In the SSA code above there are no explicit assignments of variables but there is some implicit value-flow to the formal parameters of blocks, e.g., the return value from the non-tail call to `fib_0` in block `L_1415` is bound to the parameter of the continuation block (`L_1417`) called `x_1223`.

It should also be noted that the arithmetic performed in the transfers of the blocks `L_1409`, `L_1413`, and `L_1417` use the `Overflow` handler called `L_4216` that simply raises the variable `global_28` which is defined as the explicit `Overflow` exception.

The control-transfers in the basic blocks represents the control-flow graph of the program and the optimizations performed in the SSA phase use this information.

Chapter 3

First improvement

In this chapter we study a first example that the existing analyses of MLton cannot optimize and we discuss an existing analysis that can be extended to handle this example as well.

First we describe a situation where some dead code exists at compile time. We give some examples of SSA code containing obvious dead code and discuss what information is required to detect it. Next we describe an existing analysis operating on SSA code and discuss how to extend this analysis with some additional information that enables detection of the dead code from the example.

With this information we are able to perform some program simplifications which is the goal of the analysis. After the transformations we describe the implementation of the improvement as an optimization pass and how it fits together with the existing analysis. The analysis is then run on the examples given and the MLton benchmark suite and the results are compared to those without the improved analysis regarding binary size, compile time, and run time.

3.1 Examples

This first improvement is mostly based on an artificial example. The improvement is relatively simple so it is included and evaluated upon later in this chapter.

The following is a simple SML program that is a small variation of listing 2.1 in chapter 2 that the MLton compiler is able to optimize:

```
exception E1

fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)

val k = (fib 42) handle E1 => (print "error"; 0)
```

Listing 3.1: Exception example 1

After the optimizations in the SSA phase of the compiler there are no traces left of the exception `E1` since it is never used. This example illustrates how unused exceptions are removed after they have been marked as unused by one of the optimizations.

If we extend this example with an additional exception we are able to find a case where the compiler does not remove a handler that cannot be fired:

```
exception E1
exception E2

fun fib1 ~1 = raise E1
  | fib1 0 = 0
  | fib1 1 = 1
  | fib1 n = fib1(n-1) + fib1(n-2)

fun fib2 ~1 = raise E2
  | fib2 0 = 0
  | fib2 1 = 1
  | fib2 n = fib2(n-1) + fib2(n-2)

val l = (fib1 42) handle E1 => 0
val k = (fib2 42) handle E1 => (print "E2"; 0)
```

Listing 3.2: Exception example 2

This example contains two functions that each can raise a different exception. If one function is called with a handler for the exception of the other function the handler is not removed. Although this example is artificial it will be a straight forward analysis to catch this case. Benchmarks will then be able to tell if this scenario actually occurs in real-world programs and if it has an impact on the binary size of the programs compiled with MLton.

In the SSA form in MLton all exceptions are constructors of the same datatype called *exn*. The earlier type checking phase guarantees that variables that are raised with the `raise` keyword have the type *exn*. This is not directly stated in the BNF in the production for the raise transfer, so we instead write:

$$S ::= \ldots \mid \texttt{raise}\ \ x \mid \ \ldots\ , \text{ where } x : exn$$

This is understood as the raise transfer can raise a variable x that has the type *exn*. The type rule is implicit in the BNF. This means that there cannot be any constructor call in a raise statement, so the exception that is raised has already been constructed and bound to a variable in a single assignment.

To understand how handlers work in SSA form in MLton we have translated the SML example from listing 3.2 and saved the intermediate SSA output. It is the handler in the call on line 15 that we wish to detect as dead and eliminate since `fib2` cannot raise the `E1` exception.

The call to `fib2` on line 15 in listing 3.2 is translated to the following SSA code:

```
 L_131 ()
    fib2_0 (global_173) NonTail {cont = L_132, handler =
         Handle L_133}
```

Listing 3.3: Call to `fib2` in SSA

The handler for this call is the label called `L_133` and the code for that label in SSA form is:

```
L_133 (x_185: exn)
    case x_185 of
      E1_0 => L_143 | _ => L_144
```

Listing 3.4: Handler to the `fib2` call

We see here that the handler is a case transfer where the single parameter is deconstructed and matched against the `E1_0` constructor that corresponds to the `E1` exception from the example. The default case points to `L_144` which is the road to the top-level handler.

If we can guarantee that the variable `x_185` can never contain exception values constructed with `E1_0` we can remove the corresponding case and only the default case will remain. Simplifications to the program that this analysis can enable are described in a later section.

To make that guarantee about `x_185` we take a look at the `fib2_0` function in SSA form:

```
fun fib2_0 (x_412: word32) = L_288 ()                        1
  L_288 ()                                                   2
    case x_412 of                                            3
      0xFFFFFFFF => L_291 | 0x1 => L_290 | 0x0 => L_289      4
          | _ => L_292
  L_291 ()                                                   5
    raise (global_409)                                       6
  L_290 ()                                                   7
    return global_15                                         8
  L_289 ()                                                   9
    return global_14                                         10
  L_292 ()                                                   11
    L_293 (x_412 - global_15) Overflow => L_294 ()           12
  L_294 ()                                                   13
    raise (global_16)                                        14
  L_293 (x_413: word32)                                      15
    fib2_0 (x_413) NonTail {cont = L_295, handler =          16
        Caller}
  L_295 (x_414: word32)                                      17
    L_296 (x_412 - global_18) Overflow => L_294 ()           18
  L_296 (x_415: word32)                                      19
    fib2_0 (x_415) NonTail {cont = L_297, handler =          20
        Caller}
  L_297 (x_416: word32)                                      21
    L_298 (x_414 + x_416) Overflow => L_294 ()               22
  L_298 (x_417: word32)                                      23
    return x_417                                             24
```

Listing 3.5: The `fib2` function in SSA

By enumerating all the raise transfers in the `fib2_0` function we see that it has the possibility of raising the variables `global_409` and `global_16` on lines 6 and 14, respectively. By definition 1 the values of the two global variables are never changed from their definition and they are defined as follows:

```
global_16: exn = Overflow_0 ()
global_409: exn = E2_0 ()
```

Listing 3.6: The two relevant global definitions

These are constructor application definitions. One is the `Overflow` exception that is raised explicitly after an overflow occurs in a arithmetic transfer and the other is the expected `E2` exception. If we can make the analysis aware that the variables that can be raised from the function `fib2_0` contains the two globals we can use this information to transform the program. These transformations will be described after an improvement to an existing analysis has been formulated.

3.2 Existing analysis

The MLton compiler has 18 different optimization passes on the SSA form. These range from constant propagation over inlining to tail-call optimization. Also included in these optimizations are dead code elimination in various forms. Dead code elimination is a necessity in a whole program compiler since the standard library is simply appended to the code to be compiled and it is rarely the case that all of the standard library is used and therefore some of it can be eliminated.

Most of the rough dead code elimination has been done once the compiler reaches the SSA optimization pass so the dead code elimination performed here is either to remove unused code left behind by the other optimizations or to utilize the properties of the SSA form to perform more sophisticated dead code elimination.

One of the optimization passes on the SSA form is called `RemoveUnused` and its purpose is to mark constructs in the code as either *used* or *unused* and subsequently remove those that are marked unused. This optimization will obviously have to be conservative but also aggressive with the goal of reducing the resulting binary in size.

To understand this analysis better, we formulate it as extendable constraints that encapsulate its behavior. The implementation of the analysis will also be described briefly to give an understanding of the algorithm used to move the data flow from the constraints around.

3.2.1 Constraints

In this section we describe the data flow constraints that the existing analysis, `RemoveUnused`, generates. First we define lattices that are used to model the information we associate to the different constructs in the program, e.g., variables and functions.

We define a lattice V that associates each variable Var in the program to either the value *used* or *unused*:

$$V : \mathit{Var} \rightarrow \{\mathit{used}, \mathit{unused}\}$$

This function lattice relates variables to a simple two point lattice that models if a variable is used in the program or not. The set has a partial ordering where *unused* $\sqsubseteq$ *unused* $\sqsubseteq$ *used* $\sqsubseteq$ *used* and the functions are ordered pointwise. The definition of a variable being used will be given in the constraints shortly. Initially the analysis marks all variables in the program as *unused*.

Next we have two different names for the same lattice structure that associates SSA functions F to powersets of variables:

$$F_{rai} : F \rightarrow \wp(\mathit{Var})$$
$$F_{ret} : F \rightarrow \wp(\mathit{Var})$$

The sets of variables are ordered by inclusion. F_{rai} and F_{ret} are ordered pointwise. The names F_{rai} and F_{ret} corresponds to the variables that can be raised or returned, respectively, by a given function. For each function in the program the variables that can be raised and returned from that particular function are modeled by these lattices.

The constraints that model the flow of the program are written in a form where a premise is above the line and the constraints to be generated if the premise is fulfilled is below the line. The constraints are not generated and solved but rather used as a tool to formally express the behavior of the analysis. The constraints can be extended to capture the behavior of the improvements that will be introduced later. The data flow is expressed with the constraints but the implementation of this data flow does not use constraints directly.

To be able to distinguish between label/function definitions and calls in the BNF the following convention is used: When a label/function definition is mentioned it has capital letter parameters, e.g., $\mathtt{l}(X_1, ..., X_n)$, and calls have lower case letters as parameters, e.g., $\mathtt{l}(x_1, ..., x_n)$.

The first constraint is for the `Goto` construct:

$$\frac{\mathtt{l}(X_1, ..., X_n) \in P \qquad \mathtt{l}(x_1, ..., x_n) \in P}{V[\![x_1, ..., x_n]\!] \sqsubseteq V[\![X_1, ..., X_n]\!]}\ \textsc{Goto}$$

This rule is read as follows: If a label `l` exists in the program as a label definition with the formal parameters $X_1, ..., X_n$ and a call to the same label `l` exists in the program as a call with the actual parameters $x_1, ..., x_n$ then information flows in the V lattice from the actual parameters to the formal parameters.

Two more rules for control transfers are the return and raise rules. These rules are almost identical except they use different lattices to carry information.

$$\frac{(\texttt{return}\ x) \in P \qquad \mathit{func}(\texttt{return}\ x) = \texttt{f}}{F_{ret}[\![\texttt{f}]\!] \supseteq \{x\}}\ \textsc{Return}$$

$$\frac{(\texttt{raise}\ x) \in P \qquad \mathit{func}(\texttt{raise}\ x) = \texttt{f}}{F_{rai}[\![\texttt{f}]\!] \supseteq \{x\}}\ \textsc{Raise}$$

Both these rules use a helper function called *func* that is used to name the SSA function in which a statement exist. The signature of this function is $B \rightarrow F$, so it takes a block statement and returns a function. The resulting function is only used for its name.

The return and raise rules specifies that the F_{ret} and F_{rai} lattices at least include the variables that are returned or raised, respectively. This information is then used in the following non-tail call rule where the handler is another label in the program:

$$\frac{\begin{array}{c}\texttt{f}(X_1, ..., X_n) \in P \qquad \texttt{l}'(X_r) \in P \qquad \texttt{l}''(X_h) \in P \\ (\texttt{f}(x_1, ..., x_n)\ \texttt{NonTail}\ \texttt{\{cont=l}',\texttt{handler=l}''\texttt{\}}) \in P \\ F_{ret}[\![\texttt{f}]\!] = \{r_1, ..., r_m\} \qquad F_{rai}[\![\texttt{f}]\!] = \{h_1, ..., h_k\}\end{array}}{\begin{array}{c}V[\![x_1, ..., x_n]\!] \sqsubseteq V[\![X_1, ..., X_n]\!] \\ V[\![r_1]\!] \sqcup ... \sqcup V[\![r_m]\!] \sqsubseteq V[\![X_r]\!] \\ V[\![h_1]\!] \sqcup ... \sqcup V[\![h_k]\!] \sqsubseteq V[\![X_h]\!]\end{array}}\ \textsc{NonTail call - label}$$

The above rule for non-tail calls is more complex since it has to deal with continuation and exception flow. Like the goto rule information flows from the actual parameters of the call to the formal parameters of the called function. Next a join of all the possible returned variables from the called function flows into the parameter of the continuation label. Similarly with the raised variables, a join of all the possible raised variables from the called function flows into the parameter of the handler label. There are two other productions of calls in the BNF and these are tail-calls and non-tail-calls where the handler is the calling function. The rule for the `Caller` handler:

$$\frac{\begin{array}{c}\mathtt{f}(X_1,...,X_n) \in P \qquad \mathtt{l}'(X_r) \in P \\ (\mathtt{f}(x_1,...,x_n)\ \mathtt{NonTail\ \{cont=l',handler=Caller\}}) = S \\ S \in P \qquad \mathit{func}(S) = \mathtt{f}' \qquad F_{ret}[\![\mathtt{f}]\!] = \{r_1,...,r_m\}\end{array}}{\begin{array}{c}V[\![x_1,...,x_n]\!] \sqsubseteq V[\![X_1,...,X_n]\!] \\ V[\![r_1]\!] \sqcup ... \sqcup V[\![r_m]\!] \sqsubseteq V[\![X_r]\!] \\ F_{rai}[\![\mathtt{f}']\!] \supseteq F_{rai}[\![\mathtt{f}]\!]\end{array}} \text{ NonTail call - caller}$$

The difference here is that the variables raised by the called function do not flow to the argument of a handler but instead to the set of raised variables for the function the call is in. This is the same way the non-tail call rule works but here the returned variables are handled the same way:

$$\frac{\begin{array}{c}\mathtt{f}(X_1,...,X_n) \in P \qquad S \in P \\ \mathit{func}(S) = \mathtt{f}' \qquad (\mathtt{f}(x_1,...,x_n)\ \mathtt{Tail}) = S\end{array}}{\begin{array}{c}V[\![x_1,...,x_n]\!] \sqsubseteq V[\![X_1,...,X_n]\!] \\ F_{ret}[\![\mathtt{f}']\!] \supseteq F_{ret}[\![\mathtt{f}]\!] \\ F_{rai}[\![\mathtt{f}']\!] \supseteq F_{rai}[\![\mathtt{f}]\!]\end{array}} \text{ Tail call}$$

The tail-call rule has both its return and raise variables passed to the caller of the function. Until now the rules have been about the flow of information in the V lattice but the simple two-point lattice is never raised from unused to used. This is accomplished by the following rules.

$$\frac{(X \leftarrow x) \in P}{V[\![x]\!] = \mathit{used}} \text{ Definition}$$

$$\frac{(X \leftarrow \mathtt{p}(x_1,...,x_n)) \in P}{V[\![x_1]\!] = ... = V[\![x_n]\!] = \mathit{used}} \text{ Primitive Definition}$$

$$\frac{(\mathtt{l}(x_1\ \mathtt{op}\ x_2)\ \mathtt{Overflow\ =>\ l'}) \in P}{V[\![x_1]\!] = V[\![x_2]\!] = \mathit{used}} \text{ Arithmetic}$$

$$\frac{(\mathtt{case}\ x\ \mathtt{of}\ V_1\mathtt{=>l}_1\ \mathtt{|}\ ...\ \mathtt{|}\ V_n\mathtt{=>l}_n\ \mathtt{|}\ \mathtt{\_=>l'}) \in P}{V[\![x]\!] = \mathit{used}} \text{ Case transfer}$$

The analysis marks variables as used in four different places; when a variable is: (1) used in the expression of a let binding, (2) used as an parameter to a primitive function, (3) used as a parameter in a arithmetic transfer, or (4) deconstructed in a case transfer. The rules above correspond to these scenarios.

3.2.2 Implementation

The implementation of the existing analysis `RemoveUnused` is done with a single pass over the source tree in SSA form that visits all the reachable code and marks it as *used*. As described earlier the implementation does not generate constraints and therefore it can do with just one pass over the entire program.

There are data structures for the different constructs in the program, e.g., variables, functions, type constructors, and labels. These structures model instances of the particular constructs and expose different properties and operations. The main information in each of these structures is the two point lattice V. This lattice can then be lifted for each of the constructs when the analysis has traversed a place where the construct is used. The structures are associated with each construct in the program and this association is expressed by the function lattice. The formal description of the analysis above only models this *used*/*unused* information for variables and not the other constructs in the program. We have chosen to focus on variables and their values in the improvement of the analysis. The rest is removed for clarity.

Constraints are not used as part of the implementation but only used to formally capture the essence of the analysis and to make it easier to describe improvements to the analysis.

As mentioned above the analysis is a single pass over the source tree rather than generating and solving constraints. Since it is a single pass some measures have to be taken to ensure that result is not dependent on the order in which the tree is traversed. It could be that a label's formal parameters are visited before any call to that label is visited and we therefore cannot know the abstract value of the actual parameters that will flow into the formal parameters at this time.

This is handled by introducing listeners or observers as known from the observer programming pattern. These listeners are anonymous functions that are attached to the lattice for the formal parameters. These anonymous functions will be called if the *used*/*unused* lattice is lifted from *unused* to *used* and they will propagate the information for those variables. These variables can themselves have attached functions that will further propagate the information.

An entire function can have all its variables connected with listeners before an actual call to that function is visited. Before the call is visited none of the structures for variables in the function are marked as *used* and when the call is visited the listeners convey this information through all the reachable statements in the function.

The reason the analysis works with a single pass is because of the simplicity of the lattices used. After a variable is marked as used it cannot change again and we have at least calculated a post fix-point for the values. There is no need to iterate the fix-point computation since once the variables are marked as used they cannot change. They can either be set by the analysis visiting them and marking them as used, as described in the last four rules in the previous section, or they can be set by information propagation from the anonymous functions attached as listeners. This propagation will function as the fix-point computation.

3.3 Improvement

We need to keep track of which *exn* constructor each variable can hold and which *exn* constructors each function can possibly raise. With this information we should be able to distinguish each function's potentially raised exceptions and remove impossible ones from the handler.

To keep track of the *exn* constructors we introduce two new lattices:

$$E : F \to \wp(exn)$$
$$E : Var \to \wp(exn)$$

These lattices relate functions and variables to powersets of *exn* constructors. The sets are ordered by inclusion and the functions are ordered point-wise. A variable that is defined in a let-binding with a constructor application has a singleton set related to it. A variable that is a formal parameter or is assigned the value of a formal parameter have several elements in the lattice corresponding to its ϕ node in SSA. Functions also have a powerset of *exn* constructors related to them and these represent the possible exceptions they can raise.

First we need a constraint that can capture the *exn* values as they are constructed. This can happen in assignments of variables so a new constraint rule will look like this:

$$\frac{(X \leftarrow \mathtt{c}(x^*)) \in P \qquad \mathtt{c} : exn}{E[\![X]\!] = \{\mathtt{c}\}}\ \textsc{Exn Constructor Def.}$$

The notation $\texttt{c}:exn$ means that c is of the type *exn*. When a constructor application of the *exn* datatype is present in the program we ensure that the exception set for the defined variable contains the specific constructor. We then extend the definition rule.

$$\frac{(X \leftarrow x) \in P}{\begin{array}{c} V[\![x]\!] = used \\ E[\![X]\!] = E[\![x]\!] \end{array}} \textsc{Definition}$$

The above rule now ensures that the newly defined variable contains the same information as its right-hand side. With the properties of SSA and the optimizations performed, this rule will happen rarely if not at all. Since variables are defined once and not altered a simple assignment of one variable to another will not contribute any expressiveness to the program since instead of X one could simply use x. It is stated here so that our analysis is not dependent on the SSA code being optimized already. Next we need to update the raise rule.

$$\frac{(\texttt{raise}\ x) \in P \qquad func(\texttt{raise}\ x) = \texttt{f}}{\begin{array}{c} F_{rai}[\![\texttt{f}]\!] \supseteq \{x\} \\ E[\![\texttt{f}]\!] \supseteq E[\![x]\!] \end{array}} \textsc{Raise}$$

Now this rule ensures that the exception set for each function will contain all the *exn* constructors that the variables in the raise transfers can hold.

The non-tail call rule is extended with another constraint that flows information about all the *exn* constructors the called function can raise into the parameter of the handler.

$$\frac{\begin{array}{c} \texttt{l}(X_1, ..., X_n) \in P \qquad \texttt{l}'(X_r) \in P \qquad \texttt{l}''(X_h) \in P \\ (\texttt{f}(x_1, ..., x_n)\ \texttt{NonTail}\ \{\texttt{cont=l}',\texttt{handler=l}''\}) \in P \\ F_{ret}[\![\texttt{f}]\!] = \{r_1, ..., r_m\} \qquad F_{rai}[\![\texttt{f}]\!] = \{e_1, ..., e_k\} \end{array}}{\begin{array}{c} V[\![x_1, ..., x_n]\!] \sqsubseteq V[\![X_1, ..., X_n]\!] \\ V[\![r_1]\!] \sqcup ... \sqcup V[\![r_m]\!] \sqsubseteq V[\![X_r]\!] \\ V[\![e_1]\!] \sqcup ... \sqcup V[\![e_k]\!] \sqsubseteq V[\![X_h]\!] \\ E[\![\texttt{f}]\!] \subseteq E[\![X_h]\!] \end{array}} \textsc{NonTail call - label}$$

The single variable that is the parameter to the handler now contain a number of possible values. The other non-tail call rule with `Caller` as the handler is extended by the same principle but here the exception constructors do not flow to a variable but is propagated back to the calling function.

$$\frac{\begin{array}{c}\texttt{l}(X_1, ..., X_n) \in P \qquad \texttt{l}'(X_r) \in P \\ (\texttt{f}(x_1, ..., x_n)\ \texttt{NonTail {cont=l}'\texttt{,handler=Caller\}}) = S \\ S \in P \qquad \mathit{func}(S) = \texttt{f}' \qquad F_{ret}[\![\texttt{f}]\!] = \{r_1, ..., r_m\}\end{array}}{\begin{array}{c}V[\![x_1, ..., x_n]\!] \sqsubseteq V[\![X_1, ..., X_n]\!] \\ V[\![r_1]\!] \sqcup ... \sqcup V[\![r_m]\!] \sqsubseteq V[\![X_r]\!] \\ F_{rai}[\![\texttt{f}']\!] \supseteq F_{rai}[\![\texttt{f}]\!] \\ E[\![\texttt{f}']\!] \supseteq E[\![\texttt{f}]\!]\end{array}}\ \textsc{NonTail call - caller}$$

Lastly the tail-call rule is extended with both the exception and return information propagated to the calling function.

$$\frac{\begin{array}{c}\texttt{f}(X_1, ..., X_n) \in P \qquad S \in P \\ \mathit{func}(S) = \texttt{f}' \qquad (\texttt{f}(x_1, ..., x_n)\ \texttt{Tail}) = S\end{array}}{\begin{array}{c}V[\![x_1, ..., x_n]\!] \sqsubseteq V[\![X_1, ..., X_n]\!] \\ F_{ret}[\![\texttt{f}']\!] \supseteq F_{ret}[\![\texttt{f}]\!] \\ F_{rai}[\![\texttt{f}']\!] \supseteq F_{rai}[\![\texttt{f}]\!] \\ E[\![\texttt{f}']\!] \supseteq E[\![\texttt{f}]\!]\end{array}}\ \textsc{Tail call}$$

Here all the information is simply propagated from the called to the calling function. With this information the analysis should be able to remove the checks for *exn* values that the handler can never receive.

Omitted here are the rules for the simple propagation of *exn* values from actual parameters to formal parameters. These rules are similar to the ones described in the previous section regarding the V lattice, but they flow information in the E lattice instead.

3.4 Program transformations

With the information the improved analysis provides we are able to conservatively know which exception constructors the different handlers can expect to receive. We can then remove cases for non-occurring exceptions and in the end get a smaller compiled binary by eliminating the dead code from it.

After the analysis has completed and we have a set of exception value each variable can contain, the transformation of the program begins. For each case-transfer that deconstructs the variable of the type *exn* we look up the set of value from the analysis and compare them to the list of cases in the transfer. If there exists a case for a *exn* value that is not in the list of possible values we remove this case.

If a handler is found only to handle exceptions that can never arise, as in the example in listing 3.3, all of the cases for these non-occurring constructors will be removed and only the default case will remain. The case-transfer can in this situation be replaced by a label call to the label in the default case.

After this optimization a run of the existing analysis called `RemoveUnused` will remove the blocks for which the cases are removed, if they are not used anywhere else in the program. This will further remove everything those blocks refers to if it is not used anywhere else in the program, etc.

3.5 Implementation

As mentioned earlier the implementation does not explicitly model constraints and solve them. The analysis is originally based on the existing `RemoveUnused` optimization but the implementation is no longer a single pass over the source tree. It is several passes over the source where each pass contributes to the dataflow captured by the constraints.

First the implementation visits all the global definitions in the top of the program where it finds all the definitions of variables of the type *exn*. These definitions are constructor definitions where a constructor of the exception datatype is called. The called constructor is added to the structure, specified as E in the previous section, for the defined variable like the rule *Exn Constructor Def.* specifies.

The global definitions are traversed one time and are followed by several passes over the functions in the program. The first traversal of the functions and their blocks initializes the E structure for all variables used as parameters to either a function or a label. The structures will be initialized with the empty set of values. This initialization pass is performed to ensure we can safely flow information from one variable to another in the following passes without worrying about non-initialized structures. This first iteration also creates an E structure for each function that will hold the information about the potentially raised exceptions for that function.

Since this implementation does not use listeners to propagate the dataflow it uses a fix-point calculation. The fix-point calculation performs the dataflow through the program and reaches a fixed state. Several steps are taken in each fix-point calculation in accordance with the improved rule from the previous section.

- Data in the E structure flows from actual to formal parameters.
- Raise-transfers are visited and the set of exception constructors in E of the raised variables flows into the E structure for the current function.
- Raised exception constructors for functions called as tail-calls or with `handler = Caller` flows into the calling functions E structure.
- Raised exception constructors for functions called as non-tail calls flows into the parameter of the specified handler.

When the analysis reaches a fixed state it stops and the transformation phase begins. The transformation visits all the blocks of all the functions and looks for case-transfers where the variable t being tested is of the *exn* type. When it encounters such a case-transfer it collects the set of possible exceptions constructors for t and iterates through the cases. Each case is compared to the possible values of t; if the tested value is found it is not touched, if the tested value is not found the case is removed from the resulting list of cases.

When all the cases are compared to the values a final check looks at the number of remaining cases. If there are no remaining cases the entire case-transfer is replaced with a goto-transfer (label call) to the label in the default case. This requires that the case-transfer contains a default case and that is not always true. There is an earlier pass of the SSA tree that ensures that a case-transfer either has a default case or is exhaustive. In this implementation non-default but exhaustive case-transfers are not transformed. Such an exhaustive case-transfer is only possible for the compiler to generate since the user does not know which exceptions the program in SSA form will contain when writing the code in SML.

After the transformation is performed another pass of the existing optimization `RemoveUnused` is performed to remove potential dead code generated by the transformation above.

3.6 Benchmark

To see if the improvement has any effect it we compare the results from the example program in listing 3.2 both without the improvement and with the improvement. The example program has a single print call exactly in the handler the analysis will detect as dead and later remove. After the first improvement analysis is complete the existing `RemoveUnused` analysis will remove all the code related to the print statement that are imported from the standard library. This will ensure that this small example will have a reduced binary size compared to the original without the improvement.

If we study the resulting SSA code with the improvement implemented we see that the call to `fib2_0` now has a different handler:

```
L_128 ()
   fib2_0 (global_167) NonTail {cont = L_129, handler =
        Handle L_130}
```

Listing 3.7: Call to `fib2` after improvement

The handler is now label `L_130` that has been transformed compared to listing 3.4. It now has a label call in place of the case transfer in the non-improved version. Recall that the original example had a case-transfer that had a single case for the `E1_0` exception and a default case.

```
 L_130 (x_181: exn)
   L_134 (tuple_0, messagers_0, x_181)
```

Listing 3.8: Handler to `fib2` call after improvement

This label call to label `L_134` is from the default case in the original version. The original default case was a transfer to a label without parameters and this label had the call to label `L_134` as its only content. The replacement of the case-transfer with the label call to the default case has then been replaced with the call to `L_134` directly. The small improvement where a call to a label with no parameters and no statements are replaced with the transfer for the called label is performed by a pass called `Shrink`. This small pass is performed by all optimizations passes on the SSA language after they have performed their function. It is a simple pass that removes instances like the one described above.

As described earlier an extra iteration of `RemoveUnused` is performed after the first improvement analysis. This will potentially remove dead code that has become dead after the first improvement analysis is done transforming the program. In figure 3.1 we compare the sizes of the binary and SSA files for the example above so we can get a measure of the amount of removed code.

Improvement	SSA size	Binary size
Without	85.2 KB	188.0 KB
With	71.1 KB	178.4 KB

Figure 3.1: Binary size comparison

There is a 10 KB reduction in the binary size that comes from the removed print statement in the original SML program in listing 3.2. The difference in compile time is not measurable in this example or for a self compile so it is negligible.

3.6.1 Benchmark suite

The MLton benchmark suite consists of 43 programs of different complexity. The programs range from a implementation of the Fibonacci sequence to the hamlet example that is a single file consisting of almost 23.000 lines of code. The benchmark suite is used to measure three different aspects; compile-time, run-time, and binary size of the benchmark programs. When running the benchmark suite measuring these three factors there are no hits from improvement one. This means that there are no instances in the benchmark suite where a handler is handling an exception that can not occur.

3.7 Assessment

The first improvement of removing unused exception handlers was not founded on SSA code from the benchmark suite. The reason the improvement was implemented came from SML code that was constructed just to exploit a shortcoming in the compiler. This approach was a gamble since we could not predict how many instances that would occur in the benchmark suite. As it turned out there were no instances in the benchmark suite. This shows that the improvement might not be an optimization but more in the direction of a programming error.

We might consider a dead handler form this improvement as a reason to generate a compiler warning instead of optimizing it away. By silently removing the handlers because they can never be used the programmer might think he handles an exception that can actually occur. If we instead issue a compiler warning that a handler does not handle any exceptions the programmer might revisit the code and take action. In the example in listing 3.2 the programmer might actually have wanted to catch the `E2` exception instead of `E1` from the call to `fib2`. A compiler warning of a dead handler would in this case give the programmer a hint of what might be an actual error.

Chapter 4

Second improvement

The best way to improve the optimizations in MLton would be to detect and simplify large amounts of dead code and preferably in something that the compiler generates or is included from the standard library. We will hand-analyze the generated code the compiler outputs in SSA form to detect places where some additional information will make the analysis able to detect dead code that it otherwise would not, similarly to the first improvement. The difference this time is the focus on standard library code or compiler generated code and not so much on artificial examples.

This section will study generated SSA code and find some examples of code from the MLton benchmark suite where some dead code might be detected by static analysis. Different kinds of analyses are discussed in relation to the examples with the goal of finding one that will detect the most dead code without being too complex or too expensive.

One thing that meets the eye when looking at the generated code and the BNF from earlier is that most of the arithmetic is surrounded by generated overflow handlers. In the BNF we recall the arithmetic transfer:

$$S ::= \ldots \mid \texttt{l}(A)\ \texttt{Overflow => l} \mid\ \ldots$$

where A is some arithmetic operation on two variables. This is one of the two ways arithmetic can be done in SSA form. The second is from primitive arithmetic functions such as `Word32_add`, `Word8_mul`, etc. These primitive operations are not governed by overflow handlers and are therefore only used in specific places in the standard library and not used by the compiler for arithmetic in the user programs. They can of course be used by optimizations that detect that arithmetic cannot overflow in a transfer and then transform the program by replacing an arithmetic transfer with a primitive arithmetic definition.

By compiling the entire MLton benchmark suite and keeping the intermediate SSA output for each program we examine the arithmetic transfers and look for patterns or simple cases where information could detect overflow handlers as dead.

We generally look for two scenarios: (1) case by case local arithmetic between a constant and a variable that has been bounded by a comparison to a constant or another bound, or (2) patterns that occur often and where we believe some arithmetic is guaranteed not to overflow. The second scenario might for example occur in places where arrays are traversed and information about the array length will guarantee that an array index will not overflow.

4.1 Examples

Example 1

Some examples are found by searching the SSA code of the benchmark suite programs for places with an arithmetic transfer immediately preceded by a comparison against a constant. In the SSA output for the file `hamlet.sml` the following snippet is found:

```
L_25120 (x_16013: word32)                                    2898
  x_10771: bool = WordU32_lt (x_10769, global_33)            2899
  case x_10771 of                                            2900
    true => L_22171 | false => L_24271                       2901
L_22171 ()                                                   2902
  loop_355 (x_10769 + global_5) Overflow => L_24272 ()       2903
```

Listing 4.1: Snippet from SSA output for `hamlet.sml`

It is worth noticing that there are no other call sites to the label `L_22171` than the one in line 2901, so we are guaranteed to have been through the comparison on line 2899 when arriving at that label. The variables with the `global` prefix are defined in the start of the program as follows:

```
global_5:  word32 = 0x1
global_33: word32 = 0x3B7
```

Listing 4.2: The two relevant global definitions

By definition 1 they are constant. The comparison that must be fulfilled in order for the addition to occur ensures that the variable `x_10769` is less than global_33. This ensures that the addition on 32-bit words cannot overflow since we at most add 0x3B7 and 0x1.

To better visualize the flow in the SSA examples in this chapter we typeset them as flow graphs. Each label is a node and the transfer for this label is its outgoing edges. A goto or call transfer is a simple edge with no information whereas a case transfer has the value of each case on the outgoing edges. Arithmetic transfers has a dashed line for the overflow exception it can potentially raise.

The hamlet example above is visualized in figure 4.1.

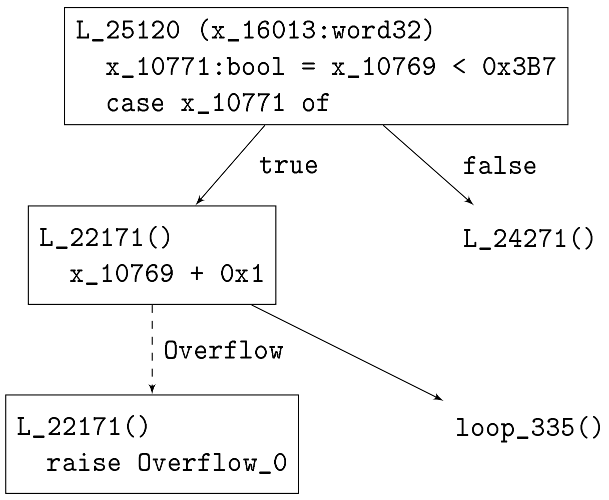

Figure 4.1: Flow graph for the hamlet SSA snippet

It is the dashed overflow line we aim to detect as dead and later remove in a transformation of the program.

Example 2

The next example is more complex and is a not a local optimization as example 1 since it requires knowledge of interprocedural flow of values. It is from the benchmark called DLXSimulator.sml and the SML code in question is:

```
fun exp2 0 = 1
  | exp2 n = 2 * (exp2 (n-1))
```

Listing 4.3: Snippet from DLXSimulator.sml

The intuition of this example is to remove the overflow handler from the `n-1` arithmetic. This subtraction can potentially overflow (technically underflow) but if we can guarantee that the input value to the `exp2` function is always zero or positive the overflow check might be superfluous. So to analyze this example we focus on the calls to `exp2` to see if we can guarantee something about its parameter.

The SSA output for the SML snippet above is:

```
fun exp2_0 (x_1715: word32) = L_1601 ()                       1665
  L_1601 ()                                                   1666
    case x_1715 of                                            1667
      0x0 => L_5493 | _ => L_1602                             1668
  L_5493 ()                                                   1669
    return global_6                                           1670
  L_1602 ()                                                   1671
    L_1604 (x_1715 - global_6) Overflow => L_5492 ()          1672
  L_5492 ()                                                   1673
    raise (global_45)                                         1674
  L_1604 (x_1716: word32)                                     1675
    exp2_0 (x_1716) NonTail {cont = L_1606, handler =         1676
        Caller}
  L_1606 (x_1717: word32)                                     1677
    L_5089 (global_19 * x_1717) Overflow => L_5492 ()         1678
  L_5089 (x_1718: word32)                                     1679
    return x_1718                                             1680
```

Listing 4.4: Snippet of SSA output from `DLXSimulator.sml`

We see that the interesting arithmetic transfer is on line 1672 where the input variable `x_1715` has the variable `global_6` subtracted from it. The globals from this example are as follows:

```
global_6:  word32 = 0x1
global_19: word32 = 0x2
global_45: exn = Overflow_0 ()
```

Listing 4.5: Global definitions from listing 4.4

These are all the globals that occur in the `exp2_0` function. The SSA function in listing 4.4 is visualized as a flow graph in figure 4.2. The only new addition the flow graph notation is the dotted line that is a continuation of a function call.

It is one of the dashed lines we aim to eliminate namely the one from `L_1602` to `L_5492`.

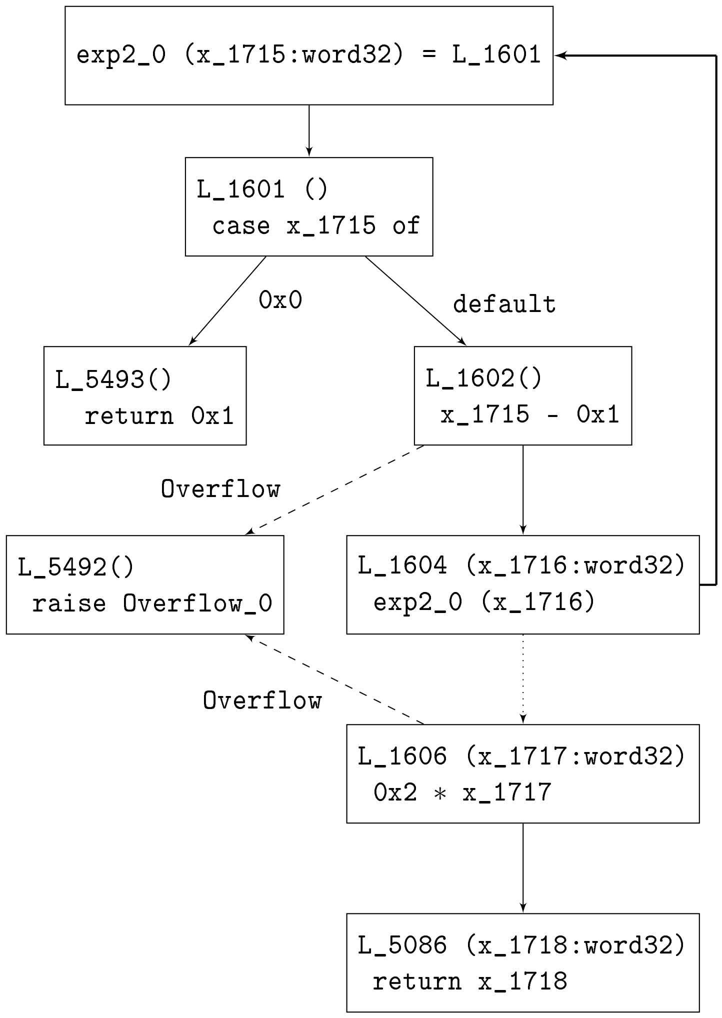

Figure 4.2: Flow graph for the `exp2_0` function

There is a recursive call to `exp2_0` in label `L_1604` (line 1676 in listing 4.4) that corresponds to the call `exp2(n-1)` in listing 4.3. First we look at the recursive call where we assume the parameter to `exp2_0` is zero or positive. Afterwards we try to show that this assumption is valid by looking at the external calls to `exp2_0`.

Now assume that in label `L_1601` we have the following:

$$\texttt{x\_1715} \in [0; 2^{31} - 1] \tag{4.1}$$

Since the width of the words are 32 bit and one bit is used for the sign the maximum value is $2^{31} - 1$. The case transfer in label `L_1601` branches on `0x0` and returns. This means that in the default case in label `L_1602`, together with 4.1, we know that:

$$\texttt{x\_1715} \in [1; 2^{31} - 1] \tag{4.2}$$

By performing the arithmetic transfer in label `L_1602` we flow our knowledge about `x_1715` to the parameter for `L_1604`. So together with 4.2 we know that

$$\texttt{x\_1716} = \texttt{x\_1715} - 1, \text{ therefore}$$
$$\texttt{x\_1716} \in [0; 2^{31} - 2]$$

The variable used as the actual parameter to the recursive call is `x_1716` so the assumed invariant is maintained by this case transfer followed by the arithmetic transfer.

Now we need to look at the external call sites to the `exp2_0` function and see what bounds we have on the parameters from them to ensure that the assumption about the input is valid.

By searching through the SSA output of the entire `DLXSimulator` benchmark the following two call sites outside of the function itself are found.

```
loop_153 (x_2449: word32)
  exp2_0 (x_2449) NonTail {cont = L_5348, handler =
      Handle L_5476}


loop_154 (x_2451: word32)
  exp2_0 (x_2451) NonTail {cont = L_5352, handler =
      Handle L_5476}
```

Listing 4.6: External call sites to `exp2_0`

These call sites look similar in that they both are in a generated label prefixed with `loop`. In both of the call sites the parameter to the label is directly used as a parameter to the function call. Hence we now need to focus on the call sites to both `loop_153` and `loop_154`. First we look at call sites to `loop_153`:

```
L_1493 ()
  [...]
  loop_153 (global_7)

L_5349 ()
  loop_153 (x_2449 + global_6) Overflow => L_5350 ()
```

Listing 4.7: Call sites to `loop_153`

There are two call sites to `loop_153` where one is in label `L_1493` called with a global constant and one in label `L_5349` called as an arithmetic transfer. The snippet in label `L_1493` is there because that code does not influence the call or the parameter to `loop_153`. If we take a look at the situation for `loop_154` we see that it is almost similar:

```
  L_1565 (BlockOffsetBits_0: word32)
    loop_154 (global_7)

  L_5353 ()
    loop_154 (x_2451 + global_6) Overflow => L_5354 ()
```

Listing 4.8: Call sites to `loop_154`

Again two call sites and one with the parameter that is a global constant and one that is the result of an arithmetic transfer. The global constants in the program are `global_6` and `global_7`. `global_6` defined in listing 4.5 and the other is defined here:

```
global_7:  word32 = 0x0
```

Listing 4.9: Global definition in `DLXSimulator.sml`

The two call sites that has `global_7` as the parameter fulfills the invariant proposed that the parameter to `exp2_0` is zero or positive. The last thing we need to consider now is the arithmetic transfers where the result becomes the parameter to `exp2_0`. These two transfers are additions of `global_6` (`0x1`) and the original parameter to the same call. Since that parameter is initially zero and then added to one and passed as a parameter again, it means the parameter is either zero or zero added with one a number of times.

If we look at the call sites to the `exp2` function in the original SML code:

```
fun log2 x =
  let
    fun log2_aux n = if exp2 n > x
                     then (n-1)
                     else log2_aux (n+1)
  in
    log2_aux 0
  end
```

Listing 4.10: Call sites to `exp2` in SML

This single call is translated into two calls in loops in SSA. The parameter `n` to `log2_aux` is originally zero and then incremented by one for the next call. Therefore the invariant is maintained here as well and we have established that the parameter to `exp2_0` is always zero or positive so our transformation of the `n-1` arithmetic transfer is valid.

4.2 Improvement

Overflow or underflow of integers is basically concerned with the number of bits used to express a certain value. If the number of bits required are larger than what the type can hold the overflow happens. We therefore need to model the number of bits used by variables in the program. Previous work has looked at how this information can be expressed and used in an analysis of a program [Stephenson et al., 2000]. Although the application of the results from the analysis differ we look at the different proposals by Stephenson et al. [2000] on how to model this information.

Stephenson et al. [2000] propose three different lattice structures to express the number of bits used for a variable. The first and most straightforward lattice proposed is a lattice that directly model the number of bits used for a variable.

$$\begin{tikzcd}
\top \arrow[d, dash] \\
30 \arrow[d, dotted, dash] \\
1 \arrow[d, dash] \\
0 \arrow[d, dash] \\
\bot
\end{tikzcd}$$

Figure 4.3: Number of bits needed to represent the value of a variable

This structure is easy to implement but not accurate. Every increment of a variable will raise the lattice one bit to soundly handle the worst case, even though the same number of bits might actually be used. The next lattice they propose is not relevant for our purpose since it is a vector of lattices that can be either zero, one, top, or bottom. They include it since they aim to eliminate unused bits potentially in the middle of a bit string. The third structure they propose is a regular interval lattice [Cousot and Cousot, 1976], or range lattice, depending on the literature [Harrison, 1977], that keeps track of a variables lower and upper bound. This lattice is more accurate in their example since it benefits from the exact precision of arithmetic and this is the lattice we will use in our improvement.

For our improvement we will use an interval lattice for each variable in the program. For n bit words let $MAX = 2^{n-1} - 1$ and $MIN = -(2^{n-1})$ then we define the interval lattice IL:

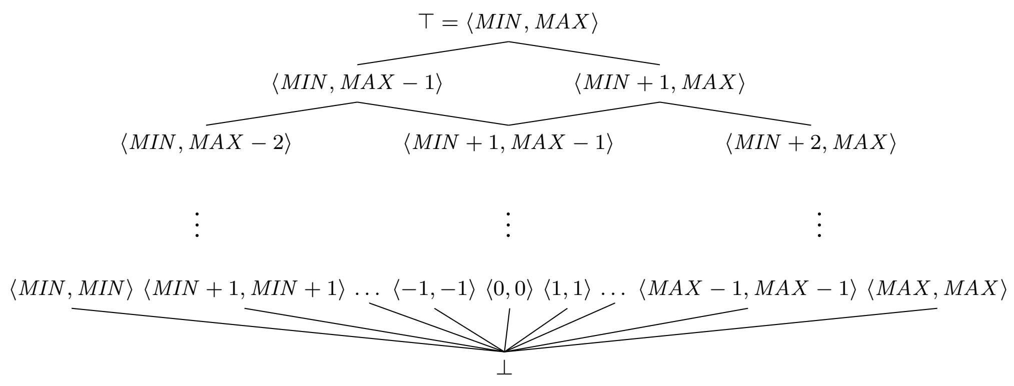
$\top = \langle MIN, MAX \rangle$
$\langle MIN, MAX - 1 \rangle$
$\langle MIN + 1, MAX \rangle$
$\langle MIN, MAX - 2 \rangle$
$\langle MIN + 1, MAX - 1 \rangle$
$\langle MIN + 2, MAX \rangle$
$\langle MIN, MIN \rangle$ $\langle MIN + 1, MIN + 1 \rangle$ … $\langle -1, -1 \rangle$ $\langle 0, 0 \rangle$ $\langle 1, 1 \rangle$ … $\langle MAX - 1, MAX - 1 \rangle$ $\langle MAX, MAX \rangle$
$\perp$

Figure 4.4: Interval lattice *IL*

We define the union of intervals $\sqcup$ as the operator that takes two intervals and returns an interval. It is defined as:

$$\langle a_l, a_h \rangle \sqcup \langle b_l, b_h \rangle = \langle min(a_l, b_l), max(a_h, b_h) \rangle$$

The ordering of the interval lattice $\sqsubseteq$ is using the union operator and is defined as:

$$\langle a_l, a_h \rangle \sqsubseteq \langle b_l, b_h \rangle \Leftrightarrow \langle b_l, b_h \rangle = \langle a_l, a_h \rangle \sqcup \langle b_l, b_h \rangle$$

We also define an interval intersection operator $\sqcap$ to be the interval containing all integers in both intervals such that:

$$\langle a_l, a_h \rangle \sqcap \langle b_l, b_h \rangle = \langle max(a_l, b_l), min(a_h, b_h) \rangle$$

Note that the intersection between two disjoint intervals yields $\perp$. The element $\top$ means that we do not know the value statically or that the values can utilize the entire interval of the type.

With the interval lattice IL we are able to define our lattice that relates variables to intervals:

$$I : Var \rightarrow IL$$

The *IL* lattice is ordered like described above and the function lattice is ordered point-wise. The analysis will first mark all variables with the bottom element denoting that no information is available.

We define a helper function used to look up the abstract value of a variable. The first version of this helper function is simple and might seem superfluous but it is necessary since it will be extended later on. We define a function L with the signature: $\mathit{Var} * \mathit{Label} \to \mathit{IL}$.

$$L(x, l) = I[\![x]\!]$$

This lookup function is used to retrieve the interval for the variable x when it is needed in the label l. Right now it just returns the interval from the I lattice directly and discards the label parameter since it is a global property.

4.2.1 Value propagation

To gain information for each variable we must consider what constructs in the SSA program that gives rise to information and how to update the interval lattice accordingly. The first and most obvious piece of information we can get for a variable is when it is explicitly defined as a literal. We create a rule that will capture the behavior of a constant definition of a variable on the lattice I.

$$\frac{(X \leftarrow v) \in P \qquad v : \mathit{word}}{I[\![X]\!] = \langle v, v \rangle} \text{ CONSTANT DEFINITION}$$

From definition 1 we have that this interval for the variable is constant since the variable is a constant. This information in the interval lattice must be propagated throughout the program. We will use a helper function *label* that returns the label for the block in which a statement exist in the SSA program text, similar to the *func* function from the previous improvement. The propagation rules for the goto transfer is as follows.

$$\frac{\begin{array}{c} \mathtt{l}(X_1, ..., X_n) \in P \qquad \mathtt{l}(x_1, ..., x_n) = S \\ S \in P \qquad i \in \{1, \ldots, n\} \qquad \mathit{label}(S) = \mathtt{l}' \\ x_i : \mathit{word} \end{array}}{L(x_i, \mathtt{l}') \sqsubseteq I[\![X_i]\!]} \text{ GOTO}$$

This is the goto transfer where a simple flow from actuals to formals are performed. We only consider the variables of word type and generate a constraint rule for each of those formals. Function calls are similar in design.

$$\frac{\begin{array}{c}\mathtt{f}(X_1,...,X_n) \in P \qquad \mathtt{l}'(X_r) \in P \\ (\mathtt{f}(x_1,...,x_n)\ \mathtt{NonTail}\ \mathtt{\{cont=l',handler=}H\mathtt{\}}) = S \\ S \in P \qquad i \in \{1,\dots,n\} \qquad \mathit{label}(S) = \mathtt{l}'' \\ x_i : \mathit{word}\end{array}}{L(x_i, \mathtt{l}'') \sqsubseteq I[\![X_i]\!]}\ \textsc{NonTail call}$$

$$\frac{\begin{array}{c}(\mathtt{f}(x_1,...,x_n)\ \mathtt{NonTail}\ \mathtt{\{cont=l',handler=}H\mathtt{\}}) = S \\ \mathtt{l}'(X_r) \in P \qquad S \in P \qquad X_r : \mathit{word}\end{array}}{\top \sqsubseteq I[\![X_r]\!]}\ \textsc{NonTail call - word return}$$

$$\frac{\begin{array}{c}\mathtt{f}(X_1,...,X_n) \in P \qquad (\mathtt{f}(x_1,...,x_n)\ \mathtt{Tail}) = S \\ S \in P \qquad i \in \{1,\dots,n\} \qquad \mathit{label}(S) = \mathtt{l} \\ x_i : \mathit{word}\end{array}}{L(x_i, \mathtt{l}) \sqsubseteq I[\![X_i]\!]}\ \textsc{Tail call}$$

Notice that if the parameter to the continuation label is of type word it is marked as $\top$. These constraint rules are the basic propagation of information throughout the program through joins of abstract values in the formal parameters. We still need to consider the arithmetic transfer that will apply an operation on two word values and pass the result to a label as a parameter. These operations needs to be applied to the abstract values of intervals as well. We therefore define some operations on intervals that correspond to operations on word values.

Let $a, b, c \in Var$, let $\mathtt{l}$ be the current label, and $L(a, \mathtt{l}) = \langle a_l, a_h\rangle$, $L(b, \mathtt{l}) = \langle b_l, b_h\rangle$ then

$$\begin{aligned} c = a + b &\Rightarrow I[\![c]\!] = \langle a_l, a_h\rangle \mathbin{\bar{+}} \langle b_l, b_h\rangle = \langle a_l + b_l, a_h + b_h\rangle \\ c = a \times b &\Rightarrow I[\![c]\!] = \langle a_l, a_h\rangle \mathbin{\bar{\times}} \langle b_l, b_h\rangle = \langle a_l \times b_l, a_h \times b_h\rangle \\ c = a - b &\Rightarrow I[\![c]\!] = \langle a_l, a_h\rangle \mathbin{\bar{-}} \langle b_l, b_h\rangle = \langle a_l - b_h, a_h - b_l\rangle \end{aligned}$$

With the abstract operations above we are able to define the arithmetic transfer rule.

$$\frac{\begin{array}{cc}\mathtt{l}(X) \in P & \mathtt{l}(x_1 \ \mathtt{op}\ x_2) = S \\ S \in P & \mathit{label}(S) = \mathtt{l}'\end{array}}{(L(x_1, \mathtt{l}') \ \bar{\mathtt{op}}\ L(x_2, \mathtt{l}')) \sqsubseteq I[\![X]\!]} \ \textsc{Arithmetic}$$

Arithmetic can also occur as primitive operations in definitions and these use the same abstract operations. We need two helper functions that will return the resulting interval element from one or two interval arguments, respectively, and a primitive function.

$$\mathit{prim_1}(p, \langle x_l, x_h\rangle) = \begin{cases} \langle 0, 0\rangle \ \bar{-}\ \langle x_l, x_h\rangle & \text{, if } p = \mathtt{Word\_neg} \\ \langle 0, \mathit{MAX}\rangle & \text{, if } p = \mathtt{Vector\_length} \\ \top & \text{, otherwise} \end{cases}$$

$$\mathit{prim_2}(p, \langle x_l^1, x_h^1\rangle, \langle x_l^2, x_h^2\rangle) = \begin{cases} \langle x_l^1, x_h^1\rangle \ \bar{+}\ \langle x_l^2, x_h^2\rangle & \text{, if } p = \mathtt{Word\_add} \\ \langle x_l^1, x_h^1\rangle \ \bar{\times}\ \langle x_l^2, x_h^2\rangle & \text{, if } p = \mathtt{Word\_mul} \\ \langle x_l^1, x_h^1\rangle \ \bar{-}\ \langle x_l^2, x_h^2\rangle & \text{, if } p = \mathtt{Word\_sub} \\ \top & \text{, otherwise} \end{cases}$$

The binary functions that are considered here are the operators for addition, subtraction, and multiplication. If a word value is returned from any other primitive function with two arguments it is assigned the abstract value $\top$. Similarly the unary functions considered are the negation function, the length of a vector, and every other primitive function with one parameter that is assigned to a word type are assigned the abstract value $\top$. The `Vector_length` function returns the abstract value of all positive values including zero since a vector with negative length is not possible. The parameter to `Vector_length` is ignored since we do not model the length of vector types so we return the abstract value for the worst case. The following functions are used in the primitive definition rules for unary and binary primitive functions.

$$\frac{\begin{array}{cc}(X \leftarrow p(x)) = S & S \in P \\ X : \mathit{word} & \mathit{label}(S) = \mathtt{l}\end{array}}{I[\![X]\!] = \mathit{prim_1}(p, L(x, \mathtt{l}))} \ \textsc{Primitive Definition - unary}$$

$$\frac{(X \leftarrow p(x_1, x_2)) = S \qquad S \in P \qquad X : word \qquad label(S) = \mathtt{1}}{I[\![X]\!] = prim_2(p, L(x_1, \mathtt{1}), L(x_2, \mathtt{1}))} \text{ Primitive Definition - binary}$$

The rules above are all used to propagate the abstract values to formal parameters by joining them. Since our lattice is of finite height and the ordering operator on our lattices is monotone the propagation will reach a fix point if iterated. Without any widening heuristic this could in the worst case result in $2^{32} - 1$ fix point iterations which is unfeasible.

We need to apply a widening heuristic since the arithmetic in the program could be placed in a loop or in a recursive function call and we have no information about the number times the arithmetic is applied. The arithmetic could potentially be applied indefinitely and the resulting interval would reach its fix point with one or both boundaries at *MIN* or *MAX*.

Widening

The first approach to widening could be to let an interval jump to *MIN* or *MAX* after k number of down or up judgments. The case for $k = 1$ is explained by Cousot and Cousot [1977] but in their example "*[...] the widening is a rough operation which introduces a great loss of information*". This will not be very accurate and we could be in a case where the large jumps reaches a fix point that is far from the optimum. It will be fast to reach a fix point since we are guaranteed to only change each variables abstract value k times. A narrowing approach will be needed to refine the intervals after a jump to the *MIN* or *MAX*.

To make the analysis more accurate, albeit potentially slower, we will make more and smaller jumps to reach a convergence in the fix point calculation. Also shortly proposed by Cousot and Cousot [1977] is the notion of using several steps of widening before widening to the limits. This approach is what we will continue with and implement in our analysis and it will be described in detail here.

Let B be a set of integer values. The elements of B will be used as steps in the widening function. Several heuristics to chose the elements of B can be employed and in our example we will chose B to be the set of all word constants in the program including *MIN* and *MAX*. As mentioned this is a heuristic and one could collect empirical data and compare different approaches but this is not in the scope of this project.

Now we can define a widening function w on an interval that will be called on the resulting interval of a join when the join coarsen the bounds the intervals.

$$w(\langle l, h\rangle) = \langle \max\{x \in B | x \leq l\}, \min\{x \in B | x \geq h\}\rangle$$

The resulting interval from w will have the its lower bound as the largest element in B that is less than or equal to the lower bound of the input. Similarly its upper bound is the smallest element in B that is equal to or greater than the upper bound of the input.

In the rare case where a program contains every integer constant in the range from *MIN* to *MAX* for a particular word size this widening heuristic will be the same as not applying widening at all. Since most programs have a much smaller set of integer constants in the program text it will be faster to make a jump between the constants until the widening function reaches a fix point.

In the implementation of the analysis the widening above is used. Narrowing is not implemented and could be a subject for further study. The narrowing could be done by performing an iteration of the propagation without widening only changing the *MIN* and *MAX* values. This will increase the precision of the abstract values while still being sound.

Refinement of abstract values

With the abstract value propagation and the widening in place the analysis is able to run and produce results. It is possible that these results contain some places where we can perform transformations of the program but we can do much better. As in many interval/range analyses information about boolean comparisons can be used to refine the existing information we have on variables.

When two variables are compared with a less-than operator we can infer information about both variables. In SSA the boolean comparisons are always primitive function calls where the result is bound to a boolean variable. This primitive function call does not give rise to information right away but only when a case transfer is testing on the boolean variable can we use the comparison information. In the case transfer the control can change to the true branch or the false branch of the boolean comparison. We then have to use the information from the boolean comparison down through the control paths.

To aid this propagation of refined information down through the control paths we use what is called a dominator tree as described by Appel [1998, chap. 19]. A dominator tree in this case is a tree of SSA blocks where each blocks children are the blocks that it immediately dominates. The immediate dominator of a block is unique so this structure becomes a tree.

In the MLton compiler the SSA infrastructure has a precomputed dominator tree for each function. This dominator tree can be traversed just like the dominator indifferent approach of just visiting the blocks in the order in which they appear in the list. Traversing the blocks in the dominator tree allows us to visit case transfers before we visit the different control paths it can lead to. This will enable us to propagate refined information about the abstract values we obtain from the comparisons down the corresponding control paths.

We need to formalize three things; (1) how to connect the refined information from a comparison and the resulting boolean variable, (2) how to propagate this information down the control paths when a case transfer tests on a boolean variable with connected information, and (3) how to extract the propagated refined information from the control paths.

The first step is to connect the information gained from a comparison to the resulting boolean variable. We get potentially four pieces of information from a comparison; for both the true and the false branch we have a refinement for both variables. We define A to be the lattice mapping a boolean variable to an interval lattice. This lattice A requires some additional notation to be useful. We write $A_{\mathtt{T},y}[\![X]\!]$ to mean the interval lattice for the variable y if the boolean variable X has evaluated to `true` and $A_{\mathtt{F},y}[\![X]\!]$ to mean the interval lattice for the variable y if the boolean variable X has evaluated to `false`.

The lattice above is only used to temporarily store information about the refinements obtained from comparisons. First when a case transfer dispatches on the boolean variable will the refined lattices be made accessible for the analysis in the control paths. We need a constraint rule for the primitive function definition when the defined variable is of boolean type. We define two operators $\prec$ and $\succeq$ that both take two intervals and return an interval. These operators are used to bound the first interval given as argument by the second argument.

$$\begin{aligned}\langle a_l, a_h\rangle \prec \langle b_l, b_h\rangle &= \langle a_l, a_h\rangle \sqcap \langle a_l, b_h - 1\rangle\\ \langle a_l, a_h\rangle \succeq \langle b_l, b_h\rangle &= \langle a_l, a_h\rangle \sqcap \langle b_l, a_h\rangle\end{aligned}$$

These operators are used in the rule for the primitive comparison less-than. This comparison will assign four intervals to the A lattice for the given boolean variable.

$$\frac{\begin{array}{c} (X \leftarrow \texttt{Word\_lt}(x_1, x_2)) = S \qquad S \in P \\ X : \mathit{bool} \qquad \mathit{label}(S) = \texttt{l} \end{array}}{\begin{array}{c} A_{\texttt{T},x_1}[\![X]\!] = L(x_1, \texttt{l}) \prec L(x_2, \texttt{l}) \\ A_{\texttt{T},x_2}[\![X]\!] = L(x_2, \texttt{l}) \succeq L(x_1, \texttt{l}) \\ A_{\texttt{F},x_1}[\![X]\!] = L(x_1, \texttt{l}) \succeq L(x_2, \texttt{l}) \\ A_{\texttt{F},x_2}[\![X]\!] = L(x_2, \texttt{l}) \prec L(x_1, \texttt{l}) \end{array}} \ \textsc{Primitive Definition - Less-Than}$$

We have connected the boolean variable with the four resulting intervals from the comparison and we now need to propagate this information down the control paths when a case transfer tests on a boolean variable. We need a way to refine the intervals for a variable from its global abstract value in the I lattice. This refinement should be on a control path basis and not a global property.

By taking advantage of the dominator tree we can assign a mapping from blocks in the SSA program to variables to interval lattices. When the analysis visits a case transfer where the test is on a boolean variable that have been assigned the four intervals from the rule above we assign mappings from the blocks of the two cases to both variable each with an interval lattice.

Let a fact F be a mapping from a SSA block to a mapping from a variable to an interval lattice: $F : B \rightarrow \mathit{Var} \rightarrow \mathit{IL}$. Initially the interval lattice IL is $\bot$ for all variables and this is interpreted as non-existing fact, that is: no fact for the specific variable is assigned the block. This lattice F will require the entire subtree of the assigned block to know which blocks have facts assigned which is infeasible. This is solved by recursively looking at a blocks ancestor in the dominator tree whenever we need to know the abstract value for a variable. If we do not find an interval that is different from $\bot$ we interpret this as there is no refined value for the specific variable and we will use the global abstract value from the I lattice.

Facts need to be created for the relevant control paths which the following rule for the case transfer does.

$$\frac{\begin{array}{c} (\texttt{case } x \texttt{ of true=>l | false=>l}') \in P \qquad x : \mathit{bool} \\ A_{\texttt{T},y}[\![x]\!] = i \qquad A_{\texttt{T},z}[\![x]\!] = i' \\ A_{\texttt{F},y}[\![x]\!] = i'' \qquad A_{\texttt{F},z}[\![x]\!] = i''' \end{array}}{\begin{array}{c} i \sqsubseteq F[\![l]\!][\![y]\!] \qquad i' \sqsubseteq F[\![l]\!][\![z]\!] \\ i'' \sqsubseteq F[\![l']\!][\![y]\!] \qquad i''' \sqsubseteq F[\![l']\!][\![z]\!] \end{array}} \ \textsc{Case Transfer - Bool}$$

The case transfer is also able to give information about the intervals if the tested values are words types and are equal to the bounded elements in the interval. For simplicity we only consider case transfers with a single case for a constant word value and a default case.

$$\frac{(\texttt{case}\ x\ \texttt{of}\ v\texttt{=>l}\ \texttt{|}\ \texttt{\_=>l}')\in P \qquad x : \textit{word} \\ I[\![x]\!] = \langle x_l, x_h\rangle \qquad x_l = v}{\langle x_l + 1, x_h\rangle \sqsubseteq F[\![\texttt{l}']\!][\![x]\!] \\ \langle v, v\rangle \sqsubseteq F[\![\texttt{l}]\!][\![x]\!]}\ \textsc{Case Transfer - Lower}$$

$$\frac{(\texttt{case}\ x\ \texttt{of}\ v\texttt{=>l}\ \texttt{|}\ \texttt{\_=>l}')\in P \qquad x : \textit{word} \\ I[\![x]\!] = \langle x_l, x_h\rangle \qquad x_h = v}{\langle x_l, x_h - 1\rangle \sqsubseteq F[\![\texttt{l}']\!][\![x]\!] \\ \langle v, v\rangle \sqsubseteq F[\![\texttt{l}]\!][\![x]\!]}\ \textsc{Case Transfer - Upper}$$

These two rules removes a single value from the edge of an interval and propagate this refined information to the default case. Then they propagate the constant interval to the path from the value case. These rules can be extended to include series of values that each can refine that interval in the default case but this is omitted since it is not clear how much this happens in real-world programs. We create a generic case transfer rule where each value literal is used for a constant interval that is propagated through the respective control path.

$$\frac{(\texttt{case}\ x\ \texttt{of}\ v_1\texttt{=>l}_1\ \texttt{|}\ \dots\ \texttt{|}\ v_n\texttt{=>l}_n\ \texttt{|}\ \texttt{\_=>l})\in P \\ i \in \{1,\dots,n\} \qquad x : \textit{word}}{\langle v_i, v_i\rangle \sqsubseteq F[\![\texttt{l}_i]\!][\![x]\!]}\ \textsc{Case Transfer - Literals}$$

We now have the refined information assigned to the start of the control path and we need to access this information during the traversal of the dominator tree. Every time we need the abstract value of a variable we first need to look at the refined intervals in the ancestors in the dominator tree of the blocks we are currently visiting. If this yields no result other than $\bot$ we use the global value as normal. To enable this search in the ancestor chain we redefine our lookup function L to include the recursive search in the ancestors.

$$L(x,l) = \begin{cases} L_A(x,l) & \text{, if } L_A(x,l) \neq \bot \\ I[\![x]\!] & \text{, otherwise} \end{cases}$$

Now the redefined lookup function first calls a second lookup function L_A. If the second lookup function returns something different than $\perp$ the refined value is returned, if not then the global value from the I lattice is returned like before. The L_A function is defined as a recursive function that returns an element from the *IL* lattice. We define a helper function *anc* that takes a label as input and returns the ancestor in the dominator tree for that label.

$$L_A(x,l) = \begin{cases} F[\![l]\!][\![x]\!] & \text{, if } F[\![l]\!][\![x]\!] \neq \perp \\ \perp & \text{, if } anc(l) = \emptyset \\ L_A(x, anc(l)) & \text{, otherwise} \end{cases}$$

This function searches the dominator ancestor tree for a value different than $\perp$ meaning that a refined value exist for the variable x. If it reaches a label with no ancestor in the dominator tree and no element other than $\perp$ is found for the variable then we return $\perp$ to indicate that L should return the global value for the variable.

The analysis is now complete and the propagation and subsequent refinement of values is iterated until there is no change in the state of the lattices used in the analysis. This state we call a fix point and we are able to perform the detection of dead overflow handlers and perform the transformation of the program.

4.3 Program transformations

After the analysis is done and we have reached a fix point for the abstract values for all the word variables we transform the program. Our goal is to remove checks for overflow where it can never occur. We need to visit all arithmetic transfers and look at the abstract values for the arguments. To detect an arithmetic transfer that cannot possibly overflow we need to perform the operation on the abstract values and test if the resulting interval is within the bounds of the word size.

For each arithmetic transfer let a and b be the arguments to the arithmetic operator `op` and let *MIN* and *MAX* be the smallest and largest number, respectively, that the resulting variable can hold. These two numbers depend on the number of bits in the word and whether or not the value is signed by using one bit for the sign. Let $I[\![a]\!] = \langle a_l, a_h \rangle$ and $I[\![b]\!] = \langle b_l, b_h \rangle$ and a `op` $b = c$ then we have $I[\![c]\!] = I[\![a]\!]\ \bar{\texttt{op}}\ I[\![b]\!] = \langle c_l, c_h \rangle$.

Now to detect if an arithmetic transfer is guaranteed not to overflow we look at the bounds of the resulting interval. If $c_l > MIN$ and $c_h < MAX$

no overflow can happen and we are able to transform this arithmetic transfer by removing the overflow check.

To remove the overflow check we must translate the arithmetic transfer into a primitive function call and a goto transfer. Hence we translate

$$\texttt{l}(x_1\ \texttt{op}\ x_2)\ \texttt{Overflow => l}'$$

into

$$\begin{aligned}&x \leftarrow p_{\texttt{op}}(x_1, x_2)\\&\texttt{l}(x)\end{aligned}$$

where x is a fresh variable and where $p_{\texttt{op}}$ is the primitive function corresponding to the operator in the arithmetic transfer; `Word_add`, `Word_sub`, or `Word_mul`. We add a statement before the transfer in the block and we change the arithmetic transfer to a goto transfer.

The size of the intermediate SSA output is not reduced much from the transformation. The transformation itself will remove only a few instructions in the resulting binary. The most notable, albeit small, decrease in binary size comes from an additional run of the analysis `RemoveUnused` that might remove the block $\texttt{l}'$ from the example above and every block this dominates, if $\texttt{l}'$ is not used anywhere else in the program.

4.4 Implementation

The implementation of the analysis is structured in two parts like the previous improvement; analysis and transformation. The analysis follows the constraint rules from the previous section but does not model or solve these constraints explicitly. Rather the behavior expressed by the constraint rules are captured in a fix-point dataflow analysis.

The analysis and transformation is around 1.000 lines of code and the interval lattice implementation is about 400 lines of codes including blank lines.

The interval lattice contains the relevant operations to intersect, join, arithmetic operations, etc. It also contains the code to restrict an interval lattice to be less than another interval lattice used in the boolean primitives. Structurally an interval lattice contains a value and a size. The size represents the number of bits in the type of the variable modeled by the lattice and the value is either $\top$, $\perp$, or a pair of interval elements. An interval element is either *MIN*, *MAX*, or an integer x so that $\mathit{MIN} < x < \mathit{MAX}$.

The size is used by the lattice to know when an operation might need to go from an integer interval element to either *MIN* or *MAX*. It should be mentioned that the pair of interval elements (*MIN*, *MAX*) is the same as the value $\top$ and that a pair like (*MAX*, *MIN*) is not legal. A normalization function that is called after each operation ensures that no illegal values are produced and that (*MIN*, *MAX*) is replaced by the value $\top$.

The analysis itself starts by assigning an interval lattice to all word-typed variables in the program assigning them the value $\perp$. This includes globals, formal parameters and variables assigned in blocks. After an iteration over the word-typed globals each one will have its interval lattice updated to reflect its identical constant interval bounds.

There are two functions used to perform the fix-point calculation; they are called `propagate` and `visitFunction`. The first function is used to join the values of actual parameters of all calls and gotos in the program to their respective formal parameters. This includes performing the arithmetic abstractly on the lattices in the arithmetic transfers and joining the resulting value on the formal parameters for the called label. The second function is used to traverse each SSA function and to extract information from primitive operations like `Word_add` and `Vector_length` that will be used in the subsequent propagation of values.

The `visitFunction` is called for each function in the program and traverses each function following its dominator tree. The dominator tree is precalculated for each function when the code is translated into SSA form. The dominator traversal will visit all the blocks in the function at some point but the order in which the blocks are traversed is advantageous to the propagation of the bounded intervals produced by the boolean primitive functions. These boolean variables have the four states expressed in the A lattice from the comparison assigned to it. These are the two cases for each variable used in the comparison. One for the true case and one for the false case for each variable. When a case transfer matches on a boolean variable with the four states in the lattice A assigned to it these states will be propagated down in the dominator tree at the respective control paths. This behavior is identical to the one expressed in the constraint rules in section 4.2.1.

This allows the analysis to have different intervals for the same variable for different control paths making the analysis *flow sensitive*. To allow for this extra information to be stored it is not enough to have the variable assigned an interval in the I lattice since this is global property. Therefore each block in the program can have a list of facts (Fs) assigned. A fact is a variable and an interval lattice that represents a refined interval for the variable. To establish a lookup chain for the facts each block is also assigned its nearest ancestor in the dominator tree that contains a fact. If a block is not assigned a fact it is not in the ancestor chain. This allows the compiler to search through the ancestor chain first for an interval for a given variable; if no fact for that specific variable is found in the ancestor chain the global interval lattice is taken from the I lattice.

This use of a dominator tree in this case ensures that a boolean comparison of variables will be visited before any blocks dominated by this comparison. This ensures that the assignment of facts has happened before they are used. If the blocks are traversed in the order in which they appear in the SSA program text this property is not guaranteed.

After the iteration of these functions has reached a fixed state, where all the joins in the iteration no longer changes the state of the interval lattices, the information obtained is used to transform the program. We now have a conservative approximation of the intervals for each word-typed variable and we can perform the abstract arithmetic in the arithmetic transfers and detect potential dead overflow handlers. The transformation is a rewrite of the entire program where almost every instruction is reused as it were except the arithmetic transfers that are rewritten as described in the previous section.

4.5 Benchmark

In this section we look at the effect of the analysis and transformation on the examples found earlier in this chapter. We then extend the search for changes to the entire MLton benchmark suite where different measures will tell how many instances of the transformation has taken place and how that is reflected in the binary sizes of the resulting files. We will look at compile time and run time of the improved programs as well.

First it should be mentioned that the compiler with the improvement of course compiles the benchmark suite with no problems. Besides that it passes the regression suite for the MLton compiler consisting of 271 SML programs that tries to catch the corner cases of the compiler. It also compiles and runs the MLton compiler itself.

Example 1

If we look at the first example in listing 4.1 we have an overflow arithmetic that can't possibly overflow. When compiling the `hamlet.sml` file with the improvement and search for the specific instance from example 1 we find the snippet shown in listing 4.11.

```
loop_355 (x_10769: word32)                                  2790
    x_10772: bool = WordU32_lt (x_10769, global_33)         2791
    case x_10772 of                                         2792
      true => L_22171 | false => L_24271                    2793
  L_22171 ()                                                2794
    x_18279: word32 = Word32_add (x_10769, global_5)        2795
    loop_355 (x_18279)                                      2796
```

Listing 4.11: Snippet from improved SSA output for `hamlet.sml`

We can see that the overflow arithmetic transfer has been translated into a primitive function definition that performs the same operation but no longer actively checks for overflow. This example has been transformed as expected from the initial study of overflow arithmetic. The information required to transform this example comes from an assignment to the global variable `global_33` and from the boolean primitive definition of `WordU33_lt`.

When the definition of the boolean variable `x_10772` is visited in line 2791 in listing 4.11 four new facts are created and assigned to it. Since the abstract value of `x_10769` before the boolean definition has no relevance we will here assign it the value $\top$. So we let $L(\texttt{x\_10769}, \texttt{loop\_355}) = \top$ and of course $L(\texttt{global\_33}, \texttt{loop\_355}) = \langle 951, 951\rangle$. Now the boolean variable `x_10772` have the following four facts assigned to it:

`True`:

$$A_{\texttt{T},\texttt{x\_10769}}[\![\texttt{x\_10772}]\!] = \langle \mathit{MIN}, 950\rangle$$
$$A_{\texttt{T},\texttt{global\_33}}[\![\texttt{x\_10772}]\!] = \langle 951, 951\rangle$$

`False`:

$$A_{\texttt{F},\texttt{x\_10769}}[\![\texttt{x\_10772}]\!] = \langle 951, \mathit{MAX}\rangle$$
$$A_{\texttt{F},\texttt{global\_33}}[\![\texttt{x\_10772}]\!] = \langle 951, 951\rangle$$

The boolean variable now holds potential facts to be propagated down the control paths if it is matched in a case transfer with a `true` and a `false` case. This match happens in line 2792 where the `true` facts are added to the label `L_22171` and the `false` facts are added to the label `L_24271`. As a technicality it should be mentioned that in the implementation, since the abstract value for the variable `global_33` has not changed from its global value, no fact is propagated for this global variable. This minimizes the ancestor chain that contains the facts and that will in turn minimize the time is takes to look up a fact.

The facts are now assigned to the labels in the case transfer and the labels will be visited by the analysis since they are below in the dominator tree. When the analysis visits the original arithmetic transfer on line 2903 in listing 4.1 it needs the abstract value for the two variables used as operands for the arithmetic operator. These values will first be searched for in each entry in the ancestor chain and if not found there looked up in the global environment I. The global variable is not added to the facts for the label `L_22171` since it has not changed from its global value but the abstract value for the variable `x_10769` is found in the list of facts for the label. The analysis then uses this value for the test of the unreachable overflow.

In the example above the abstract operation is performed with the two values found in the list of facts and global environment, respectively: $\langle MIN, 950\rangle \mathbin{\bar{+}} \langle 1, 1\rangle = \langle MIN + 1, 951\rangle$. Neither $MIN + 1$ or 951 are above or below the boundaries for a 32 bit word. The arithmetic transfer meets the requirements for not producing overflow at run-time and is transformed.

Example 2

The second example with its main part shown in listing 4.4 is also transformed after the new analysis. The improved version of the SSA function `exp2_0` is shown in listing 4.12.

```
fun exp2_0 (x_1715: word32): {raises = Some (exn),
    returns = Some (word32)} = L_1601 ()
  L_1601 ()
    case x_1715 of
      0x0 => L_5493 | _ => L_1602
  L_5493 ()
    return global_6
  L_1602 ()
    x_3059: word32 = Word32_sub (x_1715, global_6)
    exp2_0 (x_3059) NonTail {cont = L_1606, handler =
        Caller}
  L_1606 (x_1717: word32)
    L_5089 (global_19 * x_1717) Overflow => L_5492 ()
  L_5492 ()
    raise (global_45)
  L_5089 (x_1718: word32)
```

```
      return x_1718
```

Listing 4.12: Snippet of improved SSA output from `DLXSimulator.sml`

By comparing with the original code before the improvement it can be observed that some transformation has taken place. The arithmetic transfer we identified earlier as one where overflow is known not to be occurring on run-time is translated into a primitive function definition and a goto transfer. The label `L_1602` now directly calls the `exp2_0` function recursively instead of first going through label `L_1604` as in listing 4.4. This transformation is performed by a generic pass called `Shrink` that is performed after every other optimization pass. The `Shrink` pass will transform a goto transfer to a target block that contains no statements directly into the transfer from the target block.

The invariant described earlier that the input to the `exp2_0` function is zero or positive is valid. By looking at the abstract values after the a fix point is reached we get the following value for the parameter:

$$L(\texttt{x\_1715}, \texttt{exp2\_0}) = \langle 0, MAX \rangle$$

If we follow this value to the first block in the dominator tree `L_1601` we see a case transfer on a word variable. The rule called *Case Transfer - Lower* has created two facts for both the branches of the case transfer. There is the simple constant interval that is assigned the label `L_5493` and the refined interval where one is added to the lower bound of the tested value and added to the default case. This means that we have the following:

$$L(\texttt{x\_1715}, \texttt{L\_1602}) = \langle 1, MAX \rangle$$

In the original example in listing 4.4 we have an arithmetic transfer in label `L_1602` with the following operation and operands: `x_1715 - global_6` that in abstract values translate into the following: $L(\texttt{x\_1715}, \texttt{L\_1602})$ $\bar{-}$ $L(\texttt{global\_6}, \texttt{L\_1602})$ that when our fix point is reached is the same as: $\langle 1, MAX \rangle \bar{-} \langle 1, 1 \rangle = \langle 0, MAX - 1 \rangle$. Since neither 0 nor $MAX - 1$ is below or above the possible values no overflow can happen and the arithmetic transfer is up for transformation.

The invariant is upheld by the result from above, $\langle 0, MAX - 1 \rangle$, being the input to the recursive call to `exp2_0`. Without the refined information from the case transfer the subtraction would result in the value for the variable `x_3059` being widened to $\langle MIN, MAX \rangle$.

4.5.1 Benchmark suite

As mentioned in the section for the improvement 1 the MLton benchmark suite consists of 43 programs of different complexity. The benchmark suite is used to measure three different aspects; compile-time, run-time, and binary size of the benchmark programs. To measure the effect of the second improvement we use these benchmarks and compare them with the MLton compiler[1] without our improvement.

When considering the three factors we measure we need to focus on what the improvement we have done achieves. We remove overflow handlers which in itself is a small transformation and the expected binary size reduction of that transformation alone is small. A larger binary reduction must come from subsequent transformation where larger chunks of now detectable dead code is removed.

The run time is expected to be reduced by the transformation but with only a few arithmetic overflow checks removed the arithmetic must be in hot code that is run intensively to make a difference. So the run time reduction depends on the number of times some arithmetic is performed at run-time and the amount of those arithmetic operations that have their overflow check removed by the transformation.

Compile time is a property that depends on the implementation of the improvement pass. We expect the compile time to be increased since our analysis is far from optimized. Our widening is potentially slow since it might require many steps to reach a fix point. We put only a small emphasis on compile time since this can be reduced considerably by optimizing the implementation and the fix point computation.

So the first interesting measure we will look at is the number of overflow handlers that are transformed in the benchmark suite. In figure 4.1 we see the number of transformed instances and the total number of overflow handlers in the entire benchmark suite.

Handlers	Transformed
4240	913

Table 4.1: Number of transformed overflow handlers

The number of overflow handlers that the improvement is able to transform is 22% of the total number of overflow handlers. This is probably the most important measure of the improvement since the improvement can only remove overflow handlers and nothing else. So the percentage of

[1]Latest version from repository: 20110614

removed handlers is the basis for the measured binary size and run time reductions.

In figure 4.2 on page 56 we see the binary sizes (in bytes) of all the programs in the benchmark suite both with and without our improvement and we see how much of a difference there is. Each program in the benchmark also has the number of overflow handlers that are transformed in its column called hits.

We see a reduction in every benchmark program regarding binary size where some reductions are small and a few are relatively large. As mentioned the binary size of the transformation itself is small but after the transformation new code might be detected as dead and removed by the `RemoveUnused` or the `Shrink` pass that is performed after the improvement.

There are two binary size reductions that are considerably larger than the rest. The delta reduction for `flat-array` and `mandelbrot` are almost 12KB. The large reduction in these two programs are partly caused by all overflow handlers being removed in both cases. This leads to the overflow exception itself being removed from the program since it is no longer used. Some more code regarding the top-level exception handling and reporting is then transformed and further reductions take place. All these subsequent reductions come from the `Shrink` and the `RemoveUnused` passes.

Figure 4.3 on page 57 shows the run time of the benchmark programs with and without the improvement and again a delta column shows the difference between the two. As mentioned the run time is depended on both the number of removed overflow handlers and the number of times these transformed arithmetic operations are performed. The run time is an average of several runs to reduce fluctuations in the results.

The delta numbers that are below one tenth of a second should be considered so small that the fluctuations in the measurements are a relevant factor. There are benchmarks with a considerable number of hits but with almost no run time reduction but as mentioned before the number of times the arithmetic is performed at run-time is an important factor.

There are some measurements where a considerable reduction in run time has taken place. Benchmarks such as `matrix-multiply`, `imp-for`, `mandelbrot`, and `tailfib` all have a considerable reduced run time. In figure 4.5 on page 53 we visualize the reduced run time in a few benchmarks. The time reduction relative to the total run time make the optimization noticeable in these benchmarks. The four mentioned benchmarks all have relatively few removed overflow handlers but the removed ones are in critical places. In the Fibonacci benchmark the arithmetic with minus one and minus two in the tail recursive calls are removed. These are performed

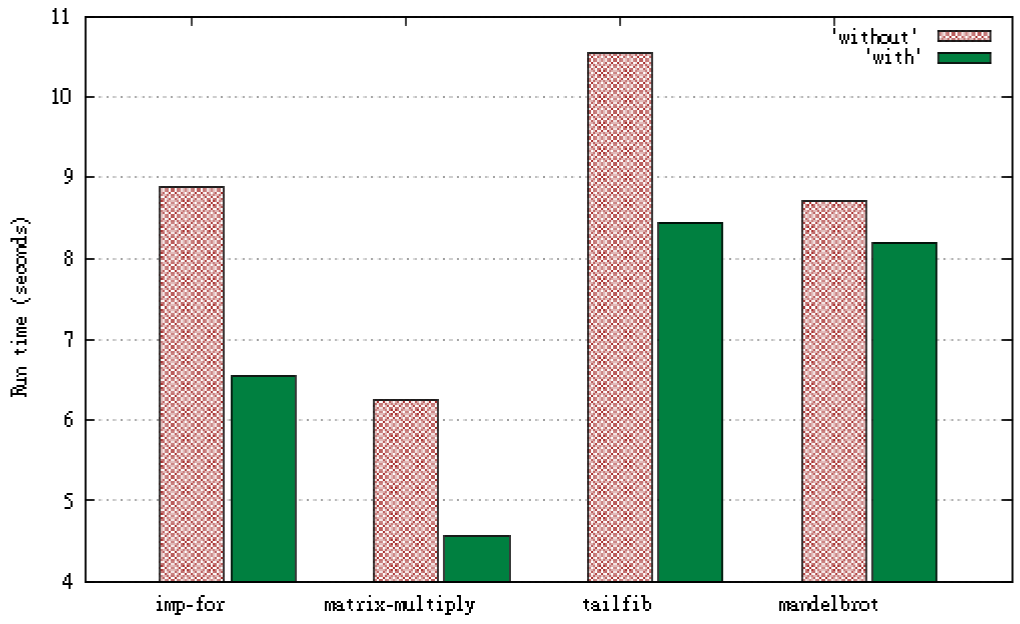

Figure 4.5: Examples of reduced run times

many times when the program is run. The same is the case for the other benchmarks that have hot arithmetic transformed so it does no longer check for overflow.

An interesting observation is the reduction in `tailfib` contra the missing reduction in the non-tail call version of Fibonacci numbers called `fib`. The `fib` benchmark does not have the arithmetic for minus one and minus two transformed. The reason for this is that there are no boolean comparisons in the program but rather a case transfer that matches on zero and one. The SSA for the case transfer can be seen in listing 4.13.

```
L_1422 ()
    case x_1226 of
      0x0 => L_4235 | 0x1 => L_4234 | _ => L_1423
```

Listing 4.13: Case matching in `fib`

This case is not handled by our constraints since there are more than one constant literal being tested for. An extension to the implementation could be to handle the case with an arbitrary number of constant literals refining an interval. When the abstract value for the variable `x_1226` is widened to have a lower bound of zero a refined interval would flow to the default case with a lower bound of two, since the case for zero and one are part of the case transfer.

The compile time for the new improvement is bounded by the fix point calculation that takes up most of the compile time. Figure 4.4 on page 58 shows the compile time for each program in the benchmark suite.

Again the delta figures that are below one tenth of a second are within the deviation of the measurements and are not necessarily a reflection of the change from the new improvement.

The widening function depends on the number of integer constants in the program. This means that large programs with many different integer constants take longer to reach a fix point. There are a few benchmarks with a notably increase in compile size; `hamlet`, `mlyacc`, `model-elimination`, and `vliw` all have an increase above one second and some have an increase above 50%. The compile time for the MLton compiler itself is also increased significantly.

Further work on the implementation will be to optimize the code and work on the fix point calculation. The code is not production ready but it will self-compile, pass the regression tests and the benchmark suite.

4.6 Assessment

Our second improvement was directly based on SSA generated code from the programs in the benchmark suite. This is in contrast to the first improvement that was based on an artificial example. This approach ensured that we at least had the few examples we found where we could perform some optimization. As it turned out the analysis can find much more. Not only did we detect and transform the places we identified by studying the SSA code but the analysis was able to gather information that would be difficult to do by hand. The whole program propagation of values and several layers of boolean comparison lead to 913 transformations in the benchmark suite.

The examples we identified by hand has been presented after the analysis has transformed them and the changes in the code has been discussed. These few examples are only a small fraction of the total number of transformations. There are places where a program performs arithmetic on the result from a primitive call to `Vector_length`. Some of these places comes from the translated user programs and some from the included standard library. Most of the transformations come from the boolean comparisons that bound integer intervals from above or below. These bounds refine the intervals and give the much needed information used to make the actual transformations.

Benchmark	# hits	Size without	Size with	Δ
barnes-hut	25	158,891	157,407	-1,484
boyer	4	224,771	224,419	-352
checksum	6	117,667	117,267	-400
count-graphs	19	138,959	138,415	-544
DLXSimulator	22	188,590	188,046	-544
fft	12	136,160	135,264	-896
fib	4	117,551	117,423	-128
flat-array	4	117,119	104,631	-12,488
hamlet	41	1,219,895	1,219,175	-720
imp-for	4	117,367	117,063	-304
knuth-bendix	25	170,930	170,278	-652
lexgen	29	261,261	260,637	-624
life	7	137,239	136,647	-592
logic	4	181,663	181,583	-80
mandelbrot	5	117,383	104,643	-12,740
matrix-multiply	8	119,063	118,359	-704
md5	20	137,998	137,502	-496
merge	7	118,831	118,463	-368
mlyacc	79	574,885	572,357	-2,528
model-elimination	67	680,992	679,920	-1,072
mpuz	12	122,479	122,095	-384
nucleic	4	268,215	268,023	-192
output1	14	141,718	140,902	-816
peek	19	141,830	141,398	-432
psdes-random	9	120,959	120,239	-720
ratio-regions	18	139,747	139,427	-320
ray	34	223,099	222,491	-608
raytrace	69	308,684	307,084	-1,600
simple	36	290,826	290,218	-608
smith-normal-form	53	249,054	247,326	-1,728
tailfib	5	117,399	117,079	-320
tak	4	117,567	117,383	-184
tensor	34	163,437	162,653	-784
tsp	19	143,086	142,806	-280
tyan	45	201,270	200,726	-544
vector-concat	3	118,683	118,587	-96
vector-rev	5	118,391	118,007	-384
vliw	44	435,832	434,920	-912
wc-input1	24	161,532	160,628	-904
wc-scanStream	24	169,436	168,532	-904
zebra	14	202,158	201,934	-224
zern	32	141,925	140,501	-1,424

Table 4.2: Binary sizes of the benchmark suite (sizes are in bytes)

Benchmark	# hits	RT without	RT with	Δ
barnes-hut	25	3.73	3.72	-0.01
boyer	4	14.30	14.34	0.04
checksum	6	7.44	7.44	0.00
count-graphs	19	6.17	6.15	-0.02
DLXSimulator	22	5.03	5.01	-0.02
fft	12	3.79	3.78	-0.01
fib	4	13.62	13.87	0.15
flat-array	4	6.94	6.92	-0.02
hamlet	41	9.60	9.59	-0.01
imp-for	4	8.88	6.55	-2.33
knuth-bendix	25	5.88	5.83	-0.05
lexgen	29	5.46	5.45	-0.01
life	7	6.28	6.29	0.01
logic	4	5.48	5.48	0.00
mandelbrot	5	8.70	8.20	-0.50
matrix-multiply	8	6.25	4.56	-1.69
md5	20	11.65	11.38	-0.27
merge	7	6.09	6.09	0.00
mlyacc	79	5.61	5.61	0.00
model-elimination	67	9.59	9.56	-0.03
mpuz	12	6.02	6.00	-0.02
nucleic	4	4.23	4.13	-0.10
output1	14	7.42	7.41	-0.01
peek	19	17.10	17.26	0.16
psdes-random	9	7.19	7.31	0.12
ratio-regions	18	21.69	21.68	-0.01
ray	34	5.02	5.00	-0.02
raytrace	69	3.71	3.70	-0.01
simple	36	4.58	4.57	-0.01
smith-normal-form	53	2.64	2.63	-0.01
tailfib	5	10.54	8.45	-2.09
tak	4	6.16	6.69	0.53
tensor	34	12.94	12.94	0.00
tsp	19	6.31	6.29	-0.02
tyan	45	6.16	6.14	-0.02
vector-concat	3	12.43	12.42	-0.01
vector-rev	5	9.08	9.42	0.34
vliw	44	5.68	5.69	0.01
wc-input1	24	10.99	10.90	-0.09
wc-scanStream	24	7.85	7.82	-0.03
zebra	14	7.44	7.46	0.02
zern	32	6.03	5.89	-0.14

Table 4.3: Run times (RT) of the benchmark suite (times are in seconds)

Benchmark	# hits	CT without	CT with	Δ
barnes-hut	25	2.39	2.38	-0.01
boyer	4	2.50	2.67	0.17
checksum	6	1.94	1.88	-0.06
count-graphs	19	2.06	2.17	0.11
DLXSimulator	22	2.48	2.80	0.32
fft	12	2.04	2.04	0.00
fib	4	1.89	1.89	0.00
flat-array	4	1.90	1.91	0.01
hamlet	41	10.42	29.57	19.15
imp-for	4	1.94	1.89	-0.05
knuth-bendix	25	2.28	2.33	0.05
lexgen	29	2.83	3.34	0.51
life	7	2.00	2.02	0.02
logic	4	2.32	2.32	0.00
mandelbrot	5	1.90	1.87	-0.03
matrix-multiply	8	1.92	1.89	-0.03
md5	20	2.04	2.06	0.02
merge	7	1.90	1.86	-0.04
mlyacc	79	5.92	14.29	8.37
model-elimination	67	5.44	9.21	3.77
mpuz	12	1.91	1.91	0.00
nucleic	4	3.38	3.49	0.11
output1	14	2.03	2.04	0.01
peek	19	2.02	2.09	0.07
psdes-random	9	1.98	1.94	-0.04
ratio-regions	18	2.16	2.34	0.15
ray	34	2.56	2.98	0.42
raytrace	69	3.31	4.20	0.89
simple	36	2.98	3.55	0.57
smith-normal-form	53	2.70	2.98	0.28
tailfib	5	1.88	1.84	-0.04
tak	4	1.90	1.86	-0.04
tensor	34	2.36	2.62	0.26
tsp	19	2.06	2.12	0.06
tyan	45	2.48	3.06	0.58
vector-concat	3	1.90	1.87	-0.03
vector-rev	5	1.91	1.88	-0.03
vliw	44	4.16	6.75	2.59
wc-input1	24	2.18	2.38	0.20
wc-scanStream	24	2.22	2.47	0.25
zebra	14	2.54	2.74	0.20
zern	32	2.08	2.16	0.08

Table 4.4: Compile times (CT) of the benchmark suite (times are in seconds)

Chapter 5

Related work

In this section we present related research in the topics of this thesis. We cover exception analysis in ML and interval analysis in general. We will also mention different approaches to handling overflow of integer arithmetic in other languages where minimizing the overhead is also an issue.

An analysis over SSA form in general is presented by Ziarek et al. [2008] where they introduce functional SSA, a variant of classical SSA. Ziarek et al. [2008] presents a tuple flattening transformation and incorporate this into the MLton compiler. The analysis is based on a grammar of functional SSA, a formal semantic for the tuple flattening program transformation, and a proof of type safety and correctness of the transformation. [Fluet and Weeks, 2001] also present an analysis over SSA and use this in the MLton compiler. They introduce a small subset of the SSA form in MLton called FOL and present a grammar for this language. The analysis is used to transform functions that always returns to the same place into continuations. Their approach is to use the dominator tree for the analysis much like we have done in our interval analysis. A comparison between SSA and continuation-passing style is presented by Kelsey [1995] that also presents a small grammar for a SSA procedure.

5.1 Exception analysis of ML

Several approaches to exception analysis in ML has been proposed in the literature. One of the main focus areas has been on the distinction between precision and efficiency of the analyses. Type and Effect systems extend the type system to have an effect component that states the side-effects that can arise from applying a function of a particular function type. Guzmán and Suárez [1994] extend the type system for a subset of ML with the notion of escaping exceptions, though they do not model exceptions as constructors with arguments. With this extension of the type system they are able to infer not only the types of the program but the exceptions as well using an extended type inference algorithm.

Unification of equality constraint approaches generally have an efficient run time but are often imprecise since they cannot model the direction of the value flow within the program. Pessaux and Leroy [1999] use an extended type system that uses unification to keep track of escaping exceptions on a subset of ML. In addition to adding effects on function types they extend the types for integers and exceptions to restrict the values those types can have. To represent these possible values they use a notion of rows representing sets and the unification is done on these.

A different approach uses inclusion constraints over set-expressions [Yi and Ryu, 2002], also known as control-flow analysis. These are theoretically more precise but have a worse run-time complexity since they are able to model the direction of the value flow within the program quite accurately.

An attempt at unifying the constraint-based approaches has been made [Fähndrich and Aiken, 1997, Fähndrich et al., 1998]. Fähndrich et al. [1998] instantiate their general constraint framework BANE with an analysis for uncaught exceptions in ML. Their resulting uncaught exceptions reported by the implementation for `ml-lex` are the same as the ones reported by Yi [1998].

Yi [1998] take a different approach using a collection analysis based on abstract interpretation techniques. This approach is usually regarded as impractical on real-world programs since the run time scales horribly on the input size. The approach is, however, good in the sense that the design of the analysis can be derived once you have the abstraction of the concrete semantics of the language. The language used by Yi [1998] is an intermediate language that the SML programs are translated into.

What the above analyses have in common is the end goal of the result; they all want to report the uncaught exceptions of the program. While this is a very different problem than the analysis itself, some choose to skip this detail altogether [Pessaux and Leroy, 1999] while others are more focused on the aspect of visualization [Fähndrich et al., 1998, Guzmán and Suárez, 1994], but the motivation for the analysis is based on the notion of reporting of uncaught exceptions back to the user.

Fähndrich et al. [1998] divide exceptions in ML into four categories where one is called *pervasive exceptions*. These are control-flow exceptions raised by primitive operations by the built-in system, e.g., `Overflow` and `Subscript`. They choose to ignore this category of exceptions in their analysis since they do not model integer constants or integer arithmetic. The result of this is that their analysis assumes these exceptions can arise from any of the primitive operations.

5.2 Interval analysis and overflow

Interval analysis was first proposed by Cousot and Cousot [1976] but was also concurrently being developed under the name of range analysis by Harrison [1977]. The approach by Cousot and Cousot [1976] has been to work with solid foundations in language semantics and thereby extracting α and γ functions used to translate between values and their abstract counterparts. The other branch called range analysis is developed with a more practical sense in mind. Harrison [1977] develops his range analysis on the specific requirements he faces when trying to implement the analysis in working compilers. The approach is more ad-hoc and the reasoning subject as widening, narrowing and termination is more loose than in the case of Cousot.

Recent work on interval/range analysis comes from the embedded world [Stephenson et al., 2000] where minimizing the number of bits needed to represent the values in the program have a physical effect of the manufacturing of silicon chips. Stephenson et al. [2000] use bi-directional data-range propagation to both propagate data-range forwards and backwards over a program's control paths. They present their *Bitwise* compiler that uses an extended SSA form as an intermediate language. The extension of the SSA form is the notion of range-refinement functions. These functions serve the purpose of refining the ranges much like our approach with the four intervals connected to boolean variables and the creation of facts assigned to blocks.

In a language like Scheme we do not have an overflow error on arithmetic but instead operate with arbitrary precision integers. Fixnums are integers that fit into a machine word and these are not closed under the normal integer operations. When a fixnum no longer fits in a machine word it gets widened to a bignum where this widening can be considered equivalent to our notion of raising an overflow exception. Shivers [1990] proposes the use of range analysis on integers in Scheme to implement efficient integer arithmetic on fixnums. By knowing that an operation on fixnums cannot be widened to a bignum there is no reason to check for it and a more efficient operation can be achieved. This is the same approach we use to eliminate overflow checks.

A dynamic language like LISP does not require the specification of types at compile time so run-time type checking is required. To know information about types at run-time tags are added on data items. Steenkiste [1991] mentions that these tags can constructed in such a way that a single type check on the result of an arithmetic operation can determine if an overflow has happened and if both operands are short integers. By setting the bits in the tags in this way eliminates the need to check independently for short integer operands and short integer result. If the result is not of the short integer type either the operands were not short integers or an overflow has happened. Steenkiste [1991] improves the run time of programs with heavy short integer arithmetic much like we have improved the run time of programs by eliminating overflow checks.

Chapter 6

Conclusion

In this thesis we have explored the intermediate output from the industrial strength MLton compiler in a search for places where static analysis could improve the resulting binaries regarding size and run time. We have formulated, implemented and measured two different improvements regarding exception handlers. The first improvement was based on an artificial example program and we were unsure about the effectiveness of the analysis on real-world programs. As it turned out the MLton benchmark suite contained no places where the analysis could improve the programs.

The next improvement was based on real programs from the benchmark suite of MLton and several examples were found which guaranteed at least those hits when running and measuring with the benchmark suite. After the implementation it turned out that there were many places besides the few examples that could be caught by the analysis. We were able to transform 22% of all the overflow handlers on arithmetic in the benchmark suite resulting in small binaries and reduced run time on a few benchmark that were arithmetic heavy.

Our initial hypothesis was that the binaries generated by the MLton compiler contained dead exception handlers that could be detected and removed with traditional static analysis. We studied and found dead code and formulated analyses that can detect that information. After experiments we showed that the amount of dead code the analysis could find exceeded our expectations and improved the binaries from MLton overall.

6.1 Further work

The second analysis with the overflow handlers was the most successful and will be the basis for this further work section. As we discussed in the benchmark section the compile time is increased and this might be a problem while compiling larger programs.

Compile time

The compile time is bounded by the general implementation and the widening heuristic used. There have been no attempts to speed up the general implementation by changing data structures or other techniques. This will be an obvious first step. Experiments with different widening techniques that would measure the number of hits in the benchmark suite and the compile times can help the search for a faster and maybe even better widening heuristic.

Intraprocedural return values

In our constraints we defined a word type variable returned from a function to be set to the abstract value $\top$. This might be unnecessary imprecise since we might know some information about the return value. The returned value from a function is always a variable and since we have an abstract value for all word variables we could join all the possible abstract values returned from a function into a single abstract value. This returned abstract value will then be joined on the parameter to the continuation to the function. Whether this improvement will enable the compiler to eliminate even more overflow handlers is unknown.

Vector lengths

Our treatment of vector lengths assumes lengths are in the interval $\langle 0, MAX \rangle$. To improve on this we might track vector lengths as part of the analysis. Vectors of word values are also used to represent strings in the SSA form in MLton so by incorporating a sound approximation of string lengths into the analysis might give an upper bound on the primitive call to `Vector_length`. This will further increase the precision of the information in the interval analysis.

Primitive functions

Another aspect of the analysis is the information obtained from the calls to primitive functions. We get information from functions such as `Vector_length` and `Word_neg` but there are several primitive functions we have not considered. A thorough assessment of each primitive function might show that information can be extracted from more of them.

Bibliography

A. Appel. *Modern compiler implementation in ML.* Cambridge University Press, 1998. ISBN 0521582741. 5, 40

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976. 34, 61

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: http://doi.acm.org/10.1145/512950.512973. URL `http://doi.acm.org/10.1145/512950.512973`. 39

M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. *Static Analysis*, pages 114–126, 1997. 60

M. Fähndrich, J. Foster, A. Aiken, and J. Cu. Tracking down exceptions in standard ML programs. *EECS Department, UC Berkeley*, 1998. 60

M. Fluet and S. Weeks. Contification using dominators. *ACM SIGPLAN Notices*, 36(10):2–13, 2001. 59

J. Guzmán and A. Suárez. An extended type system for exceptions. In *Record of the 5th ACM SIGPLAN workshop on ML and its Applications*, pages 105–78153, 1994. 59, 60

W. Harrison. Compiler analysis of the value ranges for variables. *Software Engineering, IEEE Transactions on*, SE-3(3):243 – 250, may 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.231133. 34, 61

R. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995. 59

R. Milner, M. Tofte, R. Harper, and D. Macqueen. *The Definition of Standard ML - Revised.* The MIT Press, rev sub edition, May 1997. ISBN 0262631814. URL `http://www.worldcat.org/isbn/0262631814`. 2

F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 276–290. ACM, 1999. ISBN 1581130953. 60

O. Shivers. Data-flow analysis and type recovery in scheme. *Topics in Advanced Language Implementation*, 1990. 61

P. Steenkiste. The implementation of tags and run-time type checking. *Topics in Advanced Language Implementation, ed. P. Lee*, 1991. 62

M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. In *ACM SIGPLAN Notices*, volume 35, pages 108–120. ACM, 2000. 34, 61

K. Yi. An abstract interpretation for estimating uncaught exceptions in Standard ML programs* 1. *Science of Computer Programming*, 31(1): 147–173, 1998. ISSN 0167-6423. 60

K. Yi and S. Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs* 1. *Theoretical Computer Science*, 277(1-2):185–217, 2002. ISSN 0304-3975. 60

L. Ziarek, S. Weeks, and S. Jagannathan. Flattening tuples in an ssa intermediate representation. *Higher-Order and Symbolic Computation*, 21(3): 333–358, 2008. 59