Institut für Programmstrukturen
und Datenorganisation (IPD)

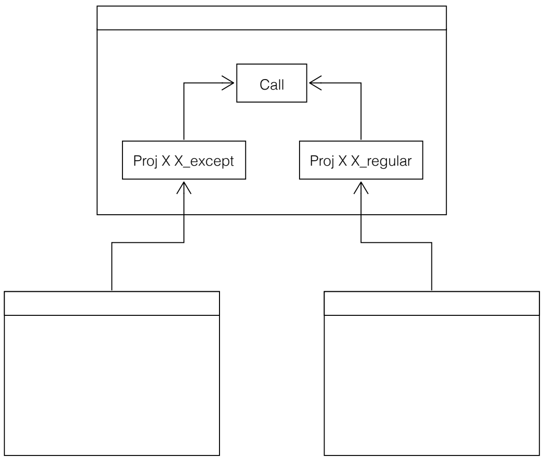
Lehrstuhl Prof. Dr.-Ing. Snelting

Exception Support in a Graph-Based Intermediate Representation

Bachelorarbeit von

Jonas Haag

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuende Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald Dipl.-Inform. Manuel Mohr

Bearbeitungszeit: 11. Januar 2016 – 31. April 2016

Abstract

Exceptions are a well-established technique for handling errors in computer programs. Compilers and intermediate representations must provide special support for exception handling. In this work we present a model for representation of exceptional control flow in the graph-based intermediate representation FIRM and implement it in the LIBFIRM compiler. To verify our implementation, we integrate it into the BYTECODE2FIRM Java compiler.

Ausnahmen sind eine verbreitete Technik zur Fehlerbehandlung in Computerprogrammen. Sie bedarf in Compilern und Zwischenrepräsentationen dedizierter Unterstützung. In dieser Arbeit präsentieren wir ein Modell zur Repräsentation des Kontrollflusses von Ausnahmen in der Graph-basierten Zwischenrepräsentation FIRM und implementieren es im Compiler LIBFIRM. Wir verifizieren unsere Implementierung durch Integration in den Java-Compiler BYTECODE2FIRM.

Contents

1 Introduction

Thorough and accurate error handling is one of the most important drivers for high software quality. It prevents software from ending up in unwanted state and provides backup strategies for erroneous or otherwise unforeseen circumstances.

Most modern programming languages dedicate special syntax and semantics to error handling. In object-oriented languages this is usually realized with *exceptions*. A function that does not know how to deal with a certain situation may *throw* an exception object. Control flow is then transferred to the place the function has been called from, expecting that the caller (or any of the caller's callers, etc.) know how to deal with the situation.

Since exceptions change the control flow of a program, it is necessary to know for compilation whether an operation may throw an exception. Compilers and (perhaps more importantly) intermediate representations (IRs) must therefore include special support for exceptions.

Having its roots in compilation of the C programming language, the graph-based intermediate representation FIRM as implemented by LIBFIRM does not contain the concept of exceptions.

In this work we present a model for representation of exception flow in graph-based IRs, and in the FIRM intermediate presentation in particular. We provide a description of our model in chapter 3. We implement it in LIBFIRM and two of its frontends, CPARSER (C) and BYTECODE2FIRM (Java), and add the `-fexceptions` flag to the CPARSER frontend to enable exception representation. In BYTECODE2FIRM we enable exception representation by default. We also develop an exception handling runtime for the BYTECODE2FIRM frontend.

In chapter 4 we evaluate our model and our implementation using the well-known SPEC benchmark suite. We compare the performance of programs built by CPARSER with and without the `-fexceptions` flag enabled. Our benchmarks show that it has some noticeable but small negative effect on program performance.

2 Preliminaries

2.1 Compilers and Intermediate Representations

2.1.1 Static Single Assignment Form

Compilers are typically split into three components. The input program is first parsed by the *frontend* to an *abstract syntax tree* (AST). The *backend* deals with code generation for the target language (machine code or another high-level programming language). Between frontend and backend the input program often undergoes multiple transformation and optimization steps. These are part of the third component, the *middleend*.

As performing transformations on the syntax representation of the input program would be too cumbersome, the frontend first translates it to another representation. This representation is called an *intermediate representation* (IR). Unlike the abstract syntax tree it focuses less on the syntactical and more on the semantic aspects of the input program [1].

State-of-the-art intermediate representations for imperative programming languages are based on *Static Single Assignment* (SSA) form, which simplifies many optimizations [2]. A program is in SSA form if and only if each name ("variable") is assigned exactly once, i.e. we can identify its value with its definition.

Imperative programs are generally not written in SSA form. To obtain an input program's SSA representation, it must therefore be translated to such. This transformation is called *SSA construction*. It is based on the idea of renaming the variables in the program so that each variable has a unique place where it is assigned (i.e., defined). Consider the simple example program in figure 2.1. There the two assignments to `a` have been changed to assign to `a_1` and `a_2`; thus, the transformed program is in SSA form.

However, this raises the question of how to translate code that uses the `a` variable after the assignment in the source program. The `a` variable could refer to any of its definitions. This depends on which branch is taken when the program is executed.

If we do not know which of the two SSA versions of the variable will hold the correct value when the program is executed, how should we translate `foo(a)`? In SSA, this is accounted for by an artificial operation called a Φ function. It "selects" the correct definition depending on the actual control flow of the executed program. Note that computers cannot actually execute Φ functions; they must therefore be removed before code generation in a process called *SSA destruction*.

```
if ... {
  a = 1;
} else {
  a = 2;
}
foo(a);
```

(a) Original program

```
if ... {
  a_1 = 1;
} else {
  a_2 = 2;
}
foo(?);
```

(b) How should we translate `foo(a)`?

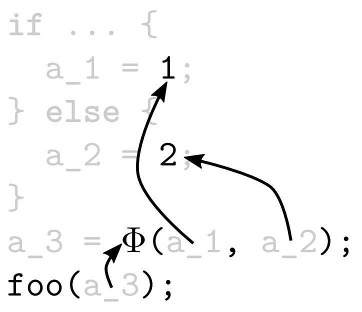
```
if ... {
  a_1 = 1;
} else {
  a_2 = 2;
}
a_3 = Φ(a_1, a_2);
foo(a_3);
```

(c) SSA form

Figure 2.1: SSA representation of a simple program.

2.1.2 Dependency Graphs

In SSA-based intermediate representations, names are identified with their unique definition. We may thus replace any occurrence of a name with its value. The intermediate representation can then do aside with the concept of variables and names altogether. The result of this is an IR that explicitly models all data dependencies: All operations refer directly to their operands' definitions. We obtain what is essentially a *graph* of data dependencies:



The idea of dependency graphs forms the basis for the work of Trapp on *explicit dependency graphs* (EDGs) [3]. In addition to data dependencies Trapp's EDGs also contain explicit control-flow and memory dependencies. Control-flow is represented using classic control-flow graphs. Blocks are linked to their control-flow predecessors. Memory dependencies are modeled using a special "memory" output of operations that may cause side effects. Details on EDGs may be found in [3].

2.1.3 The Firm Intermediate Representation

Firm [4] is an intermediate representation based on the work of Trapp [3]. It works exclusively on SSA form and can be used to represent a program in the middleend and backend stages of compilation, including code generation. Firm graphs are a form of explicit dependency graphs.

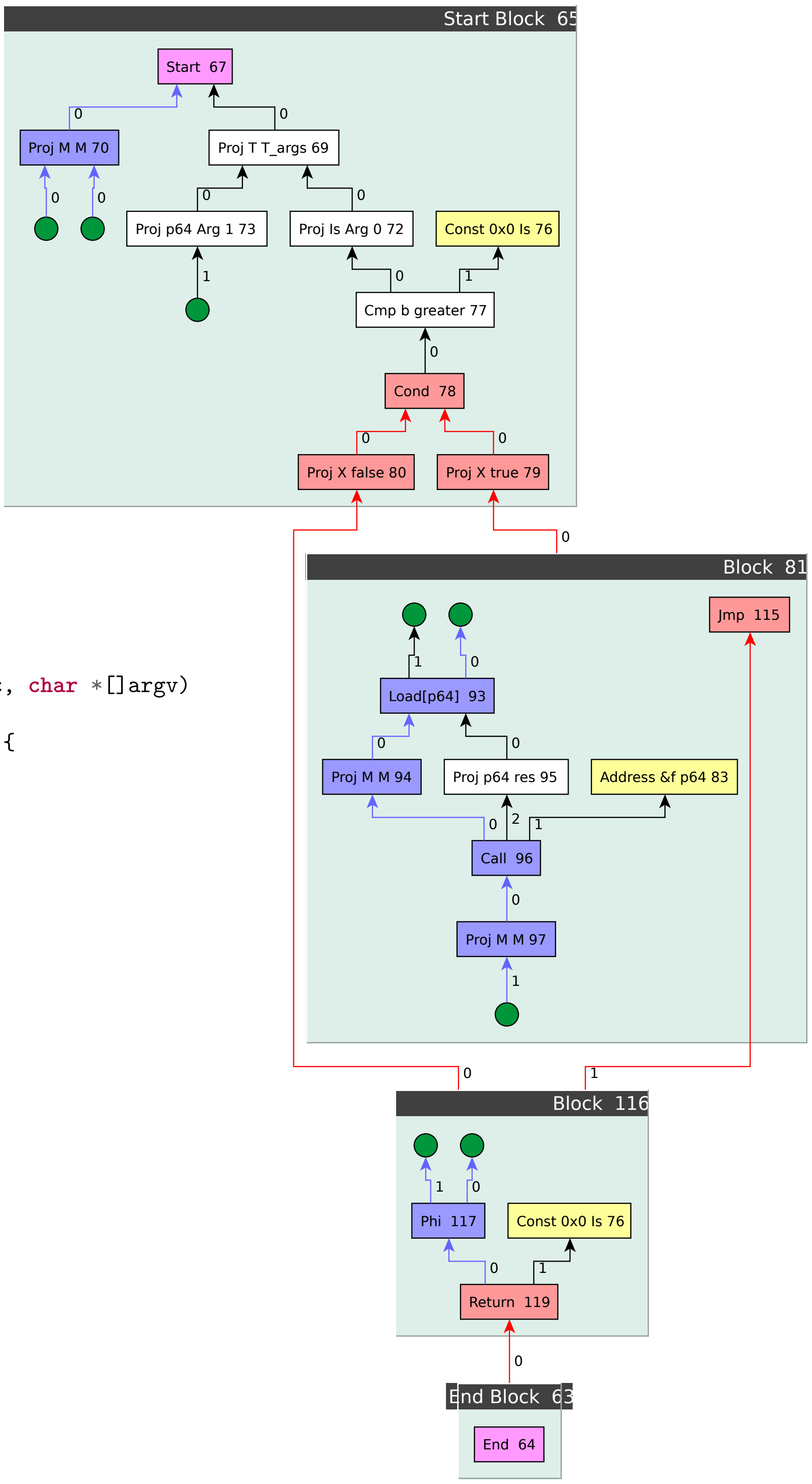
Each of the input program's functions is represented by an EDG. We introduce the structure of Firm graphs with the help of a small example program provided in figure 2.2.

Operations such as arithmetic, comparisons and control-flow statements are represented as vertices in the graph. Data and control-flow dependencies are represented as input edges. They refer to the outputs of other vertices. All operations belong to exactly one basic block, which is represented as another, special block vertex.

Operations may have more than one output. For example the `Start` vertex in our example has two outputs M and T_{args}, where M is the initial memory state and T_{args} is a tuple of function parameters. These outputs are combined into a tuple (M, T_{args}). Tuple entries may be accessed using a special `Proj` instruction. In our example the vertex `Proj Is Arg 0 72` selects the `int argc` parameter from the tuple of function parameters.

Firm graphs employ a type system that assigns a *mode* to every vertex. Built-in modes include `T` (tuple), `M` (memory state, blue vertices), `P` (pointer), `Is` (signed integer) and `b` (bit). Control-flow operations have mode `X` (red vertices).

Operations that access memory have a special mode `M` output that is the program's memory state after their execution. In our example, `Load[p64] 93` reads data from, and `Call 96` may write to memory. The behavior of the program clearly depends on the order these operations are executed in. To encode the correct execution order in the graph, the call operation is made dependent on the load operation's memory state.



```
void f(char *);

int main(int argc, char *[]argv)
{
  if (argc >= 0) {
    f(argv[0]);
  }
  return 0;
}
```

Figure 2.2: A small C program and its FIRM graph.

2.1.4 LibFirm and Related Projects

libFirm [5] is a Free [6] implementation of the Firm intermediate representation. It is written in the C programming language and features an extensive set of target-independent optimizations and code generation backends for the x86, AMD64, SPARC and ARM architectures.

Several frontends exist for libFirm, the perhaps most actively developed one being cparser [7] for the C programming language. Other frontends include bytecode2firm, which supports translation of Java bytecode [8] to Firm graphs, and X10 [9].

2.2 The Call Stack

Procedure calls are usually implemented using a runtime stack of *call frames* called the *call stack*. A procedure's call frame is put onto the call stack when the procedure is entered, kept on stack while it executes and removed from the stack when the procedure is left. The call frame holds various data that is required for the procedure to run, like the parameters it has been called with and the local variables it has created. It also contains a *return address*, i.e. the address of the code that should be executed after the procedure, which typically is the instruction after the call site (the address of the call instruction) [1].

2.2.1 Calling Convention

The call stack is usually implemented by compilers using a special *stack pointer* register whose value is shared among all procedures and may be freely adjusted to push or pop values onto or from the stack. When a procedure is left the stack pointer value is reverted to its value at entry of the procedure as to not interfere with the caller. This poses a problem if the amount of data pushed onto the stack is unknown at compile time, because in that case the compiler also cannot know by which amount the stack pointer has to be changed when leaving the procedure. In C, this situation may arise when using the `alloca` function or a variable-length array (whose size is determined at runtime).

To solve the problem of unknown stack frame sizes, an additional *base pointer* is used that holds the value of the stack pointer just after the procedure was entered. It too is stored in a special register called the *base pointer register*. Upon exit of the procedure the stack pointer is restored from the base pointer register. Since the caller likely also uses the base pointer register for this very same purpose, the "old" base pointer value is kept on the call stack so that it can be restored when leaving the procedure. Saving the base pointer register value in memory, and generally saving any register value in memory, is also called *spilling*, and the address where it is saved is called *spill slot*.

These low-level details of how to create and destroy stack frames are part of a standard called the *calling convention*. Many different calling conventions exists; the one we described above is the most commonly used one and a de facto standard for the X86 architecture [10].

2.2.2 Stack Unwinding

Debuggers (and other tools, as we will see in the next section) can use call frames to walk the call stack. In particular, they can "go back in time" to see what caused a particular procedure to be called ("trace back"), which may help understanding errors in the program. This is called *stack unwinding*.

2.3 Exceptions and PEIs

In programming, *exception handling* is used to cope with unexpected or unlikely events that happen to a computer system running a program. This may include illegal computations like division by zero resulting from bugs in the program or user input that has not been validated sufficiently thoroughly, access to files and data that do not exist, and other domain specific unlikely or error conditions.

Exceptions remove the necessity of explicit error handling as known from programming languages such as C. In C, an error in a subroutine is typically indicated by a nonzero or negative return value, while the actual result of the subroutine is written to a memory address specified by the caller. As an example, a C program that uses the `read` syscall must check for error at every call site. This gets particularly cumbersome if such calls are made in multiple places in the code (figure 2.3a). Compare this to equivalent code written in the Java programming language, which supports exception handling (figure 2.3b). Another issue with explicit error handling is that if forgotten the program may behave in unexpected ways and even be vulnerable to attacks.

We provide an example of "real world" exception handling code in figure 2.4. It shows excerpts from the source code of Java Development Kit 8.

There are many notions of *exceptions* in the literature. In this paper we use the term "exception handling" as a shorthand for *structured, non-resumable exception handling* as described in [11]. In structured, non-resumable exception handling, a function that does not know how to deal with a certain erroneous or otherwise unexpected situation may *throw* an exception. Control is then transferred to the place the function has been called from, expecting that the caller know how to deal with the situation. This step is repeated until the end of the call chain is reached, in

```c
ssize_t bytes_read;
bytes_read = read(...);
if (bytes_read < 0) {
  /* Error handling */
  ...
} else {
  ...
  while (...) {
    ssize_t tmp = read(...);
    if (tmp < 0) {
      /* Error handling */
      ...
    } else {
      bytes_read += tmp;
      ...
    }
  }
}
```

(a) Explicit error handling in C.

```java
int bytes_read;
try {
  bytes_read = f.read(...);
  ...
  while (...) {
    bytes_read += f.read(...);
    ...
  }
} catch (IOException err) {
  /* Error handling */
  ...
}
```

(b) Implicit error handling using exceptions in Java.

which case the program is aborted, or until a caller indicates that it knows how to handle the exception. The latter is called *catching* an exception, and the code that deals with the caught exception is called *landing pad*.

In object oriented programming languages exceptions are usually objects, and landing pads define which types of exception objects they can handle (*catch type*). The list of landing pads that may catch an exception is then restricted to those matching the type of the exception object that has been thrown. Here, a landing pad "matches" an exception (or type thereof) if the exception type is compatible to the landing pad's catch type.

The landing pad code has the following options to proceed with the exception situation. First, if the exception was caused by some temporary condition like a network outage, the landing pad may simply try to run the exact same code a second time. Second, if there's some other way to properly deal with the situation, like falling back to some default value, or displaying a error message, the landing pad may fully resolve the exceptional case and resume operation of the program. Lastly, if the landing pad does not know how to recover from the situation (this cannot always be known upfront), it may *rethrow* the exception, in which case the exception handling routine described above is continued.

Some programming languages provide a special *finally* instruction that is guaranteed to be executed whenever the corresponding `try` block is left. In particular, it is executed if an exception is raised in the `try` block. This is why it is commonly used to clean up resources used by the calling function (e.g. temporary files or database connections). It is a shorthand for catching all possible exceptions, doing the cleanup, and then rethrowing the exception.

Since exceptions change control flow, it is necessary to know for compilation whether an operation may throw an exception or not. This includes calls to other functions that can throw unhandled exceptions. We therefore make the following definition.

Definition ("throws", Potentially Excepting Instruction)**.** *A function or piece of code* "may throw" ("throws") *if and only if an exception that is not immediately caught can be thrown by the function or by any function it calls.*

A call to a function that may throw or some other instruction that may invoke a piece of code that may throw is called a Potentially Excepting Instruction *(PEI).*

```
/* From: java/nio/files/File.java */
boolean isSymbolicLink(Path path) {
  try {
    return readAttributes(path, ...).isSymbolicLink();
  } catch (IOException ioe) {
    return false;
  }
}

void createAndCheckIsDirectory(Path dir, ... attrs) throws IOException {
  try {
    createDirectory(dir, attrs);
  } catch (FileAlreadyExistsException x) {
    if (!isDirectory(dir, ...))
      throw x;
  }
}

/* From: java/util/concurrent/ArrayBlockingQueue.java */
ArrayBlockingQueue(int capacity, ..., Collection<? extends E> c) {
  ...
  this.items = new Object[capacity];
  this.lock.lock();
  try {
    ...
    try {
      for (E e : c) {
        ...
        items[i++] = e;
      }
    } catch (ArrayIndexOutOfBoundsException ex) {
      throw new IllegalArgumentException();
    }
    ...
  } finally {
    this.lock.unlock();
  }
}
```

Figure 2.4: Exception handling examples from Java Development Kit 8. The `isSymbolicLink` method falls back to a default answer of `false` in case of an exception. The `createAndCheckIsDirectory` method tries to create a new directory of the given path. If a file already exists at that path, an exception is thrown. The exception is re-thrown if the existing file is not a directory; otherwise, it is ignored. The `ArrayBlockQueue` constructor creates a new array with given capacity and fills it with the entries of some given collection. If at some point of the copy operation the array runs out of space, an `ArrayIndexOutOfBoundsException` is thrown, which is turned into an `IllegalArgumentException`, indicating to the caller that the specified capacity is inadequate. The constructor code also shows an example of `finally` usage. Here it is used to ensure that the lock that was obtained earlier in the code is properly released in all cases.

2.3.1 Zero Cost Exceptions

Exceptions are commonly implemented using a technique called *Zero Cost Exceptions* in compilers (GCC, Clang [12], [13], . . .). It is the de facto standard for low-level exception implementation.

The main idea of Zero Cost Exceptions is that exceptions should add negligible runtime overhead if no throwing code paths are ran into (hence the name). Due to their nature of modeling the exceptional case, the runtime performance of actually throwing and catching exceptions (including walking the call chain etc.) does not have to be heavily optimized. Note that for programming languages that feature subtyping [14] we cannot do aside with runtime support for exception handling (and thus runtime overhead) entirely. This is because in general it is not possible to decide at compile time which landing pad is to be used for a PEI.

Many compilers implement Zero Cost Exception Handling in a way compatible to the Itanium Exception Handling ABI [15]. It separates the runtime exception handling implementation into two independent parts:

- A so-called *unwind library* that knows how walk the low-level representation of the call chain (call stack).
- A *personality routine* that encodes the language-specific parts of the exception handling process like knowing how to find and select the correct landing pads.

libunwind [16] is a Free unwind library implementation with support for many CPU architectures.

A common way to implement the personality routine is to add an *exception table* to the machine code of any function or method that contains some kind of exception handling. For example, the GNU Compiler Collection and the Clang compiler use a proprietary table layout based on the DWARF `.eh_frame` format. When asked by the unwind library to determine if there is any suitable landing pad in the current call frame, the personality routine can do a lookup in the exception table. The exception table holds a list of landing pads and the types of exceptions that they can deal with.

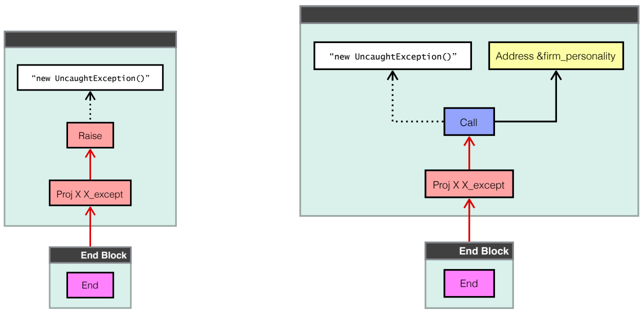
2.3.2 bytecode2firm Landing Pad Design

We explain how BYTECODE2FIRM represents `catch` blocks as landing pads (i.e., basic blocks) in FIRM.

First consider the simple case of an isolated `throw` statement—one that is not in scope of a Java `try` block—in figure 2.5. Since there cannot be any other matching landing pad in the same method, runtime exception handling must always be invoked, to look for matching landing pads further up the call chain. This is represented by BYTECODE2FIRM as a call into a special function `firm_personality`.

```
void isolated() {
  throw new
    UncaughtException();
}
```

(a) Source program


"new UncaughtException()"
Raise
Proj X X_except
End Block
End
"new UncaughtException()"
Address &firm_personality
Call
Proj X X_except
End Block
End

(b) Intermediate graph **(c)** Lowered graph

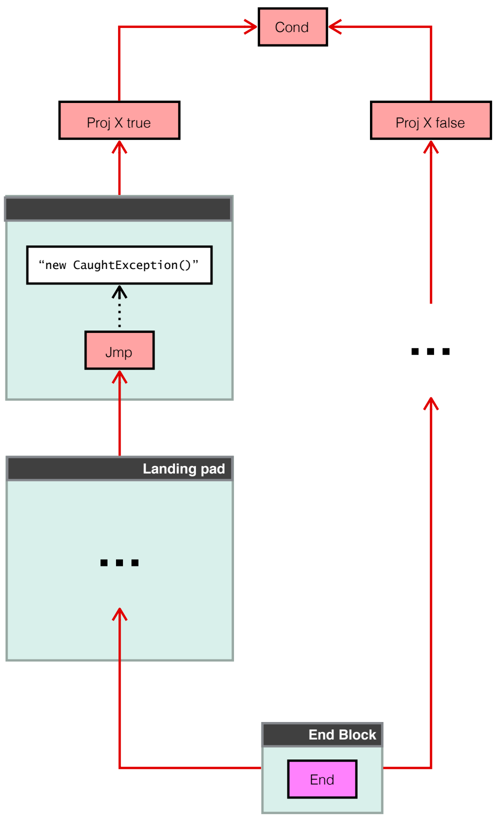
Figure 2.5: Isolated Java `throw` statements always directly invoke runtime exception handling. Note that `throw` statements are represented as `Raise` vertices in the intermediate graph, and are later lowered to a `Call` to runtime exception handling.

Another simple case is if a `throw` statement is surrounded by a `try` block and its matching landing pad is known at compile-time. The `throw` statement can then simply be translated to a jump to that landing pad (see figure 2.6).

Java allows for multiple `catch` clauses in each `try` block. BYTECODE2FIRM employs a chain of type checks using Java's `instanceof` machinery to select the correct target at runtime. In fact, the current implementation always adds a `Jump` to the beginning of the type-check chain, even if some exception types are known to never match a certain operation. In figure 2.7 we give an outline of a landing pad with instance checks. There we also account for the case that none of the catch types matches the exception: In this case BYTECODE2FIRM resorts to runtime exception handling to look for a matching handler in the calling frame. To accomplish this, it adds a "fallback" call to `firm_personality` at the end of each landing pad.

```
void knownLandingpad() {
  try {
    if ... {
      throw new CaughtException();
    } else {
      ...
    }
  } catch (CaughtException exc) {
    /* Landing pad */
    ...
  }
}
```

(a) Source program



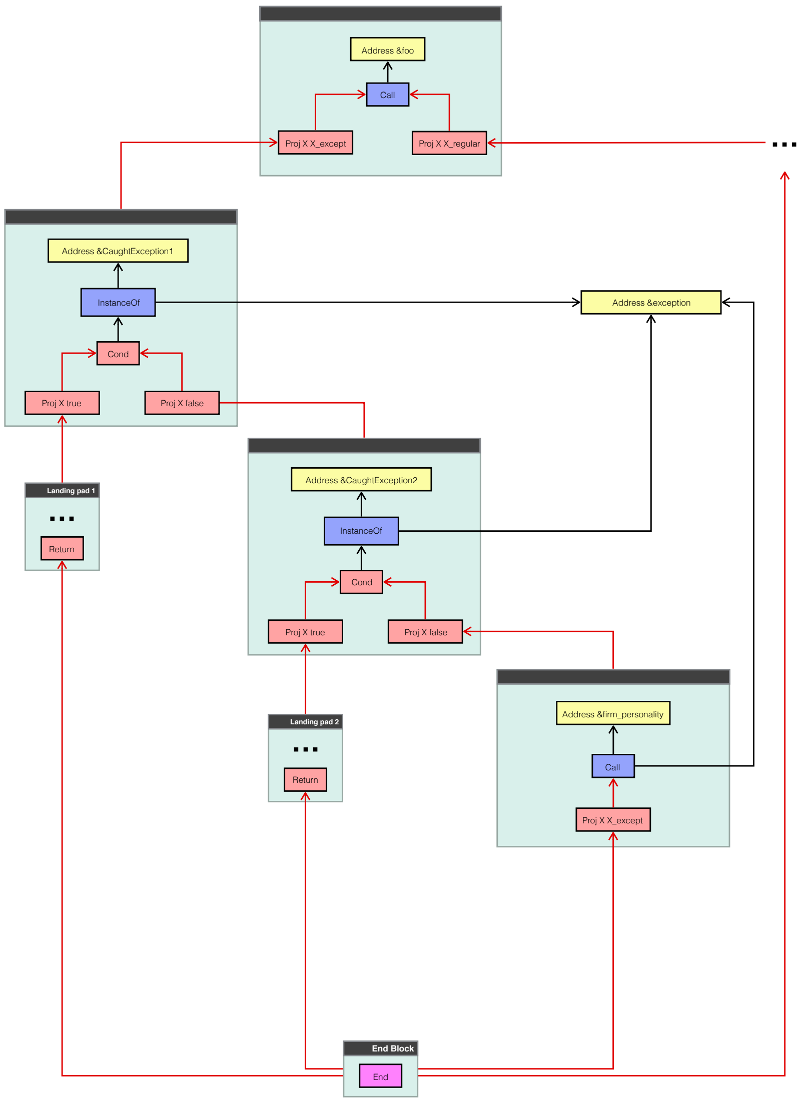
(b) Intermediate/lowered graph

Figure 2.6: If a `throw` statement's landing pad is known at compile time, it can be lowered to a jump to the landing pad.

```java
void foo() throws CaughtException1, CaughtException2, UncaughtException;

public void multipleCatch() throws UncaughtException {
  try {
    foo();
  } catch (CaughtException1 exc) { /* Landing pad 1 */
    ...
  } catch (CaughtException2 exc) { /* Landing pad 2 */
    ...
  }
}
```

(a) Source program



(b) Lowered graph

Figure 2.7: Multiple `catch` clauses are translated into a chain of type checks.

3 Design and Implementation

The main goals of our work are twofold. First, we want to adapt LIBFIRM so that it can represent exception flow in FIRM graphs. This is described in section 3.1: *Representation of Exception Flow in* LIBFIRM. Second, we want to add exception support to BYTECODE2FIRM, which also implicitly verifies that exception support in LIBFIRM functions correctly. This is described in section 3.2: *Implementation of Java Exception Handling*.

For testing purposes, we extend CPARSER with exception flow representation as well. This may also serve as a basis for C++ support in CPARSER in the future.

3.1 Representation of Exception Flow in libFirm

Several kinds of PEIs exist in the C and Java programming languages:

- Read or write access to memory may fail for illegal memory addresses.
- Arithmetic may underflow or overflow, or divide by zero.
- Function calls except if their callees except.
- `throw` statement are used to manually trigger an exception.
- ...

The effect of an exception triggered by a PEI largely depends on the type of the PEI (see list above) and on the input language semantics. For instance, a program written in the C programming language that tries to dereference a null pointer—and in general access any illegal memory address—is usually terminated by the operating system immediately. In contrast, a null pointer dereference in the Java programming language throws a `NullPointerException`. Similarly, an integer division by zero throws an `ArithmeticException` in Java, whereas it has undefined behavior in C [17].

Indeed, there exists no runtime mechanism for catching exceptions thrown by a PEI in the C programming language. In addition, BYTECODE2FIRM does not handle any of the implicit exceptions that can be thrown by a PEI in Java (like null pointer dereference, illegal array index access, division by zero). Therefore, in this work we only consider explicit `throw` statements and calls to functions containing such. We ignore the fact that other instructions may implicitly throw an exception.

We use a classic approach to represent the fact that a throwing call may change control of a program: We make it a control-flow operation in the FIRM graph, similar to the *If* and jump instructions, by adding two new `mode_X` outputs, `X_regular` and `X_except`. They represent the "regular" (no exception thrown) and exceptional cases. Code that follows the call in the regular case is put into a new basic block that is connected to the call's `X_regular` output through a `mode_X Proj` instruction; code that should be executed in case an exception was thrown is put into a new basic block connected to the `X_except` output.

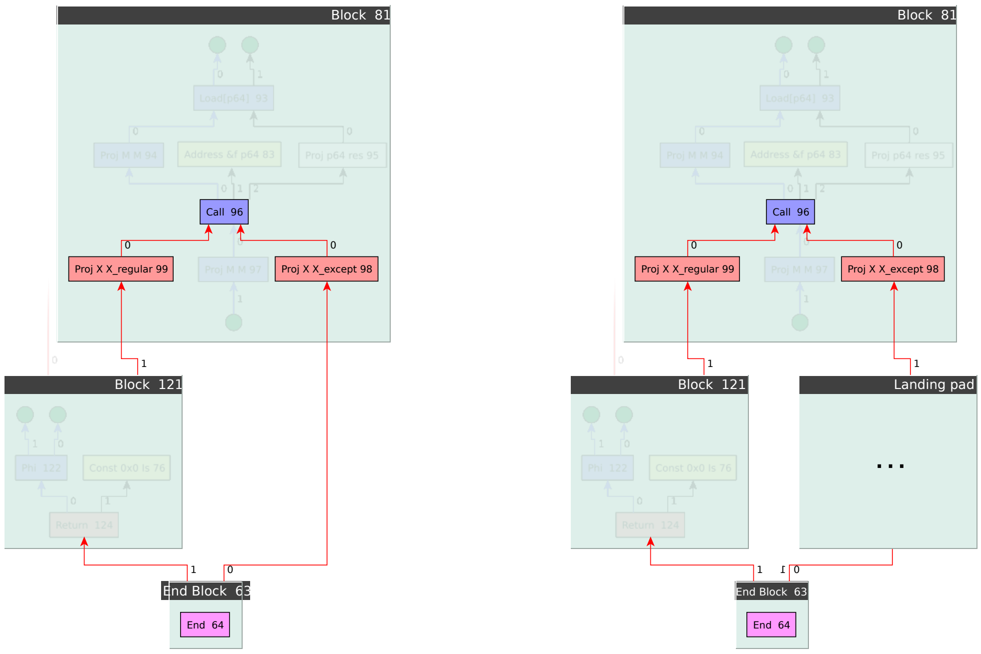
If no landing pad has been given for the call, or the language does not support specifying landing pads in the first place (e.g. C code compiled with `-fexceptions`), then we directly route the call's `X_except` output to the graph's end block.

If the program has one or more landing pads associated with the throwing call, they are placed in the `X_except` block. We will discuss the structure of these landing pad blocks in section 3.2: *Implementation of Java Exception Handling*.

In figure 3.1 we show the FIRM graph of the C program from figure 2.2 compiled with `-fexceptions` and illustrate how the graph would look if we had a landing pad to represent.

Note that despite the fact that from a FIRM graph it might look like `X_except` control flow is translated to a simple jump instruction in the code generation phase, this is generally not the case. In section 2.3.1: *Zero Cost Exceptions* we describe a common technique to implement exceptions that involves a special *unwind library* and a language-specific *personality routine*. The representation of exception flow described in this chapter is unconcerned with the details of how exceptions are implemented in the code generation phase.

The addition of the `X_regular` and `X_except` outputs alone gives raise to a multiple considerations that have to made in the various phases of our implementation LIBFIRM, some of which we will discuss in the following.



(a) Without landing pad **(b)** With landing pad

Figure 3.1: FIRM graph of C program from figure 2.2 compiled with `-fexceptions`.

3.1.1 Jump Instruction After Calls

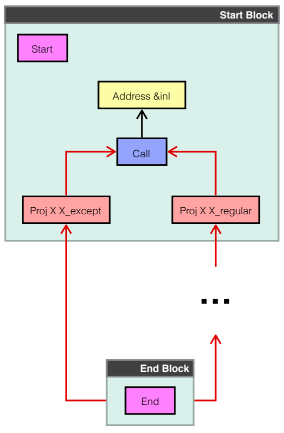
We have just seen that the code that should be executed after a throwing call is placed in a different basic block (the `X_regular` block). Code generation therefore needs to insert a jump instruction after the call (unless we can "fall through" to the next basic block). No such instruction is inserted for changing control flow to the `X_except` block for the reasons described in the introduction to section 3.1: *Representation of Exception Flow in* LIBFIRM.

To implement this change, we defined a custom emitter for `Call` instructions in each of the backends (in the file `<architecture>_emitter.c`).

```
_main:
    ...
    call _f
    jmp L1
# ... other blocks ...
.L1: # End block
    ...
    ret
```

```
void o1() {
  inl();
  ...
}
```

(a) Source code



(b) Intermediate graph

Figure 3.2: Outline of G_1.

3.1.2 The Inline Optimization

The *inlining* optimization replaces a call to a function with the called function's body. This removes the runtime overhead that comes with function calls such as copying the call arguments onto the stack and branching to some other code location (which may hurt CPU cache performance). It also allows for subsequent optimization of the code that results from the inlining process.

Some special considerations have to be made with respect to inlining functions that contain exception control flow. Consider the two FIRM graphs G_1 and G_{inl} in figures 3.2 and 3.3.

G_{inl} makes three calls that may cause an exception to be thrown:

- A call to `f`. The landing pad attached through E_f catches the exception thrown by `f` and allows G_{inl} to return without an exception.
- A call to `g`. The landing pad attached through E_g does some cleanup but leaves G_{inl} with an exception.
- A call to `h`. It has no landing pad attached and always leaves G_{inl} with an exception.

Now, when inlining graph G_{inl} into G_1, how should we proceed with the E_f, E_g, E_h `X_except` edges?

First note that the E_f branch should not actually be considered exception flow from the inliner's point of view as it ends in a normal `Return` vertex. The exception has been handled entirely inside G_{inl} and control flow continues normally. On the other hand, G_{inl} may be left through the E_g and E_h edges with an exception, from which it follows that G_1 too must be expected to have exception flow (since it does not catch the exceptions thrown by `g` and `h` in G_{inl}). We can represent this fact by connecting the E_g and E_h branches to the resulting graph's end block, as shown in figure 3.4.

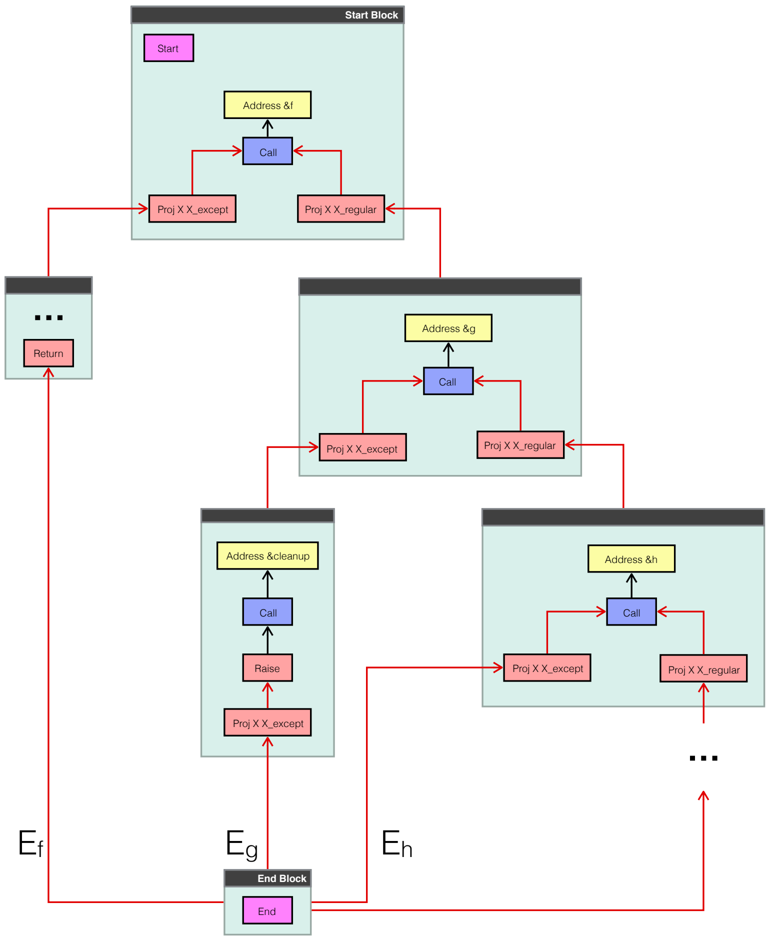
A limitation of this approach is that it does not help us if in addition to the landing pads in G_{inl} we also have landing pads in the outer graph. Consider G_2 in figure 3.5, which has a landing pad itself. How can we include this in the resulting graph?

Let use recall the behavior that is expected from the control flow of the original graph G_2 in case an exception is thrown in `inl`: We expect it to be transferred to the call's landing pad in any case, regardless of any other exception handling that may have happened inside the callee G_{inl}. We can also be sure that if an exception is thrown in the callee, control will come from one of its exception edges (E_g, E_h). Therefore, we simply connect the callee's exception edges to the "outer" landing pad that is incident to the call that is being inlined. Since by inlining that call vanishes, we do not copy the edge that connected the outer landing pad over to the inlined graph.

The final inline result graph is shown in figure 3.6.

```
void inl() {
  try {
    f();
    try {
      g();
      h();
      ...
    } catch (Exception e)
      cleanup();
      throw e;
    }
  } catch ... {
    ...
    return;
  }
}
```

(a) Source code



(b) Intermediate graph

Figure 3.3: Outline of G_{inl} that is to be inlined.

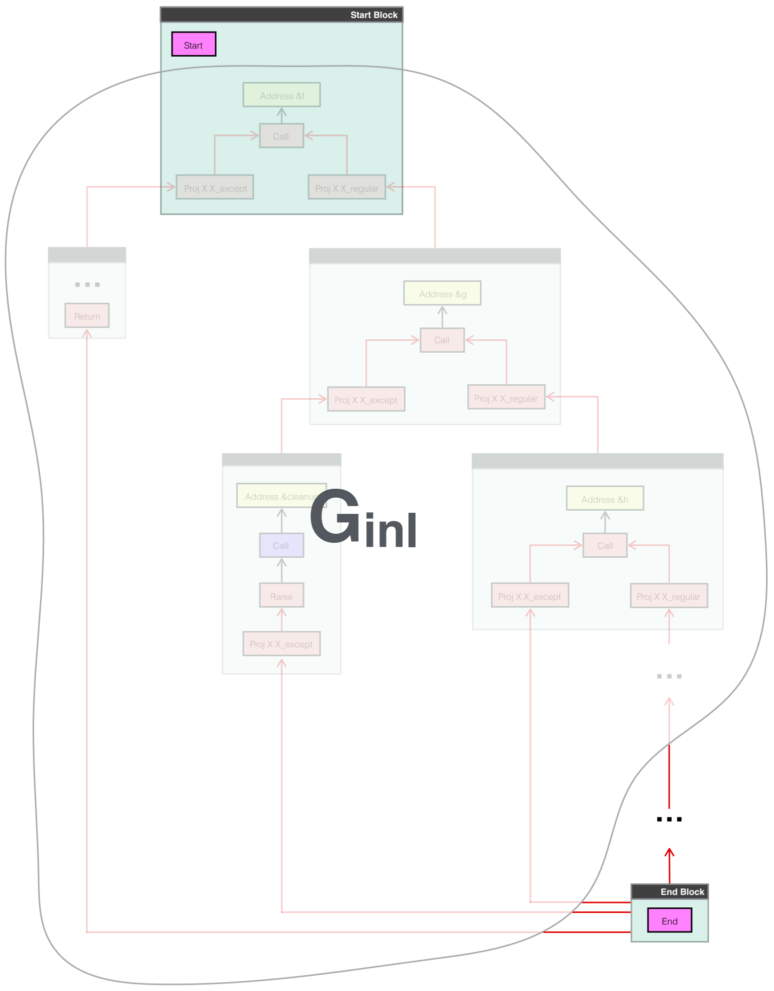
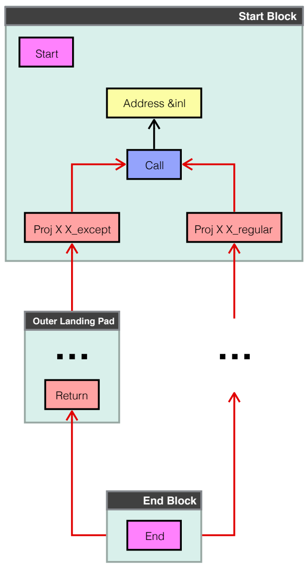


Figure 3.4: G_{inl} inlined into G_1 with exception edges simply connected to the end block. The ellipsis inside the grey area represent code that follows the call to `h` in G_{inl}; the ellipsis outside it represent code that follows the call to `inl` in G_1.

```
void o2() {
  try {
    inl();
    ...
  } catch ... {
    /* Outer landing pad */
    ...
  }
}
```

(a) Source code



(b) Intermediate graph

Figure 3.5: Outline of G_2.

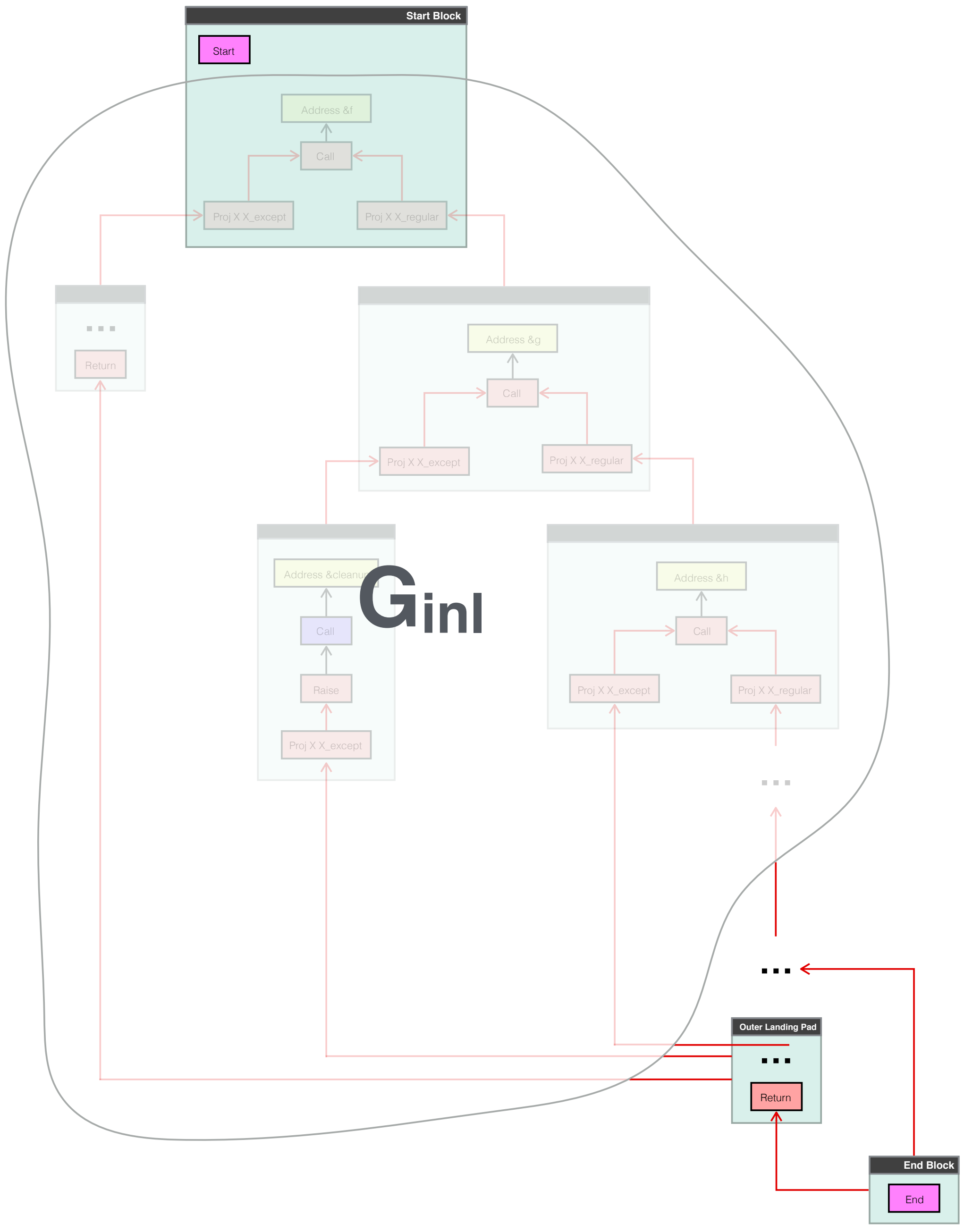


Figure 3.6: G_{inl} inlined into G_2 with exception edges connected to the "outer" landing pad.

3.1.3 Ignoring Exception Edges in the X87 Simulator

Intel's X87 [18] instruction set is a subset of the X86 architecture for fast floating point operations. All X87 instructions work on a so-called *Floating Point Stack*, that is, a set of registers exclusively used for floating point operations. X87 computations pop their operands from and push their result to the floating point stack. Compilers must therefore "wrap" X87 computations with code that pushes operands onto the stack from main memory and pops the result from the stack back to main memory.

LibFirm's X86 backends support X87 instructions for sped up floating point computations. It is implemented using a technique called *X87 simulator*. The simulator walks through the instructions of all basic blocks in the Firm graph and computes the changes done to the floating point stack. Every block has a begin and end stack state. Simulation starts with an empty stack for the start block, but the there may be blocks whose initial stack is non-empty when the simulator reaches them. This happens if control is transferred to those blocks at a point in the program where the floating point stack is not empty. In particular, every block begins with its predecessor's stack state.

If a block has multiple predecessors then it has to be made sure that the floating point values used by that block lie on the stack in same order regardless of from which predecessor block control has been transferred. This problem is solved by permuting the floating point stack at the end of the predecessor block if necessary.

A special case arises for basic blocks that are landing pads. When a landing pad is entered, the callee that threw the exception was left by a call to runtime exception handling (`firm_personality`). X86 calling conventions require that the floating point stack be empty before a call. Thus, unlike "normal" blocks, a landing pad's initial stack state must always be expected to be empty.

We satisfy this requirement by special-casing `X_except` edges in the X87 simulator (`x86_x87.c`).

3.2 Implementation of Java Exception Handling

We want to use bytecode2firm to compile and execute Java programs that involve (runtime) exception handling. For this, we must translate `catch` blocks into libFirm landing pads, and we must translate `throw` statements into a low-level equivalent that invokes runtime exception handling. For translation of `catch` blocks we incorporated the solution we presented in section 2.3.2. We now present our implementation of runtime exception handling.

Runtime exception handling deals with the situation that an exception is thrown that is not caught immediately: In particular, as described in section 2.3.2, there exists no matching landing pad in the same call frame. So, we must extend our search to the previous call frame. If we find a matching landing pad in the previous frame, we can continue execution at that landing pad. Otherwise, the search is continued in the previous-but-one frame, etc., until we either find a matching landing pad or we have reached the top of the call chain. In this case the program is simply aborted.

We use libunwind as a framework for this exception handling process. It already deals with walking the call chain and knows how to resume our program at a landing pad. For each frame libunwind sees during its walk, we check if that frame contains a landing pad matching the thrown exception. This is done using an exception table that is placed in the LSDA (*Language Specific Data Area*) section of the frame. If the lookup is successful, we instruct libunwind to resume execution of the program at the place we looked up in the table.

Note that a successful table lookup does not guarantee that the target landing pad actually *matches* the exception—this is checked using the chain of `instanceof` checks described in section 2.3.2 and may very well result in runtime exception handling being called again. Put another way, the table merely serves as way to find the beginning of the type-check chain.

Our exception table uses a simple proprietary format: it maps the addresses of PEIs to the addresses of their unique associated landing pads. To look up a landing pad in the unwind process, we perform a linear search in the exception table using the instruction pointer of the current call frame as key, and, if the lookup succeeds, resume execution at the landing pad's address.

In figure 3.7 we provide an example exception table. An outline of the Java source program is shown in figure 3.7a. It contains three calls to `f`, `h` and `g`. Exceptions thrown in `f` are being taken care of in landing pads 1 and 2; exceptions thrown in `g` and `h` in landing pads 3 and 4. As we have already seen in the previous section, runtime exception handling always starts the "type search" in the first landing pad provided for each call site. Here we first look at landing pad 1 for the call to `f` and at landing pad 3 for the calls to `g` and `h` (see table figure 3.7b). Figure 3.7c shows how the exception table is represented in assembly code. It also outlines the exact binary representation of the table, which is a "quad" (8 byte unsigned integer) for the number of entries in the table, followed by that many key-value pairs of size $2 \cdot 8$ byte each.

```
public void tableExample() {
    try {
        f();
    } catch ... { /* Landing pad 1 */
    } catch ... { /* Landing pad 2 */
    }

    try {
        g();
        h();
    } catch ... { /* Landing pad 3 */
    } catch ... { /* Landing pad 4 */
    }
}
```

(a) Outline of source program

Call	Resumption Target
f()	Landing pad 1
g()	Landing pad 3
h()	Landing pad 3

(b) Exception table

```
# void Test::tableExample()
_tableExample:
    ...
    call _f
.LE114: ...
    call _g
.LE115: ...
    call _h
.LE116: ...
.L616:          # Landing pad 1
    ...
.L615:          # Landing pad 3
    ...

___tableExample_LSDA:
    # Number of entries
    .quad 3
    # Handler for "Call f": Landing pad 1
    .quad .LE114
    .quad .L616
    # Handler for "Call g": Landing pad 3
    .quad .LE115
    .quad .L615
    # Handler for "Call h": Landing pad 3
    .quad .LE116
    .quad .L615
```

(c) Outline of generated assembly code with exception table

Figure 3.7: Exception table example.

3.2.1 Analysis of our Design

Our exception handling model supports all Java exception handling features, including multiple `catch` and `finally` clauses, re-throwing exceptions and recursion. Our exception table layout is much simpler than the ones other compilers like GCC use. Indeed, while other compilers usually encode exception class identifiers and action types (`catch`, `finally`), our design almost completely separates Java exception handling semantics from the libFirm library. Selection of the correct landing pad is encoded in a way that is transparent for the compiler. We need only provide the compiler with a target block at which the program should continue to execute in case of an exception for each PEI.

The disadvantage of this approach is that no structured data about `catch` (and `finally`) clauses is available. For example, it is impossible to decide if a certain exception will be caught without executing its matching landing pads. This also means that runtime exception handling must always execute all `instanceof` checks that appear on its way up the call chain. In addition, the exception table format used by GCC and Clang is more compact and thus uses less overall space in the target binary. This is mainly due to their usage of instruction address *ranges* as keys in the exception table, an optimization we do not yet employ.

On the other hand, we can use binary search in our runtime exception handling algorithm to find the correct landing pad. Therefore, if T is the number of PEIs in a `try` block, at most $\lg T$ steps are necessary to find the beginning of the correct type-check chain for a PEI. Finding a matching `catch` clause may then require up to C checks, where C is the number of `catch` clauses in the PEI's surrounding `try` block. Let R further be the maximum recursion depth of the program. Together we get a runtime error handling complexity of

$$O(\lg T \cdot C \cdot R).$$

An entry in the exception table takes up eight bytes for each of key and value of the entry. The table also has a header of eight bytes size. Together, we get an exception table size of

$$T * 16\text{B} + 8\text{B}.$$

3.3 DWARF Unwind Information

In section 2.2: *The Call Stack* we described how stack unwinding is related to the call stack and calling convention that is used by the compiler. In particular, an unwind library needs to be able to retrieve each call frame's predecessor stack and base pointers in order to walk "back in time".

We use the *Call Frame Information* (CFI) assembler directives specified in the DWARF standard to provide the following information for all call frames:

- The start and end addresses of the procedure's assembly code.
- The value of the stack pointer at the point of entry into the procedure ("old value").
- The value of the base pointer at the point of entry into the procedure ("old value").

For start and end addresses, the assembly code of each procedure is surrounded by `.cfi_startproc` and `.cfi_endproc` directives (figure 3.8).

For the old stack pointer value we use the `.cfi_def_cfa` directive. It defines the *Canonical Frame Address* (CFA), which is the value of the stack pointer at the call site in the previous call frame (i.e. the stack pointer before the `call` instruction in the previous frame). Its value is generally one machine word off from the stack pointer at procedure entry; thus it is trivial to compute that value from the CFA. It is given either relative to the base pointer in the current call frame, or, if the frame pointer is omitted (e.g. by using the `-fomit-frame-pointer` compiler option), relative to the stack pointer in the current frame. As the base pointer typically does not change within a procedure's code, only one `.cfi_def_cfa` directive per procedure is necessary if using a base pointer. If the Canonical Frame Address is given stack-pointer relative, a directive is necessary for each stack pointer change. See figure 3.9 for an example of the two cases.

The base pointer spill slot can be specified using the `.cfi_offset` directive. It is given CFA-relative (figure 3.10).

```
_main:
    .cfi_startproc
    pushq %rbp
    ...
    popq  %rbp
    retq
    .cfi_endproc
```

Figure 3.8: Assembly code of a C `main` procedure with CFI start and end addresses.

```
_main:
    .cfi_startproc
    pushq %rbp
    movq  %rsp, %rbp
    .cfi_def_cfa %rbp, 16
    ...
    retq
    .cfi_endproc
```

(a) With base pointer (no `-fomit-frame-pointer`)

```
_main:
    .cfi_startproc
    subq $56, %rsp
    .cfi_def_cfa_offset 64
    addq $16, %rsp
    .cfi_def_cfa_offset 48
    ...
    retq
    .cfi_endproc
```

(b) Without base pointer (`-fomit-frame-pointer`)

Figure 3.9: Canonical Frame Address directive with and without base pointer.

```
_main:
    .cfi_startproc
    pushq %rbp
    .cfi_offset %rbp, -16
    movq  %rsp, %rbp
    .cfi_def_cfa %rbp, 16
    ...
    retq
    .cfi_endproc
```

Figure 3.10: Spill slot of old base pointer value given through `.cfi_offset` directive.

3.3.1 Other Callee-Save Registers

To be able to resume execution in an arbitrary call frame—as needed in section 3.2: *Implementation of Java Exception Handling*—we must restore the state of registers when that call frame was active. Above we explain how we provide the old states of the stack and base pointer registers to an unwinder using Call Frame Information. In addition, we need to restore the values of all other callee-save registers.

These values are necessary for regular execution of the program, too: When a stack frame is left, the procedure must restore the caller's state of callee-save registers. Therefore, if a callee-save register is modified in a procedure, it is spilled before modification, so its original value can be retrieved on procedure exit.

For an unwinder to be able to restore callee-saved registers, it needs to know the spill slots used by the procedure. This information is typically provided by extending the use of `.cfi_offset` directives to all callee-save registers. However, this requires that spill slots are unique in procedure, i.e. a register is always spilled to the same slot. To illustrate this point, consider the assembly code in 3.11.

Execution always starts at `A`, then branches to either `B` or `C` and finally calls `some_PEI` in `D`. Note that different spill slots for the `r12` register are used in B and C. Now, if

```
A:
    ...
    je B
    jmp C
B:
    mov %r12, -8(%rbp)
    .cfi_offset %rbp, -16
    jmp D
C:
    mov %r12, -16(%rbp)
    .cfi_offset %rbp, -24
    jmp D
D:
    call some_PEI
```

Figure 3.11: Non-unique register spill slots generated by LIBFIRM.

we were to unwind the call stack from `some_PEI`, which of the spill slots should we expect to find `r12` at?

Contrary to other compilers, LIBFIRM's spilling implementation does not guarantee unique spill slots[1], so we cannot use Call Frame Information for providing callee-saves.

As a workaround, we force a reload of all callee-save registers before any call to runtime exception handling.

[1]There may exist branches in the procedure that do not modify certain registers, making spills of those registers redundant. Other compilers employ an optimization called *shrink wrapping* that removes the redundant spills from those branches. In LIBFIRM this optimization is inherent and cannot be disabled.

4 Evaluation

We compare the SPEC CPU2000 [19] performance of LIBFIRM proper with our development branch. CPU2000 is a suite of macro benchmarks for comparison of CPU and compiler performance. It consists of several medium-sized integer and floating point benchmark programs whose performance is evaluated using a standardized set of input data.

For C code compiled without `-fexceptions`, we expect our branch to have no significant slowdown, since the changes made should have no effect on compilation without exception support. A small slowdown is expected for compilation with `-fexceptions`: Some optimizations limit their work to block level, and since `-fexceptions` adds a new `X_regular` block for every PEI, fewer instructions overall are covered by these optimizations.

The benchmarks were run on an Intel Core i7 2600 (quadcore, 3.40 GHz) machine with 16 GB RAM, using the CPARSER frontend. We ran the suite three times, each of which already executes each benchmark three times. Table 4.1 shows our benchmark results. Runtime is given in arithmetic mean seconds of the three runs.

As expected, there is a slight performance decrease of 0.6% average when `-fexceptions` is enabled. The time it takes to compile all programs of the SPEC CPU2000 suite increases by about 10.6% average (2.3% standard deviation) when `-fexceptions` is enabled. In table 4.2 we provide some other basic metrics of the assembly code generated by our branch with and without exceptions enabled.

To evaluate the effect of exception support on Java programs, we obtained some basic statistics on the graphs constructed by BYTECODE2FIRM using LIBFIRM's `statev` module. We used the test cases provided with the BYTECODE2FIRM test suite as basis for our statistics. None of the cases involve exception handling; thus they are suitable for assessing the effect of our exception representation model in FIRM graphs. We expect the number of basic blocks to increase compared to compilation without exceptions, and the number of instructions per basic block to decrease by a factor of roughly 3 (see chapter 6: *Related Work*).

Table 4.3 shows a comparison of the graphs constructed by BYTECODE2FIRM with exception support enabled and disabled. The expected decrease of instructions per basic block can be seen on the very right; it averages 2.4. We also see a standard deviation of 3.15 that is larger than the mean—it is due to the fact that the code samples we used are not representative for "real world" Java programs. We cannot

Table 4.1: SPEC CPU2000 results (Mean runtime in seconds; less is better). "Baseline" is performance without `-fexceptions`. σ is the sample standard deviation runtime of 100 benchmark runs. Statistically confidence of runtime change is given by p values on right.

Benchmark	**Baseline**		**`-fexceptions`**		**Change**	
	Runtime	σ	Runtime	σ	Runtime	p Value
164.gzip	62.74	0.20	62.47	0.17	-0.43%	0.000
175.vpr	47.06	0.38	46.78	0.28	-0.59%	0.000
176.gcc	23.52	0.12	23.15	0.09	-1.57%	0.000
177.mesa	58.82	0.58	58.72	0.25	-0.16%	0.116
179.art	28.04	0.17	28.59	0.48	1.96%	0.000
181.mcf	22.60	0.19	22.56	0.18	-0.15%	0.191
183.equake	30.67	0.55	30.41	0.28	-0.84%	0.000
186.crafty	28.72	0.11	28.65	0.11	-0.25%	0.000
188.ammp	92.65	0.73	92.68	0.90	0.03%	0.700
197.parser	60.67	0.16	60.76	0.16	0.16%	0.000
253.perlbmk	56.13	0.16	56.57	0.16	0.78%	0.000
254.gap	28.49	0.22	31.05	0.14	8.98%	0.000
255.vortex	43.48	0.32	43.79	0.38	0.72%	0.000
256.bzip2	49.81	0.22	49.30	0.26	-1.02%	0.000
300.twolf	66.40	0.28	67.58	0.30	1.78%	0.000
Geometric mean					0.6%	

Table 4.2: `-fexceptions` assembly code metrics. "Baseline" is performance without `-fexceptions`.

Metric	**Baseline**	**`-fexceptions`**
# basic blocks	164 139	170 526 (+3.9%)
# instructions	1 612 577	1 648 548 (+2.2%)
# Jump instructions	81 648	86 786 (+6.3%)
# instructions/block (mean)	9.82	9.67 (-1.5%)

Table 4.3: Comparison of FIRM graphs constructed by BYTECODE2FIRM with and without exception support.

Test case	With exceptions			Without exceptions			Change
	Blocks	Insns	$\frac{\text{Insns}}{\text{Block}}$	Blocks	Insns	$\frac{\text{Insns}}{\text{Block}}$	$\frac{\text{Insns}}{\text{Block}}$
AccessStaticVariable	4	57	14.3	2	47	23.5	1.7
Arrays	507	3784	7.5	290	3148	10.9	1.5
Classes	17	144	8.5	6	109	18.2	2.1
ControlFlow	621	3794	6.1	343	3187	9.3	1.5
CreateObject	16	120	7.5	2	88	44.0	5.9
EntityCopies	11	121	11.0	5	97	19.4	1.8
HelloWorld42	3	23	7.7	1	14	14.0	1.8
InstanceOf	335	2254	6.7	252	2027	8.0	1.2
InstanceVars	254	1977	7.8	91	1550	17.0	2.2
InvokeX	222	1879	8.5	29	1484	51.2	6.1
OOO	31	376	12.1	10	319	31.9	2.6
PrimArith	672	4397	6.5	269	3703	13.8	2.1
SimpleArrayTest	12	90	7.5	7	71	10.1	1.4
SimpleCall	6	65	10.8	2	48	24.0	2.2
Strings	54	419	7.8	3	307	102.3	13.2
Geometric mean							2.40
Standard deviation							3.15

use a more representative sample due to the limited number of Java programs that BYTECODE2FIRM can translate correctly.

We did not study the performance implications of exception support due to the lack of Java benchmarks that BYTECODE2FIRM can compile. This is left for further work.

5 Conclusion and Further Work

We have presented our model for representation of exception flow in the Firm intermediate representation. In our model, potentially excepting instructions (PEIs) end their basic blocks and split control flow into exceptional and regular (no exception thrown) branches. The addition of our exception representation to the libFirm compiler did not require substantial refactoring: the largest change we had to make was the adaption of libFirm's inline optimization process with respect to how exception edges are handled. In total, of libFirm's roughly 115 000 lines of code, we changed less than one percent.

We showed that our model can be used to represent "real world" exception handling by integrating it into the bytecode2firm Java compiler. We also implemented a exception handling runtime on the basis of our Firm exception representation. It uses an exception table that is a simple map of PEI addresses to their unique associated landing pads. With our changes, bytecode2firm can be used to compile Java programs that involve (runtime) exception handling.

Our experimental shows that our representation can cause a significantly smaller instruction/basic-block density. In particular, our measurements show a 2.4 times smaller mean density for Java programs.

In code generation, we must insert a jump instruction after a PEI if and only if the PEI's regular successor block is *not* scheduled directly after the PEI's containing block. In the future, libFirm's block schedulers may be adapted so that regular successor blocks are preferably scheduled directly after PEIs, making the jump instruction redundant.

Our SPEC2000 benchmarks show that C programs compiled with `-fexceptions` (which enables exception representation in cparser) have slightly worse performance than programs compiled without the flag enabled. Analysis of the generated machine code shows that the two main causes for the slowdown are the jump instructions explained above, and a combination of other peephole optimizations that were missed due to smaller basic blocks. We believe that these optimizations can be adapted accordingly, which remains for future work.

Both bytecode2firm's exception handling runtime and exception table generation in libFirm benefit from the simplicity of our exception table format. The runtime type-checks, however, are costly; they will easily dominate the overall cost for runtime exception handling in most cases. Performance of exception handling may be improved in the future by adopting an exception table format similar to the one GCC uses. It encodes the types of exceptions that are caught by a particular handler, making most of the type checks redundant.

The number of type checks that have to be done by the exception handling runtime may further be reduced using static analysis. In particular, our implementation tests all `catch` clauses for a match that it "sees" when walking the call chain. Static analysis could provide us with different "entry points" into the type-check process.

6 Related Work

In their work on Marmot [20], Fitzgerald et al. model exception flow as control flow edges originating at *blocks* rather than *operations*. Any block that contains a PEI has an edge to each of the block's landing pads, i.e. an edge for each type of exception to be caught. PEIs do not end basic blocks. Exception edges have different semantics than normal control-flow edges: While normal edges have a unique source and destination, exception edges have as source all PEIs in the block. For runtime support, Fitzgerald et al. use an instruction pointer range based approach similar to the one we employ.

The advantage of the approach taken with Marmot is that block-level optimizations can generally give better results. However, it complicates SSA construction and data-flow analysis like computation of dominance frontiers. In addition, block-level exception edges are too coarse-grained for some of Marmot's analyses, requiring an additional bit-vector of exception information about each operation in a block.

A similar approach was taken by Choi et al. with *Factored Control Flow Graphs* (FCFGs) [21]. They are similar to the graphs used in the Marmot system in that they use block-level exception edges. FCFGs too require that PEIs be special-cased in data-flow analysis. Choi et al. show that FCFGs reduce the number of basic blocks necessary to represent Java programs by a factor of roughly 3. However they do not compare performance of their Java compiler "Jalapeño" with and without FCFGs, neither do they present benchmarks for Java programs compiled with Jalapeño.

The Java HotSpot compiler [13, chapter 2.5] uses the same concept.

In the initial design of the FIRM intermediate representation, PEIs also do not end the basic block. Instead an abstract variable `Except` is used to group operations that are enclosed by a `try` block [3, 22]. Operations in a protected region (`try` block in the source language) depend on a common `Except` instance that marks the beginning and end of the protected region. This guarantees that the grouped operations cannot be moved out of the protected region. If the input language puts further restrictions on the order of the operations in the protected region, this can be modeled by introducing additional definitions of `Except` variables. For example the Java programming language defines *precise* exceptions which require that the operations in a `try` be kept in the order they are given in the input program.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*. Addison wesley, 1986.

[2] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 12–27. [Online]. Available: http://doi.acm.org/10.1145/73560.73562

[3] M. Trapp, "Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen." Ph.D. dissertation, Oct. 2001.

[4] M. Braun, S. Buchwald, and A. Zwinkau, "Firm—a graph-based intermediate representation," Karlsruhe Institute of Technology, Tech. Rep. 35, 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470

[5] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sep. 2002. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1717

[6] "What is free software and why is it so important for society?" [Online]. Available: https://www.fsf.org/about/what-is-free-software

[7] M. Braun, C. Mallon *et al.*, "cparser - a C99-frontend." [Online]. Available: http://pp.ipd.kit.edu/git/cparser/

[8] "Java language and virtual machine specifications." [Online]. Available: https://docs.oracle.com/javase/specs/

[9] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau, "An X10 compiler for invasive architectures," Karlsruhe Institute of Technology, Tech. Rep. 9, 2012. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112

[10] A. Fog, "Calling conventions for different C++ compilers and operating systems," Tech. Rep., Dec. 2015. [Online]. Available: http://agner.org/optimize/calling_conventions.pdf

[11] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[12] "Exception handling in LLVM." [Online]. Available: http://llvm.org/docs/ExceptionHandling.html

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot^{TM} client compiler for Java 6," *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 1, pp. 7:1–7:32, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1369396.1370017

[14] J. Reynolds, *Theories of Programming Languages.* Cambridge University Press, 1998. [Online]. Available: https://books.google.de/books?id=X_ToAwAAQBAJ

[15] "Itanium C++ ABI: Exception handling, revision 1.22." [Online]. Available: https://mentorembedded.github.io/cxx-abi/abi-eh.html

[16] "The libunwind project." [Online]. Available: http://www.nongnu.org/libunwind/

[17] I. Jtc, "Sc22/wg14. iso/iec 9899: 2011," *Information technology—Programming languages—C. http://www. iso. org/iso/iso_catalogue/catalogue_ tc/catalogue_detail. htm*, 2011.

[18] "Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture," Tech. Rep., May 2011. [Online]. Available: http://download.intel.com/design/processor/manuals/253665.pdf

[19] "Standard performance evaluation corporation." [Online]. Available: https://spec.org/

[20] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: An optimizing compiler for java," *Software-Practice and Experience*, vol. 30, no. 3, pp. 199–232, 2000.

[21] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for the analysis of java programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 5. ACM, 1999, pp. 21–31.

[22] M. Trapp, G. Lindenmaier, and B. Boesler, "Documentation of the intermediate representation FIRM," Universität Karlsruhe, Fakultät für Informatik, Tech. Rep. 1999-14, Dec 1999. [Online]. Available: http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz

Erklärung

Hiermit erkläre ich, Jonas Haag, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift