Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume59.html` 14 pages

Annotations for Portable Intermediate Languages

Fermín Reig [1,2]

Department of Computing Science
University of Glasgow
Scotland, UK

Abstract

This paper identifies high-level program properties that can be discovered by static analysis in a compiler front end, and that are useful for classical low-level optimizations. We suggest how intermediate language code could be annotated to convey these properties to the code generator.

Keywords: language-independent back ends, code generation, code optimization, compilers, garbage collection.

1 Introduction

Writing a state-of-the-art code generator that targets several architectures entails a high implementation effort. At the very least, the code generator performs instruction selection, register allocation and emission of assembly language or machine code. In practice we want the code generator also to perform low-level optimizations (independent of the source language), such as instruction scheduling, common subexpression elimination, loop-invariant code hoisting, unreachable code elimination, copy propagation, peephole optimizations, etc.

Because of the costs involved in implementing an optimizing code generator, using an off-the-shelf, source-language-independent back end is very appealing to compiler writers. MLRISC [GL00], VPO [BD88] and the GCC back end [Sta01] are all examples of freely-available code generators.

Portable compiler target languages are also of interest to compiler writers. C has been used as a portable assembly language in compilers for Prolog,

[1] Funded by a doctoral grant from Gobierno de Navarra.
[2] Email:`reig@dcs.gla.ac.uk`

Mercury, Scheme, SML, Haskell, APL, and many more. C ensures wide portability and reasonable performance, since good C compilers exist for virtually every target. More recently, JVM bytecode has been gaining popularity as a portable assembler for source languages other than Java. `C--` has been proposed as a portable compiler target language that lacks the shortcomings of C and JVM bytecode as assembly languages [JRR99].

To obtain high-quality object code, it is important that the compiler front end shares with the back end any high-level information that can be used to perform aggressive low-level optimizations. Program information that may be readily discovered by the front end can be expensive or even impossible to rediscover in the back end. In monolithic compilers, the different phases can share information about the source program via in-memory data structures like symbol tables or program dependence graphs, or via auxiliary files for cross-module optimizations.

When using an off-the-shelf back end, the only way to communicate program properties to the code generator is via constructs in the intermediate language. The portable back ends and intermediate languages mentioned above provide little or no support to encode high-level semantic information. For instance, with VPO the only information that can be passed to the code generator is whatever can be expressed as register transfer lists.

Our contribution is to identify high-level program properties that can be used to perform aggressive back-end optimizations and/or reduce optimized compilation time. We focus on static program knowledge that is readily available at the front end, or that is the result of source-language-dependent analysis. We propose suitable annotations for intermediate languages to convey this information to the code generator.

There exist several systems that use annotations to pass program information to the back end, but the ones we know are specific to one source language (see the related work in section 3). In this paper, we are interested in communicating static program properties to *language-independent back ends*, regardless of the source language (which may be procedural, object-oriented, or declarative).

2 Annotations for Low-Level Optimizations

Compiler front ends for modern programming languages have abundant information about programs: types, side effects, control flow due to exceptions, etc. We shall assume the following principle:

Do not throw away high-level information that can be exploited for low-level optimizations and that the back end cannot (easily) recover.

In this section we propose a series of annotations to a generic intermediate language to convey static program properties to the code generator. The annotations can be inferred by the compiler, but may also be provided by the programmer, in source languages that support it. In the latter case, the compiler translates source-level annotations into annotations of the intermediate language. Since we do not refer to any specific existing back end or intermediate language, we do not propose concrete syntax. We intend the annotations as suggestions to designers of compiler intermediate languages.

N.B. In the rest of the paper, uses of the term *procedure* can be generalized to either procedure, method, or function.

2.1 Control Flow of Procedure Calls

Some procedures do not return normally to their caller, but terminate the program abruptly. The standard libraries of many programming languages include non-returning procedures: `exit`, `abort` and `longjmp` in C; `System.exit` in Java; `error` and `System.exitWith` in Haskell; etc. In GNU C, programmers can annotate functions as `noreturn` [Sta01].

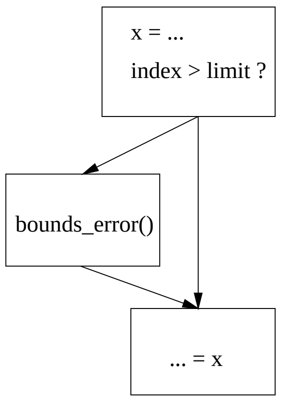
We propose that calls in the intermediate language can be annotated as non-returning. The back end can use this information to remove unreachable code and to perform better register allocation.

If a call to procedure P does not return, control will never reach the instruction immediately following the call. In the back end, the unreachable code elimination phase can delete all code that is dominated by the call. (A statement S_1 *dominates* statement S_2 if every control path that reaches S_2 passes through S_1.)

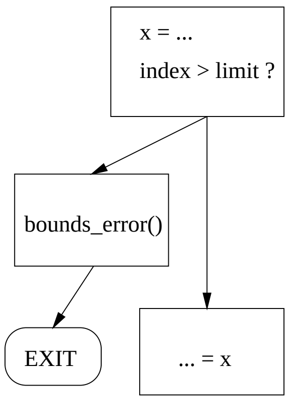
Sometimes, calls to non-returning procedures are not present in the source program, but are generated by the front end during the translation to low-level code. For instance, in a language with run-time error checking, the front end emits an array-bounds check for every array access (except where it can statically determine that the index is within bounds). These checks typically contain a call to a non-returning error procedure. The low-level code may end up with many such calls that do not return. If they are identified, the back end can perform better register allocation. For instance, in the following code sequence:

```
x = ...;
...
if (index > limit) {
   array_bounds_error();
}
... = x;
```

`x` cannot be allocated to a scratch (caller-saves) register, since this register might be overwritten by `array_bounds_error`. The register allocator can either spill `x` across the call, or allocate it to a callee-saves register. However, if the call is annotated as non-returning, `x` can be allocated to a scratch register, which is cheaper than spilling or using a callee-saves register. (Using a callee-saves register in a procedure imposes an indirect cost of saving it before its first definition and restoring it after its last use.) Essentially, the annotation lets the back end transform the control-flow graph (CFG) as shown in figure 1. In the CFG 1(b), `x` is not live across the call to `array_bounds_error`, and thus may be allocated to a scratch register.



(a) CFG when the call does return.



(b) CFG when the call does not return.

Fig. 1. Effect of annotation on the CFG.

2.2 Control Flow due to Exceptions

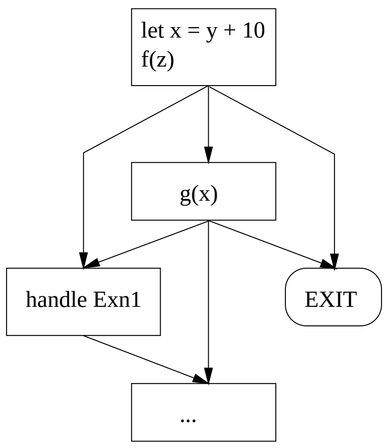
In programming languages that provide an exception mechanism, this is the preferred mechanism for signalling error conditions. For instance, Java throws `IndexOutOfBoundsException` for illegal array accesses, rather than terminating the program.

In a control-flow graph representation of the program, a call that may raise an exception terminates a basic block and creates a control-flow edge to the handler as well as an edge to the next statement after the call. For example, this Caml code:

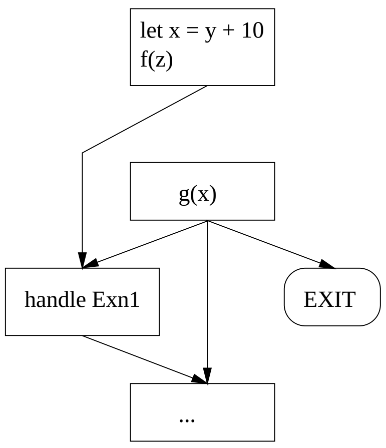
```
let x = y + 10 in
(try
  f(z); g(x)
with Exn1 ->
  handle_exn());
...
```

has the CFG of figure 2(a). Notice that only `Exn1` is caught in this example. If any other exception were thrown by `f` or `g`, control would flow to the `EXIT` pseudo-node of the CFG.

A portable intermediate language must have a way of conveying to the back end the additional control-flow edges due to exceptions. Without them,



(a) Additional flow edges due to exceptions.



(b) CFG when `Exn1` is guaranteed to be thrown.

Fig. 2. CFG due to exceptions.

the code generator cannot build an accurate CFG. In `C--`, extra edges for exceptions are expressed via the `also` annotation [RPJ00]. MLRISC has a similar construct. For the example above, in `C--` the calls `f(z)` and `g(x)` require an `also` annotation to the handler code.

Exception analysis [YR01,LP00] is useful to diagnose programming errors due to possibly uncaught exceptions. Interestingly, knowing that an exception is always thrown is useful information for back-end optimizations. We propose an `always` annotation for these situations. The annotation carries the control-flow successor of a call. In `C--`, it carries a *continuation*, in MLRISC it carries a label, and similarly in other intermediate languages. With `always` annotations, the back end can simplify the CFG by removing edges that will never be taken at run-time. In the example, if the static analysis can prove that `f(z)` always throws exception `Exn1`, the call can be annotated to give the CFG of figure 2(b).

The annotation makes it clear that the only possible successor to the call `f(z)` is the exception handler. The code `g(x)` is now unreachable. Moreover, if there are no other uses of `x`, the definition `let x = y + 10` is dead code and can be deleted. This, in turn, might enable further dead code elimination if, for example, `y` is used only in the definition of `x`.

On the other hand, if the static analysis can prove that `f(z)` always throws an exception different from `Exn1`, then the call can be annotated as non-returning (as described in section 2.1) to indicate that the only successor is the `EXIT` pseudo-node. In this case, everything dominated by the call in the original CFG is now unreachable.

All the front end has to do is annotate the statement `f(z)` during the translation to intermediate code. Standard back-end optimizations like unreachable and dead code elimination take care of the rest.

2.3 Branch Prediction

Accurate branch prediction is extremely useful for several low-level optimizations. Code motion [Hai98], control and data speculation [DKK^{+}99], instruction scheduling [YS98] and good spill-cost estimates in register allocation all benefit from accurate predictions of the outcome of conditional branches.

The back end can make good guesses for some branches: backward loop branches are likely to be taken, equality tests of floating point numbers are likely to yield false, etc. (Ball and Larus [BL93] apply simple local heuristics to MIPS machine instructions and report average hit rates of 80%.) However, using knowledge of high-level properties of the program, the compiler front end can make better predictions than the back end of the outcome of some conditional branches. Below we list a few examples.

The conditional tests of runtime error checks inserted by the front end (arrays bounds checking, invalid argument type for an operation, etc) are likely to fail. In functional languages, pattern matching on lists is more likely to succeed on non-empty lists. If a program contains a non-exhaustive pattern match, the compiler typically inserts a catch-all pattern with code that reports a pattern match failure (by raising an exception, for instance). The conditional branch to this code can be annotated by the front end as very likely not to be taken. Dynamic allocation of objects in garbage-collected languages is translated in many implementations into low-level code similar to this:

```
ptr = allocate_memory(sz);
if (ptr > limit) {
  garbage_collect();
}
... use ptr ...
```

The comparison `ptr > limit` will fail in most cases. In GNU C, there is a primitive mechanism for the programmer to provide branch prediction information (`__builtin_expect`).

These are a few of many possible examples. In all cases, the front end has the advantage of high-level information to make better predictions than the back end.

We propose that all constructs of the intermediate language that have multiple control-flow successors accept probability annotations. These constructs include conditional branches, `if then else`, `switch`, and indirect jumps where the possible targets are listed. The extra control-flow edges for exceptions introduced by `also` annotations (section 2.2) can also be annotated with branch probabilities.

2.4 Data Prefetching

Data prefetching aims to reduce the cost of cache misses by transferring data from main memory to the processor before it is actually used in a computation. Many current processors have some form of prefetch instruction. Judicious placement of prefetch instructions can substantially reduce the execution time of memory-bound programs. The compiler back end can insert prefetch instructions during the instruction scheduling phase. Occasionally, however, it is the front end and not the back end that knows best where it can be profitable to insert a prefetch instruction. In these cases, a portable intermediate language should be equipped with prefetch annotations.

In garbage-collected languages, writes to newly-allocated objects have cache miss rates close to 100%. (On machines with write-validate caches, write misses do not cause the processor to stall. But on machines with fetch-on-write or write-around policies write misses are costly.) If we use sequential allocation from a contiguous free space, prefetch annotations can be inserted at allocation points [App98]. Sometimes, high-level language constructs can serve as prefetch hints: in C99 [ISO99], the use of `static` in this function prototype

```
float dot_product(float x[static 6], float y[static 6]);
```

is a promise to the compiler that `x` and `y` point to arrays containing at least six floats. A C99 front end can emit a prefetch annotation before every call to `dot_product`. Finally, programmers can provide prefetch hints at the source level that are to be translated to prefetch annotations in the low-level intermediate language.

2.5 Pointer Maps for Accurate Garbage Collection

To support accurate garbage collection, compilers emit *pointer maps* that describe to the collector the location of local variables and temporaries that contain pointers to heap-allocated objects. Pointer variables local to a procedure can be either in registers or in the activation frame of that procedure. The compiler back end emits a pointer map for every program point where a procedure can be suspended for garbage collection—*gc-points*. Allocation points are gc-points, since an allocation is precisely what can start a garbage collection. Function calls are gc-points as well, since the callee, or anything it calls, may try to allocate memory.

The total size of the pointer maps of a program varies from system to system. The Java compiler described in [ADM98] builds maps that consume an average of 57% of the JVM bytecode. Tarditi [Tar00] uses a compact representation to achieve an average of 3.6% of code size for optimized Java programs compiled to the x86. Compact representations and compression schemes trade space consumption for increased decoding time during garbage collection.

Not enough attention has been paid in the garbage collection literature to the issue of *unnecessary* pointer maps. (To our knowledge, only [DMH92] mentions it, but very briefly.) Many call sites are not gc-points, because the callee (and anything it calls transitively) does not allocate memory during its execution and therefore cannot trigger a garbage collection. The pointer maps for all these call sites will never be consulted by the garbage collector, and therefore waste space in the object code. This is highly undesirable for systems where memory is scarce, like PDAs or hand-held computers. The standard libraries of many garbage-collected languages contain a wealth of procedures that cannot not start a garbage collection (mathematical, character manipulation, traversal of data structures, I/O, interface to the operating system, etc). Calls to these procedures can be annotated by the front end, so that the back end does not emit pointer maps for them.

In a multithreaded environment, however, all calls are potentially gc-points, even if the called function itself does not allocate: if the active thread starts a collection, the collector has to scan the stack and registers of suspended threads as well. Thus, in the presence of multithreading, pointer maps are needed for all suspension points, and no call should be annotated as not being a gc-point. However, even in languages that support threads, many programs are single-threaded. If the compiler can determine this for a program, then it is safe for the front end to annotate calls that are not gc-points.

2.6 Memory Aliasing

Disambiguation of memory accesses is vital for several back-end optimizations including redundant load/store elimination, scheduling for hiding memory latency, load/store hoisting, and common subexpression elimination of load/stores. For instance, in the code shown below:

```
*p = ...;
if(...) {
  ... use p ...
} else {
  x = *q;
}
...
```

if there are no other uses of `p`, the statement `*p = ...;` can be hoisted into the true branch, provided that `p` and `q` cannot point to the same memory location.

We propose annotating loads, stores and calls with sets of tags that represent abstract memory regions. Two memory access operations may refer to the same run-time location if their tag sets intersect. Calls take two sets of tags, `ref` and `mod`, for locations that are read and written, respectively. If a procedure accesses only private memory, calls to it are annotated with empty `mod` and `ref` sets. This style of alias annotations is folklore; it is described, for example, in [App98].

2.7 Side Effects of Functions

Some functions do not perform I/O operations, nor access the global variables of the program, nor raise exceptions. A *pure* function returns the same result when called with the same arguments, independent of the calling context. In Haskell, all functions that do not have monadic types are pure. In ML, type and effect systems [BK00] can assign non-computational types to pure functions. The standard libraries of many programming languages contain many pure functions (mathematical, string manipulation, etc.). In GNU C, programmers can annotate function declarations as `pure` or `const`. (The return value of a `pure` function depends on the parameters and/or global variables. `const` is more strict, since the result may not depend on the contents of global variables.)

In the compiler back end, calls to pure functions are subject to dead code elimination and code motion optimizations, like loop invariant hoisting, common subexpression elimination, and partial redundancy elimination [BC94]. For example, in the following code:

```
y = cos(z);
...
for (...) {
  vec[i] = cos(x) + ...
}
```

if the call to the cosine function has no side effects and there are no uses of `y`, then the assignment `y = cos(z);` is dead code. Also, if the back end can determine that `x` is loop-invariant, then the second call to `cos` can be hoisted outside the loop.

We propose a `no_effects` annotation for function calls in the intermediate language. As in GCC, it is useful to distinguish between accessing global memory and performing other side effects like I/O or raising exceptions. The `no_effects` annotation would not guarantee anything about memory reads and stores. This is communicated to the back end with `mod/ref` lists instead. In the previous example, the statement `y = cos(z);` cannot be deleted if it can modify a global location that can be read later in the program. Calls to `cos` would be annotated with `no_effects` and empty `mod/ref` lists. (This is the same as a `const` function using GCC's annotations.) In this code:

```
for (...) {
  ... f(x) ...
  vec[i] = g(x) + ...
}
```

if `x` is loop-invariant, the call `g(x)` can be hoisted outside the loop if it is annotated `no_effects` *and* it has no read/write conflicts of global memory with `f`.

Notice that if a function may diverge for some of its inputs, deleting a call that is dead code, or moving it across basic blocks, may change the program result when compiled with optimizations turned on. This may not be allowed by the language definition. If this is the case, non-termination should be considered an effect and only calls that can be proven to terminate should be annotated with `no_effects`.

Effects annotations are also useful when the target is a stack-based language. Consider the following code:[3]

```
x = f(a);
y = g(b);
z = h(y,5,x);
```

where `f` has no side effects, `g` might have side effects, and `f` and `g` don't access the same global variables. In the absence of any annotations, the back end emits the following code:

```
push a
call f
push b
call g
swap
push 5
swap
call h
```

With appropriate `no_effects` and `mod/ref` annotations, the calls to `f` and `g` can be permuted to save the two `swap` instructions:

```
push b
call g
push 5
push a
call f
call h
```

3 Related Work

Annotations have been used successfully to assist back-end optimizations in several compilers. Cho et al. [CTS+98] communicate data dependence information of C programs to the GCC back end. The data is stored in separate files and is retrieved by the back end via a set of provided query functions. They report average speedups of 11% on a MIPS R10000. Several recent papers describe the use of Java class file attributes to store program information.

[3] This example was suggested by Andrew Kennedy.

[KC01] describes a framework to annotate JVM bytecode with information collected off-line. They convey counts of local variable usage, and control-flow information. The annotations are then used by dynamic compilers to guide optimization. They report reduced optimized compilation overhead by 78% and speedups of 7% on average for a set of Java programs. In [PQVR$^+$01], Pominville et al. encode optimization information in class file attributes for elimination of array bounds checks. They show speedups of up to 10% in the Kaffe virtual machine. The Related Work sections of these papers mention other systems that optimize programs using annotations generated by the compiler front end.

Some of these related projects exploit static program information for language-specific optimizations, like array bounds checking elimination in Java. In contrast, the focus of this paper is on conveying information that can be used for language-independent optimizations in the back end.

4 Future Work

It remains to evaluate the improvements that can be obtained with a back end that can take advantage of the annotations proposed in this paper. An empirical study should help measure the relative importance of each annotation in practice and thus serve as a guide to implementors of language-independent compiler back ends.

We do not address here the issue of verification when the intermediate representation is mobile and can be altered by malicious third parties. Tampering with branch prediction and prefetching annotations could result in slower execution time, but would not alter the original semantics of the program. The rest of the annotations proposed in this paper are sensitive and would need a safety mechanism if they are transmitted across insecure channels. Proof-carrying code [Nec97] could be a solution for these situations.

Acknowledgement

I wish to thank David Watt and Andrew Kennedy for their detailed comments and suggestions. Simon Peyton Jones and Norman Ramsey commented on an early draft of this paper. The anonymous reviewers made suggestions that improved the contents and the presentation of the paper.

References

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages

269–279, Montreal, Canada, 17–19 June 1998. *SIGPLAN Notices* 33(5), May 1998.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 159–170, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.

[BD88] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 329–338, Atlanta, Georgia, 22–24 June 1988. *SIGPLAN Notices* 23(7), July 1988.

[BK00] Nick Benton and Andrew Kennedy. Monads, effects and transformations. In Andrew Gordon and Andrew Pitts, editors, *Electronic Notes in Theoretical Computer Science*, volume 26. Elsevier Science Publishers, 2000.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 300–313, Albuquerque, New Mexico, 23–25 June 1993. *SIGPLAN Notices* 28(6), June 1993.

[CTS+98] S. Cho, J. Tsai, Y. Song, B. Zheng, S. Schwinn, X. Wang, Q. Zhao, Z. Li, D. Lilja, and P. Yew. High-level information : An approach for integrating front-end and back-end compilers. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, pages 346–355, Washington - Brussels - Tokyo, August 1998. IEEE USA.

[DKK+99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, (Q4), 1999.

[DMH92] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 273–282, San Francisco, California, 17–19 June 1992. *SIGPLAN Notices* 27(7), July 1992.

[GL00] Lal George and Allen Leung. MLRISC: A framework for retargetable and optimizing compiler back ends. Unpublished report available from `http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/html/index.html`, November 2000.

[Hai98] Max Hailperin. Cost-optimal code motion. *ACM Transactions on Programming Languages and Systems*, 20(6):1297–1322, November 1998.

[ISO99] ISO/IEC 9899:1999 standard for the C programming language (C99), December 1999. Available from `http://www.iso.ch`.

[JRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, pages 1–28. LNCS 1702, September 1999. Invited paper.

[KC01] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI)*, 20–22 June 2001.

[LP00] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000.

[Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 15–17, 1997.

[PQVR$^+$01] Patrice Pominville, Feng Qian, Raja Vallee-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *CC 2001*, volume 2027 of *LNCS*, pages 334–354, 2001.

[RPJ00] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)*, pages 285–298, Vancouver, British Columbia, 18–21 June 2000. *SIGPLAN Notices* 35(5), May 2000.

[Sta01] Richard M. Stallman. *Using and Porting the GNU Compiler Collection (GCC)*. Free Software Foundation, 2001. Available from `http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc.html`.

[Tar00] David Tarditi. Compact garbage collection tables. In Tony Hosking, editor, *Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press. ISMM is the successor to the IWMM series of workshops.

[YR01] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. Invited paper.

[YS98] Cliff Young and Michael D. Smith. Better global scheduling using path profiles. In *Proceedings of the 31st Annual International Symposium* *on Microarchitecture*, pages 115–123, Dallas, Texas, November 30–December 2, 1998. IEEE Computer Society TC-MICRO and ACM SIGMICRO.