Compiling Mercury to high-level C code

Fergus Henderson and Zoltan Somogyi

{fjh,zs}@cs.mu.oz.au
Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

Abstract. Many logic programming implementations compile to C, but they compile to very low-level C, and thus discard many of the advantages of compiling to a high-level language. We describe an alternative approach to compiling logic programs to C, based on continuation passing, that we have used in a new back-end for the Mercury compiler. The new approach compiles to much higher-level C code, which means the compiler back-end and run-time system can be considerably simpler.
We present a formal schema for the transformation, and give benchmark results which show that this approach delivers performance that is more than competitive with the fastest previous implementation, with greater simplicity and better portability and interoperability.
The approach we describe can also be used for compiling to other target languages, such as IL (the Microsoft .NET intermediate language).

Keywords: compilation techniques, programming language implementation, logic programming, Mercury, C, GNU C.

1 Introduction

Nowadays many implementations of high-level languages compile to C [6, 9, 15]. We have used the technique ourselves in the original implementation [7, 10, 14] of Mercury, a strongly-typed declarative programming language which supports functional and logic programming.

The popularity of compilation to C is not surprising, because its benefits are by now well known:

- C code can be portable.
  C compilers exist for almost every important hardware architecture. In comparison to generating assembler or writing a JIT compiler, generating portable C code greatly reduces the effort required to port the high-level language implementation to a different hardware architecture.
- C is efficient.
  High quality C compilers are widely and often freely available. Generating native code via C can result in considerably better performance than writing an interpreter, and the performance will usually be close to what could be obtained by generating assembler directly, especially if there is a close match between the source language and C. Indeed the performance may well be better in practice, since more resources are available for improving the C compiler's optimizer than would be available for a less established language.

- C has critical mass.
  There is good tool support, lots of programmers know C, there is lots of existing C code to link to, and so on.
- C is higher level than assembler.
  This can make compiling to C much easier. Again, the benefit is greatest if there is a close match between the source language and C.

Unfortunately, however, logic programming (LP) languages are *not* a good match with C. There are two key problems: tail recursion and backtracking.

Tail recursion. In logic programs, recursion is the primary method of iteration. To ensure that recursive loops can operate in constant space, logic programming implementations perform tail call optimization, not just for directly recursive tail calls, but also for tail calls to other procedures (which might be part of an indirectly recursive loop).

However, C programs generally use explicit looping constructs for iteration, and C implementations generally don't optimize tail calls. Even those implementations which do make some attempt at this generally only do so in a very limited set of circumstances. The problem is that the semantics of C make it very difficult for the C compiler to perform tail call optimization if any local variable has its address taken. Furthermore for most C programs, the payoff of optimizing the difficult cases is likely to be very small.

When compiling to C, an LP language compiler can recognize directly tail-recursive loops and output C looping constructs for them. But if procedure calls in the LP language are to be mapped to function calls in C, then tail calls other than directly recursive tail calls can't be optimized so easily. Inlining can reduce some indirectly recursive loops to directly recursive loops, but it won't handle the general case; indirect tail recursion can even span module boundaries.

Backtracking. The presence of nondeterminism and backtracking in LP languages leads to a completely different model of procedure calling.

In C and other traditional languages, each procedure is called and then, after some processing, the procedure will normally return to the caller. The stack frame can be allocated on calls and deallocated on returns.

In contrast, Prolog and other languages that support backtracking use a four-port model (CALL, EXIT, REDO, FAIL) [4]. A procedure is called (CALL), does some processing, and then returns an answer to the caller (EXIT); but after an answer has been returned, the caller can re-enter the procedure to look for more solutions (REDO). If there are more solutions, the procedure will return another answer again (EXIT), and the caller may again ask for more solutions (REDO). Eventually, when there are no more solutions, the procedure will FAIL; only then can the stack frame be deallocated.

The traditional solution. Because of the difference in procedure calling model imposed by backtracking, LP predicate calls and exits cannot be mapped directly to C function calls and returns. Instead, LP language compilers that target C use their own data areas for parameter passing and storing local variables. These data areas are then manipulated explicitly in the generated C code.

This solution also helps solve the problem with tail calls, because the LP language compiler has complete control over the LP data areas. The C stack can be kept to a fixed size using a driver loop, e.g.

```
typedef void * Func(void);
void driver(Func *entry) {
    register Func *fp = entry;
    while (fp != NULL) {
        fp = (Func *) (*fp)();
    }
}
```

with each C function returning the address of the next C function to call. Various optimizations on this basic model (such as loop unrolling, using GNU C extensions, inline assembler jumps, etc.) are possible, and were exploited by earlier versions of the Mercury compiler [10].

Drawbacks of the traditional solution. The traditional approach means that the LP language implementation will effectively define its own virtual machine, with the virtual machine instructions typically implemented as C macros.

This approach is workable, but because it doesn't use the C calling convention, it unfortunately discards many of the advantages of compiling to a high-level language:

- The LP language compiler needs to do much of the work of a traditional compiler, including allocating variables to virtual machine stack slots or virtual registers.
- The generated code is low level, and hard to read.
- The performance is not as good as it could be, because the LP compiler often ends up working against the C compiler, rather than with it. For example, if GNU C extensions are used to map virtual machine registers into real registers, then those registers can't be used by the C compiler for other purposes. Another example is that because all data manipulation is done via the LP implementation's data structures, rather than local variables, the C compiler's ability to analyze possible aliasing may be significantly inhibited, which can harm the C compiler's ability to optimize the code.
- There is a forced trade-off between efficiency, simplicity, and portability; the optimizations mentioned above, which are needed to achieve good efficiency, compromise portability and/or increase complexity of the source code.

In this paper, we describe an alternative approach to transforming logic programs to C, using continuation passing to handle nondeterminism, that avoids these drawbacks.

2 Preliminaries

Abstract Syntax The transformation described in this paper takes as its input logic programs which have been reduced to a simplified intermediate form: unifications are flattened, and each predicate has only one clause (multiple clauses having been converted into explicit disjunctions). The abstract syntax for goals in our simplified core language is as shown in Figure 1.

```
goal → (goal , goal)                 Conjunction
     ; (goal ; goal)                 Disjunction
     ; not(goal)                     Negation
     ; (goal -> goal ; goal)         If-then-else
     ; once (goal)                   Pruning
     ; pred(var, ...)                Calls
     ; var = var                     Unification (assignment/test)
     ; var = functor(var, ...).      Unification (construction/deconstruction)
```

Fig. 1. Mercury core abstract syntax.

Procedures A key aspect of our translation is that it is mostly a one-to-one mapping. Each procedure in the logic program is mapped to one C function (possibly containing nested functions, as explained below). Each head variable in a clause is mapped to a corresponding C function parameter, and each non-head variable is mapped to a corresponding C local variable.

Types and Modes Since Mercury is a (mostly) statically typed and moded language — i.e. the programmer declares and/or the compiler infers the type of each variable, and whether each parameter is input or output — we have full type and mode information available when generating code. So each Mercury type is mapped to the corresponding C type, with input parameters passed by value, and output parameters passed by reference (i.e. using pointer arguments in the generated C code).

However, the transformation scheme described here does not require static type or mode information; for a dynamically typed and dynamically moded LP language, it would be possible to map every type in the source language to a single C type, and to always pass arguments by reference, though this would of course add the usual run-time overheads for dynamic typing and dynamic modes.

Procedures with Multiple Modes An important feature of logic programming is that it supports multi-moded predicates. For example, the same predicate `append/3`, can be used either to append two lists together (the '`(in, in, out)`' mode), or to find all the ways of splitting a single list into two sublists (the '`(out, out, in)`' mode).

For multi-moded Mercury predicates, each mode of the predicate is treated as a different procedure, and so we generate a different C function for each mode of the predicate.

Determinism Analysis The transformation requires that each (sub-)goal in the abstract syntax be annotated with its determinism, which indicates how many times that goal can succeed (EXIT) each time it is invoked (CALLed).

For Mercury, this information is readily available, since the Mercury language includes determinism declarations (optional for procedures local to a module, but mandatory for procedures exported for use by other modules). The compiler's determinism checking and inference [11] produces the information that we need.

We use a simplified form of the determinism categories used by the Mercury language, which takes into account only those distinctions which are important for code generation:

- **m_det** indicates that the goal will succeed exactly once (unless it does not terminate, or throws an exception)
- **m_semi** indicates that the goal will succeed at most once
- **m_non** indicates that the goal may succeed any number of times

For example, '`(in, in, out)`' mode of `append/3` has determinism m_det, while the '`(out, out, in)`' mode has determinism m_non.

Some other LP languages, such as Turbo/PDC/Visual Prolog, also have similar compile-time determinism checking/inference. For other LP languages the determinism information could be obtained by static analysis of the program. This kind of analysis has been done by optimizing Prolog compilers such as Parma [16], Aquarius Prolog [17] and Ciao-Prolog [3].

The transformation scheme relies fairly heavily on having determinism information about every goal (and accurate determinism analysis in turn also requires accurate type and mode information). If determinism information isn't available, it would be possible to use a conservative approximation — in the worst case assigning the determinism m_non to every procedure — but this will significantly reduce the efficiency of the generated code.

3 Our Transformation Scheme

Continuation Passing Style. For nondeterministic procedures, we generate code using an explicit continuation passing style. Each nondeterministic procedures gets translated into a function which takes an extra parameter which is a function pointer that points to the success continuation. On success, the function calls its success continuation, and on failure it returns.

To keep things easy, our transformation generates code which may contain nested functions (as in Pascal, or GNU C). Our use of nested functions is restricted to what are often known as "downward closures": when we take the address of a nested function, we only ever do two things with it: pass it as a continuation argument, or call it. The continuations are never returned and never stored inside heap objects or global variables. These conditions are sufficient to ensure that we never keep the address of a nested function after the containing function has returned, so we won't get any dangling continuations.

If the target language doesn't support nested functions (or, like GNU C, doesn't support them *efficiently* enough) then after the transformation is complete, we have a separate pass that transforms the generated C code into a form that does not use nested functions, by explicitly passing a pointer to an environment struct to each function that was originally nested. Due to space limitations, we do not describe how we do this conversion. Techniques for implementing nested functions are well described in the literature (e.g. [2]).

Calling Convention. In each procedure, we declare a local variable '`bool succeeded`'. This is used to hold the success status of m_semi sub-goals. The transformation schemas below show local declarations for the '`succeeded`' variable in all the places where they would be needed if we were generating them locally. However, in our current implementation we actually just generate a single '`succeeded`' variable for each procedure. This is simpler, but may not be quite as efficient.

The calling convention for sub-goals is as follows.

- m_det goal: On success, fall through. (May overwrite '`succeeded`'.)
- m_semi goal: On success, set '`succeeded`' to TRUE and fall through. On failure, set '`succeeded`' to FALSE and fall through.
- m_non goal: On success, call the current success continuation. On failure, fall through. (May overwrite '`succeeded`' in either case.)

Notation. We use the following notation to distinguish between calls in the different code models:

Code model	**Notation**	**Definition**
m_det	$[\![$ do *Goal* $]\!]$	Execute *Goal* (which must be m_det).
m_semi	$[\![$ succeeded = *Goal* $]\!]$	Execute *Goal*, and set '`succeeded`' to TRUE if the goal succeeds and FALSE if it fails.
m_non	$[\![$ *Goal* && *Cont*() $]\!]$	Execute *Goal*, calling the success continuation function *Cont*() every time it succeeds, and falling through when it fails.

We also use the following notation for the transformation rules used by our translator:

situation:	$[\![$ *construct* $]\!]$	$\Longrightarrow$	*code*

This means that in the situation described by *situation*, the specified *construct* should be translated by the LP language compiler into the specified *code*. The *code* will in general be a mixture of C code and fragments inside $[\![\ldots]\!]$ which need to be further translated.

3.1 Converting Between Different Code Models

If a m_*foo* goal occurs in a m_*bar* context, where $foo \neq bar$, then we need to modify the code that we emit for the goal so that it conforms to the calling convention expected for m_*bar*.

Normally determinism analysis will ensure that the determinism expected by the context is more permissive than the determinism of the goal. (There is one exception, "commits"; they are dealt with below.) So we only have the following cases to deal with:

m_det *Goal* in m_semi context:

```
⟦ succeeded = Goal ⟧
⟹
⟦ do Goal ⟧
succeeded = TRUE;
```

m_det *Goal* in m_non context:

```
⟦ Goal && Cont() ⟧
⟹
⟦ do Goal ⟧
Cont();
```

m_semi *Goal* in m_non context:

```
⟦ Goal && Cont() ⟧
⟹
bool succeeded;

⟦ succeeded = Goal ⟧
if (succeeded) Cont();
```

3.2 Code for Conjunctions

Code for empty conjunctions ('`true`') is trivial, and if the first goal is m_det, it is also straight-forward:

m_det goal:

```
⟦ do true ⟧
⟹
/* fall through */
```

m_semi goal:

```
⟦ succeeded = true ⟧
⟹
succeeded = TRUE;
```

m_non goal:

```
⟦ true && Cont() ⟧
⟹
Cont();
```

m_det *Goal*:

```
⟦ (Goal, Goals) ⟧
⟹
⟦ do Goal ⟧
⟦ Goals ⟧
```

If the first goal is m_semi, then there are two cases: if the conjunction as a whole is m_semi, things are simple, and if the conjunction as a whole is m_non, then we do the same as for the m_semi case, except that we also (ought to) declare a local '`succeeded`' variable.

m_semi *Goal* **in m_semi conjunction:**

$[\![$ `succeeded =` `(`*Goal*`,` *Goals*`)` $]\!]$

$\Longrightarrow$

$[\![$ `succeeded =` *Goal* $]\!]$

```
if (succeeded) {
    [[ Goals ]]
}
```

m_semi *Goal* **in m_non conjunction:**

$[\![$ *Goal* `&&` *Goals* $]\!]$

$\Longrightarrow$

```
bool succeeded;

[[ succeeded = Goal ]]
if (succeeded) {
    [[ Goals ]]
}
```

The really interesting case comes when the first goal is m_non. In that case, we need to create a new local continuation function `succ_func`$_n$`()` which we use as the continuation when generating code for the first goal. The continuation function just evaluates the remaining goal(s), with the original continuation function.

m_non *Goal***:**

$[\![$ `(`*Goal*`,` *Goals*`) &&` *Cont*`()` $]\!]$

$\Longrightarrow$

```
succ_func_n() {
    [[ Goals && Cont() ]]
}

[[ Goal && succ_func_n() ]]
```

3.3 Code for Disjunctions

Code for empty disjunctions (‘`fail`’) is trivial:

m_semi goal:

$[\![$ `succeeded = fail` $]\!]$

$\Longrightarrow$

```
succeeded = FALSE;
```

m_non goal:

$[\![$ `fail &&` *Cont*`()` $]\!]$

$\Longrightarrow$

```
/* fall through */
```

Code for non-empty disjunctions differs depending on the code model of the disjunction, and on the determinism of the goal that is the first disjunct.

(a) m_det disjunction:

m_det *Goal***:**

$[\![$ `do (`*Goal* `;` *Goals*`)` $]\!]$

$\Longrightarrow$

```
[[ do Goal ]]
/* Goals is unreachable */
```

m_semi *Goal***:**

$[\![$ `do (`*Goal* `;` *Goals*`)` $]\!]$

$\Longrightarrow$

```
bool succeeded;

[[ succeeded = Goal ]]
if (!succeeded) {
    [[ do Goals ]]
}
```

(b) m_semi disjunction:

m_det *Goal*:
```
⟦ succeeded =
    (Goal ; Goals) ⟧
⟹
bool succeeded;

⟦ do Goal ⟧
succeeded = TRUE
/* Goals is unreachable */
```

m_semi *Goal*:
```
⟦ succeeded =
    (Goal ; Goals) ⟧
⟹
bool succeeded;

⟦ succeeded = Goal ⟧
if (!succeeded) {
    ⟦ succeeded = Goals ⟧
}
```

(c) m_non disjunction:

m_det *Goal*:
```
⟦ (Goal ; Goals)
    && Cont() ⟧
⟹
⟦ Goal ⟧
Cont();
⟦ Goals && Cont() ⟧
```

m_semi *Goal*:
```
⟦ (Goal ; Goals)
    && Cont() ⟧
⟹
bool succeeded;

⟦ succeeded = Goal ⟧
if (succeeded) Cont();
⟦ Goals && Cont() ⟧
```

m_non *Goal*:
```
⟦ (Goal ; Goals)
    && Cont() ⟧
⟹
⟦ Goal && Cont() ⟧
⟦ Goals && Cont() ⟧
```

3.4 Code for if-then-else

m_det *Cond*:
```
⟦ (Cond -> Then ; Else) ⟧
⟹
⟦ Cond ⟧
⟦ Then ⟧
```

m_semi *Cond*:
```
⟦ (Cond -> Then ; Else) ⟧
⟹
bool succeeded;

⟦ succeeded = Cond ⟧
if (succeeded) {
    ⟦ Then ⟧
} else {
    ⟦ Else ⟧
}
```

m_non *Cond*:
```
⟦ (Cond -> Then ; Else) ⟧
⟹
bool cond_n;

void then_func() {
    cond_n = TRUE;
    ⟦ Then ⟧
}

cond_n = FALSE;
⟦ Cond && then_func() ⟧
if (!cond_n) {
    ⟦ Else ⟧
}
```

If-then-elses with m_det and m_semi conditions translate easily into C, as shown in the left-hand column above.

Mercury also allows if-then-elses with m_non conditions, in which case execution can backtrack from the *Then* part back into the *Cond* part. (This is unlike Prolog's standard if-then-else, which always prunes over the *Cond*, but like e.g. SICStus Prolog's `if/3`.) Handling these is a little more tricky. We need to ensure that execution won't backtrack into the *Else* if the *Cond* succeeds, even if it is later backtracked over. To do this, we introduce a fresh boolean variable (which we call $cond_n$) to record whether or not *Cond* has ever succeeded, as shown in the right-hand column above.

We also use the translation rules for if-then-else to handle negations, because we handle $[\![$ `not(`*Goal*`)` $]\!]$ as if it were $[\![$ `(`*Goal* `-> fail ; true)` $]\!]$.

Note that there are some complications with if-then-else and liveness-accurate garbage collection, but due to lack of space we cannot elaborate on these.

3.5 Code for Commits

Most LP languages provide some way to execute a nondeterministic goal, find the first solution, and prune away all the other solutions to that goal. For example, Prolog has '`!`' ("cut") and `once/1`, while Mercury has committed choice nondeterminism and automatic pruning of nondeterministic goals with no output variables.

With our continuation-based approach for handling nondeterminism, implementing commits requires some way of unwinding the stack. Depending on the exact target language (which may be e.g. C, GNU C, C++, etc.) there are several different ways in which this can be done:

- using `setjmp()` / `longjmp()`
- using GNU C's `__builtin_setjmp()` / `__builtin_longjmp()`
- exiting nested functions via GNU C non-local gotos that jump to their containing functions
- using catch/throw
- by testing a flag after each call

The first four alternatives, which are all preferable to the last one, are quite similar. In our implementation, we wanted to support multiple different target languages. So we transform the code to an intermediate representation which abstracts away the differences between the first four approaches using '`TRY_COMMIT`' and '`DO_COMMIT`' operations.

In the Mercury compiler, places where pruning is required show up after determinism analysis as calls to m_non goals in m_det or m_semi contexts; these are equivalent to calls to '`once/1`' in Prolog. Mercury has no direct equivalent to Prolog's cut, so we don't give a transformation schema for handling cut, but the `TRY_COMMIT`/`DO_COMMIT` operations shown below would also be quite suitable for implementing Prolog's cut (including `!/1` as in e.g. SWI-Prolog, as well as the standard `!/0`).

The Abstract Transformation The transformation rules below are the abstract version, using `TRY_COMMIT`/`DO_COMMIT`.

m_non in m_semi context:
$[\![$ `succeeded = once(`$Goal$`)` $]\!]$
$\Longrightarrow$
```
COMMIT_TYPE ref;
void success() {
   DO_COMMIT(ref);
}
TRY_COMMIT(ref, {
   [[ Goal && success() ]]
   succeeded = FALSE;
}, {
   succeeded = TRUE;
})
```

m_non in m_det context:
$[\![$ `do once(`$Goal$`)` $]\!]$
$\Longrightarrow$
```
COMMIT_TYPE ref;
void success() {
   DO_COMMIT(ref);
}
TRY_COMMIT(ref, {
   [[ Goal && success() ]]
}, {})
```

setjmp/longjmp When using `setjmp()/longjmp()`, the abstract operations mentioned above are defined as follows: '`COMMIT_TYPE`' is '`jmp_buf`', '`DO_COMMIT(ref)`' is '`longjmp(ref,1)`', and '`TRY_COMMIT(ref,s1,s2)`' is '`if (setjmp(ref)) s2 else s1`'.

Care is required when using `longjmp()/setjmp()`, because the ANSI/ISO C standard says that `longjmp()` is allowed to destroy the values of any non-volatile local variables in the function that called `setjmp()` which have been modified between the `setjmp()` and the `longjmp()`.

To avoid this, whenever we generate a commit, we put it in its own nested function, with the local variables (e.g. `succeeded`, plus any outputs from the goal that we are committing over) remaining in the containing function. This ensures that none of the variables which get modified between the `setjmp()` and the `longjmp()` and which get referenced after the `longjmp()` are local variables in the function containing the `setjmp()`.

Due to lack of space, we omit discussion of the other alternatives.

3.6 Calls

Generating code for individual calls is straight-forward. Predicate calls are mapped directly to C function calls:

m_det call:
$[\![$ `do` p`(`A_1`, `A_2`, ...)` $]\!]$
$\Longrightarrow$
p`(`A_1`, `A_1`, ...);`

m_semi call:
$[\![$ `succeeded =`
   p`(`A_1`, `A_2`, ...)` $]\!]$
$\Longrightarrow$
`succeeded = `p`(`A_1`, `A_2`, ...);`

m_non call:
$[\![$ p`(`A_1`, `A_2`, ...)`
   `&&` $Cont$`()` $]\!]$
$\Longrightarrow$
p`(`A_1`, `A_2`, ..., `$Cont$`);`

The only significant complication is that output arguments should be passed by reference. The details for handling this are straight-forward but tedious, so we omit a detailed description, and instead refer interested readers to the Mercury compiler sources.

3.7 Unifications

The code generated for unifications is straight-forward, but will depend on the exact data representation chosen. For m_semi unifications, we need to set `succeeded` to indicate whether the unification succeeded or not. The other details are much the same as in traditional approaches to compiling logic programs to C, so we only give a sample:

m_det deconstruct:

```
⟦ succeeded =
    (X = f(A1, A2, ...)) ⟧
⟹
/* extract arguments */
A1 = arg(X, f, 1);
A2 = arg(X, f, 2);
...
```

m_semi deconstruct:

```
⟦ X = f(A1, A2, ...) ⟧
⟹
/* tag test */
⟦ succeeded =
    (X = f(_, _, ...)) ⟧
if (succeeded) {
    /* extract arguments */
    A1 = arg(X, f, 1);
    A2 = arg(X, f, 2);
    ...
}
```

Here `arg()` could be defined as a C macro or function in the runtime library.

4 Implementation and Benchmarks

We have implemented this approach in a new back-end for the Mercury compiler, which is included, together with the original back-end, in Mercury 0.10 and 0.10.1 (released April 2001). The choice of back-end is controlled by a compiler option.

We have tested the correctness of our implementation by successfully bootstrapping the compiler using the new back-end, and by passing, on several architectures, all the appropriate tests (several hundred) in the Mercury test suite.

Program	SLOC	time (in seconds)			size (in kb)		
		asm_fast.gc	hlc.gc	ratio	asm_fast.gc	hlc.gc	ratio
mmc	171474	17.20	20.36	1.18	6320	4504	0.71
compress	385	26.73	18.98	0.71	1328	944	0.71
icfp2000	4341	65.36	30.74	0.47	1848	1148	0.62
icfp2001	458	33.02	32.17	0.97	1344	952	0.71
nuc	3120	40.53	31.39	0.77	1392	1040	0.75

Fig. 2. Benchmark speed ratios

We have evaluated our new compilation scheme, which we refer to as high level C (`hlc.gc`), by comparing it to our old scheme [14], which was the fastest existing Mercury implementation. The old scheme compiles Mercury to C code that is so low level that it uses C only as a portable assembler [10]; the fastest version of this scheme, `asm_fast.gc`, uses small pieces of assembly code as well as GNU C extensions. The two schemes use identical data representations and the same garbage collector [1]. The results are shown in Figure 2. The `SLOC` column shows the number of Source Lines of Code for each benchmark, excluding comments and blank lines. The following two groups of columns compare `asm_fast.gc` and `hlc.gc` with respect to CPU times and executable sizes.

The mmc test case is the Mercury compiler translating a large source file. Compress is a Mercury version of the 129.compress benchmark from the SPECint95 suite. The next two entries involve our group's entries in recent ICFP programming contests. The 2000 entry is a ray tracer that generates .ppm files from a structural description of a scene, while the 2001 entry is a source-to-source compression program for a hypothetical markup language. Nuc is a Mercury version of the pseudoknot benchmark, executed 1000 times. The benchmark machine was a Gateway 5150XL laptop (700 MHz PIII, 256 Mb, Linux 2.2.18). Further details of the test setup are available from our web site.

On the two floating-point intensive programs (icfp2000 and nuc), the high level C back end already outperforms the old back end, principally because it boxes floating point values only when they are stored on the heap, not when they are stored on the stack. Storing unboxed floating point values on the stack comes naturally when the C compiler is managing the stack frames, but doing the same in our low-level back-end would be difficult, because it would significantly complicate our stack slot allocation algorithm. The high level C back end also outperforms the old back end on compress, mainly because compress's work is dominated by complex integer expressions, and the C compiler can store the intermediate results in machine registers whereas the Mercury compiler must usually put them in virtual registers that are actually stored in memory. In both cases, the new back end wins because it does better at reusing the development effort already invested in existing C compilers.

For the remaining two programs, the picture is mixed. On mmc, the high level C back end is slower than the old back end; on icfp2000, it is about the same speed. One advantage of the old back end is that it has a more streamlined calling convention. However, the old back end relies on exploiting GNU extensions to C for its efficiency. Projects that need to use a C compiler other than gcc (e.g. Microsoft Visual C) cannot use these extensions. Without those extensions, the low level back end loses much of its speed; e.g. the time for mmc increases from 17.20s to 27.42s. Since the high level C back end does not need to use gcc extensions for its speed, it consistently outperforms the old back end on such projects. This last point makes our scheme especially useful for commercial users, who often need to link Mercury programs with software such as Microsoft Foundation Classes, and therefore need to use Microsoft compilers.

5 Related Work

The idea of implementing nondeterminism by invoking a continuation on success and falling through on failure is not new. It has been proposed several times in the literature, in several contexts — for example Prolog meta-interpreters and implementing nondeterminism in languages such as Lisp [5] — and it is closely related to the idea of binarization of Prolog programs [15] and to the implementation of generators in languages such as Icon [13]. However, few of these papers have formal translation rules, and few consider the optimization opportunities presented by knowledge of determinism information, which can derived by program analysis even for languages such as Prolog in which determinism is not a fundamental concept. Few have practical, well-tested implementations. None have translation rules *and* practical implementations.

The only papers that we know of that use a translation scheme that is reasonably closely related to the one presented in this paper are [12] and [18], which describe schemes for translating strongly typed variants of Prolog to Pascal and C respectively. Both transformations have significant limitations. They do not handle if-then-else or the Prolog cut operator, nor do they handle nested disjunctions. While they both have examples showing how one can exploit determinism information to generate better code, they do not describe, even informally, the rules that govern the generation of that better code.

6 Conclusions

We have presented a scheme for translating Mercury to high level C code. The new compilation scheme has already shown itself to be competitive with our previous scheme for compilation to low level C, beating its performance for programs that are floating point intensive and in environments where one cannot use gcc as the C compiler and therefore cannot use GNU C extensions to the C language.

Furthermore, this performance is achieved with a model that is in our opinion significantly simpler than earlier approaches such as the original Mercury compiler or WAM-based Prolog to C compilers. We have no need for additional global data structures, such as virtual machine registers, environment or choice point stacks, and the like, and we avoid the need to do our own register or stack slot allocation. Since the C code that we generate is closer to what an ordinary C programmer would write, the C compiler can be expected to optimize it better, and it is less likely to trigger obscure bugs in the C compiler.

Our translation scheme can also be adapted to target languages other than C. We have used it as the basis of the Mercury code generator that targets IL, the intermediate language of the .NET Common Language Runtime [8], and as the basis of an (as yet incomplete) code generator that emits Java. Most of the difficult issues in those ports concern issues such as data representation that are orthogonal to the topic of this paper; the adaptation of the translation scheme has been relatively straightforward.

The main drawback of our translation scheme is that we cannot guarantee tail call optimization for indirectly recursive tail calls, unless there is explicit support for this in the target language (as is the case for IL and C--).

Acknowledgements. We would like to thank David Overton, Ralph Becket, Bernard Pope, Kevin Glynn, and the anonymous referees for reviewing earlier drafts of this paper, and Microsoft for their financial support.

References

1. H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.
2. T. M. Breuel. Lexical closures for C++. In *Proceedings of the 1988 USENIX C++ Conference*, pages 293–304, Denver, Colorado, 1988.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog system. reference manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. Available from http://www.clip.dia.fi.upm.es/.
4. L. Byrd. Understanding the control of Prolog programs. Technical Report 151, University of Edinburgh, 1980.
5. M. Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.
6. P. Codognet and D. Diaz. wamcc: Compiling Prolog to C. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 317–331, Kanagawa, Japan, June 1995.
7. T. Conway, F. Henderson, and Z. Somogyi. Code generation for Mercury. In *Proceedings of the Twelfth International Conference on Logic Programming*, pages 242–256, Portland, Oregon, December 1995.
8. T. Dowd, F. Henderson, and P. Ross. Compiling Mercury to the .NET Common Language Runtime. In *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability*, pages 70–85, Firenze, Italy, September 2001.
9. B. Hausman. Turbo Erlang: approaching the speed of C. In E. Tick, editor, *Implementations of logic programming systems*, pages 119–135. Kluwer, 1994.
10. F. Henderson, Z. Somogyi, and T. Conway. Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*, Portland, Oregon, December 1995.
11. F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.
12. J. F. Nilsson. On the compilation of a domain-based Prolog. In *Proceedings of the Ninth IFIP Congress*, pages 293–298, Paris, France, 1983.
13. J. O'Bagy and R. E. Griswold. A recursive interpreter for the Icon programming language. In *Proceedings of the 1987 SIGPLAN Symposium on Interpreters and Interpretive Techniques*, pages 138–149, St. Paul, Minnesota, 1987.
14. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–December 1996.
15. P. Tarau, K. D. Bosschere, and B. Demoen. Partial translation: towards a portable and efficient Prolog implementation technology. *Journal of Logic Programming*, 29(1–3):65–83, October–December 1996.
16. A. Taylor. LIPS on a MIPS: results from a Prolog compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185, Jerusalem, Israel, June 1990.
17. P. Van Roy and A. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.
18. J. Weiner and S. Ramakrishnan. A piggy-back compiler for Prolog. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 288–296, Atlanta, Georgia, June 1988.