Optimizing P#: Translating Prolog to more Idiomatic C#

Jonathan J. Cook
Laboratory for Foundations of Computer Science,
University of Edinburgh, Edinburgh, EH9 3JZ, U.K.
Jon.Cook@ed.ac.uk

Abstract

P# is our Prolog implementation which operates by translating Prolog to C# source code. The C# produced by the latest version (1.1.3) of P# was very unlike the code which a human programmer might produce. In this paper we show how, aided by mode and type annotations, more idiomatic C# can be generated by translating tail-recursion into `while` loops and applying liveness analysis to remove unnecessary variables. We apply this optimization to semi-deterministic and deterministic predicates which do not have non-deterministic predicates beneath them in the call tree, since such predicates enjoy relatively simple control flow. In this paper we demonstrate the benefit of generating C# code that is closer to that which a human programmer might produce and which can therefore be compiled efficiently and well by a C# compiler. This improvement in the high-level code generated by P# significantly speeds up the execution of a range of benchmarks that we have compiled. A secondary benefit of this approach is that the generated code is easier for a human programmer to read or to modify without inadvertently breaking the code.

1 Introduction

Many Prolog [4] implementations compile Prolog to a high-level language. The reference system for translation to C is wamcc [5]. Prolog implementations based on Java have been developed, for example Prolog Café [18] and Jinni [11].

Microsoft's .NET framework [16] allows multiple languages to interoperate by compiling them to the .NET intermediate language, MSIL. The principal .NET language is C# [14]. C# is similar to Java, although it has some C++ language features not found in Java, and other novel language features found in neither language. These include management capabilities for objects, allocation controls and strong name and versioning support.

P# [17] is our Prolog implementation, which translates Prolog to C# source code. P# is a modified port of Prolog Café from Java to C#. By translating to C# rather than to MSIL, we can obtain a closer integration of Prolog code with C# code. Also, much optimization is done for us by the C# compiler, so our compiler is less complicated than a Prolog to MSIL compiler would have to be.

P# has support for concurrency based on the ability to fork threads and to then wait for shared variables to be instantiated on those forked threads. P# also inherits linear logic features from Prolog Café. Discussion of the development of P# and the addition of concurrency features to P# can be found in [6]. The version of P# described in [6] can be downloaded from the P# homepage [17].

Much effort has been invested in the optimization of Prolog implementations. A particularly successful technique which underlies almost all Prolog implementations is the Warren Abstract Machine (WAM) [1]. Indexing [8] is an example of an optimization of the WAM.

In this paper, we discuss a major optimization of P#. Our optimization is based on the exploitation of semi-deterministic predicates. A predicate is *semi-deterministic* if it always either fails or succeeds with exactly one solution. If a predicate is semi-deterministic then there may be backtracking from one clause of the predicate to the next, if an earlier clause fails at some point. A semi-deterministic predicate which only calls other semi-deterministic predicates has the property that an individual clause will not be executed more than once by backtracking. In such cases we can do away with the Prolog stacks, which govern backtracking, and simulate in C# the fairly simple flow of control which is permitted for such a predicate. A predicate is *non-deterministic* if it may produce more than one solution.

A more specific class of predicates than the semi-deterministic predicates is that of the *deterministic* predicates. A predicate is deterministic if it always succeeds exactly once. Deterministic predicates occur frequently in idiomatic Prolog. Often, they are the result of coding a function in Prolog. When one wishes to code a predicate which will be used as a function, one generally expresses this as a Prolog predicate, some of whose arguments are *input* arguments, with the other arguments being *output* arguments. Input arguments are arguments which are known to be instantiated on entry into the predicate, and output arguments are those which are not instantiated on entry into the predicate, but which will be instantiated on exit from the predicate. The property of an argument of being input or output is referred to as its *mode*.

Developer experience suggests that these functional predicates often perform some simple utility, that is, they frequently occur as leaf predicates. Such a predicate might, for example, concatenate two lists. Another feature of leaf predicates is that, often, they are called frequently. Thus, if we can reduce the time taken to execute such predicates, we may effect a significant optimization.

In section 2, we introduce the compilation scheme which was originally used for all P# predicates. In section 3, we describe and explain a modified compilation scheme which is used for certain predicates. Section 3 ends with a discussion of how we now compile the Eight Queens Problem. In section 4, we present and discuss performance data which demonstrates the improvement in efficiency which can be obtained by translating to more idiomatic code. In the remaining sections we discuss future and related work and give conclusions.

2 P#'s Original Compilation Scheme

P# originally compiled all predicates using a continuation passing scheme which was inherited from Prolog Café [3]. This scheme was in turn inherited from jProlog [12], and is referred to as *binarization* [20]. A supervisor function calls each predicate to be executed, and each predicate called returns the next predicate to be called. A conjunction of goals in the body of a predicate is coded by returning a chain of predicate calls. Disjunction is handled by the run-time system. As an example, consider the Prolog predicate:

```
a(X) :- b(X), c(2).
a(X).
```

This is compiled to the following C# code:

```
namespace JJC.Psharp.Predicates {
using ...
```

```
public class A_1 : Predicate {
    static internal readonly Predicate A_1_1 = new Predicates.A_1_1();
    static internal readonly Predicate A_1_2 = new Predicates.A_1_2();
    static internal readonly Predicate A_1_sub_1 = new Predicates.A_1_sub_1();
    public Term arg1;
    ...
    public override Predicate exec( Prolog engine ) {
        engine.aregs[1] = arg1;
        engine.cont = cont;
        return call( engine );
    }
    public virtual Predicate call( Prolog engine ) {
        engine.setB0();
        return engine.jtry(A_1_1, A_1_sub_1);
    }
    ...
}

sealed class A_1_sub_1 : A_1 {
    public override Predicate exec( Prolog engine ) {
        return engine.trust(A_1_2);
    }
}

sealed class A_1_1 : A_1 {
    static internal readonly IntegerTerm s1 = new IntegerTerm(2);
    public override Predicate exec( Prolog engine ) {
        Term a1;
        Predicate p1;
        a1 = engine.aregs[1].Dereference();
        Predicate cont = engine.cont;
        p1 = new Predicates.C_1(s1, cont);
        return new Predicates.B_1(a1, p1);
    }
}

sealed class A_1_2 : A_1 {
    public override Predicate exec( Prolog engine ) {
        return engine.cont;
    }
}
}
```

This code is very similar in structure to that produced by Prolog Café, except that it is C#, not Java, and uses namespaces. The P# runtime system contains a class `Term`, which has subclasses: `IntegerTerm`, `DoubleTerm` (for floats), `ListTerm`, `StructureTerm`, `SymbolTerm` (for atoms), `VariableTerm` and `CsObjectTerm` (for interoperation with C#) which represent different types of

Prolog terms.

Having been generated by the P# compiler, this C# code together with the code for the other predicates in the user's program and the P# Dynamic Link Library (DLL) is presented to the C# compiler, resulting in an executable.

3 Idiomatic Compilation

We now discuss how this compilation scheme can be changed so that certain predicates are compiled to C# code which is closer to the code which a C# developer would have produced. This new compiler has several phases. The first phase compiles each predicate into an abstract syntax tree representation of a C# class containing a method called `idiomatic()` which consists of a block for each clause. Within each block, input variables are first extracted from the arguments, then each goal is executed in turn and finally output variables are copied back into the arguments. The second phase attempts to convert any recursive calls in the `idiomatic()` method into jumps back to the first block of the idiomatic method. If this proves possible, the third phase attempts to convert the body of the method into a `while` loop. The fourth phase then performs a liveness analysis, and the final phase converts the abstract syntax tree to C# code.

We will refer to those predicates which are compiled to more idiomatic C# as *idiomatic predicates*, and to those which will continue to use the original compilation scheme as *non-idiomatic predicates*. If we are going to translate some semi-deterministic predicates into more idiomatic C#, then we will find that when control passes from a non-idiomatic predicate to an idiomatic predicate, the arguments must be converted accordingly. A significant improvement in efficiency should be obtained if idiomatically compiled predicates work with native `int`s rather than with `IntegerTerm`s. Thus, when calling an idiomatic predicate which uses integers from non-idiomatic code we should convert the `IntegerTerm`s to `int`s. In order to call a non-idiomatic predicate from an idiomatic one, we would have to start a new Prolog interpreter—which would lead to an unacceptable performance penalty.

In light of the above, it seems sensible to idiomatically compile a large a slice as possible at the bottom of the call stack. Thus, whenever a non-idiomatic predicate calls an idiomatic one, we convert its arguments; and an idiomatic predicate never calls a non-idiomatic one. When an idiomatic predicate calls another idiomatic predicate there is no need for the conversion of arguments.

The `exec()` method of a non-idiomatic predicate is replaced in the idiomatic case by a method similar to the following which calls the `idiomatic()` method.

```
public override Predicate exec( Prolog engine ) {
    int outarg3 = 0;
    bool success = idiomatic( ( arg1.Dereference() ),
        ((IntegerTerm)( arg2.Dereference() )).value( ), out outarg3 );
    if( success ) {
        arg3.Unify( new IntegerTerm( outarg3 ), engine.trail );
        return cont;
    } else
        return engine.fail();
    }
}
```

In order to call an idiomatic version of a predicate from another idiomatic predicate it is essential that we know its type and mode signature. Thus, when multiple files are compiled at different times, it is necessary for each file to see the type and mode signatures of the others.

3.1 Generating naïve idiomatic code

We now discuss the first phase of the translation into idiomatic code. This produces code which despite being more recognisable as C# code is still not particularly idiomatic.

In the following, we assume that each clause of a predicate which is to be idiomatically compiled is a conjunction of goals. Disjunctions and the if-then-else construct can be compiled by creating dummy predicates.

A semi-deterministic predicate executes as follows. The first head goal which matches the calling query is found and then that clause begins to execute. If the end of the clause is reached with all the goals having succeeded then the predicate succeeds and exits at that point. If at any point one of the conjoined goals fails, and we have not encountered a cut, then we backtrack to the call to the predicate and execute the next clause that matches the calling query. If one of the goals fails and we have encountered a cut, then the predicate fails at that point. When all the matching clauses have been tried and have failed, the predicate fails and the call returns.

Figure 1 illustrates this control flow. The thick horizontal lines represent the execution of the different clauses of the predicate. The lines beginning in the middle of the horizontal lines indicate control flows which can occur between goals in the clause. The lines beginning at the end of the horizontal lines indicate control flows which can occur just after the final goal of the clause, that is, success.

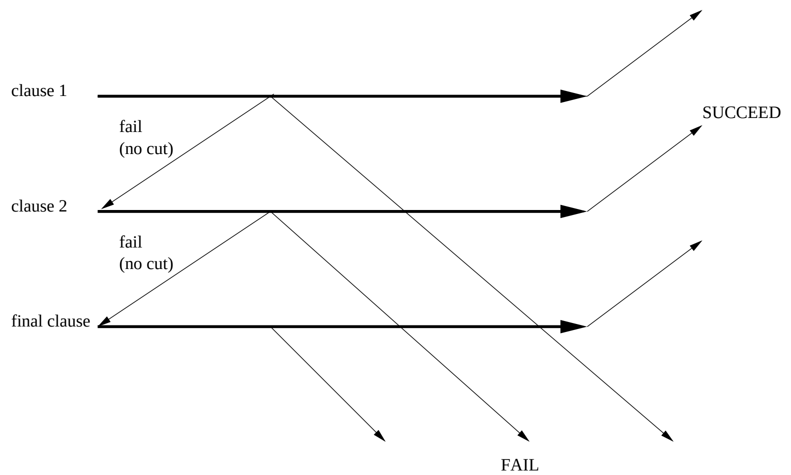

Figure 1: Control flow for a semi-deterministic predicate that only calls other semi-deterministic predicates

The difference with a predicate that may produce more than one solution because it calls predicates which themselves may have more than one solution, is that there may be backtracking to before a previous goal within a clause. This greatly complicates the control flow.

There are several ways in which this can be implemented. An elegant, but also inefficient, approach is to use exceptions. Another approach would be to use `do ... while( false )` blocks and jump to the next block by using a `break` statement. If Java was our target language, this would be the best approach. We opted to use C#'s `goto` construct. The `goto` construct is often shunned because of its tendency to produce unstructured code, see [7]. However, it is arguably acceptable to use the `goto` construct in a tightly controlled and structured way, see [13]. Indeed the C# language places significant restrictions on the use of the `goto` construct. It is not permitted to jump into a different block or into or out of a loop, for example.

We structure the code in the following way. Each clause is compiled into a block of C# code. Each block is preceded by a label of the form `clause1`, `clause2`, ...

The first thing that must be done in each clause block is to find whether that clause matches the arguments which have been passed to the predicate. In this stage, we also want to unpack any lists or structures and extract the values of input variables embedded in such structures.

For each argument of the predicate, we recurse through the structure of the argument, keeping track at each point of the path through the structural tree of that argument to the current point. Thus, on encountering the head: `p( f([X,Y], 5) )`, when processing the first argument, on reaching the variable `Y` we have a path of `list( tail, structure( f/2, 1, arg(1) ) )`. This means that we start with the first argument. This is a structure with functor `f/2`, we take its first argument. This is a list, and we take its tail. In the C# code this is translated into: `Term Y = arg1[1].Tail`.

In reaching this point, we will have already generated code which checks that the first argument really does have a functor of `f/2`, and that its first argument is a list. The code, assuming that it occurs in the first clause, is as follows:

```
if( !(  (arg1.Functor() != null) && (arg1.Functor().Equals( "f" )) &&
        (arg1.Arity() == 2) && (arg1[1].IsList())  ) ) goto clause2;
```

When the value of 5 is reached a C# statement is produced which tests that this part of the argument is 5: `if( !( arg1[2].Equals( 5 ) ) ) goto clause2;`.

In order that all this works, the class `Term` in the runtime system of P# is extended with several methods and properties. The `Head` and `Tail` properties return the head and tail of a `ListTerm`, failing with an exception if passed something other than a `ListTerm`. The `Term` class becomes an indexer, so that if the `Term`, `t` is a `StructureTerm`, then `t[2]` returns the second argument of the structure.

An `Equals(string s)` method is added to the `Term` class. This returns true if the `Term` is a `SymbolTerm` representing a symbol with functor `s` and arity 0. A similar method is provided, which deals with integers. Finally an `Equals(Term t)` method is provided which performs a content equality test.

Next, all of the variables which might be used within the clause are declared and those which are input values are initialised to the arguments passed to the idiomatic method.

As the clause is a conjunction of goals, each is executed in turn, using a `goto` to jump to the next clause if a goal fails. At the point that a cut occurs a Boolean variable called `cut`, which is initially set to false, is set to true. At the start of each clause after the first, we test the `cut` variable, and return false if `cut` is true, indicating that the predicate has failed. If there are any output arguments, then they must be set to default values before the method returns, thus in order to fail we jump to a block at the end of the method labelled `fail:`.

For example, if `a/2` is idiomatic, then the call `a( 1, 2 )` is translated to

```
if( !Predicates.A_2.idiomatic( 1, 2 ) ) goto <nextclause>;
```

We translate the Prolog relational infix predicates: `=:=`, `=\=`, `<`, `=<`, `>` and `>=` directly into their corresponding C# operators which have the same semantics.

After all of the goals have been executed, we need to assign the values held by variables representing output arguments to the output arguments of the idiomatic method. The idiomatic method returns true or false on success or failure respectively.

3.2 Tail Recursion Converted to Iteration

Often a predicate will contain a recursive call. In this case we have to decide between translating the call into recursion or iteration in the C# code. Recursion is easier from an implementation point of view, however iteration is preferable as it avoids the risk of stack overflow and is often more efficient. The only case where translation to iterative code can be done in a natural way is when the recursive call occurs as the final goal in the final clause. This is because at such a point we can merely jump back to the first clause having modified the arguments appropriately and maintain the same semantics for the code. Fortunately, for efficiency reasons, *tail recursion* is the most common form of recursion in Prolog code.

In order to translate a recursive call into iterative code, it is also necessary that any output argument in the head of the tail-recursive clause, is matched by an identical output argument in the recursive call itself. For example, if we have a predicate `p/2` where only the second argument is an output argument, then the clause: `p( X, Y ) :- X1 is X - 1, p( X, [Y] )` cannot be converted to iterative code (using the current scheme), whereas `p( X, Y ) :- X1 is X - 1, p( X1, Y )` can. In this latter case the code generated for the recursive call, prior to liveness analysis and rewriting as a `while` loop, will be: `arg1 = X1;  goto clause1;`.

The conversion to iteration would not be performed by the C# compiler if it were not performed by the idiomatic code generator.

3.3 Rewriting Blocks as a while Loop

In cases where the tail recursive optimization has been applied, the intermediate code generated often contains code of the following form:

```
clause1: {
    if( <Condition> ) goto clause2;
    <Code Block 1>
    return true;
}
clause2: {
    <Code Block 2>
    goto clause1;
}
```

Code of this form can be rewritten as the following, provided that there are no other `goto` statements in the code:

```
while( <Condition> ) { <Code Block 2> }
<Code Block 1>
return true;
```

Usually, there are other `goto`s however, namely statements of the form `if( cut ) goto fail;` which occur in the C# code for all but the first clause. Since the `fail:` block is usually small, it is acceptable to in-line this block wherever a `goto fail` occurs.

Multiple base cases and/or multiple step cases can also be handled, but we omit details here.

One major impediment to the use of `while` loops is instances where there are other branches to the next clause within the code for a clause. This occurs when a non-tail recursive call is made at some point in the body of one of the clauses. This is not a problem, provided that the non-tail recursive call cannot fail. If it cannot fail, then the `goto` will never be executed. Thus, we have a motivation for distinguishing deterministic predicates from those which are merely semi-deterministic.

3.4 Liveness Analysis

The code produced by the above tends to have more variables than are necessary. Often code similar to the following is produced.

```
int X = arg1;  int _1 = arg2;  int Z;  Z = X + 1;  arg1 = Z;
```

where `_1` is never used. This can be replaced by `arg1 = arg1 + 1` by applying liveness analysis [2] to the code. This is a standard technique which involves analyzing which variables are live at the same time at each point during the execution of the program. If two variables are never live at the same time, then one of the variables can be consistently changed to the other.

Using the `while` loop rewrite and liveness analysis, the following predicate, `len`, which can be used to find the length of a list:

```
len( [], Z, Z ).
len( [_|T], A, Z ) :- A1 is A + 1, len( T, A1, Z ).
```

has the idiomatic method shown below:

```
public static bool idiomatic( Term arg1, int arg2, out int arg3 ) {
    while( !( (( arg1 ).Equals( "[]" )) ) ) {
        if( !( ((arg1).IsList( )) ) ) { arg3 = 0;  return false; }
        arg1 = (arg1).Tail;
        arg2 = ( arg2 + 1 );
    }
    {   arg3 = arg2;  return true; }
}
```

The extraneous brackets, { and }, at the end of this method are necessary in some cases to avoid a difficulty with C#'s variable scoping rules. Notice also that the liveness analysis has determined that the head of `arg1` should be removed, and that `arg2` should be incremented in each iteration of the loop.

3.5 Example Code—The Eight Queens Problem

Suppose that we are using the following code to solve the Eight Queens Problem:

```
queens(N,Qs) :- range(1,N,Ns), queens(Ns,[],Qs).

queens([],Qs,Qs).
queens(UnplacedQs,SafeQs,Qs) :- select(UnplacedQs,UnplacedQs1,Q),
    not_attack(SafeQs,Q), queens(UnplacedQs1,[Q|SafeQs],Qs).
```

```
mode( not_attack( in, in ) ).
type( not_attack( term, int ) ).
not_attack(Xs,X) :- not_attack(Xs,X,1).

mode( not_attack( in, in, in ) ).
type( not_attack( term, int, int ) ).
not_attack([],_,_) :- !.
type( not_attack/3, 2, [N1=int] ).
not_attack([Y|Ys],X,N) :- X =\= Y+N, X =\= Y-N, N1 is N+1, not_attack(Ys,X,N1).

select([X|Xs],Xs,X).
select([Y|Ys],[Y|Zs],X) :- select(Ys,Zs,X).

mode( range( in, in, out ) ).
type( range( int, int, term ) ).
range(N,N,[N]) :- !.
type( range/3, 2, [M1=int] ).
range(M,N,[M|Ns]) :- M < N, M1 is M+1, range(M1,N,Ns).
```

This code, less the mode and type declarations, forms one of the benchmarks provided with Prolog Café. Observe that we have given mode and type declarations to the predicates `not_attack/2`, `not_attack/3` and `range/3` as these are the predicates which can be idiomatically compiled. The type declarations with one argument refer to the types of the arguments of the predicate as a whole. The type declarations with three arguments: the functor, the clause number and the types of the variables, give the types for variables in the clause of that number which cannot be inferred from the type declaration for the clause as a whole. In fact, it is possible to infer that `N1` and `M1` are `int`s, see the section on future work. If a type is not given, the generic type, `term`, is used.

The predicate `select/3` cannot be idiomatically compiled as it is non-deterministic. Since `select/3` occurs below `queens/2` and `queens/3` in the call tree, these two predicates cannot be idiomatically compiled. All non-idiomatic predicates continue to be compiled using the original Prolog Café/P# compilation scheme.

The predicate `range/3` is compiled into the following method. The predicate `range/3`, takes a pair of integers and produces the list of consecutive integers starting with the first integer and ending with the second. In this example, some white-space has been removed to save space.

```
public static bool idiomatic( int arg1, int arg2, out Term arg3 ) {
   bool cut = false;
   clause1: {
      if( !( (arg2 == arg1) ) ) { goto clause2; }
      arg3 = new ListTerm( new IntegerTerm( arg1 ), Term.Nil );
      return true;
   }
   clause2: {
      Term Ns;
      if( !( arg1 < arg2 ) ) { goto fail; }
      int M1 = ( arg1 + 1 );
      if( !( Predicates.Range_3.idiomatic( M1, arg2, out Ns ) ) ) { goto fail; }
```

```
        arg3 = new ListTerm( new IntegerTerm( arg1 ), Ns );
        return true;
    }
    fail: { arg3 = null;  return false; }
}
```

Notice that we could not compile this into a `while` loop because the output argument in the recursive call is not the same as in the head of the clause containing the call.

The predicate `not_attack/3` is translated into a `while` loop.

4 Performance Measurement

We benchmarked both the idiomatic and original versions of P#, against Jinni 2004 [11] (using Sun's Java SDK 1.4.2), MINERVA 2.4 [15] and SICStus Prolog 3.10.1 [19] as shown in Table 1. The speed-up column gives the factor by which P# is speeded-up by the optimization.

Table 1: Speed-up due to the use of idiomatic code and mode/type declarations (times in ms)

Benchmark	idiomatic P# time	original P# time	P# speed-up	Jinni time	Minerva time	SICStus time
browse	125	1360	11	1250	703	63
poly_25	1609	6781	4.2	6172	3500	226
queens (10 all)	438	1954	4.5	1250	1609	78
queens (16 first)	203	1234	6.1	734	1078	63
nreverse (2000)	485	4922	10	8047	921	62
tak	31	10969	350	11094	3938	437
zebra	140	140	1.0	62	78	16

The tests were carried out on a 2 GHz Pentium 4 machine with 512 Mb of memory running Windows XP Professional. All times are in milliseconds.

The `zebra` benchmark is not changed as its computational predicates are all non-deterministic. The speed-ups of `queens` and `browse` are almost entirely due to the idiomatic compilation of the `not_attack/3` predicate and the list concatenation predicate respectively. All of the predicates of the `poly_25` benchmark, which computes $(1 + x + y + z)^{25}$ symbolically, could be idiomatically compiled.

It should be noted that the optimizations we employed are of particular benefit to numerical code and that Jinni, MINERVA and SICStus Prolog did not have the benefit of mode or type declarations. Observe that the 'most numerical' benchmark, `tak`, which computes the Takeuchi function with arguments 24, 16 and 8, experienced the most significant speed-up. However, some of the other benchmarks involve list operations, and these too were speeded up. The idiomatically compiled list operation predicates can operate on heterogeneous lists.

5 Future Work

It is possible to infer more of the types than is done at present. For example, if `A` is an `int` then so is `A+1`, and given `A1 is A + 1`, we can infer that `A1` is an `int`. Also, the programmer currently has to specify which predicates can be idiomatically compiled. Rectifying these issues will form part of future work.

A possible extension is to convert failure driven loops in Prolog to iterative loops in C#. We will not always be able to convert such a loop to an iterative C# loop, as it is possible that one of the predicates called in the loop, possibly via other predicates, cannot be compiled to idiomatic code.

We would also like to compile primitives that modify the database (assert etc.) to more idiomatic C#, but there is a problem due to the fact that the database can be modified from non-idiomatic code and then called from idiomatic code or *vice versa*.

Finally, we intend to idiomatically compile concurrent code.

6 Related Work

Work has been done on translating Mercury [9] to high-level C, see [10]. That paper lists the advantages of translation to higher-level code. Generating low-level code usually leads to the Prolog compiler having to do more work, producing less readable code and often ending up working against the compiler of the language that Prolog is being compiled to.

As with Mercury we exploit type and mode information, but we do not attempt to idiomatically compile non-deterministic predicates. This is because our existing scheme for compilation of such predicates is as efficient as a more idiomatic compilation. P# generates highly idiomatic and readable code in many instances, often being able to translate a simple predicate into a `while` loop. As such simple predicates often occur as leaf predicates that are called frequently, this significantly reduces the number method calls, and allows us to avoid this overhead.

As we do not need to implement cuts for non-deterministic predicates, there is no need to unwind the stack when a P# cut occurs as there is when a commit occurs in a non-deterministic Mercury predicate. Thus, with P# Prolog, testing a cut flag after each failure is an acceptable solution.

7 Conclusion

The efficiency of P# and the readability of the code it produces can be significantly improved by compiling to more idiomatic C# with the assistance of mode and type declarations. This is because the C# compiler is designed to compile code written by human programmers. Particularly significant improvements are observed for Prolog programs that are predominantly numerical. Our compilation scheme avoids the overheads of the Prolog stacks used by the WAM in situations where they are not necessary. Our technique tends to be able to compile those predicates which are called most frequently. Thus, even when only a few of the program's predicates can be idiomatically compiled, efficiency will often still be improved.

Acknowledgments

I would like to acknowledge Mutsunori Banbara and Naoyuki Tamura, the authors of Prolog Café, the tool on which ours is based.

I would also like to acknowledge the kind advice and assistance of Stephen Gilmore; and the support of the EPSRC. The helpful comments of the anonymous referees are gratefully appreciated.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT press, 1991.

[2] A. W. Appel. *Modern compiler implementation in C.* Cambridge University Press, 1998.

[3] M. Banbara and N. Tamura. Translating a Linear Logic Programming language into Java. *Electronic Notes in Theoretical Computer Science*, 30(3), 1999.

[4] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer, 4th edition, 1994.

[5] P. Codognet and D. Diaz. WAMCC: Compiling Prolog to C. In *International Conference on Logic Programming*, pages 317–331, 1995.

[6] J. J. Cook. P#: A concurrent Prolog for the .NET Framework. *Software: Practice and Experience*, 34(9):815–845, 2004.

[7] E. W. Dijkstra. Goto considered harmful. *Communications of the ACM*, 11(3):147–8, 1968.

[8] W. Hans. A complete indexing scheme for wam-based abstract machines. In *PLILP 1992*, pages 232–244, 1992.

[9] F. Henderson et al. *The Mercury Language Specification.* `http://www.cs.mu.oz.au/research/mercury/information/documentation.html`.

[10] F. Henderson and Z. Somogyi. Compiling mercury to high-level C code. In *Computational Complexity*, pages 197–212, 2002.

[11] Jinni home page. `http://www.binnetcorp.com/Jinni/`.

[12] jProlog home page. `http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/`.

[13] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6(4):261–301, 1974.

[14] J. Liberty. *Programming C#.* O'Reilly, 2001.

[15] MINERVA home page. `http://www.ifcomputer.com/MINERVA/`.

[16] The Microsoft developer .NET home page. `http://msdn.microsoft.com/net`.

[17] P# home page. `http://www.lfcs.ed.ac.uk/psharp`.

[18] Prolog Café home page. `http://pascal.cs.kobe-u.ac.jp/~banbara/PrologCafe/index-jp.html`.

[19] SICStus Prolog home page. `http://www.sics.se/sicstus/`.

[20] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, pages 159–173. Springer, LNCS 456, 1990.