Using impurity to create declarative interfaces in Mercury

Tyson Dowd, Peter Schachte, Fergus Henderson, and Zoltan Somogyi
Department of Computer Science and Software Engineering
University of Melbourne
Victoria 3010, Australia
{trd,schachte,fjh,zs}@cs.mu.oz.au

ABSTRACT

The logic/functional language Mercury allows the programmer to annotate predicates and functions to mark impure predicates, allowing impure code to be safely integrated into a declarative language. By using purity declarations with the foreign language interface, programmers can take advantage of many of the features of a high level programming language while writing imperative code to interface with existing imperative libraries. This paper outlines the purity system in Mercury and how it affects operational semantics, compares this purity system with other approaches to declaring impurity in a pure language, and gives an extended example of how impurity and foreign language interfaces can work together to simplify the chore of writing declarative interfaces to libraries.

1. INTRODUCTION

Purely declarative languages often face the problem of interfacing with external systems that do not share the same clean semantics. In particular, external libraries will often provide an API consisting of operations that have side effects. The solution to using such interfaces from a declarative language is to develop a way of expressing the operations one wishes to perform on these systems in some declarative way.

One of the first external interface problems a declarative language needs to tackle is that of input and ouput. IO has the property that it must be done sequentially, and cannot be undone. Monads[8], linear types[7] and unique modes[5] are popular techniques used to model this behaviour; they all provide sequencing and finality in a declarative fashion.

Various techniques have been developed for coping with this problem in Mercury (which probably apply equally to other declarative languages). For some time while developing libraries for Mercury, we would write a large amount of C code with a declarative interface. As we improved our foreign function interface, we found that we could model more atomic operations using smaller pieces of C code that were modelled as performing IO. These atomic operations could be chained together to perform a sequence of operations in an imperative style, with the sequencing of IO forcing the commands to be executed imperatively. However, this approach didn't fit well when writing code that wasn't performing IO.

This paper explores another approach to interfacing with impure code, by placing annotations on predicates and functions indicating their level of purity. These annotations affect the operational semantics of the impure predicates, preventing the compiler from performing optimizations and transformations that rely upon the language being side-effect free, while allowing the interface programmer to use many of the language features of Mercury. Our mechanism allows programmers to define a point at which the interface is pure, and aims to keep the impurity very visible so as to warn program maintainers of the code behind the interface about the possible fragility of code they may be about to modify.

While we use the example of interfacing Mercury to C, the techniques we present should work for interfacing any declarative language to any imperative language. Within the scope of this paper, C is sufficiently general to represent the required features of any imperative language. We also give examples of similar Haskell techniques where appropriate. However, since foreign language integration is an ongoing research topic in the Haskell community, our examples may not be in synchronization with the very latest research proposals or development implementations.

The rest of this paper is organized as follows. Section 2 introduces the relevant aspects of the Mercury language and describes how other declarative languages handle impurity. Section 3 describes the purity declaration system in Mercury. Section 4 gives an example of how we can use the purity declarations in Mercury, and describes the implementation. Section 5 presents comparisons of the Mercury purity declaration approach with other methods for creating declarative interfaces.

2. BACKGROUND

2.1 Mercury

Mercury is a pure logic/functional programming language intended for general purpose large-scale programming.

We assume familiarity with the basic concepts of Mercury, including types, modes and determinisms; these are introduced in the Mercury tutorial [2] and described in detail in the Mercury language reference manual [5], which are available from the Mercury home page `http://www.cs.mu.oz.au/mercury/`. In particular we will be using the Mercury foreign language interface and unique mode system, which are described in the reference manual.

It is worth mentioning that the examples of Mercury given in this paper use purity declarations on functions, a language feature which is still under implementation. The current stable release, version 0.9.1[1], (and even the recent development releases) don't yet support purity declarations on functions. Functions are not fundamentally different to predicates in Mercury, except in their syntax. Our use of functions is just a convenience, if they are replaced with predicates this code will work with the current stable release.

2.2 Impure Interfaces

It is very common to want to access an imperative library while developing a declarative program, either to access systems level features or to leverage the work of others who have developed powerful libraries. Typically libraries written in language such as C provide only an imperative interface. When writing applications in declarative languages (and particularly new declarative languages) it is common to spend a great deal of time writing declarative interfaces in C.

One common technique for writing such interfaces is to write a new declarative interface in C, and then use a foreign function interface to call the declarative interface. This allows the programmer complete flexibility in designing the declarative interface, but requires programming in C, with none of the niceties of your declarative language at your disposal, such as automatic memory management, advanced data structures and pattern matching.

While the absence of such features makes programming in languages such as C rather painful, it becomes even worse when writing large bodies of code – as is typically required when turning an imperative interface into a declarative one. These code bodies need to be integrated into runtime systems such as the garbage collector, which may mean painstakingly annotating C code with garbage collection information.

The obvious solution to this problem is to write less C code. This is typically done by using the system that performs IO to perform arbitrary imperative functions. Haskell uses monads to solve such problems, for example:

```
getChar :: IO Char
getChar = ccall getchar
putChar :: Char -> IO ()
putChar = ccall putchar
```

In Mercury the same thing is expressed using unique modes and a uniquely threaded IO state argument.

```
:- pred getchar(char, io__state, io__state).
:- mode getchar(out, di, uo) is det.
:- pragma import(getchar(out, di, uo),
        [will_not_call_mercury], "getchar").
```

The `will_not_call_mercury` annotation has a similar meaning to GHC's `unsafe`: it says that this code can be relied upon not to call back into Mercury, which allows the compiler to generate tighter, less conservative code. The use of unique IO states is very similar to using an IO monad.

This technique works fine for performing impure operations that are performing IO with the real world. However, the presence of IO monads or IO states introduces a problem. Any time we have some code and want to perform one of these operations, we need to return the monad from the code or pass the IO state around to it.

This is fine if we are actually performing IO, since we want to make sure all IO operations are consistent. But there are times in which we want to perform some side-effects that are independent of IO. For example, a store implemented in C doesn't necessarily change the external world, and even if it does write to disk or communicate across the network, these changes may be structured in such a way that they are invisible to the program, and hence independent of all other IO operations.

A solution to this problem used in many Haskell implementations is to use the function `unsafePerformIO`, which makes IO monads magically disappear.

```
noMonadFoo = unsafePerformIO (foo)
```

This will perform foo, but will discard any IO monad. This allows the programmer to put a declarative interface on impure code, without using the IO system. When unsafePerformIO is used, the programmer incurs a proof obligation. In theory they need to prove that the set of operations performed by unsafePerformIO are side-effect free (although there is no way for the compiler to check such a proof is correct, or even if it has been provided). Individual sub-actions in the argument to unsafePerformIO can perform whatever side-effects they wish, it is the total operation that must be side-effect free.

Another possible solution is to find some other way to implement the operational semantics we get from our IO system. This is the approach taken by the Mercury purity declaration system, which may limit the re-ordering and optimization opportunities of the compiler in much the same way as a uniquely threaded IO state, but do not require IO states to be threaded through the code.

3. PURITY DECLARATIONS

The Mercury impurity system is designed with the aim of making it easier for programmers to write impure code. It is intended that this code be used for writing libraries that interface with C code, or system primitives for language features or extensions.

The idea behind purity declarations is to minimize the amount of foreign code that needs to be written while writing a declarative interface. Mercury provides useful high level programming features, and it easy to access other high level code written in Mercury from Mercury. The alternative is to interface to Mercury code and use Mercury features from within C, but this is as painstaking and error-prone as any imperative programming.

It is not intended that programmers use the imperative features of Mercury to do general purpose programming. Mercury's purity system is designed to discourage casual use.

3.1 Purity Levels

Mercury distinguishes between three "levels" of purity.

3.1.1 Pure

Pure predicates and functions always return the same outputs given the same inputs. They do not interact with the "real" world (i.e., do any input/output) without taking a unique `io__state` as input and returning one as output, and do not make any changes to any data structure that will not be undone on backtracking (unless the data structure would be unreachable on backtracking).

The behaviour of other pure predicates or functions is never affected by the invocation of other predicates or functions. It is possible for the invocation of pure predicates to affect the behaviour of impure predicates (for example, an impure predicate could collect benchmarking information).

By default, Mercury predicates are pure. Without using the foreign function interface or calling other impure predicates it is impossible to write impure code in Mercury.

3.1.2 Semipure

Semipure predicates are just like pure predicates, except that their behaviour may be affected by the invocation of impure predicates. That is, they are sensitive to the state of the computation other than as reflected by their input arguments, though they do not affect the state themselves.

Predicates that read some sort of global variable are typical examples of semipure.

3.1.3 Impure

Impure predicates may do almost anything, including changing the state of the computation.

They do have to be type-, mode-, determinism- and uniqueness-correct at the call boundary. The compiler must be able to reliably generate calls to them with correct calling conventions.

3.2 Least Pure

The three levels of purity have a total ordering defined upon them (which we will simply call the purity), where *pure* > *semipure* > *impure*. It is common to want to find the lowest purity level in a set of purity levels; we call the lower bound the *worst purity* of the set[1].

[1] Some wags have called *worst purity* a value judgement, but it is one we feel happy to make.

3.3 Semantics

It is important to the proper operation of impure and semipure code, to the flexibility of the compiler to optimize pure code, and to the semantics of the Mercury language, that a clear distinction be drawn between ordinary Mercury code and imperative code written with Mercury syntax.

How this distinction is drawn will be explained below; the purpose of this section is to explain the semantics of programs with impure predicates.

A declarative semantics of impure Mercury code would be largely useless, because the declarative semantics cannot capture the intent of the programmer. Impure predicates are executed for their side-effects, which by definition are not part of their declarative semantics. Thus it is the operational semantics of impure predicates that Mercury must specify, and Mercury compilers must respect.

For more general information about semantics of Mercury programs, see the Mercury Language Reference manual[5].

3.3.1 Strict Sequential Semantics

There is an operational semantics of Mercury programs called the strict sequential operational semantics. In this semantics, the program is executed top-down, starting from 'main/2', and function calls within a goal, conjunctions and disjunctions are all executed in depth-first left-to-right order. Conjunctions and function calls are "minimally" reordered as required by the modes: the order is determined by selecting the first mode-correct sub-goal (conjunct or function call), executing that, then selecting the first of the remaining sub-goals which is now mode-correct, executing that, and so on. Function application is strict, not lazy.

All Mercury implementations are required to provide a method of processing Mercury programs which is equivalent to the strict sequential operational semantics. This doesn't need to be the default method, there are more relaxed semantics that are more ameanable to re-ordering (and hence optimization and transformation).

3.3.2 Impure Semantics

The operational semantics of a Mercury predicate which invokes impure code is a modified form of the strict sequential semantics. Impure goals may not be reordered relative to any other goals; not even "minimal" reordering as implied by the modes is permitted. If any such reordering is needed, this is a mode error. However, pure and semipure goals may be reordered as long as they are not moved across an impure goal[2]. Execution of impure goals is strict: they must be executed if they are reached, even if it can be determined that that computation will lead to failure.

Semipure goals can be given what we call a "contextual" declarative semantics. They cannot have any side-effects, so it is expected that, given the context in which they are

[2] Pure goals cannot be moved across an impure goal because impure goals can do *anything*, including modify input arguments (see the example in Section 4) or change the execution state of the program. The mode system cannot help the compiler decide which re-orderings are safe across an impure goal.

called (relative to any impure goals in the program), their declarative semantics fully captures the intent of the programmer.

Thus a semipure goal has a perfectly consistent declarative semantics, until an impure goal is reached. After that, it has another (possibly different) declarative semantics, until the next impure goal is executed, and so on. Mercury compilers must respect this contextual nature of the semantics of semipure goals; within a single context, a compiler may treat a semipure goal as if it were pure.

3.4 Declaring Impure Predicates and Functions

Every Mercury predicate or function has exactly two purity values associated with it. One is the declared purity of the predicate or function, which is given by the programmer. The other value is the inferred purity, which is calculated from the purity of goals in the body of the predicate or function.

A predicate or function is declared to be impure or semipure by preceding its pred or func declaration with the word impure or semipure, respectively. That is, a declaration of the form:

```
:- impure pred Pred(Arguments...).
```

or

```
:- semipure func Func(Arguments...) = (Result).
```

declares the predicate Pred to be impure and the function Func to be semipure.

3.5 Calling Impure Code

Every call to a Mercury predicate or function also has exactly two purity values associated with it. One is the declared purity of the call, which is given by the programmer as an annotation of the call. The other value is the inferred purity, which is the purity of the predicate or function.

It is an error for the declared purity of a goal to be more pure than the inferred purity, the compiler should flag this error. The compiler should issue a warning if the declared purity of a goal is less pure than its inferred purity. This is required to be just a warning because there is no semantic problem with declaring a goal as less pure than it actually is, but generating an error could cause backwards compatibility problems (consider the effects of increasing the purity of a predicate but leaving the callers declared purity unchanged).

This rule means that if a predicate is impure or semipure, all calls to it must be preceded with the word impure or semipure, respectively. If there is no purity annotation on a goal, it is assumed to be declared pure.

```
impure predname(Arguments...)
```

If a function[3] is impure or semipure, it must be called as part of a simple unification with a variable, and this unification must be preceded by the word impure or semipure respectively.

```
impure X = funcname(Arguments...)
```

This restriction to the use of impure functions is not necessarily permanent, however it simplifies the implementation, and sidesteps possible problems with syntax and semantics of nested function calls. In addition, programmers can find the impure or semipure declarations in a uniform position – at the beginning of a goal.

Note that only predicate calls and unifications of variables with functions need to (and are permitted to) be prefixed with impure or semipure. Compound goals (e.g., if-then-else or disjunctions) never need this.

The requirement that impure or semipure calls be marked with impure or semipure allows someone reading the code to tell which goals are not pure, making code which relies on side effects somewhat less mysterious. Furthermore, it means that if a call is not preceded by impure or semipure, then the reader can rely on the call having a proper declarative semantics, without hidden side-effects.

3.6 Inferred Impurity

Predicates that are implemented in terms of impure or semipure predicates are assumed to have the worst impurity of the goals in their body. The declared purity of a predicate must not be more pure than the inferred purity; if it is, the compiler must generate an error. If the declared purity is less pure than the inferred purity, the compiler should issue a warning (this is similar to the above case for goals). Because the inferred purity of the predicate is calculated from the declared purity of the calls it executes, the worst impurity is propagated up from callee to caller through the program.

However, some predicates which call impure or semipure predicates are themselves pure. The only way for the programmer to stop the propagation of impurity is to explicitly promise that a predicate or function is pure.

Of course, the Mercury compiler cannot verify that a predicate is pure, so it is the programmer's responsibility to ensure this. If a predicate is promised pure and is not, the behaviour of the program is undefined.

The programmer may promise that a predicate is pure using the `promise_pure` pragma:

```
:- pragma promise_pure(Name/Arity).
```

[3]Impure functions are an experimental feature, however as they seem to be extremely convenient for interfacing with C functions (see the Section 4) it is likely that they will eventually become part of the Mercury implementation. Impure functions can be wrapped in impure predicates if purity declarations on functions are unavailable, but this is inelegant so we will assume they are available.

3.7 Examples

The following example illustrates how a pure predicate may be implemented using impure code. Note that this code is not reentrant and so is not intended for practical use. It is meant only as an example.

```
:- pragma c_header_code("#include <limits.h>").
:- pragma c_header_code("Integer max;").

:- impure pred init_max is det.
:- pragma c_code(init_max,
        [will_not_call_mercury],
        "max = INT_MIN;").

:- impure pred set_max(int::in) is det.
:- pragma c_code(set_max(X::in),
        [will_not_call_mercury],
        "if (X > max) max = X;").

:- semipure pred get_max(int::out) is det.
:- pragma c_code(get_max(X::out),
        [will_not_call_mercury],
        "X = max;").

:- pragma promise_pure(max_solution/2).
:- pred max_solution(pred(int), int).
:- mode max_solution(pred(out) is multi,
        out) is det.

max_solution(Generator, Max) :-
        impure init_max,
        (   Generator(X),
            impure set_max(X),
            fail
        ;   semipure get_max(Max)
        ).
```

The predicate `max_solution` is our pure predicate. It finds the maximum solution of all the solutions of the Generator predicate. We could write this code in a declarative fashion using the standard logic programming all solutions predicate (called `solutions`). The declarative code would find all the solutions to Generator and take the maximum, but would require storing all the solutions. This program has the same declarative meaning, but is significantly more efficient.

The predicate `max_solution` works by keeping the current maximum solution in a global variable in C. It generates a solution, saves it to the C variable if it is higher than the current maximum, and then fails. The failure will cause the execution engine[9] to backtrack to the generator and try to generate another solution. Eventually the generator will run out of solutions and fail, leaving `get_max` to retrieve the maximum result and return it. An important feature of the impure code is that it saves a result across a disjunction.

Note that the Generator is defined to have a determinism of `multi` which means it is guarranteed to have at least one solution. This is important, if the generator could fail immediately we might get `INT_MIN` as our maximum solution. Mercury's mode and determinism checking system will ensure that only generators that meet this requirement can be passed to `max_solution`.

The Mercury standard library contains alternatives to the `solutions` predicate such as `unsorted_aggregate` that are more general than the `max_solution` predicate given here, but will give much the same efficiency. Their declarative semantics are given in terms of solutions, but they are implemented using impure primitives that swap memory allocation to a separate "solutions" heap, copy solutions from one heap to another, use impure mutable variables to store an accumulator, and call higher order terms to add items to the accumulator. Our example predicate `max_solution` is just an instance of this that uses `set_max` and `get_max` as a mutable accumulator.

In fact, the solutions predicate itself is implemented using `unsorted_aggregate` with a list accumulator and `cons` to add items to the accumulator. The list of solutions is sorted, and finally `solutions` is promised to be pure.

4. EVALUATION

The Mercury impurity system, in conjunction with the Mercury foreign function interface gives the programmer a powerful tool to build declarative libraries based on foreign functions.

Our example in this case is the zlib general purpose compression library, by Jean-loup Gailly and Mark Adler. This library is very widely distributed, and supports various interfaces to perform compression. In particular we will examine the stream-based compression interface that does buffered compression.

4.1 The C Interface

The zlib stream-based compression and decompression interface is typical of imperative libraries - a structure containing compression stream information is passed to a bunch of C functions.

```
typedef struct z_stream_s {
    Bytef    *next_in;
    uInt     avail_in;
    uLong    total_in;
    ... etc ...
} z_stream;
```

In typical C fashion, the structure is passed by reference and destructively updated by these functions.

```
int deflateInit(z_stream *strm, int cmplevel);
int deflate(z_stream *strm, int flush);
int deflateEnd(z_stream *strm);

int inflateInit(z_stream *strm);
int inflate(z_stream *strm, int flush);
int inflateEnd(z_stream *strm);
```

These functions initialize streams, compress (deflate) or decompress (inflate) and finalize the `z_stream` data structure.

The streams are buffer based, we can compress and decompress on the fly without requiring the entire input stream, hence a parameter controls flushing partial compressions or decompressions. We can also specify the compression level.

There are three problems that stop us from using the C functions as a declarative interface. First, `strm` is passed by reference and destructively updated; section 4.3 shows how this can modelled naturally in Mercury. Second, these operations must be performed in the correct order, because executing them out of order can lead to memory corruption. Section 4.5 shows how correct order can be enforced. Third, although the C structure `z_stream` is the same for both compression and decompression, you can't safely mix these operations on the same stream. Section 4.4 gives a programming technique that prevents such mistakes.

This C interface is a perfect candidate for using impurity declarations. Our plan is to use these functions directly as our impure building blocks, and then to create a declarative interface by fixing the destructive update, the operation mixing and ordering problems. We are also going to need to add some additional C code to initialize, read and modify the `z_stream` data structure.

For this code we will assume the presence of a library that provides the type `buffer` for creating and modifying buffers of bytes. The example will only show the compression half of the interface in detail; the decompression operations are symmetrical.

4.2 The Impure Mercury Interface

We begin by declaring impure Mercury functions that are a direct mapping from the C functions.

```
:- impure func deflateinit(z_stream, int) = int.
:- pragma import(deflateinit(in, in) = out,
        [will_not_call_mercury], "deflateInit").

:- impure func deflate(z_stream, int) = int.
:- pragma import(deflate(in, in) = out,
        [will_not_call_mercury], "deflate").

:- impure func deflateend(z_stream) = int.
:- pragma import(deflateend(in) = out,
        [will_not_call_mercury], "deflateEnd").
```

Similarly we define impure functions for the inflation functions, which are almost identical to these functions.

We need to write code to allocate and perform some basic initialization of the `z_stream` structure, given a pointer to the start of the output buffer and its length.

```
:- impure pred z_stream_init(buffer_internal::in,
        int::in, z_stream::out) is det.
:- pragma c_code(z_stream_init(Buf::in, Size::in,
        ZStr:: out), [will_not_call_mercury], "

    z_stream *stream;
        /* Allocate a new stream */
    stream = GC_malloc(sizeof(z_stream));
        /* Initialize some fields to NULL */
    stream->zalloc = Z_NULL;
    stream->zfree = Z_NULL;
    stream->opaque = Z_NULL;
     /* Set the start of the output buffer */
    stream->next_out = (Bytef *) Buf;
     /* Set the size of the output buffer */
    stream->avail_out = Size;
     /* Return the stream to Mercury */
    ZStr = (Word) stream;
").
```

Note that Mercury currently uses the Boehm (et al)[3] conservative garbage collector for C, so the allocation code does not need to be accompanied by information for the garbage collector. The other dynamically allocated data structures in our example system are allocated by Mercury code, so for them we do not even need to call `GC_malloc`.

We will require a number of predicates for getting and setting fields in the C struct:

```
:- impure pred z_stream_set_input_string(
        z_stream::in, string::in) is det.
:- impure pred z_stream_set_total_out(
z_stream::in, int::in) is det.
:- semipure pred z_stream_get_total_out(
        z_stream::in, int::out) is det.
:- pragma c_code(z_stream_get_total_out(
                ZStr::in, Out::out),
        [will_not_call_mercury], "
        z_stream *stream = (z_stream *) ZStr;
        Out = stream->total_out;
").
```

and similar definitions for other fields.

Although we could write more complicated C code that is pure, it is expedient to write it quickly in a typical C style – our goal is to write the pure code in Mercury. In a more sophisticated foreign function interface it is quite likely that these sort of predicates would be generated automatically from the C struct definition.

The above functions and predicates are all less than pure. We don't want to allow users of our pure interface to see (or even worse, use) these impure functions, so it is a good idea to hide them in a separate module that isn't visible at a library level (if we are exposing the declarative zlib as a Mercury library), or a sub-module that isn't visible from a module level (if we want to expose zlib as a Mercury module).

4.3 Destructive Update

Our first task in putting a pure front-end onto this library is to deal with the issue of pass by reference destructive update. We will replace the use of a destructively updated variable with a pair of uniquely moded variables.

```
:- type z_info ---> z_info(z_stream, buffer).
```

```
:- type z_stream ---> z_stream(c_pointer).
:- type compressed_string == buffer.
```

The type `z_info` will represent a compression stream. It contains the C data structure `z_stream` and a simple byte buffer. A pointer to somewhere in the internals of the byte buffer is stored by the `z_stream`, which will use it as an output buffer while processing a stream of input. To eliminate the possibility of someone disrupting with this delicate structure, `z_info` is defined as an abstract type. We also define `compressed_string` as as an abstract type, we will simply use a buffer of bytes as its concrete representation.

We can then define our pure external interfaces as operating on `z_info`.

```
:- pred init_compression(z_info::uo) is det.
:- pred compress(string::in, z_info::di,
        z_info::uo) is det.
:- pred finish_compression(compressed_string::out,
        z_info::di) is det.
```

Note that these have no purity declaration on them, so they have the default annotation of being pure. Each of these predicates can be defined in terms of a combination of impure predicates above, or any of the following impure predicates we define.

```
:- impure pred init_stream_with_buffer(
        z_stream::out, buffer::out) is det.
init_stream_with_buffer(ZStr, Buffer) :-
    alloc_buffer(1024, Buffer),
    BufferIntrn = get_buffer_internal(Buffer),
    impure z_stream_init(BufferIntrn,
        1024, ZStr).

:- pragma promise_pure(init_compression/1).
init_compression(ZInfo) :-
    impure init_stream_with_buffer(ZStr, Buffer),
    ComprLevel = 5,
    impure _Result = deflateinit(ZStr, ComprLevel),
    impure make_z_info(ZStr, Buffer, ZInfo).
```

For brevity[4], we've used a few utility predicates that are not defined in this paper. The names of these predicates should make their operation clear.

This code sequence allocates a new output buffer, inserts it into the `z_stream`, calls `deflateInit` and creates a new `z_info` structure.

The `z_info` parameter is given the mode `uo` (unique out) which asserts that this is the only reference to `z_info` in existence.

[4]The bodies of these utility predicates are more verbose than they should be, because some parts of the uniqueness system in Mercury are incomplete and require the programmer to make extra calls to generate the right uniqueness information. In particular, embedding unique structures inside unique structures does not yet work.

The effects of the impure goals are to make sure each goal is executed even if it binds no variables (such as the call to `deflateInit`), and to prevent the compiler from re-ordering the calls. It is worth noting that the necessity to place `impure` at the start of each impure goal makes it quite difficult for readers to miss the fact that impure operations are taking place.

The code to implement compression is no more complex.

```
:- pragma promise_pure(compress/3).
compress(S, ZInfo0, ZInfo) :-
    z_info_get_z_stream(ZInfo0, ZStream),
    z_info_get_buffer(ZInfo0, Buffer),
    impure z_stream_set_input_string(ZStream, S),
    impure _ResDeflate = deflate(ZStream, 0),
    impure make_z_info(ZStream, Buffer, ZInfo).
```

Here impurity ensures that we set the input string before we call `deflate`.

The modes for the `z_info` parameter in compress are `di` (destructive input) which says that the input parameter is destroyed after the call (if it is the last reference, it is gone forever) and uo (meaning the same as in `init_compression`). Since the input parameter is destroyed and the output is unique, the implementation is free to re-use the input parameter as the output. This is how Mercury models destructive update in a semantically sound fashion.

Note that like other error checks, we don't check for buffer overflow in this sample code.

Finally we implement the end of compression, where we retrieve the entire compressed string.

```
:- pragma promise_pure(finish_compression/2).
finish_compression(C, ZInfo0) :-
    z_info_get_z_stream(ZInfo0, ZStr),
    z_info_get_buffer(ZInfo0, Buffer0),
    FlushParam = 4,
    impure _ResDeflate = deflate(ZStr, FlushParam),
    impure _ResDeflateEnd = deflateend(ZStr),
    semipure z_stream_get_total_out(ZStr,
        TotalOut),
    semipure z_stream_get_next_out(ZStr,
        NextOut),
    split_buffer(Buffer0, TotalOut, NextOut, C, _).
```

Note that in this code we use calls to semipure predicates. These goals can be re-ordered if required by the modes of their argument, but not past impure predicates. Since they directly follow an impure predicate, and data dependency says that the call to `split_buffer` needs their results, the compiler has no need to reorder these goals.

This predicate doesn't return `z_info`, it just destructively consumes one.

These three pure predicates thread the compression stream information into a pair of uniquely moded in, out pairs. This is a very common coding style in Mercury, there is special syntax (called DCG syntax) to simplify code using a pair of threaded arguments, and work on an extended DCG syntax that will support multiple threaded arguments is underway.

4.4 Separating Streams

The current interface is much more declarative than before, but Mercury programmers can still shoot themselves (and each other) in the foot by calling compress on a stream that was initialized for decompression or vice-versa. Such mistakes may result in memory corruption.

Since one of the fundamental Mercury beliefs is that if you can detect errors statically, you should (usually phrased as "static analysis is your friend"), we'd like to fix this problem. The easiest way to do this is to define two sorts of streams, compression streams and decompression streams, and change our pure predicates so that they will only generate and operate on the appropriate stream.

This leads us to add two more abstract types to the pure interface.

```
:- type compression_stream == z_info.
:- type decompression_stream == z_info.
```

Where the type `z_info` is used in the compression code above, we use `compression_stream` instead.

4.5 Enforcing Ordering

Our final problem is that programmers can execute operations in the wrong order, possibly causing memory corruption. Usually we use one of two solutions. Firstly, we can embed some sort of status indicator in the `z_info` data type, and check it before each operation to make sure the operation is valid. Secondly we can change the type of the threaded data structure at each operation to make sure the type system ensures that we can't call an operation in the wrong order.

However, if we look at the modes of the predicates again:

```
:- mode init_compression(uo) is det.
:- mode compress(in, di, uo) is det.
:- mode finish_compression(out, di) is det.
```

The only way the user of our pure library can obtain a `compression_stream` is using `init_compression`. If they call `finish_compression` the `compression_stream` is destroyed. And in the meantime you can call compress as many times as they like.

This is exactly the calling sequence we want to impose on users, and it will be statically checked by the uniqueness analysis system in Mercury.

Note that zlib does input validity checking, so we don't need to worry about users supplying compressed strings to the decompression predicate (they might for example supply the same compressed string twice. Of course our example code in this paper doesn't actually check for errors returned from zlib.

4.6 Summary

With a small amount of mostly straightforward work, we have turned an imperative library into a declarative one, while writing a very small amount of C code (most of which a more sophisticated C interface would generate automatically).

Apart from the lack of error checking, plenty of improvements are possible to this code. The compression level should be able to be set, and users could benefit from some control over the flushing of buffers. Mapping the return code integers into Mercury enumerations would make writing interfaces even easier, and encourage using good style to handle all possible return conditions as a Mercury switch.

It is easy to extend this interface to implement advanced features – for example it is easy to create a new compress predicate that returns partial results at each call to compress, instead of simply collecting the entire compressed string at once. This may work out nicely in a streaming application which chunks of input data dribble in while compressed output data dribbles out.

Now we have a pure interface, we can easily use it implement other pure interfaces. It could be used as a front end to our standard file handling operations to provide a compressed file reading and writing library. We can also provide simple interfaces to provide for common uses, for example:

```
compress(String, CmprString) :-
        init_compression(CStream),
        compress(String, CStream, CStream1),
        finish_compression(CmprString, CStream1).

decompress(String, CmprString) :-
        init_decompression(DStream),
        decompress(CmprString, DStream, DStream1),
        finish_decompression(String, DStream1).
```

5. RELATED WORK

There is a good miniature survey on systems to introduce imperative features into declarative languages given by Peyton Jones[8]. In this discussion we limit ourselves to those systems that share some similarities with the purity system of Mercury.

5.1 Effect Systems

Effect systems[6] have been used in non-declarative programming languages such as ML to isolate different parts of the program that can perform side effects. These effects share some similarities to the impurity annotations we place upon Mercury predicates and functions.

An effect annotation can describe whether a procedure has no side effects (equivalent to our pure), reads storage (similar to our semipure, although less general), writes storage (a specific case of impure) or allocates (which can be any level of purity in our system).

This effects system is aimed primarily at controlling side-effects that may inhibit optimizations such as parallelization, and doesn't seem to be as general as the Mercury impurity declarations which can include situations such as impure predicates which change the control flow of the program.

The side effect system is also not intended to be used to build pure interfaces, rather to mark out regions of independent impure code. The Mercury foreign code interface includes thread-safe annotations to help foreign code achieve independence, however this is by no means a full solution to the problem of executing impure code in parallel.

5.2 Monads and unsafePerformIO

The closest analogue to the uniquely moded Mercury `io__state` system and purity declarations is the Haskell IO monad and `unsafePerformIO`.

When importing functions from C in Haskell, the programmer can optionally have them return the IO monad. This is similar to deciding that a C function is impure in Mercury and declaring it impure. (Actually Mercury programmers also have the choice of using a pair of `io__states` with C functions – the impurity system doesn't eliminate this style).

The IO monad encapsulates any stateful side-effects that the function may generate. IO propogates upward through the program, so we end up with the same problem the Mercury programmer has with upwards propogating impurity. To create an interface without the IO monad, the Haskell programmer uses unsafePerformIO – this achieves much the same effect as the Mercury `promise_pure` pragma. While this scheme looks almost equivalent to the Mercury scheme, there are a number of problems with it.

In a way, using IO to encapsulate any impure function is an overloading of the IO monad. Some impure functions need to be executed in a particular order, or else the system will fail horribly due to memory corruption. In a normal functional library, programmers can call IO operations without fear of crashing their system. Because IO is used to encapsulate any impurity, programmers need to distinguish between normal IO operations and special impure IO operations that must be called in exactly the right order. While we expect programmers writing such libraries to be reasonably competent, this is still something of a burden to impose upon them and subsequent maintainers of their code. This is particular a problem for middle layers of declarative interfaces, where the impure primitives may look like relatively benign IO operations, and the unsafePerformIO is too far away to raise warning alarms.

While writing code using Mercury purity declarations, the required impure prefixes on goals remind the programmer that there are impure operations going on, and ordering is critical. The Mercury system allows the programmer to separate the concerns about flagging impure code so that it is executed sequentially, and modelling it in a declarative manner.

The Mercury system also has a concept of semipure operations, which allows limited re-ordering of goals. Semipure is particulary useful in logic programming because it allows reversible code to be written.

```
:- semipure pred process_data(int, int, int, data).
:- mode process_data(in, in, out, in) is det.
:- mode process_data(out, in, in, in) is det.

... defined using different C code for each mode ...

:- semipure pred process_all(int, int, int, int, data).
:- mode process_all(in, in, in, out, in) is det.
:- mode process_all(out, in, in, in, in) is det.
process_all(A, B, C, ABC, Data) :-
        semipure process_data(A, B, AB, Data),
        semipure process_data(AB, C, ABC, Data).
```

A system without semipure predicates will not be able to reorder the two calls to `process_data` depending upon whether A or ABC is the output parameter.

One other problem with the unsafePerformIO based scheme (and probably the most technically compelling reason why Mercury has adopted impurity declarations instead) is that unique modes such as those are used in the Mercury `io__states` are difficult to use in conjunction with backtracking. Since unique modes are used to model unique values, when we backtrack the changes made to the unique value need to be undone. We call these backtrackable unique modes *mostly unique*. Restoring old mostly-unique values on backtracking requires Prolog-like trail and imposes a considerable performance cost on the entire program, as well as being difficult to use and difficult to mix with normal unique modes. However, even backtrackable unique modes have the limitation that they cannot model sequential operations over choice points. For an example such as `max_solution`, if we tried to model the impure updates using mostly unique modes, there would be no way for the compiler to associate the mostly unique values of `M0` in one branch of the disjunction with those in the other.

```
max_solution(Generator, Max) :-
        init_max(M0),
        (   Generator(X),
            set_max(X, M0, M1),
            fail
        ;
    get_max(Max, M0)
        ).
```

The compiler may decide to reorder the branches of the disjunction, or even prune some branches away if they are guaranteed to fail, and the modes cannot prevent this. So for impure code that interacts with backtracking, impurity declarations are a much simpler solution than overloading the modelling of IO.

6. CONCLUSIONS AND FURTHER WORK

Evaluation of the Mercury impurity system in conjunction with the foreign language interface has revealed that it is quite a useful addition to the language from a systems programming point of view.

One idea that this work raises that seems quite reasonable and useful is to assume by default that any predicates or functions defined using foreign code are impure. With this assumption the programmer would have to explicitly write a `promise_pure` declaration if the code happened to be pure. The current system contains an unstated assumption that foreign code is pure, but this seems to be an error prone assumption, and not one that programmers are forced to confront. Changing the default would force programmers to consider the issue. This approach is already taken with foreign code with regard to foreign code calling back into Mercury, but adding new assumptions about such code raises backwards compatibility issues.

Although this paper has only used a few techniques for eliminating impurity by modelling it in other ways, it seems that there is a range of such techniques that would be worthwhile cataloging and exploring further. Naming and formalizing such techniques might ease the programmers burden of proof when promising that impure code has been transformed into pure code.

A general system for assertions (`promises`) about Mercury predicates is being added to the language, so the `promise_pure` pragma may be worth integrating into this system. If we make foreign code impure by default, it seems likely that a `promise_semipure` will need to be added. For completeness it might be worth allowing an optional `pure` declaration on predicates and goals.

We also hope to automate some of the more monotonous tasks of generating the impure primitives. A tool similar to c2hs[4] would be well suited to the task. While it is impossible for such a tool to reason about the purity of C functions in general, we can at least make conservative assumptions and let the programmer make more sophisticated judgements by hand.

We would like to thank the Australian Research Council for their support.

7. REFERENCES

[1] The Mercury Team, Mercury 0.9.1. Available from `<http://www.cs.mu.oz.au/mercury/>`, Feb. 2000.

[2] R. Becket. Mercury tutorial, 1999. Available from `<http://www.cs.mu.oz.au/research/mercury/tutorial>`.

[3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18:807–820, 1988.

[4] M. M. T. Chakravarty. C - Haskell, or yet another interfacing tool. In *Implementation of Functional Languages, 11th International Workshop Functional Programming*, 1999. To appear in a collection of selected paper of the workshop.

[5] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, and C. Speirs. The Mercury language reference manual. Available from `<http://www.cs.mu.oz.au/mercury/>`, 2000.

[6] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects, 1991.

[7] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, pages 202–219, Eindhoven, The Netherlands, June 1991.

[8] S. L. Peyton Jones and P. L. Wadler. Imperative functional programming. Bericht Nr. 92-19, RWTH Aachen, Fachgruppe Informatik, Aachen, DE, 1992. Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages.

[9] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.