Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

Markus Mohnen
Lehrstuhl für Informatik II, Aachen University of Technology
Ahornstraße 55, 52056 Aachen, Germany
email: `markusm@zeus.informatik.rwth-aachen.de`

October 6, 1995

Abstract

In this paper we will show how to embed functional concepts in an imperative host language in a *natural way*. This is achieved by interpreting functional programs as *specifications* of imperative programs. We will introduce a refined *notion of laziness*, which allows us to interpret lazy computations as special constructors, which encapsulate the suspended computation. Even higher-order concepts can be represented efficiently, if we restrict the possibility to use partial applications. Essentially, we disallow the use of sequences of partial applications as hidden data structures. By considering a setting in which the unit of translation is a *module*, this approach is the key to an *integration* of the functional programming paradigm and the imperative programming paradigm on a *module level*. Functional components of the resulting *hybrid* module architectures *may use* and *may be used by* imperative components without need for special *inter-language interfaces*. Beside the 'best of both worlds', this concept offers the possibility to *migrate* 'module by module' from purely imperative systems to hybrid systems and to do *rapid prototyping* using functional modules. Experimental results with a prototype version of a translator will show that we can compete with traditional functional languages like `Haskell`.

1 Introduction

Implementations of functional languages are usually based on *abstract machines*, which are implemented by *assembler macros* [FH88, PL92, Thi94, Loo90]. In order to reuse existing parts of compiler technology, like run-time systems and low-level optimisations, and to acquire portability, often an imperative language like `C` is used as 'portable assembler language' [Pey93, SG91, LH92].

The structure of the original functional program is lost during this compilation. This is due to the fact that an *additional run-time environment*, which maintains tasks like *data* *structure representation*, *higher-order features*, and simulation of *lazy evaluation*, is introduced by this process. The run-time system of the host language is merely used for function activation and parameter passing. Usually, the type system of the target language is completely ignored, the intrinsic possibilities for data structures or simulation of functional parameters remain unused. In [Pey93] it is argued that "*The mis-match between C and a non strict higher-order language is too great*" to map functional concepts directly to imperative pendant.

In this paper, we want to demonstrate that this statement is not true. We are going to use well-known techniques from the graph-reduction approach *without* an artificial run-time system, which would be introduced by the the simulation of an abstract machine.

We will show how to embed functional concepts in a *natural way* in the imperative host language. The underlying idea is to interpret functional programs as *specifications* of imperative programs. The difference between this approach and the one taken in [Pey93, SG91, LH92] is that the translation of specifications is *transparent* to the programmer, whereas the usual compilation of functional programs is a black-box. Functional specifications can be seen as a more compact syntactical variant of imperative programs.

We will map each functional type and each function to an equivalent imperative type, or imperative procedure, respectively. In this context, only higher-order expressions will cause a conflict between the aims 'natural embedding' and 'complete implementation of concepts' (see Section 5). Therefore, we will restrict the possibility to use partial applications. This restriction disallows the use of sequences of partial applications as (hidden) data structures, which can also be expressed using proper data structures. However, the use of partial applications as functional parameter for functions like *`map`* or *`filter`* is still possible.

Since the standard evaluation strategy of imperative languages is call-by-value, we treat the functional specifications also as eager specifications. In Chapter 6 we will discuss a possibility for representing lazy evaluation. Therefore, we have to define a new notion of laziness. Usually, the whole program is either interpreted either in a lazy way or in an eager way. We suggest a refinement of this approach, where 'lazy' or 'eager' are properties of functions. This allows us to interpret lazy computations as special constructors, which are eagerly evaluated Of course, the traditional approach can be simulated by tagging all functions as lazy.

But instead of translating single types or functions, we will consider a scenario where the unit of translation is a *module*, i.e. a collection of type and function definition. The resulting modules in `C` (or `Modula-2`, or `Ada`, or an other imperative language) will again be units of compilation for the imperative compiler. Now it is possible to *integrate* the functional paradigm and the imperative programming paradigm on a *module level*. This means that we can design *hybrid architectures*, where some modules may be programmed using a functional specification and others in the imperative language. Functional components of such an hybrid architecture *may use* and *may be used by* imperative components. Of course, in order to do so, a translator for the functional specifications must be able to understand the imperative interfaces, e.g. to extract type information from `Modula-2` definition modules or `C` header files. On the other hand, the interfaces of functional specifications are mapped to corresponding imperative interfaces in a way which is visible for the designer of the architecture. Therefore, purely imperative components can in turn use functional parts, by using the generated imperative versions of the interface. Since we neither have an additional run-time system for the functionally specified parts, nor hide the way the specifications are translated to imperative programs, there is no need for special *inter-language interfaces* in order to exchange data between the two worlds. Algebraic data structures which were generated using constructors can be used by imperative procedures, including suspended computations due to lazy evaluation. Even higher-order functions like `map` or `filter` can be incorporated from imperative components.

The benefits of such an alliance are:

from the functional world: the easy use of (possible infinite) *data structures*, including the possibility to define functions via structural recursion; *higher-order features* and *lazy evaluation* which reveal advanced possibilities for decomposing problems and 'gluing' existing components together [Hug90]; *referential transparency relative* to the imperative context.

Of course, (unrestricted) referential transparency can not be maintained. However, we can still be sure that functionally specified components are referentially transparent *relative* to the imperative parts used by them.

from the imperative world: easy *input and output*; access to *operating and window systems*; *in-place update* of data structures and arrays, or, more general, direct storage management for memory sensitive parts of a program; the notion of *state*.

Briefly, everything which is not natural for functional programs, and needed (only) for time and/or memory sensitive components.

from the integration: the possibility to *migrate* 'module by module' from purely imperative systems to hybrid systems; *rapid prototyping* using functional modules.

Since the creation of software 'from scratch' is the exception in software industry, functional languages have little access to areas of software development where only maintenance of existing software is performed. Therefore, a 'soft' migration may be a chance to increase the acceptance of the functional programming paradigm.

Functionally specified prototypes are known to be an efficient and reliable way. The resulting programs are more concise and easier to understand than corresponding imperatively designed prototypes [HJ94]. However, experience shows (90/10 rule) that the larger part of a software system is not object to strict memory and/or speed limitations. Hence, the prototype character of most of the functional components will vanish, and in the end only a small amount will have to be rewritten imperatively.

Moreover, this approach has the advantage that we can spare the time for implementing optimisations, which can be done be the imperative compiler. Candidates are for instance: unnecessary local variables, which are introduced by the translation by duplicating parameters (see Figure 3, variable `L`) or used only once, or tail recursive calls. Furthermore, very short procedures, like the procedures associated with selectors, e.g. `ConsSel1`, can be handled as macros ('inline functions') instead of being called normally. With our approach, we only have to consider those optimisations which can not be detected by an optimising imperative compiler, since they are connected with the functional structure of the underlying program.

It is not the objective of this paper to formally describe a translation mapping to a fixed imperative language. Instead, we will describe a general approach, resulting in a family of functional specifications for a variety of imperative languages. During the course of the paper, we will describe concepts of functional languages, discuss alternative possibilities for their imperative representation, and give requirements to be met by the target language for each concept.

We finish this introduction with a brief outline of the rest of the paper. In the two following sections, we will describe how to translate first-order, monomorphic algebraic data types and function definitions. Polymorphic and higher-order extensions are introduced in Section 4 and Section 5. After that, we will demonstrate how lazy evaluation can be incorporated. The paper concludes with a report on the state of an implementation and some experimental results.

2 Data Structures

In this section we will describe how to handle data structures based on *constructors*, also referred to as *algebraic data* types [PL92], or *sum of product* types. For the moment, we suspend *polymorphism* (Section 4) and *functional* entries (Section 5) from the discussion. Consider the (classical) functional way of defining a type of lists over integers:

```
datatype ListOfInt ::= Nil | Cons(int,ListOfInt)
```

Such a definition introduces four kinds of objects:

1. the *type identifier* (in this case `ListOfInt`),
2. the *constructors* (`Nil` and `Cons`),
3. for each constructor an associated *constructor test* (`isNil`, `isCons`), which tests whether a given object of type `ListOfInt` has this top constructor, and
4. for each constructor and each argument position of this constructor, a *selector* (`ConsSel1` and `ConsSel2`) which retrieves the corresponding arguments.

Note that constructor tests and selectors are often used indirectly by pattern matching. Each constructor appearing in a pattern on the left hand side of a function definition is simulated through corresponding tests and selectors.

2.1 Finding an Imperative Type

In denotational semantics, we represent constructor based types by sets which are the *sum* (i.e. the distinguishable union) of sets associated with the constructors, which are in turn the *product* of sets for the argument types of the constructors. This is also the key idea for the construction of corresponding imperative types. Like in the well-known graph reduction

Modula-2	C

<pre>TYPE ListOfInt = POINTER TO ListOfIntN;
TYPE ListOfIntNTags = (tagNil,tagCons);
TYPE ListOfIntN =
     RECORD
      CASE Tag : ListOfIntNTags OF
      tagNil: |
      tagCons:
               arg1 : INTEGER;
               arg2 : ListOfInt;
      END;
     END;</pre>
	<pre>typedef struct _ListOfInt *ListOfInt;
	struct _ListOfInt {
	    enum {tagNil, tagCons} tag;
	    union {
	        struct { } Nil;
	        struct {
	            int arg1;
	            ListOfInt arg2;
	        } Cons;
	    } data;
	};</pre>

Figure 1: `Modula-2` and `C` type definitions for the `ListOfInt` definition

approach, we represent the constructor terms of the denotational semantics as connected cells in a heap component.
Consequently, our first conditions on properties of the imperative target language are:

pointer types and allocation facilities *(C1)*
definability of (maybe recursive) sum-of-product types *(C2)*

In `Modula-2` or `Pascal` this concept is covered by 'variant records'. In `C`, we can create product types using `struct` and union types using `union`. In order to make the union distinguishable, we have to add a tag field explicitly. In Figure 1, corresponding `Modula-2` and `C` type definitions for the *`ListOfInt`* definition are listed. In the sequel, we will give only `C` fragments.

2.2 Constructors, Constructor Tests, and Selectors

For each constructor of a given type, we have to define a function which creates the corresponding data cell in the heap. This can easily be done by using the allocation facilities for the type (which exist according to condition (C1)). The newly allocated cell is then initialised with the appropriate values.

The associated constructor tests and selector functions are straightforward tests of the tag fields, and selection of components of the appropriate product, respectively. See Figure 2 for the `C` code obtained from the constructor *`Cons`* by this translation.

For simplicity, we have used the standard library function `malloc` in the example, although in a concrete implementation a function should be used which performs a defined exception, if not enough memory is available. Possible exception handling could be the abortion of the program, or the invocation of a garbage collector.

```
ListOfInt Cons(int arg1, ListOfInt arg2)
{
    ListOfInt result;
    result=(ListOfInt)malloc(sizeof(struct _ListOfInt));
    result->tag=tagCons;
    result->data.Cons.arg1=arg1;
    result->data.Cons.arg2=arg2;
    return result;
}
int isCons(ListOfInt p) { return (p->tag==tagCons); }
int ConsSel1(ListOfInt p) { return p->data.Cons.arg1; }
ListOfInt ConsSel2(ListOfInt p) { return p->data.Cons.arg2; }
```

Figure 2: The `C` example for all functions associated with the constructor *`Cons`*

3 Function Definitions

Each monomorphic, first-order function definition of the functional specification is mapped to a corresponding imperative representation. Firstly, we will consider only plain function definitions. Later, the possibility for the definition of functions via pattern matching will be added.

3.1 Plain Function Definitions

Functional expressions may contain all constructs which may appear in expressions of the imperative language. This is necessary to provide access to basic data types and basic functions[1]. Of course, no expression with implicit assignments are permitted, e.g. `C`'s `++` and friends are not allowed.

A direct translation into imperative expressions can be performed by mapping function applications on the specification level to procedure calls on the imperative level. The value is returned as result of the function. The only problem which may occur is due to the deterministic choice, which may not be available in imperative expressions. In this case, a 'branch–lifting' has to be applied, which moves the branches to the outermost level of the expression. Then they can be implemented using the branch available on the statement level.

Consider the following sample program, a function for computing the length of a list:

```
function length : ListOfInt -> int ;
         length (L) = if (isNil(L)) then 0
                          else 1+length(ConsSel2(L));
```

A corresponding `C` definition can be found in Figure 3. Note that the auxiliary variable `L` is not necessary at this point, but the introduction of patterns can be done easier this way.

[1] Algebraically, we assume that the functional specification and the imperative host language share a common signature.

```
int length(ListOfInt arg1)
{
  ListOfInt L = arg1;
  return isNil(L)
    ? 0
    : 1+length(ConsSel2(L));
}
```

Figure 3: `C` version of *length*

3.2 Function Definitions with Pattern Matching

The translation of patterns requires two steps: firstly, we have to test a condition which checks whether the actual parameters match the patterns, and secondly the variable bindings induced by the pattern have to be established. The condition is just a conjunction of constructor tests for the constructors at the appropriate positions of the pattern. The access to variables can be done in two ways. The first possibility is to replace each variable by the appropriate combination of selectors. The obvious disadvantage is that, if a variable is used more than once, the selector sequence has to be evaluated again each time the variable is used. A better approach is to introduce new variables and to assign the value of the selector sequence once. Now, we simply can use the translation scheme developed for plain function definitions to translate each right hand side.

If the target language supports the declaration of blocks containing new local variables inside function definitions (like `C`), the translation of a function definition with patterns can be done by enclosing each plain translation of a right hand side of a pattern in such a block where the appropriate variables are defined and assigned. The complete translation is composed as a sequence of tests for the condition of a pattern and the corresponding block (see Figure 4). If the patterns do not cover all cases, we have to add a 'default behaviour' for the case that no pattern succeeded. This will normally be a program abortion. However, in our examples this is not necessary.

If the host language does not support such variable declarations, we can wrap each right hand side into a new procedure, where the local variables are declared and assigned. These procedures can be either inside the definition of the function under consideration, if supported (like in `Modula-2`), or outside.

3.3 Local Function Definitions

By using a ***where*** construct (like in Miranda [Tur90]), we allow the definition of local functions inside functions. If the target language supports local definitions of procedures, we could use this facility directly. But this approach would be inadequate in the context of higher-order features. In Section 5 we are going to describe in detail how to implement parameters and results of functional type through procedural parameters of the imperative language. However, imperative languages disallow the use of locally defined functions as procedural parameters.

Therefore, we have to map locally defined functions to globally defined procedures, which is essentially a technique called λ-*lifting* in the functional world (see [Joh85, Hug83]). But instead of transporting variable bindings by introducing additional parameters, we use a set of global variables. These variables are set by the outer function and copied by the inner function to correspond local variables. They contain the values of local variables of the outer function. Of course, the global variables must be set immediately before the call, in order to avoid an overwriting due to recursion. If the target language does not support a *sequence* of expression level (e.g. `C`'s `,` operator) we have to flatten the expression in order to preserve the evaluation order.

This approach has the advantage, that the type of the newly created additional global procedure corresponds to the original local function, which makes it possible to use this procedure as parameter.

The special case that the locally defined function takes no parameters, i.e. is a constant, can simply be translated by creating one new local variable in the procedure.

The following example program defines a *quicksort* function which uses locally defined functions *filtgt* and *filtle* for creating the lists of those elements which are less than or equal to (resp. greater than) the current pivot.

```
function qs : ListOfInt -> ListOfInt ;
  qs(Nil) = Nil;
  qs(Cons(a,L)) = append(qs(filtle(L)),Cons(a,qs(filtgt(L))));
  where filtle : ListOfInt -> ListOfInt ;
                 filtle(Nil) = Nil;
                 filtle(Cons(b,L)) = if (b<=a) Cons(b,LN) else LN;
                 where LN : ListOfInt;
                        LN = filtle(L);
        filtgt (analogous)
```

The corresponding `C` program (see Figure 4) demonstrates both the general approach and the special case, where the locally defined function has arity zero.

4 Polymorphic Types and Functions

Until now, we have only considered monomorphic data structures and functions. In order to extent the approach to polymorphic data structures and functions, we will represent polymorphic components by pointers. Consequently, instances of polymorphic components which are basic values are stored in a boxed way. The condition for the imperative language is:

universal pointer type, convertible into all pointer types *(C3)*

For this purpose, `ANSI C` defines `(void *)`, and the `Modula-2` module `SYSTEM` contains the type `ADDRESS`.

For each polymorphic construct a single imperative counterpart is defined. Instantiation is achieved through appropriate type conversions. Consider the following polymorphic constructs:

```
int qs_filtle_a;
ListOfInt qs_filtle(ListOfInt arg1)
{ if (isNil(arg1))
    return Nil();
  if (isCons(arg1)) {
    int b = ConsSel1(arg1);
    int a = qs_filtle_a;
    ListOfInt L = ConsSel2(arg1);
    ListOfInt LN = qs_filtle(L);
    return (b<=a)?Cons(b,LN):LN;
  }
}
int qs_filtgt_a;
ListOfInt qs_filtgt(ListOfInt arg1)
{ ... }
```

```
ListOfInt qs(ListOfInt arg1)
{
  if (isNil(arg1))
    return Nil();
  if (isCons(arg1)) {
    int a = ConsSel1(arg1);
    ListOfInt L = ConsSel2(arg1);
    return
      append(
       qs((qs_filtle_a=a,
           qs_filtle(L))),
       Cons(a,
            qs((qs_filtgt_a=a,
                qs_filtgt(L)))));
  }
}
```

Figure 4: `C` fragment for *qs* with local definitions and pattern matching

```
datatype List(a) ::= Nil | Cons(a,List(a));
function split : List(a) -> List(List(a));
         split(Nil) = Nil;
         split(Cons(x,L)) = Cons(Cons(x,Nil),split(L));
```

The imperative representation of a polymorphic type is the monomorphic representation of the type where all type parameters are instantiated by the universal pointer type. All polymorphic and monomorphic uses of this type in a type or function declaration are replaced by this (minimal) instance. Calls to instances of polymorphic functions must be converted appropriately, and if a basic value is instantiated, the values must be 'boxed'. A pointer to the corresponding heap cell is then used as instance. On the other hand, if a polymorphic result appears in a context where a basic value is expected, it must be 'unboxed', i.e. the pointer to must be dereferenced.

For the above example, a `C` program looks like in Figure 5. Note that both parameter type *List(a)* and result type *List(List(a))* of *split* are mapped to the imperative type `List`. This is safe as long as *split* is used in a purely functional context, since there the original types are checked. However, if an imperative module uses the result of `split` in a context, where a 'list of lists' is not valid, the type checker of the imperative compiler is not able to detect this error, since the only information it receives is that `split` returns a list.

5 Higher-Order Features

Higher-order functional languages are characterised by the ability to pass functions as parameters to functions and return functions as result of functions. Moreover, *partial applica-*

```
typedef struct _List *List;
struct _List {
    enum {tagNil, tagCons} tag;
    union { struct { } Nil;
              struct { void *arg1;
                         List arg2;
        } Cons;
    } data;
};
List Nil(void) { ... }
int isNil(List p)
{return(p->tag==tagNil);}
...
```

```
List split (List arg1)
{ if (isNil(arg1)) return Nil();
  if (isCons(arg1)) {
     void *x = ConsSel1(arg1);
     List L  = ConsSel2(arg1);
     return
       Cons((void *)Cons(x,Nil()),
            split(L));
  }
}
```

Figure 5: Polymorphic example in `C`

tions[2]can be used to create new functions *at runtime*. These functions are built from existing functions by the transport of variables bindings. Since such a newly created function can also be returned as result of a function, it is possible that the binding of parameter variables will exist longer than the corresponding function call. As an extremal example, consider the following program, which uses partial applications of *`empty`* and *`update`* to represent an array of type *`int`* with index type *`int`*.

```
empty : int -> int
empty(i) = 0
update : (int -> int) x int x int x int -> int
update(array,newi,newval,i) = if (i==newi) then newval
                              else array(i);
upto : int -> (int -> int)
upto(n) = if (n==0) then empty
          else update(upto(n-1),n,n);
```

The call *`upto(n)`* creates an array, where the i-th entry is i, if $0 \le i \le n$, and 0 otherwise.

Until an entry is requested, for instance by *`(upto(n))(j)`*, all n incarnations of *`update`* and the appropriate bindings of *`array`*, *`newi`*, and *`newval`* have to be remembered.

Therefore, implementations of functional languages must represent all arguments of functional type by *closures* which are usually allocated on a *heap*. A closures are a tuple of a function and a sequence of already given parameters for this function. Implicitly, the closures contain information on how many arguments are *needed* to evaluate the function, how many arguments are *present*, and the *values* of those arguments which are present. For instance, the evaluation of *`upto(4)`* creates five closures cells (see Figure 6).

[2]Alternatively, functions can be defined by λ-*abstractions*, which can be translated to partial applications by a transformation known as *lambda lifting* (see [Joh85, Hug83]).

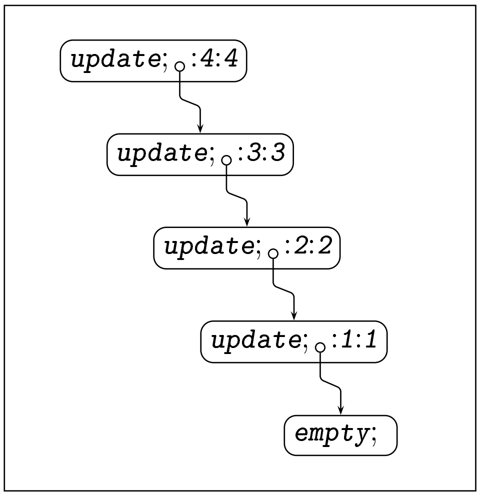

Figure 6: The heap cells created by `upto(4)`

On the other hand, even if an imperative languages allows

parameters and results of procedural type *(C4)*

the dynamic creation of new functions in the way described above is not permitted. The underlying philosophy is to represent procedural parameters and results only by their code address. This is also the reason why languages with nested procedure definitions do not allow locally defined procedures as parameters.

In order to embed functional higher-order concepts in a natural way, we have to follow the imperative restrictions. The easiest way would be to simply disallow partial applications. Since this would be a severe restriction, we suggest a compromise between those extremal approaches. The underlying observation is that there are partial applications where at most one *active incarnation* exists during execution. The suggested approach is to disallow all other cases, e.g. situations like in the above example. From the theoretical point of view, this is of course no real restriction, since in the presence of constructors already first-order programs are universal. But also from the practical point of view, this is not a severe restriction. Structures like the connected cells of *update*'s are essentially a hidden data structure, which can equivalently be programmed by using algebraic data types. E.g. partial applications as arguments for *map* or *fold* are still allowed. Maybe it is even possible to develop a method which transforms the (some of the) unwanted partial applications to proper data structures.

In [Moh95] we have described a way how to determine an approximation of these properties by abstract interpretation. The techniques presented there can easily be incorporated in a translator for functional specifications.

Consider the following higher-order version of the *quicksort* function, which can be defined by

```
qsho : List(int) -> List(int)
qsho(Nil)=Nil;
qsho(Cons(a,L)) = append(qsho(filter(<a,L)),
                         Cons(a,qsho(filter(>=a,L))));
filter : (int->bool) x List(int) -> List(int)
filter(p,Nil) = Nil;
filter(p,Cons(x,L)) = if (p(x)) Cons(x,filter(p,L))
                      else filter(p,L);
```

The partial applications `<a` and `>=a` are used as functional arguments for *filter*. Since the functional argument of *filter*, in contrast to *update*, is *not inherited*, we can be sure that `<a` and `>=a` are only used until the surrounding calls to *filter* terminate. Moreover, we can see that during the execution of *filter*, there can not be a recursive call to *qsho*. Therefore, there will always be only a single active closure for `<a` resp. `>=a`.

In these cases, we can transform the program by replacing the partial applications by locally defined functions:

```
qsho : List(int) -> List(int)
qsho(Nil)=Nil;
qsho(Cons(a,L)) = append(qsho(filter(qsho1,L)),
                         Cons(a,qsho(filter(qsho2,L))));
where qsho1 : int -> bool;
      qsho1(x) = x<a;
      qsho2 : int -> bool;
      qsho2(x) = x>=a;
```

If we map the functional type to the appropriate procedural type, we can translate this program essentially in the way described in Section 3.

6 Lazy Evaluation

Until now, we have only considered eager evaluation, since this is the natural strategy for imperative programs. Now, we extent the approach to support lazy evaluation, i.e. delayed evaluation of arguments on demand.

The implementation of lazy evaluation is usually performed by representing arguments which are not yet evaluated through *closures*. These closures essentially contain the *function* which is to be called if the evaluation to head-normal form (hnf) is requested, and the *arguments* for this particular function call.

Similar to constructors, closures are implemented by heap cells. Therefore, our approach is to implement unevaluated components of a data structure by special *Lazy* constructors. Since these constructors are not visible to the programmer, there are no associated constructor tests or selectors. Obviously, in order to handle lazy evaluation of components of basic type, we have to represent basic values in a boxed way.

Usually, the whole program is either interpreted either in a lazy way or in an eager way. We suggest a refinement of this approach, were 'lazy' or 'eager' are properties of functions.

A function tagged as 'lazy' is not evaluated if it is called, but instead a `Lazy` constructor is create, which represents the suspended computations. The actual computation is triggered, if a constructor test is applied to this `Lazy` constructor.

Note that we take a different approach than *strictness analysis* [Myc80], where *argument positions* of a function are tagged. For the sequel, we assume that all functions which are to be evaluated in a lazy way, are tagged.

In general, we now have two alternatives for obtaining such annotations: on one hand, we can assume lazy evaluation as default strategy, and use strictness analysis to detect 'eager situations.' In these cases, the translator could insert statements to trigger the actual computation, whereas all other cases only create closures for a delayed evaluation. On the other hand, we can leave the annotations to the programmer, keeping eager evaluation as default. To our opinion, the latter is more effective, since the programmer (should) exactly know which functions are to be evaluated in a delayed way. Moreover, call-by-value specifications simply fit more natural in an imperative context. Maybe it is possible to automatically detect functions which must be evaluated lazily. The simplest examples of such functions are those which contain an obvious not terminating recursion, like `fiblist` in the following example.

Each function tagged as lazy is realised in the imperative language by two procedures: the first is used in all expressions, where the original function occurred, and creates a `Lazy` constructor. The second performs the actual evaluation associated with the right hand side, and is never directly called. Only its address is stored in the `Lazy` constructor, which is created by a call to the first procedure.

The constructor tests trigger the evaluation, if they are applied to a such a `Lazy` constructor. The result of the evaluation to hnf is then used to overwrite the `Lazy` constructor. Thereby we can ensure that each suspended evaluation is actually evaluated only once. In order to be able to overwrite the `Lazy` constructor, we need the

ability to assign structures addressed by pointers *(C5)*

The next example defines a function `fib`, which computes the n-th Fibonacci number by selecting the n-th element form the (infinite) list of Fibonacci numbers. The sub list starting with the number $n1 + n2$ is generated by a call to the (lazy) function `fibl`, taking $n1$ and $n2$ as arguments.

```
function nth : int x List(a) -> a;
         nth(1,Cons(x,L)) = x;
         nth(n,Cons(x,L)) = nth(n-1,L);
lazy function fibl : int x int -> List(int);
         fibl(n1,n2) = Cons(n1+n2,fibl(n2,n1+n2));
function fib : int -> int;
         fib(n) = nth(n,fibl(0,1));
```

Now, the `C` fragment for the type `List` contains room for an additional constructor `Lazy`. Since we have to assign a fixed type to this, we need to pack the arguments of any lazy function into a single value. Therefore, the next condition for the target language is:

definability of sum types   *(C6)*

A pointer to such a value is then stored in the `Lazy` constructor represented as a universal pointer, together with a procedure which will do the actual evaluation. A function which evaluates a `Lazy` constructor to hnf is added, and calls to this function are inserted in the constructor tests for the normal constructors. The resulting (and final) `C` fragment for the type `List` is in Figure 7.

The procedures associated with the `fibl` function require an additional type in order to save the arguments of a call to `fibl` in a single value. The constructor test `isCons` in the procedure `nth` causes the actual evaluations in this example.

7 A Proposal for a Module System

In this section, we will give some prospects for a module system. Even in the absence of an explicit module concept of the imperative language, we can support modularisation. The conditions to be met by the imperative language are

separately compilable units   *(C7)*
hiding of entities from a unit   *(C8)*

In `C` the units of compilation are files, type declarations are automatically local to a file, and variables and functions can be made file-local by the prefix `static`.

A functional module defines types and functions, and contains an interface which makes parts of these definitions visible to other modules. Exporting an algebraic type (for instance `List`) automatically makes the type and the associated constructors, tests, and selectors visible to the outside. In order to support abstract types, the type must be exported in a special way which hides the implementation of the type, i.e. the constructors are encapsulated in the module. Of course, in order to export a function, all types which are needed for the function must also be contained in the interface.

During the translation, according imperative versions of the interface are created. In `C`, this interface is contained in a header file, which includes all visible parts. All components of the functional module which are not part of the interface are declared as *static* is the associated `C` file. In the case of abstract type only the actual type name, i.e. a pointer type, is contained in the header file. Therefore, no other module can access the internals of such a type.

8 Experimental Results

Currently we are implementing a prototype version of a translator to `C`. Although it was not the intent of this approach to concur with existing sophisticated compilers for functional language like the Glasgow Haskell Compiler (`ghc`), we have done some run-time experiments. The abstract machine which is the foundation of `ghc` is the Spineless Tagless G-Machine (STG) as described in [PL92]. The test programs can hardly be called benchmarks, but still the results are useful for a first impression.

```
typedef struct _List *List;
struct _List {
    enum {tagLazy, tagNil, tagCons} tag;
    union {
       struct { List (*eval)(void *); void *args; } Lazy;
       struct { } Nil;
       struct { void *arg1; List arg2; } Cons;
    } data;
};
List Lazy(List (*eval)(void *), void *args)
{  List result=(List)malloc(sizeof(struct _List));
   result->tag=tagLazy;
   result->data.Lazy.eval=eval; result->data.Lazy.args=args;
   return result;
}
void evalList (List p)
{  List help=p->data.Lazy.eval(p->data.Lazy.args);
   *p=*help;
}
List Nil(void) { ...}
int isNil(List p)
{ while (p->tag==tagLazy) evalList(p); return (p->tag==tagNil); }
...
void *nth (int arg1, List arg2)
{  if (n==1 && isCons(arg2)) {
      int  n =arg1; void *x=ConsSel1(arg2); List L=ConsSel2(arg2);
      return  x; }
   if (isCons(arg2)) {
      int  n =arg1; void *x=ConsSel1(arg2); List L =ConsSel2(arg2);
      return  nth(L); }
}
typedef struct {int arg1; int arg2; } _fibl_args;
List _fibl(_fibl_args *args)
{  int *help=malloc(sizeof(int));
   *help=args->arg1+args->arg2;
   return Cons((void *)help,fibl(args->arg2,*help))
}
List fibl(int arg1, int arg2)
{  _fibl_args *args = malloc(sizeof(_fibl_args));
   args->arg1=arg1; args->arg2=arg2;
   return Lazy((List (*)(void *))_fibl,(void *)args);
}
int fib(int arg1) { return *((int *)nth(arg1,fibl(0,1))) }
```

Figure 7: `C` procedures associated with the lazy example

qsort	100	200	300	400	500	600
ghc	0.06	0.27	0.64	1.16	2.30	4.93
C	0.11	0.51	1.24	2.25	3.66	5.35
C in pl.	0.06	0.31	0.82	1.47	2.43	3.54

natlist	10^4	10^5	10^6
ghc	0.21	3.05	†
C	0.04	0.39	9.30

Table 1: Experimental results: `ghc` vs. us

All measurements (see Table 1) were done on a Sun Sparcstation 10 with 32 MB RAM. The unit is seconds. We have done three experiments: *qsort([n,...,1])* for varying n with `Haskell` and `C`, the same with an in-place version of *append* for the `C` version, and *nth(n,natlist(1))* where *natlist(1)* creates the (potentially infinite) list of natural numbers. The *qsort* example contains eager computations and higher-order features, and the `natlist` program forces lazy evaluation. Of course, in order to minimise the effect of sophisticated optimisations of `ghc` (which we have not implemented yet) we disallowed `ghc` optimisation of the code. Moreover, to avoid delays due to garbage collection, we took a heap space of 16MB.

The results for *qsort* show that our approach can compete very well with `ghc`. Of course, the in-place version finally beats `ghc`. The *natlist* was a surprise; our representation of suspensions seems to be more efficient that the one `ghc` uses. For $n = 10^6$ we tried 32 MB heap, but still the `ghc` version aborted due to exhausted heap.

9 Conclusions

In this paper we have presented an alternative point of view of functional programs. Functional concepts can be translated to imperative programs in a natural way. We have demonstrated how to use well-known techniques from the graph-reduction approach without using an artificial run-time system, which would be introduced by the the simulation of an abstract machine.

Of course, the translation presented here can be refined. For instance, special kinds of recursion can be translated to loops; data structures with exactly one nullary constructor do not need extra heap cells for this constructor; et cetera.

We have given conditions for each concept we have considered which are to be met by the imperative host language in order to support this translation. For each concept, these conditions were:

data structures	*(C1)*	*pointer types and allocation facilities*
	(C2)	*definability of (maybe recursive) sum-of-product types*
polymorphism	*(C3)*	*universal pointer type, convertible into all pointer types*
higher-order features	*(C4)*	*parameters and results of procedural type*
lazy evaluation	*(C5)*	*ability to assign structures addressed by pointers*
	(C6)	*definability of sum types*
module system	*(C7)*	*separately compilable units*
	(C8)	*hiding of entities from a unit*

In order to embed the concept of higher-order functions, we had to restrict the possibility to use partial applications. This restriction disallows the use of sequences of partial applications as (hidden) data structures, which can also be expressed using proper data structures. However, the use of partial applications as functional parameters for functions like `map` or `filter` is still possible. It would be interesting to investigate, whether programs can be transformed automatically to meet this restriction.

Since the standard evaluation strategy of imperative languages is call-by-value, we also treated the functional specifications in an eager way. To allow lazy computations, we have defined a new notion of laziness, where 'lazy' or 'eager' are properties of functions. This allows us to interpret lazy computations as special constructors, which are eagerly evaluated. Of course, the traditional approach can be simulated by tagging all functions as lazy. Maybe it is possible to find conditions for functions which must be tagged as lazy, e.g. functions with a non terminating recursion.

With this approach we can interpret functional programs as specifications of imperative programs. By considering a setting where the unit of translation is a module, this approach is the key to an integration of the functional and the imperative programming paradigms on a module level. Without necessity for special inter-language interfaces, functional and imperative components of the resulting hybrid module architectures can interact and share data structures.

Beside the 'best of both worlds', this concept offers the possibility to module by module' from purely imperative systems to hybrid systems and to do rapid prototyping using functional modules.

Experimental results with a prototype version of a translator have shown that we can compete with compilers for functional languages like `Haskell`.

References

[FH88] A.J. Field and P.G. Harrison. *Functional Programming.* Addison–Wesley, 1988.

[HJ94] P. Hudak and M.P. Jones. Haskell vs. Ada vs. C++ vs. Awk. ...: An Experiment in Software Prototyping Productivity. Technical report, Yale University, Department of Computer Science, 1994.

[Hug83] R. J. M. Hughes. *The design and implementation of programming languages.* Dr. phil. thesis, Programming Research Group, Oxford, July 1983.

[Hug90] J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, chapter 2, pages 17—42. Addison–Wesley, 1990.

[Joh85] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In Jouannaud, editor, *Proceedings of the IFIP Conference on Functional Programming and Computer Architecture*, number 201 in LNCS, pages 190—205. Springer Verlag, 1985.

[LH92] K. Langendoen and P.H. Hartel. FCG: a Code Generator for Lazy Functional Languages. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, number 641 in LNCS. Springer, 1992.

[Loo90] R. Loogen. *Parallele Implementierung funktionaler Programmiersprachen (in German)*. Number 232 in Informatik Fachberichte. Springer, 1990.

[Moh95] M. Mohnen. Efficient closure utilisation by higher-order inheritance analysis. In A. Mycroft, editor, *Proceedings of the Second Static Analysis Symposium*, number 983 in LNCS. Springer Verlag, 1995.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the International Symposium on Programming*, number 83 in LNCS, pages 269–281, 1980.

[Pey93] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine, Version 2.5. Technical report, Department of Computing Science, University of Glasgow, 1993.

[PL92] S.L. Peyton Jones and D. Lester. *Implementing Functional Languages: A Tutorial.* International Series in Computer Science. Prentice Hall, 1992.

[SG91] W. Schulte and W. Grieskamp. Generating Efficient Portable Code for a Strict Applicative Language. In J. Darlington and R. Dietrich, editors, *Declarative Programming Sasbachwalden*, Workshops in Computing. Springer, 1991.

[Thi94] P. Thiemann. *Grundlagen der funktionalen Programmierung (in German).* B.G. Teubner, 1994.

[Tur90] D. Turner. An overview of Miranda. In D.A. Turner, editor, *Research Topics in Functional Programming*, chapter 1, pages 1—16. Addison–Wesley, 1990.