The JUMP-machine: a Generic Basis for the Integration of Declarative Paradigms

Manuel M.T. Chakravarty* Hendrik C.R. Lock†

May 25, 1994

Abstract

Implementation techniques for functional languages on the one hand and for logic languages on the hand differ considerably. This complicates the development of flexible and efficient abstract machines and thus good compilers for multi-paradigm languages.

The JUMP-machine is an abstract machine specifically designed for the efficient implementation of multi-paradigm languages. It consists of a minimal core realizing some basic operational abilities which is extended with components supporting different declarative features in an orthogonal way.

Results obtained with a compiler for the functional logic language Guarded TermML that is based on the JUMP-machine show that it is possible to amalgamate different paradigms in an efficient way, i.e., unused features do not give rise to substantial penalties in the execution of other orthogonal components.

1 Introduction

A crucial requirement for the application of declarative languages is the availability of efficient implementations. There exist already sophisticated techniques for the implementation of functional [PL92, Pey92] and of logic languages [War83, AK90, Roy90]. Unfortunatly, implementation techniques for different paradigms are often dissimilar and do not easily blend.

In this work we present an abstract machine model, called the JUMP-machine, which integrates different paradigms smoothly. It leads to an uniform and efficient implementation technique for multi-paradigm languages. The main difference between the JUMP-machine and other abstract machines amalgamating paradigms is twofold: first, the JUMP-machine efficiently implements a whole class of languages in an uniform way and is not specialized towards a specific language like [BCM89, Han91, MKLR90], and, second, it is designed to support the generation of efficient fully compiled code (contrasted to, e.g., byte code) in an easy and straightforward way. In contrast to this, most other abstract machines for functional logic languages are based on an extended WAM instruction set which is not very well suited for full compilation, as has been pointed out for example in [Roy90] and [Loc92].

A compiler for the functional logic language Guarded TermML (GTML) [Loc88, Cha91a] was implemented using the JUMP-machine as an abstract machine model. This implementation shows that it is possible to generate efficient machine code for functional logic languages without incurring a severe penalty through amalgamating paradigms, i.e., purely functional and purely logic programs execute as efficiently as they would in a single-paradigm implementation. Especially, deterministic computations within functional logic programs are carried out as efficiently as in an optimizing implementation of a purely functional language.

In this paper we will show how the JUMP-machine integrates the execution of a variety of functional and/or logic languages. Therefore, we will first describe more precisely the class of languages covered in this paper in terms of the operational features needed to implement them. Then, we introduce the mechanisms used in the JUMP-machine to realize these operational features and show how they are

*TU Berlin, Germany, chak@cs.tu-berlin.de
†IBM Heidelberg, Germany, lock@vnet.ibm.com

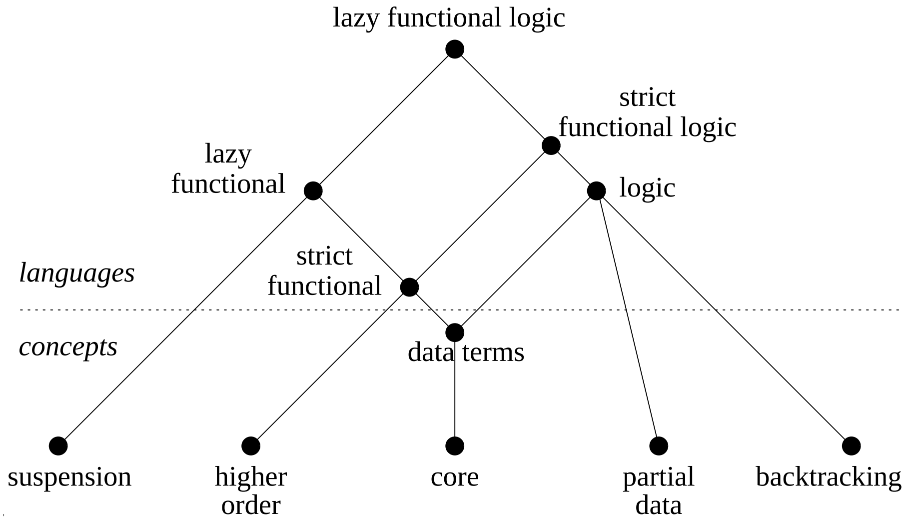

Figure 1: Functional and logic operational features supported by the JUMP-machine.

combined in an orthogonal way. We also sketch how other paradigms may be integrated in the JUMP-machine. Finally, we introduce the functional logic language Guarded TermML and give some results obtained with an implementation of this language based on the JUMP-machine.

2 The JUMP-machine

Abstract machines for the implementation of multi-paradigm languages are normally derived from an existing machine of one of the paradigms that are to be amalgamated. For example, abstract machines for the implementation of functional logic languages are commonly derived either from one of the standard machines developed for the translation of functional languages, e.g., the G-machine or the CAM (for examples see [Loo91, MKLR90, Müc92]), or from the Warren Abstract Machine, WAM for short (for examples see [BCM89, Han91]) which is used to implement logic languages, especially Prolog. Within these approaches, the derivation process is mainly extensional.

In contrast to this mere extension of an existing machine we tried to systematically categorize the features and techniques incorporated by different machines and to identify the common parts and the extensions used to support a distinct declarative feature or paradigm. After this analysis, starting from a minimal common core we added support for different features of declarative languages in the form of extensions which can be combined in an orthogonal way to implement different multi-paradigm languages.

The JUMP-machine is able to efficiently implement nearly all of the functional logic languages proposed in the last years in an uniform way. In order to strengthen this claim we give a classification of functional logic languages in terms of operational features supported by the JUMP-machine. This can be visualized in form of the triangular diagram shown in Figure 1. It is seperated in two parts. The part below the dashed line contains operational features supported by the JUMP-machine. The upper part represents the design space of functional logic languages which is covered by this features. In the center of the lower part is the operational core combining the basic features necessary to implement a first-order applicative language on basic unstructured data—this core forms the starting point for all following extensions. To its left are the operational extensions necessary for functional languages and to its right those for logic languages. Straight above the core are nested term structures which are common to functional and logic programming. The main functional features are higher-order functions and suspensions—the latter are used to implement laziness. The logic features are partially defined data and backtracking. Together with the nested term structures located above the core comes pattern-matching and in the context of partially defined data structures also unification.

Using some of the operational features (or extensions) in the lower part strict and lazy functional languages like SML and Haskell [H+92] can be found half-way up on the left side of the triangle. On the right side are logic languages like Prolog [SS86]. Towards the top there are strict functional logic languages and lazy first-order narrowing languages (not explicitly shown in the diagram). Finally, the upper tip of the triangle represents lazy functional logic languages like Babel [MR99, MR92] and GTML [Loc88, Cha91a] where all of the operational features are employed. All of these languages lie in the scope of the JUMP-machine as we will see in the remainder of this paper. Note that not all of the languages that can be implemented with the operational features present in the diagram are explicitly marked in the figure or explicitly mentioned—an example are functional languages with logic variables (but without backtracking) [DGK93].

A thorough discussion of existing machines and their classification in terms of the presented lattice of operational features can be found in [Loc92]. Detailed descriptions of the JUMP-machine and the translation techniques employed for different functional logic languages are given in [Loc92, Cha91b, CL91].

2.1 Components of the machine

We will now describe in more detail the components of the JUMP-machine, their interaction, and which operational features they support. This will be done in an extensional way, i.e., starting with a simple *core machine* that is common to most abstract machines for the implementation of declarative languages, we will extend this core in order to support the different operational features that form the basis of the presented diagram like data terms, higher order functions, and backtracking. Therefore, we will concentrate on the extensions necessary to implement functional logic languages. In [Köh92, CK92] it is shown how a constraint-based, concurrent language can be implemented on the basis of the JUMP-machine.

2.1.1 The core and data terms

The core machine used to implement the operational core has a conventional stack-based architecture—similiar to the machine models used in the implementation of imperative languages. Basically, the core is able to execute first-order applicative programs that may operate on simple, unstructured data like numbers only. The core consists of the following main parts:

a code area where the static program code resides,

a stack called the *dump*, that is used to store the *activation records* of nested function calls, and

an argument passing mechanism which may be implemented using a seperate stack or be merged into the dump.

The code area may contain instructions to move simple data, to execute the standard arithmetic, relational and logical operations, and to alter the control flow with conditional and unconditional branch instructions.

In order to be able to implement more sophisticated data structures, i.e., nested data terms (in the diagram located above the core), we have to add a *heap* which is used to store objects representing complex data. We call these objects *term objects* in the following. The term objects can be processed using the already introduced instructions, i.e., data movement to construct the objects and conditional branches to discriminate between different kinds of objects. Usually, together with term objects, a *case* or *switch* instruction is added which allows an efficient implementation of pattern-matching, and, in the context of partial data structures, also unification (see [PL92, MKLR90, Roy90, AEH94] for details).

For now we will not go into the details of representing term objects. We will first introduce the functional and logic components used to extend the core machine in an abstract way, i.e., stating their properties and what operational features from the lower part of the diagram they implement. Then, when moving to the problem of unifying the components in a single machine, we will present more details.

2.1.2 Functional extensions

First we look at the operational features that are to the left of the core in the diagram and are needed in order to be able to implement the full scale of modern functional programming languages. We need to add support for higher-order functions (including currying) and lazy evaluation. The former consists of two parts, namely the ability to treat *function labels as data* and to *support partial applications*. More precisely, function labels as data implies that the abstract machine must be able to pass function labels as arguments and to store them in term objects. Furthermore, in addition to branches where the target address is statically determined, it must be possible to perform an indirect jump to a function label which is stored in some given location. Partial application is the application of a function to less arguments than it demands in order to be evaluated. In this context it is also possible to apply an expression, which is not a function to some arguments or to apply some function to more arguments than it demands, provided the expression or the function finally evaluates to a partial application or a function—the latter can be seen as a partial application with no arguments supplied. As an example consider the following functional definitions:

```
foo x y z = ...
bar x     = foo x a
```

Evaluating the expression `bar b c` applies the unary function `bar` at two arguments, but as `bar b` evaluates to a partial application of `foo` which lacks one further argument. This finally results in the ternary function `foo` being applied to `b`, `a`, and `c`. Thus, the surplus argument `c` in the expression of `bar b c` is consumed by the result of `bar b`.

To implement this, we firstly have to be able to represent the result of an undersupplied (i.e., partial) function application in some way and furthermore, there has to be a mechanism taking care of the surplus arguments in a function application. In the following, we refer to the union of these two abilities as the support for partial applications. When the core machine together with term objects is extended with function labels as data and the support for partial applications then the resulting machine is able to implement strict functional languages like SML. Note that the mechanism of function labels as data may also be used in a logic implementation of say Prolog in order to compile the meta-call.

To execute lazy functional languages like Haskell, we furthermore have to add a mechanism that supports the suspension of expression evaluation. This is called a *suspension mechanism* in the following.

2.1.3 Logic extensions

Stepping back to the core machine, we have to support the operational features partial data and backtracking (in the diagram to the right of the core) in order to be able to execute logic languages like Prolog. To implement partial data structures the abstract machine has to be able to represent the undefined parts of a structure and to refine them. For example in the following Prolog fragment, we define and refine a partial data structure:

```
p(s(0)).
:- p(s(X)).
```

Here the data structure `s(X)` is only partially defined. By unifying `s(X)` with `s(0)` in the application of the predicate `p/1`, `X` is instantiated with `0`.

Partial data structures and their refinement are normally realized with a machine representation for unbound logical variables and a binding mechanism. Combining this binding mechanism with pattern-matching (which we added together with the term objects) leads to compiled unification.

The second basic feature of logic programming is the concept of multiple solutions—operationally this is backtracking. Thus, the abstract machine must be able to save the relevant parts of its state when alternative computations are possible; and it must be able to return to this saved state later on in order to proceed into one of the alternative computations not exploited so far. Therefore, following the example of the WAM, we represent states in computational history where alternative computations are possible using *choice points*. In the presence of free logical variables which are bound by side-effects all bindings performed since a choice point has been established have to be undone in order to reset the state of the abstract machine. This is implemented using a *trail stack* which records the bindings of logical variables.

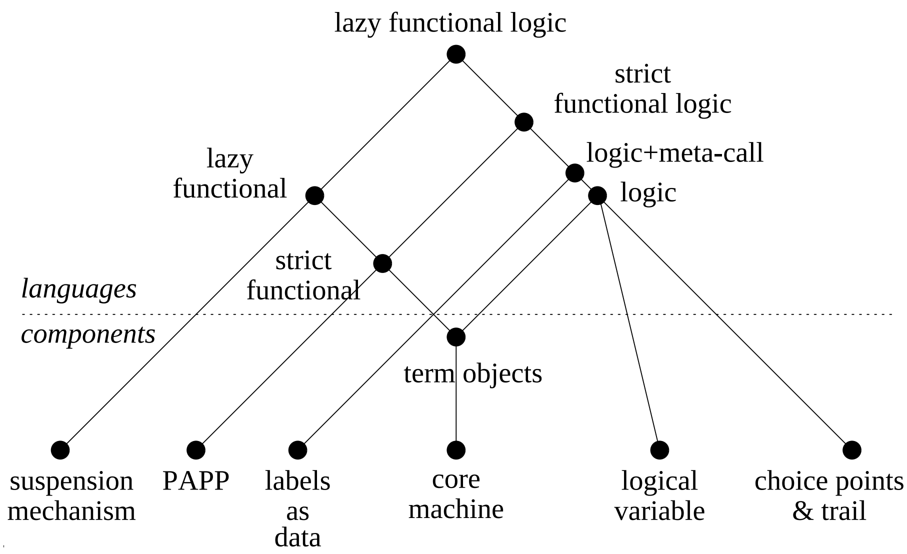

Figure 2: Design space opened up by JUMP-machine components.

When the core machine is extended with partial data structures and unification we can compile pure Prolog to machine code. Together with function labels as data the meta-call can also be compiled.

2.1.4 The component design space

Similar to the former visualization of the operational features in a diagram we can build a diagram where operational features are replaced by machine components used to implement the corresponding features. The result is given in figure 2. The operational core is implemented by the core machine at the center of the lower part of the triangle in figure 2. Like in the previous diagram the functional components are to its left and the logic ones are to its right. On the logic side, partial data and backtracking are replaced by logical variables, and the combined choice-point/trail technique respectively. On the functional side, the operational feature higher-order is replaced by the two components labels as data and support for partial applications. This allows us to identify a machine instance in the design space which implements logic languages including a meta-call.

2.2 Combining the components in a single machine

We presented the four major parts that form the JUMP-machine, namely the core machine, term objects, functional extensions, and logic extensions. Now we are coming to the crucial point of combining these four parts smoothly into a single machine. This amalgamation has to be performed at two levels: the level of control structures and the level of data structures. We will first concentrate on the amalgamation of the data structures.

On the level of data structures we have term objects, suspensions, representations for partial applications, and unbound logical variables. Furthermore, we need to represent basic values like numbers and maybe also strings in a way that is compatible with all the other data structures. The standard technique here is to tag objects, i.e., representations of different kinds of data objects are annotated with a tag value that is used to identify to which kind a data structure belongs. Apart from adding and stripping tags in order to process the “raw” data, whenever a data structure is accessed (for example in order to get its value), a case analysis has to be perfomed and depending on the kind of data, different actions have to be executed. For example, to get its value, a suspension has to be evaluated, and the value of some basic data can be obtained by stripping the tag.

When amalgamating paradigms, the number of different data structures grows. Therefore, it is more difficult to store the tag efficiently and the number of cases needed to apply some operation on an arbitrary data structure increases. To avoid this problem and to realize a uniform representation of all data objects, the JUMP-machine inherits an approach first proposed for the STGM [PS88]: all data structures are represented as closures located in the heap (a stack-like memory area which we introduced together with the term objects). A closure consists of a code pointer together with an environment. Different kinds of data structures contain pointers to different code. Whenever a data structure is to be evaluated its code is executed in the context of the closures environment. As the code depends on the kind of the data structure it "knows" what to do in order to evaluate this particular data structure. No case analysis has to be performed. This is called the *tagless approach*[1].

As the code pointer can be seen as a substitute for a tag on first sight a disadvantage appears to be that more storage space is needed for the code pointer than a tag would use. But the overhead is not as big as it seems because a suspension for example needs a code pointer anyway. Therefore, the storage needed for a suspension is even less in the tagless approach. Furthermore, term objects normally need a further tag which identifies the term constructor they represent. In the tagless approach this is not necessary as both the tag used to discriminate different kinds of data structures and the constructor tag can be represented using only one code pointer. As a result, the additional memory consumption of the tagless approach is not severe. A second objection against the tagless approach is that a big runtime penalty may be the frequent indirect jumps through code pointers of data structures which may invalidate the instruction pipelines of modern RISC-processors. However, the experiments we performed contradict this objection. Furthermore, the Glasgow Haskell Compiler which is implemented on the basis of the STGM uses the same technique and shows good performance results.

2.2.1 Active objects

The technique of representing arbitrary data structures as closures can be refined by using a pointer to a method table instead of a simple code pointer. The method table contains a set of various code pointers (i.e., labels) referencing routines performing different tasks on the associated data structure, like evaluating the structure, performing a garbage collection, or pretty printing the structure. This can be seen as a kind of object oriented approach. Each kind of data object residing in the heap possesses a collection of possibly different methods that are executed whenever a certain action has to be performed in conjunction with the data object. Further details on this kind of data structure representation and on the use of different methods can be found in [Pey92].

We call data objects representing data structures in this way *active objects* because these objects are not passively manipulated by the program but participate actively in the process of computation[2]. Active objects can be visualized as shown in figure 3. The area to the right of the dashed line represents the heap where the active objects reside. They consist of a code pointer referencing the method table in the code area to the left of the dashed line and some arguments. These arguments are the same as the environment in the case of a normal closure, i.e., data local to the object. The number of arguments may vary even within a single class of active objects, representing the same kind of data structures, as long as the code referenced by an object fits to that object's layout and expected behaviour. Nevertheless, objects may share complete method tables or at least some of the routines referenced in the method table.

In contrast to a normal closure which contains a direct reference to the associated evaluation code, active objetcs use a further indirection via the method table. This adds an additional dereference operation whenever an active object is to be evaluated. When machine code (assembler code) is directly generated this additional indirection can be eliminated. This is done by replacing the pointer to the method table by a direct pointer to the evaluation code (label). The remainder of the method table is simply placed in the memory area in front of the evaluation code label. This optimization cannot be applied when C-code is generated instead of machine code.

The concept of active objects is very flexible and the key feature to the smooth integration of different paradigms in the JUMP-machine. As a consequence of the active nature of this data structure representation even some of the problems when amalgamating control structures can be solved as we will show when describing how unbound logical variables are represented in the JUMP-machine.

[1] Which is where the 'T' in STGM comes from.

[2] This property of active objects lead to the name of the JUMP-machine. In order to apply an operation (like evaluation) to an object the code *jumps* through the objects method table.

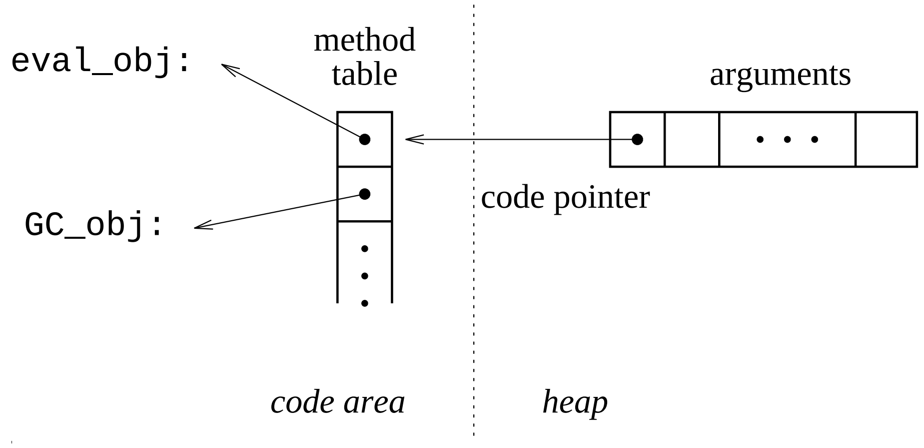

Figure 3: Uniform representation of active objects.

In the following, we show how the machine components introduced previously are added to the core machine which is enriched with active objects. Note that the addition of active objects requires an additional control structure in the core, namely indirect jumps.

2.2.2 Term objects

Term objects are used to represent a data constructor together with its arguments. In the compiled code a constructor is identified by a unique value called the constructor tag—in a strongly typed language the value needs only to be unique within the type.

An active object, which represents a constructor, references an evaluation method, which simply returns the constructor tag when executed. The arguments of such an active object are simply references to the constructor's arguments. The case or switch instruction necessary for an efficient implementation of pattern-matching, uses the tag value as the index in a vectored jump. The vectored jump can even be moved into the objects evaluation method, which saves one jump instruction. This optimization is called vectored returns—for details see [Pey92].

2.2.3 Higher-order features

As previously described, we need to be able to treat function labels as data and to support partial applications in order to implement higher-order features. To treat labels as data and perform indirect jumps using these labels is already a basic concept in the core machine enriched with active objects.

Partial applications consist of a function applied to too few arguments. Whenever a partial application is applied to some arguments it has to check if, together with the already supplied arguments, the function is applicable, i.e., if the sum of all arguments is at least as high as the arity of the function. If enough arguments are present the function is called; otherwise, a new partial application has to be build from the old one together with the new arguments. The code performing these operations for a function of arity n that is already applied to k arguments ($k < n$) is called the *uncurry_n_k*-code and constitutes the evaluation method of the active objects used to represent partial applications of n-ary functions at k arguments—these active objects are called *PAPP* objects for short.

The surplus arguments that may occur in conjunction with partial applications can be stored on the argument stack of the core machine. Some additional care has only to be taken in conjunction with lazy evaluation as we will explain below.

2.2.4 Suspension mechanism

Representing suspended evaluation using active objects is straightforward as a closure always represents a suspended computation and active objects are merely a generalization of closures. The evaluation method is the code performing the computation, which is represented by the suspension, and the objects arguments are the computations environment, i.e., a valuation for its free variables.

To obtain lazy evaluation we need to add the concept of sharing, i.e., a suspension shared by different computations is evaluated at most once. Thus, the data structures in the heap are graphs not trees. When a suspension is evaluated, it not only returns its result but also overwrites its active object with the result—this is called *updating* the suspension. In general, this cannot be done directly as the result may need more heap space as the suspension object occupies. Therefore, we introduce indirection objects. They contain a single argument which is another active object. When evaluated, an indirection object proceeds by executing the evaluation code of the object referenced by its single argument.

Combining updating of suspensions with PAPP objects leads to some difficulty. Consider the following functional program fragment:

```
let g = foo 1
in (g 2, g 3)
```

where `foo` is a binary function. In the context of lazy evaluation `g` will be bound to a suspension which when evaluated applies `foo` at `1`. As this suspension is shared between `(g 2)` and `(g 3)` it is evaluated as soon as one of this applications of `g` is evaluated. When `(g 2)` is computeted, first `2` will be pushed onto the argument stack and then the shared suspension bound to `g`, which is `(foo 1)`, is evaluated. This results in `1` also being pushed onto the argument stack. Now, `foo` is called. As two arguments are on the stack the binary function `foo` can be evaluated directly. Unfortunatly, the result of the shared suspension `(foo 1)` which is a PAPP, was never used to update the suspension. The PAPP was not even created—which is a nice optimization when no updating has to be performed. The problem in the example is that `foo` takes both arguments at once when it is allowed to take only the topmost, namely `1`.

In contrast to the difficult trip wire techniques normally used [PL92], we propose a simpler solution which needs only one additional machine register, called the *argument count register*. It contains the number of arguments that may be savely consumed from the argument stack. In the above example this is one when `foo` is called. The argument count register has to be stored in the activation records which are build to manage recursive function calls. Whenever the register is saved, its value is reset to zero and when the corresponding activation record is popped, the saved value is loaded back into the register. If a suspension is executed which needs to be updated, it simply creates a new activation record storing the old value of the argument count register and resets the register to zero—this hides all pending arguments from function calls occuring while evaluating this suspension.

A similar technique was independently invented in the second version of the STGM [Pey92]—but with an argument stack base register instead of an argument count register.

2.2.5 Unbound logical variables

Using active objects, the integration of unbound logical variables realising partial data structures is very simple. When a logical variable is evaluated, nothing special happens: it simply returns. But some operations, like unification, need to know when they encounter unbound logical variables. Therefore, whenever an expression is evaluated two return labels are provided—forming a *return vector*. The first return label is used in the case of returning from an unbound logical variable, otherwise, the second is used. Thus, the context of an evaluation determines which special actions have to be taken in the case of a free variable. Instead of testing a tag on the callers side and then branching to the correct code the callee directly returns to the appropriate code portion of the caller. For example, in the context of an unification operation, the code referenced by the return label for free variables binds the variable to the term it has to be unified with.

Notice the smooth interplay between for example logical variables and suspensions using active objects. A unification instruction simply executes the evaluation method of its arguments. This is done in the form of a function call where an appropriate return vector is passed. If one argument is a suspension, it is 'automatically' evaluated. This in turn may lead to the executions of evaluation methods of further objects within the suspensions environment. If the final evaluation method executed belongs to an unbound logical variable this will return to the unification instruction using the first entry in the return vector. The code referenced by the first entry will lead to the binding of the logical variable executed lastly to the other argument of the unification operation. In this process not a single test has to be performed.

To bind a logical variable its object is overwritten using an indirection object (the same kind that is also used to update suspensions). Therefore, unbound logical variables have to be represented by an active object possessing one unused argument—in order to allocate enough memory to be able to store an indirection node in the same location.

The usage of return vectors allows to treat unbound variables differently in different contexts without extra cost. For example in purely functional parts of a GTML program the evaluation of an unbound logical variable leads to a runtime error. This is implemented by placing the label of an error routine in the first entry of the return vector in such a context.

2.2.6 Backtracking

The basic mechanisms used to implement backtracking in the JUMP-machine are similiar to the WAM. Choice points are placed on the dump stack in order to protect activation records that may be used in alternative computations from being discarded. Furthermore, a trail stack is used to record the binding of logical variables that have to be undone on backtracking—in the JUMP-machine this means to replace the pointer to the method table of an indirection object by the pointer to the method table of an unbound logical variable. As in the WAM, one simple store operation suffices.

Nevertheless, there are some very interesting differences between the WAM and the JUMP-machine. In the rule selection mechanism of the original WAM, a choice point is created whenever a case analysis on the outermost constructor of the leftmost argument that is part of the unification process does not lead to an unique selection of a single rule. In the JUMP-machine unification uses exactly the same type of case analysis that is employed when doing pattern-matching. Therefore, in the case of uniform patterns [MKLR90], no choice point is generated as long as no unbound variable in an argument leads to a situation where more than one rule applies. This means that choice points are only build in order to manage the different bindings of a single logical variable. Therfore, these choice points are called *variable choice points* in the JUMP-machine. Only in the case of non-uniform patterns, so-called *rule choice points* which are similiar to the choice points of the WAM have to be used.

As explained above, logical variables are identified by the return label they use when being evaluated. The code referenced by the return label used by logical variables is the only code that builds up variable choice points. The code referenced by the other label is the same as the one used in pattern-matching. This leads to an on-the-fly detection of deterministic computations. When only evaluation codes of ground terms are executed in a rule selection process, no choice point is ever created in the case of uniform patterns. If the patterns are non-uniform, at least a reduction of the number of choice points is achieved.

In conjunction with the *uniformity transformation* presented in [Loc92], redundant unification steps can be eliminated and deterministic subcomputations be exploited in order to reduce the search space in the case of narrowing—this is very similiar to the techniques proposed in [AEH94, Han94].

Optimizations to the WAM in order to obtain the same computational properties as decribed above have been proposed in [MKLR90, LW91]. In the JUMP-machine these optimizations are naturally present.

2.3 JUMP-machine languages

In contrast to other abstract machines for declarative languages the JUMP-machine does not possess a single machine languages but two. This allows to express JUMP-machine programs on two different abstraction levels which has three main advantages:

1. the properties of each language level support different kinds of analysis' and code optimizations,

2. compilers based on the JUMP-machine are highly portable even when assembler code is generated, and

3. the compilation process is split in smaller and therefore simpler transformation steps which makes them easier to comprehend.

We will now provide a short description of the most important properties of both language levels. It is not possible to describe the JUMP-machine languages in more detail here. Extensive descriptions of the languages and translation schemes mapping different classes of functional logic languages onto these intermediate languages are given in [Loc92, Cha91b, CL91].

2.4 JUMP-machine code

The machine language which is more abstract is called *JUMP-machine code*, *JCode* for short. It contains function and data type definitions. The code is block structured and has symbolic variables. All the operational features of the JUMP-machine are already explicitly visible in JCode-programs, e.g., pattern-matching and unification are replaced by switch instructions performing an explicit case analysis and unbound logical variables have to be explicitly created.

The most important instructions are the assigment, term creation, a conditional instruction, a switch instructions, an arbitrary choice instruction for non-uniform rule selection, creation of unbound logical variables and of suspensions, and of course function calls with explicit, uncurried parameter passing and special instructions to perform tail calls.

This level still abstracts completely from the memory mapping including the JUMP-machine stacks and registers. Furthermore, a variable may be defined only once in a single possible control flow. Therefore, it is impossible to overwrite old values; aliasing is also impossible.

As a result, dataflow oriented optimizations like copy propagation, dead code elemination, and certain operations to calculate a good memory mapping are very simple. Additionally, code transformations like, for example, inlining of function calls can be performed easily on the JCode level.

2.5 TreeCode

The lower language level is represented by a language called *TreeCode*, short *TCode*. It can be viewed as a kind of complex assembler code for the JUMP-machine—which is still completly independent from any concrete target machine. The code is linear and consists mainly of labels, branches, arbitrary assigments, indirect jumps, memory accesses, and the typical arithmetical and relational operations.

Furthermore, memory mapping is explicit on this level and the JUMP-machine's stacks and registers are directly manipulated. There is even machine independent support for the allocation of machine registers. This is done in the form of an infinite set of pseudo-registers storing intermediate results which are candidates for being placed into concrete hardware registers in the last compilation step, i.e., transformation of TCode to the final assembler code.

On this level, optimizations like machine independent common subexpression elemination within access paths, elemination of branch cascades, and preparations for good register allocation can be performed.

Translating TCode to concrete machine code of CISC and especially of RISC microprocessors is very simple due to its low abstraction level. Therefore, the only machine dependent phase of a compiler based on the JUMP-machine is simple and short.

2.6 Properties of the JUMP-machine

We finish the description of the JUMP-machine with some of its general properties and justifications for some design decisions. An important goal in the design of the JUMP-machine was to support the development of simple compilers that produce fast code. As a result we looked for implementation techniques leading to efficient code without the need of difficult optimizations, especially without the need for global program analysis. As we will see in the next subsection the GTML-compiler generates code comparable to good single-paradigm compilers without the need for any sophisticated optimization techniques. On the other hand the abstract machine should be able to incorporate the results of global analysis and other techniques when they are present. This is definitely possible in the JUMP-machine as standard optimizations from the area of functional implementations like strictness, boxing, and sharing analysis are directly applicable on the level of JCode. Also optimizations like the jump vector technique of the STGM can be directly used in the machine. JCode also supports the application of code transformation techniques. Furthermore, traditional optimizations from the area of logic programming like mode analysis and also many of the techniques described in [Roy90] can be used. In addition to this high-level optimizations, traditional optimizations like copy propagation, dead code elemination, common subexpression elemination, and register allocation can be used without difficulties.

In order to produce fast code we decided to generate fully compiled code instead of byte code that has to be interpreted. Using byte code would involve the overhead of a case analysis in the interpreter for every abstract machine instruction and for example the concrete machine registers could not be utilized easily in order to store intermediate results. Not using machine registers leads to a higher memory access rate which often slows down computations considerably.

We also did not use C as a kind of macro assembler as is often done when implementing declarative languages. Some of the difficulties when using C as an intermediate language in the context of the tagless approach to data representation are described in [Pey92]. For example, the Glasgow Haskell Compiler originally generated C code only; the new versions provide an assembler code generator for the SPARC also. One example for an optimization not possible when generating C code is the removal of the additional indirection introduced by the method table (see section 2.2.1).

The main advantage when generating C instead of machine code is the better portability. But as a lot of non-standard tricks (often possible only when using the GNU C Compiler) have to be applied in order to produce efficient C code, porting such an compiler is not as easy as it may seem. Furthermore, native code compilers based on the JUMP-machine are not difficult to port. We ported the GTML-compiler from the MC68020 to the SPARC-processor (not using all features of the processor) within roughly one week—this included understanding the processor architecture.

One argument for generating C code is that C code generation allows to exploit all the knowledge about optimizations encoded within C compilers. This is misleading as we do not need most this knowledge in a compiler for a declarative language. All the optimizations within a C compiler have to be very conservative and are often difficult because they have to deal with arbitrary side effects, aliasing, and similiar problems. Most of the optimizations that are very difficult to implement in a C compiler are fairly easy when compiling well behaved functional logic languages and some like loop unrolling are totally useless. Additionally, these optimizations have a disadvantage in common: they are very slow.

As a result we think it makes sense to generate machine code and therefore, an abstract machine should be well suited for direct machine code generation. In additon to the generation of machine code a C code generator is useful in order to be able to do a provisional quick port of the compiler even if this results in slower code.

3 The GTML-Compiler

We implemented a compiler for a functional logic language located at the top of the triagular diagram of figure 1 using the JUMP-machine. Despite the fact that we did not implement additional sophisticated optimization techniques the code produced by the compiler is efficient and comparable to optimizing implementations of purely functional and purely logic compilers like the Glasgow Haskell Compiler (Version 0.19) and Quintus Prolog (Version 3.1.3).

In the following, we shortly introduce Guarded TermML, state some facts about the implementation, and will give some timing results.

3.1 Guarded TermML

The language definition of Guarded TermML (GTML) can be found in [Cha91a] (unfortunately, this is in German) and some of the key features are descibed in [Loc88, DHL91]. Here we will merely outline the basic concepts of the language.

GTML can be seen as the composition of a lazy functional language and a restricted form of Prolog together with an interface between both parts of the language. It is possible to write purely functional, purely logic, and functional logic programs in GTML.

The functional part is a mixture between ML and lazy functional languages like Haskell. It is strongly typed and the evaluation order can be freely choosen between strict and lazy. All of the standard features like product-sum data types, pattern-matching, higher order functions, and currying are available. Furthermore, list-comprehensions can be used and the module system supports the definition of abstract data types.

The logic part is a strongly typed subset of Prolog, which does not support the meta-call, and the assert and retract predicates. The meta-call was considered to be not very important because the functional part of the language supports higher-order features and with a general meta-call it would not be possible to fully compile all GTML programs—terms that are called within the program might be build up dynamically. Assert and retract where omitted for similiar reasons.

The interactions between functional and logic part are twofold:

1. Logic goals can be used instead of boolean guards in the rule selection process of functions and in list-comprehensions. Thereby, solutions from the logic world can be used in the functional part. Logic guards in the rule selection process allow to exploit the first solution of the goal only. In list-comprehensions they can be used similiar to the bagof-predicate of Prolog.

2. Arbitrary functional expressions may be used as arguments to predicates. They are reduced on demand of the unification procedure applied in the process of clause selection. In the rule selection process of functions called in such a logic context unification is applied instead of pattern-matching, i.e., narrowing is used instead of simple reduction.

The combination of both types of interaction allows an arbitrary interlocking of functional and logic computations.

We will now give two short examples for each of the two interaction mechanisms. In the following GTML fragment, logic is used by the functional part:

```
append ([], Xs, Xs).
append ([X|Xs], Ys, [X|Zs]) :- append (Xs, Ys, Zs).

fun prefix (S, L) | append (P, S, L) -> P.
```

The partial function `prefix` is defined when there exists a prefix list `P` such that `P` appended at `S` results in `L` and `L` is finite. In this case the prefix `P` is returned.

In the second program example we are using a functional expression in a predicate:

```
fun take (0, _)      -> [].
fun take (N, [X|Xs]) -> [X | take (N - 1, Xs)].
frags (N, Xs, take (N, Ys)) :- rotate (Xs, Ys).
```

where `rotate` succeeds iff `Ys` is a rotation of `Xs`.

3.2 The compiler

The GTML-compiler translates the declarative source code into symbolic assembler code. Currently, there exists a code generator for the MC68020 and the SPARC-processor. One translating TCode to C is nearly finished.

The main restriction of the current version of the Compiler is that it does not support the lazy narrowing strategy presented in [CL91]. As a consequence functional expressions in argument positions of predicates are simply reduced and not narrowed. If such a reduction encounters an unbound logical variable a runtime error occures, as in the reduction process of expressions not occuring in a logic context. Nevertheless, the compiler implements all the features of the JUMP-machine described in this paper and their combinations. This includes the interleaving of unification with reductions steps and backtracking in the presence of reduction.

The current version of the compiler does not employ sophisticated optimization techniques and does not even use all of the local optimizations possible when using the JUMP-machine like, for example, vectored returns to eliminate one of the three jumps always needed when performing the case analysis used in pattern-matching and unification—this optimization originates from the STGM. Also some low-level optimizations possible on the TCode level like common subexpression elimination within access paths are not implemented yet. Even without these optimizations the mere structure and organisation of the JUMP-machine leads to fast code and especially, performs deterministic computations in a functional logic program within a factor of three compared to the Glasgow Haskell Compiler.

3.3 Efficiency

The following benchmarks were executed on a Sun 4/330 and also on a Sun 10/30. We had to use the Sun4 because no Quintus-Prolog-Lisence was available for a Sun10. On the other hand the relative performance of GTML to C is much better on the Sun10. This is because a certain optimization with respect to the memory mapping is currently not implemented in the GTML-compiler and both machines have different memory access times. This optimization will be integrated in the future.

The competitors are Quintus Prolog version 3.1.3, the Glasgow Haskell Compiler version 0.19 (using optimization level `-O2`), and the GNU C Compiler 2.5.8 (using optimization level `-O2`).

We performed six different benchmarks testing different aspects of the GTML implementation. Not all benchmarks were implemented with all systems, e.g., partial data structure, the backtracking benchmarks were realized with Quintus Prolog and GTML only.

The results of the benchmarks timings can be found in Table 1 (all times are given in seconds). In

	C		GHC		QP	GTML (fun/log)	
	Sun4	Sun10	Sun4	Sun10	Sun4	Sun4	Sun10
nfib	2.9	1.7	10.7	3.5	38.8	12.5/50.0	3.4/10.2
ffaba	-	-	-	-	1.3	1.5/1.7	0.4/0.5
islist	0.7/18.1	0.3/9.6	1.9	0.7	3.4	4.2/3.9	1.4/1.3
btree	2.0	1.0	2.8	0.8	6.4	7.0/8.5	1.8/2.5
btreep	-	-	-	-	4.3	-/6.5	-/2.0
back	-	-	-	-	2.7	-/3.7	-/1.0

Table 1: Results of the benchmarks in seconds.

the table are always two times given for the execution of GTML. The left one is the running time of a functional and the right one the running time of a logic version of the benchmark. We will now briefly discuss the results of these benchmarks:

nfib is the standard nfib-routine. The bad results of Quintus Prolog and the logic version of GTML are caused by the fact that in the functional case the compilers can use unboxed values. This is not easily possible in the logic case. Comparing the execution times of the functional version of GTML with C on both the Sun4 and the Sun10 shows the impact of the different memory access times of both machines.

ffaba is a pattern-matching test originally employed to test attribute grammar evaluators. Starting from the term `f (f (a, b), a))` all the `a` are replaced by the term itself. The same procedure is applied again on the result. This is done five times. Then, the resulting term consists of a term containing 2^{16} references to a term with 2^{16} `a`-nodes.

islist traverses a list of 10000 elements a 100 times. The two running times given for C are an iterative and a recursive version. It is interesting to note that the logic version of GTML in this case is faster than the functional one. This is because the logic version only traverses the list without returning a result and the functional version has to return a truth value.

btree Builds a perfectly balanced binary tree from 2^{14} numbers and sums it up. It is interesting to note that on a Sun10 the GHC is even faster than C, despite the fact that the C program destructively updates the tree whereas the functional program has to copy parts of the tree. The reason is that the C program does not use an optimized memory management scheme but allocates the nodes of the tree with the standard `malloc`-routine.

btreep is similiar to btree but uses a partial data structure in order to represent the binary tree. Therefore, no nodes have to be copied when inserting a new node and no functional versions of the program are given.

back tests the speed of backtracking. It executes 100000 backtrack operations while traversing a list.

The results clearly show that the JUMP-machine efficiently implements functional and logic computations. The benchmark programs islists and btree (even btreep) can be written in a mixed style using both functions and predicates calling each other. No penalty in comparision to the given running times can be observed in this case.

4 Conclusion

The JUMP-machine is an abstract machine model for the efficient implementation of multi-paradigm languages. It consists of a core machine which is extended with components supporting different declarative features. These features are integrated using active objects—a generic mechanism providing an uniform representation of different kinds of data objects. In contrast to other abstract machines for multi-paradigm languages the JUMP-machine is not specialized towards a specific language but implements a whole class of languages efficiently. Furthermore, it supports the easy generation of fully compiled code.

A compiler for the functional logic language Guarded TermML (GTML) was implemented using the JUMP-machine. It shows that it is possible to compose different paradigms efficiently using the proposed techniques.

The lazy narrowing strategy presented in [CL91] is not implemented in the GTML-Compiler until now. We will implement it in the near future. Currently, we are working on the implementation of a concurrent declarative language that utilizes simple constraints in order to describe process and communication structures. In this work we are using the flexibility of the active objects extensively. The basic principles of this work are described in [Köh92, CK92]. Furthermore, we are investigating if it is possible to build more complex constraint solvers efficiently ontop of the simple constraints. This would lead to a different approach of integrating constraint solvers into functional logic languages. Instead of being a black-box hidden in the unification mechanism the solver would be written in the source language on the basis of some primitive constraint mechanisms.

We thank Martin Simons and Gaby Keller for their helpful comments, and the anonymous referee for his valuable suggestions.

References

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 268–279, 1994.

[AK90] Hassan Aït-Kaci. The WAM: A real tutorial. Technical report, Digital Equipment Corporation, 1990.

[BCM89] P.G. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF: a WAM-based compilation of conditional narrowing. In *Proceedings of the 6th Int. Conf. of Logic Programming*, 1989.

[Cha91a] Manuel M.T. Chakravarty. Die funktional logische Programmiersprache Guarded TermML: Sprachdefinition. Studienarbeit, GMD Research Center Karlsruhe, Mai 1991.

[Cha91b] Manuel M.T. Chakravarty. Entwicklung und Bau eines Übersetzers für eine funktional logische Programmiersprache. Master's thesis, University of Karlsruhe, Oct 1991.

[CK92] Manuel M.T. Chakravarty and Martin Köhler. Equational constraints, residuation, and the parallel JUMP-machine. In *4th Int. Workshop on the Parallel Implementation of Functional Languages*, 1992.

[CL91] Manuel M.T. Chakravarty and Hendrik C.R. Lock. The implementation of lazy narrowing. In J. Maluszynski and M. Wirsing, editors, *PLILP*, pages 123–134, 1991. LNCS 528.

[DGK93] John Darlington, Yi-ke Guo, and Martin Köhler. Functional programming languages with logical variables: A linear logic view. In *Programming Languages and Logic Programming*, 1993.

[DHL91] Roland Dietrich, Birgit Heinz, and Hendrik C.R. Lock. Guarded functional programming and lazy streams, 1991. GMD Forschungsstelle an der Universitat Karlsruhe.

[H+92] P. Hudak et al. Haskell special issue. *ACM SIGPLAN Notices*, May 1992.

[Han91] Michael Hanus. Efficient implementation of narrowing and rewriting. In *Workshop on Processing Declarative Knowledge*, 1991. LNCS.

[Han94] Michael Hanus. Lazy unification with simplification. In *Proceedings of the European Symposium on Programming*, 1994.

[Köh92] Martin Köhler. Parallelisierung funktionaler Programmiersprachen durch Residuation. Master's thesis, Technical University of Berlin, Oct 1992.

[Loc88] Hendrik C.R. Lock. Guarded TermML. In *Workshop on Implementations of Lazy Functional Languages*, 1988. Aspenas, Univ. Göteborg, Sweden.

[Loc92] Hendrik C.R. Lock. *The Implementation of Functional Logic Programming Languages*. PhD thesis, Technical University of Berlin, 1992. While working for the GMD Karlsruhe.

[Loo91] Rita Loogen. From reduction machines to narrowing machines. In *CCPSD, TAPSOFT*, pages 438–454, 1991. LNCS 494.

[LW91] Rita Loogen and Stephan Winkler. Dynamic detection of determinism in functional logic languages. In *PLILP*, 1991. LNCS 528.

[MKLR90] Juan J. Moreno-Navarro, Herbert Kuchen, Rita Loogen, and Mario Rodriguez-Artalejo. Lazy narrowing in a graph machine. In *Conf. on Algebraic and Logic Programming*, 1990. LNCS 463.

[MR92] J.J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:189–223, 1992.

[MR99] J.J. Moreno-Navarro and M. Rodriguez-Artalejo. BABEL: A functional and logic language based on constructor discipline and narrowing. In *First International Conference on Algebraic and Logic Programming*, pages 223–232, 1999. LNCS 343.

[Müc92] Andy Mück. Camel: an extension of the categorial abstract machine to compile functional logic programs. University of Passau, 1992.

[Pey92] Simon L. Peyton-Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, version 2.4. Technical report, Department of Computing Science, University of Glasgow, 1992.

[PL92] Simon L. Peyton-Jones and David R. Lester. *Implementing Functional Languages: a Tutorial*. Prentice-Hall, 1992.

[PS88] Simon L. Peyton-Jones and J. Salkild. The spineless tagless G-machine. In *Workshop on Implementations of Lazy Functional Languages,* Aspensas, Schweden, 1988.

[Roy90] Peter Lodewijk Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming*. PhD thesis, University of California, Berkley, Dec 1990.

[SS86] Sterling and Shapiro. *The Art of Prolog*. MIT Press, 1986.

[War83] David H.D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, October 1983.