Equational Constraints, Residuation, and the Parallel JUMP-machine

Manuel M.T. Chakravarty Martin Köhler

September 23, 1992

Abstract

We will present an extension of conventional functional programming languages by a simple form of equational constraints. The resulting language allows the explicit specification of parallel algorithms on an abstract basis without any knowledge of the used execution model.

To decribe the operational semantics of the extension, we incorporate residuation and unification into the operational model of (compiled) graph reduction. We will further describe a modified version of the JUMP-machine that allows the implementation of such a language on parallel hardware.

1 Introduction

A lot of research is directed towards the parallel implementation of functional programming languages. The usual idea is to exploit the implicit parallelism found in functional programs through the use of so called parallelizing compilers [HP90, HKL92, NPS91]. Such a compiler attempts to spot computations that are independent of each other, and hence may be evaluated in parallel. This method leads to a couple of serious problems, among which task granularity is one of the most important. It is (at least until today) impossible for the compiler to spot which subcomputations within an arbitrary functional program are usefully computed in parallel, considering the extra cost of process creation on (preferable) another processor. To circumvent this problem sometimes more or less machine dependent annotations are used to mark those computations which the programmer wants to execute in parallel [Ach91].

Our aim is to invent an extension to functional languages that allows the programmer to specify the different processes in an explicit but abstract way. Here, *abstract* means that the programmer is able to describe parallelism without any knowledge about the implementation of the used language. Instead she designs the programs on the basis of an abstract view of the languages semantics. So, the programmer chooses which computations ought to be computed in parallel and hence controls the granularity employed. In addition, the use of a restricted form of logical variables allows the description of an abstract kind of bidirectional communication.

In the second part of this paper we will introduce some modifications to the JUMP-machine (an abstract machine for the compilation of functional logic languages). The modified machine (called JUMP*) may be used to implement the introduced constraint functional language on parallel and sequential computers.

2 Constraints and functions

Given a "standard" lazy functional languages, we introduce equational constraints as annotations for expressions. These constraints have to hold for the expression to be valid. An invalid expression is semantically considered to be bottom, i.e. it has no legal value. Together with the constraints we allow the use of a restricted form of the logical variable, which is unbound when it is declared (meaning, no value is associated with such a variable) and may then be instantiated in another part of the program, i.e. a value can later be bound to the variable. These variables are called *residuation variables* or *residuals* and are very similiar to the I-structures of Id Nouveau [ANP89, NPA86] and the residuals in Life [AKP91].

In contrast to the free variable in logic programming languages like Prolog a residual retains its value once it is bound, i.e. there is no concept of indeterminism or operationally speaking as backtracking.

2.1 The language

Annotating an expression with some constraints means that after an arbitrary expression there may be a comma separated list of constraints enclosed in double angle brackets (`<<` and `>>`). In addition to the constraints, the brackets may contain an enumeration of newly declared residuals, which may be used within the constraints and the annotated expression.

A *simple* constraint consists of two expressions seperated by an equation symbol (`=`), which states that both expressions must be unifiable. More complex constraints may be build by so called *constraint functions*, i.e. functions which don't return a domain value, but a set of constraints. Such a function is of result type `void` and its body expression consists of a constraint list only.

Instead of giving a formal syntactic description of the language[1], we will give an example. This is a parallel version of the well known `map`-function, often called `farm`.

```
farm f []      -> [];
‘    f (x:xs) -> (x’ : farm f xs) <<x’: x’ = f x>>.
```

While traversing the source list given as second argument to `farm`, a result list of equal length is constructed. The elements of the resulting list are constrained to be the result of applying the function `f` to the corresponding elements in the source list. Through the use of a constraint, the evaluation of every application of `f` to a list element may be done in parallel. Additionally, the construction of the result list is independent of the calculation of the list elements: `x’` is a residual, a fill-in, representing the function applications result.

It is guaranteed that the value represented by a residual is available when it is used within a computation. Operationally, this is realized through a suspension of any process which tries to access an unbound residual. Such a process is reactivated as soon as the residuals value is computed. Hence, the unbound nature of residuals is not noticeable from within a program.

2.2 Some examples

This subsection presents a first example program showing the granularity control and the abstract communication. A second example demonstrates the use of constraint functions.

The first example uses the function `farm` introduced in the preceding subsection. In addition, we introduce a function `sum` which sums up a list of numerical values. Both functions are combined in one expression.

```
sum []      -> 0;
sum (x:xs) -> x + sum xs.

eval sum (farm f l).
```

The function `f` maps elements of the input list `l` to a number. These numbers are summed up by `sum`. The traversal of the input list, the construction of the resulting list, and the summation of the numbers is done by one process. But the evaluation of every single application of `f` is independently done within a seperate process (through the constraint in the definition of `farm`). Note that the elements of the result list are distinct residuals serving as fill-ins until the constraints are evaluated. The resulting distribution of work is reasonable if a single computation of `f` is (very) expensive compared to a simple addition of two numbers, and the list is not long enough to justify the use of a tree-like reduction instead of the linear one implemented by `sum`. In this example we obviously controlled the distribution of work to different processes explicitly but on an abstract level of expression, and the residuals served as an abstract means of communication.

An example of a higher order constraint function is `mapc`. A function very similiar to the conventional `map`. It applies a given constraint function to the elements of a list. The definition is as follows:

[1] Which won't be very interesting because the only invention are the constraint annotations.

```
mapc          :: (a -> Void) -> [a] -> Void.
mapc c []     -> <<>>;                          -- the empty constraint
mapc c (x:xs) -> <<c x, mapc c xs>>.
```

The second equation applies the constraint function `c` to the current list element `x` and recursivly descends on the remaining list elements producing further constraints.

3 Operational semantics

For a better understanding of the new language features we will present briefly the operational semantics of the new language. A detailed description can be found in [Köh92].

As mentioned previously, every given constraint may be evaluated in parallel to all the other constraints and the expression annotated with that constraint, i.e. each constraint represents a process. Thus, the state of a computation is specified by a list of currently existing processes. Some of these may be waiting for a residual that is not bound yet—they are called *suspended* processes. The others are called *active*. As soon as an expression is evaluated, a new process is created for every constraint within the annotated constraint list, if there is any. If the process of a simple constraint finishes its unification or a constraint function is completly evaluated, the corresponding process dies.

Just before the processes implied by a constraint list are forked, the newly defined residuals of that constraint list are created—they are in an unbound state at the time of there creation. As mentioned previously, a process trying to access the value of an unbound residual is suspended until that value is computed, i.e. the value is bound to the residual. So, binding a residual may result in a couple of processes to be reactivated. The special case that all processes are suspended, i.e. waiting for residuals to be bound, is called a *dead lock*—it causes a runtime error.

Given this semantics one may ask: What are we gaining and what price do we have to pay? The price is the loss of referential transparency. That means it is not always possible to replace every occurence of a variable with its value without changing the meaning of the transformed expression. To abandon the referential transparency is widely regarded to be a bad move as such. But there are different ways to lose this property, which lead to different consequences. Here we lose the referential transparency through the creation of residuals. Given a residual generator function `make_res` with the following definition

```
make_res () -> x <<x:>>.
```

and the expression

```
LET x = make_res () IN 42 <<x = 1, x = 2>>
```

we observe that this expression has no value since a single residual can't be bound to the value `1` and `2`, i.e. `1` and `2` are not unifiable. Note that the constraints of the expression have to be computed even if the expression, namely `42`, doesn't depend on any residuals. Replacing all occurences of `x` with the right hand side of the local binding gives

```
42 <<make_res () = 1, make_res () = 2>>
```

Now there are two different residuals created within the first and the second constraint. One is bound to `1`, and the other to `2`, which poses no problem and allows the expression to be evaluated to it's value of `42`. More about the issue of referential transparency as well as an abstract and an operational semantics of a strongly related language, namely a subset of Id Nouveau, can be found in [JPP91].

We conclude, that as long as once shared values are never unshared standard reasoning techniques about functional programs can be applied to our language. So, we don't lose to much, but we are gaining the ability to explicitly describe parallel algorithms on an abstract level. Additionally, the residuals are useful in program development—independent form any parallel execution. For example, to realize open data structures.

4 Towards JUMP*

The JUMP-machine is an abstract machine designed to implement functional, logic, and functional logic languages [Loc92, CL91, Cha91]. The machine integrates the components necessary for these different language classes through the utilization of a uniform closure technique. It supports the direct generation of machine code due to features for easy register allocation and an as far as possible target machine independent structure.

In the following we introduce some extensions and modifications allowing the compilation of the above constraint functional language to parallel and as well sequential hardware. We start with a new kind of objects, namely residuation objects and two process management instructions, namely `FORK` and `DIE`. These modifications are visible on the level of the abstract JUMP-machine code (called *JCode*). Thereafter, we sketch the modifications necessary on the levels underlying JCode, i.e. the behaviour of the machine while executing a JCode program. These modifications are motivated with some examples. But first of all comes a short description of the original JUMP-machines main features.

4.1 The original JUMP-machine

The JUMP-machine consists of a *conventional core* similiar to the abstract machines used to translate imperative programming languages. This core is extended in two directions. The first introduces features necessary for the execution of functional languages, like, for example, suspensions. The second set of additional features supports backtracking and unification (resp. free logical variables) and is used to implement logic languages. Every object (suspensions, terms, logical variables, etc.) necessary for any of this features is represented by using a uniform closure technique. So, both extensions may be used simultaneously and hence allow the translation of functional logic languages.

4.1.1 JCode

We will illustrate the important points of the JCode stucture using two examples instead of an enumeration of all instructions and their precise meaning. Let us look at the translation of the function `sum` first—it is as follows:

```
FUN sum [a1]
{
  SWITCH CALL a1 OF
    TERM nil []: {RET BASIC 0;}
    TERM cons [x, xs]:
    {
      v1 := EVAL x;
      v2 := CALL sum [xs];
      v3 := add_int (v1, v2);
      RET BASIC v3;
    }
  END;
}
```

The function receives one argument, namely `a1`. Depending on the value of that argument two cases are distinguished. The first is selected if the value is `nil`, i.e. the empty list. In that case, the function returns the (basic) value `0`. In the second case, a cons-cell is decomposed into `x` and `xs`. Subsequently, the value of `x` is assigned to `v1`, the result of the (recursive) application of `sum` to the tail of the list (`xs`) is assigned to `v2`, and both are added. The result is returned.

The `CALL` behind the keyword `SWITCH` is a consequence of the machines jumping philosophy. Any object is represented as a closure and jumping through the object results in the corresponding WHNF (weak head normal form)—even if it is a suspension previously.

As a second example, we will look at the function `map`—defined as follows:

```
map f []     -> [];
map f (x:xs) -> f x : map f xs.
```

The translation results in the following JCode:

```
FUN map [a1, a2]
{
  f := a1;
  SWITCH CALL a2 OF
    TERM nil []:
    {
      v1 := create_term (nil, []);
      RET TERM nil v1;
    }
    TERM cons [x, xs]:
    {
      v3 := SUSP [f, x] {JUMP f [x];};
      v4 := SUSP [f, xs] {JUMP map [f, xs];};
      v5 := create_term (cons, [v3, v4]);
      RET TERM cons v5;
    }
  END;
}
```

In the case of the empty list, a new term object is created using the function `create_term`—it is a `nil` object containing no arguments. In the second case, two suspensions are created—both contain a tail call. Further, a new cons-cell object is created which receives the two suspensions as arguments.

4.2 Extending JCode

In the following we restrict ourselves to the conventional core plus the functional extensions. We don't need a fully fledged functional logic machine, but we will reintroduce some things, like, for example, unification.

Other than that, all we need to express a program written in the languages from section 2 into JCode are residuation objects (R-objects for short) and two process management instructions. This leads us to J*Code.

4.2.1 Residuation objects

R-objects are used to represent the residuals introduced within constraint lists. These objects are very similiar to the free logic variables from the JUMP-machine extension used to implement logic languages, since they enjoy a special treatment as soon as an unbound R-object is evaluated (in a functional context). But in contrast to free logical variables, this causes no runtime error leading to abnormal program termination (or the creation of a choice point in the case of a logical context). Instead, the process trying to evaluate such an object is supended until the object is bound. Then normal execution resumes. So, the evaluating process never notices that it encountered an unbound R-object. To implement this mechanism, every R-object needs some kind of list containing the processes waiting for it to be bound.

To create R-objects, we introduce a new kind of expression, namely `create_res ()`. It returns a reference to the new R-object. To evaluate simple constraints, we also need an unification procedure binding R-objects that are part of the two terms being unified. In contrast to the unification of the logic extension, no backtracking is done if an unification fails—instead a runtime error occurs. The JCode unification procedure is called `unify`.

4.2.2 Creating processes

The code necessary to compute a constraint is enclosed in a `FORK`-instruction. Such an instruction consists of the keyword `FORK` followed by a list of the forked codes free variables and the forked code itself. It results in the creation of a new process which computes the given code in parallel to all the other processes.

Communication between the different processes is established via the data accessed through the free variables of the processes codes where R-objects make it possible for a process to return data back to its parent, i.e. the process that forked it.

To signal the termination of a process, the instruction `DIE` is used. A process encountering this instruction stops to exist at onces.

4.2.3 Some examples

To illustrate the use of the extended JCode, we will present the translation of two of the above example functions. The first one is `farm`:

```
FUN farm [a1, a2]
{
  f := a1;
  SWITCH CALL a2 OF
    TERM nil []:
    {
      v1 := create_term (nil, []);
      RET TERM nil v1;
    }
    TERM cons [x, xs]:
    {
      x' := create_res ();
      FORK [f, x, x']
      {
        v2 := SUSP [f, x] {JUMP f [x];};
        unify (x', v2);
        DIE;
      };
      v3 := SUSP [f, xs] {JUMP farm [f, xs];};
      v4 := create_term (cons, [x', v3]);
      RET TERM cons v4;
    }
  END;
}
```

The interesting part is the second case of the `SWITCH`-instruction. It corresponds to the source text fragment `[x'|farm f xs] <<x': x' = f x>>`. According with the source text, a new residuation object is created and the constraint is forked as a parallel process. After that, the expression is evaluated, i.e. a new cons-cell is created. Within the forked process the application of `f` is suspended and unified with the previously created R-object. It is not possible to simply bind the suspension to the R-object because it may be that the parent process (that forked the constraint process) is faster and binds the R-object first. On a first glance, the instant binding of the suspended application to the R-object may look unreasonable, but we will see in the next subsection that it is the desired behaviour.

Our second example is the function `mapc`. A straight forward translation is as follows:

```
FUN mapc [a1, a2]
{
  c := a1;
  SWITCH EVAL a2 OF
    TERM nil: {DIE;}
    TERM cons [x, xs]:
    {
      FORK [c, x] {JUMP c [x];};
      FORK [c, xs] {JUMP mapc [c, xs];};
      DIE;
```

```
    }
  END;
}
```

Here two new processes are created within every recursion. The useful one of both computes the application of the constraint function c to an element of the list. The other one only initiates the next level of recursion. After forking both processes the parent process dies at once. This is not very efficient—but a simple optimization leads to the following code:

```
FUN mapc [a1, a2]
{
  c := a1;
  SWITCH EVAL a2 OF
    TERM nil: {DIE;}
    TERM cons [x, xs]:
    {
      FORK [c, x] {JUMP c [x];};
      JUMP mapc [c, xs];}
    }
  END;
}
```

Here the process doomed to die is reused for the code original assigned to a new process within the second FORK-instruction. The FORK- and the DIE-instruction are amalgamated. This can be seen as some kind of tail call—a *tail fork*.

4.3 Implementing J*Code

To implement the J*Code the original JUMP-machine has to be seriously modified. In the following we will motivate some of the major modifications using the example functions `farm` and `sum` definied above.

4.3.1 Lazy communication and communication objects

Lets have a closer look at the code of the function `farm`—in particular at the translation of the constraint expression `<<x': x' = f x>>`:

```
x' := create_res ();
FORK [f, x, x']
{
  v2 := SUSP [f, x] {JUMP f [x];};
  unify (x', v2);
  DIE;
};
```

After the creation of a new R-object, a process is forked. This process creates a suspension of the expression `f x` and binds that suspension to the R-object. This behaviour is correct with regard to the non-strict semantics of the source language, but it seems unreasonable to create a new process on another processor, only to create a suspension.

To evaluate that behaviour further, we have to look at the interior of the JUMP*-machine. Generally, a forked process accesses some environment variables, i.e. variables free in the constraint expression. In the above example, this are the variables `f`, `x`, and `x'`. If the process is forked on another processor, there has to be a mechanism to access data located on a remote processor (in the example the processor of the parent process). This is done via so called *communication objects* (C-objects for short). There are two different kinds of C-objects, namely *target C-objects* and *source C-objects*. A pair of a target object and a source object are together used as a one-way communication channel that is able to transfer data from the source to the target. The communication is initiated as soon as the value of the target object is requested, i.e. a jump through the code of the target object is executed. By convention of the

JUMP*-machine, jumping through an objects code is a request to return the WHNF, i.e. the "value" of that object. Hence, it is sufficient to communicate a WHNF, i.e. a term node or a basic value. If a term node has one or more arguments, new C-objects are created for them. As soon as that WHNF is received by the target object, the target is overwritten with the WHNF (or an indirection node referencing the WHNF). This mechanism guaranties that only those parts of a data structure are communicated that are really needed and further that a WHNF is communicated only once (to a given processor). We call this *lazy communication*. Using an object pair for the communication minimizes the size of the messages since no return adress has to be transmitted (every C-objects knows its partner).

Lets return to the process forked in the `farm` example. As already observed, a newly created suspension is bound to the R-object `x'`. If the process is forked to another processor, the binding is established via a pair of C-objects—source object on the processor where the suspension was created. As soon as the code of the target C-object is executed, an execution of the suspension is forced on the remote processor to obtain its WHNF. This may be called a *remote jump* (in accordance with remote procedure calls). Hence, the remote processor evaluates the data structure resulting from the application of `f` to the argument `x` step by step as it is requested.

4.4 Split code

Lets have a closer look at the communication requests from the view point of the requesting process. To continue the `farm`-example, we will examine the following expression: `sum (farm f l)` where `f` is an expensive function (in terms of time) and `l` a list. Speaking operationally, the evaluation of `(farm f l)` is suspended and passed as a parameter to `sum`. Here (again) the JCode for the function `sum`:

```
FUN sum [a1]
{
  SWITCH CALL a1 OF
    TERM nil []: {RET BASIC 0;}
    TERM cons [x, xs]:
    {
      v1 := EVAL x;
      v2 := CALL sum [xs];
      v3 := add_int (v1, v2);
      RET BASIC v3;
    }
  END;
}
```

The expression `CALL a1` forces that the passed suspension is evaluated to WHNF. Thereby, the function `farm` creates a new R-object and forks a process applying `f` to the first element of the list `l`. In addition, a cons-cell is created. Hence, the second case in `sum`s `SWITCH`-instruction is selected, `x` is bound to the new R-object, and `xs` references a suspension to the next recursion step in the evalauation of `farm`. Unfortunatly, the next instruction, `v1 := EVAL x`, suspends the whole computation of `sum` until the first application of `f` is evaluated. As a result, the function `farm` won't compute the applications of `f` to the list elements in parallel, but one after the other.

Looking at the data dependencies within the three assignments in the second case of `sum`s `SWITCH`-instruction, it is obvious that the WHNF of `x` is requested within the first assignment, but it is not needed until the `add_int (v1, v2)` is executed. Hence, we may execute the instruction `v2 := CALL sum [xs]` while waiting for the WHNF of `x`. A code fragment that may be executed while waiting for a communication to be finished is called *split code*. It is easy to calculate the split code of an instruction on the JCode level, using the data dependencies of the variables. Split code is calculated for all `EVAL`- and `CALL`-instructions. These are the only instructions that may initiate a communication. The execution of the split code is realized through a continuation vector technique. The same technique is used in the original JUMP-machine to descriminate logical variables from "normal" terms.

In our above example the use of `v2 := CALL sum [xs]` as a split code leads to a recursive traversal of the list `l`. For every list element, `farm` forks a process applying `f` to that element and creates a residual leading to a communication channel for the result of the forked process. When the last process is forked `sum` suspends until that process finished its computatation. After that, `sum` is only suspended a second time if any of the forked processes needs clearly more time than the last one. With that, we are able to implement the behaviour intended by the constraint functional specification.

5 Conclusion

We introduced an extension of standard lazy functional languages with residuals and constraints. Such a language is well suited for the specification of parallel algorithms because it allows some abstract kind of granularity control and process communication. We further showed how to extend the JUMP-machine to be able to execute such a language in parallel and presented some important implementation issues of the new JUMP*-machine.

5.1 Related work

The I-structures of Id Nouveau [ANP89, NPA86] are very similiar to the residuals. But Id Nouveau uses them within a special form of array. There is no explicit granularity control in Id Nouveau. The goals followed in the Id Nouveau implementation with the Threaded Abstract Machine (TAM) shares [CSS+91] some fundamental ideas with the JUMP*-machine, but the implementation techniques are quite different.

A sequential implementation of residuation exists for the language LIFE [AKMV92], which also combines residuals with constraints.

An idea very similiar to residuation are the *futures* or *promises* used in LISP to spawn a parallel computation and simultaneously construct a fill-in for the result within a data structure.

5.2 Further work

Currently, we are working at the low level mechanisms that are used to implement J*Code. Thereafter, we will start to implement a compiler for a constraint functional language. The compiler should be able to produce code for sequential hardware and for parallel MIMD-machines. Following our tradition our goal will be the direct generation of assembly code.

5.3 Acknowledgements

Hendrik C.R. Lock was a great help through a couple of inspiring email discussions and IRC[2]-sessions.

References

[Ach91] Peter Achten. Annotations for load distribution. In *3nd International Workshop on Parallel Implementation of Functional Languages*, May 1991. University of Southampton.

[AKMV92] Hassan Aït-Kaci, Richard Meyer, and Peter Van Roy. *Wild LIFE, a user manual*, May 1992. draft version.

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 255–274, Aug 1991. LNCS 528.

[ANP89] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages*, pages 598–632, Oct 1989.

[Cha91] Manuel M.T. Chakravarty. Entwicklung und Bau eines Übersetzers für eine funktional logische Programmiersprache. Master's thesis, University of Karlsruhe, Oct 1991.

[2] IRC is the Internet Relay Chat program

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

[CSS+91] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *4th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, Apr 1991.

[HKL92] Guido Hogen, Andreas Kindler, and Rita Loogen. Automatic parallelization of lazy functional programs. In *ESOP '92*, 1992.

[HP90] K. Hammond and S.L. Peyton Jones. Some early experiments on the grip parallel reducer. In *2nd International Workshop on Parallel Implementation of Functional Languages*, 1990. University of Nijmegen.

[JPP91] Radha Jagadeesan, Keshav Pingali, and Prakash Panangaden. A fully abstract semantics for a first-order functional language with logic variables. *ACM Transactions on Programming Languages and Systems*, 13(4):577–625, Oct 1991.

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

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

[NPA86] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical report, MIT Laboratory for Computer Science, 1986. CSG Memo 265.

[NPS91] Eric Nöcker, Rinus Plasmeijer, and Sjaak Smetsers. The parallel ABC machine. In *3nd International Workshop on Parallel Implementation of Functional Languages*, Jun 1991. University of Southampton.