Adding Function Arguments to Moded Guarded Definite Clauses

Huntbach, Matthew

Department of Computer Science

Research Report No. RR-03-01 ISSN 1470-5559 April 2003

Adding Function Arguments to Moded Guarded Definite Clauses

Matthew Huntbach

Adding Function Arguments to Moded Guarded Definite Clauses

Matthew Huntbach

Department of Computer Science, Queen Mary University of London
mmh@dcs.qmul.ac.uk

Abstract. Guarded Definite Clauses is a simple programming language which originated from attempts to introduce concurrency into the logic programming paradigm. Moded GDC introduces a new syntax for GDC which ensures all programs are well-moded and makes the moding visually apparent. A well-moded GDC program is one where a logic variable can be considered a channel with a direction of data flow, ensuring variables are guaranteed a single writer but may have multiple readers. We introduce a type system which guarantees that some variables are linear (single reader as well as single writer), allowing the polarity of data flow to be reversible and thus giving easy process interaction. We then show how certain patterns of communication can be viewed as higher-order functions, and introduce syntactic sugar which enable moded GDC programs to be written in a style which resembles conventional functional programming.

29 April 2003

Guarded Definite Clauses

Guarded Definite Clauses [Hu&Ri 99], or GDC (also known as Flat Guarded Horn Clauses, or Flat GHC [Ueda 01]) works as follows. We have a fixed global collection of process descriptors, and a global set of variables to which new variables may be added. The variables are single-assignment: once given a value, that value may never be changed or withdrawn. However a variable will initially be unassigned. A variable may be assigned a constant, or a tuple of values. The arguments to the tuple consist of constants, variables or further tuples. When a variable is assigned a tuple containing variables, the variables themselves need not have been assigned values.

A program consists of a set of processes, each of which has access to a subset of the global collection of variables and a name which associates it with one of the global set of process descriptors. A process descriptor consists of a parameter list and fixed set of rules (or clauses). Each rule has a left-hand side (or guard) and a right-hand side (or body). When a process is created, the variables in its parameter list are set to variables from the global set, or to constants or to tuples. The left-hand side of each rule consists of conditions on the variables the process has access to. The right-hand side consists of a collection of new process creations and variables assignments.

At any point in the computation, a rule may either be applicable, inapplicable, or undetermined for a process. A rule is applicable if the process's variables are sufficiently bound to be able to say that the conditions definitely hold. A rule is inapplicable if the process's variables are sufficiently bound to say that the conditions definitely do not hold. A rule is undetermined if it would require further binding of the process's variables to say whether it was applicable or inapplicable. At any point in computation, if one of the rules of a process is applicable, the process may rewrite to the collection of goals and assignments on the rule's right-hand side. If all of a process's rules are inapplicable, the process fails. If it has no applicable rule but at least one undetermined, a process will suspend until assignments from other processes make either one rule applicable or all rules inapplicable. If more than one rule is applicable, any of the applicable rules may be chosen indeterminately to rewrite. When a process rewrites, any variables local to its body are considered new variables, and are added to the global set of variables.

A more formal description of the operational semantics of GDC is given by Ueda in [Ueda 01]. It can also be considered as a specialisation, using a Herbrand constraint system, of Saraswat's concurrent constraint programming, whose semantic foundations are described in [Sara 91]. The left-hand sides of rules are asks to the constraint handler in Saraswat's model, while the assignments in the right-hand sides are tells. We refer to the operation which tests whether a particular variable has a particular value (and suspends if it is not sufficiently bound to tell) as an “ask”, and the operation which assigns a value to a variable as a “tell”, to reflect the constraint programming nature of GDC. A “comparison ask” tests whether two variables have values in a particular order and suspends if either is unbound.

A Modified Syntax for GDC

The traditional syntax for GDC is Prolog-like, reflecting its origin in attempts to parallelise the logic programming model pioneered by Prolog. Here, for example, is code for a non-deterministic merge of two lists:

```
merge([H|T],L,M) :- M=[H|M1], merge(T,L,M1).
merge(L,[H|T],M) :- M=[H|M1], merge(L,T,M).
merge([],L,M) :- M=L.
merge(L,[],M) :- M=L.
```

The only difference between this and Prolog code is the separation of the ask and tell, which in Prolog is combined in unification. The first clause above in Prolog would be:

```
merge([H|T],L,[H|M1]) :- merge(T,L,M1).
```

However, clause selection in GDC involves pattern matching but no assignment to variables in the goal, so the variable assignment is made explicit in the body. Also, Prolog tries the clauses in order, but there is neither clause order nor goal order in GDC. In GDC, a goal `merge(S1,[a|S2],S3)` matches the head of the second clause above, but `S1` is insufficiently bound to match the head of the first clause. Prolog would bind `S1` to `[H|T]` and use the first clause, GDC would resolve using the second clause. The result in GDC in this case is to pass the constant ‘`a`’ on the output list possibly before anything else about the input lists is known: other processes may be producing them concurrently. Thus the result is concurrent stream merger.

Prolog execution is a search for a set of constraints on the variables in an initial goal. If an attempt is made to assign an inconsistent constraint, that is binding a variable to one value when it has already been bound to another, execution backtracks and constraints are removed. There is no such backtracking in GDC, execution consists of monotonically binding the global set of variables. As a consequence of this execution model, the arguments to a GDC process are almost always divided into those which are intended to be used for input and those intended for use for output [Ueda 96].

Clauses in the Prolog-like representation of GDC take the form

```
Head :- Guard | Body.
```

where `Guard` and `Body` are a conjunction of goals. The guard must consist only of system tests, and is often empty, the separating bar is not written if it is empty. In our description, following Saraswat, the pattern matching of head arguments should be considered part of the guard. So where a clause is written in Prolog-like representation $\mathsf{name}(P_1,\ldots P_m)$ `:- Guard | Body`, we have a standard set of variable names for the head arguments $V_1,\ldots,V_m$. For each clause, if P_i is a variable, we replace it by V_i in the entire clause. Otherwise, we replace P_i by V_i in the head and add $V_i{=}P_i$ to the guard, asking whether the variable V_i refers to has been set to a value which matches P_i. Now each clause has an identical head.

Much work has gone into analysing GDC programs (see [Ueda 01] and references therein) to determine the modes of arguments to processes, but we propose to eliminate the necessity for that by building in the modes as part of the syntax. We propose therefore, that a process call should take the form $\mathtt{name}(A_1,\ldots,A_m)\mathtt{->}(B_1,\ldots,B_n)$, where $A_1,\ldots,A_m$ are the input arguments, and $B_1,\ldots,B_n$ the output arguments, with $m,n\geq 0$. The second set of brackets is omitted if n is 1 or 0, and the `->` is also omitted if n is 0. The output arguments must be variables, but the input arguments may be variables, constant or tuples. This makes the pattern of data flow in GDC code visually apparent.

Since each clause has an identical head in our representation, the head can be written just once with the arguments divided into input and output, we write it as $\mathtt{\#name}(I_1,\ldots,I_m)\mathtt{->}(O_1,\ldots,O_n)$, where both input and output arguments must be variables. The result is to reduce the verbosity of the code, making it clearer visually what conditions are required for a rule to fire. We make a few other cosmetic changes. This gives the following representation for the stream merger program above:

```
#merge(l1,l2)->m
{
 l1=h:t || m<-h:m1, merge(t,l2)->m1;
 l2=h:t || m<-h:m1, merge(l1,t)->m1;
 l1=[]  || m<-l2;
 l2=[]  || m<-l1
}
```

Rules are represented as a set with braces as delimiters, and semicolons as separators. A double bar separates the two parts of a rule. We have abandoned the requirement, inherited from Prolog, that variables must have an initial capital letter. If we want to assign the constant `'b'` to the variable `a`, `a=b` in the rule body does this, while a<-b assigns the variable `b` to the variable `a`. We represent list patterns by `h:t` rather than Prolog's `[H|T]`.

The above is a legal program in the language Aldwych [Hunt 03] which we have devised. Full Aldwych, however, involves more extensive syntactic sugar which enables stream merger to be written in a much more concise way than above. We can regard the relationship between GDC and Aldwych as similar to the relationship between lambda-calculus and a functional programming language like Haskell.

A Mode System for GDC

An ask may only ask the value of one of the input arguments to the process. Given a process heading `#name(`I_1`,…,`I_m`)->(`O_1`,…,`O_n`)`, an ask of the form I_k`=pat`, where $1 \le k \le m$, may be used in a clause guard, but not of the form O_k`=pat` for $0 \le k \le n$. Here `pat` is a pattern, which may either be a constant, or a tuple with a functor name and arguments `funct(`I_{m+1}`,…,`I_{m+p}`)` $p \ge 1$, treating `h:t` as equivalent to `:(h,t)` with `:` the functor name. A pattern may be considered as introducing additional input arguments to a clause. Any input variable either from the header of the process or from a tuple in the guard may not be used either in an output position of a new process creation in the body, or as a variable which is assigned a value in the body. An input variable may be used any number of times, including zero, in an input occurrence in the body of a clause. An input occurrence is either an argument in an input position of a new process creation, a variable which is assigned to an output variable, or an argument to a tuple which is assigned to an output variable or used as an argument in a new process creation.

Each output variable must be used exactly once in an output occurrence in the body of each clause. An output occurrence is either an output position as an argument in a new process creation, or being assigned a value. An output variable may also be used any number of times in an input occurrence.

A new variable is one that comes neither from the header of the process nor from a tuple in an ask. It is used for local communication with the processes created in a rule body. It must be in exactly one output occurrence in a rule body, and may be in any number of input occurrences. This represents the actual creation of a new variable, which is added to the global set of variables. Variables local to a clause through introduction in a tuple in an ask, are not new variables, they are local names for existing variables. It can be seen from this that although variables are conceptually global, in fact they are localised to a small number of processes. Access to a local variable is only passed out if it is assigned directly or through a tuple to another output variable. To illustrate this, consider:

```
#p(a,b)->c
{
 || r(a)->v, q(b,v)->c
}
```

Since the guard to the one clause for a `p` process is empty, the process will always transform to a `q` process and an `r` process with `v` as a variable communicating between them but not accessible by any other process. Now consider

```
#p(a,b)->c
{
 || r(a)->v, q(b,v)->u, c=f(u)
}
```

Although `u` is introduced as a local variable, the output variable `c` is bound to the one-argument tuple `f(u)`, resulting in read access to `u` passing to whichever processes have read access to `c`.

The effect of the moding is to give variables a single-writer multiple-reader property. The firing of any rule will either create an assignment for variables it has write access to or pass the write access to exactly one new process. In the first version of `p` above, write access to `c` is passed to `q`, in the second version `c` is assigned a value and write access to it disappears. Multiple read access to a variable is created when it has more than one read occurrence in the body of a clause, for example in

```
#p(a,b)->c
{
 || r(a)->v, q(a,v)->c
}
```

Now both the `r` and `q` process have read access to `a`. This also shows how an input variable need have no input occurrences, as that is the case with `b` here.

Above, we suggested the arguments to a tuple in an ask would be variables. These variables may be used in further asks in the same guard. For example, we could have I_k=funct1(I_{m+1},…,I_{m+p}),I_{m+j}=funct2(I_{m+p+1},…,I_{m+p+q}). An alternative way of writing this is to replace I_{m+j} with funct2(I_{m+p+1},…,I_{m+p+q}) in the argument list I_{m+1},…,I_{m+p}. Although these two representations are regarded as equivalent, the former can be used if it is required to give a name in the clause body to the `j`th argument of the functor variable I_k has been assigned.

Back-communication through Linear Variables

An important aspect of GDC is back communication [Greg 87]. This means a process binding a variable to a tuple, which is then read by another process which binds some of the variables within that tuple. This enables easy two-way interaction between processes. We could extend our syntax for moded GDC by separating arguments to a tuple in a similar way to that we used to separate arguments for a process. So v=funct(I_{m+1},…,I_{m+p})->(O_{n+1},…,O_{n+q}) with $p,q\geq0$, in the guard is an ask which tests whether `v` has been bound to a tuple with functor name `funct` and arguments I_{m+1},…,I_{m+p},O_{n+1},…,O_{n+q} where O_{n+1},…,O_{n+q} are reply variables. The reply variables are treated as output variables in the body of the clause, that is, they must have exactly one output occurrence, and can have any number of input occurrences. As previously, we have assumed the input arguments are variables, they are then treated as input variables for the rule, that is they may only be used in input occurrences. An input argument may, however, be a pattern, it is then treated as if it were a variable with a separate ask for the variable to be the given pattern. Note this means the input arguments may contain further output variables. If `p` is 0, the above ask can be written v=funct->(O_{n+1},…,O_{n+q}), if `q` is 1, the brackets around the output variables can be omitted.

The assignment which causes a variable to be bound so that the ask v=funct(I_{m+1},…,I_{m+p})->(O_{n+1},…,O_{n+q}) becomes applicable is u=funct(A_1,…,A_p)->(B_1,…,B_q), This counts as an input occurrence of the variable for any A_i which is a variable, and an output occurrence for each B_j which must be a variable. For any A_i which is not a pattern, it is equivalent to the `i`th input argument being a new variable, and that variable assigned the pattern A_i separately.

There is a problem with this: it leads to the guarantee that every variable has a single writer being broken. We have shown how an input argument to a process may be duplicated, leading to multiple readers of the variable which the argument represents, or may not occur at all in the body of a clause. Suppose we have `r(a)->v,q(b,v)->`c where `r` binds `v` to a tuple with reply variables. Then a clause for `q` which causes its second argument to occur in more than one read occurrence would provide multiple writers to the reply variables, since any process that has read access to v could ask for its value and hence have write access to its reply variables. A clause for `q` which does not pass its second argument to any read occurrence would mean the reply variables to `v` would lose their writer and would never get assigned values.

We resolve this problem by having two types for variables, linear and non-linear. A linear variable is guaranteed to have exactly one reader and one writer. Non-linear variables may only be assigned tuples that have no reply variables, and no linear variables or tuples containing linear variables as arguments. A linear variable which is an input to a clause, either as one of the arguments to the process, or as an argument to a tuple in an ask, must occur just once in the clause either to have its value asked in the guard, or in an input occurrence in the body. A linear variable which is an output to a clause, either as one of the arguments to the process or as a reply variable to a tuple in an ask, must occur exactly once in an output occurrence only in the clause. A linear variable which is local to a clause body must be in exactly one read occurrence and one write occurrence only. Process headers will indicate which arguments must be non-linear variables, and patterns in asks will indicate which arguments to tuples must be non-linear. An argument indicated as non-linear will fail to match with a tuple that contains linear arguments or reply variables when asks are made.

Here is an example of a simple GDC process which involves back-communication:

```
#vendingMachine(cost,in)
{
 in=pay(n)->(item,change), n>=cost ||
    item=chocolate, change<-n-cost;
 in=pay(n)->(item,change), n<cost  ||
    item=nothing, change<-n
}
```

A process created by `vendingMachine(c,`*`link`*`)` will suspend until both `c` and *`link`* have become bound. Italics are used to indicate linear variables. The variable *`link`* needs to be bound to a tuple with functor name `pay` and three arguments. The second and third arguments are reply arguments. This is indicated by the pattern `pay(n)->(item,change)`. The first argument of the tuple *`link`* and the second argument of the process need to be bound to numerical values which can be compared using a comparison ask.

Higher Order Functions in GDC

One of the main arguments given in favour of functional programming, for example in Hughes' influential paper [Hugh 89], is that higher order functions, which they provide with ease, are a good way to modularise and develop programs. GDC handles interaction between components with ease due to back-communication, and it handles indeterminacy with ease. Both of these may be handled only awkwardly in functional programming, the monadic approach [Peyt 00] being favoured as a way of handling them without compromising the purity of functional programming. However GDC does not give a way of passing process descriptors as arguments, and constructing new ones and binding variables to refer to them, so it seems to lack the crucial tool of higher order programming.

Higher-order logic programming has been considered, at one end of the scale in terms of a formal extension to logic programming theory [Na&Mi 98], at the other in terms of various ad hoc extensions for practical programming [War 82]. We believe, however, that back-communication in concurrent logic programming can be used for much of what requires higher-order functional programming in functional programming.

We represent a function by a linear variable with write access. Assigning to that variable a tuple with one input argument and one output argument causes the output argument to be bound to the function of the input argument. Using an empty functor name, we can write the function `square` as:

```
#square(in)
{
 in=(arg)->result || result<-arg*arg
}
```

The simplest higher order function just takes a function and an argument and gives the result of applying the function to the argument, so:

```
#apply(arg)->(func,result)
{
 || func=(arg)->result
}
```

So the conjunction of calls `square(f),apply(x)->(f,y)` will assign the square of `x` to `y`. The first reply argument to `apply` could be set to a linear variable which is linked to a process representing any function, so long as the function does not return a linear variable. For example, we could have:

```
#addOne(in)
{
 in=(arg)->result || result<-arg+1
}
```

Then the call `apply(x)->(f,y)` will assign to `y` the sum of `x` and 1, providing the single reader of *f* is `addone(f)`. We have shown that reference to a particular function can be abstracted out, to give a general process descriptor which can be specialised to refer to a particular function through an argument.

The following process will gives the effect of function composition:

```
#compose(h)->(f,g)
{
 h=(arg)->res || f=(arg)->res1, g=(res1)->res
}
```

Now

`square(sq),addOne(inc),compose(h)->(sq,inc),apply(x)->(h,y)`

will cause `y` to be assigned the value of x^2+1.

We can also show currying, for example, in an addition function:

```
#add(in)
{
 in=(arg,res) || addn(arg,res)
}

#addn(n,in)
{
 in=(arg)->res || res<-n+arg
}
```

We need a different form of apply to give the effect of a function which returns a function when applied to an argument:

```
#applyf(arg,result)->func
{
 || func=(arg,result)
}
```

Now `add(f1),applyf(x,f2)->f1` means that `f2` represents the function of adding `x` to its argument. We could have

`square(f3),compose(f4)->(f2,f3),apply(y)->(f4,z)`

to set `z` to $(x+y)^2$.

There are two big objections to the above. Firstly, the moding is counterintuitive. To give the effect of a process taking a function as an argument, it has to give a variable representing that function as one of its outputs, while a process which is thought of as producing a function takes a variable representing it as one of its inputs. Secondly, each function can only be used once. We might want

`square(sq),apply(x)->(sq,z),apply(z)->(sq,y)`

to set `y` to x^4, but it would be rejected as illegal due to the variable `sq` having more than one output occurrences.

Let us deal with the second problem first. A simple solution would be just to create a separate variable for each time we want to use the function:

`square(sq),square(sq2),apply(x)->(sq1,z),apply(z)->(sq2,y)`

But this cannot be done if the function variable is a parameter, say:

```
#twice(t)->f
{
 t=(arg)->res || apply(arg)->(f,b), apply(b)->(f,res)
}
```

where we would want

```
square(sq), twice(f)->sq, apply(x)->(f,y)
```

to set `y` to `x`^{4}, but the body of the clause for `twice` is illegal.

A solution to this is to modify the way we represent functions so that a function call "returns" a variable which may be used to continue applications of the function. Given the reverse polarity of modes for linear variables used for functions this would actually be an input variable. Here is code for `square` using this convention:

```
#square(in)
{
 in=(arg,next)->result || result<-arg*arg, square(next);
 in=halt ||
}
```

The convention is that the function continuation variable is assigned the constant `'halt'` if it is not to be used. We change `apply` so that an application gives us a function continuation variable:

```
#apply(arg,cont)->(func,result)
{
 || func=(arg,cont)->result
}
```

We might also have a version of `apply` for when we know we do not need to use the function again:

```
#applyOnce(arg)->(func,result)
{
 || func=(arg,'halt')->result
}
```

although the same would be achieved by the call `apply(arg,'halt')->(func,result)`. So this gives us a version of `twice` where we don't want continued use of the function:

```
#twice(t)->f
{
 t=(arg)->res ||
   apply(arg,cont)->(f,b), apply(b,'halt')->(cont,res)
}
```

Similarly, the following gives us a process which maps a function onto a list:

```
#map(list)->(func,result)
{
 list=[] || func=halt;
 list=h:t ||
   apply(h,cont)->(func,h1),map(t)->(cont,t1),result=h1:t1
}
```

Consider also mapping a function onto a tree, where a tree is either the tuple `tree(l,r)` where `l` and `r` are trees, or the tuple `leaf(n)`:

```
#mapTree(tree)->(func,result)
{
 || map1(tree,'halt')->(func,result)
}

#map1(tree,cont)->(func,result)
{
 tree=leaf(n) || apply(n,cont)->(func,n1), result=leaf(n1);
 tree=tree(l,r) ||
    map1(l,func1)->(func,l1),
    map1(r,cont)->(func1,r1),
    result=tree(l1,r1)
}
```

Note that `apply(n,`*`cont`*`)->(`*`func`*`,n1)` in the first clause could be written as just *`func`*`=(n,`*`cont`*`)->n1`. But suppose we write it as *`func`*`=(n)->n1:`*`cont`* and use `[]` instead of `'halt'` where `:` is the infix list functor and has lower precedence than `->`. This gives us:

```
#mapTree(tree)->(func,result)
{
 || map1(tree,[])->(func,result)
}

#map1(tree,cont)->(func,result)
{
 tree=leaf(n) || func=(n)->n1:cont, result=leaf(n1);
 tree=tree(l,r) ||
    map1(l,func1)->(func,l1),
    map1(r,cont)->(func1,r1),
    result=tree(l1,r1)
}
```

Then, for example, the function square would be represented by:

```
#square(in)
{
 in=(arg)->result:next || result<-arg*arg, square(next);
 in=[] ||
}
```

Now it can be seen that in `map1(t,`*`cont`*`)->(`*`func`*`,t1)`, the linear variables *`cont`* and *`func`* form a difference list [Cl&Ta 77], where components of the difference list are tuples of the form `(n)->n1`. The difference list would conventionally be written *`func-cont`*.

The logic programming convention of the difference list `L1-L2` is that `L2` is a suffix of `L1`, and the list represents the conventional list of those elements in the list `L1` is bound to up to the start of `L2`. It is a useful convention because given two difference lists `L1-L2` and `L2-L3`, the two lists appended are `L1-L3`, there is no overhead of an append operation. If `L2` is unassigned, it enables items to be added to the end of the list. Binding `L2` to `[x|L3]` means the difference list `L1-L3` represents the list represented by `L1-L2` with `x` added to the end. If `L1` is unassigned, binding `L1` to `[x|L3]` can be considered as adding `x` to the front of the difference list `L1-L2` and making `L3` a new insertion point, further items can be added into the list is closed off by binding the insertion point variable to `L2`.

In a concurrent setting, if we have `n` processes with access to difference lists `L1-L2`, `L2-L3`, …, `Ln-[]`, we can consider this as the processes creating an ordered list `L1` without any ordering required on when an individual process adds an item to the list. A process with difference list `La-Lb` adds nothing to the overall list by binding `La` to `Lb` It adds `x` to the overall list by binding `La` to `[x|Lc]` and passing the difference list `Lc-Lb` for further insertions. If it passes `La-Lc` to process `p` and `Lc-Lb` to process `q` it joins `p` and `q` in the chain of processes adding to the list, with `p`'s additions coming before `q`'s.

This is in effect how we are representing the function in the tree map above and how functions can be represented generally. The process which outputs a function actually inputs a list of calls *`l`* of the form `(arg)->result`. It is guaranteed sole access to this list due to linearity, so it can always write `result` to the appropriate value with respect to `arg`. What is considered as the input of a function actually involved the input of a linear variable *`cont`* and the output of a linear variable *`func`*, with *`func-cont`* being a portion of the chain of difference lists which together form *`l`*.

We could introduce syntactic sugar for this into our moded GDC which would give a syntax which appears to have functions as arguments. This is done as follows. We introduce a separate function type of variable and argument, indicated by an initial capital letter. A function variable must have a single writer, but may have multiple readers. Assume for now this means in any process which has write access to a function variable, exactly one output occurrence of the function variable must occur in each clause. In any process which has read access to a function variable, the variable may only occur in input occurrences in the clauses, and may occur any number of times including zero. A new function variable may be introduced in a clause body, in which case it must have one writer and may have any number of readers. A function variable may not be an argument to a tuple which is assigned to a non-linear variable, and its value may not be asked in a guard.

A process heading with functional variable inputs is converted to one without by changing each input functional variable `F` to a pair of linear variable, *`f`* as output and *`c`* as input. Thus

`#proc(`I_1`,…,F,…,`I_m`)->(`O_1`,…,`O_n`)`

becomes

`#proc(`I_1`,…,`*`c`*`,…,`I_m`)->(`O_1`,…,`O_n`,`*`f`*`)`

In any clause where `F` is not used in the body, the assignment *`f<-c`* is added to the body. If `F` is passed as an input argument once in the body, but not called as a function, it is replaced by *`c`* and *`f`* added to the output arguments, the equivalent of the difference list *`f-c`*. If `F` is passed twice as an argument but not called as a function, the first occurrence of `F` is replaced by the equivalent of the difference list *`f-f1`*, the second by *`f1-c`*. And so on for more occurrences of `F`.

We use juxtaposition for function calling, as in languages like Haskell. So `F x` represents a call of the function represented by `F` with argument `x`. It is used as a value in the code, so could be an argument to a process creation or assigned to an output variable. To give ordinary moded GDC, `F x` is replaced by `y`, and `f` in the difference lists above is replaced by *`f0`*, with the additional assignment *`f`*`<-(x)->(y):`*`f0`* added to the body. If there are two calls to the function, `F x1` and `F x2`, they are replaced by `y1` and `y2`, with the extra assignment *`f`*`<-(x1)->(y1):(x2)->(y2):`*`f0`* and so on.

In any clause where a function variable is local to the clause, occurrences of the function are converted into difference lists as above with the empty list `[]` used in the place of *`c`*.

This enables us to write `mapTree` as follows:

```
#mapTree(tree,Func)->result
{
 tree=leaf(n) || result=leaf(Func n);
 tree=tree(l,r) ||
    mapTree(l,Func)->l1,
    mapTree(r,Func)->r1,
    result=tree(l1,r1)
}
```

As some additional syntactic sugar, if a process has just one output, we allow it to be anonymous. This is indicated by `<` in the header rather than `->` followed by a name. Assignment to the anonymous output is indicated by `>` followed by the value rather than the output variable name, followed by `->` followed by the value. Also, `f(`a_1`,…,`a_n`)` in an input occurrence is treated as `r` in that occurrence with an additional process `f(`a_1`,…,`a_n`)->r` added to the clause body. If we want the tuple `f(`a_1`,…,`a_n`)` in the input occurrence, we write that `=f(`a_1`,…,`a_n`)`. This cuts down on the verbosity by removing superfluous variable names. It results in following more compact version of `mapTree` being possible:

```
#mapTree(tree,Func)<
{
 tree=leaf(n)   ||>=leaf(Func n);
 tree=tree(l,r) ||>=tree(mapTree(l,Func),mapTree(r,Func))
}
```

We now need to consider how to write the processes that output function variables. These must be converted to processes that input a stream of tuples with reply variables. In our sugared syntax, we write the header of a process that produces a function variable representing function `f` with argument `x` as `#f[x]<` followed by a set of clauses *S* which have input variable `x` and assign to the anonymous output variable. This is translated into:

```
#f(s)
{
 s=[] || ;
 s=(x)->r || r<-f1(x)
}

#f1(x)<
S
```

Thus we would write the process which produces the variable representing the square function as:

```
#square[x]<
{
 ||>x*x
}
```

As more syntactic sugar, `{ || body }` (that is a single clause with an empty guard) may be written `== body`, giving us the compact representation

```
#square[x] <==> x*x
```

Higher order functions using stream merger

The difference list representation of function calls has the convenience of a minimal overhead. However, it does enforce a fixed order in which calls are listed, which can be a source of deadlock. For example, consider

```
p(F,a)->b, q(F,b)->(a,c)
```

This will become

```
p(f1,a)->(f,b), q(f2,b)->(f1,a,c)
```

where *f* is the stream of messages to the process providing the function. The difference list *f*-*f1* represents the function calls from `p`, and *f1*-*f2* the function calls from `q`. Calls from `q` will not be passed on until `p` has produced its complete list of calls, but that completion may depend on knowing the value of `a`, computed by `q`. To overcome this, first consider how the list of calls would be handled using list append rather than difference lists:

```
p(a)->(f1,b),q(b)->(f2,a,c), append(f1,f2)->f.
```

This maintains the same ordering of calls as previously, but if we replace list append with indeterminate stream merger:

```
p(a)->(f1,b), q(b)->(f2,a,c), merge(f1,f2)->f
```

calls will now be passed into the stream `f` in the order in which they are generated.

So the input of a function can be represented by the output of a stream of tuples representing calls to that function. Each reference to the function can be represented by a separate output stream, with the streams being merged together to form the one stream that is passed to the process that handles the function calls. An individual function call can be represented by putting a tuple for it on the front of the output stream, so, for example:

```
p(F,a)->b, q(F,b)->(a,c), r(F c)->d
```

is represented by:

```
p(a)->(f1,b), q(b)->(f2,a,c), r(c1)->d, merge(f1,f2)->f0,
f<-(c)->c1:f0
```

If a function is not passed to any processes, it is represented by just the list of calls made to it. So:

```
p(F a)->b, q(F b)->(a,c), r(F c)->d
```

is represented by:

```
p(a1)->(b), q(b1)->(a,c), r(c1)->d,
f<-(a)->a1:(b)->b1:(c)->c1:[]
```

This means that streams are eventually closed off. When the stream to the process handling function calls becomes the empty stream, the process can be terminated, a form of garbage collection.

Stream merger could in practice be replaced by ports as suggested by Janson et al [Jans 93]. Ports can be considered as a lower level and hence more efficient way of implementing what can be considered in the abstract as stream merger. Some of the overhead associated with the explicit use of streams may also be removed by partial evaluation [Hunt 89].

Conclusions

We have identified a simple but inherently concurrent operational model of computing. The model is implemented by the programming language Guarded Definite Clauses, but has to some extent been hidden by that language's origin from attempts to introduce parallelism into logic programming, and consequent Prolog-like syntax. In particular, in GDC there is almost always a direction of flow of data in a variable from a single source to possibly several destinations, but in the traditional syntax this is implicit. We introduce a syntax which makes it explicit, and which also rules out the possibility of programs that can fail due to attempts to bind variables to contradictory values.

Although GDC loses much of logic programming's inheritance, particularly backtracking, it remains based on the logic variable. The logic variable has been recognised as a key concept for easy programming of concurrency and interaction, hence many attempts, such as [Hanu 99], to incorporate it into other programming paradigms. A logic variable can not only be passed as part of a structure while being bound elsewhere, it can be passed as part of a structure and bound by a process that inputs that structure. Our syntax keeps this power through annotating some variables as linear and hence suitable for reversing the input-output polarity.

Finally, we show how we can use this back-communication of logic variables to overcome one of the deficits of logic programming as compared to functional programming, its lack of a higher order function capacity.

References

[Cl&Ta 77] K.L.Clark and S-A.Tärnlund. A first order theory of data and programs. In *Information Processing 77 (IFIP 77)*, B.Gilchrist (ed), Elsevier/North-Holland, 939-944.

[Greg 87]. S.Gregory. *Parallel Logic Programming in PARLOG.* Addison-Wesley.

[Hanu 99] M.Hanus. Distributed programming in a multi-paradigm declarative language. *Proc. Int. Conf. on Principles and Practice of Declarative Programming (PPDP'99) LNCS 1702*, Springer, 376-395.

[Hugh 89] J.Hughes. Why functional programming matters. *Computer Journal 32 (2)* 98-107.

[Hunt 89] M.Huntbach. Meta-interpreters and partial evaluation in Parlog. *Formal Aspects of Computing 1* 193-211.

[Hunt 03] M.Huntbach. Features of the concurrent language Aldwych. *ACM Symposium on Applied Computing (SAC'03)*, 1048-1054.

[Hu&Ri 99] M.Huntbach and G.Ringwood. Agent-Oriented Programming: from Prolog to Guarded Definite Clauses. *LNCS 1630* Springer.

[Jans 93] S.Janson, J.Montelius, and S.Haridi. Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming*, G.Agha, P.Wegner, and A.Yonezawa (eds.). MIT Press.

[Na&Mi 98] G.Nadathur and D.Miller. Higher order logic programming. In *Handbook of Logic in AI and Logic Programming, Volume 5*, Oxford University Press, D.M.Gabbay, C.J.Hogger and J.A.Robinson (eds), Oxford University Press, 499-590.

[Peyt 00] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions and foreign-language calls in Haskell. In *Engineering Theories of Software Construction*, C.A.R.Hoare, M.Broy, and R.Steinbruggen (eds) IOS Press.

[Sara91] V.A.Saraswat, M.Rinard and P.Panangaden. Semantic foundations of concurrent constraint programming. *Principles of Prog. Lang. Conf. (POPL'91)*, 333-352.

[Ueda 96] K.Ueda. Experiences with strong moding in concurrent logic/constraint programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95), LNCS 1068*, Springer, 134-153..

[Ueda 01] K.Ueda. Resource-passing concurrent programming. In *4th Int. Symp. on Theoretical Aspects of Computer Science (TACS'01), LNCS 2215*, Springer, 95-126.

[Warr 82] D.H.D.Warren. Higher order extensions to Prolog: are they needed? *Machine Intelligence 10*, 441-454.