The Core Language of Aldwych

Matthew HUNTBACH
Department of Computer Science, Queen Mary University of London

Abstract. Aldwych is a general purpose programming language which we have developed in order to provide a mechanism for practical programming which can be thought of in an inherently concurrent way. We have described Aldwych elsewhere [13] in terms of a translation to a concurrent logic language. However, it would be more accurate to describe it as translating to a simple operational language which, while able to be represented in a logic-programming like syntax, has lost much of the baggage associated with "logic programming". This language is only a little more complex than foundational calculi such as the pi-calculus. Its key feature is that all variables are moded with a single producer, and some are linear allowing a reversal of polarity and hence interactive communication.

Keywords. Concurrency, logic programming, linear variables, single-assignment variables.

Introduction

It has been noted since the observations of Landin [18] that a complex programming language can be understood by showing a translation into a tiny core language which captures the essential mechanisms of its programming style. This idea has been most influential in the field of functional programming languages which can be considered as just "sugared lambda-calculus". Modern computing, however, tends to be about interaction as much as calculation. An early attempt to build a programming language based on an abstract model of interaction was occam with its basis in CSP [4]. More recently, the pi-calculus [19] has received much attention as the suggested basis for a model of interactive computing. Unlike CSP, the pi-calculus is a name-passing calculus, meaning that communication channels can themselves be passed along communication channels, leading to the communication topology changing dynamically as code is executed. There have been some attempts to build languages which are "sugared pi-calculus", for example PICT [21], but even when sugared this model seems to be difficult for programmers to use practically.

We have been working on building a programming language with an abstract concurrent model which uses the concept of shared single-assignment variables rather than pi-calculus's channels. It is another name-passing calculus, since a variable may be assigned a value which is a tuple containing variables. Our work in this area sprang from earlier work in concurrent logic languages [12]. Although these languages have been proposed as practical programming languages in their own right, with an application area in parallel programming [5], our experience with them suggested they had serious defects. Firstly, their lack of structure meant it was difficult to scale them up from toy examples to large scale use. Secondly, in their attempt to emulate the logic programming style of Prolog, they led to programs where the data flow could not easily be detected. This was despite the fact that in reality programmers in them almost always had an intended mode for every variable with a single producer [29].

We considered building a programming language which compiles to an underlying concurrent logic form, but which has a rich set of "derived forms" to enable more practical programming. Although this is not a new idea (see [3] for a survey), unlike previous attempts to build logic-programming based object-oriented languages, our intention was not to "combine" object-orientation with logic programming. Rather, we felt the very simple underlying operational model of the concurrent logic languages would be a good core language for developing a richer language which made no claims itself to be logic-oriented but which enabled practical programs to be written in a style where concurrency is a natural feature rather than an awkward add-on extra.

This language is being developed under the name "Aldwych" and we describe some of its features elsewhere [15]. Early in the development of Aldwych it became clear that a key feature would be for all variables to be moded, that is with a single producer identified clearly by the syntax and one or more consumers. Another key feature was the division of variables into linear and non-linear, where non-linear variables have a single consumer as well as a single producer. This enables the consumer-producer relationship to be reversed with ease.

Many of the complexities of implementing the concurrent logic languages disappear when moding can be guaranteed, and when there is also a clear indication of which variables are linear implementation can be even more efficient [30]. Since the modes and linearity of variables in the language to which Aldwych compiles can be guaranteed, there is no need for any mechanisms to analyse it. In fact the underlying language represents such a dramatic simplification of committed choice logic languages, which in turn are a dramatic simplification of the logic programming model (see [28] for a discussion of the paring down of logic programming features or "de-evolution" of logic programming in the search for efficient concurrent implementation) that it no longer makes sense to give references to it which emphasise and elaborate on its more complex logic programming ancestry.

The purpose of this paper, therefore, is to describe the underlying operational model of the language into which Aldwych compiles in a way that does not attempt to link it to more general concepts of logic programming or describe stages in its de-evolution which are no longer relevant to its current state. The model can be described in terms of a few reduction rules. Full Aldwych is described in terms of "derived forms" which translate to the simpler model here, thus this paper complements our previous papers which describe those derived forms.

1. A Relational Language

In a conventional imperative language, the computation

```
f(g(x),y)
```

is taken as a command that the code for `g` with argument `x` is fully evaluated and gives a value which becomes the first argument to `f`. The same applies in a strict functional language. We can regard the construct as a shorthand for evaluating `g(x)`, putting the result in a variable and using that variable as the first argument for `f`:

```
z<-g(x); f(z,y)
```

where the semi-colon is a sequencing operator, all code before the semi-colon is completed before code after it is executed.

Suppose we replace the sequencing operator by one which doesn't have the property of ordering computations, let us use a comma:

```
z<-g(x), f(z,y)
```

We could now, given a suitable architecture, evaluate `g(x)` and `f(z,y)` in parallel. The variable `z` could be regarded as a "future" [9]. The computation `f(z,y)` may use it as a placeholder, passing it to other computations or incorporating it into data structures while `g(x)` is still computing its value. The computation `f(z,y)` will suspend, however, if it needs to know the actual value of `z` in order to progress.

We could flatten code out, replacing all embedded calls $g(x_1,\ldots,x_n)$ by a variable `z` with the call $z<-g(x_1,\ldots,x_n)$ occurring beforehand. If the call $z<-g(x_1,\ldots,x_n)$ is always placed before the call which has `z` as an argument, and the calls are executed in order we will obtain strict order evaluation in functional programming terms. We could however execute the calls in some other order, possibly involving some parallelism. If we use the comma operator as above, then the only necessary sequencing is that imposed by the necessity for a computation to suspend if it needs to know the value of a variable until that value has been computed.

Note that the possibility of parallelism does not imply its necessity. The extreme alternative to the conventional sequential execution of `z<-g(x), f(z,y)` is that `f(z,y)` is evaluated while `g(x)` is suspended, and `g(x)` is only evaluated if `f(z,y)` suspends due to the need to know the value of `z`. Suppose we employ the convention that the rightmost computation is picked for progression, but if it needs the value of a variable and that variable has not yet been given a value, the computation that gives that variable a value is picked, and if that one is also suspended due to the need to know the value of another variable, the computation which gives that variable a value is picked and so on. We will then obtain what is known in functional programming as "call-by-need" or "lazy evaluation". The advantage to this is that if a computation gives a value to a variable but there is no computation left which has that variable as an argument, that computation can be abandoned without evaluation.

Clark and Gregory [2] suggested a "relational language" which worked like the above description of a flattened functional language, with no suggested ordering on computations apart from that imposed by computations suspended while waiting for others to bind variables. The intention, however, was that the potential parallelism would be exploited. This language was influenced by logic programming in that the function call assignment to a variable $y<-f(x_1,\ldots,x_n)$ was written as a relation $r(x_1,\ldots,x_n,y)$. Logic programming in its Prolog form and most other varieties does not have a concept of a direction on variables, whereas in functional programming all variables have a direction with just one computation that can write to them. Clark and Gregory's relational language imposed a direction on variables by giving a mode declaration to its relations, so the above relation `r` would have mode $r(m_1,\ldots,m_n,m_{n+1})$ with m_1 to m_n being `?` meaning "input" and m_{n+1} being `^` meaning "output". Furthermore, it insisted that each variable have just one producer, though it may have many consumers. So a computation would consist of a collection of relations which shared variables, but each variable had to occur exactly once in the position in a relation which had mode declaration `^`.

The result of this directionality was that the arguments were "strong" [7]. That is, for each argument of a relation call, the argument was either completely constructed by that call (if an output argument) or by another call (if an input argument). The right to set a variable to a value remained solely with one relation call, that relation call might set the variable to a structure containing further variables, but it had either to take those variables from its own input variables or set up new relation calls to construct their values.

The result appeared to be a rather restricted syntax for first order functional programming. Lacking embedded function calls there was a proliferation of variables introduced merely to take the result of one call and make it an argument to another. The lack of facilities for higher order functions might be considered a serious weakness given the importance which many advocates of functional programming give them [11]. However, the making of all variables used for communication explicit, and the flat structure of computations with a single environment of variables, led to an attractively simple programming model. As we shall show below, it also had the advantage of being able to handle non-determinism with ease whereas this is a problem in functional programming.

2. Non-Determinism

Since the programming model does not rely on the lambda-calculus of functional programming it can cope with situations where there is more than one way of progressing a computation and the outcome will differ depending on the choice made. As an example, consider the following declaration:

```
#p(x,y)->z
{
 x=a || z=b;
 y=c || z=d;
:
      || z=e
}
```

The syntax used here is not the Prolog-like one of Clark and Gregory's relational language, but one we have developed and will describe further in this paper. It is used so that programs in this core language will be a subset of the full Aldwych language. The # is used to denote the introduction of a new procedure name (we will use this term rather than "relation"). Input and output modes are denoted by separating them in the procedure heading, so that if the heading is #p(u_1,…,u_m)->(v_1,…v_n) then u_1,…,u_m have input mode, and v_1,…,v_n have output mode; we omit the brackets around v_1,…,v_n when n is 1, and we omit -> and the brackets when n is 0.

The description of a procedure consists of a list of sets of rules. A set of rules is enclosed by braces, with a semicolon as the separator between rules, and for convenience "}:{" denoting the end of one set and the start of another may be written ";:". Each rule consists of two parts, a left-hand side (lhs) and a right-hand side (rhs). The lhs contains tests of variable values ("asks" in terminology introduced by Saraswat [23] for concurrent logic programming) and the rhs contains variable assignments (in Saraswat's terminology "tells"). So x=a, where x is a variable and a is a constant means "wait until x is given a value and test that it is a" when it is on the lhs, and "set x to a" when it is on the rhs.

The first set of rules in the procedure p above means that a call p(u,v)->w will set w to b if u gets set to a, and will set w to d if v gets set to c. If both u is set to a and v is set to c, w could be set to either b or d. A functional programming computation, whether strict or lazy, would insist that u be evaluated before proceeding either to assign b to w or to go on to test the value of v. In a parallel setting, however, we may have u and v being computed in parallel and be content to react accordingly depending on which computation finishes first without being forced to wait for the other. Having received the news that u is set to a, we could kill off the computation of v if there is no other computation that has v as an input argument, and similarly we could kill the computation of u if we receive the news that v is set to b [8].

The multiple sets of rules in our notation mean that the conditions for rules to apply can be tested sequentially if that is required. If and only if the conditions for none of the rules in the first set applies, the second set is used, and so on. In the above example there are only two sets, and the last set has a single rule with an empty lhs meaning no conditions are required for its application. So in our call p(u,v)->w if both u becomes set to something other than a, and v becomes set to something other than c, the final set of rules is used and causes w to be set to e.

If there is always a final rule set consisting of a single unconditional rule, a relation call in our notation can never fail. This contrasts with Prolog where failure due to no applicable rules is a natural part of the system and causes computation to backtrack to the point where a previous non-deterministic choice was made, and to change the choice made there. Such a backtracking may be practical in the single-processor sequential computation like Prolog, but is impractical in a concurrent or parallel system where one non-determinate choice may have sparked off several distributed computations, and impossible if the variable is linked to an effect on some physical system in the real world: the real world does not backtrack. We discuss in more detail the arguments against non-determinism combined with backtracking (termed "don't know non-determinism" [17]) in an earlier work [12], although doubt over the usefulness of automated backtracking in programming languages can be found much earlier than that [27].

3. Back Communication

Handling non-determinism is one aspect where a relational as opposed to functional approach to programming languages gives increased power, particularly in a concurrent setting. Another is the "logic variable" [10] used to provide "back communication" [7]. Building on the relational language described above, the idea here is that the input-output modes are weakened. In particular, a computation may bind a variable to a structure containing further variables, but leave a computation which inputs that structure to provide a value for some of those variables. The relational language of Clark and Gregory was developed into Parlog which provided such back communication, at the same time a number of similar languages were developed which were given the general name "committed choice logic languages" [24].

Given back communication, the mode system of the relational language broke down. Parlog's mode system applied only to the arguments of a relation at top level, and not to the individual components of structured arguments. It existed only to enable Parlog programs to have a more superficial appearance to Prolog programs where assignment to variables is done through pattern matching with clause heads. The other committed choice logic languages used explicit assignment operators to give values to variables, as did Parlog when the variables were arguments to tuples used for back communication. The languages ended up as modeless – there was no syntactic way of discovering which computation could actually assign a value to a variable, in fact the possibility of several different computations being able to bind a single variable was opened up, and handled in a variety of different ways. This then necessitated elaborate mechanisms to rediscover intended modes in code, since practice revealed that programmers almost always intended every variable to have just one computation that could write to it, and knowledge of the intended moding could greatly improve the efficiency of implementation [29].

In our notation, we extend moding to the terms of compound arguments. On the lhs we have $\texttt{x=t(i}_1\texttt{,…,i}_m\texttt{)}$ meaning a test that `x` is bound to a tuple with tag `t` and `m` arguments $\texttt{i}_1\texttt{,…,i}_m$, all of which are taken to have input mode (that is, they will be assigned by the computation which is giving a value to `x`). On the rhs $\texttt{x=t(i}_1\texttt{,…,i}_m\texttt{)}$ means that `x` is assigned a tuple with tag `t` and `m` arguments $\texttt{i}_1\texttt{,…,i}_m$, all with input mode, that is the computation which does the assignment must have $\texttt{i}_k$ as an argument or must provide another computation which gives $\texttt{i}_k$ a value for 1≤k≤m. We also allow $\texttt{x=t(i}_1\texttt{,…,i}_m\texttt{)->(o}_1\texttt{,…,o}_n\texttt{)}$ on the lhs, where $\texttt{o}_1\texttt{,…,o}_n$ are output variables, meaning that the computation which has this test must provide values for $\texttt{o}_1\texttt{,…,o}_n$ in the rhs of the rule. We allow $\texttt{x=t(i}_1\texttt{,…,i}_m\texttt{)->(o}_1\texttt{,…,o}_n\texttt{)}$ on the rhs, meaning that $\texttt{o}_1\texttt{,…,o}_n$ will be used in the rhs, but that a computation which takes in the value of `x` will give values to $\texttt{o}_1\texttt{,…,o}_n$.

As an example, consider the following:

```
#map(xs)->(ys,f)
{
 xs=cons(x,xs1) ||  f=ask(x,cont)->y,
                    map(xs1)->(ys1,cont),
                    ys=cons(y,ys1);
 xs=empty || ys=empty, f=done
}

#square(queries)
{
 queries=ask(u,cont)->v || v<-u*u, square(cont);
 queries=done ||
}
```

with the following initial computations:

```
map(list1)->(list2,stream), square(stream)
```

The result of executing this will be that a list in variable `list1` composed of tuples with tag `cons`, first argument an integer and second argument a further list (with `empty` indicating the empty list), is taken as input, and a square function is mapped onto it to produce the list in `list2`. This shows how back communication can be used to obtain a higher-order function effect. The input of a function is represented by the output of a stream of queries taking the form `ask(i,cont)->o`, where `i` is the argument to the function, `o` the result, and `cont` the rest of the stream giving further queries to the same function, or set to `done` if the function is not to be used any more. The code is not elegant, but the point is the higher order effect can be achieved within this model, and could be incorporated in to a language which is based on this model but uses derived forms to cover commonly used patterns at a more abstract level for use in practical programming.

However, this back communication leads to the problem that since a variable may occur in several input positions, if it is set to a tuple which includes output arguments, those output arguments will be become duplicated. Each of the computations which takes the tuple as an input could become a writer to its output arguments. One way of avoiding this, adopted for example in the logic programming language Janus [SKL 90], was to insist that every variable must be linear, that is occur in exactly one input position and one output position. This however acts as a considerable constraint on the power of the language, meaning that we cannot use variables as "futures" in the Multilisp way [9].

Our solution to the problem is to adopt a system which involves both modes and linearity. So arguments to a procedure or to a tuple may be one of four types: input-linear, output-linear, input-non-linear and output-non-linear. Only a linear variable may be assigned a tuple value which contains output arguments or linear arguments either input or output. A non-linear variable may only be assigned constants or tuples all of whose arguments are input-non-linear. In the above example, the arguments `f` to `map` and `queries` to `square` should be denoted as linear, as should the variable `cont` in the first rule for `map` and the first rule for `square`.

4. Computation

We can now describe our operational model in more detail. A computation in our notation consists of a set of procedure calls which take the form $p(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)$ with $m,n\geq 0$, where each i_h and o_k, $1\leq h\leq m$, $1\leq k\leq n$, are variable names, and a set of variable assignments which take the form either `v=t` or `v<-u`. In the variable assignments, `v` and `u` are variables, and `t` is a term which takes the form $s(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)$, $m,n\geq 0$, where each i_h and o_k, $1\leq h\leq m, 1\leq k\leq n$ are variable names, and `s` is a "term tag", that is an atomic value. For notational convenience in a term, if `n` is 1 the second set of brackets are omitted, if `n` is 0 the -> is also omitted, and if `m` is 0 the first set of brackets is omitted.

The moding is used to ensure that every variable occurs exactly once in an output position, where an output position is `v` in `v=t` or `v<-u`, or o_k, $1\leq k\leq n$, in $p(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)$, or o_k, $1\leq k\leq n$, in $v=s(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)$. A non-linear variable may occur in any number of input positions, but every linear variable must occur in exactly one input position, where an input position is i_k, $1\leq k\leq m$, in $p(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)$, or i_k, $1\leq k\leq m$, in $v=s(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)$, or `u` in `v<-u`.

We can regard a procedure call $p(t_1,\ldots,t_m)\text{->}(v_1,\ldots,v_n)$ where $t_1,\ldots,t_m$ are terms and $v_1,\ldots,v_n$ are variables, as a shorthand for $p(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n), i_1\text{<=}t_1,\ldots,i_m\text{<=}t_m, v_1\text{<-}o_1,\ldots,v_n\text{<-}o_n$. Here $v_k\text{<=}t_k$ is $v_k\text{<-}t_k$ if t_k is a variable, otherwise it is $v_k\text{=}t_k$. The point of this is to give each procedure call a fresh set of variables as its arguments. We also allow `v<-e`, where `e` is an arithmetic expression involving variables. Similarly, assignment to a variable of a tuple which contains non-variable arguments can be regarded as shorthand for an assignment which contains only variable arguments with separate assignments of terms to the arguments where necessary. So $u\text{=}s(t_1,\ldots,t_m)\text{->}(v_1,\ldots,v_n)$ is considered shorthand for $u\text{=}s(i_1,\ldots,i_m)\text{->}(v_1,\ldots,v_n), i_1\text{<=}t_1,\ldots,i_m\text{<=}t_m$. An output argument in a tuple or procedure call can only ever be a variable.

The first computation rule is that `v<-u,u=t` transforms to `v=t,u=t`, written:

$$\texttt{v<-u,u=t} \longrightarrow \texttt{v=t,u=t}$$

There is no concept of ordering on the computations, so `u=t,v<-u` also transforms to `v=t,u=t`. Note that if `u` is a linear variable we can say `v<-u,u=t` transforms to `v=t`, since `v<-u` is the one input occurrence of `u` and `u=t` the one output occurrence, and the variable `u` thus occurs nowhere else. If `u` is a linear variable, then `v<-u` is only allowed if v is also a linear variable, although v<-u is allowed if v is a linear variable but u is not. Since u=t where t contains linear variables and/or output positions is only allowed if u is a linear variable, this ensures that output positions of any variable and input positions of linear variables do not get duplicated.

We also have v<-u,u<-w transforms to v<-w,u<-w or

$$\texttt{v<-u,u<-w} \longrightarrow \texttt{v<-w,u<-w}$$

Similar to above, if u is linear, we can transform v<-u,u<-w to just v<-w.

Arithmetic is dealt with by the computation rule that v<-e where e is an arithmetic expression transforms to v=n when there are assignments $u_i=m_i$ for all variables in e, and replacing each variable u_i by m_i in e and evaluating e gives n.

A procedure call $p(i_1,\ldots,i_m)\texttt{->}(o_1,\ldots,o_n)$ is linked to a set of rules initially given by the procedure declaration for p, and we assume there is a universal fixed set of named procedure declarations. Each procedure call produces a new copy of these rules, where if the procedure heading is $\#p(u_1,\ldots,u_m)\texttt{->}(v_1,\ldots,v_n)$, any occurrence of u_h, $1 \leq h \leq m$, in the rules is replaced by i_h, any occurrence of v_k, $1 \leq k \leq n$, in the rules is replaced by o_k, and any other variable in the rules but not in the header is replaced by a fresh variable. The replacement of a procedure call by a set of rules initialised with entirely fresh variables can be regarded as a step in the computation.

The basis of the rule for procedure rewrite, which we develop in more detail later, is that given x=a and a set of rules including the rule x=a||body, where a is a constant, we rewrite the set of rules to body. Note that, unlike the pi-calculus, the assignment is not consumed once used, and the variable may never be re-used in an assignment. We can show this by the computation rule:

$$\texttt{x=a,\{…;x=a||body;…\}:…} \longrightarrow \texttt{x=a,body}$$

We allow more than one test on the lhs, so we can generalise this to:

$$x_1=a_1,\ldots,x_n=a_n,\{\ldots;x_1=a_1,\ldots,x_n=a_n||\texttt{body};\ldots\}:\ldots \longrightarrow x_1=a_1,\ldots,x_n=a_n,\texttt{body}$$

The ordering of the assignments and the ordering of the tests is irrelevant, as is the ordering of the rules in $\{\ldots;x_1=a_1,\ldots,x_n=a_n||\texttt{body};\ldots\}$. However, the rules following ":" in the rule set cannot be employed at this stage.

A rule is discarded if there is an assignment to a constant other than the one being tested for in the rule:

$$\texttt{x=a, \{…; …,x=b,…||body; …\}:…} \longrightarrow \texttt{x=a,\{…;…\}:…} \textit{ if } a \neq b$$

If all rules in the first set have been discarded, we can go on to consider the rules in the second set:

$$\{\}:\{\text{rule}_1;\ldots;\text{rule}_n\}:\ldots \longrightarrow \{\text{rule}_1;\ldots;\text{rule}_n\}:\ldots$$

We allow rules with an empty lhs which rewrite unconditionally, so:

$$\texttt{\{…; ||body; …\}:…} \longrightarrow \texttt{body}$$

Another way of thinking of this is as individual assignments picking off tests from the lhs of rules until a lhs becomes empty and the above rule applies, in which case we have:

$$\texttt{x=a,\{…; …,x=a,…||body; …\}:…} \longrightarrow \texttt{x=a,\{…;…,…||body;…\}:…}$$

Or, since ordering of rules and tests does not matter:

$$\texttt{x=a,\{x=a,T||body;R\}:S} \longrightarrow \texttt{x=a,\{T||body;R\}:S}$$

where `T` is a set of tests, `R` a set of rules, and `S` a list of sets of rules, and the existence of computation rules to reorder `T` and `R` (but not `S`) is assumed. We have also (indicating discarding a rule when one test fails, moving to a second set of rules when all rules are discarded, and using a rule when all its tests succeed):

`x=a,{x=b,T||body;R}:S` $\longrightarrow$ `x=a,{R}:S` *if* `a≠b`

`{}:S` $\longrightarrow$ `S`

`{T||body;R}:S` $\longrightarrow$ `body` *if* `T` is the empty set

Here `body` consists of further procedure calls and assignments which application of the last computation rule above causes to be added to the top-level set of procedure calls and assignments. The assignments in `body` will then cause other sets of rules to become rewriteable.

We allow ordering tests on the lhs of rules, which will fail if their arguments are orderable:

`x=a,y=b,{x>y,T||body;R}:S` $\longrightarrow$ `x=a,y=b,{T||body;R}:S` *if* `a>b`

`x=a,y=b,{x>y,T||body;R}:S` $\longrightarrow$ `x=a,y=b,{R}:S` *if* `a≤b`

`x=a,y=b,{x>y,T||body;R}:S` $\longrightarrow$ `x=a,y=b{R}:S` *if* `a` and `b` are not orderable by >

The precise definition of "orderable" (whether numerical, or applying more widely, for example, alphabetic ordering of tags) is not relevant for this paper.

Also a wait test allows suspension until a variable is bound to any value:

`x=a,{wait(x),T||body;R}:S` $\longrightarrow$ `x=a,{T||body;R}:S`

and type tests give dynamic typing:

`x=a,{integer(x),T||body;R}:S` $\longrightarrow$ `x=a,{T||body;R}:S` *if* `a` is an integer

`x=a,{integer(x),T||body;R}:S` $\longrightarrow$ `x=a,{R}:S` *if* `a` is not an integer.

Our computation rules, as given so far, have not taken account of variables being assigned or tested for tuples containing further variables. In the full rules we allow tests on the lhs of a rule of the form $\mathtt{x=s(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n)}$, where $m\geq 0$ and $n\geq 0$. The notational convenience for omitting brackets described previously may again be used. The variable names $\mathtt{i_1,\ldots,i_m}$ and $\mathtt{o_1,\ldots,o_n}$ must all be new variable names, with rule scope so they can be re-used in other rules. For the purposes of the describing the operational behaviour, all arguments to tuples in tests must be variables. However, for notational convenience we can write a input tuple argument in a test as a non-variable, and take this as being shorthand for introducing a separate variable and testing it, so `x=s(…,t,…)->(…)` is shorthand for `x=s(…,y,…)->(…),y=t` where `y` is a new variable name, and `t` a term.

Given this, the computation rule for matching an assignment against a test is:

$\mathtt{x=s(u_1,\ldots,u_m)\text{->}(v_1,\ldots,v_n),\{x=s(i_1,\ldots,i_m)\text{->}(o_1,\ldots,o_n),T||body;R\}:S} \longrightarrow$
$\mathtt{x=s(u_1,\ldots,u_m)\text{->}(v_1,\ldots,v_n),\{i_1\text{<-}u_1,\ldots,i_m\text{<-}u_m,v_1\text{<-}o_1,\ldots,v_n\text{<-}o_n,T||body;R\}:S}$

If `x` is a linear variable, we could at this point add to `body` on the rhs an indication that if the lhs becomes empty and the rule is chosen, the assignment can be removed as this is the one permitted reading of the variable.

Now we need to deal with `x<-y` occurring on the lhs of rules (which can only occur temporarily after the application of the above computation rule). If we are testing that `x` has a particular tuple value, and `x` is matched against variable `y`, then we are testing that `y` has that pattern, replacing an internal variable in the test with an external one. So:

`{x<-y,x=t,T||body;R}:S` $\longrightarrow$ `{x<-y,y=t,T||body;R}:S`

We must also take account of the other tests that may occur on the lhs, for example:

`{x<-y,wait(x),T||body;R}:S` $\longrightarrow$ `{x<-y,wait(y),T||body;R}:S`

If the lhs of a rule consists only of variable assignments, there are no further tests, so the rule can be applied but the variable assignments must be retained for use with `body`:

$\{x_1\texttt{<-}y_1,\ldots,x_n\texttt{<-}y_n||\texttt{body;R):S} \longrightarrow x_1\texttt{<-}y_1,\ldots,x_n\texttt{<-}y_n\texttt{,body}$

This ensures the internal variables of `body` are linked with external variables. Note that since a linear variable cannot be assigned to a non-linear variable, this rule is conditional on there being no $x_i\texttt{<-}y_i$ where y_i is denoted as linear but x_i is not. If there is such a match, the rule becomes inapplicable:

`{x<-y,T||body;R}:S` $\longrightarrow$ `{R}:S` *if* `y` is linear and `x` is non-linear.

A rule also becomes non-applicable if it involves matching tuples of differing arities, or tuples of differing tags:

$\texttt{x=s(}u_1,\ldots,u_m\texttt{)->(}v_1,\ldots,v_n\texttt{),\{x=s(}i_1,\ldots,i_p\texttt{)->(}o_1,\ldots,o_n\texttt{),T||body;R\}:S} \longrightarrow$

`{R}:S` *if* $p\neq m$

$\texttt{x=s(}u_1,\ldots,u_m\texttt{)->(}v_1,\ldots,v_n\texttt{),\{x=s(}i_1,\ldots,i_m\texttt{)->(}o_1,\ldots,o_p\texttt{),T||body;R\}:S} \longrightarrow$

`{R}:S` *if* $p\neq n$

$\texttt{x=}s_1\texttt{(}u_1,\ldots,u_m\texttt{)->(}v_1,\ldots,v_n\texttt{),\{x=}s_2\texttt{(}i_1,\ldots,i_m\texttt{)->(}o_1,\ldots,o_p\texttt{),T||body;R\}:S} \longrightarrow$

`{R}:S` *if* $s_1\neq s_2$

5. Procedure Structure

As already noted, a procedure consists of a header giving a name and two lists of arguments, one for input, and one for output, followed by a list of sets of rules. Each rule consists of a list of tests forming the lhs and a list of computations forming the rhs. A test on the lhs takes the form $\texttt{x=s(}u_1,\ldots,u_m\texttt{)->(}v_1,\ldots,v_n\texttt{)}$, with a small number of other tests permitted, such as the comparison tests `x>y`, the wait test `wait(x)` and dynamic type tests. The computations on the rhs consist of assignments `x<-y`, $\texttt{x=s(}u_1,\ldots,u_m\texttt{)->(}v_1,\ldots,v_n\texttt{)}$, and procedure calls $\texttt{p(}u_1,\ldots,u_m\texttt{)->(}v_1,\ldots,v_n\texttt{)}$. Here `x`, `y`, each u_i and v_j are variable names. No other syntax is required, though a certain amount of syntactic sugar may be used to make the notation more readable, such as using a term instead of a variable, so that $\texttt{p(}\ldots\texttt{,t,}\ldots\texttt{)->(}v_1,\ldots,v_n\texttt{)}$ is shorthand for $\texttt{p(}\ldots\texttt{,y,}\ldots\texttt{)->(}v_1,\ldots,v_n\texttt{),y=t}$ on either the lhs or the rhs, with `y` an otherwise unused variable.

In order to ensure correct moding, with variables having exactly one producer, and in the case of linear variables exactly one consumer, the following conditions apply in use of variables:

1) In any test $x=s(u_1,…,u_m)->(v_1,…,v_n)$ on the lhs, if n>0 or any u_i is indicated as linear, x must be linear. There will be a notational indication to show which variables are to be treated as linear.

2) In any test $x=s_1(u_1,…,u_m)->(v_1,…,v_n)$ on the lhs, x must be either an input argument to the procedure, or occur as one of the w_is in another test $y=s_2(w_1,…,w_p)->(z_1,…,z_q)$ on the same lhs.

3) No variable occurring as u_i or v_j in $x=s_1(u_1,…,u_m)->(v_1,…,v_n)$ on the lhs may occur in the procedure header or as w_h or z_k in another test $y=s_2(w_1,…,w_p)->(z_1,…,z_q)$ on the same lhs.

4) Every output variable to the procedure, and every extra output variable in a rule, that is one of the v_is in any $x=s(u_1,…,u_m)->(v_1,…,v_n)$ on the lhs, must be used in exactly one output position on the rhs. An output position is x in $x=s(u_1,…,u_m)->(v_1,…,v_n)$ or in x<-y, or any v_i in $x=s(u_1,…,u_m)->(v_1,…,v_n)$ or any v_i in $p(u_1,…,u_m)->(v_1,…,v_n)$.

5) If a linear variable occurs as x in a test $x=s(u_1,…,u_m)->(v_1,…,v_n)$ on the lhs, it must not occur at all on the rhs.

6) Any input linear variable either from the procedure heading or occurring as one of the u_is in a test $x=s_1(u_1,…,u_m)->(v_1,…,v_n)$ on the lhs which does not occur in a test as y in $y=s_2(w_1,…,w_k)->(z_1,…,z_h)$ on the lhs must be used exactly once in an input position on the rhs. An input position is y in x<-y or any u_i in $x=s(u_1,…,u_m)->(v_1,…,v_n)$ or any u_i in $p(u_1,…,u_m)->(v_1,…,v_n)$.

7) Any variable that occurs only on the rhs of a rule must occur in exactly one output position. If it is a linear variable, it must also occur in exactly one input position, otherwise it can occur in any number of input positions.

A new variable is introduced under condition 7 when a procedure call rewrites using one of its rules. We refer to this as a "local variable". If the variable is introduced with its output occurrence as one of the v_is in $p(u_1,…,u_m)->(v_1,…,v_n)$, the procedure has itself set up a computation to give the variable a value. If, however, it is introduced as one of the v_is in $x=s(u_1,…,u_m)->(v_1,…,v_n)$ in the rhs where x is not itself a local variable, the variable will be given its value by the procedure which has x as in input. This is a form of what is called "scope extrusion" in pi-calculus. Scope extrusion of read access to a variable is given when it is used as one of the u_is in $x=s(u_1,…,u_m)->(v_1,…,v_n)$. If the procedure heading is $p(i_1,…,i_m)->(o_1,…,o_n)$, write access to a variable x can also be passed out of the procedure by $x<-i_k$ and read access passed out by $o_h<-x$. Also if we have $y=s(u_1,…,u_m)->(v_1,…,v_n)$ as a test on the lhs, write access to a variable x can also be passed out of the procedure by $x<-u_i$ on the rhs and read access passed out by $v_j<-x$. Otherwise, access to a variable remains private within the procedure where it was created and it cannot be interfered with by another procedure.

Although values given to variables are not rescinded, condition 5 can be seen as dictating consumption of a value sent on a linear variable considering it as a channel. If a rule with a linear variable test is used to progress computation, that linear variable cannot be used again, so in practice the assignment to it could be deleted. If a reference count is kept to record the number of readers of a non-linear variable, the assignment to the non-linear variable could in practice be deleted if that reference count drops to zero.

6. Dynamic Communication Topology

If a procedure call has output access to two variables `X` and `Y` (from here we will adopt the convention that linear variables are indicated by an initial capital letter), with input access to `X` and `Y` being two separate procedures, a direct communication channel can be made between those two procedures. `X=t1->c,Y=t2(c)` will establish a one-way communication channel from the call which inputs `X` to the call which inputs `Y`. If this linking variable is itself linear, as in `X=t1->C,Y=t2(C)`, a channel which may be reversed in polarity is established.

Let us consider an extended example. We have a dating agency which for simplicity has access to just one girl client and one boy client, shown by computation:

```
agency(Girl,Boy), girl()->Girl, boy()->Boy
```

Here an `agency` call has two input linear variables, and a `girl` and `boy` call each produce one linear output variable. The agency call must wait until both the girl and the boy request an introduction. The boy's request contains a channel on which he can send his first message to the girl he is put in contact with, while the girl will send a request which sends back a channel on which a message from a boy will be received. This is programmed by:

```
#agency(Boy,Girl)
{
 Boy=ask(Channel1), Girl=ask->Channel2 || Channel2<-Channel1
}
```

The output linear variable of the `girl` call is set to the input linear variable of the `boy` call. Now we can set up code to let them communicate:

```
#girl()->Dating
{
  || Dating=ask->Channel, goodgirl(Channel)
}

#boy()->Dating
{
  || Dating=ask(Channel), Channel=hello->Reply, goodboy(Reply);
  || Dating=ask(Channel), Channel=hello->Reply, badboy(Reply)
}
```

Sending a message on a channel and waiting for a reply is implemented by binding the channel variable to a tuple containing just one variable of output mode, and then making a call with that variable as input which suspends until the variable is bound. It can be seen that the message a `girl` call sends on the `Dating` channel reverses the polarity of that channel with the reversed channel renamed `Channel`, while the message a `boy` call sends on `Dating` keeps the polarity with `Channel` being a continuation of the same channel in the same direction.

For the sake of interest, we will let the `boy` call become non-deterministically either a `goodboy` call or a `badboy` call. A `goodboy` call sends the message `hello`, waits for the reply `hi` back, then sends a `kiss` message and waits for a `kiss` message back. When that happens it sends another `kiss` message in reply and so long as a `kiss` message is replied with a `kiss` message this continues forever. A `badboy` call sends a `bed` message when it receives a `kiss` message. We show here a `girl` call which can only become a `goodgirl` call, where a `kiss` message is replied with a `kiss` message, but a `bed` message is replied with a `no` message that has no reply variable, thus ending communication. Either type of `boy` call, on receiving a `no` message can do no more, the call is terminated. Otherwise, the recursive calls represent a continuation of the call. Here is how this is all programmed:

```
#goodboy(Channel)
{
 Channel=hi->Me || Me=kiss->Her, goodboy(Her);
 Channel=kiss->Me || Me=kiss->Her, goodboy(Her);
 Channel=no ||
}

#badboy(Channel)
{
 Channel=hi->Me || Me=kiss->Her, badboy(Her);
 Channel=kiss->Me || Me=bed->Her, badboy(Her);
 Channel=no ||
}

#goodgirl(Channel)
{
 Channel=hello->Me || Me=hi->Him, goodgirl(Him);
 Channel=kiss->Me || Me=kiss->Him, goodgirl(Him);
 Channel=bed->Me || Me=no
}
```

In the first two rules of each procedure here, `Channel` is an input channel on which is received a message which causes a reversal of polarity, so a message can be sent out on it which again reverses its polarity to receive a further message in reply. Effective two-way communication is established. A recursive call turns a transient computation into a long-lived process, the technique introduced by Shapiro and Takeuchi [25] to provide object-based programming in a concurrent logic language.

An alternative way of setting up this scenario would be for the agency call to take the initial initiative and send the `boy` and `girl` call a channel on which they communicate rather then them having to request it. In this case, the `agency`, `boy` and `girl` procedures will be different although the `goodboy`, `badboy` and `goodgirl` procedures will remain the same. The initial set-up is:

```
agency->(Girl,Boy), girl(Girl), boy(Boy)
```

with procedures:

```
#agency->(Girl,Boy)
{
  || Girl=tell(Channel),Boy=tell->Channel
}

#girl(Dating)
{
 Dating=tell(Boy) || goodgirl(Boy)
}
```

```
#boy(Dating)
{
 Dating=tell->Girl || Girl=hello->Her,goodboy(Her);
 Dating=tell->Girl || Girl=hello->Her,badboy(Her)
}
```

A third way of setting it up would be for the `boy` call to take the initiative while the `girl` call waits for the agency to communicate:

```
agency(Boy)->Girl ,boy()->Boy, girl(Girl)
```

with the code for the `agency` procedure:

```
#agency(Boy)->Girl
{
 Boy=ask(Channel) || Girl=tell(Channel)
}
```

Here the `boy` procedure used will be the same as the first version given above, and the `girl` procedure the same as the second.

These examples show how the communication topology can be dynamic. We initially have a `boy` and `girl` call which both have a communication link with an `agency` call, but have no direct communication with each other. We show three different ways in which a direct communication link can be obtained, one in which the `boy` and `girl` call take the initiative jointly, another in which the `agency` call takes the initiative, and the third in which only the `boy` call takes the initiative.

Note that the examples shown here have no final default rule, thus it could be argued the whole program could fail if a call bound a variable to a value which its reader had no rule to handle. However, moding means we can always add an implicit default rule to prevent failure. In this rule, all output variables of the procedure are set to a special value indicating an exception. All input linear variables become the input variable to a special exception-handling procedure, which for any tuple the variable becomes bound to sets all output variables of the tuple to the special value indicating exception and makes all input linear variables the argument to another call to this procedure.

7. Conclusions and Related Work

The work described here can be considered a presentation of the work done by Reddy [22] oriented towards a language that can be used for practical programming. Reddy's work is inspired by Abramsky's computational interpretation [1] of linear logic [6]. We extend Reddy's typed foundation by allowing non-linear as well as linear variables, but our typing extends only as far as is necessary for modes to establish the single-writer multiple-reader property. Other attempts to build practical programming languages which add linearity to concurrent logic programming, such as Janus [26], have insisted that all variables be linear.

Our language could also be considered as a re-presentation of a committed choice logic language [24] which avoids logic programming terminology or the attempt to maintain some backward compatibility with Prolog that we argue elsewhere [14] was a contributing factor to these languages gaining little acceptance. Our strong moding expressed in the syntax of the language makes programs much easier to understand since it is always clear from where a variable receives its binding. It also means that the problem of dealing with the rare possibility of more than one computation wishing to bind a variable, which led to many of the variations discussed in [24], does not occur.

Another computation model related to ours is Niehren's delta-calculus [20]. Like our notation, the delta-calculus represents functions as relations with an explicit output variable and an assignment operator. The delta-calculus also uses linear types to enforce single assignment to variables. Unlike our language, the delta-calculus is higher order, that is variables may be assigned procedure values and used as operands. Although our language is first-order, we have shown elsewhere [15] how the effect of higher order functions can be obtained using the standard techniques for representing objects in committed choice logic languages [25], a function can be considered as just an immutable object which has only one method (application).

Our work originates from attempts to build an object-oriented language on top of concurrent logic programming under the name "Aldwych" [13]. Previous attempts to do so [3] had been criticised for losing some of the flexibility of concurrent logic programming [16]. However these languages did have the benefit of being much less verbose than the equivalent code expressed directly as concurrent logic programming. Our intention was to have a syntax in which common patterns of using the underlying concurrent logic language were captured, as little as possible of the operational capability was lost, and the direct translation into concurrent logic programming kept in order to maintain a clear operational semantics. Aldwych enables procedures to be written and thought about in a style that resembles object-oriented programming, and also in a style that resembles functional programming, with functions that can be curried.

Our use of linear variables arose from practical necessity, but its closeness to Reddy's work establishes a stronger theoretical justification for it. This paper originates from the necessity to provide a clear description of the language underlying Aldwych. In previous papers introducing Aldwych we have described it in terms of translation to a concurrent logic language, but due to the plethora of such languages, and the mistaken belief that they are "parallel Prologs", our intention has not always been clear. In this paper we define an operational semantics for the underlying language in terms of reduction rules and under the assumption, which will be true in any translation from Aldwych, that all variables are moded such that they will have a single writer which can be found using the syntax of the language.

References

[1] S.Abramsky. A computational interpretation of linear logic. *Theoretical Computer Science* 111:3-57 (1993).

[2] K.L.Clark and S.Gregory. A relational language for parallel programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*. 171-178. (1981).

[3] A.Davison. A survey of logic programming based object oriented languages. In *Research Directions in Concurrent Object Oriented Programming*. G.Agha, P.Wegner, A.Yonezawa (eds) MIT Press (1993).

[4] D.Q.M.Fay Experiences using Inmos proto-OCCAM™ *SIGPLAN Notices* 19 (9) (1984).

[5] I.Foster and S.Taylor *Strand: New Concepts in Parallel Programming*, Prentice-Hall (1989).

[6] J.-Y.Girard. Linear logic. *Theoretical Computer Science* 50:1-102 (1987).

[7] S.Gregory. *Parallel Logic Programming in PARLOG.* Addison-Wesley. (1987).

[8] D.H.Grit and R.L.Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Trans. Prog. Lang and Sys*, 3(1):49-59. (1981).

[9] R.H.Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang and Sys*, 7(4):501-538. (1985).

[10] S.Haridi, P Van Roy, P.Brand, M.Mehl, R.Scheidhauser and G.Smolka. Efficient logic variables for distributed computing. *ACM Trans. Prog. Lang and Sys*, 21(3):569-626. (1999).

[11] J.Hughes. Why functional programming matters. *Computer Journal*, 32(2):98-107. (1989).

[12] M.M.Huntbach and G.A.Ringwood. *Agent-Oriented Programming*. Springer LNCS 1630 (1999).

[13] M.Huntbach. The concurrent language Aldwych. *Proc. 1st Int. Workshop on Rule-Based Programming (RULE 2000)* (2000).

[14] M.Huntbach, The concurrent language Aldwych. *World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001)* XIV:319-325.

[15] M.Huntbach. Features of the concurrent language Aldwych. *ACM Symp. on Applied Computing (SAC'03)* 1048-1054 (2003).

[16] K.M.Kahn. Objects – a fresh look. *Proc. 3rd European Conf. on Object-Oriented Programming (ECOOP 89).* S.Cook (ed), Cambridge University Press.

[17] R.A.Kowalski. *Logic for Problem Solving*. Elsevier/North Holland (1979).

[18] P.J.Landin The mechanical evaluation of expressions. *Computer Journal* 6 (4):308-320 (1964).

[19] R.Milner, J.Parrow and D.Walker. A calculus of mobile processes. *J of Information and Computation*, 100:1-77 (1992).

[20] J.Niehren. Functional computation as concurrent computation. *Proc. 23rd Symp. on Principles of Programming Languages (PoPL'96)* 333-343 (1996).

[21] B.C.Pierce and D.N.Turner. Pict: a programming language based on the pi-calculus. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press (2000).

[22] U.S.Reddy. A typed foundation for directional logic programming. *Proc. 3rd Int. Workshop on Extensions of Logic Programming*. Springer LNCS 660:282-318 (1993).

[23] V.A.Saraswat, M.Rinard and P.Panangaden. Semantic foundations of concurrent constraint programming. *Principles of Prog. Lang. Conf. (POPL'91)*, 333-352 (1991).

[24] E.Y.Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys* 21(3):413-510 (1989).

[25] E.Shapiro and A.Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing* 1:25-48 (1983).

[26] V.A.Saraswat, K.Kahn and J.Levy. Janus: a step towards distributed constraint programming. *Proc. 1990 North American Conf. on Logic Programming*. MIT Press. 431-446 (1990).

[27] G.J.Sussman and D.V.McDermott. From Planner to Conniver – a genetic approach. *Proc AFIPS Fall Conference* 1171-79 (1972).

[28] E.Tick. The de-evolution of concurrent logic programming languages. *J. Logic Programming* 23(2): 89-123 (1995).

[29] K.Ueda. Experiences with strong moding in concurrent logic/constraint programming. *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, Springer LNCS 1068:134-153. (1996).

[30] K.Ueda. Linearity analysis of concurrent logic programs. *Proc. Int Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications.* T.Ito and T.Yuasa (eds) World Scientific Press (2000).