A Model for Concurrency Using Single-Writer Single-Assignment Variables

Matthew HUNTBACH
School of Electronic Engineering and Computer Science
Queen Mary, University of London
mmh@eecs.qmul.ac.uk

Abstract. This paper describes a model for concurrent computation based on single-writer single-assignment variables. The description is primarily graphical, resembling the interaction nets formalism. The model embodies rules in a process which may require two or more communications from other processes to respond. However, these are managed by a partial evaluation response on receiving a single communication.

Keywords. Concurrency, computational model, single-assignment, linear variables.

Introduction

This work originates from an attempt to redescribe the committed choice concurrent logic programming model in terms of the dynamics of handling its variables rather than in terms of logical deduction. The key to the success of this model in terms of its ability to be mapped with ease onto multiprocessor architecture was the concept of the "logic variable" combined with a strict moding on the direction of information flow, but with the property of "back communication". This means every variable has exactly one writer, and can only be assigned once, but an assignment is a structure containing further variables some of which have the reverse mode, so the reader of the structure becomes their writer.

We demonstrate the model using a graphical notation. The description in this paper is informal, but it is hoped the intuitive nature of the notation will make it more clear how this model works operationally than earlier, more formal, textual descriptions.

1. The Logic Programming Background

In a previous paper [1] we introduced what was described as "The Core Language of Aldwych". Aldwych was an attempt to build on earlier work to provide object-oriented programming structures in a concurrent logic programming framework [2]. The concurrent logic programming languages had a brief period of success in the 1980s, being seen then as the first attempt to build practical programming languages which were inherently concurrent. Although they gained little practical use, they were influential in the development of Erlang [3], a language which obtained widespread use for concurrent programming [4], and some years later was relaunched with an emphasis on it being a functional language [5]. Aldwych had a similar aim, the idea was that although there was much of value in the concurrent logic programming languages in terms of their ease for handling concurrency, they had failed to gain much practical use because their simple syntactic structure did not scale up well to large sized programs.

Observation of attempts to use the concurrent logic languages showed there were common patterns of code usage, which we might now recognise as “design patterns” [6]. The most widely used was to give the effect of an object with a mutable state through a logic programming predicate in which the state was an argument, so the changed state was an argument to a recursive call. Access to an object was managed by “streams” of “calls”, where a stream was a list whose head was the first call to the object, and whose tail was a list of further calls. Concurrent access to objects was managed by non-deterministic merging of several streams into one. A “call” was a tuple, which could give the effect of a method call giving a return value. This worked because concurrent logic programming has the idea of “back communication” [7]. Back communication comes when a computation may bind a variable to a tuple containing further variables. Some of those further variables may also be bound by the computation, or it sets up further computations to bind them, but back communication is when the reader of the initial variable which was bound to the tuple is expected to become the writer of one or more variables in the tuple.

There were then four things which lay behind the power of the concurrent logic model for concurrent programming:

1. Non-determinacy was inherent in the model, enabling a concurrent computation to behave in different ways for the same input. A computation which is the reader of two variables whose values are being computed concurrently may react as soon as one of them receives a value without having to wait for the other. What it does depends on which gets a value first. Non-deterministic stream merger is a good example, the head value of whichever stream gets a head value first is passed on as the head value of the merged stream.
2. Back communication gives a two-way interaction between computations, managed very simply as variable binding. A computation sends a tuple containing an unbound variable to another computation just by binding a shared variable to which it has read access and the other has write access. It receives its reply as the value of that unbound variable to which it has read access and the other computation was given write access.
3. Variables may not be re-assigned. Once a variable has been given a value, that value can never be over-written. This is the standard “logic variable” property [8]. It means the complexity of managing shared access to mutable variables that dominates concurrent programming with threads in standard programming languages is avoided.
4. Variables serve as “futures” [9] that is, placeholders for values being computed concurrently. When a computation “sends” a tuple as described above, as well as unbound variables intended for back communication, it may include in the tuple variables which are unbound because another computation has the task of writing to them. The receiver of the tuple may send them on further in other tuples without waiting for them to acquire a value.

Each of these is handled naturally in the concurrent logic programming model, whereas other computational models may require special syntax and operators to provide them. In some ways it was because these aspects were handled so simply in concurrent logic programming that their power was not recognised. However, it was also hidden because the logic programming philosophy was to see code in the language as static statements of facts, and thus to hide the dynamics of how variables became bound to values that were in accordance with those facts. So it was logic programming philosophy not to indicate a direction of flow through variables, or to see computations as having specifically read or write access to variables.

Concurrent logic programming had an unfortunate start because its proponents were apologetic about it, tending to emphasise its restrictions and to see it as an intermediate stage which in time would be developed to full "parallel Prolog", spending much time trying to develop more elaborate models which were closer to that aim. The concurrent programming language Strand [10] (winner of the 1989 British Computer Society award) started the process of looking positively at the clean concurrent mechanisms of this model instead of apologetically at it as over-restrained logic programming, but it kept the logic programming syntax with its lack of syntactically clear modes on variable occurrences.

Our work derives from wishing to rescue the simple underlying model from its logic programming "baggage", and to use it as a foundational calculus for programming languages which are naturally concurrent.

2. Aldwych-Core

Aldwych used the Strand model of concurrent logic programming in its most pared down "flat" committed choice form. The semantics of the various Aldwych constructs were originally described in terms of a translation to Strand or a similar language (hence the name as "Aldwych turns into Strand" as both programming languages and London street names). However, it became clear that this was unsatisfactory, as the operational model of Strand and similar languages was still unnecessarily complicated by their logic programming inheritance. In particular for most purposes they required variables to have a single writer, but this was not enforced, and there were rare cases where variables had multiple writers, which then required mechanisms to deal with more than one writer trying to bind a single variable. There was also a practical need to break Aldwych's reliance on programming language implementations which were no longer being supported. So, a new description of the underlying model was developed. This work led to the discovery that underneath was a model comparable in its simplicity to other fundamental models of concurrency, such as CSP [11] and the π–calculus [12]. In our previous paper [1] we called this "The Core Language of Aldwych", in this paper we shorten that to "Aldwych-Core".

Although the development of a practical naturally concurrent general purpose programming language remains part of our aims, we have found that Aldwych-Core works well in providing an operational semantics for other programming languages which can then be used to explore how concurrency impacts on their key features. An informal sketch of the use of Aldwych for modelling mutable variables, object-oriented programming, and higher order functions can be found in another of our previous papers [13]. The refined and simplified model presented in this paper opens the way for use of Aldwych-Core in work oriented around semantics of programming language constructs.

Aldwych-Core strictly enforces a mode on all variables, ensuring that each unbound variable always has exactly one computation that has write access to it. Once a variable is bound, no computation can have write access to it, giving the single-assignment property. The combination of strict enforcement of modes and back communication resulted in identifying the need for enforced linearity [14] in any variable which could be used for back communication. In addition to the single-writer property, variables classified as linear must also have exactly one computation with read access (possibly indirectly through being in a tuple which is assigned to a variable with a single reader). The distinction between linear and non-linear variables is enforced syntactically. Only a linear variable may be bound to a tuple which contains back communication. Non-linear variables may only be bound to tuples which contain only non-linear variables. The reason for this is that a non-linear variable can have multiple readers or no readers: so if it were possible to bind it to a tuple with back communication, the back communication variables could have zero or multiple

writers, so breaking the requirement that every variable has exactly one writer. This applies indirectly, so if a non-linear variable could be bound to a tuple containing no back communication but containing linear variables, the duplication or deletion of readers of that variable would duplicate or delete readers of the linear variables in the tuple, which would duplicate or delete writers for any back communication in tuples to which they are bound.

Our earlier paper describes a syntax for Aldwych-Core which is based on enforcing the single-writer property rather than on writing "clauses" which attempt to have some resemblance to predicate logic notation. It describes an operational model which is broken down into smaller steps than those conventionally associated with logic programming. Rather than the logic process of unification of call with pattern in clause head, each individual variable match and assignment was described as a separate stage.

3. A Diagrammatic Representation of Aldwych-Core Agent Networks

The Aldwych-Core universe of computation can be described as follows. We have a collection of computational agents and a collection of variables. Each computational agent has read access to zero or more variables and write access to zero or more variables. We have a collection of assignments, each of which has read access to zero or more variables and write access to exactly one variable. Every variable must occur in exactly one write position, that is there must be either exactly one agent which has write access to it or exactly one assignment which has write access to it. If a variable is designated as "linear", it must occur in exactly one read position, that is there must be either exactly one agent which has read access to or exactly one assignment which has read access to it, and no other agent or assignment has read access to it. A variable designated as "non-linear" may occur in any number of read positions, so it may have zero readers, one reader, or more than one reader. An assignment can be considered the simplest form of agent, one which declares a static relationship between input and output, and Aldwych computation consists of agents reducing until they become just a network of assignments.

This describes the universe of computations because an agent may have the same internal structure. What is a read position internally is a write position externally. So for any computation, the world may be considered just another agent with which it interacts through variables it shares with the world.

In our previous paper, we showed this textually, but another way of representing Aldwych-Core computations is diagrammatically. If we represent an agent by a circle, variables it has write access to are represented by arrows leading from it, variables it has read access to are represented by variables leading to it. So what is textually written as `g(x)->z, f(z,y)->v` can be shown diagrammatically as either of figures 1 and 2.

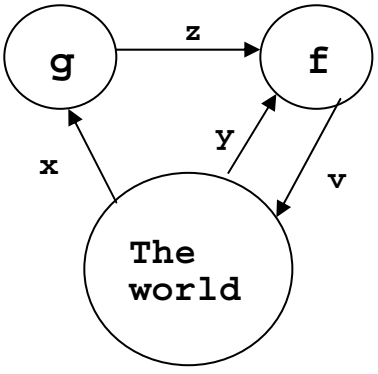

Figure 1. The world as an agent.

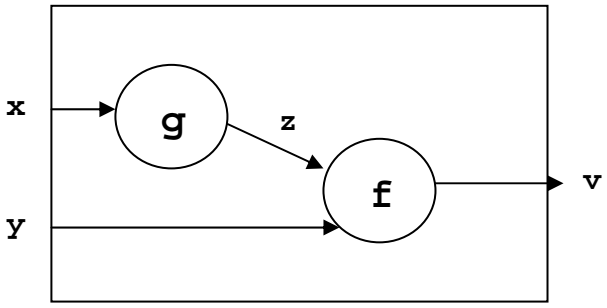

Figure 2. Computational agent with internal agents.

In figure 1, "the world" is shown as everything except the agents in question. We could consider them as a "not the world" agent that interacts with "the world" (figure 3).

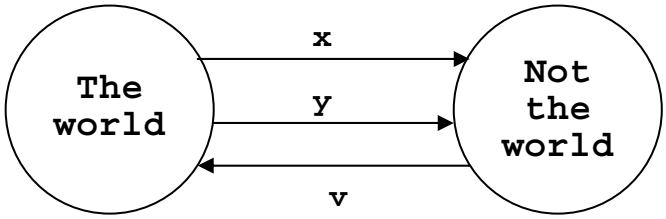

Figure 3. Computation as world versus not-the-world.

This is the insight of game semantics [15] which describes computation in terms of "player" ("not-the-world") versus "opponent" ("the world"). Any agent may have an internal state which consists of a network of agents linked by variables with the variables to which it has read access being variables to which an agent representing "everything else" has write access, and vice versa. Given a network of agents, we can arbitrarily group together *any* of the agents and regard the result as a single agent. Arcs between agents included in the group become considered as internal variables, but there is no requirement for the resulting graph structure not to be disjoint. For example, the agent with the diagram in figure 4 appears to the world outside as a single agent which takes three inputs and gives two outputs. The clean nature of Aldwych-Core is shown by the way we can arbitrarily take any processes from the universe and give the result an operational meaning which has no dependency on anything else. The agent shown in figure 4 can be written textually as `g(x)->u, f(y,z)->v` but the variable names in the diagram are just to give a link to the textual representation. There is no significance in the particular name used, and the diagrammatic representation does not require arcs to be labelled with variable names.

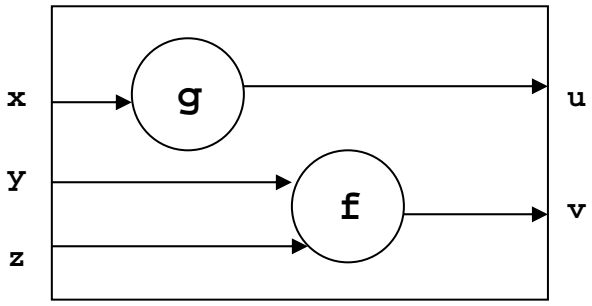

Figure 4. An agent with disjoint components.

We need to distinguish between linear and non-linear variables and to allow multiple readers of non-linear variables. In the diagrammatic notation, linear variables are shown by double-lined arrows. Multiple readers are shown by the use of a "duplicator" (indicated by a small square); we are influenced in this by a similar concept in Lafont's Interaction Nets [16], another diagrammatic model of computation. A duplicator and linear variables are shown in figure 5.

The diagram in figure 5 represents what is written textually as:

```
g(x)->z, f(y,z)->v, h(z,P)->Q
```

using the convention that in the textual representation linear variables are indicated by initial upper-case letters. The order in which the component agent representations are given in the textual representation has no significance. The order in which the variables are given

for individual agents is significant, in the diagrammatic notation anti-clockwise from 12 o'clock for read access and clockwise from 12 o'clock for write access corresponds to left-to-right textually. The notation `u<-v` indicates a variable-to-variable assignment, shown diagrammatically by a variable name on the arc leading from a duplicator. If the arc from the duplicator to the agent labelled `f` in the diagram were separately labelled `w`, that would indicate what is written textually as:

```
g(x)->z, w<-z, f(y,w)->v, h(z,P)->Q.
```

Variable-to-variable assignments are not assignments in Aldwych-Core terms, they just transfer the actual assignment of one variable to another, and can be executed before the actual assignment is available. Section 6 shows reduction rules to cover this.

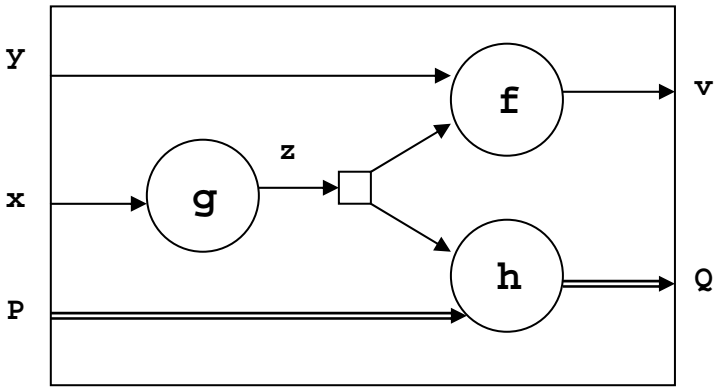

Figure 5. An example with a duplicator and linear.

Aldwych-Core assignments are assignments of tuples to variables, written textually as `v=t` where `t` must be a tuple, not a variable. Diagrammatically they are indicated by triangles, with the tuple tag inside the triangle. So the diagram of figure 6:

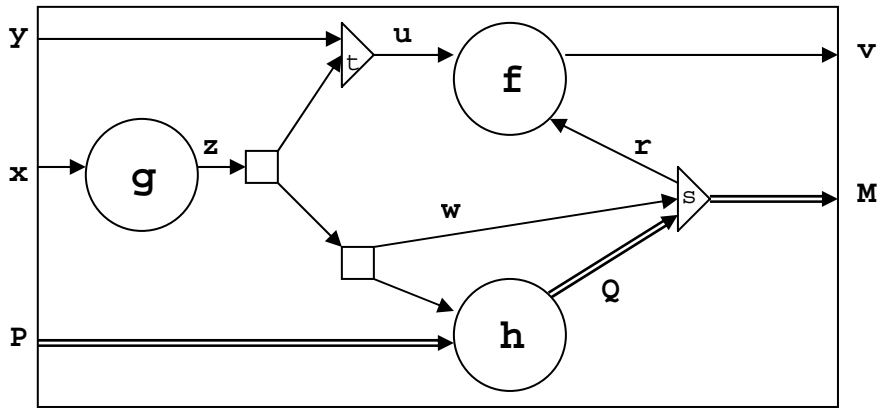

Figure 6. An example with assignments and back communication.

represents what is written textually as:

```
g(x)->z, w<-z, f(u,r)->v, h(z,P)->Q, u=t(y,z), M=s(w,Q)->r.
```

This indicates a case of back communication, in the assignment `M=s(w,Q)->r`. The whole may be seen as an agent which the world sees as `Agent(y,x,P)->(v,M)` where the back communication will result in the reader of `M` becoming the writer of `r`. We could show this as in figure 7:

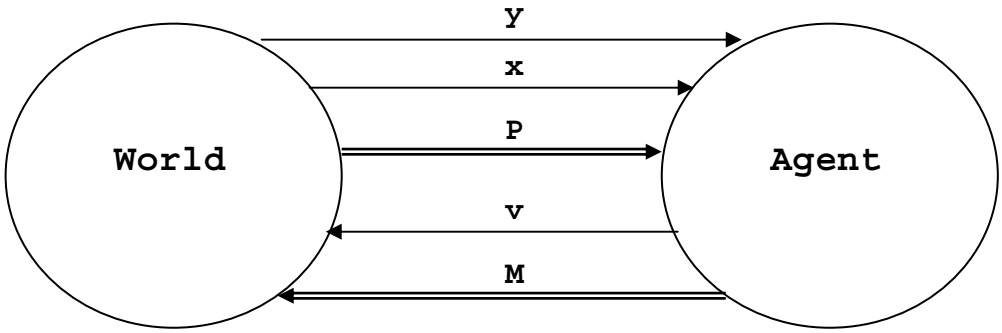

Figure 7. The agent of figure 6 communicating with the world.

Now suppose we were to redraw the boundaries of `Agent`, so that the assignment **`M=s(w,Q)->r`** is no longer part of it. This would give the situation in figure 8:

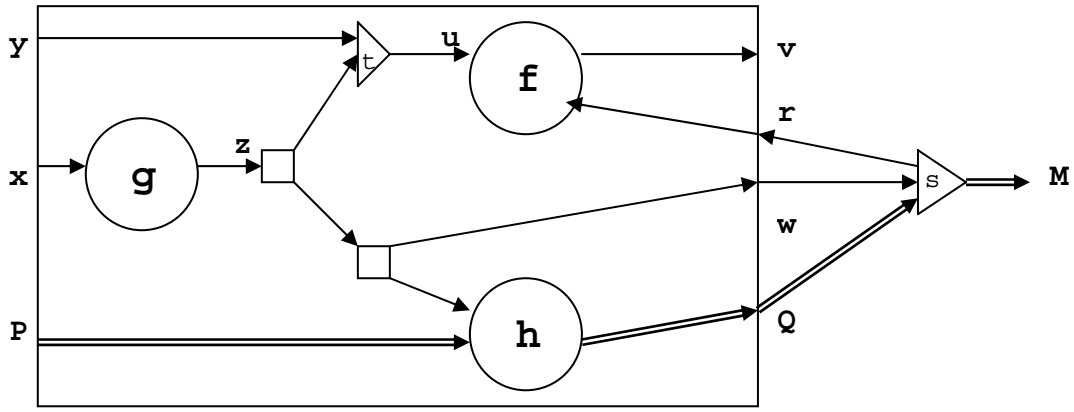

Figure 8. Redrawing the boundaries of figure 6.

which could be shown as in figure 9:

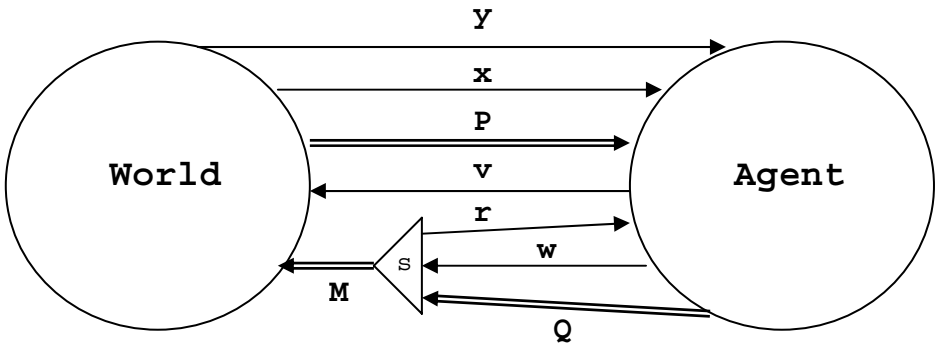

Figure 9. The agent of figure 8 communicating with the world and an assignment.

We could then redraw the boundaries of the world to obtain what is shown in figure 10:

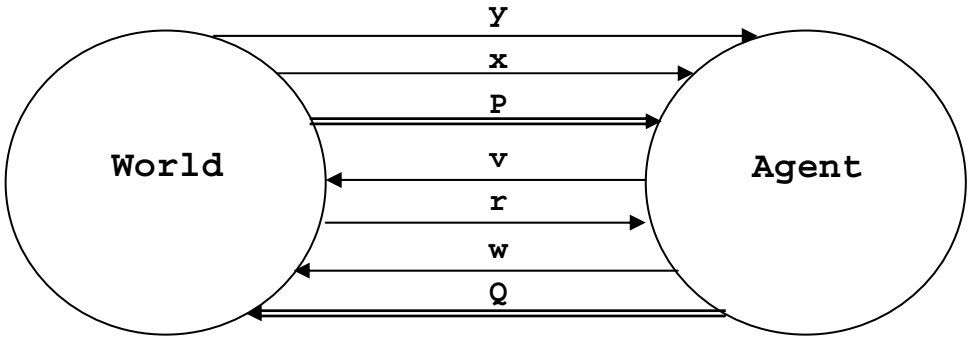

Figure 10. Redrawing the boundaries of figure 9.

The result here is that the communication from **Agent** to **World** through **M** has been replaced by communication from **Agent** to **World** through **w** and **Q** and from **World** to **Agent** through **r**. Our model assumes there is unbounded nondeterminism in this communication, that is the rearrangement of agent boundaries so that an assignment is extruded by one agent and absorbed by another is done at whatever pace the agents decide on; there is no central scheduler which orders it. We can only say that an assignment is absorbed at some time arbitrarily later than the time it is extruded.

The diagrammatic notation gives a more intuitive feel than the textual rules of our earlier paper for how the single-writer single-assignment model works to express concurrency. Assigning a tuple to a variable can be seen as communication across a channel, with one variable in the tuple designated as the continuation of the channel for further communications. The duplicator mechanism extends this to the multiple reader of non-linear variables. In this case we want an agent to be able both to extrude an assignment and keep it to use internally. Diagrammatically this is shown by a rearrangement in which the duplicator duplicates the tag of a tuple and then passes the duplication up to each of the variables in the tuple. As an example, from figure 11:

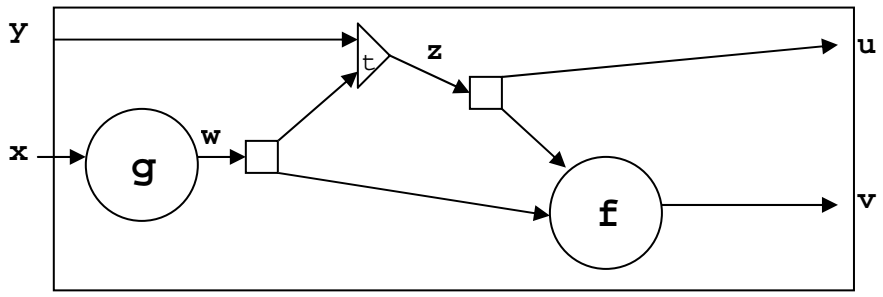

Figure 11. An assignment going in to a duplicator.

we obtain figure 12:

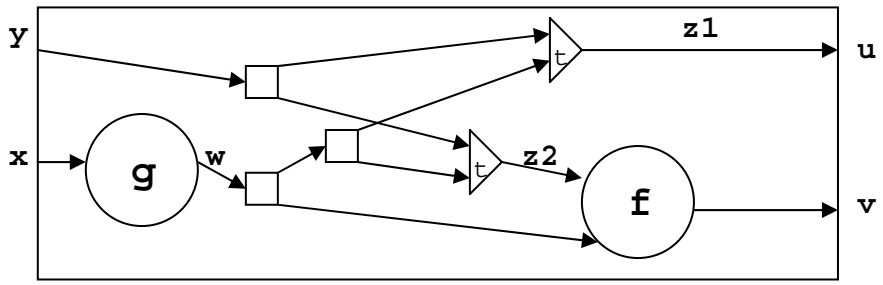

Figure 12. The assignment of figure 11 duplicated by the duplicator.

Textually, this is equivalent to the expression:

```
g(x)->w, z=t(y,w), f(z,w)->v, u<-z
```

transforming to:

```
g(x)->w, z1=t(y,w), z2=t(y,w), f(z2,w)->v, u<-z1
```

which is then followed up by the extrusion of the assignment to `u` via `z1`. This is written as `u=t(y,x)` in the world where `u` is read, while the agent becomes:

```
g(x)->w, z2=t(y,x), f(z2,w)->v, y1<-y, w1<-w
```

and shown diagrammatically in figure 13:

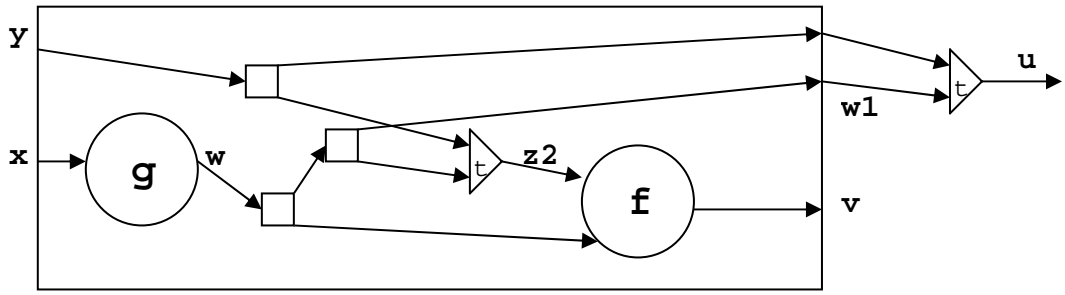

Figure 13. Extrusion of the duplicated assignment.

4. A Diagrammatic Representation of Aldwych-Core Rules

An agent could be a simple assignment, or a network of assignments as in figure 14, which represents `agent(y,x)->(u,v,w)`, which internally is `u=t(x,y), v=t(y,x), w<-x`. It could be an agent in the real world set up with an Aldwych-Core interface through the variables it has read and write access to, in which case we can say no more about it.

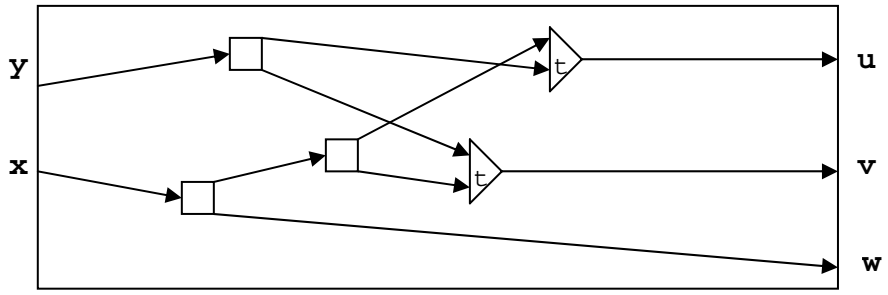

Figure 14. An agent consisting only of assignments.

An agent representing an Aldwych-Core process however has a structure consisting of a collection of rules. Rules have a left-hand side which consists of matches and a right-hand side which is an Aldwych-Core agent. A match is the reverse of an assignment, so is shown diagrammatically as such, a triangle in the opposite direction. It has an arrow leading into the point representing a variable being matched, arrows leading out on the other side are standard variables in the tuple, arrows leading in are variables for back communication. Each match in a rule must be either to one of the variables to which the process has read access, or to one of the variables introduced but not for back communication in another

match in the same rule. The right hand side of the rule is an agent which has read access to all variables which are input to the process or introduced in a match in the rule except linear variables which are matched. The match of a linear variable represents the consumption of that variable. Duplicators are used to show non-linear variables both matched and passed to the right hand side agent, but non-linear variables may not be matched twice. The right-hand side agent has write access to all variables to which the process has write access and also all variables introduced for back communication in matches. An example is given in figure 15:

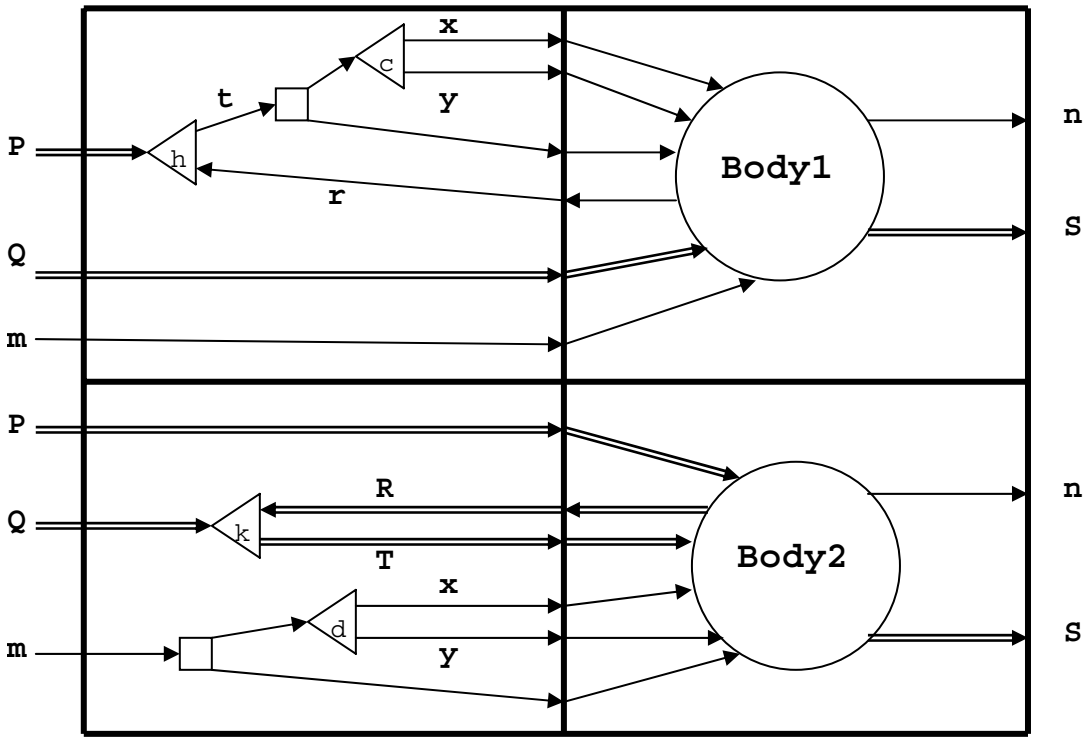

Figure 15. A process with two rules.

This represents a process which has read access to `P`, `Q` and `m` and write access to `n` and `S`, so it could be written as `Process(P,Q,m)->(n,S)` if `Process` were a template for these rules in the form suggested by our previous paper [1]. The process has two rules, the first matches `P` with `h(t)->r` and then matches the `t` which came from that tuple with `c(x,y)`. The second matches `Q` with `k(T)->R` and also matches `m` with `d(x,y)`. Textually, this would be written as:

```
(P,Q,m)->(n,S)
{
 P=h(t)->r, t=c(x,y)  ||  Body1(x,y,t,Q,m)->(n,S,r);
 Q=k(T)->R, m=d(x,y)  ||  Body2(P,T,x,y,m)->(n,S,R)
}
```

The variables internal to a rule are local, so the `x` and `y` in the first rule have no connection with the `x` and `y` in the second rule. In fact the names are added to the arrows just to link the diagram with the textual representation, the purely diagrammatic representation has no need for them. In the first rule there is a match for the linear variable `P` but no match for the linear variable `Q`, so `Q` is passed to the agent that forms the body of the rule but `P` is not. In the second rule, it is the other way round. The variable `m` is not linear, so even though it is matched in the second rule it is still passed to the agent that

forms the body. The first rule shows an example of a variable, `t`, which is introduced in a match and then matched itself in a further match. The body of each rule has to write to the external variables of the process `n` and `S` and also to any back communication variables introduced in the match, so `r` in the first rule and `R` in the second. Matches with back communication or with any linear variables whether for back communication or not (for example `T` in the second rule is linear but not for back communication) can only be with linear variables.

In this example, `Body1` and `Body2` could be any network of agents so long as it has arcs for variables leading to the required output variables, and arcs from all input linear variables leading to readers or directly to the outputs. However, a non-linear variable may be passed in to a process and then not used. This should be explicitly indicated by the use of an "eraser" (as with the duplicator, the term comes from interaction nets). Figure 16 shows a third rule to add to the two in figure 15, but in this third rule the input `m` is not used.

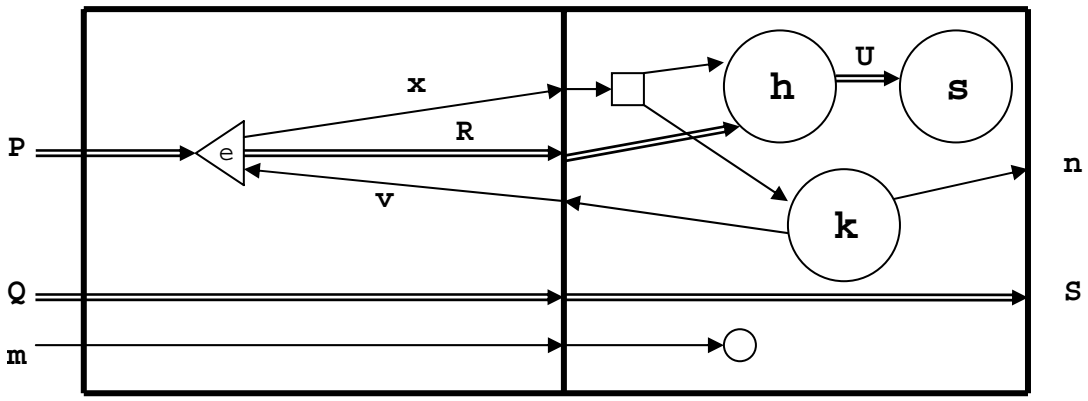

Figure 16. An extra rule for figure 15, showing content of body and an eraser.

The rule can be written textually as:

```
P=e(x,R)->v || h(x,R)->U, s(U), k(x)->(n,v), S<-Q
```

The small circle to which the arrow from `m` leads indicates the eraser. It might be though that as `s(U)` has no output variable it is equivalent to an eraser. However, because `U` is a linear variable it could be bound to a tuple with back communication going back to `h(x,R)->U`. A linear variable cannot be directed to an eraser because it must be read to provide a writer for any back communication there might be in a tuple to which it is set.

The two rules of figure 15 are enough to demonstrate the non-determinacy of Aldwych-Core. If `P` is bound to `h(u)->v`, and `u` is bound to `c(i,j)`, and `Q` is bound to `k(C)->D`, and `m` is bound to `d(a,b)` then either rule applies. Aldwych-Core does not have a mechanism for specifying which is to be taken. However, this would eliminate the rule of figure 16 as `h(u)->v` to which `P` is bound does not match with the `e(x,R)->v` on the rule's left-hand side. Unbounded non-determinism means if `P`, `Q` and `m` were bound in a particular time order we cannot assume the process will receive them in that order, so long as there is no dependency in the bindings. For example if `P`, and the variable which matches `t` within the binding for `P`, are bound before `Q` and `m` are bound, it is not necessarily the case that the first rule of figure 15 is applied rather than the second. If it happened, however, that an agent which was a reader of `n` was also the writer of `m`, we could guarantee that the second rule of figure 15 would not be applied. This is because, unlike the first rule and the rule in figure 16, the second rule requires `m` to be bound, and if that is dependent on `n` being bound it will only happen when one of the other rules has already been applied.

It is important to be clear that the rules represent alternatives. If there were a computer implementation on this notation, they should be shown stacked on top of each other using a third dimension. The limitations of two dimensions mean in the next section we see assignments to linear variables apparently duplicated as they enter the rules, but the single selection means no real duplication takes place.

5. Assignment Absorption

Aldwych-Core computation works by selecting one rule from those of a process where there is a full match with the arguments passed to process, and then reducing the process to the body on the right-hand side of the rule. The other rules are discarded, the choice of a rule is a commitment from which there is no backtracking. Assignments in the body will then be extruded, as described under section 3 above, they will then interact with the processes which have read access to the variables which are assigned. A common pattern is that the body contains a recursive copy of the original rules, so this can be seen as a continuation of the original process, communicating with the assignments as "messages" to other processes, essentially the Actor [17] model of computation. Aldwych-Core as described in our previous paper had named templates for rules, so a recursive call would be indicated by a call to the same template as the one which provided the original rules. Full Aldwych [13] provides a syntactic sugar for hiding this, enabling code to be written in a way which resembles CSP and other message-passing calculi. A variable bound to a tuple consisting of a message and a variable for future messages can be considered a channel, the syntactic sugar gives a notation which hides the shared variable implementation. We are currently working on a modified version of Aldwych-Core called "Anonymous Aldwych" which takes a little of the syntactic sugar in full Aldwych into the core computation mechanism to remove the need for named templates. This enables a process to be described completely by its rules instead of having a dependency on an external set of named templates.

In our previous paper, we explained the Aldwych-Core computation mechanism in terms of reduction steps which are smaller than the full step of selecting a rule. Each matching of an assignment against a match in a rule is one reduction step, but as rules may have more than one match, the step may not complete the action of selecting a rule for use. The step of making a single match may be considered a form of partial evaluation [18], it modifies the rules so they reflect the situation where some but not all of the inputs are provided. As explained in section 1 above, where variable values are not needed Aldwych-Core computation handles variables whose binding is being computed elsewhere as "futures". Partial evaluation is evaluation where the values of some variables are not known, so leaving "residual code" to handle them when they become available. Some of the power of Aldwych-Core comes from the way in which partial evaluation is a natural part of its normal computation mechanism.

The diagrammatic notation makes the partial evaluation nature of Aldwych-Core's computation mechanism clearer, and shows further similarity with the interaction nets formalism. As an example, let us consider the case of figure 15, in an environment where `P` is bound to `h(u)->v` and `m` is bound to `d(w,z)`. This is not enough to allow the selection of either rule, as the first requires additionally that `u` is bound in a way to match `c(x,y)` and the second requires additionally that `Q` is bound in a way to match `k(T)->R`. It is, however, enough to discard the rule in figure 15 because the assignment `P=h(u)->v` does not fit the match `P=e(x,R)->v`. The match is ruled out by both difference in tuple tag and difference in number and types of argument (where a "type" is the composition of the mode and the linearity of the variable in the tuple).

The partial evaluation of a process and an assignment converts the process to a new set of rules in which any back communication variables in the tuple of the assignment are added as process variables with write access, and the other variables are added with read access. The variable which is assigned to is removed from the external variables of the process. If a variable is matched in the rule and it is linear, the variable is consumed and removed altogether from that rule. If it is not matched in the rule, or it is non-linear, the variable is added as a local variable to the body of the rule. If the variable is matched, the local variables in the match are linked directly to the external variables of the assignment. In general, if we have a linear variable `v` to which a process has read access and it is assigned $t(i_1,\ldots,i_n)->(o_1,\ldots,o_m)$, the absorption of this variable is as follows. The variables $i_1,\ldots,i_n$ are added with read access, $o_1,\ldots,o_m$ are added with write access, `v` is removed as an external variable. Any rule which has a match for `v` which does not have tag `t` and `n` inputs and `m` back communications (with also matching linearity for each of these) is removed altogether.

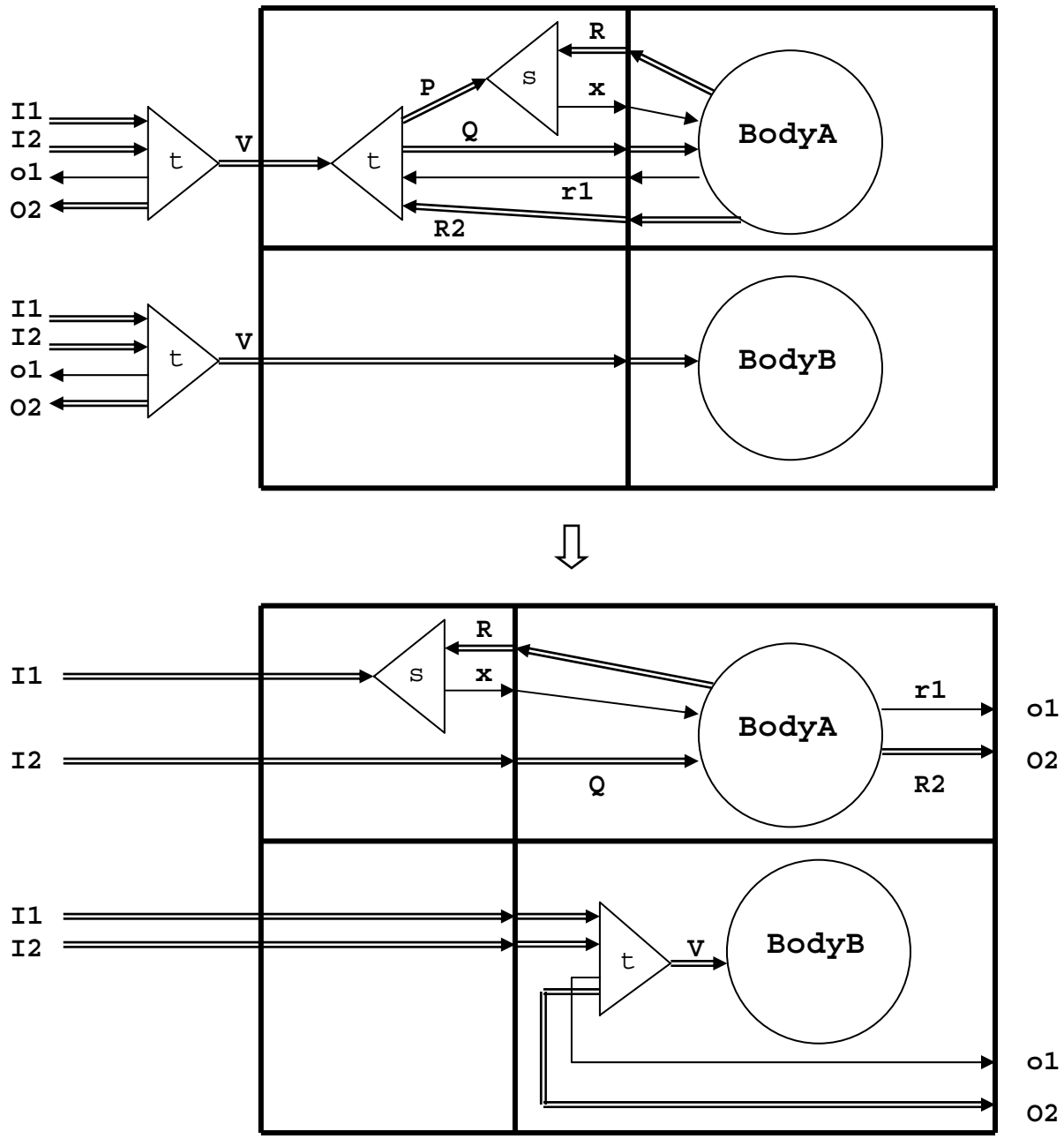

Figure 17. Absorption of an assignment.

The diagram in figure 17 shows an example (using different rules than figures 15 and 16). Textually this represents the situation where we have the assignment written `V=t(I1,I2)->(o1,O2)` and the process written:

```
(V,…)->(…)
{
 V=t(P,Q)->(r1,R2), P=s(x)->R, … || BodyA(x,Q,…)->(r1,R2,R,…);
                               … || BodyB(V,…)->(…);
 …
}
```

The ellipses here indicate the possibility of more external variables with read and write access, more matches in the rules, more internal variables from those other matches and more rules. The variable `V` is assigned a tuple `t(I1,I2)->(o1,O2)`. There are two rules left after the exclusion of those which have an incompatible match for `V`, the first has a compatible match, the second has no match for `V`. Other input and output variables for the process are not shown as their links remain unchanged by the absorption. The transformation absorbs the assignment, resulting in:

```
(I1,I2,…)->(o1,O2,…)
{
 I1=s(x)->R, … || Q<-I2, BodyA(x,Q,…)->(r1,R2,R,…), o1<-r1, O2<-R2;
             … || V=t(I1,I2)->(o1,O2), BodyB(V,…)->(…)
}
```

The lack of the final ellipsis here indicates the removal of other rules which have incompatible matches for `V`, the other ellipses indicate that other external variables and other matches and other internal variables from the other matches are unchanged.

It can be seen here that the linear variable `V` is consumed by the match in the first rule, but as it is not read in a match in the second rule, it is passed in and becomes a local variable in its body. The added variable to variable assignments in the first rule are used in the place of any sort of substitution. If there were no other matches in the second rule, computation could commit to it and discard the first rule.

The variation on assignment absorption when an assignment to a non-linear variable is absorbed, so the match does not result in the variable being consumed, can be shown by considering the second rule in figure 15 with the assignment `m=d(i1,i2)`. Figure 18 shows this diagrammatically.

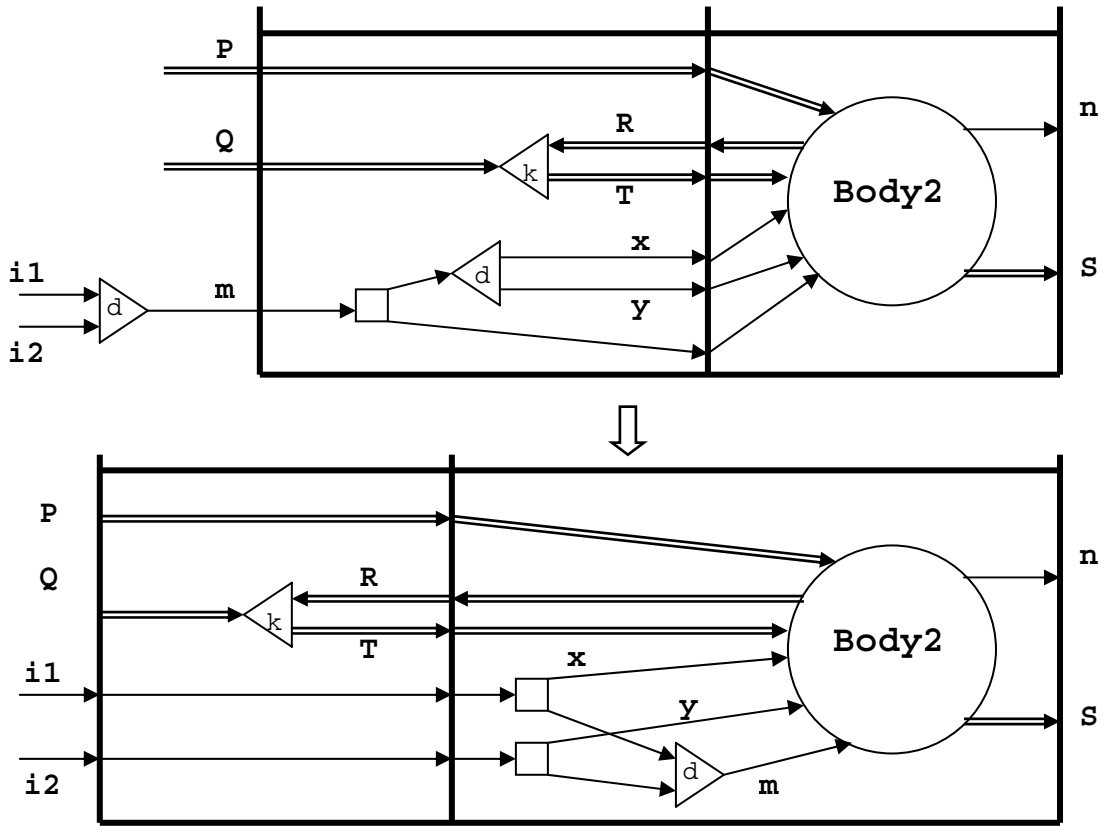

Figure 18. Absorption of an assignment to a non-linear variable.

6. Reduction of Duplicators and Erasers and Deletion of Irrelevant Tasks

The effect of variable to variable assignment as shown in the textual representation of figure 17 is covered by the following reduction rules when considered textually:

```
x<-y, y<-z              ⇒      x<-z, y<-z
x<-y, y=t(…)            ⇒      x=t(…), y=t(…)
X<-Y, Y<-Z              ⇒      X<-Z
X<-Y, Y=t(…)->(…)       ⇒      X=t(…)->(…)
```

The first two represent the effect of what is done through the duplicator in the diagram notation. The linking variable `y` is non-linear, so is not consumed in the reduction. In the second two, the linear variable `Y` is used and so consumed. In the diagram notation, the effect is shown simply by different labels at each end of the arrow.

The eraser, as shown in figure 16, can be used to establish a form of garbage collection. If a duplicator has one branch leading to an eraser, the duplicator can be removed. If a tuple is read only by an eraser, the tuple can be removed, with the eraser being passed to the variables to which it has read access, causing further application of eraser reductions. Figure 19 shows a diagrammatic representation of these two reduction rules for an eraser.

A refinement to this allows a whole process to be removed, without further evaluation of its rules if all the variables to which it has write access lead to erasers, and all the variables to which it has read access are non-linear. Figure 20 shows this diagrammatically.

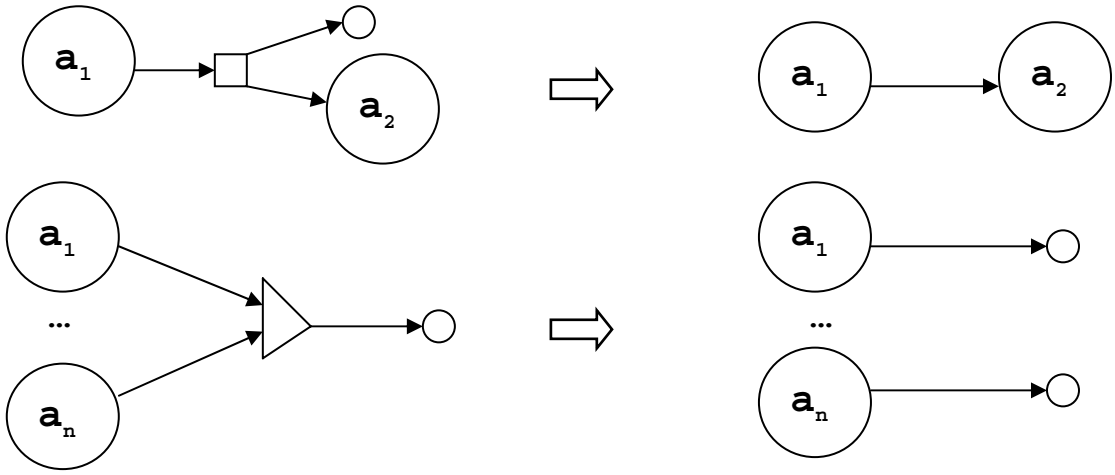

Figure 19. Reduction rules for erasers.

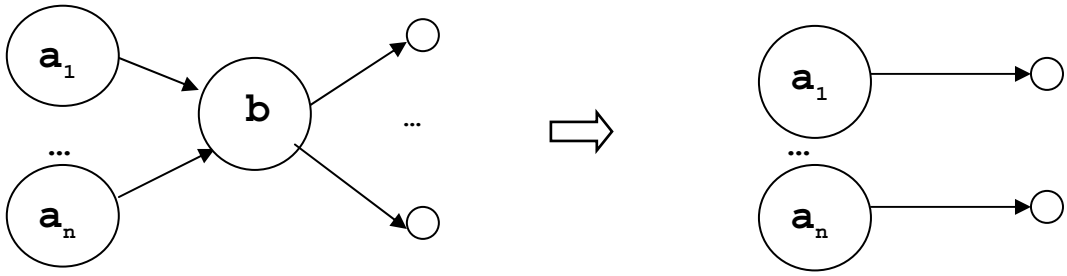

Figure 20. Deletion of an irrelevant process.

The process `b` here has no way of communicating to the world because none of the variables it writes to are used and none of the variables it reads are linear and so may lead to back communication. The deletion of `b` under these circumstances is the deletion of irrelevant tasks as proposed by Grit and Page [19].

7. Speculative Computation and Partial Evaluation

In the Aldwych-Core model, computation may commit to a rule if that rule has no matches. The body of the rule then becomes linked in directly with the world as an agent. Any assignments to variables to which the original process had read access may then be extruded to be absorbed by the readers of those variables. The absorption of the final assignment which removes the last match of a rule, and the commitment to that rule are separate reduction steps. However, as the only observable interaction an agent has with the world is to extrude an assignment, the steps of absorbing assignments and making a commitment are observed as a single step.

Assignments that are absorbed are trapped within rules, as shown in figures 17 and 18. The assignment is freed if computation commits to a rule because it then becomes part of the general network of agents. The assignment `V=t(I1,I2)->(o1,O2)` may be absorbed by `BodyB` in figure 17 and the assignment `m=d(i1,i2)` may be absorbed by `Body2` in figure 18 if computation commits to the rules which contain them. Viewing the process as an agent which continues as the body of the rule to which it commits, these absorptions are not observable, although they may lead to further commitments and an assignment which can be extruded being reached.

Suppose the absorption of `V=t(I1,I2)->(o1,O2)` by `BodyB` or the absorption of `m=d(i1,i2)` by `Body2` were done before the process had committed to the rule containing them. In terms of what is observable, it makes no difference if the absorptions were done at that stage. This would count as "speculative computation", that is computation which is done before it is known whether it is needed. This may be seen just as a useful trick to keep processors busy in a multi-processor system [20]. In artificial intelligence terms, it may be seen as a form of "planning": an agent making plans for the various interactions it may have in future with the world. Yet another way of viewing this is as more partial evaluation, with specialised versions of `BodyB` and `Body2` being produced as "residual code" for the circumstances when its is known that `V` will take the form `t(I1,I2)->(o1,O2)` and it is known that `m` will take the form `d(i1,i2)` with `I1`, `I2`, `i1`, and `i2` as variables to be assigned.

The idea of partial evaluation is that it produces specialised code for future use a multiple number of times, as it is expected the circumstances for which it is specialised will re-occur. Speculative computation, however, is a one-off computation of values before it is known whether they will be needed. In Aldwych-Core there is no difference in representation between the right-hand side of a rule interacting with the external variables of the process, and a computation in general interacting with the world. The body of a rule may be evaluated as if it is a full computation in the world. One way of thinking of this is as the imagined world of an intelligent agent. Alternatively, we may consider the world in which our agents are computing to be the imagined world of a super-agent.

In Anonymous Aldwych, an assignment may be absorbed by a recursive process and thus the absorption is realised as partial evaluation rather than speculative computation because the resulting rule set is not restricted to a once-only execution. Care must be taken to ensure there is what is known as a "stop criterion" in partial evaluation [21]. This is where a partial evaluation step is taken within the imagined world resulting from another partial evaluation step and the outer step is recognised as a variant of the inner. A simple case is when the two are identical except for variable names, a more complex case is any other situation where the relationship between the two is such that there is no guarantee the partial evaluation process will terminate. In some circumstances this leads to the partial evaluation being abandoned, in others it generates recursive residual code.

Although we have not yet integrated partial evaluation into the implemented version of Aldwych, nor formally described its operational steps as an aspect of the Aldwych-Core model, our earlier work on the partial evaluation of concurrent logic programming languages [22, 23] gives a lead on the techniques that could be employed.

Conclusion

The imposition of strict modes on logic variables enabled our graphical description to be developed. Our combination of moding with linearity enables us to have a guaranteed single writer for every variable, in a way that was not possible with earlier attempts to put mode information on logic variables, such as in Parlog [7]. Full moding on variables is not found in Erlang [5], a language with an inheritance from concurrent logic programming, and whose recent revival suggests this paradigm has something valuable to offer as we move into the era where multicore processors are the norm rather than something exotic.

The foundational core we have developed here has rich possibilities. We have already used it to demonstrate its basis for describing functional and object-oriented programming [13]. In section 7 of this paper we sketch some interesting further directions, stemming from the way concurrent evaluation handling unbound variables as "futures", which is a natural part of our model, relates closely to partial evaluation.

Implementations of Aldwych and Aldwych-Core have existed for longer than the models described in this paper. The formal model of Aldwych-Core derives from an abstraction of what were originally purely practical techniques used to build a reimplementation of a programming language first conceived as "parallel logic programming". What first appeared as limitations on the richer logic programming model came to be appreciated as strengths, enabling us to identify a new model for concurrent programming which is powerful, distinct from other foundational models, and straightforward enough to be describable using simple intuitive diagrams.

References

[1] M. Huntbach, The Core Language of Aldwych. *Communicating Process Architectures 2007*, pp 51-66, IOS Press, Amsterdam, 2007.

[2] E.Y. Shapiro and A. Takeuchi, Object Oriented Programming in Concurrent Prolog, *New Generation Computing,* 1 (1983) pp 25-48.

[3] J. Armstrong, A History of Erlang. *Third ACM Conference on the History of Programming Languages*, 2007, pp. 6.1-6.26

[4] J. Larson, Erlang for Concurrent Programming. *Comm. of the ACM* 52(3) 2009 pp.48-56.

[5] J. Armstrong *Programming Erlang: Software for a Concurrent World*. ISBN:193435600X, Pragmatic Bookshelf, Amsterdam, 2007.

[6] E. Gamma et al. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley 1994.

[7] S. Gregory, *Parallel Logic Programming in Parlog*. ISBN: 0201192412, Addison-Wesley 1987.

[8] S. Haridi et al. Efficient Logic Variables for Distributed Computing. *ACM Trans. on Programming Languages and Systems* 21(3) 1999 pp.569-626.

[9] R.H. Halstead, Multilisp: a Language for Concurrent Symbolic Computation. *ACM Trans. on Programming Languages and Systems* 7(4) 1985 pp.501-538.

[10] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*. ISBN:0-13-850587-X, Prentice Hall, 1990.

[11] C.A.R. Hoare. Communicating Sequential Processes. *Comm. of the ACM* 21(8) 1978 pp.666-677.

[12] R. Milner, J. Parrow and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Information and Computation* 100 (1) 1992, pp.1-77.

[13] M. Huntbach. Features of the Concurrent Programming Language Aldwych. *ACM Symposium on Applied Computing* 2003 pp.1048-1054.

[14] S. Abramsky. A Computational Interpretation of Linear Logic. *Theoretical Computer Science* 111 (1993) pp.3-57.

[15] M. Hyland, Game Semantics. In: A.M. Pitts and P. Dybjer, Editors, *Semantics and Logics of Computation*, Cambridge University Press, Cambridge (1997), pp. 131–194.

[16] Y. Lafont. Interaction Nets. *Proc. 17th ACM Symposium on Principles of Programming Languages* (1990) pp.95-108.

[17] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence* 8(3) 1977 pp.323-364.

[18] L. Beckman et al. A Partial Evaluator and Its Use as a Programming Tool. *Artificial Intelligence* 7 (1976) pp.319-357.

[19] D.H. Grit and R.L. Page. Deleting Irrelevant Tasks in an Expression-Oriented Multiprocessor System. *ACM Trans. on Programming Languages and Systems* 3(1) 1981 pp.49-59.

[20] F.W. Burton. Speculative Computation, Parallelism and Functional Programming. *IEEE Transactions on Computers* 34(12) 1985 pp.1190-1193.

[21] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. *Static Analysis 1998* G. Levi (ed) pp.230-245 Springer LNCS 1503, 1998.

[22] M. Huntbach. Meta-interpreters and Partial Evaluation in Parlog. *Formal aspects of Computing* 1 (1) 1989 pp.193-211.

[23] M. Huntbach and G. Ringwood. Partial Evaluation. *Agent-Oriented Programming: From Prolog to Guarded Definite Clauses*. Springer Lecture Notes in Artificial Intelligence 1630, 1999 pp.279-317.