Lambda Calculus in Core Aldwych

CPA 2015, University of Kent

Matthew Huntbach
School of Electronic Engineering and Computer Science
Queen Mary, University of London
matthew.huntbach@qmul.ac.uk

An Abstract Model for Computation with Agents acting Concurrently

- The universe consists of Agents which have shared access to Variables
- Each variable must have exactly one agent which can write to it (single writer)
- A variable may have any number of agents which can read it (but see linearity later)
- Once a variable has been written to, it cannot be written to again (single assignment)

Processes

- A process is an agent defined by a set of rules
- A rule has a left hand side (lhs) consisting of matches to variables which the process has read access to
- A rule has a right hand side (rhs) which must put all variables to which it has write access in a write position
- A process commits to a rule when all the rule's lhs matches are made, it converts to the rule's rhs
- A process commits to one rule and cannot backtrack

Compound Agents

- A process is one form of agent
- An assignment is one form of agent, it cannot be reduced further
- An alias is an agent which links a variable in read position to another in write position
- A collection of agents is itself an agent, termed a compound agent
- If a variable is in a read position in one component of a compound agent and in a write position in another, it may be purely internal to the compound agent
- A variable designated as linear must be internal if it in a read and write position in a compound agent

Assignment and Aliasing

- The values assigned to variables are tuples, consisting of a tag and further variables
- When an agent assigns a tuple to a variable, a variable in the tuple may be a new variable, in which case it must set up an agent to write it
- When an agent reads a variable, it obtains read access to the variables in the tuple and may pass on that read access in its own assignments
- An agent may directly assign a variable the value of another variable (aliasing)

Passing Read Access

```
agent(in1, in2)->out
{
 …
 in1=tagA(x) || out=tagB(in2,x);
 …
}
```

A reader of `out` then becomes a reader of `in2` and `x`, communication with the writers of `in2` and `x` are set up.

There is no requirement that `x` or `in2` have already been written to.

No agents are created, so the agent terminates.

New Variable and Agent

```
agent(in1, in2)->out
{
 …
 in2=tagC(x) || out=tagD(in1,y), sub(x)->y;
 …
}
```

A reader of `out` then becomes a reader of `in1` and `y`.

As `y` is a new variable, a new agent is set up to write to it.

There is no requirement that `x` or `in1` have already been written to.

The assignment to `out` takes place before `y` has been assigned a value, the ordering of the rhs has no relevance.

Indeterminacy

```
agent(in1, in2)->out
{
 …
  in1=tagA(x) ‖ out=tagB(in2,x);
  in2=tagC(x) ‖ out=tagD(in1,y), sub(x)->y;
 …
}
```

Both rules could be applicable, the choice of which to use is then indeterminate.

If `in1` has been assigned, the agent can choose the first rule without waiting to see if the second rule becomes applicable and vice versa

Multiple Reads (1)

```
agent(in1, in2)->out
{
 …
  in1=tagA(x), in2=tagC(y) ||
       out=tagD(z), sub(x,y)->z;
 …
}
```

The rule is applicable only if both `in1` and `in2` have been appropriately assigned.

Multiple Reads (2)

```
agent(in1, in2)->out
{
 …
  in1=tagE(x,in1a), in1a=tagF(y) ||
       out=tagG(z), sub(x,y)->z;
 …
}
```

The rule is applicable only if `in1` has been appropriately assigned, and the second variable in the tuple to which it has been assigned has also been appropriately assigned. The lhs can be written `in1=tagE(x,tagF(y))` as shorthand

Internal Assignment

```
agent(in1, in2)->out
{
 …
  in1=tagE(x,tagF(y)) ‖ z=tagG(x), sub(z,y)->out;
 …
}
```

The lhs can be written `sub(tagG(x),y)->out` as shorthand.

In normal evaluation there is no interaction between `z=tagG(x)` and `sub(z,y)->out` until the rule is committed to. Speculative evaluation (see later) would allow it.

Passing Write Access

```
agent(in1, in2)->out
{
 …
  in1=tagH(x) || sub1(x)->y, sub2(y)->out;
 …
}
```

The variable `out` is not directly assigned, instead a new process is set up to write to it.

The variable `in2` is ignored, but the variable `out` must be written to directly or indirectly

Multiple Writes

```
agent(in1, in2)->(out1, out2)
{
 …
  in1=tagH(x,y) || out2=tagI(z),
              sub1(x)->(w,out1), sub2(in2,y,w)->z;
 …
}
```

The variable `out2` is directly assigned, but `out1` is not.

Duplication

```
agent(in1, in2)->out
{
 …
  in1=tagJ ‖ sub3(in2)->y, sub4(in2,y)->out;
 …
}
```

Read access to the variable `in2` is duplicated here.

The tuple with tag `tagJ` has no internal variables.

Linear input variables cannot be ignored or duplicated.

Aliasing

```
agent(in1, in2)->out
{
 …
  in1=tagK || out<-in2;
 …
}
```

Read access to the `in2` is passed to the readers of `out`.

Or write access to `out` is passed to the writer of `in2`.

Continuation

```
agent(in, state)->out
{
 …
  in=tagL(x,in1) ||
       out=tagM(y,out1),
       agent(in1,state1)->out1,
       sub5(x,state)->(y,state1);
 …
}
```

A recursive process is created, it may be regarded as a continuation with state change from `in2` to `in2a`.

Anonymity and List Notation

```
(in, state)->out
{
 …
 in=[] ‖ out=[];
 in=[x|in1] ‖ out=[y|out1],
       *(in1,state1)->out1,
       (x,state)->(y,state1) {
                                …
                              }
 …
}
```

Message Sending

```
(in, state)->(out1, out2)
{
 …
 in=[] ‖ out=[];
 in=[tag1(x)|in1] ‖ out1=[tag2(y)|out11],
       *(in1,state1)->(out11,out2),
       process1(x,state)->(state1,y);
 in=[tag3(x,y)|in1] ‖ out2=[tag4(z)|out21],
       *(in1,state1)->(out1,out21) ,
       process2(x,y,state)->(state1,z);
 …
}
```

Stream Merging

```
(in1, in2)->out
{
 in1=[] || out<-in2;
 in2=[] || out<-in1;
 in1=[mess|in11] || out=[mess|out1],
      *(in11,in2)->out1;
 in2=[mess|in21] || out=[mess|out1],
      *(in1,in21)->out1;
}
```

Variables can only have a single writer, but this gives the effect of a stream with multiple writers

Back Communication

- As described so far, the agent which assigns a tuple to a variable must provide writers to the variables in the tuple, directly or indirectly
- With back communication, a variable may be assigned a tuple where the reader of the tuple becomes the writer of variables in the tuple.
- The writer of the tuple becomes the reader of these “back communication” variables.
- To maintain the single-writer property, a tuple with back communication variables must have exactly one reader.

Linear Variables

- As described so far, all variables must have exactly one writer but may have any number of readers: 0, 1 or more than 1.
- A linear variable (indicated by initial upper case letter) must have exactly one writer and also exactly one reader.
- An assignment of a tuple with back communication must be to a linear variable
- An assignment of a tuple which contains linear variables must be to a linear variable even if it has no direct back communication

Output of Tuple with Back Communication

```
agent(In1, In2)->Out
{
 …
 In1=tagA(x)->y || Out=tagB(z,In2)->y, sub(x)->z;
 …
}
```

The reader of `Out` becomes the reader of `In2`, a reader of `z` and the writer of `y`.

As `y` is a back communication variable, it must be in a write position in the rhs.

Input of Tuple with Back Communication

```
agent(In1, In2)
{
 …
 In1=tagC(x)->y || sub(x,In2)->y;
 …
}
```

The agent becomes the writer of `y`, and the writer of `In1` is the reader of `y`.

The variable `In2` must have a single reader to be the writer of back communication variables in a tuple it is assigned to.

The agent does not need any direct output variables.

Input of Tuples with Back Communication

```
agent(In1, In2)
{
 …
 In1=tagD(x), In2=tagE->y || y<-x;
 In1=tagF(X), In2=tagG->Y || Y<-X;
 …
}
```

The first rule provides one-way communication between the writer of `In1` and the writer of `In2` as it is through a non-linear variable.

The second rule provides potential two-way communication.

Aldwych

- The model of computation described, “Core Aldwych”, originated from concurrent logic programming
- Aldwych was an attempt to add syntactic sugar to concurrent logic programming to provide a richer structure to make it easier to use (“Aldwych turns into Strand”)
- The concurrent language Erlang had a similar origin

Core Aldwych

- Core Aldwych is an executable programming language
- Core Aldwych breaks down the unification of logic programming into single variables assignments and matches
- Concurrent logic programming did not take the final step of enforcing the single writer property throughout by the use of linear variables
- The possibility of multiple writers of variables was a major factor preventing the development of clear semantics for concurrent logic programming

Nothing Global

- The behaviour of any Aldwych agent has no depenency on external factors apart from assignments to variables to which it has read access
- There is no global clock, a process may commit to its rhs whenever it has a rule with an empty lhs (all matches made), but may wait an arbitrary amount of time before doing so
- There is no sequential composition
- There is no priority ordering on processes
- There is no global namespace, variables are defined by their readers and writers
- There is no global namespace for processes, all agents can be defined using anonymous rule sets and recursion

Factory Processes (1)

- The following replaces the need for an explicit named set of rules to append lists:

```
(S)
{
 S=[];
 S=[(In1,In2)->Out|S1] || *(S1),
    (In1,In2)->Out {
    In1=[]    || Out<-In2;
    In1=[H|T] || Out=[H|Rest], *(T,In2)->Rest
   }
}
```

- `S=[(A,B)->C|S1]` could be used rather than `append(A,B)->C`

Factory Processes (2)

If `S` were read by the following process:

```
(S)
{
 S=[] ||;
 S=[(In1,In2)->Out|S1] || *(S1),
    (In1,In2)->Out {
    In1=[]    || Out<-In2;
    In2=[]    || Out<-In2;
    In1=[H1|T1], In2=[H2|T2] ||
       Out=[H1,H2|Rest], *(In1,In2)->Rest
   }
}
```

then `S=[(A,B)->C|S1]` would give a different sort of stream joining, so setting a particular reader of `S` gives a higher order effect

Synchronisation

Synchronisation can be implemented by back communication.

The following process zips together two input streams of numbers, adding together their respective elements:

```
(In1,In2)->Out
{
 In1=[] || Out<-In2;
 In2=[] || Out<-In1;
 In1=[(x1)->ret1|In1a], In2=[(x2)->ret2|In2a] ||
    ret1=done, ret2=done, Out=[(x)->ret|Outa],
    (x1,x2,In1a,In2a,ret)->(x,Outa) {
       ret=done || x<-x1+x2, **(In1a,In2a)->Outa;
    }
}
```

It will not add the next pair of numbers until confirmation has been received of the sum of the previous pair being read.

Asynchronous version

The following process zips together two input streams of numbers, adding together their respective elements:

```
(In1,In2)->Out
{
 In1=[]|| Out<-In2;
 In2=[]|| Out<-In1;
 In1=[(x1)|In1a], In2=[(x2)|In2a] ||
    Out=[(x)|Outa], x<-x1+x2,
    *(In1a,In2a)->Outa;
}
```

Unlike the previous version, this process will zip the numbers without waiting for their sums to be read.

Model of Computation

- Core Aldwych is not suggested as a language for direct programming
- It is suggested as a model of computation for concurrent programming
- A model of computation is used to provide a semantics for more complex languages
- A model of computation sacrifices usability in order to gain simplicity: full and accurate description of its behaviour is very short
- Features which make code easier to understand in human terms are translated to the model

Translations to Models

- Translation of higher level features to models of computation establish precisely what those features mean
- The translation may identify possible variations of the features
- The model should be able to cover identified variations of the higher level features, so that a different choice in the feature behaviour gives a different translation
- Operational rules on the model should reflect operational rules on the higher level language

Models of Concurrent Computation

- Introducing concurrency into computation means assumptions based on sequential behaviour no longer apply
- Classic example with mutable variables:

```
{var x=0; x=x+1, x=x+2; return x}
```

  could return `1`, `2` or `3`
- A model of concurrent computation would need to be able to show this
- It would need to cover variations in the exact handling of mutable variables which would vary the possible behaviour

Core Aldwych Representation of Mutable Variables

A variable is represented by a process which inputs a stream of messages representing get and set operations on the variable, and outputs a stream of messages representing access to the value it holds:

```
(S)->Val
{
 S=[] || Val=[];
 S=[()->Val1|S1] ||    Val=[], *(S1)->Val1;
 S=[(Val1)|S1]   ||    merge(Val1,Val2)->Val,
                       *(S1)->Val2;
}
```

The second rule handles a set message, the third rule handles a get message, the first rule terminates the variable.

General Core Aldwych Representation

- An entity is represented by an agent which inputs a stream of messages representing communications sent to the entity by other entities
- It has an output stream for each of the entities which form part of it
- Shared access to entities is represented by joining separate input streams to them
- Concurrent access is represented by indeterminate merger of streams to it
- Sequential access is represented by appending streams to it

Variation Example

The following is a slight variation on the previous representation of a mutable variable:

```
(S)->Val
{
 S=[] || Val=[];
 S=[()->Val1|S1] ||    Val=[], var(S1)->Val1;
 S=[(Val1)|S1]   ||    append(Val1,Val2)->Val,
                       *(S1)->Val2;
}
```

The merger of streams to the representation of the variable's value is replaced by an append. The consequence is that when a value is obtained from a variable more than once, each access locks it from other accesses through the same variable.

Operational and Denotational Semantics

- Running Core Aldwych code with an interpreter which allows explicit choice of options in indeterminacy reveals all possible code behaviour
- A smarter way of doing this is to employ code transformation to reduce Core Aldwych code to a normal form
- For example, the initial Core Aldwych code for

  ```
  {var x=0; x=x+1, x=x+2; return x}
  ```

  reduces to the simplest code for indeterminately returning `1`, `2` or `3`.

Partial Evaluation

- Partial evaluation is executing code with some variables unbound, this is a normal aspect of Core Aldwych's behaviour, halting only when a variable needs its value matched
- The operational model of Core Aldwych involves "assignment absorption", a process which requires more than one assignment to commit is modified when it has received just one
- To obtain all-solutions code, a process should only commit if it has only one possible rule, otherwise it suspends with an empty-lhs rule

Speculative Evaluation

- Speculative evaluation means evaluating all possibilities before a commitment is made, so the code on the rhs of each rule in the suspended process is similarly evaluated
- Assignment absorption brings assignments into the rhs, possibly allowing it to be reduced
- Recognising an assignment absorption as identical in all but variable names to a previous one enables it to be replaced by the rules of the previous one, or by a recursive call if it is internal to those rules

Composition

- Two process may be composed into one which has the read and write variables of both less linear variables which one reads and the other writes, these become inner variables
- The rules of the composed process are the rules of the two processes with any rule which has a match to an internal variable removed, and each rule from one process has the call to the other process added to its rhs
- Recognising a composition as identical in all but variable names to a previous one enables it to be replaced by the rules of the previous one, or by a recursive call if it is internal to those rules
- Our aim is to combine partial evaluation, speculative evaluation and composition to transform any compound agent to a normal form (its denotational semantics).

Lambda Calculus

- Lambda calculus is recognised as the standard model for sequential computation
- Lambda calculus resembles Core Aldwych in having no defined order of reduction (so potentially parallel), and no global nameset
- Unlike Core Aldwych, lambda calculus makes no distinction between processes and values, a variable may be set to a function, and functions passed as arguments to other functions (higher order functions)
- Lambda calculus is based on computation as about taking input and evaluating to a result rather than interaction
- Lambda calculus does not have indeterminacy, an individual function application cannot choose alternatives

Higher Order Programming

- A criticism of logic programming, which applies to Core Aldwych, is that as the process rules are separate from the process values, it does not offer the flexibility of higher order programming
- Showing that lambda calculus can be represented in Core Aldwych shows there is no need to complicate the model with higher order aspects
- The factory process technique gives the effect of higher order programming
- A function is passed to a Core Aldwych process in the form of a back communication variable used as a stream of calls to the function

Representing a lambda expression

The lambda expression λx.exp, where exp contains a single free variable is represented by `L` where the reader of `L` is as below, and `V` represents the free variable (a stream read by the process which represents its value):

```
(L)—>V
{
 L=[] || V=[];
 L=[Call|L1], Call=(Res)—>X ||
    exp(Res)—>(X,V1),
    *(L1)—>V2, merge(V1,V2)—>V;
}
```

Then (λx.exp) n is given by `R` in `L=[(R)—>N|L1]` where `N` represents n, with `L1` used for λx.exp subsequently.

Higher order lambda expression

The lambda expression λx.λy.exp, where exp contains a single free variable is represented by:

```
(L)->V
{
 L=[] || V=[];
 L=[Call|L1], Call=(Res)->X ||
    (Res)->(X,V1) {
      Res=[] || X=[], V1=[];
      Res=[C|Res1], C=(R)->Y ||
          exp(R)->(X1,Y,V3), *(Res1)->(X2,V4),
          merge(X1,X2)->X, merge(V3,V4)->V1;
    },
    *(L1)->V2, merge(V1,V2)->V;
}
```

Input-Output Reversal

- An expression is represented by a process which inputs a stream of messages, each message representing an individual access to the expression
- Access to that expression is then represented by output of streams which are merged to form the input stream
- If a process represents a function, the form of the message sent to it representing a call to the function is `(Res)->Arg`
- Here `Arg` is an output of messages to the representation of the argument, and `Res` is an input of messages to the process created which represents the result of the function call

Y Combinator

- The Y combinator is used to give recursion in lambda calculus. It is defined as a function Y where Y f, that is Y applied to f, evaluates to the expression f (Y f).
- If `F` represents a λ-expression f, or more strictly `F` is a variable set to a stream read by the process which represents f, then Y f is represented by the stream `R` with:

```
F=[(R1)->F1], merge(F1,R)->R1
```

Lazy Evaluation

- `F=[(Z)->X]` represents `Z` set to a call of the function represented by `F` with `X` its argument and no continuation for further calls to that function
- If we have

```
lazyapply(Res)->(F,X)
{
 Res=[] || F=[], X=[];
 Res=[Mess|Res1] || F=[(Z)->X], Z=[Mess|Res1];
}
```

then `lazyapply(Z)->(F,X)` is the call applied lazily, `F` will only be sent a message if the result of the call is accessed.

Representation of Expressions

With lazy evaluation let z=x+y in exp end is represented by:

```
(Z)->(X,Y)
{
 Z=[] || X=[], Y=[];
 Z=[M|Z1] || X=[()->xval], Y=[()->yval],
     zval<-xval+yval, constant([M|Z1],zval);
},
exp(E)->(Z,…)
```

where `exp(E)` represents exp, with ... its free variables, and:

```
constant(S,val)
{
 S=empty() ||;
 S=[Mess|S1], Mess=()->ret || ret<-val, *(S1,val);
}
```

Types

- The representation of constants as a process which takes messages of the form `()->val` maintains the principle that all entities are represented by processes which take a stream of messages, but it is clearly inefficient
- Partial evaluation can transform to more efficient code
- The messages that can be sent on a stream indicate the type of expression it represents
- A function takes messages of the form `(Res)->Arg` but if it is a function from integers to integers both `Res` and `Arg` must be lists of tuples of the form `()->val` where `val` must be an integer.

Message Order (1)

The lambda expression λx.λy.(f x)+(f y) is represented by:

```
(L)—>F
{
 L=[] || F=[];
 L=[Call|L1], Call=(Res)—>X ||
    (Res)->(X,F1) {
      Res=[] || X=[], F1=[];
      Res=[C|Res1], C=(R)->Y ||
          F3=[(Fx)->X1,(Fy)->Y],
          Fx=[()->vfx], Fy=[()->vfy],
          sum<-vfx+vfy, constant(R,sum),
          *(Res1)—>(X2,F4),
          merge(X1,X2)->X, merge(F3,F4)—>F1;
    },
    *(L1)—>F2, merge(F1,F2)—>F;
}
```

Message Order (2)

- Inside this process, the assignment:

```
F3=[(Fx)->X1,(Fy)->Y]
```

  represents the calls `f x` and `f y`.
- However, this imposes an order on the calls, they would be made the other way round if it were:

```
F3=[(Fy)->Y,(Fx)->X1]
```

- With the general format given previously it should be:

```
F5=[(Fx)->X1], F6=[(Fy)->Y], merge(F5,F6)->F3
```

- Does it matter?
- Not in lambda calculus where a function applied to a particular argument always gives the same value (except if a function application is non-terminating, and lazy evaluation is being used)

Interaction

- We have already seen how Core Aldwych can represent mutable state, with the mutable variable representation
- More generally, Core Aldwych can interact with anything, so long as it has a Core Aldwych interface (through single-writer single-assignment variables)
- Lambda calculus is a closed system, Core Aldwych is an open system
- The order of messages is an issue when the messages may be interacting with a mutable world
- As already seen, in some cases we would want to replace the indeterminate merger of streams with appending streams to give closer control

Y Combinator Revisited (1)

- With the Y combinator representation suggested previously where Y f is represented by the stream `R` with:

```
F=[(R1)->F1], merge(F1,R)->R1
```

  when a call is made to the recursive function Y f it is done through a message sent on `R` which is passed to `R1` then read through `F` which generates further messages through `F1`.
- So when Y f is applied to x, the recursive calls it generates are mixed up with Y f applied to other arguments and the recursive calls those applications generate
- We could not use

```
F=[(R1)->F1], append(F1,R)->R1
```

  as there could be deadlock with the append process waiting for `F1` to be assigned and that requiring `R1` to be assigned

Y Combinator Revisited (2)

- The following works to represent Y f where the stream `R` is used to send messages to Y f and the function f is given by the reader of the stream `F`:

```
(R)->F
{
 R=[] || F=[];
 R=[Call|More] ||  R1=[Call|F1],
                   F=[(R1)->F1|F2],
                   *(More)->F2;
}
```

- The more general point is that this suggests Core Aldwych as a more fundamental model as it captures variations in behaviour which lambda calculus does not

Summary

- Core Aldwych is presented here as a model of computation comparable with other widely used models of computation
- Its distinguishing factor is interaction between processes expressed through shared single-writer single-assignment variables
- As lambda calculus is the longest established and most widely used model of computation, it is important to show that Core Aldwych can model lambda calculus
- Lambda calculus assumes computation is determinate with a single end goal, interesting questions are raised when it is put into an environment involving interaction and indeterminacy

Current work

- A sequential implementation of Core Aldwych exists as an executable programming language with simulated concurrency
- A representation in Core Aldwych of procedural programming with mutable variables has been implemented
- A representation of lambda calculus in Core Aldwych has been implemented, with several variants such as optional lazy evaluation
- A representation of channel-based computation has also been implemented in Core Aldwych

Future work

- The original aim of Aldwych was to provide a practical language for concurrent programming, with Erlang (which has a similar background) its nearest equivalent
- It would be good to experiment with Core Aldwych on real concurrent architecture
- Some experimentation has been done with partial evaluation of Core Aldwych, originally to deal with efficiency issues in translation from Aldwych
- If partial evaluation could be brought to the point where any Core Aldwych program could be translated to a normal form, it would provide an effective denotational semantics for concurrent programming