

The U*-model

Peter Seymour

15 November 2008

Abstract

This paper outlines the U*-model of computing within the context of the Breathe System. It is presented as a universal model of computing and several examples are given demonstrating its flexibility. Given its history that predates the Breathe System it can alternatively be viewed in other contexts as purely universal.

1 History

The U*-model was constructed prior to the Breathe System (see [1] and [2]) so the parts concerning a model's context were specified in general terms. However, these general terms lead directly to the interfaces used in the Breathe System so the definitions used here for types, register synchronisation and signals are the formal definition of a model interface.

This paper shows the original universal model as one of the models that the Breathe System supports. Originally the domain of computing was viewed as model-centric but with the advent of the Breathe System the focus has changed to the structure of the environment with models being interchangeable components. Since the text of the following sections has not been greatly altered it should be read with this mind. For instance the notation used reflects the original notation which has since been superseded within the Breathe System.

2 Introduction

The purpose of the U*-model is to effectively describe a very large class of computing systems. Several examples are given that show how universal this model really is. The aim is to provide a working model in which old and future computing systems can co-exist.

This covers software, hardware and abstract models. By describing all of these in common terms permits this to happen.

The basis of the U*-model centres around types, stacks and registers. A type is simply a name given to a set of states which any instance of it can take. The states are ordered as a cycle with an initial state chosen. A stack represents a finite ordered collection of type instances with access only to the top most elements. The set of stacks is enumerated. Registers correspond to individual type instances but allow the states to be manipulated through register machine instructions.

The U*-model executes a series of operations on the stacks moving and combining their elements or acting on the states. Communication with the wider Breathe System is achieved by synchronising some of the registers with device registers.

3 The U*-model

Technically the U*-model is a family of models since the set of types and number of stacks are parameters. However, in this text the term *model* refers to any member of the family i.e. the set of types and number of stacks are fixed. A *model instance* is then the specific operations and instructions performed when executing, it is analogous to a software program.

3.1 Types

There are a finite number of primitive types coming in two varieties: built-in and user-defined. The built-in types are *pointer*, *deconstruct*, *valueof*, *clone*, *action*, *break*, *reverse-swap* and *null* or in symbols $\&$, $*$, $\$$, $\%$, $\@$, $!$, \mathbb{S} , $-$, respectively. I refer to the user-defined types as u_1, \dots, u_k and these are parameters in the U^* -model family. An element of type t can take on one of $|t| > 0$ states which in turn is associated with an element of $\mathbb{N}_{|t|}$ and yields an ordering to the type instances. This ordering is required by the register machine. $|t|$ is the *size* of t . With the exception of *pointer* the built-in types have *size* 1 which is referred to as being a *tag* type. The user types and *pointer* type can have any finite *size*. Again these *sizes* are parameters in the U^* -model.

The main class of type constructors is *array_n* or $[\]_n : t_1, \dots, t_n \rightarrow [t_1, \dots, t_n]$ which allows $n \geq 0$ (but finite) heterogeneous elements to be ordered where t_k is any type. The second is *structure_n* or $\{ \}_n : u_{i_1}, \dots, u_{i_n} \rightarrow \{u_{i_1}, \dots, u_{i_n}\}$ that defines a new type from a finite set of other types (user-defined or built-in). A *structure* type is atomic and its type is immutable hence it will be treated like a user type u_i for some i . Two structures are of the same type if and only if they correspond to the same user type. Hence *structures* use *name* equivalence. An *array* type can be altered so will always be written $[t_1, \dots, t_k]$. Two *array* types $[t_1, \dots, t_k]$ and $[t'_1, \dots, t'_{k'}]$ are the same if and only if $k = k'$ and $t_i = t'_i$ for each i . Hence *arrays* use *structural* equivalence.

These are recursive definitions but the closure of all the types produced is the countably infinite set of available types, T .

The notation used for types is of the form $x:t$ for an instance x of type t . If t has *size* 1 it may be written as $:t$. Examples are $10:\&$, $:\ast$ and $21:int$ for a user type *int*.

Since a *type* of *size* n can be thought of as the first n ordinals it is declared as $t=n$. A more realistic example including *structures* is:

```
year=10000
month=12
day=31
```

```
date={year,month,day}
hour=12
minute=60
second=60
time={hour,minute,second}
datetime={date,time}
```

With example instances of $\{11,5,0\}:time$ and $\{\{2006,08,15\},\{18,06,15\}\}:datetime$.

3.2 Stacks

A stack is initially empty and since it is unbounded will never overflow. Under various conditions elements are popped from the stack which may cause an underflow. In this case the model instance terminates in error. The number of stacks appearing in a given U^* -model is equal to the *size* of the *pointer* type. Therefore each stack corresponds to a state of the *pointer* type which enumerates them.

3.3 Push and Op

There are two manipulations that can be performed on a stack. The first is to push an element on to the top.

```
s <- 21:int
s <- t
s <- :*
s <- [ :add, 1:int, 2:int ]
```

As shown above where s and t are stacks and the use of a stack on the right hand side denotes a *pointer* to it.

The second very important operation is simply called *op*.

```
s()
```

It treats the top element as a functional taking none, the first or both of the next two elements as arguments. The top element and the arguments as required are removed, the operation performed and the results if any pushed back. It is necessary to

define it this precisely as the *op'd* stack might be referenced directly or indirectly. The nature of all the behaviours is broken down by the top element and the next element is understood to be the argument.

pointer: The argument is pushed on to the stack pointed to by the *pointer* and does not return an element.

deconstruct: For an argument of type *pointer* the top element of the pointed to stack is removed and nothing returned. For an argument of type *array* each element of the array is returned back to the stack from right to left so that the top element is the first element of the old *array*. For any other type nothing is returned.

valueof: For an argument of type *pointer* a copy of the top element of the pointed to stack is returned. For all other types the argument is returned.

clone: Two copies of the argument are returned.

action: The argument is subjected to the function @ and the result returned. This delegates to the register machine and is explained later.

break: This does not take an argument nor return a result. It increments the *break* counter.

reverse-swap: Two arguments are taken and nothing returned. If exactly one argument is of type *pointer* then the pointed to stack is cleared and the other argument pushed onto it. If both arguments are of type *pointer* then the pointed to stacks have their contents swapped in reverse order. Namely the top element of each stack now appears as the last element on the other stack. In the case that both arguments point to the same stack then its contents are reversed. In all other cases nothing happens.

null and u_k : An argument is taken but nothing returned.

array: The argument is appended to the right of the *array* element which is then returned.

Only *break* does not take an argument and it could take a dummy value, however, it complicates descriptions without any gain. The list could be shorter in that *clone* is redundant.

```
t <- s
t <- :$
t()
t <- s
t()
```

This is equivalent to cloning the top element of *s*. It is significant that *s* appears nowhere on the left since this fragment would not work on a dynamically determined stack. More on this later as we see that only a fixed set of stacks is needed to perform arbitrary manipulations on all available stacks.

3.4 Routines

A *routine* is a named sequence of *push* and *op* instructions over a fixed set of stacks.

3.5 Actions

There is a partial function @ mapping elements from the set of types to itself. By default @ is defined nowhere but definitions can be supplied on a type by type basis. When the *action* operation is performed @ is applied to the argument. If @ is not defined for the type of the given argument then the model instance halts in error. A definition is specified using the form $x \rightarrow y$ where *x* and *y* are elements with *non-structural* primitives named in the case they are not *tags*. For example:

```
[ :add, x:int, y:int ] -> r:int
[ :get_hours, {h,m,s}:time ] -> hh:hour
```

Each *action* comprises a sequence of optionally labelled instructions on the registers which are the named primitives in the argument and result. In the first example these are *x*, *y* and *r* whereas in the second they would be *h*, *m*, *s* and *hh*. Two of the instructions are from a modified register machine.

```
l: r' -> r
```

This changes the state of `r` to the next state in some fixed enumeration. It is labelled 1 and will continue to the next instruction unless it is the last. If `r` was previously in the final state it is changed to the first state which is special and denoted by 0. All the registers corresponding to primitives in the result are initially in state 0. The analogy here is addition by 1 modulo the *size* of the type which is carried over by the notation $|r|$.

A traditional register machine would also provide a decrement instruction with a test to prevent the register value dropping below zero. This is not necessary since $|r| - 1 \geq 0$ applications of `r' -> r` achieves a decrement.

A test is still needed to provide conditional behaviour and is written:

```
1: r = 0 ? 11 : 12
```

The register machine will next execute the instruction labelled 11 if `r` is in its first state otherwise the instruction labelled 12 will be executed.

When the point of execution reaches the end of the instructions the *action* terminates. A label may be supplied after all the instructions to allow an early exit.

An example that resets a register to its first state would look like:

```
continue:
  r' -> r
  r = 0 ? exit : continue
exit:
```

Notice how this makes no assumptions about $|r|$ and forms a generic solution which will apply across all types.

3.6 Persistent and external registers

A register or structure of registers in the case of a structural type may be declared. These can be referenced by any *action* and maintain persistent state across *action* applications. Initially all the primitive registers are in state 0. A reasonable use might be to store the last time an event occurred or generate a temporary register for some calculation:

```
{last_h,last_m,last_s}:time
temp:int
```

Any of these registers can be *bound* to identical *external* registers that fall outside of the U*-model. Within the context of the Breathe System these are held by devices. It is assumed these *external* copies are monitored or updated by some process independently such as a device. An *action* may *synchronise* the *internal* copy in either direction depending on how it is declared.

```
"char_out" <- char_out:character
"char_in"  -> char_in:character
```

This says that `char_out` can be *synchronised-out* while `char_in` can be *synchronised-in*. If it is required that *synchronisation* occur both ways then a double headed arrow is used.

In an *action*, the following instructions:

```
? <- char_out
```

will copy the state of the *internal* register to the *external* copy whereas:

```
? -> char_in
```

will copy the state of the *external* register to the internal copy. The question mark representing the *external* register. The naming indicates that some *external* process will monitor `char_out` and perhaps display a character. In the case of `char_in` it could be updated by some input source. The use of quotes intentionally highlights that it lies *outside* the U*-model and a meaning only makes sense in context. It is not permitted to *synchronise* a register in a direction it is not declared for.

For a structural type say,

```
"today" -> {y,m,d}:date as today
```

an overall name is needed to express the *synchronisation* of the whole *structure*. That would be done by:

```
? -> today
```

3.7 Signals and Interrupts

Routines are processed from top to bottom with no ability for an alternative control flow. *Signals* by contrast are composed from routines and allow some degree of flow control. A *signal* is in essence a production over the *routines* and *signals*. These are expressed in a restricted BNF notation. Suppose there are two *routines* R1 and R2 then they can be chained together in a signal S:

$\langle S \rangle ::= R1 R2$

This *signal* itself can be used from another *signal* T by:

$\langle T \rangle ::= R1 \langle S \rangle$

T will then process the instructions in R1 followed by R1 (again) and then R2. A definition can be recursive.

$\langle T \rangle ::= R1 \langle T \rangle R2$

At points of recursion the *break* counter is referred to. If no *breaks* have been issued the recursion will occur. If the *break* counter is positive then it is decreased by 1 and the next element from the production processed or the parent production consulted in the case of tail recursion. Initially the *break* counter is 0. In effect the productions determine a cyclic graph with the nodes being the *routines*. A point of recursion is identified by an edge leading back to a previously seen node. Multiple *breaks* may be issued before a point of recursion so it is possible to *exit* cycles from any level. When the last *routine* of the initial production is processed the model instance halts. Processing begins at an initially chosen *signal*.

The signals do not necessarily have to form a connected graph meaning there are multiple ways to *start* the model instance. Therefore one model instance may be used in a variety of ways.

There is a special *signal* that processes *interrupts*. When this *signal* is encountered in a production the model instance *yields* and an undetermined but finite sequence of signals is substituted in place by an *outside* process e.g. Breathe devices. The substitution can vary from one occurrence to another. It is as if

the model instance has allowed itself to be *interrupted* and perform some *signals* it does not directly control. In a production this is written using a question mark:

$\langle T \rangle ::= R1 ? \langle T \rangle$

So after each time R1 is processed the model instance *yields* and processes zero or more other *signals*. A question mark is used as it was for an *external* register to highlight the two areas reliant on external context. It is worth noting that a U*-model explicitly exposes itself to its context and can choose not to or to what degree it does. An *external* process can never *interfere* with a model instance in a previously not requested manner.

4 Examples

This section sketches out the basics of using the U*-model to describe some example machines, runtimes and programs. Each one is quite distinct and gives a good indication of the capabilities available. For actual use they would need to be extended, for instance none of them deal with input or output but adding suitable Breathe devices would be easy, see [3]. This and other extended functionality would most likely be added by including various *actions*, however, creating a large list here does nothing to aid readability.

4.1 A Turing Machine

A Turing machine is an abstract computing device created by Alan Turing primarily to explore the limits of computation [4]. It comprises an infinite tape of symbols and a head that can move over the tape one step at a time reading and writing symbols. The process is driven by internal state changes referring to a finite set of deterministic rules. This description of a Turing machine uses three stacks to represent the tape. One holds the cell under the head and the other two hold the cells to the left and the right. Moving the head involves transferring cells from left to right via the head or vice-versa. Two additional stacks are used to hold the current state and direction. These are declared as follow:

```

left
head
right
state
direction

```

Each state and direction will be denoted by a *tag* type. In the case of direction these are:

```

L = 1
R = 1
HALT = 1

```

The states will be specific to a given problem so an example set will be shown later.

To move the head requires shifting elements from the `left` and `right` stacks via the `head` stack. Since both of these stacks are finite but are being used to represent half of an infinite tape each means *null* elements must be inserted to avoid running off the end. The end of the tape is marked by `[:-]` such that a sequence of operations will insert a *null* element when it is present but will leave *tag* types unaltered.

The full routine to move the head to the right requires the following sequence of instructions namely `go_right`:

```

right <- :%      /extend
right()
right <- :*
right()

head <- left     /shift
head()
right <- head
right()

```

In the first block the element to the right is *cloned* and then *deconstructed*. For a *tag* type this has no overall effect but for the marker it leaves a copy of `:-` behind. In the second block the first pair of instructions take the top element on `head` and pushes it onto `left`. The final pair move the top element of `right` onto `head`.

Similarly for `go_left`:

```

left <- :%      /extend

```

```

left()
left <- :*
left()

head <- right   /shift
head()
left <- head
left()

```

The premise is to first process all Turing instructions that move the head to the right. When a direction change is encountered this loop is broken and all instructions moving left are processed.

To react to direction changes the following four actions are used. They have no associated instructions since this runtime only uses type information. By using *op* on the resulting type a *break* will be issued if the directions differ otherwise nothing happens.

```

[:L, :R] -> :!
[:R, :L] -> :!
[:L, :L] -> :$
[:R, :R] -> :$

```

There are no *actions* using the `HALT` type so when this is encountered the model instance terminates.

Each Turing instruction is represented by an *action*. The current state and cell value is mapped to the next state, new cell value and direction for the head to move. Since Turing states and directions are types no instance values are considered. A Turing instruction has the form:

```

[:s, :c] -> [:ss, :dd, :cc]

```

Here `s` is the current Turing state and `c` the cell value under the head. On the right `ss` is the new Turing state, `cc` the cell value to be written and `dd` the direction to move the head. These are processed by the `compute` routine:

```

state <- head    /move the state
state()

head <- []       /combine state and cell
head()
head()

```

```

head <- :@      /process the Turing
head()        /instruction

head <- :*      /unpack
head()

head <- state  /save state
head()

direction <- head /copy new direction
direction <- :$
direction()

direction <- [] /make transition array
direction()
direction()

direction <- :@ /process
direction()

direction()

head <- direction /save direction
head()

```

First the state is moved to the head stack. It is then combined with the current cell to form an *array*. The @ function is applied yielding the right hand side of a Turing instruction. This composite result is *deconstructed*. The new state is moved to the state stack. The new direction copied onto the direction stack. Note the old direction sits below it whereas the state has been replaced. These two directions are combined into an *array*. The @ function is again applied. The resulting element is *op'd* possibly issuing a *break*. The new direction is then moved from head to direction.

Now finally all this comes together with a few signals:

```

<r_cycle> ::= go_right compute <r_cycle>
<l_cycle> ::= go_left compute <l_cycle>
<cycle> ::= <r_cycle> <l_cycle> <cycle>
<start> ::= prepare load <cycle>

```

The start signal first processes a routine called load that will configure the initial tape contents and puts the Turing machine into its initial state. The routine prepare assigns the initial direction that the head will move:

```

direction <- :R /initial direction

head <- :-
left <- [:-] /end markers
right <- [:-]

```

Then <r_cycle> and <l_cycle> will be processed indefinitely unless the Turing machine halts. Each one in turn moving the head and computing a Turing instruction. The model instance terminates when a Turing instruction returns a direction of HALT and unsuccessfully attempts to find an *action*.

A concrete example of unary addition can now be easily described. The tape contains either blank cells represented by *null* elements or marked cells using a *tag* type mark. The Turing states are S1, S2 and S3.

The load routine for 2 + 1 is:

```

right <- :mark
right <- :-
right <- :mark
right <- :mark

state <- :S1

```

Initially the head begins one to the left of the first marked cell. The Turing instructions are:

```

[:S1, :-] -> [:S1, :R, :-]
[:S1, :mark] -> [:S2, :R, :-]
[:S2, :-] -> [:S3, :R, :mark]
[:S2, :mark] -> [:S2, :R, :mark]
[:S3, :-] -> [:S3, :HALT, :-]
[:S3, :mark] -> [:S3, :R, :mark]

```

This is a pleasing description of a Turing machine since each component is represented cleanly and with minimal syntax. Indeed it wouldn't look out of place as an abstract description in its own right.

The *signals* and *routines* can be considered a runtime for the domain of Turing machines. While the

direction *actions* and types are analogous to a hardware CPU and form a machine. The remaining *actions* corresponding to the Turing instructions and the load routine form an instance of a program.

4.2 Whirlwind I

The Whirlwind computers were developed in the late 1940s with the aim of obtaining high speed computing with a simplified instruction set. The instructions were known as orders. By restricting the orders available they could be encoded as numerical values in a uniform way and stored in main memory with other data. This example is based on the R-127 report from 1947 [5] but no claim is made that it is a faithful representation. Indeed it differs in one key aspect: This representation encodes orders as a type different from numeric values so the storage registers don't always contain values. More about this is mentioned at the end. It does, however, make for an elegant representation of the basic ideas of the Whirlwind I system (WWI).

WWI had 2048 storage registers that were the main memory of the system. All processing was done via the arithmetic element using a small number of internal registers. These are the accumulator (AC), two arithmetic registers (AR and BR) along with a program counter (PC). Numerical values were 16-bit binary numbers using 9's complement where the conversion between sign only requires flipping each bit.

This suggests a type:

```
value=65536
```

Then the storage registers as stacks:

```
S[2048]
```

On the register machine side there are four registers representing the internal registers of the arithmetic element:

```
AC:value
AR:value
BR:value
PC:value
```

An *action* is used to retrieve the location of the next order where *pc* is a *tag* type and the value of *p* will equal the value of PC upon completion:

```
:pc -> p:&
```

The first seven orders all have the same representation and invoke an *action* to alter the internal registers. Each *action* with the exception of one increments PC as its last change. This advances the point of execution to the next order. For instance *ca S(x)* will clear AC and add the contents of the storage register numbered *x*. PC is then incremented. By taking a *tag* type *ca* the following *action* is formed that will perform the necessary changes to the internal registers:

```
[v:value, :ca] -> [:$]
```

Here *v* is the value on top of *S[x]* with the order encoded as:

```
[:$, S[x], :ca]
```

The other orders handled this way are:

ad x which adds the contents of *x* to AC.

cs x which clears AC and subtracts the contents of *x*.

su x which subtracts the contents of *x* from AC.

mr x which transfers AC to BR and forms the two-register product of AC and *S(x)*. The most significant 16 bits go into AC and are rounded according to the lower 16 bits.

mh x which transfers AC to BR and forms the two-register product of AC and *S(x)*. The most significant 16 bits go into AC and the lower 16 bits into BR.

dv x which transfers AC to BR and forms the quotient of BR and *S(x)* in BR.

The two orders **sl n** and **sr n** perform a bit shift of the 32-bit value formed by taking AC as the most significant bits and BR the lower then moving it *n* bits left or right respectively. These are represented by:

```
[:$, n:value, :sl]
[:$, n:value, :sr]
```

At this point it makes sense to look at the main processing *routine* which uses an evaluation stack `eval`.

```
eval <- :pc      /get address of next order
eval <- :@
eval()

eval <- :$      /retrieve order copy
eval()

eval <- :*      /unpack
eval()

eval()         /pre-compute

eval <- []      /compute
eval()
eval()

eval <- :@
eval()

eval <- :*      /post-compute
eval()

eval()
```

Running through this block by block. The location of the next order is found. The order copied over to the evaluation stack. As it's an array it is *deconstructed*. The pre-compute phase is achieved by performing an *op* on the top two elements. The new top two elements are placed in an array. The function `@` applied. The resulting array is *deconstructed*. An *op* is then performed as the post-compute phase.

There need only be one *signal* that repeats this *routine* indefinitely. To halt the system an order can be encoded:

```
[:$, :ht, :-]
```

With an *action* on `ht` that will allow a *break* to be issued in the post-compute phase:

```
[:ht, :-] -> [!]
```

Returning in detail to `ca`. The pre-compute phase is `:$` on `S[x]` bringing in the value from `S(x)`. The compute phase will apply `@` to `[v:value, :ad]` yielding `[:$]`. In the post-compute phase this array is *deconstructed* and the following *op* simply removes the `:$`.

The shift orders work the same except that `:$` will have no effect on `n:value` in the pre-compute phase.

The order `ts S(x)` transfers the value in AC to `S(x)`. The encoding for this is:

```
[:*, S[x], :ts, S[x]]
```

In the pre-compute phase the old value at `S[x]` is removed. The *action* used will return a more complicated element than before:

```
[:ts, x:&] -> [x1:&, v:value]
```

The contents of AC are put into `v` and `x1` made equal to `x`. In the post-compute phase this acts like a closure and pushes `v` onto `x1`.

To achieve control flow `sp x` loads the program counter with `x` so that the next instruction is read from `S(x)`. For conditional control flow `cp x` does the same but only if `AC > 0`. To effect the change of PC one of two *actions* are used:

```
[:sp, x:&] -> [:$]
[:cp, x:&] -> [:$]
```

In these `sp` and `cp` are *tag* types. The first *action* puts `x` into PC but no increment is performed. The second does so only if `AC > 0` otherwise it increments PC as per usual.

The encoding of the orders is then:

```
[:$, :sp, x:&]
[:$, :cp, x:&]
```

This completes the basic operations of WWI, however, one is missing. The original design had an order `td S(x)` that would replace the top 11 bits at `S(x)` with those in AC. These 11 bits were the address part of the instructions (11 bits allows all 2048 storage registers to be accessed). As pointed out above this is

not as such possible in this representation since the data values are of a different type than the orders. There is an approximation though that would most likely give enough functionality to compute the intended programs of the system. By adding not one order but an order for each of the other orders an update of not only the storage register but the order type itself could be made. For instance to replace the order or value in storage register $S(x)$ by the order $ca\ S(AC)$ would require the use of $tdca\ S(x)$. Here $S(AC)$ denotes the storage register referred to by the top 11 bits of AC . This can be encoded by:

```
[:* , x:& , :tdca , x:&]
```

The *action* that is used in the compute phase is:

```
[:tdca , x:&] -> [x1:& , [:$ , y:& , :ca]]
```

Setting $x1$ equal to x and y equal to the top 11 bits of AC . Essentially a closure has been returned that will place onto $S[x]$ the encoded order $[:$, S(AC), :ca]$. A similar scheme can be followed for the other orders. A further extension is to include these new orders in the scheme with $tdtdca$ for instance but it has to stop somewhere.

An alternate representation could have been given that would take an arbitrary value and treat it as an order. By using an *action* that selects its return according to the order code section the order is carried out. However, the return type of this *action* would need to be sufficiently general to capture all the possible operations. It would require a stage to transfer a value from a storage register, a stage to clear a storage register etc. Any one of these stages might do nothing of course. This is more along the lines of a general microprogramming solution but loses all of the elegance of the representation just presented. It seems unnecessary to go this far to capture enough functionality to do useful calculations. In fact $tdca$ is able to change the type of order which was not accounted for in the original system.

This lack of completeness is not a symptom of weakness in the U^* -model but rather the product of applying strong typing rules where there were previously none. It will always be the case that forcing an un-typed system to be typed will present problems.

4.3 Assembler and Microcode

This example displays a general way of modelling a very primitive assembly language. There are many variations that could be made and it could be extended quite considerably. At its heart is the idea of building each instruction from those available in the U^* -model. In this sense the operations that the U^* -model supports become the microinstructions. This was seen in the Whirlwind example to some extent but due to the simplistic nature of the orders a specific encoding could be used. In those encodings the three phases for each order were preset. In a general assembler though there will be a wider variation in the phases required. It might be necessary to do multiple fetches of operands or even some amount of preprocessing. These phases are controlled by a sequence of microinstructions just as a real CPU might do. From the assembler instruction the relevant sequence of microinstructions is formed by applying @.

The target machine (either real or virtual) will only have a few primitive types normally along the lines of variously sized integers and floating point numbers. For clarity this example just deals with a single **word** type that has *size* equal to that of the *pointer* type. Memory is then the set of stacks and words are able to hold addresses.

A common operation required by some instructions is the need to convert between the **word** type and the *pointer* type. The following *actions* achieve this.

```
p:& -> w:word
w:word -> p:&
```

As with the other examples I leave the details of the *action* definitions until a better notation is introduced. Suffice to say these ones do a straight assignment noting that both types have the same *size*.

Each assembler instruction will reside as the top element of its own stack. Its representation is that of an array whose first elements is a *tag* type denoting the instruction followed by all its operands. These operands might be references to memory and hence a *pointer* or could be immediate values of type **word**. For instance:

```
[:add , x , y]
```

```
[:mov, x, 21:word]
[:jmp, 1]
```

Each of *x*, *y* and *l* are names of stacks designated as significant for some reason.

Control passes ordinarily from one instruction to the next except where a branch is encountered. The current point of execution is held as a *pointer* in a specially designated stack *pc* which is short for program counter. In order to move to the next instruction there is an advance *action*.

```
[:adv, p:&] -> q:&
```

Where *adv* is a *tag* type and *q* equals the state following *p*.

To execute an assembler instruction a copy of it is first fetched onto a *pipeline* stack by the *routine* *fetch*. This in turn applies *@* to get the array of microinstructions which will subsequently be *deconstructed*.

```
pipeline <- [!:] /push tail break

pipeline <- pc /copy address in pc
pipeline <- :$
pipeline()

pipeline <- :$ /copy instruction
pipeline()

pipeline <- :@ /create microinstructions
pipeline()

pipeline <- :* /unpack
pipeline()
```

The microinstructions themselves are also represented as an *array* of elements. To process a microinstruction the runtime *deconstructs* the *array* and *ops* the resulting elements. This is a very flexible scheme allowing new elements to be introduced to an evaluation stack when required. It also allows existing elements to be combined by means of the an empty *array*. Notice that the *fetch* *routine* first pushes a *break* onto the *pipeline* stack. This allows the runtime to detect the end of the microinstructions and move to the next assembler instruction.

Each microinstruction is processed by the *step* *routine* on an evaluation stack *eval*.

```
pipeline <- eval /microinstruction to eval
pipeline()
```

```
eval <- :* /unpack
eval()
```

```
eval() /compute
```

Elements are moved from *pipeline* rather than taking a copy as in *fetch*. The *adv* *routine* increases the program counter.

```
pc <- [:adv]
pc()
```

```
pc <- :@
pc()
```

Finally putting all this together are a handful of *signals*.

```
<cycle> ::= step <cycle>
<process> ::= fetch <cycle> adv <process>
<start> ::= load <process>
```

After loading the code and setting *pc* to point to the first instruction a loop commences. This fetches an assembler instruction and decodes it into the corresponding sequence of microinstructions. By means of *<cycle>* each one is processed until *[!:]* is found. This *breaks <cycle>* at which point the program counter is increased.

A sequence of microinstruction may need to invoke more *actions* to achieve its overall goal. So for instance at the heart of an addition instruction there needs to be an *action* to do the arithmetic. It would be possible to reuse the *tag* type for both the assembler instruction and the microinstruction but to aid readability the microinstruction version will be prefixed by *m*. Using addition as a starting point two *actions* are constructed:

```
[:madd, x:word, y:word] -> r:word
```

Naturally `r` will be the sum of `x` and `y`. A similar construction can be used for any unary or binary operator.

```
[ :add, x:&, y:&] -> [[:$,x1:&], [[:$,y1:&],
  [[:madd]], [], [:@], [:* ,x2:&], [x3:&]]
```

Here `x`, `x1`, `x2` and `x3` are to be equal. Similarly for `y` and `y1`. The sequence of microinstructions roughly translates as: fetch `x`, fetch `y`, load the instruction and bind to the first operand, bind to the second operand, compute, clear the target stack and finally store.

To perform conditional control flow a comparison operator can be introduced. The result of this comparison will be held in flags. These flags will be part of the register machine and have a two state type. Supposing only a strict less than operator is required there can be a flag:

```
less:boolean
```

Where `boolean` is a type of *size 2*. It will be set or cleared after performing the *action*:

```
[ :mcmp, x:word, y:word] -> :-
```

Ideally all instructions can affect flags so each *action* corresponding to an assembler instruction is able to alter them.

Now it is possible to perform a conditional jump:

```
[ :mj1, p:&, q:&] -> pp:&
```

This is slightly more involved. If `less` is set then `pp` should equal the state previous to `p` otherwise to `q` (which will be the current value on `pc`). The reason being that `adv` will be processed regardless of the outcome at this stage. The decoding to a corresponding microinstruction sequence will be a similar scheme as for `add` and is shown later.

Where an immediate operand, `w`, is needed the microinstruction to load it will be:

```
[ :$, w:word]
```

Loading a stack address is a little more complicated and can be done more than one way:

```
[:* , [s]]
[:-, :-, s]
```

The model thus presented makes no use so far of registers in the traditional assembler sense. There is in fact no need to distinguish between registers and memory since either will be held in stacks. They can of course be introduced as specially named stacks but to keep this example simple none are now given. An alternative model may keep a fixed set of registers in actual registers of the register machine but that is a bigger change.

By influence of the Z80 architecture [6] a few instructions can be mapped relatively simply to an *array* of microinstructions. Consider `ld a,b` that takes the value in `b` and puts it into `a` where either can be memory locations or registers. This will be formed for stacks `a` and `b` by:

```
[[:$,b], [a]]
```

If `b` is an immediate value it becomes:

```
[[:$,b:word], [a]]
```

Indirect memory access would be expressed by `ld a,(b)` where the indirection occurs through `b`.

```
[[:$,b], [:@], [:$], [a]]
```

The use of the *action* taking a `word` element to a corresponding *pointer* is critical.

Storage the other way `ld (a),b` works by:

```
[[:$,b], [:$,a], [:@], []]
```

Again the reverse *action* is important here and [] demonstrates using existing elements on `eval`.

Instructions such as `add a,b` and `cmp a,b` have been explained.

Finally for conditional behaviour `jmp l` and `j1 l:`

```
[[:*,pc], [pc,l]]
[[:$,pc], [[:mj1,l]], [:@], [:* ,pc], [pc]]
```

Here the label `l` is the address of a stack containing the target instruction and `pc` will be passed along with instruction. The unconditional jump removes

the old program counter value and replaces it. The conditional jump uses `mjl` to select between 1 and the contents of `pc`

Many of the ideas in this example have consequences beyond assembler. For any high-level language a set of microinstructions can be formed and the above runtime still works. Since stacks can hold any type there is no reason not to take primitive elements beyond simple numeric types. The introduction of microinstructions does seem to introduce a significant amount of fetching and storing. Much of this will be rendered unnecessary as consecutive instructions fetch previously stored results. By introducing higher-level instructions this can in part be avoided. Another option is to directly translate a source language to microinstructions. These can be further optimised to avoid any duplication. The problem is in no way specific to this model though.

In conceptual terms of machines a clear distinction can be seen for this representation. The *routines* and *signals* compose something like the core of a CPU in traditional hardware. It goes through various stages of fetching and processing instructions. The *actions* corresponding to microinstructions such as for `madd` are akin to operations in the CPU or ALU. There is a definite machine concept incorporating the underlying word type and instruction set. An actual program would be loaded by some *routine*.

4.4 The ACF Runtime

ACF stands for *atomic coroutine fragments* and is a scheme of runtimes rather than a particular formulation. Its principal aim is to decompose a computation into blocks such that each block is always processed in its entirety. Each block will take the now familiar form of an array of instructions. The type of instructions may resemble microinstructions or could be encoded instructions or something completely different. The choice only affects the part of the runtime that processes these individual instructions and is left open.

Rather than expressing the computation as a series of assembler like instructions laid out in memory each block is linked to other blocks forming a control flow graph. Coroutines maybe broken down into

such blocks with their multiple entry and exit points becoming block ends. For instance a loop may have a prefix block to set up some variants then move to a test block. This test block goes to either the body of the loop or the next block depending on the result of the test. The body of the loop will be a block (or a subgraph of blocks) that links back to the test block.

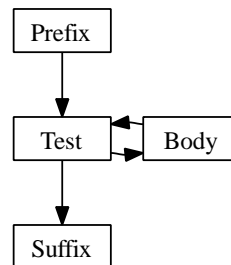


Figure 1: Control flow for a loop

This block processing can be achieved by processing instructions on an evaluation stack in sequence until an instruction issues a *break*. One more instruction is then processed than will dictate the control flow namely which block to fetch next. It does this by leaving a *pointer* to a block on a stack called *pipeline*. Initially *pipeline* will contain a *pointer* to the first block to process. The next block is fetched and processed in the same manner. This process repeats until the extra instruction controlling flow issues a *break* and the whole model terminates.

The *fetch routine* is somewhat familiar:

```
pipeline <- :$ /copy
pipeline()
```

```
pipeline <- :* /unpack
pipeline()
```

The *step routine* can take many forms as noted but one of the simplest would be:

```
pipeline <- eval /move
pipeline()
```

```
eval <- :* /unpack
```

```
eval()
```

```
eval()          /compute
```

It is possible to make the final instruction split into two parts. The first part issues the *break* during **step** and the remaining part will be left on the evaluation stack. To process this final part the **branch routine** only has to perform an *op*:

```
eval()
```

Tying these up are the *signals*:

```
<step>      ::= step <step>
<cycle>    ::= fetch <step> branch ? <cycle>
<execute> ::= load <cycle>
```

As interrupts have not appeared in the previous examples they are included here. They could have been located in **<step>** but it may well prove useful to guarantee no external influences during the processing of a block. This is the true nature of ACF that these blocks are atomic and once started will be finished.

Conditional instructions follow a few distinct patterns. To terminate the model instance `[:!, :!]` will suffice. To branch unconditionally `[:!, pipeline, 1]` will load the pipeline with a *pointer* to some stack labelled 1. The result of some potentially complex computation may remain on the evaluation as a *pointer* to the next block. In this case `[:!, pipeline]` will assign it to **pipeline**. Consider an *action*:

```
[:select, l1:&, l2:&, ..., n:nat] -> l:&
```

As usual **select** is a *tag* type and corresponds to a function able to select the *n*'th labelled stack from a fixed list. Further more complicated *actions* could be devised for a multitude of different concepts.

The power of the scheme comes through being able to add in higher level instructions than in the previous examples. Direct facets of a programming language can be modelled by adding *actions*. For instance an easy way to implement a switch statement in C would use the label selection above. Also by

grouping a program into blocks of instructions without any constraint on those instructions means a compiler can perform its normal optimisations and transformations. The problem of refetching previously stored data can be solved by optimally folding and rearranging instructions within a block. Given the atomic nature of these blocks a compiler has a good degree of freedom in the transformations it can apply. In a concurrent system this would be important since a context switch could not occur within a block. However, each block is finite so will always yield to the interrupt mechanism ensuring no dead locks. These concepts are explored in the next example.

4.5 Concurrency

Despite the system only allowing a single path of execution in terms of state changes it is possible to simulate concurrent processes. This concept has been successfully implemented at Ericsson with their language Erlang [7]. Using the ACF runtime as a base the aim is to interleave the execution of each fragment. This works since each fragment necessarily takes only a finite number of steps to complete so blocking cannot possibly occur. In this sense it is wait-free. Every process will require its own copy of an **eval** stack and a *pointer* to the next fragment. These must be preserved when a process switch occurs but can be combined onto a single stack for storage. A circular buffer of process descriptors is maintained using three stacks **processed**, **pending** and **current**. Each descriptor is an instruction to swap in the relevant stack or is a terminator descriptor that contains a *break*. When **pending** is empty it is swapped with **processed**.

The *advance routine* takes the next pending process descriptor and places it on to **processed**. It also leaves two copies on **current**. The first is used to swap the state in and the second to swap it out again.

```
pending <- current
pending()
processed <- current
```

```
processed <- :$
processed()
```

```
current <- :%      /make second copy
current()
```

The *swap routine* unpacks the descriptor and executes the swap (or *break*).

```
current <- :*      /unpack
current()
```

```
current()          /perform swap
```

In order to reset the circular buffer a *routine* can perform a *reverse-swap* on the *processed* and *pending* stacks. This takes advantage of the reversing nature of the operation to retain the original order. The *reset routine* is then:

```
current <- :-      /remove copy
current()
```

```
current <- processed
current <- pending
current <- :S
current()          /swap
```

Finally there are two *routines* to extract the fragment address from the per-process stack and its inverse to put it back.

The *routine unpack*:

```
eval <- pipeline
eval()
```

The *routine pack*:

```
pipeline <- eval
pipeline()
```

Adopting a similar notation to the standard ACF runtime leads to the following *signals*.

```
<work>      ::= fetch <step> branch
<stay>      ::= unpack <work> pack
<cycle>     ::= <stay> swap advance
              swap <cycle>
<round>     ::= advance swap <cycle>
              reset <round>
<execute>   ::= load <round>
```

The load signal must ensure that *pending* is configured with a list of process descriptors and a final terminator. Each descriptor indicates a process stack which must contain a *pointer* to its first fragment on top. For example:

```
evalA <- intial_fragment

pending <- [!:]
pending <- [:S, eval, evalA]
```

There are some interesting consequences of this model. Any process may make a fixed number of alterations to the process list by operating on *pending* and *processed*. For example adding some new processes to either side of itself, removing processes or clearing all processes. A process may even remove itself.

Interrupts can be added at various places within the scheme to yield the system after a step, fragment or even full round of processes.

It is possible to allow more than one fragment to execute during each process stay. Adding a *reduce* stack that contains a sequence of *:\$* elements followed by *:!* and consulting this stack after each fragment would issue a *break* only after a certain number of fragments. Any process can prematurely yield by placing an artificial *:!* on top or extend its stay by adding *:\$*. This fine grained control allows a process to execute a number of fragments atomically. The scheme to reset this reduction stack can use a *reverse-swap* with a *null* element to clear any remaining contents.

Handling inter-process communication becomes simpler than most models since each fragment can be arbitrarily long but will act atomically.

Asynchronous messaging between processes can be achieved if a new per-process stack is added to model a message queue. The swapping mechanism must then move this along with *eval*. This relies on the swap reversing the order. Messages arrive when the queue is attached to the process but are received when it is brought into context. These two events will occur when the queue is facing in opposite directions enabling FIFO messaging.

5 Action Definitions

Until now very little has been said about defining *actions* although the details of the four instructions of the register machine have been explained. That is itself enough to develop the *actions* from the examples of the last section, however, each definition would have been long and demonstrated little. Even with the tools of this section the definitions would have been very repetitive. The notation and constructions used here mimic general programming techniques building everything from smaller units of common functionality. The method used is that of defining a series of macros which themselves can be composed of other macros. Then an *action* can be formed by expanding all these macros recursively yielding a rather unreadable but correct sequence of the four available instructions.

An *action* declaration gives rise to a set of registers from the element to which @ is being applied and another set from the resulting element. There is no reason to distinguish between them within an *action* and they are all referred to as arguments. In technical terms an *action* uses the call by name convention. The registers of the resulting element will all be in their special state 0. Each register has a *size* that reflects the number of states the register can be in or alternatively how many times it must be advanced before it returns to its current state. Another concept of the register machine is a label that can refer to a specific instruction. These are written using a name prefixed by @. Thinking of each line of the expanded definition as either one of the four core instructions or a label is useful.

Often some form of temporary storage is required during the processing of an *action*. As described earlier a persistent register can be declared for this purpose. For convenience its beneficial to associate this persistent register with the *action*. Since it is directly associated it is said to be local to the *action*. This can be done by writing after an *action* declaration a list of registers enclosed in square brackets:

```
[t:value, done:boolean]
```

It's important to be clear that this is not a new concept. These local registers are still persistent reg-

isters and semantically accessible from all *actions*, however, they are syntactically inaccessible. This prevents problems with name clashing. They are not really temporary since persistent registers hold their state between applications.

Where a common sequence of instructions is to be reused they can be declared as the body of a macro enclosed in curly braces. Each macro has a name and zero or more arguments each of which in effect uses the call by name convention. These arguments will take one of three types: register, label or parameter. A register type is either an *action* argument, a persistent register (local registers naturally included) or a macro argument of register type. In short anything that is a register. It is written as just a name. A label type is either a label declared in the current instruction sequence (possibly after the point of use) or a macro argument of label type. It is written as a name prefixed by @. A parameter is a natural number that can be used as argument to a macro and subsequently passed to other macros. It is written as a name prefixed by # and instances are the numbers themselves.

When a macro is to be used its name is written followed by the instances to bind to the arguments separated by commas. The types must match otherwise the macro expansion is malformed. A macro may also have local storage and uses the same notation as for an *action*. It is extended slightly to allow the register type to be the same as one of its register arguments. This is indicated by writing ~x where x is an argument of register type. When a macro is expanded a fresh version of each local register is declared. Any labels that appear in the body of the macro are created for each expansion and although share the same name are in different and inaccessible scopes.

The final piece of notation allows for a sequence of instructions, labels or macro expansions to be repeated. A parameter enclosed by parentheses is followed by a body enclosed in curly braces. For instance to advance a register r three times:

```
(3)
{
  r' -> r
```

```
}
```

Everything becomes more clear with plenty of examples. Considering registers as natural numbers modulo their *size* motivates:

```
inc( r )
{
  r' -> r
}
```

To test if a register is in state 0 and go to one label if true or another if false.

```
tst( r, @zero, @nonzero )
{
  r=0 ? zero : nonzero
}
```

To unconditionally start processing at another label:

```
jmp( @label )
[ r:tag ]
{
  r=0 ? label : label
}
```

The register *r* will permanently be in state 0 of which there will be one copy for each expansion of the macro.

Simple conditional tests for 0:

```
ifz( r, @label )
{
  r=0 ? label : next
@next
}
```

```
ifnz( r, @label )
{
  r=0 ? next : label
@next
}
```

This is legal since expansion is syntactic. The label will either associate with the next register machine instruction or be at the end of the definition. Where

multiple labels associate with the same instruction they are interpreted to be identical.

To clear a register namely putting it in state 0:

```
clr( x )
{
@loop
  inc x
  ifnz x, loop
}
```

In contrast to traditional register machines that require a decrement style instruction this can be built. It essentially adds $|x| - 1 \geq 0$ to *x*:

```
dec( x )
[ t:~x ]
{
  clr t
  inc t

@loop
  inc x
  inc t
  ifnz t, loop
}
```

To transfer state between registers is more complicated as an intermediate register is required. When expressing this as a macro the intermediate register is local and takes the same *size* as one of the arguments. It's imperative not to make it the same *size* as the wrong one though since where the argument *sizes* differ modulo assignment is expected. Comments are added for clarity.

```
mov( x, y )
[ temp:~y ]
{
@loop1
  inc temp          /starts+ends 0
  dec y
  ifnz y, loop1    /temp=y, y=0

  clr x            /x=0
@loop2
  ifz t, done
```

```

inc x
inc y          /restore y
dec temp
jmp loop2
@done
}

```

The following examples build part of normal arithmetic and are largely self-descriptive:

```

add( x, y )
[ t:~y ]
{
  mov t, y
@loop
  ifz t, done
  inc x
  dec t
  jmp loop
@done
}

```

```

sub( x, y )
[ t:~y ]
{
  mov t, y
@loop
  ifz t, done
  dec x
  dec t
  jmp loop
@done
}

```

One final example makes use of parameters. It is so far possible to set a register to state 0 but not say its n 'th one. This amounts to setting a register to state 0 and then having n copies of the advance instruction. The following macro achieves this neatly:

```

set( x, #n )
{
  clr x
  (n)
  {
    inc x
  }
}

```

Hopefully it is now possible to believe that the *actions* used in the examples can be realised. Multiplication, division and finding a remainder can be built from addition and subtraction. Using the conditionals *ifz* and *ifnz* allows for selection.

References

- [1] "Towards A Universal Model Of Computing", Peter Seymour, 2008.
- [2] "The Breathe System", Peter Seymour, 2008.
- [3] "A Compendium Of Devices", P. Seymour, 2008.
- [4] "On Computable Numbers, with an application to the Entscheidungsproblem", A.M. Turing, 1936.
- [5] "Whirlwind I Computer Block Diagrams", R.R. Everett and F.E Swain, MIT Servomechanisms Laboratory, 1947.
- [6] "Z80 Pocketbook", J.B. Vonk, Glentop, 1986.
- [7] "Concurrent Programming in ERLANG", Joe Armstrong, Robert Virding, Claes Wilkström and Mike Williams, Ericsson, 1996,