

Register Machines
Storing Ordinals and
Running for Ordinal Time

Ryan Siders

Outline

- Assumptions about Motion and Limits on Programming
- Without well-structured programming:
- Well-Structured Programs:
- Ordinal Computers with Well-Structured Programs:
- Well-Structured programs halt.
- Scratch Pad, Clocks, and Constants
- Abstract Finite-time Computability Theory
- Jacopini-Bohm Theorem holds if...
- How many registers are universal?
- 3 registers, No. 4 registers, Yes.
- Main Theorem: Decide Any Constructible Set
- Recursive Truth Predicate
- Pop
- Push
- The Program to find Truth
- Zero?

Assumptions about Motion and Limits on Programming

Input, Memory, and Time for Hypercomputation:

- Time is wellfounded so that the run is deterministic.
- We want input and memory to be ordinals, too.
- Then we can compare input length to run-time and define complexity classes.

Definition An Ordinal Computer stores ordinals in its $0 < n < \omega$ registers, runs for ordinal time, and executes a program that can alter the registers.

At Limit times, we want the following to be determined:

- which command line is active, and
- what the values of the ordinals are.

They are determined by:

- assumptions on the behaviour of the registers and flow of control in the program (limiting the universe in which the programming model can exist), and
- limitations on the class of programs that can be used.

Without well-structured programming:

Definition: *An Infinite-time Ordinal-storing Register Machine is a register machine storing ordinal values and running for ordinal time, with a programming language including the three instructions:*

- Erase: $\text{Zero}(x)$;
- Increment: $x ++$;
- Switch: $\text{if } x = y \text{ goto } i \text{ else } j$;

whose state (the registers' values and active command) obeys the following three rules at a limit time:

- If the command " $\text{Zero}(x)$ " is called at each time $t \in T$, then x is 0 at time $\sup T$, too.
- At limit times, command passes to the lim-inf of the commands which have been active cofinally often.
- Until it is zero'd, a register's value increases continuously.

Well-Structured Programs:

A program is “well-structured” if goto switches are only used to model the following two commands:

- `if($x = 0$)` (instructions);
- `for($x = y; x < z; x ++$)` (loop); where neither `Zero(x)` nor `Zero(z)` is among the instructions in the loop.

NB: Our for loop tests whether $x < z$, and then executes.

Ordinal Computers re-defined: Wellfounded programs can mimic any program, so we can rewrite some of our assumptions about how machines behave at limits as assumptions about how programs are written.

Ordinal Computers with Well-Structured Programs:

Definition: *An Ordinal Computer is a register machine storing ordinal values and running, for ordinal time a well-structured program, in which the registers' values at limit times obeys the following three rules:*

- The variables used in switches depend in a wellfounded way on monotone variables (variables which are never set to Zero; see “Flickering lamps,” below).
- If the program says to loop states until $x \geq z$, then the machine does not stop looping states simply because it reaches a limit time, but only when $x \geq z$.
- Until it is zero'd, a register's value increases continuously.

Flickering lamps: *We can work in a universe where:*

- “if a lamp is turned off infinitely often (and perhaps turning it on again in between), at times limiting to λ , then at the limit time λ , it is off,” *or write programs where:*
- “after turning a lamp off infinitely often, at times limiting to λ , then at the limit time λ , a wellstructured program turns it off (or on) again, before asking whether it is off.”

Well-Structured programs halt.

Lemma All well-structured ordinal computer programs halt.

Example: `for($x = y; x < z; x ++$) ($z = f(x, y, z);$
 $y = g(x, y, z)$)` halts at (if not before) the least λ which is closed under f and g .

Example:

`for($b = a + 1; a < b; a ++$) ($b ++$)` halts at the least limit ordinal $> b$.

But this program never halts:

Example:

- $b = a;$
- 1. `if ($b = a$) ($b ++$);`
- $b ++; a ++;$
- `if ($b \neq a$) (goto 1)`
- `else (halt);`

since at limit times line-control passes to line 1.

Scratch Pad, Clocks, and Constants

Lemma In a well-structured computation, the following hold: 1. If all registers are set to 0 repeatedly (after any time t , each register is again set to 0 at some time $t' > t$), then there is a time at which all registers are simultaneously zero. 2. Any active loop index is equal to the clock at all times ϵ_α .

As a result, we find that there are fundamentally different natural roles for the registers in a computation: as a clock, as scratch pad, and as a constant.

Register Types *In a well-structured computation we can identify three types of registers:*

- 1. registers which are zero'd infinitely often,
- 2. registers which are never zero'd, and so are equal to the clock at `exp`-closed times,
- 3. registers which are neither incremented nor zero'd.

Abstract Finite-time Computability Theory

Definition For M any model, an M -register machine stores (has some variables x_i referring to) elements a_i of M , and runs for finite time. Its programming language assigns variables to elements of M with the following two commands:

- 1. assign x_i to a_j (duplicate);
- 2. assign x_i to a_j ; assign x_j to a_i ; (swap);

The constants, functions, and relations in M are assumed to be computable, so the programming language includes the following three commands

- 3. $a_i = f^M(a_1 \dots a_n)$;
- 4. $a_i = c^M$;
- 5. If $R^M(a_1 \dots a_n)$, then i , else j .

The storage of elements of M in the register machine is like the assignment of variables to elements of M in a finite-variable pebble game. The M -register machine can set a register to any constant in M , set a register a_1 to $f(a_1 \dots a_n)$, for any n -ary function f , and switch its state, depending on whether $M \models R(a_1 \dots a_n)$ or not.

Jacopini-Bohm Theorem holds if...

Analyzing the flow of control in a computer program, without looking at what is in the registers, was pursued by Jacopini-Bohm, proving

Theorem: *Any Goto program is equivalent to a While program. The Jacopini-Bohm holds so long as the model in question “has counting stacking and pairing.”*

See: J. Tucker and J. Zucker, *Computable functions and semicomputable sets on many sorted algebras*, in S. Abramsky, D. Gabbay and T Maibaum (eds.) *Handbook of Logic for Computer Science*, Volume V, Oxford University Press, 317-523, pp. 485,486.

Definition *Let Γ be the pairing function taking (a, b) to the wellorder of pairs $(c, d) <^2 (a, b)$, where $(c, d) <^2 (a, b)$ iff $\max(c, d) < \max(a, b)$, or $\max(c, d) = \max(a, b)$ and $c < a$, or $\max(c, d) = \max(a, b)$ and $c = a$ and $d < b$.*

Definition *A “binary stack” codes a finite, monotonically decreasing sequence $(\beta_i : i < n)$ as $\sum_i 2^{\beta_i+1} = 2^{\beta_0+1} + 2^{\beta_1+1} + \dots + 2^{\beta_{n-1}+1}$, where 2^α is ordinal exponentiation.*

How many registers are universal?

Computable functions using two registers computes a fragment of arithmetic:

- Suppose our rule about “for” loops relax, and only the index were not allowed to be zero'd. Then $y++$; for($x = 0; x < y; x++$) (Zero(y); for($y = y; y < x; y++$)($x++$); $y++$) halts on input 0, 0 with register values ω^ω .
- Let $\rho_3(\alpha) = \alpha \bmod \omega^\omega$.
- Let ρ_4 be the identity below ω^ω , and be $\omega^\omega + \rho_3$ above ω^ω .
- Let $\rho_5(\alpha, \beta)$ be the pair $(\rho_4(\alpha), \rho_4(\alpha) + \rho_4(\beta - \alpha))$ if $\alpha \leq \beta$ and be undefined if $\alpha > \beta$.
- 2 registers compute $x \mapsto x \times \omega$ and $x \mapsto n \times x$ (for finite n), but are weaker than the theory of $(Ord, <, c_0, c_1)$, where c_0 and c_1 are constants naming any two \times -closed ordinals; this is much weaker than the theory of $(Ord, +)$.
- An Ordinal computer with two variables can not compute the $+$ of two input values.

3 registers, No. 4 registers, Yes.

An Ordinal computer with three variables can simulate a finite Turing machine.

An Ordinal computer with three variables still reflects below $\epsilon_{\omega \times 4}$.

Corollary: *Stacking ordinals requires more registers than stacking finite numbers. Moving an infinite ordinal onto a stack by incrementing the stack once every few time-steps requires infinite time. So the stacking operation with which OC^n simulates a finite-time $(\omega_1, +, \times, a \rightarrow \omega^a)$ -register machine (an abstract register machine must limit continuously without losing any information.*

We wrote a universal program using 12 registers. A reduction to four registers is possible:

- three registers are oscillating stacks as for a finite Turing Machines.
- The fourth variable mimics a stack element. When that element is erased by limiting behavior, we copy it from the fourth element.

Main Theorem: Decide Any Constructible Set

Proposition *There is a universal ordinal computer program so that for every set of ordinals S which exists in L , there is a formula $\phi(x, \alpha_0, \dots, \alpha_n)$ defining S , so that if n is the single number Gödel-coding the formula ϕ and $\alpha = \Gamma(\dots\Gamma(\alpha_0, \alpha_1)\dots, \alpha_n)$ is the polynomial-size "stacking" of the parameters $(\alpha_0 \dots \alpha_{n-1})$, the program P halts on input $\Gamma(\alpha, \beta)$ and decides whether $\beta \in S$.*

A finite-time Ordinal-storing machine (a la Tucker-Zucker) can unpack $\phi(x, \alpha_0, \dots, \alpha_n)$ from α and substitute β . We seek a program P so that if $\phi(\alpha_0, \dots, \alpha_n)$ is presented to P as the ordinal α , P halts iff α codes a true formula.

Conversely, the definition of the program P exists within L , so OC computation reflects into L . That is, anything OC-computed from finite ordinal parameters $\alpha_0 \dots \alpha_n \in L$ is thereby constructed in L .

Theorem *A set S of ordinals is ordinal computable from some finite set of ordinal parameters if and only if it is an element of the constructible universe L .*

Recursive Truth Predicate

Definition *Let T be the recursive truth predicate, defined by: $T(\alpha) = \text{True}$ if and only if $(\alpha, <, \Gamma, T \text{ below } \alpha) \models \phi_\alpha$, where Γ is the ordinal pairing function, where the sentence ϕ_α is coded by α , has a finite number of ordinal parameters.*

Deciding whether $\alpha \models \phi$ when $\phi = \exists x\psi(x)$ requires us to push each proposed x onto a stack, and then check $\psi(x)$.

Pop

Pop, taking two parameters (Stack, Threshold) and referencing the global variable \$₋ in which the program has received as its input a formula whose truth is to be witnessed, (and which serves as an upper bound to all formulas and all searches) is the following routine:

```
MON SmallStack := 0;
MON TempStack := 0;
for  $\epsilon$  from 0 to $- (
  for  $\alpha$  from 0 to Stack (
    if ( $\alpha + 2^\epsilon + \text{SmallStack} = \text{Stack}$ ) (
      if ( $\epsilon > \text{Threshold}$ ) (return  $\epsilon$ );
      for TempStack to Smallstack ();
      for Smallstack to  $2^\epsilon$  ();
      for  $\kappa$  from 0 to TempStack (Smallstack++)
    )
  )
)
```

Pop doesn't really change the stack. It just reads the next element, past a certain threshold.

Pop reads least element 2^ϵ of the Stack, such that ϵ is at least as large as the parameter Threshold.

Push

Push, a program taking two parameters (Stack, β), is the following routine:

```
Stack ++;  
for  $\iota$  from 0 to  $2^{\beta+1}$  (  
  if ( $\neg$  ( $2^{\beta+1}$  divides Stack)) (Stack ++)  
)
```

where β divides α is the routine:

```
MON  $\gamma = 0$   
for  $\iota$  from 0 to  $\alpha$  (  
  for  $\kappa$  from 0 to  $\beta$  ( $\gamma ++$ )  
  if ( $\gamma = \alpha$ ) (Return Yes);  
  if ( $\gamma > \alpha$ ) (Return No)  
);
```

Return No

Push(β onto Stack) erases all stack values less than β .

Push(β onto Stack) increases the Stack to the next full multiple of $2^{\beta+1}$.

IsEmpty, taking the single input Stack, is the routine:

```
ORD  $\alpha = \text{Pop}(\text{Stack}, 0)$ ;  
if ( $2^\alpha = \text{Stack}$ ) (return "True")  
  
else (return "False")
```

IsEmpty(Stack) returns the value "True" when the stack is a singleton, $2^{\$}$.

The Program to find Truth

```
Determining the Truth Value of ( $\$_$ ):
ORD  $\alpha = 0$ ;
ORD  $\beta = 0$ ;
ORD  $\nu = 0$ ;
MON Stack = 0;
ORD TruthValue = Unknown;
Push( $\$_$  onto Stack);
for  $\iota$  from 0 to  $2^{\mathcal{S}}$ - (
   $\beta = \text{Pop}(\text{Stack}, 0)$ ;
  If ( $\beta$  is a limit) (TruthValue = Unknown);
  If IsEmpty(Stack) (Stack ++); # That is, "if Stack =  $2^\beta$ ."
  if TruthValue is Unknown (
    if  $\beta$  is a successor ordinal (Stack ++;  $\beta = 0$ );
     $\alpha = \text{Pop}(\text{Stack}, \beta + 1)$ ;
    if  $\beta$  is not a successor ordinal and  $\alpha = \beta + 1$  (
       $\beta = \alpha$ ; TruthValue = False;
    );
    if  $\beta$  is not a successor ordinal and  $\alpha \neq \beta + 1$  (
      Push( $\beta$  onto Stack);
    )
  );
);
while TruthValue is Known (
  if  $\beta$  is the largest element on the stack (return TruthValue);
   $\alpha = \text{Pop}(\text{Stack}, \beta + 1)$ ;
  Let  $\nu = H(\alpha - 1, \beta - 1, \text{TruthValue})$ ;
  if ( $\nu = \text{True}$ )( $\beta = \alpha$ ; TruthValue =  $\nu$ );
  if ( $\nu = \text{False}$  and  $\alpha = \beta + 1$ )( $\beta = \alpha$ ; TruthValue =  $\nu$ );
  if ( $\nu = \text{False}$  and  $\alpha \neq \beta + 1$ )(
    TruthValue = Unknown;
    Push( $\beta$  onto Stack)
  )
);
)
```

Zero?

Can we get rid of the command Zero in the programming language?

- Track the ordinal $\alpha(t)$ as $\alpha(t) = \alpha_+(t) - \alpha_-(t)$, the difference between two monotonic registers;
- then when α appears in a switch, some ordinal must be increased, to run from α_- to α_+ . If we use dummies, and insist that the dummies are always j variables, then we increase the real variables frequently.
- if we use α_- , then mentioning α in a switch, or to set the value of another register, destroys it. We can destroy one register and make many copies of it, but we should foresee how often we will need it.
- Perhaps this is like the consumption of variables in declarative programming, or the logic of non-copiable quantum states.