Computing the Recursive Truth Predicate on Ordinal Register Machines

Peter Koepke[1] and Ryan Siders[2]

[1] University of Bonn, Mathematisches Institut,
Beringstraße 1, D 53115 Bonn, Germany
`Koepke@Math.Uni-Bonn.de`
[2] University of Helsinki, Department of Mathematics and Statistics
Gustaf Hällströminkatu 2b, Helsinki 00014, Finland
`bissell@mappi.helsinki.fi`
(corresponding author)

Abstract. We prove that any constructible set is computable from ordinal parameters by a wellfounded program on an infinite-time ordinal-storing register machine.
This brings us closer to "minimal" computation of set theoretic constructibility. To that end, we describe data types and well-founded programming to consider what can be cut from the machine or programming languge.
These machines were designed to define and study run-time complexity for hypercomputation. We solve one complexity problem: deciding the recursive truth predicate is ordinal-exponential time on a register machine, and ordinal-polynomial time on a Turing machine.

1 Register Machines

Ordinal Register Machines increment and erase a finite number of registers containing ordinals; the number of necessary registers can eventually be reduced to four. They exemplify abstract model-theoretic computation. The ordinals they store can refer to the timesteps of a hypercomputation.

We motivate these machines by first considering 1. finite-time register machines storing finite register values, and 2. finite-time register machines ordinal register machines storing ordinal register values with an oracle for ordinal arithmetic.

Definition 1. *([10]) A register machine stores natural numbers and runs for finite time. A register machine program is written using the three instructions: 1.* `Zero`(x)*, which sets x to zero; 2. $x{+}{+}$ which increments x; and 3.* `If` $(x = y)$ `goto` i `else` j*, which switches the flow of control, depending on whether two registers are equal or not. Here, i and j name instructions in the program. When the program is interpreted in a machine, i and j label states of the machine.*

Definition 2. *On a register machine, a "*`For`*-program" uses* `goto` *switches only to define the commands: 3a.* `if` $(x = y)$ `(instructions)`*; 3b.* `for` $(x = y; x < z; x{+}{+})$ `(instructions)`*, where x is never set to zero within the loop.*

A "`While`*-program" has the extra instruction 4.* `Decrement`(x)*; but only uses* `goto` *switches to define: 3a.* `if` $(x = 0)$ `(instructions)`*; 3b.* `while` $(x > 0; x--)$ `(instructions)`*, where x is never incremented within the loop.*

We call finite register machine `For`*- and* `While`*-programs "well-structured."*

Definition 3. `for` x `from` 0 `to` y *means* `Zero` (x)*;* `for` $(x = x; x < y; x++)$. `if`$(a = b)$ `(instructions)` *means* `Zero`(x); `for`$(c = a; c < b; c++)(x++)$; `for`$(c = b; c < a; c++)(x++)$; `if`$(x = 0)$`(instructions)`.

Theorem 1. *([9] p. 205) 5-variable* `While`*-programs simulate Turing machines.*

Theorem 2. *([9] pp. 255-8)* `While`*-programs using 2 variables can simulate all* `While`*-programs.* `FOR`*-programs using 3 variables can simulate all* `While`*-programs.*

In this paper, we have used the word "model" in the sense of "a model of computation," and in the sense of "model theory." We try to never say a machine A "models" the behaviour of machine B, but rather that it "simulates" B. Simulating one machine on another, and building a model of one theory inside a model of another are such similar concepts, that the idea of computing over abstract models seems ripe for self-reference.

Definition 4. *(see the survey [11]) For M any model, an M-register machine stores (has some variables x_i refering 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. $a_i = f^M(a_1...a_n)$;

4. $a_i = c^M$;

5. If $R^M(a_1...a_n)$, 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 $f(a_1...a_n)$, for any n-ary function f, and switch its state, depending on whether $M \models R(a_1...a_n)$ or not.

Analyzing the flow of control in a computer program, without looking at what is in the registers, was pursued in [3], in which it is proved that "any Goto program is equivalent to a While program." Such analyses are fruitful too in "abstract computability theory," surveyed by [11].

1.1 Pairing and Stacking

Research on abstract-model recursion theory since [2] suggests that the ordinary theorems of recursion theory will lift, if equality is decidable, pairing is computable, and the domain is finitely generated. Fortunately, ordinals have an efficient notion of pairing.

Definition 5. *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 $d < b$.*

Lemma 1. *$\Gamma(a,a) \cong a$ iff a is a $\times$-closed ordinal. (so Γ is "efficient.")*

Remark 1. ([11] pp. 485,486) show that recursion theory lifts to M-register machines if M has counting and stacking. Our main result depends heavily on using the following binary stack to prove that ordinals have stacking, and that the stack is robust, as the ordinal register machine runs over a limit-ordinal time.

Definition 6. *A "binary stack" codes a finite, monotonically decreasing sequence $(\beta_i : i < n)$ as $\sum_i 2^{\beta_i+1} = 2^{\beta_0+1} + 2^{\beta_1+1} + ... + 2^{\beta_{n-1}+1}$, where 2^α is ordinal exponentiation.*

The reason for $+1$ in the exponent is that when β_{i+1} limits up to β_i, and β_i is a limit ordinal and false (it codes ϕ_{β_i}, a falsehood, so for T the Recursive Truth Predicate of definition 13, $T(\beta) =$"false") then we need to be able to check whether β_i being false witnesses the truth of β_{i-1}, before incrementing β_i. The $+1$ in the exponent allows us to identify a limit ordinal appearing as a term in the $\sum$ which is the stack as a limit of earlier stack elements and not a stack element itself.

Definition 7. *Let "Seq" be the set of finite, descending sequences of ordinals, all less than $\$\_$, ordered by their first difference.*

Lemma 2. *$(\beta_i) \mapsto \sum 2^{\beta_i}$ is an isomorphism between Seq and $2^{\$\_}$.*

Lemma 3. *If the supremum of T is α, then the supremum of $\{2^\beta : \beta \in T\}$ is 2^α.*

We will define Pop, Push, and IsEmpty for this stack in section 3, after we have learned the natural data types and definitions for ordinal computers and are comfortable writing longer routines.

1.2 A model of infinitely-long computation

Definition 8. *An Infinite-time Ordinal-storing Register Machine is a register machine storing ordinal values and runing for ordinal time, with a programming language including the three instructions: 1. Zero(x); 2. $x++$; 3. if $x = y$ goto i else j; in which the registers' values at limit times obeys the following three rules:*

R1. If the command "Zero(x)" is called at each time $t \in T$, then x is 0 at time $\sup T$, too.

R2. At limit times, command passes to the liminf of the commands which have been active cofinally often.

R3. Until it is zero'd, a register's value increases continuously.

We'll also call this model of computation an Ordinal Computer, or *OC*.

Remark 2. The study of continuous computation or abstract register machine computation over an infinite model motivates the study of hypercomputation.

Our rule 2. is the same as that in [6]. Other definitions of limit time use the lim-sup rule ([4]) or require wellfounded programs ([1], see definition 9) so at a limit time, the machine only decides whether to keep looping (see remark 3 below). These approaches are equivalent.

Definition 9. *An OC program is "well-structured" if* `goto` *switches are only used to model the following two commands: 3a.* `if`$(x_i = 0)$ *(instructions); 3b.* `for`$(x = y; x < z; x++)$ *(instructions); where neither* `Zero`(x) *nor* `Zero`(z) *is among the instructions in the loop. A* `for` *loop tests $x < z$, then executes.*

Remark 3. At any limit time λ during the run of a well-structued program, there is a unique instruction `for`$(x = y; x < z; x{+}{+})$`(loop)` for which we have checked whether $x \geq z$ at times T a cub subset of λ, but $x < z$ was true each time. Rule 2. of definition 8 requies that at a limit time, control returns to the start of the loop; the start of a `for` loop is its test, so equivalently, Rule 2'. At a limit time λ, repeat the outermost loop which has been active cofinally often. That is, after checking $x < z$ infinitely often and finding it always true, check again.

Rule 2' makes a very reasonable, but nontrivial, requirement of the machine's state at a limit time: that 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 $x \geq z$.

Theorem 3. *For well-structured computations, Rule 1 can be simplified, so as to require nothing of the machine's limiting behaviour, but only require something syntactical, about how well-structured programs are formulated: Rule 1': In a well-founded program, immediately before a switch* `if`$(x = 0)$ *is called, x was changed one last time (and not an infinite, unbounded set of times, before the switch is called).*

Then any register, at any time, is defined in terms of other variables, each of which was defined one last time beforehand, such that there is a finite tree of variables and times, on whose definition x's value at the time of the switch depends, the leaves of which are variables which are never zero'd, during the computation. So *OC* programs can be written so that switches well-foundedly depend on monotonic variables.

1.3 Discovering data types

Lemma 4. *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 data types.

Remark 4. 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 cub-often equal to the clock, and 3. registers which are neither incremented nor zero'd.

Registers that are zero'd infinitely often are the indices to short loops and "active memory." Registers that are frequently close to the clock are really just marking time (and storing type 1 variables between each other). Registers that do not change during the computation are the parameters which the computation (seen as a subroutine) was given from the outside.

Definition 10. *We call type 1. variables ORD, type 2. variables MON, and type 3 variables STO, for "ordinal," "monotone," and "static storage." In long programs, all variables will have for scope only the subroutine they are defined in, and we will declare their type before using them, for clarity.*

Lemma 5. *All well-structured ordinal computer programs halt.*

Proof: by induction on the depth of `For` loops: the maximum value ρ of the registers is a normal function in time, which has arbitrarily large fixed points. At exp-closed ordinal times, the loop-index is time, as well. At these times, the loop index and bound are equal, and the loop terminates. □

But other programs need not halt:

Example 1. A. `For`-programs halt when they reach a fixed point. B. some non-well-structured programs do not halt at all

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

B. $b=a$`; 1. if (`$b=a$`) (`$b++$`); `$b++$`; `$a++$`; if (`$b\neq a$`) (goto 1);` never halts since at limit times line-control passes to its lim-inf (def'n 8, R2).

Because of the intuitive variable types found in lemma 4 and remark 4 and the simple program-flow described in lemma 5 and remark 3, we will restrict our attention to well-structured ordinal computer programs.

1.4 Reflection of few registers

How many registers are needed to simulate an infinitary Turing machine on an ordinal computer? How many registers are clearly trivial? Four registers are universal, and three registers are fairly trivial. In the next subsection we will prove the reflection of up to three registers. Note that the program computing the universal truth predicate uses 6 variables outside any subroutine, and the longest subroutine, `Pop`, uses 6 variables. So twelve registers are enough to compute any element of the constructible universe, and hence any ordinal computer with more than twelve registers. A reduction to four registers is simply technical, using oscilating stacks as in 2, repeating all finite intervals as many times as there are stack elements, and using the last variable to store a single variable, just as it appears on the stack, and store it, after the stack limits and is erased, very high on the stack, where it won't be erased by the varying and limiting of values lower on the stack. However, we will not prove rigorously in this paper that four register suffice.

Definition 11. *Let OC^n be the set of n-register well-structured ordinal computer programs (obeying rules 1 and 2'). Say $\rho : Ord^n \to \alpha^n$ reflects OC^n if for each P in OC^n, the function f_P which takes the inputs to P to the output of P, commutes with ρ. Let L_n be the vocabulary with a function for each n-register program: $L_n = \{Ord, <, =\} \cup \{f_P : P \in OC^n\}$, and let $FO^k(L)$ be the first order formulas in the language L, to quantifier depth k.*

Definition 12. *Let ρ_0 be the function $\rho_0(\alpha) = \alpha \mod \omega$.*

Let ρ_1 be the identity below ω, and be $\omega + \rho_0$ above ω.

Let ρ_2 be the identity below $\omega \times 2$, and be $\omega \times 2 + \rho_0$ above $\omega \times 2$.

Let $\rho_3(\alpha) = \alpha \mod \omega^\omega$.

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 $\alpha > \beta$.

Lemma 6. *$\rho_1 : Ord \to \omega \times 2$ reflects OC^1, is the minimal reflection preserving $FO^1(L_1)$, and preserves even $FO^2(L_1)$. ρ_2 preserves $FO^3(L_1)$.*

Corollary 1. *$FO^3(L_1)$ is much less expressive than the 4-quantifier theory of linear order, $FO^4(<)$. Indeed, $FO^4(<)$ can define every predicate definable in $FO^3(L_1)$.*

Lemma 7. *$\rho_5 : Ord^2 \to (\omega^\omega \times 3)^2$ reflects OC^2, is minimal such that it preserves $FO^2(L_2)$, and preserves $FO(L_2)$.*

Remark 5. Suppose that the rule 3b in definition 9 were relaxed, 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 with register values ω^ω).

Corollary 2. *$FO(L_2)$ computes $x \mapsto x \times \omega$ and $x \mapsto n \times x$ (for finite n), but it is weaker than $FO(Ord, <, c_0, c_1)$, where c_0 and c_1 are constants naming any two $\times$-closed ordinals; this is much weaker than $FO(Ord, +)$. So an ORM with two variables can not compute the $+$ of two input values.*

By theorem 2, OC^3 can simulate a finite turing machine. But OC^3 still reflects into a small ordinal, and as a result, we find that 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 $(\omega_1, +, \times, a \to \omega^a)$-register machine (an abstract register machine as in definition 4) must limit continuously without losing any information.

Lemma 8. *Ord^3 reflects below $\epsilon_{\omega \times 4}$, and not lower.*

2 Recursive Truth Predicate

Our main theorem is that infinite ordinal register machines can decide all sets of ordinals which are elements of the constructible hierarchy, L, i.e.: For every set of ordinals S which exists in L, there is an ordinal computer program P and a single ordinal input, the Γ-stack $\Gamma(...\Gamma(\alpha_0, \alpha_1)..., \alpha_n)$ of $(\alpha_0...\alpha_{n-1})$, which program decides S.

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 $a_0...a_n \in L$ is thereby constructed in L.

Theorem 4. *(analogous to Theorem 5 of [8]) 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.*

We prove the theorem, that everything in L can be computed by an ordinal computer (from some ordinal parameters), by computing the "recursive truth predicate" described in [8].

The recursive truth predicate is a recursive characteristic function on the ordinals, coding all constructible sets of ordinals. It is defined as

Definition 13. *Let T be the recursive truth predicate, defined by: $T(\alpha) = True$ if and only if $(\alpha, <, \Gamma, T \restriction \alpha) \models \phi_\alpha$, where Γ is the ordinal pairing function in definition 5, where the sentence ϕ_α is coded by α, has a finite number of ordinal parameters.*

Definition 14. *(F from H): $F(\alpha) = True \iff \exists \beta < \alpha\ H(\alpha, \beta, F(\beta)) = True$:*

```
for $\beta$ from 0 to $\alpha$ (
    if ($F(\beta)$ = False and H($\alpha, \beta$, False) = True) (return True);
    if ($F(\beta)$ = True and H($\alpha, \beta$, True) = True) (return True);
);
Return False
```

That program is written in a language which allows a subroutine to call itself. First, we show that from this recursive routine, set-theoretical constructibility can be carried out.

Theorem 5. *If an OC can simulate the recursive routine in definition 14, then theorem 4 holds.*

Now we simulate the program in definition 14 using a wellordered stack of formulas on an OC.

3 Stack, Pop, Push, IsEmpty

Definition 15. `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 ($\alpha + 2^\epsilon$+SmallStack = Stack) (
         if ($\epsilon >$ 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.

Lemma 9. Pop *reads least element* 2^ϵ *of the* Stack*, such that* ϵ *is at least as large as the parameter* Threshold*.*

Definition 16. 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.

Lemma 10. Push(β onto Stack) *increases the Stack to the next full multiple of* $2^{\beta+1}$*.*

Definition 17. IsEmpty*, taking the single input* Stack*, is the routine:*

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

IsEmpty(Stack) returns the value “True” when the stack is a singleton, $2^{\$_-}$, i.e., the initial value, the truth of which we would like to determine.

Definition 18. β `is the largest element on the stack` *is the program*

```
for $\iota < 2^\beta$ (if ($2^\beta + \iota =$ Stack) (Return Yes));
Return No
```

Clearly, this halts before ι exhausts β iff β is indeed the largest stack value. On the other hand, $2^\beta + \iota =$ `Stack` never holds if β is larger than the largest stack value, nor if β is less than the largest stack value.

3.1 The Recursive Truth Predicate OC Program

Theorem 6. *The recursive truth predicate F defined in 14 is equivalent to the following program:*

```
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 $2^{\$_-}$ (
    $\beta =$ Pop(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 $++$;$\beta = 0$);
        $\alpha =$ Pop(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);
        $\alpha =$ Pop(Stack,$\beta + 1$);
        Let $\nu = H(\alpha - 1, \beta - 1,$ TruthValue$)$;
        if ($\nu =$ True)($\beta = \alpha$;TruthValue$= \nu$);
        if ($\nu =$ False and $\alpha = \beta + 1$)($\beta = \alpha$;TruthValue$= \nu$);
        if ($\nu =$ False and $\alpha \neq \beta + 1$)(
            TruthValue=Unknown;
            Push($\beta$ onto Stack)
        )
    )
)
```

Definition 19. *Call $(\alpha_i : i \leq k)$ a witnessing sequence if, for each $i < k - 1$, $\alpha_i - 1$ is true and $H(\alpha_i - 1, \alpha_{i+1} - 1, F(\alpha_{i+1} - 1)) =$ True, and α_{k-1} is false and $\alpha_k = \alpha_{k-1} - 1$. Call $WIT(\gamma)$ the least witnessing sequence $(\alpha_i : i < k)$ such* *that $\gamma = \alpha_0$ and $\alpha_{k-1} - 1$ is false and a limit. Call $W(\gamma) = \sum_{\alpha \in W(\gamma)} 2^\alpha$ the witnessing series.*

Lemma 11. *a) If Stack $= \sum_{i<n} 2^{\gamma_i+1} + W(\gamma_n)$, where γ_n does not witness γ_{n-1} (i.e.: it is not the case that γ_n is true and $H(\gamma_{n-1}, \gamma_n, 1) = 1$, nor is it the case that γ_n is the predecessor of γ_{n-1} and γ_{n-1} is false), and $W(\gamma_n)$ is a series with m terms, then in the first part of the loop TruthValue becomes known and after m iterations of the while loop, the program passes command to* `Push(`γ_{n-1} `onto Stack)`.

b) If Stack $= W(\gamma_n)$, a series with m terms, then in the first part of the loop TruthValue becomes known and after m iterations of the while loop, the program halts, returning the truth value of $\gamma_0 - 1$.

Now we prove theorem 6 by induction on the following:

Lemma 12. *Intention Lemma:*

a. If Stack is $2^{\$_-+1} + \sigma + 2^\gamma$, where $2^\gamma \times 2$ divides σ, γ is a successor, and TruthValue is Unknown, then the Stack will reach $2^{\$_-+1} + \sigma + W(\gamma)$, when $\iota \leq (\sigma \times 1/2) + 2^{\gamma-1}$, with $\beta = \gamma$ and TruthValue known.

b. If Stack is $2^{\$_-+1} + \sigma + 2^\gamma$, where $2^\gamma \times 2$ divides σ, and $\gamma - 1$ is a false limit then the stack will be $\geq 2^{\$_-+1} + \sigma + 2^\gamma + 2^\delta$, for each successor $\delta < \gamma - 1$, at some time $\iota \leq (\sigma \times 1/2) + 2^{\gamma-1} + 2^{\delta-1}$.

c. if the Stack is $2^{\$_-+1} = 2^\gamma$, and TruthValue is Unknown, then the Stack will reach $W(\gamma)$, when $\iota \leq 2^{\gamma-1}$, with $\beta = \gamma$ and TruthValue known, i.e., P halts on input $\$_-$ after at most $2^{\$_-}$ loops through the main loop, and returns the value $F(\$_-)$.

References

1. R. Bissell-Siders, *Ordinal computers.* math.LO/9804076 at arXiv.org, 1998.
2. H. Friedman, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory.* Logic Colloquium '69 (Proc. Summer School and Colloq., Manchester, 1969), pp. 361–389. North-Holland, Amsterdam, 1971.
3. G. Jacopini and C. Böhm, *Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules.* Comm ACM, 9,5 May 1966.
4. J. Hamkins and A. Lewis, *Infinite Time Turing Machines.* J. Symbolic Logic, 65(2): 567-604, 2000.
5. P. Koepke, *Infinite Time Register Machines.* Submitted to Computing In Europe 2006; this volume. 11(3): 377-397, 2005.
6. P. Koepke, *Turing Computations on Ordinals.* Bulletin of Symbolic Logic, 11(3): 377-397, 2005.
7. P. Koepke and M. Koerwien, *The Theory of Sets of Ordinals.* math.LO/0502265 at the e-print archive arXiv.org, 2005
8. P. Koepke and M. Koerwien, *Computing a Model of Set Theory.* CIE 2005.
9. M. Minsky, *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.
10. J. Shepherdson and H. Sturgis, *Computability of recursive functions*, J. Assoc. Comput. Mach. 10 217–255, 1963.
11. 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.