The Expressive Power of Higher-order Types or, Life without CONS

Neil D. Jones∗
DIKU, University of Copenhagen, Denmark
E-mail, web: `neil@diku.dk`, `http://www.diku.dk/people/NDJ.html`

Abstract

Compare first-order functional programs with higher-order programs allowing functions as function parameters. Can the the first program class solve fewer problems than the second? The answer is no: both classes are *Turing complete*, meaning that they can compute all partial recursive functions. In particular, higher-order values may be first-order simulated by use of the list constructor "cons" to build function closures.

This paper uses *complexity theory* to prove some expressivity results about small programming languages that are less than Turing complete. Complexity classes of decision problems are used to characterize the expressive power of functional programming language features. An example: second-order programs are more powerful than first-order, since a function f of type `[Bool]->Bool` is computable by a cons-free first-order functional program if and only if f is in PTIME, whereas f is computable by a cons-free second-order program if and only if f is in EXPTIME.

Exact characterizations are given for those problems of type `[Bool]->Bool` solvable by programs with several combinations of *operations on data:* presence or absence of constructors; the *order of data values:* 0, 1, or higher; and program *control structures:* general recursion, tail recursion, primitive recursion.

1 On expressivity in programming languages

Does the programming style we use affect the problems we can solve, or the efficiency of the programs we can write to solve a given problem? Some questions especially relevant to this journal:

1. Does the use of functions as data values give a greater problem-solving ability?
2. Is recursion more powerful than iteration?
3. Would the use of a strongly normalizing programming language, as proposed by several researchers, including (Crockett, Spencer, 1991), (Voda, 1994), (Turner, 1996), and (Lloyd, 1999), impose any limitations on the problems that can be solved?

This paper gives some answers to these questions, when restricted to simply typed functional programs without "cons". In particular, the answer to question 1 is

∗ This research was partially supported by the Danish Natural Science Research Council (*DART* project).

"yes," higher-order programs can solve problems that first-order programs cannot. The answer to question 2 for first-order programs is "yes" if and only if PTIME $\neq$ LOGSPACE (and so not likely to be decided in the foreseeable future). A partial (positive) answer to question 3 will be given later.

Programming styles. Factors that affect efficiency and ease of programming include the forms of *control* that are used: general recursion, primitive recursion, tail recursion, exceptions, backtracking, etc.; the *data operations* used: is data basic or higher order, is it untyped or typed by various regimes, are "logic variables" allowed, etc.; and *other semantic issues:* eager or lazy evaluation, memoization of function results, presence or absence of a global state, static or dynamic name binding, etc.

Our main goal is to understand (and delineate) the expressive power of various programming language features. In general, one can distinguish between absolute expressivity questions and relative expressivity questions. An "absolute expressivity" question on programming language feature X: "Do there exist problems that can be solved by programs with feature X, and *cannot be solved without feature* X?" A "relative expressivity" question: "Is there a problem that can be solved both with and without feature X, but such that *any solution without feature* X *is necessarily less efficient* than some solution using X?" (Efficiency will measured by time usage throughout this paper.)

How can expressivity questions be formulated in a precise and meaningful way, and how can their answers be found? Our approach is to use complexity theory, which provides a powerful means to classify problems according to their solution difficulty. In particular, there are theorems proving that, under suitable assumptions, increasing the available computation time provably enlarges the class of problems that can be solved, with analogous results for other resources such as memory.

An obstacle to studying absolute expressivity. Absolute expressivity questions are irrelevant to sufficiently strong languages. The reason is that any Turing complete language can compute all partial recursive functions (modulo data encoding) – and thus in an absolute sense all are equally expressive. We get around this by studying *Turing-incomplete* sublanguages.

Obstacles to studying relative expressivity. One difficulty is the existence of many rather efficient simulations of one programming language feature by others. A consequence is that in a strong language it is hard to answer relative expressivity questions since any expected complexity differences would be very small. This is due to the ability efficiently to "program your way around a problem" in a sufficiently strong language. Two related facts of this sort follow. (Running time is measured as the number of atomic computational steps as a function of input length n.)

- A random-access machine can simulate a first-order or higher-order functional program (or Turing machine) with at most a constant slowdown.
- A first-order functional program can simulate a random-access machine with at most a logarithmic slowdown, from time $T(n)$ to $T(n)\log T(n)$.

Such results tend to minimize the meaning of differences in programming paradigm, provided the languages involved are sufficiently rich in ways to "program around."

Our approach to studying expressivity. A language having unbounded storage/memory for data values, plus control allowing an unbounded number of operations on data values, will almost certainly be Turing complete. A way around this obstacle is progressively to restrict language features, removing one at a time until the resulting programming language is no longer Turing complete, and then to use computability and complexity theory to compare the absolute expressive power of the resulting "minilanguages." The effect is to study expressivity of language constructions using complexity theory to compare different choices of language features.

2 Overview and interpretation of results

We precisely characterize, in terms of complexity classes, the effects on expressive power of various combinations of three program restrictions. The first concerns *creation of new storage:* are constructors of structured data allowed, or not? The second concerns *the order of data values:* 0, 1, or higher. The third concerns *program control structures*: general recursion, or only tail recursion, or only primitive recursion. The links are summed up in the table of Figure 2.1, and confirm programmers' intuitions that higher-order types indeed give a greater problem-solving ability. In this paper we prove only the results of rows 3 and 4, the others being included for the sake of context.

Many combinations are Turing-complete, so such programs compute all the partial recursive functions. A classic Turing-incomplete language is got by restricting data to order 0 and control to "fold." Such programs compute the *primitive recursive* functions.

Figure 2.1 shows the effect of higher-order types on the computing power of programs of type `[Bool]->Bool`. Each entry is a complexity class, i.e., the collection of decision problems solvable by programs restricted by row and column indices. RO stands for "read-only," i.e., programs without constructors, and RW stands for "read-write."

Program class	**Data order 0**	**Order 1**	**Order 2**	**Order 3**	**Limit**
RW, unrestricted	REC.ENUM	REC.ENUM	REC.ENUM	REC.ENUM	REC.ENUM
RWPR, fold only	PRIM.REC	PRIM^1REC	PRIM^2REC	PRIM^3REC	$\text{PRIM}^\omega\text{REC}$
RO, unrestricted	PTIME	EXPTIME	EXP^2TIME	EXP^3TIME	ELEMENTARY
ROTR tail recursive	LOGSPACE	PSPACE	EXPSPACE	EXP^2SPACE	ELEMENTARY
ROPR, fold only	LOGSPACE	PTIME	PSPACE	EXPTIME	ELEMENTARY

Figure 2.1: Expressivity of several combinations of control and data orders

First, we need to justify the formulation, i.e., to argue that Figure 2.1's results say meaningful things about programming languages. Complexity theory is traditionally used to classify hardness of decision problems, so we need to link decision problems with programs and their computations.

Linking decision problems and functional programs. In complexity theory a decision problem A is a set of strings over a finite alphabet Σ, so $A \subseteq \Sigma^*$. A solution to the problem is an algorithm that, given any $x = a_1 a_2 \ldots a_n \in \Sigma^*$, decides whether or not $x \in A$. On the other hand, the effect of a functional program p is to compute an input-output function $[\![\mathtt{p}]\!]$: $In \to Out$ over data sets In, Out given by p's declarations. To solve a decision problem, program p can take as input a list of symbols, and return a truth value.

We can without loss of generality choose $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, since larger alphabets can be encoded as bit strings. The *characteristic function* f_A of set $A \subseteq \{\mathtt{0}, \mathtt{1}\}^*$, of type type $f_A : \{\mathtt{0}, \mathtt{1}\}^* \to \{\mathtt{0}, \mathtt{1}\}$, satisfies for all $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \in \{\mathtt{0}, \mathtt{1}\}$

$$f(\mathtt{a}_1 \mathtt{a}_2 \ldots \mathtt{a}_n) = \text{ if } \mathtt{a}_1 \mathtt{a}_2 \ldots \mathtt{a}_n \in A \text{ then } \mathtt{1} \text{ else } \mathtt{0}$$

Henceforth we shall identify 0,1 with `False`, `True` and encode string $\mathtt{a}_1 \mathtt{a}_2 \ldots \mathtt{a}_n$ as boolean list $[\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]$, making the analogy between decision problems and programs with input-output type `[Bool] -> Bool` exact.

More generally, Figure 2.1 concerns computational power of fully typed functional programs whose internal types τ are limited to ones formed from `Bool` and `[Bool]` by function spaces $\tau \to \tau'$ and Cartesian products $\tau \times \tau'$. No type of numbers is included, since if the numbers are bounded they can be simulated by tuples of Booolean variables; and if the numbers are unbounded, strange and unrealistic computations can be performed (as will be seen below).

Further, complexity and computability classes are invariant under many changes of data, problem, and even function representation. For example, if $f : \{\mathtt{0}, \mathtt{1}\}^* \to \{\mathtt{0}, \mathtt{1}\}^*$ is in LOGSPACE, PTIME, etc. and $|f(x)|$ is bounded by a polynomial in $|x|$, then f is computable if and only if the function "$\lambda x, i$. the i-th bit of $f(x)$" is computable. This justifies considering only programs with a single bit as output.

Restricting programs to input-output type `[Bool]->Bool` side-steps two superficial differences in expressivity having to do with computed values. One is that if a larger computation time is available, a longer results can be written out. Another is that higher-order data types allow new values to be expressed. It seems unreasonable, though, to regard either ability as greater expressiveness. If input-output data is restricted to `[Bool]->Bool` then higher-order data, whether or not used internally, have no effect on observable program behavior.

2.1 Interpretation of the results.

Figure 2.1 shows the effect of higher-order types on the computing power of various restricted program classes. Some of the table entries have analogues, more or less close, in the existing literature (see below). The formulations, definitions and contructions using functional programming are our own, and row 4, on higher-order tail recursive programs are new, to the best of the author's knowledge.

Explanation of the table. The restrictions RO and RW were explained above. With or without these restrictions, programs may have general recursion, tail recursion, or primitive recursion, yielding 6 combinations. There are only 5 rows, though, since RW=RWTR because an unrestricted program can be converted into a tail recursive equivalent by standard techniques involving a stack of activation records.

The column indices restrict the orders of program data types. An "order $k+1$" program can have functions of type $\tau \to \tau'$ where data type τ is of order k. Thus, for instance, the first column describes first-order programs, whose parameters are booleans or lists of booleans. Each entry is the collection of decision problems solvable by programs restricted by row and column indices.

Row 1: These program classes are all Turing complete. Consequently they can accept exactly the recursively enumerable subsets of $\{0, 1\}^*$.

Row 2: These programs have unlimited data operations and types, but control is limited to primitive recursion, familiar to functional programmers under the name "fold right".[1] Such first-order programs accept exactly the sets whose characteristic functions are primitive recursive (true regardless of whether data are strings or natural numbers).

Higher-order primitive recursive functions appeared in Gödel's *System T* many years ago (Girard, Lafont, Taylor, 1989), (Voda, 1997). They are currently of much interest in the field of constructive type theory due to the Curry-Howard isomorphism, which makes it possible to extract programs from proofs. Primitive recursion comes because of proofs by induction; extraction of programs using general recursion is much less natural.

Row 3: These programs have unlimited control, but allow only *read-only* access to their data. List destructor operations `hd` and `tl` are allowed, but not the constructor `cons`. Even though this may seem a draconian restriction from a programmer's viewpoint, the class of problems that can be solved is respectably large. Order 1 programs can solve any problem that lies in PTIME; order 2 programs, with first-order functions as data values, can solve any problem in the quite large class EXPTIME, etc. In general, any increase in the order of data types leads to a proper increase in the solvable problems, since it is known that PTIME is properly contained in EXPTIME, and so on up the hierarchy.

Row 4 characterizes read-only programs restricted to *tail recursion,* in which no function may call itself in a nested way. Order 1 tail recursive programs accept all and only problems in LOGSPACE, a well-studied subset of PTIME. Higher-order tail recursive programs accept problems in the (properly) larger space-bounded classes PSPACE, EXPSPACE, etc.

(Tail recursion is of operational interest because at run time (assuming eager evaluation, i.e., call-by-value semantics) the call stack depth has a constant depth bound, regardless of input data. Such a program may be converted to nonrecursive imperative form by replacing each function call by a `GOTO`, and realizing function parameter passing by assignments to global variables.)

[1] Kleene's definition of primitive recursion is a bit more general than "fold right," but is easily programmed using fold right and composition. See (Hutton, 1999) for details.

Row 5 characterizes read-only programs restricted so all recursion must be expressed using "fold right," i.e., only primitive recursion is allowed. Order 1 read-only primitive recursive programs accept only problems in LOGSPACE and are thus equivalent to tail-recursive programs. At higher orders this equivalence vanishes; the primitive recursive read-only programs' abilities to solve decision problems grow only at "half speed": a data order increase of 2 is needed to achieve the same increase in decision ability that an increase of 1 achieved for general or tail recursive programs.

Limit of rows 3, 4 and 5 It is clear that the union of the classes in row 3 equals the union for row 4 and for row 5. This is the class of problems solvable in time bounded by $2^{2^{\cdots 2^n}}$, where the height of the exponent stack is any natural number. This is well-known as the class of *elementary* sets, and was studied by logicians before complexity theory began.

Scope and contribution of this paper. The results in Rows 1 and 2 are classical, and not repeated here. We prove the results in rows 3 and 4 of Figure 2.1. The results in Row 4 appear to be new; and the results in Row 3, while in a sense anticipated by (Goerdt, 1992), are here proven for the first time in a programming language context. The results in Row 5 are obtained from (Goerdt, 1992) and (Goerdt, Seidl, 1996) by re-interpreting results from finite model theory as sketched in the Appendix.

2.2 On the questions opening this article

It has long been known that order $k+1$ primitive recursive programs are properly more powerful than order k primitive recursive programs, i.e., PRIM^{k}REC. $\subset$ PRIM^{k+1}REC. This is of little practical interest, however, since even the order 0 class PRIM.REC. is enormous, properly containing such classes as NPTIME and ELEMENTARY.

Does the use of functions as data values give a greater problem-solving ability? By Figure 2.1 the answer is "no" for unrestricted programs, and "yes" for all the restricted languages we consider. The only uncertainty is with the read-only primitive recursive programs; for these, an increase in data order of at least 2 is needed in order to guarantee a proper increase in problem-solving power.

Is general recursion more powerful than tail recursion? For first-order read-only programs, this question has classical import since, by the table's first column (rows 3, 4) this is equivalent to the question: Is PTIME a proper superset of LOGSPACE? This is, alas, an unsolved question, open since it was first formulated in the early 1970s. An equivalent question (rows 3, 5): *Is general recursion more powerful than primitive recursion?*

However the situation is different for second and higher orders. For higher-order read-only programs, the question of whether general recursion is stronger than tail recursion is also open, equivalent to EXPTIME $\supset$ PSPACE? But the answer is "yes" when comparing general recursion to primitive recursion, since it is known that EXPTIME properly includes PTIME.

On strongly normalizing languages. If we assume as usual that programs in a strongly normalizing language have only primitive recursive control, there exist problems solvable by read-only general recursive programs with data order $1, 2, 3, \ldots$, but not solvable by read-only strongly normalizing programs of the same data orders. This suggests an inherent weakness in the extraction of programs from proofs by the Curry-Howard isomorphism.

2.3 A paradox? Intensional versus extensional program behavior.

Row 3, column 1 of Figure 2.1 asserts that first-order cons-free read-only programs can solve all and only the problems in PTIME. Upon reflection this claim seems quite improbable, since it is easy (without using higher-order functions) to write cons-free read-only programs that run exponentially long before stopping. For example:

```
f x  = if  x = []  then true    else
       if  f(tl x) then f(tl x) else false
```

runs in time $\Omega(2^n)$ on an input list of length n (regardless of whether call-by-value or lazy semantics are used), due to computing `f(tl x)` again and again.

What is wrong? The seeming paradox disappears once one understands what it is that the proof accomplishes[2]. It has two parts:

- *Construction 1* shows that any first-order cons-free read-only program decides a problem in PTIME. Method: show how to simulate an arbitrary first-order cons-free read-only program by a polynomial-time algorithm.
- *Construction 2* shows that any problem in PTIME is computable by some first-order cons-free read-only program. Method: show how to simulate an arbitrary polynomial-time Turing machine by a first-order cons-free read-only program.

The method of Construction 1 in effect shows how to simulate a cons-free read-only program *faster than it runs.* It is not a step-by-step simulation, but uses a nonstandard "memoizing" semantic interpretation. (For the example above the value of `f(tl x)` would be saved when first computed, and fetched from memory each time the call is subsequently performed.)

The method of Construction 2 yields programs that almost always take exponential time to run; but this is not a contradiction since by Construction 1 the problems they are solving can be decided in polynomial time.

3 Related work and acknowledgements

Much work that relates complexity and logical or programming languages has different starting points and motivations. Many papers have as goal to characterize

[2] For first-order cons-free read-only programs see (Jones, 1997; Jones, 1999), which uses a technique from (Cook, 1971). For higher-order read-only programs, see Theorem 7.17 later in this article.

resource-bounded computational complexity classes in terms of logical, recursion-theoretic or other formalisms. Examples (ordered by increasing distance from programming languages) include recursive functions, the lambda calculus, program schemata, finite model theory, and category theory.

Work directly relating programming languages and complexity theory. This paper's aim is precisely to characterize the computational power of several restrictions of a realistic programming language. (Jones, 1999) is one of the few papers focused on that interface; it characterizes the power of first-order read-only programs in complexity terms. The LOGSPACE and PTIME entries in column 1 appear there, and in (Jones, 1997) as Theorems 24.1.7, Corollary 24.2.4 and Theorem 24.2.5. Here these results are extended to arbitrary finite data orders, and to tail recursive programs.

The results have still deeper roots. The LOGSPACE result strengthens a "folklore" result on multihead read-only Turing machines; and the PTIME result corresponds to a result on "two-way auxiliary pushdown automata" proven by Cook, before his P-NP paper (Cook, 1971). The contribution of (Jones, 1997; Jones, 1999) was to re-express Cook's result using a recursive programming language.

Relative expressivity. A significant step forward in relative expressivity (a field with many conjectures but few proven results) was (Pippenger, 1997), which showed that pure LISP must be slower by a logarithmic factor than "impure" LISP. More precisely, a certain problem (applying a permutation on-line) was proven to require time $\Omega(n \log n)$ in pure Lisp, but to be solvable in time $O(n)$ in impure Lisp with `setcar` and `setcdr`. Interestingly, (Bird, Jones, De Moor, 1998) show that the same problem can be solved in time $O(n)$ in a language with lazy evaluation, so such languages are intrinsically faster than first-order eager languages. A proof that their construction actually runs in linear time is found in (Neergaard, 1999).

Recursive function and category theory. Recursive function definition schemes can be regarded as defining programming languages, for instance the primitive recursive function definitions, and the subhierarchy studied by (Grzegorczyk, 1953) and others. (Cobham, 1964) gave an early characterization of PTIME involving external size bounds analogous to those of Grzegorczyk, but using recursion on notation. (Voda, 1994) relates primitive recursion and subrecursion to programming languages, using a data structure like that of (Jones, 1997). An "intrinsic" approach characterizing PTIME by tiered recursion on notation[3] is seen in (Bellantoni and Cook, 1992), and has inspired much work since then, some involving categorical concepts (Bellantoni, Niggl, Schwichtenberg, 1998; Hofmann, 1999).

Typed lambda calculi. A series of papers by Leivant (some with Marion) characterize complexity clases in terms of simply typed lambda calculi with recurrence constants,

[3] The idea is that recursion over inputs is allowed (first tier), but not recusrsion over computed values (second tier). A type system keeps track of the tier levels.

including (Leivant, 1989; Leivant and Marion, 1993; Leivant and Marion, 1999). In particular they study calculi extended by functions and operations on an algebra of words over $\{0,1\}$, and characterize PTIME, PSPACE and several other classes. The earlier works had rather complex formulations mostly related to ramified recurrence, but (Leivant, 1999) characterizes these classes by much simpler syntactic restrictions on the form of program control.

(Hillebrand, Kanellakis, 1996) studies expressivity of lambda calculus variants that abstract an ML subset, characterizing the regular sets, and k-Exptime and k-Expspace queries over ordered finite structures. The paper makes an odd assumption, that encodings of first-order data can vary from one program to another. (Hillebrand, 1994) studies relations among the lambda calculus, finite model theory, and databases.

Program schemata. (Paterson, Hewitt, 1970) was a pathbreaking early paper, and recursion removal has been a recurring theme in this field. Complexity characterizations related to data types appear in (Kfoury, Tiuryn, Urzyczyn, 1987) and (Kfoury, Tiuryn, Urzyczyn, 1992). The difference between recursion and iteration when higher order data are involved is studied in (Kfoury, 1997; Kfoury, 1999).

From a schematic viewpoint, answers to the questions asked about program behavior must be valid for *all possible interpetations* of the domains and base functions appearing in a given program. Such results are less directly relevant to programming languages than ours; programmers naturally use a single, fixed data interpretation.

Finite model theory. Many complexity characterizations have been made of problems involving finite model theory, with (Jones, Selman 74) apparently the first in a field developed quite considerably since then (Gurevich, 1983; Gurevich, 1984; Immerman, 1987; Goerdt, 1992).

There is a natural connection between computation by read-only programs and in finite model theory as defined by Gurevich and others, close enough that some complexity characterizations from finite model theory imply corresponding results about read-only programs. The connection requires enough definitional machinery, though, that to avoid breaking the continuity of the presentation we defer it to an Appendix. The only results we do not prove here using functional programming are those of row 5.

Finite model versions of the LOGSPACE and PTIME entries in column 1 were shown by Gurevich. Goerdt proved finite model versions of all of row 3. Some rather complex constructions establish the first 4 columns in row 5 in (Goerdt, 1992; Goerdt, 1992); and (Goerdt, Seidl, 1996) shows that the space-time alternation extends to arbitrary data orders.

Acknowledgements.

Thanks are due to Bruce Kapron, for pointing out the similarity between Gurevich and Goerdt's finite model theory results with mine on first-order read-only programs; to the anonymous referees; to Amir Ben-Amram for many useful discussions; to Peter Møller Neergaard for pointed and useful comments on an earlier draft; to Pawel Urzyczyn and Assaf Kfoury; and to Daniel Leivant for general encouragement and inspiration in this borderland between complexity theory and programming languages.

4 A functional programming language and some sublanguages

4.1 Syntax

Our programs are all expressed in a Haskell-like named combinator form. This is well-known to be equivalent to the lambda calculus with explicit binding and recursion operators λ and μ. Semantics-preserving constructions taking named combinator to lambda form and vice versa may be seen in (Goerdt, 1992), or in many other sources. For notational simplicity, syntax and semantics are first given for untyped programs.

Definition 4.1
Syntax: programs and expressions have forms given by the grammar

$$
\begin{array}{rcl}
\mathtt{p} \in \text{Program} & ::= & \mathtt{def}_1\ \mathtt{def}_2\ \ldots\ \mathtt{def}_m \\
\mathtt{def} \in \text{Definition} & ::= & \mathtt{f}\ \mathtt{x}_1\mathtt{x}_2\ldots\mathtt{x}_n = \mathtt{e}^{\mathtt{f}} \\
\mathtt{e} \in \text{Expression} & ::= & \mathtt{x} \mid \mathtt{f} \mid \mathtt{e}_1\mathtt{e}_2 \mid \mathtt{False} \mid \mathtt{True} \mid \mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\mathtt{e}_2 \\
& \mid & \mathtt{[]} \mid \mathtt{e}_1\mathtt{:}\mathtt{e}_2 \mid \mathtt{hd}\ \mathtt{e} \mid \mathtt{tl}\ \mathtt{e} \mid \mathtt{null}\ \mathtt{e} \\
& \mid & (\mathtt{e}_1,\mathtt{e}_2) \mid \mathtt{fst}\ \mathtt{e} \mid \mathtt{snd}\ \mathtt{e} \\
\mathtt{x} \in \text{Parameter} & ::= & \text{identifier} \\
\mathtt{f} \in \text{FcnName} & ::= & \text{identifier, disjoint from Parameter}
\end{array}
$$

The *main function* $\mathtt{f}_1$ of program `p` is the one in $\mathtt{def}_1$ above. (It will later be required to have type `[Bool]->Bool`, defined below.) The *definition of function* `f` has form $\mathtt{f}\,\mathtt{x}_1\mathtt{x}_2\ldots\mathtt{x}_n = \mathtt{e}^{\mathtt{f}}$, where $\mathtt{e}^{\mathtt{f}}$ is called the *body of* `f`. The number $n > 0$ of parameters in the definition of `f` is called its *arity*. The main function must have $\mathit{arity}(\mathtt{f}_1) = 1$.

A *read-only* (or *cons-free*) program is one with no constructor operator "`:`".

Program examples will be given in a Haskell-like syntax, using `if` $\mathtt{e}_0$ `then` $\mathtt{e}_1$ `else` $\mathtt{e}_2$ in place of `if` $\mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2$, and Haskell conveniences such as pattern matching, `where` and `case` expressions. We leave it to the reader to check that they may be converted into the simple syntax above.

4.2 Semantics

Our language has a closure-based call-by-value semantics (laziness would give no programming advantages, and would require a more complex semantics). Expression evaluation is based on a set of inference rules appearing in Figure 4.1, one for each form of expression in the language.

The *principal* judgement form for running program `p` is: $[\![\mathtt{p}]\!]v = w$, signifying that `p`, when given input v, terminates with output w. The form for evaluating

Expression evaluation

Some axioms

$$\frac{}{\texttt{p}, env \vdash \texttt{x} \rightarrow env(\texttt{x})} \qquad \frac{}{\texttt{p}, env \vdash \texttt{f} \rightarrow \langle \texttt{f}, \varepsilon \rangle}$$

$$\frac{}{\texttt{p}, env \vdash \texttt{False} \rightarrow \texttt{False}} \qquad \frac{}{\texttt{p}, env \vdash \texttt{True} \rightarrow \texttt{True}} \qquad \frac{}{\texttt{p}, env \vdash \texttt{[]} \rightarrow \texttt{[]}}$$

Cons and pairs

$$\frac{\texttt{p}, env \vdash \texttt{e}_1 \rightarrow v_1 \quad \texttt{p}, env \vdash \texttt{e}_2 \rightarrow v_2}{\texttt{p}, env \vdash \texttt{e}_1\texttt{:}\texttt{e}_2 \rightarrow v_1{:}v_2} \qquad \frac{\texttt{p}, env \vdash \texttt{e} \rightarrow v_1{:}v_2}{\texttt{p}, env \vdash \texttt{hd e} \rightarrow v_1} \qquad \frac{\texttt{p}, env \vdash \texttt{e} \rightarrow v_1{:}v_2}{\texttt{p}, env \vdash \texttt{tl e} \rightarrow v_2}$$

$$\frac{\texttt{p}, env \vdash \texttt{e}_1 \rightarrow v_1 \quad \texttt{p}, env \vdash \texttt{e}_2 \rightarrow v_2}{\texttt{p}, env \vdash \texttt{(e}_1\texttt{,e}_2\texttt{)} \rightarrow (v_1, v_2)} \qquad \frac{\texttt{p}, env \vdash \texttt{e} \rightarrow (v_1, v_2)}{\texttt{p}, env \vdash \texttt{fst e} \rightarrow v_1} \qquad \frac{\texttt{p}, env \vdash \texttt{e} \rightarrow (v_1, v_2)}{\texttt{p}, env \vdash \texttt{snd e} \rightarrow v_2}$$

Null test

$$\frac{\texttt{p}, env \vdash \texttt{e} \rightarrow v \quad \vdash^{null} v \rightarrow w}{\texttt{p}, env \vdash \texttt{null e} \rightarrow w} \qquad \frac{}{\vdash^{null} [\,] \rightarrow \texttt{True}} \qquad \frac{}{\vdash^{null} v_1{:}v_2 \rightarrow \texttt{False}}$$

Conditional

$$\frac{\texttt{p}, env \vdash \texttt{e}_1 \rightarrow w \qquad \texttt{p}, env \vdash^{if} \texttt{e}_2\texttt{,}\texttt{e}_3, w \rightarrow v}{\texttt{p}, env \vdash \texttt{if}\ \texttt{e}_1\texttt{e}_2\texttt{e}_3 \rightarrow v}$$

$$\frac{\texttt{p}, env \vdash \texttt{e}_2 \rightarrow v}{\texttt{p}, env \vdash^{if} \texttt{e}_2\texttt{,}\texttt{e}_3, \texttt{True} \rightarrow v} \qquad \frac{\texttt{p}, env \vdash \texttt{e}_3 \rightarrow v}{\texttt{p}, env \vdash^{if} \texttt{e}_2\texttt{,}\texttt{e}_3, \texttt{False} \rightarrow v}$$

Function call

$$\frac{\texttt{p}, env \vdash \texttt{e}_1 \rightarrow u \quad \texttt{p}, env \vdash \texttt{e}_2 \rightarrow v \quad \texttt{p} \vdash^{call} u, v \rightarrow w}{\texttt{p}, env \vdash \texttt{e}_1\texttt{e}_2 \rightarrow w}$$

$$\frac{}{\texttt{p} \vdash^{call} \langle \texttt{f}, v_1 \ldots v_{i-1} \rangle, v_i \rightarrow \langle \texttt{f}, v_1 \ldots v_{i-1} v_i \rangle} \quad (\text{If } i < arity(\texttt{f}))$$

$$\frac{\texttt{p}, [\texttt{x}_1 \mapsto v_1, \ldots, \texttt{x}_m \mapsto v_m] \vdash \texttt{e}^{\texttt{f}} \rightarrow w}{\texttt{p} \vdash^{call} \langle \texttt{f}, v_1 \ldots v_{m-1} \rangle, v_m \rightarrow w} \quad (\text{If } \texttt{p} \text{ contains } \ \texttt{f}\ \texttt{x}_1\texttt{x}_2 \ldots \texttt{x}_m = \texttt{e}^{\texttt{f}}) \,)$$

Program execution:

$$\frac{\texttt{p}, [\texttt{x} \mapsto v] \vdash \texttt{e}^{\texttt{f}} \rightarrow w}{[\![\texttt{p}]\!] v = w} \ (\text{If } \texttt{p} \text{ begins with } \ \texttt{f x} = \texttt{e}^{\texttt{f}})$$

Figure 4.1: Expression evaluation

expressions is $\texttt{p}, env \vdash \texttt{e} \rightarrow v$, signifying that expression $\texttt{e}$ in program $\texttt{p}$ evaluates to value v if its free variables have values defined by environment *env*. Environments, values, etc. are defined by

$$\begin{array}{rcl} env \in Env & ::= & \text{Parameter} \rightarrow Value \\ u, v, w \in Value & ::= & \texttt{True} \mid \texttt{False} \mid (v, w) \mid [\,] \mid v : v \mid cl \\ cl \in Closure & ::= & \langle \texttt{f}, v_1, \ldots, v_i \rangle \end{array}$$

A "closure" is a device to represent a data value which is a function. In λ-calculus implementations a closure consists of a λ-abstraction together with an environment holding the values of all its free variables. In our named combinator syntax, a simpler form can be used: a closure is a function name together with an incomplete parameter list, written $\langle \mathtt{f}, v_1 \ldots v_i \rangle$.

The rules are rather traditional for a call-by-value semantics. The rules for `if`, calls and `null` test are slightly nonstandard, in order to make the rules locally deterministic (term defined in Section 7). For example, to evaluate `if` $\mathtt{e}_1$ $\mathtt{e}_2$ $\mathtt{e}_3$, first evaluate the test $\mathtt{e}_1$, and then use an auxiliary judgement $\vdash^{if}$ to select between `then` and `else` branches.

4.3 ***Types***

A typed program has explicit simple types, without polymorphism. A complete program `p` must have type `p::[Bool] -> Bool`. All occurrences of a function name `f` in a program must have the same type throughout the program; but parameter name types need only be consistent within each function definition.

Definition 4.2
Types have a usual syntax and semantics with two basic types: Booleans and lists of Booleans.

$$\tau \in \mathit{Type} \quad ::= \quad \mathtt{Bool} \mid \ \mathtt{[Bool]} \mid \tau\mathtt{->}\tau \mid \mathtt{(}\tau\mathtt{,}\tau\mathtt{)}$$

Judgement $\mathtt{e} :: \tau$, signifying that *expression* `e` *has type* τ is defined in Figure 4.2. A fully type-annotated function definition has form

$$\mathtt{f}^{\tau_1 \to \tau_2 \to \ldots \tau_m \to \tau} \mathtt{x}_1^{\tau_1} \mathtt{x}_2^{\tau_2} \ldots \mathtt{x}_m^{\tau_m} \quad = \quad \mathtt{e}^{\tau}$$

Henceforth, all programs are assumed to be well-typed and fully annotated, i.e., a Church-style syntax is used. For readability type superscripts are omitted from program examples when clear from context.

Definition 4.3
The *order* of a type is defined by order(`Bool`) = order(`[Bool]`) = 0; order$((\tau, \tau'))$ = max(order(τ), order(τ')); and order$(\tau \to \tau')$ = max(1 + order(τ), order(τ')).

Definition 4.4
Program `p` has *data order* k if every τ, τ_i in any defined function has order k or less. Thus `f` above has order $k+1$ if at least one τ_i or τ has order k, justifying the usual term "first-order program" for one that manipulates data of order 0.

Definition 4.5
Type τ denotes a set of values $[\![\tau]\!]$ defined as follows.

$$\begin{array}{lcl}
[\![\mathtt{Bool}]\!] & = & \{\mathtt{True}, \mathtt{False}\} \\
[\![\mathtt{[Bool]}]\!] & = & \{[\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] \mid \mathtt{a}_i \in [\![\mathtt{Bool}]\!], 1 \le i \le n\} \\
[\![\mathtt{(}\tau\mathtt{,}\tau'\mathtt{)}]\!] & = & \{(v_1, v_2) \mid v_1 \in [\![\tau_1]\!], v_2 \in [\![\tau_2]\!]\} \\
[\![\tau_1\mathtt{->}\tau_2]\!] & = & \{f : [\![\tau_1]\!] \to [\![\tau_2]\!]\}
\end{array}$$

$$\frac{}{\mathtt{x}^\tau :: \tau} \quad \frac{}{\mathtt{f}^\tau :: \tau} \quad \frac{}{\mathtt{False :: Bool}} \quad \frac{}{\mathtt{True :: Bool}} \quad \frac{}{\mathtt{[] :: [Bool]}}$$

$$\frac{\mathtt{e}_1 :: \tau \to \tau' \quad \mathtt{e}_2 :: \tau}{\mathtt{e}_1\mathtt{e}_2 :: \tau'} \qquad \frac{\mathtt{e}_1 :: \mathtt{Bool} \quad \mathtt{e}_2 :: \tau \quad \mathtt{e}_3 :: \tau}{\mathtt{if}\ \mathtt{e}_1\mathtt{e}_2\mathtt{e}_3 :: \tau}$$

$$\frac{\mathtt{e}_1 :: \mathtt{Bool} \quad \mathtt{e}_2 :: \mathtt{[Bool]}}{\mathtt{e}_1\mathtt{:}\mathtt{e}_2 :: \mathtt{[Bool]}} \qquad \frac{\mathtt{e :: [Bool]}}{\mathtt{null\ e :: Bool}}$$

$$\frac{\mathtt{e :: [Bool]}}{\mathtt{hd\ e :: Bool}} \qquad \frac{\mathtt{e :: [Bool]}}{\mathtt{tl\ e :: [Bool]}}$$

$$\frac{\mathtt{e}_1 :: \tau \quad \mathtt{e}_2 :: \tau'}{(\mathtt{e}_1,\mathtt{e}_2) :: (\tau,\tau')} \qquad \frac{\mathtt{e} :: (\tau,\tau')}{\mathtt{fst\ e} :: \tau} \qquad \frac{\mathtt{e} :: (\tau,\tau')}{\mathtt{snd\ e} :: \tau'}$$

Figure 4.2: Expression types

Remarks on types: The restriction of our language to well-typed programs of type `[Bool]->Bool` is quite significant: the untyped version of the language contains the λ-calculus, and so is Turing complete.

We believe that the language could be simplified by removing pair construction and `fst`, `snd`, and still yield the same complexity class results. The Turing machine simulations would, however, become more complex.

4.4 Decision problems

Definition 4.6
Suppose program `p` of type `[Bool]->Bool` terminates on all inputs. Define:

1. Program `p` *accepts* string $\mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n \in \{\mathtt{0},\mathtt{1}\}^*$ iff $[\![\mathtt{p}]\!]\ [\mathtt{a}_1,\ldots,\mathtt{a}_n] = \mathtt{1}$.
2. The *set accepted by program* `p` is

$$\text{Accept}^{\mathtt{p}} = \{x \in \{\mathtt{0},\mathtt{1}\}^* \mid \mathtt{p} \text{ accepts } \mathtt{x}\ \}$$

Definition 4.7
Two classes of decision problems defined by syntactic restrictions on programs[4]:

$$\begin{array}{lcl} \text{RO}^k & = & \{\text{Accept}^{\mathtt{p}} \mid \text{read-only program } \mathtt{p} \text{ has data order } k\} \\ \text{ROTR}^k & = & \{\text{Accept}^{\mathtt{p}} \mid \text{read-only tail recursive program } \mathtt{p} \text{ has data order } k\} \end{array}$$

We will see that the problem of deciding question "$x \in A$?" for a set $A \subseteq \{\mathtt{0},\mathtt{1}\}^*$ can be solved by a first-order read-only program if and only if the question is decidable by a polynomial-time algorithm, i.e., iff the set A is in PTIME. Analogous results will be seen for higher types and complexity classes.

[4] The restriction "tail recursive" will be defined later, shortly before its use.

5 Turing machines and complexity classes

Definition 5.1
A Turing machine program has form $\texttt{tm} = \texttt{I}_1\ \texttt{I}_2\ \ldots\ \texttt{I}_m$ where each instruction $\texttt{I}_\ell$ has one of the forms `right`, `left`, `write S` or `if S goto` ℓ' `else` ℓ'' where $\texttt{S} \in \{\texttt{0},\texttt{1},\texttt{B}\}$ and $\ell, \ell' \in \{1, 2, \ldots, m, m+1\}$.

A *tape* is a sequence of symbols from $\{\texttt{0},\texttt{1},\texttt{B}\}$. Conceptually the tape has an infinite number of blanks to the right, even though any single computation will be finite. However, time- and space-bounded Turing machines will have natural bounds for tape length. A *configuration* C is a pair of control point and tape with scanned symbol:

$$C = (\ell, \texttt{b}_0\texttt{b}_1 \ldots \underline{\texttt{b}_i}\,\texttt{b}_{i+1} \ldots)$$

where $\ell \in \{1, 2, \ldots, m, m+1\}$ is a control point (and $m+1$ indicates "end of execution"), each $\texttt{b}_i \in \{\texttt{0},\texttt{1},\texttt{B}\}$, all but finitely many b_i's are B, and the underline indicates the scanned symbol. We call i the *scanning position* in C, and write $\mathit{scanpos}(C) = i$.

Semantics is as usual. A computation by program tm on input $\texttt{a}_1\texttt{a}_2 \ldots \texttt{a}_n \in \{\texttt{0},\texttt{1}\}^*$ (each $\texttt{a}_i \in \{\texttt{0},\texttt{1}\}$) is a series $\texttt{tm} : C_0 \vdash C_1 \vdash \ldots \vdash C_r$ where

- $C_0 = (1, \underline{\texttt{B}}\texttt{a}_1 \ldots \texttt{a}_n\,\texttt{B} \ldots)$, the initial configuration for $\texttt{a}_1\texttt{a}_2 \ldots \texttt{a}_n \in \{\texttt{0},\texttt{1}\}^*$.
- The final configuration is of form $C_r = (m+1, \texttt{b}_0\texttt{b}_1 \ldots \underline{\texttt{b}_i}\,\texttt{b}_{i+1} \ldots)$.
- Each configuration C_{t+1} for $t < r$ follows from $C_t = (\ell, \texttt{b}_0\texttt{b}_1 \ldots \underline{\texttt{b}_i}\,\texttt{b}_{i+1} \ldots \texttt{b}_q)$ by applying instruction $\texttt{I}_\ell$. In general, the tape component and scanning position of C_{t+1} are identical to those of C_t, unless changed by instruction $\texttt{I}_\ell$ as now described.
- The effect of $\texttt{I}_\ell =$ `right`, `left` are to move the scanning position one symbol right or left, though a move beyond the tape's left end has no effect. The effect of `write S` is to overwrite the scanned square's content $\texttt{b}_i$ with S. Instruction $\texttt{I}_\ell =$ `if S goto` ℓ' `else` ℓ'' sets the control point in C_{t+1} to ℓ' if $b_i = \texttt{S}$, else to ℓ'', and leaves the tape unchanged.

Definition 5.2
Let $S, T : \mathbb{N} \to \mathbb{N}$ be functions where $S(n) \geq n, T(n) \geq n$.

1. String $\texttt{a}_1\texttt{a}_2 \ldots \texttt{a}_n \in \{\texttt{0},\texttt{1}\}^*$ is *accepted* by tm iff the scanned symbol $\texttt{b}_i$ in C_r is 1.
2. Suppose Turing machine program tm terminates on all inputs. We define the *set accepted by* tm to be

   $$\text{Accept}^{\texttt{tm}} = \{x \in \{\texttt{0},\texttt{1}\}^* \mid \texttt{tm}\text{ accepts }\texttt{x}\ \}$$

3. Turing program tm *runs in time* $T(n)$ if for all $\texttt{a}_1\texttt{a}_2 \ldots \texttt{a}_n \in \{\texttt{0},\texttt{1}\}^*$, tm has a computation $\texttt{tm} : C_0 \vdash C_1 \vdash \ldots \vdash C_r$ where $r \leq T(n)$.
4. Turing program tm *runs in space* $S(n)$ if if for all $\texttt{a}_1\texttt{a}_2 \ldots \texttt{a}_n \in \{\texttt{0},\texttt{1}\}^*$, tm has a computation $\texttt{tm} : C_0 \vdash C_1 \vdash \ldots \vdash C_r$ with $\mathit{scanpos}(C_i) \leq S(n)$ for $i \in \{0, 1, \ldots, r\}$.
5. $\textsc{time}(T(n)) = \{\text{Accept}^{\texttt{tm}} \mid \text{Turing machine tm runs in time } T(n)\}$

6. $\textsc{space}(S(n)) = \{\text{Accept}^{\texttt{tm}} \mid \text{Turing machine } \texttt{tm} \text{ runs in space } S(n)\}$

Definition 5.3

$$\begin{array}{lcl} exp_0(n) & = & n \\ exp_{k+1}(n) & = & 2^{exp_k(n)} \end{array}$$

We use base 2 logarithms and assume $\log 0 = \log 1 = 1$ to avoid special cases for program inputs of length 0 or 1. Time or space bounds will have form $exp_k(a \log n)$. This is a polynomial for $k = 1$ since $2^{a \log n} = n^a$.

Definition 5.4

$$\begin{array}{lcll} \textsc{exp}^k\textsc{time} & = & \bigcup_a \textsc{time}(exp_{k+1}(a \log n)) & \text{For } k \geq 0 \\ \textsc{exp}^k\textsc{space} & = & \bigcup_a \textsc{space}(exp_{k+1}(a \log n)) & \text{For } k \geq 0 \end{array}$$

$$\begin{array}{lcl} \textsc{ptime} & = & \textsc{exp}^0\textsc{time} \\ \textsc{pspace} & = & \textsc{exp}^0\textsc{space} \\ \textsc{exptime} & = & \textsc{exp}^1\textsc{time} \\ \textsc{expspace} & = & \textsc{exp}^1\textsc{space} \end{array}$$

Sublinear time or space bounds. A bound of form $exp_0(a \log n) = a \log n$ does not fit our formulation, since $a \log n$ is less than the minimum space n required to store the input, or the time to read all of it.

For space, however, a well-studied generalization is to use a Turing machine with two tapes: a *read-only input* tape containing the input, and a *read-write work tape*, whose length is the measure of space usage. Formal definition is omitted, as we make no constructions with this model, only referring to results known from the literature. Given this extended definition, class $\textsc{space}(exp_0(a \log n)) = \textsc{space}(a \log n)$ is meaningful, and we define

$$\textsc{logspace} \quad = \quad \textsc{exp}^{-1}\textsc{space} \quad = \quad \bigcup_a \textsc{space}(a \log n)$$

Theorem 5.5

1. $\textsc{exp}^{-1}\textsc{space} \subseteq \textsc{exp}^0\textsc{time} \subseteq \textsc{exp}^0\textsc{space} \subseteq \textsc{exp}^1\textsc{time} \subseteq \textsc{exp}^1\textsc{space} \subseteq \ldots$
2. Thus

$$\textsc{logspace} \subseteq \textsc{ptime} \subseteq \textsc{pspace} \subseteq \textsc{exptime} \subseteq \textsc{expspace} \subseteq \ldots$$

3. $\textsc{exp}^i\textsc{space} \neq \textsc{exp}^{i+1}\textsc{space}$ for $i = -1, 0, 1, \ldots$ and $\textsc{exp}^i\textsc{time} \neq \textsc{exp}^{i+1}\textsc{time}$ for $i = 0, 1, 2, \ldots$

Proof

Classical (from the 1970s); recent proofs may be found in (Savage, 1998) or (Jones, 1997). $\square$

6 Simulating Turing machines by read-only functional programs

We now prove Figure 2.1's containments one way: given a problem $A \subseteq \{\texttt{0}, \texttt{1}\}^*$ in a Turing-defined complexity class, we show there is an appropriately restricted functional program that accepts A. The starting point is a Turing machine `tm` accepting A.

6.1 Simulation approach

Section 6.2 shows how to simulate a space-bounded Turing program by a tail-recursive functional program that in essence executes it one step at a time in the forward direction. The forward simulation shows all the space bounds in row 4 of Figure 2.1 except the first. It is not needed here since established in (Jones, 1999).

Section 6.4 shows how to simulate the effect of a time-bounded Turing program by a general recursive functional program that in essence does a backward execution of the Turing program. The backward simulation establishes all the time bounds in row 3 of Figure 2.1.

The main parts of both simulations are first-order, and represent information about the Turing machine's configurations using our functional language, with a temporary extension to natural numbers. At first we simply assume existence of an unbounded built-in abstract data type `Nat` for the natural numbers, and natural operations on them. Once the algorithms are understood, and bounds established for the numbers involved, the natural number type and operations will be eliminated, replaced by encodings into higher-order types realized by a "counting module". This is a set of tail-recursive read-only programs that realize the needed numeric operations without using numbers.

Definition 6.1
A *number program* is a functional program, as defined as in Section 4, but with an additional type `Nat` denoting the natural numbers: $[\![\texttt{Nat}]\!] = \{0, 1, 2, \ldots\}$, and with the operations $+, -, *, \div, \mathrm{mod}, =, \leq$ (syntax `+,-,‘div‘,‘mod‘,==,<=`). Subtraction $x - y$ yields 0 if $x \leq y$, and $\div$ is integer division. Program and data order is defined as before, with the extension that type `Nat` is considered of order 0 as well as `Bool` and `[Bool]`.

The space-time diagram of Figure 6.1 may be of assistance. Each row represents a Turing machine "configuration" as defined above.

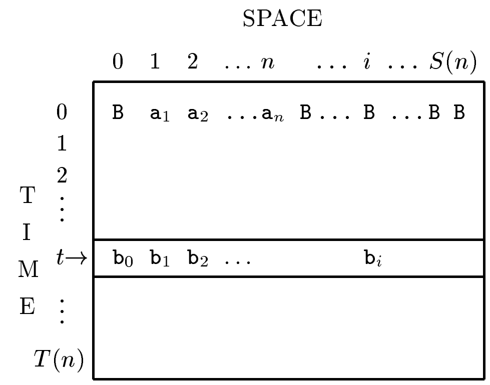

Figure 6.1: Space-time diagram of a Turing computation

6.2 Simulation of a space-bounded Turing machine

A tape containing $\mathtt{b}_0 \ldots \underline{\mathtt{b}_i}\mathtt{b}_{i+1} \ldots \mathtt{b}_m \ldots$ can be represented by a pair of numbers l, r, where

- l is the value of string $\mathtt{b}_0 \ldots \mathtt{b}_i$ *encoded as a base 3 number* (with $\mathtt{b}_0$ as most significant digit, with digit values $0, 1, 2$ for symbols B, 0 and 1, respectively).
- r is the value of string $\ldots \mathtt{b}_m \mathtt{b}_{m-1} \ldots \mathtt{b}_{i+1}$, also as a base 3 number. (Note the order reversal, and the fact that ending blanks do not contribute to r.)

Lemma 6.2
A Turing machine program $\mathtt{tm} = \mathtt{I}_1\ \mathtt{I}_2\ \ldots\ \mathtt{I}_m$ can be simulated by a read-only number program. Further, if tm runs in space $S(n)$ on inputs of length n, and $S(n) \geq n$, then the natural numbers used by $\mathtt{Tr}_{\mathtt{tm}}$ are at largest $3^{S(n)}$.

Proof
The simulation is carried out by the program $\mathtt{Tr}_{\mathtt{tm}}$ of Figure 6.2, whose variables l, r represent the Turing machine's current tape contents as above. Correctness: It is easy to see that the tape representation invariant is maintained whever a computation step is performed.

Main program:

```
runTR:: [Bool] -> Bool
execute_1,...,execute_{m+1}::    (Nat, Nat) -> Bool

runTR as                        = bits2number 0 1 as

bits2number s val []            = execute_1(s, val)
bits2number s val (False:as)    = bits2number (s+1*val) (3*val) as
bits2number s val (True:as)     = bits2number (s+2*val) (3*val) as

execute_1(l,r)                  = SIMULATE_1
...                             ...
execute_m(l,r)                  = SIMULATE_m
execute_{m+1}(l,r)              = if l`mod`3==2 then True else False
```

For $\ell = 1, 2, \ldots, m$, $\mathtt{SIMULATE}_\ell$ is defined as follows, where $\overline{\mathtt{S}} = 0, 1, 2$, respectively, if S = B, 0 or 1:

Form of instruction $\mathtt{I}_\ell$	**Expression SIMULATE$_\ell$**
right	execute$_{\ell+1}$ (l*3+r`mod`3, r`div`3)
left	execute$_{\ell+1}$ (l`div`3,r*3+l`mod`3)
write S	execute$_{\ell+1}$ (3*(l`mod`3)+$\overline{\mathtt{S}}$, r)
if S goto ℓ' else ℓ''	if (l`mod`3 == $\overline{\mathtt{S}}$) then execute$_{\ell'}$(l,r) else execute$_{\ell''}$(l,r)

Figure 6.2: Program $\mathtt{Tr}_{\mathtt{tm}}$*: tail recursive Turing machine simulation*

The Turing machine's initial tape contents, given input string $\mathtt{a}_1 \ldots \mathtt{a}_n \in \{0, 1\}^*$, is $\underline{\mathtt{B}}\mathtt{a}_1 \ldots \mathtt{a}_n \mathtt{B} \ldots$. Program $\mathtt{Tr}_{\mathtt{tm}}$ first constructs the pair (0,r) representing the tape, where number `r` (computed by "`bits2number`") encodes input list `as = [a`$_1$`,...,a`$_n$`]` as described above. Functions `execute`$_1$, etc., are executed, each simulating the corresponding Turing instruction.

Program $\mathtt{Tr_{tm}}$ clearly has data order 0, and is read-only. If program `tm` is a space $S(n)$-bounded Turing machine, then the number of nonblank symbols in any reachable configuration $(\ell, \mathtt{b}_0\mathtt{b}_1 \ldots \underline{\mathtt{b}_i}\, \mathtt{b}_{i+1} \ldots)$ is at most $S(n)$. The tape is encoded as a pair(l, r) of ternary numbers, so $l, r \leq 3^{S(n)}$. □

6.3 Tail recursion for first-order programs

Before arguing that $\mathtt{Tr_{tm}}$ is tail recursive, the term must be defined. This is straightforward for first-order programs; the more subtle higher-order case is treated later in Section 6.7.

The following definitions concern an *occurrence* of one expression `e` within another $\mathtt{e}_0$, for example, the first "`hd x`" within "`if null(hd x) then tl x else hd x`". An occurrence of `e` within $\mathtt{e}_0$ may be identified with a path from the root of $\mathtt{e}_0$'s syntax tree to a subtree containing `e`.

Definition 6.3

1. An occurrence of expression `e` in $\mathtt{e}_0$ is *in tail position in* $\mathtt{e}_0$ if either $\mathtt{e}_0 = \mathtt{e}$, or $\mathtt{e}_0$ has form `if` $\mathtt{e}_1$ `then` $\mathtt{e}_2$ `else` $\mathtt{e}_3$, and the `e` occurrence is in tail position in either $\mathtt{e}_2$ or $\mathtt{e}_3$. This definition is recursive, so in case $\mathtt{e}_0$ is a decision tree, all of its leaves are in tail position.
2. Let a *complete call* be an application $\mathtt{g}\, \mathtt{e}_1 \mathtt{e}_2 \ldots \mathtt{e}_r$ where $r = arity(\mathtt{g})$. A *tail call* is a complete call in tail position. A program `p` is *strictly tail-recursive* if every complete call $\mathtt{g}\, \mathtt{e}_1 \mathtt{e}_2 \ldots \mathtt{e}_r$ is a tail call.
3. A program `p` is *tail recursive* if there is a partial order $\geq$ on `p`'s function names such that for every complete call

$$\mathtt{f\ x1...xm\ =\ ...\ g}\, \mathtt{e}_1 \mathtt{e}_2 \ldots \mathtt{e}_k\ \mathtt{...}$$

   in `p`, either (a) `f>g`, or (b) `f≡g` and the call is a tail call.

Remarks.

A. A strictly tail recursive first-order program may be implemented imperatively (without stack or recursion) by replacing each function call by a `GOTO`, and realizing function parameter passing by assignments to global variables.

B. Condition 3 allows "mutual tail recursion," for instance

```
even x = if x = 0 then True else odd(x-1)
odd x  = if x = 0 then False else even(x-1)
```

by choosing a function name partial order with `even ≡ odd`.

C. A program satisfying condition 3 above can also be implemented imperatively since at run time the call stack depth has a constant depth bound, regardless of input data, so again a static storage layout may be used.

D. This fact implies that any tail recursive program may be translated into strict tail recursive form. The strictness requirement has, however, deleterious effects on program readability (e.g., greatly increasing the numbers of functional arguments), so we use the more general Condition 3 or its higher-order analogue in the rest of the paper.

We will discuss higher-order tail recursive programs in Section 6.7, after some examples have been seen. First, consider tail recursiveness of program $\mathtt{Tr}_{\mathtt{tm}}$ above.

Lemma 6.4
If $A \in \text{SPACE}(S(n))$ and $S(n) \geq n$, then A is accepted by a first-order read-only tail-recursive number program $\mathtt{Tr}_{\mathtt{tm}}$. Further, the natural numbers used by $\mathtt{Tr}_{\mathtt{tm}}$ on length n inputs never exceed $3^{S(n)}$.

Proof
Program $\mathtt{Tr}_{\mathtt{tm}}$ from Lemma 6.2 is tail recursive (in fact, strictly tail recursive). □

6.4 General recursive simulation of a Turing machine

Proposition 6.5
If $A \in \text{TIME}(T(n))$ where $T(n) \geq n$ and $T(n)$ is computable by a first-order read-only number program, then A is accepted by a first-order read-only tail-recursive number program $\mathtt{F}_{\mathtt{tm}}$. Further, the natural numbers used by $\mathtt{F}_{\mathtt{tm}}$ on length n inputs are never exceed $T(n)$.

Proof
Let A be accepted by Turing machine program $\mathtt{tm} = \mathtt{I}_1\ \mathtt{I}_2\ \ldots\ \mathtt{I}_m$ that runs in time $T(n)$. Idea: in the space-time diagram seen before, data propagation is local; the information (scanned symbol, program counter in `tm`, etc.) at time t and tape position i is a mathematical function of the information at time $t-1$ and tape positions $i-1, i, i+1$. This connection can be used to compute the contents of any tape cell at any time, including the final result of the computation.

Given input $\mathtt{as} = \mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n \in \{\mathtt{0},\mathtt{1}\}^*$, question "$\mathtt{as} \in A$?" is decided by computing the final tape cell in the computation of `tm` on `as`, answering "yes" if it is `1`, else "no." The functional program computes, for any time $t \in \{0,1,2,\ldots,T(n)\}$, functions

$$
\begin{array}{lcl}
\mathtt{scan}\ t & = & \text{the scanning position } i \in \{0,1,2,\ldots,T(n)\} \text{ at time } t \\
\mathtt{pgmctr}\ t & = & \text{the label } \ell \text{ of the instruction executed at time } t \\
\mathtt{tape}\ t\ i & = & \text{the tape symbol found at position } i \text{ at time } t
\end{array}
$$

All these are computed by backward simulation of Turing machine program `tm` on input $\mathtt{as} = \mathtt{a}_1\mathtt{a}_2\ldots\mathtt{a}_n \in \{\mathtt{0},\mathtt{1}\}^*$. The simulating functional program $\mathtt{F}_{\mathtt{tm}}$ appears in Figure 6.3.

Remarks on how the simulator works.

1. On input $[\mathtt{a}_1,\ldots,\mathtt{a}_n]$, program $\mathtt{F}_{\mathtt{tm}}$ first computes $t = T(n)$ (via `timebound`).

```
-- Data types                                    (Can be eliminated)
data Symbol   = Zero | One | Blank
type Label    = Lab1 | Lab2 | ...| Labm| Labm+1
-- Function types
run::         Program -> [Bool] -> Bool
scan::        Nat -> Nat
pgmctr::      Nat -> Label
tape::        Nat -> Nat -> Symbol
timebound ::  [Bool] -> Nat                      (Code not given here)
inputtape ::  [Bool] -> Nat -> Symbol
ith ::        [Bool] -> Nat -> Symbol

-- Function definitions
run as = let t = (timebound as) in (tape t (scan t) == One)
where
scan t = if (t == 0) then 0 else let i = scan (t-1) in
   case pgmctr(t-1) of
   Lab1 ->    if I1==`right` then i+1 else if I1==`left` then i-1 else i
   Lab2 ->    if I2==`right` then i+1 else if I2==`left` then i-1 else i
              ...
   Labm  ->   if Im==`right` then i+1 else if Im==`left` then i-1 else i
   Labm+1->          i

tape t i = if (t == 0) then (inputtape as i) else
   if scan (t-1) == i then case pgmctr(t-1) of
        Lab1 ->      if I1 == `write S` then S else tape (t-1) i
        ...
        Labm  ->     if Im  == `write S` then S else tape (t-1) i
        Labm+1->           tape (t-1) i
   else tape (t-1) i

pgmctr t = if (t == 0) then 1 else let i = scan(t-1) in
   case pgmctr(t-1) of
   Lab1 ->    if I1==`if S then ℓ' else ℓ''` then
              (if S == tape (t-1) i then Labℓ' else Labℓ'') else Lab2
   ...
   Labm  ->   if I1==`if S then ℓ' else ℓ''` then
              (if S == tape (t-1) i then Labℓ' else Labℓ'') else Labm+1
   Labm+1->   Labm+1

inputtape as i = if i == 0 then Blank else ith (i-1) as
ith i as =    if (i == 0) then (if head as then One else Zero) else
              if (null as) then Blank else ith (i-1) (tl as)
```

Figure 6.3: Turing machine simulation by general recursive program

2. It computes the position `i` = `scan t` of the symbol scanned at the end of the computation, and finds `tape t i`, the tape symbol scanned at time $T(n)$. If this is 1 then program $\texttt{F}_{\texttt{tm}}$ returns `True`, else `False`.
3. *Explanation of* `scan t` *code.* At start (`t`=0), position 0 is scanned. Otherwise

the position of the scanned tape symbol at time `t` is the same as at time `t-1`, unless the instruction executed at time `t-1` was `right` or `left`.

4. *Explanation of* `tape t i` *code.* At start (`t`=0), the tape is `B` $\texttt{a}_1 \ldots \texttt{a}_n$ `B...B`. Functions `inputtape` and `ith` find the appropriate symbol within this. If `t` > 0, the `i`-th tape symbol at time `t` is the same symbol as at time `t-1`, unless the instruction executed at time `t-1` wrote some symbol `S` over it.
5. *Explanation of* `pgmctr t` *code.* At start (`t`=0), instruction `1` is to be executed. If `t` > 0, the instruction executed at time `t` depends on the instruction $\texttt{I}_\ell$ executed at time `t-1`. If $\texttt{I}_\ell$ is not a test, the next instruction is $\texttt{I}_{\ell+1}$, otherwise the branch taken depends on the scanned symbol. If $\ell = m+1$, then execution has terminated so the final program counter has the value it had at time `t-1`.

Remarks on the form of the simulator. For the sake of readability, program $\texttt{F}_{\texttt{tm}}$ was written with a few violations of the requirement that all parameters must be of order 0 built from `Bool` and `[Bool]`, or natural numbers bounded by $T(n)$ for inputs of length n. We show that the violations are easily overcome.

1. Program $\texttt{F}_{\texttt{tm}}$ uses types `Label` and `Symbol`.
2. $\texttt{F}_{\texttt{tm}}$ has numerous tests such as

```
if I₂=='right' then i+1 else if I₂=='left' then i-1 else i
```

Being finite, the values of `Label` and `Symbol` can be represented using tuples of Boolean values. Further, the Turing program `tm` being simulated is *fixed*, so all the tests on $\texttt{I}_i$ can be eliminated – one can simply replace the line above by `i+1` , or `i-1`, or `i`, in case $\texttt{I}_2 = \texttt{right}$, or $\texttt{I}_2 = \texttt{left}$, or neither (respectively). □

6.5 Counting

Our current goal is to prove Figure 2.1's containments one way: given problem $A \subseteq \{0,1\}^*$ in a Turing-defined complexity class, we show that there exists an appropriately restricted functional program that accepts A, i.e., computes its characteristic function. To complete the proofs we show that read-only functional programs with data order k can count as high as $\exp_{k+1}(a \log n)$ when given an input of length n, using neither "cons" nor built-in arithmetic. Further, the arithmetic computations needed in the Turing machine simulations just seen can be done.

Definition 6.6
Let $f : I\!N \to I\!N^+$. An *f-counting module* is a tuple $\mathcal{C} = (\texttt{Nat}, \mathit{decode}, \texttt{seed}, \texttt{ztest}, \texttt{pred})$ containing

1. A data type `Nat`, built from `Bool` and `[Bool]`.
2. A function *decode* : $[\![\texttt{Nat}]\!] \to I\!N$ (a mathematical function, not a program).
3. Terminating programs

```
seed::[Bool]->Nat,   pred::Nat->Nat,   ztest::Nat->Bool
```

such that for any Boolean list $\texttt{as} = [\texttt{a}_1, \ldots \texttt{a}_n]$ with $n > 0$, and any expression `exp` built by applying `pred` and `ztest` to `seed`, the following hold.[5]

[5] For brevity we write $[\![\texttt{exp}]\!]\texttt{as}$ for the result of evaluating `exp`, instead of the longer "v, where $\texttt{p}, [\texttt{as} \mapsto [\texttt{a}_1, \ldots \texttt{a}_n]] \vdash \texttt{exp} \to v$".

$$decode([\![\texttt{seed}]\!]\texttt{as}) = f(|\texttt{as}|) - 1$$

$$[\![\texttt{ztest exp}]\!]\texttt{as} = \begin{cases} \texttt{False} & \text{if } decode([\![\texttt{exp}]\!]\texttt{as}) > 0 \\ \texttt{True} & \text{if } decode([\![\texttt{exp}]\!]\texttt{as}) = 0 \end{cases}$$

$$decode([\![\texttt{pred(exp)}]\!]\texttt{as}) = \begin{cases} decode([\![\texttt{exp}]\!]\texttt{as})\text{-1} & \text{if } decode([\![\texttt{exp}]\!]\texttt{as}) > 0 \\ 0 & \text{if } decode([\![\texttt{exp}]\!]\texttt{as}) = 0 \end{cases}$$

The first condition is that the seed function yields a representation of $f(n) - 1$, if given an arbitrary $[\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_n]$ of length $n > 0$. The remaining conditions are that `ztest` represents the zero test on natural numbers, and `pred` represents the predecessor. Lemma 6.11 will show that the other functions used in the Turing simulations can also be computed by read-only programs.

Lemma 6.7
There is a $\lambda n \,.\, n+1$-counting module $\mathcal{C}$ with data-order 0, read-only programs.

Proof
Represent a number between 0 and n by the length of a suffix `bs` of input `as` = $[\texttt{a}_1, \texttt{a}_2 \ldots, \texttt{a}_n]$. Choose `Nat = [Bool]`, and define $decode(\texttt{bs}) = |\texttt{bs}|$. This ranges from 0 to n ($n = n+1-1$) as `bs` ranges over suffixes of `as`. The following program fragment defines `seed`, `ztest` and `pred`.

```
type Nat = [Bool];
seed as  = as

ztest([])  =  True
ztest(bs)  =  False

pred([])   = []
pred(b:bs)   bs
```

The conditions above are satisfied with $\mathcal{C} = ([\texttt{Bool}], decode, \texttt{seed}, \texttt{ztest}, \texttt{pred})$. $\square$

Lemma 6.8
If $\mathcal{C}_1$, $\mathcal{C}_2$ are f- and g-counting modules (respectively) then $f+g$- and $f \cdot g$-counting modules (resp.) $\mathcal{C}^+$, $\mathcal{C}^*$ can be built from them. Further, if $\mathcal{C}_1$, $\mathcal{C}_2$ consist of data-order k, read-only programs, then so do $\mathcal{C}^+$ and $\mathcal{C}^*$.

Proof
Let $\mathcal{C}_i = (\texttt{Nat}i, decode_i, \texttt{seed}i, \texttt{ztest}i, \texttt{pred}i)$ for $i = 1, 2$. For $\mathcal{C}^*$, define the needed functions by the code:

```
type Nat* = (Nat_1,Nat_2,Nat_2);
seed* as  = (seed1 as, seed2 as, seed1 as)

ztest* (r,s,m) = (ztest1 r) and (ztest s)

pred* (r,s,m) = if (ztest2 s)
                then if (ztest1 r) then (r,s,m) else ((pred1 r),m,m)
                else (r, (pred2 s), m)
```

with

$$decode^*(r,s,m) = (m+1)\cdot decode_1(r) + decode_2(s)$$

Explanation: think of (r,s,m) as a 2-digit number rs where r ranges from 0 to $f(n)-1$, and s ranges from 0 to $m = g(n)-1$.

For $\mathcal{C}^+$, define the needed functions by the code:

```
type Nat+ = (Nat_1,Nat_2,Bool);
seed+ as  = (seed1 as, seed2 as, True)

ztest+ (r,s,flag) = (ztest1 r) and (ztest s) and not flag

pred+ (r,s,flag) = if flag
                    then if (ztest2 s) then (r, s, False)
                                       else (r, (pred2 s), flag)
                    else if (ztest1 r) then (r, s, flag)
                                       else (pred1 r), s, flag)
```

with

$$\begin{aligned} decode^+(r,s,\texttt{True}) &= decode_1(r) \\ decode^+(r,s,\texttt{False}) &= decode_1(r) + decode_1(s) + 1 \end{aligned}$$

and

Explanation: count down from $(f(n)-1, g(n)-1, \texttt{True})$ to $(0, g(n)-1, \texttt{True})$ in the first component, then to $(0, g(n)-1, \texttt{False})$, and so down to $(0,0,\texttt{False})$ in the second component, for a total of $f(n)+g(n)$ steps. $\square$

Corollary 6.9

For any positive polynomial $\pi : \mathbb{N} \to \mathbb{N}^+$, there is a π-counting module $\mathcal{C}$ with data-order 0, read-only programs.

Lemma 6.10

For any $k \geq 0$ and $a > 0$ there exists a $\lambda n \,.\, \exp_{k+1}(a \log n)$-counting module $\mathcal{C}$ defined by read-only programs with data order k.

Proof

The case $k = 0$ was just shown (since $2^{a \log n} = n^a$). Let $f(n) = \exp_{k+1}(a \log n)$, and suppose inductively that an f-counting module

$$\mathcal{C}_k = (\texttt{Nat}_k, decode_k, \texttt{seed}_k, \texttt{ztest}_k, \texttt{pred}_k)$$

with read-only programs is given, representing numbers up to $f(n)-1$ when $\texttt{seed}_k$ is given a length-n boolean list as parameter. We need to represent any number between 0 and $2^{f(n)} - 1 = \exp_{k+1}(a \log n) - 1$. Thinking of such a number as a bit string $\texttt{b}_{f(n)-1} \ldots \texttt{b}_1 \texttt{b}_0$ of length $f(n)$, it can be represented by a function from bit positions into booleans, i.e., a function $g : \texttt{Nat}_k \to \{\texttt{True}, \texttt{False}\}$. We thus construct a $\lambda n \,.\, 2^{f(n)}$-counting module

$$\mathcal{C}_{k+1} = (\texttt{Nat}_{k+1}, decode_{k+1}, \texttt{seed}_{k+1}, \texttt{ztest}_{k+1}, \texttt{pred}_{k+1})$$

where $\texttt{Nat}_{k+1}$ = $(\texttt{Nat}_k\texttt{->Bool,Nat}_k)$. In a value `(g, m)` of type $[\![\texttt{Nat}_{k+1}]\!]$, `g` is a function defining a bit string, and m (never changed) is the bit string's length. Given the "bitwise" view, function $decode_{k+1} : \mathtt{Nat}_{k+1} \to I\!N$ is naturally defined by:

$$decode_{k+1}(\mathtt{g}, \mathtt{m}) = g(0) + 2g(1) + \ldots + 2^{m-1}g(m-1)$$

where $m = decode_k(\mathtt{m})$ is the length of the bit string and $g(i) = decode_k(\mathtt{g\ i})$, where i is the coding in $\mathtt{Nat}_k$ of natural number i. The maximum representable number is thus $2^{f(n)} - 1 = decode_{k+1}(\mathtt{big}, \mathtt{m})$ where big i = True for all i, and m = $\mathtt{seed}_k$ as represents the largest number $f(n) - 1$ in $\mathtt{Nat}_k$.

This line of thought leads to the program code of Figure 6.4. The $\mathtt{seed}_{k+1}$ code constructs a list all of whose bits are 1 (True). The $\mathtt{ztest}_{k+1}$ code tests its argument to see that every bit is 0, scanning from the left until the right end is reached (found by counting m down to zero). The $\mathtt{pred}_{k+1}$(g,m) code returns ($\mathtt{onetoright}_{k+1}$,m) where bit i of $\mathtt{onetoright}_{k+1}$ is found by scanning g's bits from position i. If any among g(i-1),..., g(1), g(0) was True, then bit i of $\mathtt{onetoright}_{k+1}$ is unchanged; but if all were False, then bit i of $\mathtt{onetoright}_{k+1}$ is changed to its negation.

```
type Nat_{k+1} = (Nat_k->Bool, Nat_k)

seed_{k+1} :: [Bool] -> Nat_{k+1}
seed_{k+1} as = (big_{k+1}, seed_k as)

big_{k+1} :: Nat_k->Bool
big_{k+1} i = True

ztest_{k+1} :: Nat_{k+1} -> Bool
ztest_{k+1} (g,m) = isz_{k+1} g m

isz_{k+1} g i =   if (g i) then False else
                  if (ztest_k i) then True else isz_{k+1} g (pred_k i)

pred_{k+1} :: Nat_{k+1} -> Nat_{k+1}
pred_{k+1} (g,m)  =     if ztest_{k+1}(g,m) then (g,m) else (bits_{k+1} g, m)

bits_{k+1} g i    =     if onetoright_{k+1} g i then (g i) else not(g i)

onetoright_{k+1} g j     =     if (ztest_k j) then False else
                               let prev = pred_k j in
                               g prev || onetoright_{k+1} (g prev)
```

Figure 6.4: Counting module for level $k+1$

□

Lemma 6.11

Given a $\lambda n\,.\,\exp_{k+1}(a \log n)$-counting module $\mathcal{C}_k$, there exist read-only programs that define the following natural number functions and relations, where $c > 1$ is a natural number, assuming that no argument or result exceeds $\exp_{k+1}(a \log n)$.

- $\lambda x\,.\,0$, $\lambda x\,.\,x+1$, $\lambda x\,.\,x \cdot c$, $\lambda x\,.\,x \div c$, $\lambda x\,.\,x \bmod c$

- Relations $\leq$ and $=$

Proof

Straightforward programming using the techniques seen above. □

Theorem 6.12

$\text{EXP}^k\text{TIME} \subseteq \text{RO}^k$ for any $k \geq 0$.

Proof

Let A be accepted by Turing machine program `tm` that runs in time $\exp_k(a \log n)$. By Proposition 6.5, A is accepted by a program $\texttt{F}_{\texttt{tm}}$ using natural numbers bounded by $\exp_k(a \log n)$, and $\texttt{F}_{\texttt{tm}}$ is a data order 0 read-only functional number program. By Lemma 6.10 there exists a $\lambda n \,.\, \exp_{k+1}(a \log n)$-counting module $\mathcal{C}$ defined by read-only programs with data order k. By Lemma 6.11, the functions used in $\texttt{F}_{\texttt{tm}}$'s Turing simulation can also be computed by order k read-only programs. Combining these three ($\texttt{F}_{\texttt{tm}}$, the function definitions of $\mathcal{C}$ and those from Lemma 6.11) yields an order k read-only program that simulates `tm`. □

6.6 *Intensional versus extensional polynomial time*

Theorem 6.12 shows that read-only programs are quite expressive: even order 0 programs can decide all problems that lie in PTIME. However, this *does not imply* that these programs themselves run in polynomial time. The point is that the order k programs were defined by restrictions on program syntax, and not on their resource consumption.

While fully satisfying from a theoretical viewpoint, Theorem 6.12 is thereby much less satisfying from a programmer's perspective. The reason is that it is easy to construct data-order 0 read-only programs that *run in exponential time* (regardless of whether call-by-value or lazy semantics are used), for instance the one in Section 2.3. Thus the very same programs that characterize the time complexity class PTIME are running in times that lie outside that class.

There is no logical or mathematical conflict here. The proofs are by two simulations. The one direction (just seen) shows how, given a polynomial-time-bounded Turing machine, to build from it a read-only program that accepts the same inputs. However, it is easily verified that the constructed program will nearly always take exponential time to run! This is due to repeated solution of the same subproblems. Concretely, functions `scan`, `tape` and `pgmctr` of the Turing machine simulator of Proposition 6.5 call one another recursively, e.g., `scan t` calls both `scan(t-1)` and `pgmctr(t-1)`, both of these call both of `scan(t-2)` and `pgmctr(t-2)`, etc.

In the other direction we will show how, given an arbitrary read-only data-order 0 program (regardless of its running time), to build a polynomial-time-bounded Turing machine that accepts the same inputs. The construction uses a "memoizing" simulation technique quite different from normal functional program execution—particularly interesting since it in fact runs faster than the program it is simulating.

6.7 Tail recursion for higher-order programs

On function arities. Operationally, a higher-order value is a closure $\langle \mathtt{f}, v_1 \dots v_i \rangle$ where $i < \mathit{arity}(\mathtt{f})$. Such values are obtained by incomplete applications, e.g., a call `f(x+1)` where `f` has arity 2. Tracing control flow is complicated by the fact that a higher-order program may contain an application such as `(u v)`, where `u` is a function-valued parameter, as for instance in `twice u v = u (u v)`.

The problem of finding a satisfactory syntactic formulation of higher-order tail recursiveness is decidedly nontrivial, for instance two definitions, both reasonable from a lambda calculus viewpoint, are given in (Kfoury, 1997; Kfoury, 1999), and then proven incomparable. Rather than enter into the issues raised by those papers, we take an operational approach sufficient for the programs appearing in this paper.

Definition 6.13
Program `p` is *tail recursive* if there is a partial order $\geq$ on the function names such that for any application $\mathtt{f}\,\mathtt{x}_1 \dots \mathtt{x}_m = \dots \mathtt{e}_1\ \mathtt{e}_2 \dots$ such that $\mathtt{e}_1$ can evaluate to a closure $\langle \mathtt{g}, v_1 \dots v_{arity(\mathtt{g})-1} \rangle$, either (a) $\mathtt{f} > \mathtt{g}$, or (b) $\mathtt{f} \equiv \mathtt{g}$ and the call $(\mathtt{e}_1\ \mathtt{e}_2)$ is in tail position.

The definition is semantic, referring to all program executions, and so undecidable in general. (Abstract interpretation can, however, safely approximate it.)

Lemma 6.14
The programs in the $\lambda n\,.\,\exp_{k+1}(a \log n)$-counting modules $\mathcal{C}_k$ shown to exist in Lemma 6.11 are all tail-recursive.

Proof
First, $\mathcal{C}_0$ has no recursion at all. Second, note that functions in $\mathcal{C}_{k+1}$ call only functions in $\mathcal{C}_{k+1}$ and $\mathcal{C}_k$, or the function parameter `g`. The programs in $\mathcal{C}_{k+1}$ contain definitions of

$$\mathtt{seed}_{k+1}, \mathtt{big}_{k+1}, \mathtt{ztest}_{k+1}, \mathtt{isz}_{k+1}, \mathtt{pred}_{k+1}, \mathtt{bits}_{k+1}, \mathtt{onetoright}_{k+1}$$

Order the functions just listed linearly, with $\mathtt{seed}_{k+1}$ greatest and $\mathtt{onetoright}_{k+1}$ least; and order the functions in $\mathcal{C}_k, \dots, \mathcal{C}_0$ in the same way, but all less than $\mathtt{onetoright}_{k+1}$. Inspecting the program code in the proof of Lemma 6.10 it is easy to see that

1. The only directly recursive calls are from $\mathtt{isz}_{k+1}$ and $\mathtt{onetoright}_{k+1}$ to themselves; and these calls are in tail position.
2. The remaining calls are all to functions lower in the order, or to parameter `g`.
3. `g`, of type $\mathtt{Nat}_k$`->Bool`, is called only from $\mathtt{isz}_{k+1}$, $\mathtt{bits}_{k+1}$, and $\mathtt{onetoright}_{k+1}$. Thus the only functions callable via `g` are ones in $\mathcal{C}_k$. These are lower in the function name order, so no recursion (tail or otherwise) can ensue.

□

Theorem 6.15
$\mathrm{EXP}^{k-1}\mathrm{SPACE} \subseteq \mathrm{ROTR}^k$ for any $k \geq 0$.

Proof

First, if $k = 0$, then $\text{EXP}^{-1}\text{SPACE} = \text{LOGSPACE} \subseteq \text{ROTR}^0$ as proven in (Jones, 1999). Now suppose $k > 0$ and $A \in \text{SPACE}(\exp_k(a \log n))$, so A is accepted by a Turing machine running in space $\exp_k(a \log n)$. Without loss of generality $a \geq 1$. Now $k > 0$ implies $\exp_k(a \log n)) \geq n$, so by Lemma 6.4, A is accepted by a tail-recursive functional program $\texttt{Tr}_{\texttt{tm}}$ with natural numbers bounded by $3^{\exp_k(a \log n))}$ for inputs of length n. It is easy to verify that $3^{\exp_k(a \log n))} \leq \exp_{k+1}(2a \log n))$.

A data order k read-only functional program that accepts A can thus be obtained by combining this program $\texttt{Tr}_{\texttt{tm}}$ with those shown to exist earlier. Lemma 6.10 shows existence of a k-order $\lambda n \,.\, \exp_{k+1}(2a \log n))$-counting module, and Section 6.5 shows it can be extended to compute the functions on numbers needed by $\texttt{Tr}_{\texttt{tm}}$. By Lemma 6.14, the counting module programs are (higher-order) tail recursive. □

7 Simulating read-only functional programs in bounded time or space

The remaining task is to prove that read-only functional programs can be simulated by Turing machines in sufficiently small time or space. Unfortunately, this seems impossible by direct step-by-step simulation, due to the problem of exponential running times even for $k = 0$ pointed out in sections 2.3 and 6.6.

We resolve this problem by defining two new semantics, one to minimize time, the other to minimize space. The desired results follow by analyzing the time and memory needed to execute the alternative semantics. Key points include the fact that the semantics are deterministic and allow left-to-right satisfaction of judgements in a rule; plus an observation on the size of the sets of data values computable by order k programs on inputs of length n.

The first semantics accumulates during execution a "cache" recording, for all function calls that have been completed, both the input parameter values and the function's return value. This data structure allows repeated computations to be avoided completely, making in some cases an exponential improvement in running times.

The second semantics uses a stack of environments, pushed on function call and popped on return. Further, it uses the "tail recursion hack" to reduce memory usage: the environment on the stack top is overwritten whenever a function call in tail position is executed.

The two semantics are claimed to be equivalent to the original semantics on all programs. Their equivalence with the original are left as exercises for the reader. Proof is straightforward since both are based on familiar implementation concepts.

7.1 Some observations about the semantics

We now introduce some concepts that will be useful for establishing time and space bounds. These ideas, concerning the problem of *executing* an operational semantics, have roots in both logic programming and attribute grammars.

Input-output moding. This paper's three semantics all have a common principal judgement form, namely $[\![\mathtt{p}]\!]v = w$. Its usage is that, given $\mathtt{p}$ and v as "inputs", one wishes to find an "output" value w such that $[\![\mathtt{p}]\!]v = w$. We generalize this idea to assign a *mode* in $\{in, out\}$ to each field i of each judgement form $J(\_, \ldots, \_)$ in a given semantics. For the judgement forms in Figure 4.1 the natural modings are:

$$\begin{array}{llllll}
 & & [\![\mathtt{p}^{in}]\!]v^{in} & = & w^{out} & \text{for programs} \\
\mathtt{p}^{in}, env^{in} & \vdash & \mathtt{e}^{in} & \to & w^{out} & \text{for expressions} \\
 & \vdash^{null} & v^{in} & \to & w^{out} & \text{auxiliary for } \mathtt{null} \\
\mathtt{p}^{in}, env^{in} & \vdash^{if} & \mathtt{e}_2^{in}, \mathtt{e}_3^{in}, v^{in} & \to & w^{out} & \text{auxiliary for } \mathtt{if} \\
\mathtt{p}^{in} & \vdash^{call} & u^{in}, v^{in} & \to & w^{out} & \text{auxiliary for } \mathtt{call}
\end{array}$$

Definition 7.1

Consider inference rule: $\dfrac{J_1(\_, \ldots, \_) \quad J_2(\_, \ldots, \_) \quad \ldots \quad J_n(\_, \ldots, \_)}{J(\_, \ldots, \_)}$(Side conditions)

- The *binding* fields in this rule are the input fields of J and the output fields of $J_1, \ldots, J_n$.
- The *computed* fields in this rule are the remainder: the output fields of J and the input fields of $J_1, \ldots, J_n$.

Definition 7.2

Inference rule $\dfrac{J_1(\_, \ldots, \_) \quad J_2(\_, \ldots, \_) \quad \ldots \quad J_n(\_, \ldots, \_)}{J(\_, \ldots, \_)}(\ldots)$ is *well-moded* if

- Each binding field is a pattern built from variables, values and constructors.
- Each computed field, and each side condition, is an expression built from values, operators and variables defined in binding fields.

Definition 7.3

Rule $\dfrac{J_1(\_, \ldots, \_) \quad J_2(\_, \ldots, \_) \quad \ldots \quad J_n(\_, \ldots, \_)}{J(\_, \ldots, \_)}(\ldots)$ is *LR* if it is well-moded, and

1. For each $i = 1, 2, \ldots, n$, the input fields of J_i are computable from the binding fields of J and $J_1, \ldots, J_{i-1}$.
2. The output fields of J are computable from the binding fields of J and $J_1, \ldots, J_n$.

Remarks:

1. The output fields of a well-moded axiom are computable from its input fields.
2. If a rule has premises, conditions (1) and (2) above imply they may be satisfied in a left-to-right order.

Lemma 7.4

Each rule in Figure 4.1 is LR.

Definition 7.5

Given an inference rule R, let R' be the result of replacing every computed field in R by a new variable. A rule set is *locally deterministic* if for any two rules R_1, R_2, R'_1 and R'_2 have no common instance[6].

Definition 7.6
Consider a "principal" n-ary judgement $J_{pr}(\_, \ldots, \_)$ and any m-ary judgement $J(\_, \ldots, \_)$. Field i of J is called *structural* in field j of J_{pr} if for every proof tree T with $J_{pr}(v_1, \ldots, v_n)$ as root, in every occurrence of $J(w_1, \ldots, w_m)$ in T, field w_i is a substructure of v_j.

Lemma 7.7
The semantics of Figure 4.1 is locally deterministic. Further, field e in judgement $\mathtt{p}, env \vdash \mathtt{e} \rightarrow v$ is structural in field p of principal judgement $[\![\mathtt{p}]\!]v = w$.

Proof
Inspection of the rules reveals that a) no two rules R have unifiable binding fields in the conclusion, except for the $\vdash^{call}$ rules; b) the side conditions in the $\vdash^{call}$ rules are disjoint. Structurality of e in p follows by induction over proof trees, since field p is never changed in a rule, and every e occurring in a field or a closure is either the right side of a function definition from p, or a substructure of another e field. □

Lemma 7.8
If a rule set is LR and locally deterministic, then for any values of the principal judgement's input fields there exists at most one proof tree T with those input fields at the root. Further, T can be constructed systematically (bottom-up, left-right without backtracking), starting with a root with unfilled output fields.

The basis for the remaining proofs is an analysis of the time and space to do this for the two semantics to be presented.

7.2 An observation on cardinality

The key point for efficient simulation is the fact that during the computation of $[\![\mathtt{p}]\!][\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]$, all values of type [Bool] must be suffixes $[\mathtt{a}_i, \ldots, \mathtt{a}_n]$ of the input for $1 \leq i \leq n+1$, and thus can assume at most $n+1$ different values. This in turn bounds the cardinality of the set of order k values of type τ by $\exp_{k+1}(a \log n))$, for some a independent of n.

The following makes this precise, letting $Value^{\tau}([\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]) \subseteq Value$ be the set of all order k values of type τ that can be built while computing $[\![\mathtt{p}]\!][\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]$. Note that the last line below estimates the set of *closures* of type $\tau \rightarrow \tau'$, and not the set of *functions* of that type (there may be more closures than functions, since different expressions may compute the same function).

Definition 7.9

[6] Meaning: there is no substitution θ : *Variables* $\rightarrow$ *Values* such that $\theta R'_1 = \theta R'_2$, and the side conditions of R'_1, R'_2 are satisfied.

Let p be a read-only program with data order k, let $\texttt{as} = [\texttt{a}_1, \ldots, \texttt{a}_n] \in \{0,1\}^*$ and let τ be a type. Define value set $\mathit{Value}^\tau(\texttt{as}) \subseteq \mathit{Value}$ by the following equations.

$$\begin{array}{lll}
\mathit{Value}^{\texttt{Bool}}(\texttt{as}) & = \{\ 1, 0\ \} & \\
\mathit{Value}^{[\texttt{Bool}]}(\texttt{as}) & = \{\ [\texttt{a}_i, \ldots, \texttt{a}_n] & \mid\ 1 \le i \le n+1\} \\
\mathit{Value}^{(\tau,\tau')}(\texttt{as}) & = \{\ (v, v') & \mid\ v \in \mathit{Value}^\tau(\texttt{as}), v' \in \mathit{Value}^{\tau'}(\texttt{as})\} \\
\mathit{Value}^{\tau\to\tau'}(\texttt{as}) & = \{\ \langle \texttt{f}, v_1 \ldots v_i > & \mid\ \texttt{f}\ \texttt{x}_1^{\tau_1}\texttt{x}_2^{\tau_2}\ldots\texttt{x}_m^{\tau_m}\ =\ \texttt{e}^{\tau_{m+1}}, 0 \le i < m \text{ and} \\
 & & \quad v_1 \in \mathit{Value}^{\tau_1}(\texttt{as}), \ldots, v_i \in \mathit{Value}^{\tau_i}(\texttt{as}))\ \text{, and} \\
 & & \quad \tau = \tau_{i+1} \text{ and } \tau' = \tau_{i+2} \to \ldots \to \tau_m \to \tau_{m+1}\}
\end{array}$$

Lemma 7.10
Let p be a read-only program with data order k, and $\texttt{as} = [\texttt{a}_1, \ldots, \texttt{a}_n] \in \{0,1\}^*$. Let T be a computation tree by the rules of Figure 4.1 proving $[\![\texttt{p}]\!]\ \texttt{as} = \texttt{a}$, and let $\texttt{p}, \mathit{env} \vdash \texttt{e} \to v$ be a T node. If e has type τ then $v \in \mathit{Value}^\tau(\texttt{as})$; and if variable x of type τ' is free in e, then $\mathit{env}(\texttt{x}) \in \mathit{Value}^{\tau'}(\texttt{as})$.

Proof
An easy induction to check that each inference rule preserves the property. □

The following frequently used property asserts that the set of bounds $\mathit{exp}_k(a \log n)$ is "closed under product," if a is allowed to vary.

Lemma 7.11
For any a, b and $k \ge 1$, for all n

$$\mathit{exp}_k(a \log n) \cdot \mathit{exp}_k(b \log n) \le \mathit{exp}_k((a+b) \log n)$$

Lemma 7.12
Let p be a read-only program with data order k. For any type τ in p, there is an a such that $|\mathit{Value}^\tau(\texttt{as})| \le \exp_{k+1}(a \log n))$ for any $\texttt{as} = [\texttt{a}_1, \ldots, \texttt{a}_n]$, where $|\mathit{Value}^\tau(\texttt{as})|$ is the cardinality of $\mathit{Value}^\tau(\texttt{as})$.

Proof
Proof is by induction, first on k and then on the structure of τ. Basis $k = 0$: immediate if τ is Bool. If $\tau = \texttt{[Bool]}$ we have $|\mathit{Value}^\tau(\texttt{as})| = n+1 \le \exp_1(2 \log n))$. Products (τ, τ') are straightforward for any $k \ge 0$, by Lemma 7.11.

Induction on k: assume true for k, and consider $\mathit{Value}^{\tau\to\tau'}(\texttt{as})$ with $k+1 = \text{order}(\tau \to \tau')$. Now consider a function definition of form

$$\texttt{f}\ \texttt{x}_1^{\tau_1}\texttt{x}_2^{\tau_2}\ldots\texttt{x}_m^{\tau_m}\ =\ \texttt{e}^{\tau_{m+1}}$$

where $\tau = \tau_{i+1}$ and $\tau' = \tau_{i+1} \to \ldots \to \tau_m \to \tau$. The number of function closures of form $\langle \texttt{f}, v_1 \ldots v_i \rangle$ is at most $\Pi_{j=1}^i\ |\mathit{Value}^{\tau_j}(\texttt{as})|$. Each τ_j has order at most k, so by induction on k, $|\mathit{Value}^{\tau_j}(\texttt{as})| \le \exp_{k+1}(a_j \log n))$ for constants $a_1, \ldots, a_i$. Consequently

$$\Pi_{j=1}^i\ |\mathit{Value}^{\tau_j}(\texttt{as})| \le \exp_{k+1}((a_1 + \ldots + a_i) \log n))$$

bounds the number of closures (Lemma 7.11 again). Finally, the value of m and the number of function definitions depends only on program p and not on the input length n, so the total size of $Value^{\tau\to\tau'}(\mathtt{as})$ is bounded by a finite sum of numbers of form $\exp_{k+1}(a'\log n))$, and so is bounded by $\exp_{k+1}(a\log n))$ for some a. $\square$

$$\frac{}{\mathtt{p}, env, c \vdash \mathtt{x} \to env(\mathtt{x}), c} \qquad \frac{}{\mathtt{p}, env, c \vdash \mathtt{f} \to \langle \mathtt{f}, \varepsilon\rangle, c}$$

Function call

$$\frac{\mathtt{p}, c, env \vdash \mathtt{e}_1 \to u, c' \quad \mathtt{p}, c', env \vdash \mathtt{e}_2 \to v, c'' \quad \mathtt{p}, c'' \vdash^{call} u, v \to w, c'''}{\mathtt{p}, c, env \vdash \mathtt{e}_1\mathtt{e}_2 \to w, c'''}$$

$$\frac{}{\mathtt{p}, c \vdash^{call} \langle \mathtt{f}, v_1 \ldots v_{i-1}\rangle, v_i \to \langle \mathtt{f}, v_1 \ldots v_{i-1}v_i\rangle, c} \quad (\text{If } i < arity(\mathtt{f}))$$

$$\frac{\mathtt{p}, c, [\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_m \mapsto v_m] \vdash \mathtt{e}^{\mathtt{f}} \to w, c'}{\mathtt{p}, c \vdash^{call} \langle \mathtt{f}, v_1 \ldots v_{m-1}\rangle, v_m \to w, c' \cup \{(\mathtt{f}, v_1 \ldots v_m, w)\}} \quad \begin{array}{l}(\text{If } \mathtt{f}\ \mathtt{x}_1\mathtt{x}_2 \ldots \mathtt{x}_m = \mathtt{e}^{\mathtt{f}} \text{ in } \mathtt{p} \\ \text{ and } (\mathtt{f}, v_1 \ldots v_m, \_) \notin c)\end{array}$$

$$\frac{}{\mathtt{p}, c \vdash^{call} \langle \mathtt{f}, v_1 \ldots v_{m-1}\rangle, v_m \to w, c} \quad \begin{array}{l}(\text{If } \mathtt{f}\ \mathtt{x}_1\mathtt{x}_2 \ldots \mathtt{x}_m = \mathtt{e}^{\mathtt{f}} \text{ is in } \mathtt{p} \text{ and} \\ (\mathtt{f}, v_1 \ldots v_m, w) \in c\ )\end{array}$$

Program execution:

$$\frac{\mathtt{p}, [\mathtt{x} \mapsto [\mathtt{a}_1, \ldots, \mathtt{a}_m]], \emptyset \vdash \mathtt{e}^{\mathtt{f}} \to \mathtt{a}, c}{[\![\mathtt{p}]\!]\ [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] = \mathtt{a}} \quad (\text{If } \mathtt{p} \text{ begins with } \mathtt{f}\ \mathtt{x} = \mathtt{e}^{\mathtt{f}})$$

Figure 7.1: Semantics for time-economical expression evaluation

7.3 A time-economical caching semantics

This alternative semantics is essentially the original one, augmented with a *cache* defined by

$$c \in Cache = \mathcal{P}(\text{FcnName} \times Value^* \times Value)$$

The idea is to keep a record of all values already calculated in the current computation; and to save time in case a function is called with arguments that have been seen before. Concretely, if a call to f is encountered with complete argument sequence $v_1 \ldots v_m$, and if $(\mathtt{f}, v_1 \ldots v_m, w) \in c$, then f has been called before with these arguments, and yielded w as result. Thus no recomputation is needed, as value w can be returned at once. If c contains no such triple, then f's body $\mathtt{e}^{\mathtt{f}}$ is evaluated. When it returns with some value w a new triple $(\mathtt{f}, v_1 \ldots v_m, w) \in c$ is added to the cache.

Figure 7.1 only shows the modified rules for variables and function application; extension to the others is routine. The judgement form for evaluating expressions is $\mathtt{p}, env, c \vdash \mathtt{e} \to v, c'$. It signifies that expression e is evaluated under environment

env and cache c, yielding value v and cache c', which perhaps extends c with some newly computed argument-value pairs. The judgement form for running programs is $[\![\mathtt{p}]\!]\ [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] = \mathtt{a}$ as before.

Theorem 7.13
Let $\mathtt{p}$ be any program and $\mathtt{as}$ an input. There exists a computation tree T proving $[\![\mathtt{p}]\!]\ \mathtt{as} = \mathtt{a}$ by the inference rules of Section 4 if and only if there exists a computation tree T_{cache} proving $[\![\mathtt{p}]\!]\ \mathtt{as} = \mathtt{a}$ by the inference rules of Figure 7.1.

Proof
Omitted for brevity, since based on standard implementation concepts. □

Lemma 7.14
Let $\mathtt{p}$ be a read-only program with data order k. Then the size of a computation tree T_{cache} proving $[\![\mathtt{p}]\!]\ [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] = \mathtt{a}$ by the inference rules of Figure 7.1 is bounded by $\exp_{k+1}(a \log n))$, for a constant a independent of n.

Proof
An observation: the cache is nondecreasing, i.e., if node $\mathtt{p}, env, c \vdash \mathtt{e} \rightarrow v, c'$ is in T_{cache} then $c \subseteq c'$. By the LR property the nodes in T_{cache} can be linearly ordered, with nondecreasing caches. By determinism, no two are identical (else evaluation would not terminate.) Consequently the number of nodes $\mathtt{p}, env, c \vdash \mathtt{e} \rightarrow v, c'$ is a bound on the size of T_{cache}.

Let $\mathtt{as} = [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n]$. Cache entries are of form $(\mathtt{f}, v_1 \ldots v_m, w)$ where $\mathtt{e}^{\mathtt{f}}$ appears in $\mathtt{p}$ and $v_1, \ldots, v_m, w$ are computed values. The values v_i, w are in $Value^{\tau}(\mathtt{as})$ for their appropriate types τ, by Lemma 7.10 (which also applies to the new semantics). By Lemma 7.12, the cardinality of $Value^{\tau}(\mathtt{as})$ is bounded by $\exp_{k+1}(a \log n))$ for some a and all n. Thus the number of cache triples is also bounded by $\exp_{k+1}(a' \log n))$ for some a'.

The number of nodes in T_{cache} is at most the product of the number of cache triples, the number of environments, the number of expressions in $\mathtt{p}$, and the number of values. By earlier lemmas these are all bounded by $\exp_{k+1}(a \log n))$ for various a's and all n, so the result follows by Lemma 7.11. □

A cache-based algorithm to compute $[\![\mathtt{p}]\!]\ \mathtt{as}$

Lemma 7.15
The inference rules of Figure 7.1 are locally deterministic, and LR with respect to moding $\mathtt{p}^{in}, env^{in}, c^{in} \vdash \mathtt{e}^{in} \rightarrow v^{out}, c'^{out}$ and $\mathtt{p}^{in}, c^{in} \vdash^{call} u^{in}, v^{in} \rightarrow w^{out}, c'^{out}$.

By Lemmas 7.8 and 7.15, the encoding of Figure 7.1 in program form shown in Figure 7.2 computes $[\![\mathtt{p}]\!]\ \mathtt{as}$.

Analysis of running time

Theorem 7.16
If $\mathtt{p}$ is a terminating read-only program of data order k, then for some constant a, $\mathrm{Accept}^{\mathtt{p}} \in \mathrm{TIME}(\exp_{k+1}(a \log n))$.

```
evalprogram p as = fst(evalexp p [(x,as)] [] e)
                   where [[f x = e  def_2 ... def_m]] = p

evalexp p c env [[x]]       = (env(x), c)
evalexp p c env [[f]]       = ((f,[]), c)

evalexp p c env[[e_1 e_2]] = let (u,c')  = evalexp p c env e1 in
                             let (v,c'') = evalexp p c' env e2 in
                             evalcall p c'' u v
 - other cases -

evalcall p c (f,vs) v = let args = vs++[v] in
   if length args < (arity f p) then ((f, args),c)
   else
   case (lookupcache c f args) of
    Notin:  let (params, body) = (lookupfunction f p) in
            let newenv = zip params args in
            let (w,c') = evalexp p c newenv body in (f,args,w):c'
    Hit(w): (w,c)
```

Figure 7.2: Algorithm for time-economical expression evaluation

Proof
The algorithm above calls `evalexp` and `evalcall` a number of times proportional to the size of tree T_{cache}, which by Lemma 7.14 is bounded by $\exp_{k+1}(a \log n))$ for appropriate a.

The time to look up any one cache entry (`lookupcache`) is at most logarithmic in the cache's total size, which is also dominated by $\exp_{k+1}(a' \log n))$ for suitable a'. The amount of time for calls to `arity`, `lookupfunction` and `zip` are independent of `as`.

These operations can be implemented on a Turing machine with at most polynomial slowdown. Consequently the total time is bounded by $\exp_{k+1}(a'' \log n))$ for suitable a''. □

Theorem 7.17
$\text{RO}^k = \text{EXP}^k\text{SPACE}$.

Proof
Immediate from Theorems 6.12 and 7.16. □

7.4 A semantics that is space-economical for tail recursive programs

The previous construction's running time was bounded by the product of the number of nodes in T_{cache} and the time to process any one node. The situation is more delicate for space bounds, as the memory size goal to be achieved is considerably less than the size of the proof tree. Thus a careful analysis of the space required to traverse it will be needed.

This alternative semantics is obtained from the original one by two changes.

Expression evaluation:

$$\frac{}{\mathtt{p}, env : s \vdash \mathtt{x} \rightarrow env(\mathtt{x}), env : s} \qquad \frac{}{\mathtt{p}, s \vdash \mathtt{f} \rightarrow \langle \mathtt{f}, \varepsilon \rangle, s}$$

Function call

$$\frac{\mathtt{p}, s \vdash \mathtt{e}_1 \rightarrow u, s' \quad \mathtt{p}, s' \vdash \mathtt{e}_2 \rightarrow v, s'' \quad \mathtt{p}, s'' \vdash^{call} \mathtt{e}_1\mathtt{e}_2, u, v \rightarrow w, s'''}{\mathtt{p}, s \vdash \mathtt{e}_1\mathtt{e}_2 \rightarrow w, s'''}$$

$$\frac{}{\mathtt{p}, s \vdash^{call} \mathtt{e}_1\mathtt{e}_2, \langle \mathtt{f}, v_1 \ldots v_{i-1} \rangle, v_i \rightarrow \langle \mathtt{f}, v_1 \ldots v_{i-1} v_i \rangle, s} \quad (\text{If } i < arity(\mathtt{f}))$$

$$\frac{\mathtt{p}, [\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_m \mapsto v_m] : s \vdash \mathtt{e}^{\mathtt{f}} \rightarrow w, env' : s'}{\mathtt{p}, s \vdash^{call} \mathtt{e}_1\mathtt{e}_2, \langle \mathtt{f}, v_1 \ldots v_{m-1} \rangle, v_m \rightarrow w, s'} \quad \begin{array}{l}(\text{If } \mathtt{e}_1\mathtt{e}_2 \text{ is not in tail position and} \\ \mathtt{p} \text{ contains } \mathtt{f}\ \mathtt{x}_1\mathtt{x}_2 \ldots \mathtt{x}_m = \mathtt{e}^{\mathtt{f}})\end{array}$$

$$\frac{\mathtt{p}, [\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_m \mapsto v_m] : s \vdash \mathtt{e}^{\mathtt{f}} \rightarrow w, s'}{\mathtt{p}, env : s \vdash^{call} \mathtt{e}_1\mathtt{e}_2, \langle \mathtt{f}, v_1 \ldots v_{m-1} \rangle, v_m \rightarrow w, s'} \quad \begin{array}{l}(\text{If } \mathtt{e}_1\mathtt{e}_2 \text{ is in tail position and} \\ \mathtt{p} \text{ contains } \mathtt{f}\ \mathtt{x}_1\mathtt{x}_2 \ldots \mathtt{x}_m = \mathtt{e}^{\mathtt{f}})\end{array}$$

Program execution:

$$\frac{\mathtt{p}, [\mathtt{x} \mapsto [\mathtt{a}_1, \ldots, \mathtt{a}_m]] : [] \vdash \mathtt{e}^{\mathtt{f}} \rightarrow \mathtt{a}, s}{[\![\mathtt{p}]\!]\ [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] = \mathtt{a}} \quad (\text{If } \mathtt{p} \text{ begins with } \mathtt{f}\ \mathtt{x} = \mathtt{e}^{\mathtt{f}})$$

Figure 7.3: Semantics for space-economical expression evaluation

First, the environment is replaced by a stack of environments $s \in Stack = Env^*$. As usual, this stack is pushed when a function is called, and popped when control returns. Second, the semantics is defined so that whenever a tail call is executed, the current topmost environment on the stack is removed before pushing a new one.

The judgement form for evaluating expressions is $\mathtt{p}, s \vdash \mathtt{e} \rightarrow v, s'$ meaning that expression $\mathtt{e}$ is evaluated under environment env, which is the top of the stack s, yielding value v and final stack s'. The judgement form for running programs is, as before, $[\![\mathtt{p}]\!]\ [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] = \mathtt{a}$. The inference rules for evaluation are based on the following **stack invariant**:

1. If $\mathtt{e}$ is not in tail position and $\mathtt{p}, s \vdash \mathtt{e} \rightarrow v, s'$ then $s = s'$.
2. If $\mathtt{e}$ is in tail position and $\mathtt{p}, env : s \vdash \mathtt{e} \rightarrow v, env' : s'$ then $s = s'$.

Figure 7.3 only shows the modified rules for variables and function application; extension to the others is routine.

We check that the stack invariant holds, by induction on the size of proof trees of judgements $\mathtt{p}, s \vdash \mathtt{e} \rightarrow v, s'$. The two axioms trivially preserve it. Expressions $\mathtt{e}_1$, $\mathtt{e}_2$ in a call $\mathtt{e}_1\ \mathtt{e}_2$ are not in tail position, so $s = s' = s''$ in the first call rule. We now check cases for the judgement $\mathtt{p}, s'' \vdash^{call} \mathtt{e}_1\mathtt{e}_2, u, v \rightarrow w, s'''$.

The rule for incomplete calls leaves the stack unchanged so $s'' = s'''$, preserving the invariant. The next rule, for a complete non-tail call, increases the stack length by 1 before doing the call and then decreases the resulting stack's length by 1 on return. By inductive assumption on the body of the called function, this implies $s'' = s'''$ as desired.

The rule for a complete tail call changes stack $s'' = env : s$ to $env' : s$ (where $env = [\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_m \mapsto v_m]$. Inductively this becomes of form $env'' : s$, with the same s, and so the second part of the invariant is satisfied.

Theorem 7.18

Let p be any program and as an input. There exists a computation tree T proving $[\![\mathtt{p}]\!]\ \mathtt{as} = \mathtt{a}$ by the inference rules of Section 4 if and only if there exists a computation tree T_{tail} proving $[\![\mathtt{p}]\!]\ \mathtt{as} = \mathtt{a}$ by the inference rules just given.

Proof

Omitted for brevity, since based on standard implementation concepts. □

Lemma 7.19

The inference rules of Figure 7.3 are locally deterministic and LR.

Lemma 7.20

Let p be a tail recursive read-only program. Then any computation tree T_{tail} proving $[\![\mathtt{p}]\!]\ [\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n] = \mathtt{a}$ by the inference rules just given has stack depth bounded by the number r of names of functions defined in p.

Proof

Let $\geq$ be the partial order on p's function names. Let $\mathtt{f}\,\mathtt{x}_1 \ldots \mathtt{x}_m = \ldots \mathtt{e}_1\ \mathtt{e}_2 \ldots$ be a complete call such that $\mathtt{e}_1$ evaluates to a closure $(\mathtt{g}, v_1, \ldots, v_{arity(\mathtt{g})-1})$. Since p is tail recursive, $\mathtt{f} \equiv \mathtt{g}$ if this is a tail call, and $\mathtt{f} > \mathtt{g}$ if it is not a tail call.

Examination of the inference rules just given reveals that the stack depth is increased *only for non-tail calls*, otherwise the stack depth stays unchanged. Since there are only r distinct function names, creation of a stack s of depth greater than r is impossible. □

A space-economical algorithm to compute $[\![\mathtt{p}]\!]$ as

Lemma 7.21

The inference rules of Figure 7.3 are locally deterministic, and LR with respect to moding $\mathtt{p}^{in}, s^{in} \vdash \mathtt{e}^{in} \to v^{out}, s'^{out}$ and $\mathtt{p}^{in}, s^{in} \vdash^{call} u^{in}, v^{in}, \mathtt{e}^{in} \to w^{out}, s'^{out}$.

By Lemmas 7.8 and 7.21, the encoding of Figure 7.3 in program form shown in Figure 7.4 computes $[\![\mathtt{p}]\!]$ as.

Analysis of memory usage

Theorem 7.22

If p is a terminating tail recursive read-only program of data order $k \geq 1$, there is a constant a such that $\mathrm{Accept}^{\mathtt{p}} \in \mathrm{SPACE}(\exp_k(a \log n))$.

Proof

```
evalprogram p as = fst(evalexp p ([(x,as)]:[]) e
)                   where [[f x = e  def_2 ... def_m]] = p

evalexp p s env [[x]]       = let env:s' = s in (env(x), s)
evalexp p s env [[f]]       = ((f,[]), s)

evalexp p s env [[e_1 e_2]] =  let (u,s')  = evalexp p s e1 in
                               let (v,s'') = evalexp p s' e2 in
                               evalcall p s'' e u v
 - other cases -

evalcall p s (f,vs) v e = let args = vs++[v] in
   if length args < (arity f p) then ((f,args),s)
   else
     let (params, body) = (lookupfunction f p) in
     let newenv = zip params args in
     if not (tailposition e p) then
          let (w, env':s') = evalexp p newenv:s body in (w,s')
     else let env:s'=s in evalexp p newenv:s' body
```

Figure 7.4: Algorithm for space-economical expression evaluation

We show that the algorithm above uses memory can be implemented using at most $\exp_k(a \log n))$ bits for some a and all n. Note that any value depending only on program `p` is constant, since `p` is fixed.

Memory for the stack: First, the program uses s in a "single-threaded" way, i.e., any time a new stack s', etc. is constructed, the previous version will never be used again. Consequently only one copy of stack s is needed, and it can be stored globally.

By Lemma 7.12, at most $\log(\exp_{k+1}(a \log n)) = \exp_k(a \log n))$ bits are required to store any one variable. This implies that any stack frame (an environment *env*) on the stack occupies at most this amount multiplied by the number of `p`'s parameters. By Lemma 7.20, the stack depth is bounded by a program-dependent constant r, so the space to store the entire stack is a constant multiple of $\exp_k(a \log n)$ bits.

Local memory for the evaluation functions: Any one call of `evalexp` needs memory for the values of program `p` and expression `e`, plus the two values `u` and `v`. The first two are constant, and the others each bounded by $\exp_k(a \log n)$ bits. Function `evalcall` follows the same pattern.

Global memory for the evaluation functions: Each call from function `evalexp` to itself decreases the syntactic argument `e`, so their depth is bounded by a constant. The call from `evalexp` to `evalcall` is a tail call and so requires no extra storage. The first call from `evalcall` to `evalexp` is a non-tail call which implements a non-tail call in `p`; by Lemma 7.20 the call depth is bounded by r. The second call from `evalcall` to `evalexp` is a tail call which implements a tail call in `p`. Thus the current arguments to `evalcall` can be overwritten by `evalexp`'s new arguments.

Conclusion: the memory required to implement the algorithm is at most a constant multiple of $\exp_k(a \log n)$ bits, and so is bounded by $\exp_k(a' \log n)$ bits for some

a'. The same space bound can be realized on a Turing machine implementation. □

Theorem 7.23
$\text{EXP}^{k-1}\text{SPACE} = \text{ROTR}^k$, for any $k \geq 1$.

Proof
Immediate from Theorems 6.15 and 7.22. □

8 Appendix. Connections with finite model theory

Given a fixed many-sorted signature $\mathcal{S}$, an $\mathcal{S}$-algebra $\mathcal{A}$ specifies a carrier set for every sort in $\mathcal{S}$, and assigns a function of appropriate type as meaning to every operator symbol in $\mathcal{S}$. In finite model theory $\mathcal{S}$ is often assumed to contain fixed sorts "bool" and "ind". Algebra $\mathcal{A}$ respectively interprets these as sets $\{0,1\}$ and $\{0,1,\ldots,n\}$ (for varying n), and interprets operations $\text{MIN}, \text{MAX} : \text{ind}, \text{NEXT} : \text{ind} \to \text{ind}$ respectively as minimum 0, maximum n, and $\lambda x \,.\, \min(x+1, n)$.

One concern of finite model theory is computation of "global functions" on finite $\mathcal{S}$-algebras $\mathcal{A}$. A program to define a global function $f(\mathcal{A}, x_1, \ldots, x_m)$ is given by a term t in a simply-typed lambda calculus containing the operator symbols of $\mathcal{S}$, and either general fixpoint operators or primitive recursion. Program semantics is defined by evaluating the term, given as input a finite algebra $\mathcal{A}$ and values of appropriate types for $x_1, \ldots, x_m$. Program terms t always have type boolean.

A typical theorem (Gurevich, 1983) is that a global function f is definable by a first-order primitive recursive term if and only if there exists a LOGSPACE Turing machine which, given as input

- an encoded algebra $\mathcal{A}$ defining all the sort sets and operators in $\mathcal{S}$, and
- some encoded inputs $i_1, \ldots, i_m$ to f,

terminates with the encoded value of $f(\mathcal{A}, i_1, \ldots, i_m)$.

A connection with traditional complexity theory. What does this have to do with deciding membership in a set of strings $A \subseteq \{0,1\}^*$? A connection can be made by considering signature $\mathcal{S}$ with sorts "bool" and "ind" and an operator symbol $\varphi : \text{ind} \to \text{bool}$.

Then an input-free global function $f(\mathcal{A})$ of type $f : \to \text{bool}$ is Turing-computed, as defined above, by providing the machine with input that encodes

$$(\mathcal{A}_{\text{bool}}, \mathcal{A}_{\text{ind}}, \mathcal{A}_{\phi}, \mathcal{A}_{\text{MIN}}, \mathcal{A}_{\text{MAX}}, \mathcal{A}_{\text{NEXT}})$$

where $\mathcal{A}_{\text{bool}} = \{0,1\}$ is the interpretation of bool, and $\mathcal{A}_{\text{ind}} = \{0,1,\ldots,n\}$ is the interpretation of "ind". The third component is φ's interpretation. This has type $\mathcal{A}_{\varphi} : \{0,1,\ldots,n\} \to \{0,1\}$and so is in essence an $n+1$-bit string $\mathtt{a}_0 \ldots \mathtt{a}_n$ in $\{0,1\}^*$. The machine's output is the value of term t, a boolean value.

Thus computing $f()$ amounts to producing a 0-1 answer for the bit string $a_0 \ldots a_n$ which is given in the input interpretation, i.e., it amounts to deciding membership in a subset A of $\{0,1\}^*$. It is furthermore clear that A is in, say, LOGSPACE iff the Turing machine operates in space logarithmic in the size of $\mathcal{A}$'s representation.

Connection with functional programming. Regard any term p : bool in the lambda expression language mentioned above as a functional program `p:[Bool]->Bool` as follows. Think of the function $\mathcal{A}_\varphi = [0 \mapsto a_0, 1 \mapsto a_1, \ldots, n \mapsto a_n]$ above interpreting φ : ind $\to$ bool as `p`'s input: a string $\texttt{[a}_0\texttt{,...,a}_n\texttt{]}$ of type `[Bool]`. From this viewpoint, any variable x of type "ind" in p corresponds to a variable `x:[Bool]` in `p`. This can be done since any p variable x of sort "ind" has value in $\{0, 1, \ldots, n\}$, so its value can be represented by a suffix `x` of input $\texttt{[a}_0\texttt{,...,a}_n\texttt{]}$ (which is passed as parameter to every function defined in `p`). Now MAX and MIN are obviously computable using this representation; and NEXT can be computed with a bit of programming (even without "cons").

Conversely, from any read-only functional program `p` one can construct a lambda-term as in (Goerdt, 1992) that is equivalent in the same sense.

References

Bellantoni, S. and and Cook, S. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity* **2**, ACM Press (1992), 97-110.

Bellantoni, S., Niggl, K.-H. and Schwichtenberg, H. Characterising Polytime Through Higher Type Recursion. (Manuscript for the Dagstuhl Seminar 98 Programs: Improvements, Complexity and Meanings.)

Bird, R., Jones, G. and De Moor, O. More haste less speed: lazy versus eager evaluation. *Journal of Functional Programming* (1997), 541–547.

Cobham, A. The intrinsic computational difficulty of functions. *Proceedings of the Congress for Logic, Mathematics and Philosophy of Science*, (1964), 24–30.

Cook, S. A. Characterizations of pushdown machines in terms of time-bounded computers. Journal of the ACM **18** (1971), 4–18.

Crockett, R. and Spencer, D. Strong Categorical Datatypes I. *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings, Editor R. A. G. Seely. AMS, 1992.

Girard, J.-Y. and Lafont, Y. and Taylor, P. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science* Cambridge University Press, 1989.

Goerdt, A. Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation* **101** (1992), 201-218.

Goerdt, A. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science* **101** (1992), 45–66.

Goerdt, A. and Seidl, H. Characterizing complexity classes by higher type primitive recursive definitions, Part II. *Proceedings 6th International Meeting for Young Computer Scientists*, Lecture Notes in Computer Science **464** (1990), 148–158.

Grzegorczyk, A. Some classes of recursive functions. *Rozprawy Matematik IV* Warsaw (1953).

Gurevich, Y. Algebras of feasible functions. *Proceedings 24th Symposium on Foundations of Computer Science* (IEEE Computer Society Press) (1983), 210–214.

Gurevich, Y. Towards logic tailored for computational complexity. *Computation and Proof Theory*, Lecture Notes in Mathematics **194** (1984), 99-117.

Hillebrand, G. Finite Model Theory in the Simply Typed Lambda Calculus. Ph.D. thesis, Brown University, 1994.

Hillebrand, G. and Kanellakis, P. On the expressive power of simply typed and let-polymorphic lambda calculi. *Proceedings Logic in Computer Science* (IEEE Computer Society Press) (1996), 253–263.

Hofmann, M. Type Systems for Polynomial-time Computation. Habilitationsschrift, Darmstadt, 1999. Appeared as LFCS Technical Report ECS-LFCS-99-406, University of Edinburgh.

Hutton, G. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, vol. 9, no. 4, 355–372 (1999).

Immerman, N. Expressibility as a complexity measure: results and directions. *Proceedings 2. Conference on Structure in Complexity Theory* (IEEE Computer Society Press) (1987), 223–257.

Jones, N. D. and Selman, A.Turing machines and the spectra of first-order formulae with equality, *Journal of Symbolic Logic*, vol. 39, no. 1, pp. 139–150, 1974.

Jones, N. D. *Computability and Complexity from a Programming Perspective.* The MIT Press, 1997.

Jones, N. D. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 1999.

Kfoury, A., Recursion versus iteration at higher-orders. Foundations of Software Technology and Theoretical Computer Science, Indian Institute of Technology, Kharagpur, India, Dec 18-20, 1997. Lecture Notes in Computer Science **1346** (1997).

Kfoury, A. Recursion, tail-recursion and iteration at higher-orders, (notes for a paper). Unpublished.

Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. The hierarchy of functional programs. *Proceedings 2nd IEEE Symposium on Logic in Computer Science* (IEEE Computer Society Press) (1987), 225–235.

Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. On the expressive power of finitely typed and universally polymorphic recursive procedures. Theoretical Computer Science **93** (1992), 1–41.

Leivant, D. Descriptional characterizations of computational complexity classes. *Journal of Computer and System Sciences* **39** (1989), 51–83.

Leivant, D. and Marion, J.-Y. Lambda calculus characterizations of poly-time. *Fundamentae Informatica* **19** (1993), 167-184.

Leivant, D. and Marion, J.-Y. Ramified recurrence and computational complexity IV: predicative functionals and poly-space. *Information and Computation* (1999).

Leivant, D. Applicative control and computational complexity. Unpublished manuscript (1999).

Lloyd, J. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming* **3**, The MIT Press (1999).

Neergaard, P. Time analysis of lazy versus eager evaluation. *DIKU Technical Report*, University of Copenhagen (1999).

Paterson, M. and Hewitt, C. Comparative schematology. *MIT A.I. Lab Technical Memo no. 201* (also in *Proc. of Project MAC Conference on Concurrent Systems and Parallel Computation*) (1970).

Pippenger, N. Pure versus impure LISP. *ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery (1996), 104–109.

J.E. Savage. *Models of Computation.* Addison Wesley, 1998.

Sazonov, V. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik* **16** (1980), 319-323.

Schmidt, D. *Denotational Semantics*, Boston, MA: Allyn and Bacon, 1986.

Turner, D. Elementary Strong Functional Programming. *First International Symposium* *on Functional Programming Languages in Education*, R.Plasmeijer, P.Hartel, eds, LNCS 1022 (1996), 1–13.

Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*, The MIT Press, 1986.

Voda, P. Subrecursion as a basis for a feasible programming language. *Computer Science Logic*, Lecture Notes in Computer Science **933** (1994), 324–338.

Voda, P. A simple ordinal recursive normalization of Gödel's T. *Computer Science Logic*, Lecture Notes in Computer Science **1414** (1997), 491–509.