ESSLLI 2000 Lecture Notes: Computability and Complexity from a Programming Perspective

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

Abstract

A programming approach to computability and complexity theory yields proofs of central results that are sometimes more natural than the classical ones; and some new results as well. These notes contain some high points from the recent book [15], emphasising what is different or novel with respect to more traditional treatments. Topics include:

- Kleene's *s-m-n* theorem applied to compiling and compiler generation.
- Proof that *constant time factors do matter*: for on a natural computation model, problems solvable in linear time have a proper hierarchy, ordered by coefficient values.
- Characterisations in programming terms of a wide range of computational complexity classes. These are intrinsic: without externally imposed space or time computation bounds.
- Results on which problems possess optimal algorithms, including Levin's Search theorem (for the first time in book form).
- Boolean program problems complete for PTIME, NPTIME, PSPACE.

1 Introduction

Book [15] differs didactically and foundationally from traditional treatments of computability and complexity theory. Its didactic aim is to teach the main theorems of these fields to computer scientists. Its approach is to exploit as much as possible the readers' programming intuitions, and to motivate subjects by their relevance to programming concepts. Foundationally it differs by using, instead of the natural numbers $\mathbb{N}$, binary trees as data (as in the programming language Lisp). Consequently programs *are* data, avoiding the usual complexities and inefficiencies of encoding program syntax as Gödel numbers.

Topics covered in computability theory

1. `WHILE` programs as a computation model.
2. The *s-m-n* and universal function theorems: program specialisers and self-interpreters.
3. Undecidability, Rice's theorem, recursively enumerable and recursive sets (r.e. or semidecidable versus decidable).
4. Evidence for the Church-Turing thesis: robustness, i.e., mutual simulability among several different models of computation.
5. Unsolvable problems: a selection, including Post's Correspondance Problem, ambiguity of a context-free grammar, and Hilbert's Tenth Problem.

*This research was partially supported by the Danish Natural Science Research Council (*DART* project), and the Esprit *Atlantique* project.

6. A version of Gödel's Incompleteness theorem: provability by means of inference rules cannot capture all of truth, even for a very simple logic.
7. Model-independent results: Rogers' Isomorphism theorem, Kleene's and Rogers' Recursion theorems. Efficient implementations of the two Recursion theorems in a specific model.

Topics covered in complexity theory

1. Time and space measures for several computing models.
2. A hierarchy involving the computational resources *time*, *space*, and *nondeterminism*:

   LOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ PTIME $\subseteq$ NPTIME $\subseteq$ PSPACE $\subset$ REC $\subset$ RE
3. The existence or nonexistence of optimally efficient algorithms for various problems (Levin's Search theorem, Blum's Speedup theorem, and the Gap theorem).
4. Characterisations in programming terms of several computational complexity classes including LOGSPACE and PTIME. These are intrinsic: without externally imposed space or time computation bounds.
5. Resource-bounded reduction of one decision problem to another.
6. Problems complete for NLOGSPACE, PTIME, NPTIME, PSPACE, RE.

2 The `WHILE` and `I` languages

The simple programming language `WHILE` is in essence a small subset of LISP or Pascal. Why just this language? Because `WHILE` seems to have just the right mix of expressive power and simplicity for our purposes. *Expressive power* is important when writing programs that deal with programs as data objects. The data structures of `WHILE` (binary trees of atomic symbols) are particularly well suited to this. The reason is the `WHILE` data structures avoid the need for the technically messy tasks of assigning *Gödel numbers* to encode program texts and fragments, and of devising numerical functions to build and decompose Gödel numbers.

Simplicity is also essential to prove theorems about programs and their behaviour. This rules out the use of larger, more powerful languages, since proofs about them would simply be too complex to be easily understood. When proving theorems about programs, we use an equivalent but still simpler language `I`: identical to `WHILE`, but limited to programs with one variable and one atom.

2.1 Syntax of `WHILE` data and programs

Values in the set $\mathbb{D}_A$ of data values are built up from a fixed finite set A of so-called *atoms* by finitely many applications of the pairing operation. It will not matter too much exactly what A contains, except that we will identify one of its elements, `nil` for several purposes. To reduce notation we write $\mathbb{D}$ instead of $\mathbb{D}_{\{\texttt{nil}\}}$ when $A = \{\texttt{nil}\}$.

A value $\texttt{d} \in \mathbb{D}_A$ is a binary tree with atoms as leaf labels. An example, written in "fully parenthesised form": `((a.((b.nil).c)).nil)`.

Definition 2.1 Let $A = \{\texttt{a}_1, \ldots, \texttt{a}_n\}$ be some finite set. Then

1. $\mathbb{D}_A$ is the smallest set satisfying $\mathbb{D}_A = (\mathbb{D}_A \times \mathbb{D}_A) \cup A$. The *pairing* or *cons* operation "." yields pair $(\texttt{d}_1.\texttt{d}_2)$ when applied to values $\texttt{d}_1, \texttt{d}_2 \in \mathbb{D}$.
2. The *size* $|d|$ of a value $\texttt{d} \in \mathbb{D}_A$ is defined as follows: $|\texttt{d}| = 1$ if $\texttt{a} \in A$, and $1 + |\texttt{d}_1| + |\texttt{d}_2|$ if $\texttt{d} = (\texttt{d}_1.\texttt{d}_2)$.

Syntactic category:		*Informal syntax:*	*Concrete syntax:*
P : Program	::=	`read X1; C; write X1`	`C`
C : Command	::=	`Xi := E`	`(:= nil`^{i} `E)`
	|	`C1; C2`	`(; C1 C2)`
	|	`if E then C1 else C2`	`(if E C1 C2)`
	|	`while E do C`	`(while E C)`
E : Expression	::=	`Xi`	`(var nil`^{i}`)`
	|	`D`	`(quote D)`
	|	`cons E1 E2`	`(cons E1 E2)`
	|	`hd E`	`(hd E)`
	|	`tl E`	`(tl E)`
	|	`atom=? E1 E2`	`(atom=? E1 E2)`
D : Data-value	::=	`A | (D.D)`	`A | (D.D)`
A : Atom	::=	`nil ( | ...)`	`nil | ...`

Figure 2.1: Program syntax: informal and concrete

2.1.1 A compact linear notation for values:

Unfortunately it is hard to read deeply parenthesised structures (one has to resort to counting), so we will use a more compact "list notation" taken from the Lisp and Scheme languages, in which

`()` stands for `nil`

`(` $\mathtt{d}_1\ \mathtt{d}_2\ \cdots\ \mathtt{d}_n$ `)` stands for $(\mathtt{d}_1.(\mathtt{d}_2.\cdots(\mathtt{d}_n.\mathtt{nil})\cdots))$

The syntax of `WHILE` programs is given by the "informal syntax" part of Figure 2.1. Programs have only one input/output variable `X1`, and to manipulate tree structures built by `cons` from atoms. Informally, "`cons`" is `WHILE` code for the pairing operation so $\mathtt{cons}(d, e) = (d.e)$. "`hd`" and "`cotlns`" are its left and right inverses, so $\mathtt{hd}(d.e) = d$ and $\mathtt{tl}(d.e) = e$. For completeness, we define $\mathtt{hd}(\mathtt{nil}) = \mathtt{tl}(\mathtt{nil}) = \mathtt{nil}$

Control structures are *sequential composition* `C1;C2`, the *conditional* `if E then C1 else C2`, and the *while loop* `while E do C`. Assignments use `:=` in the usual way. Operations `hd` and `tl` (head, tail) decompose such structures, and `atom=?` tests atoms for equality. In tests, `nil` serves as "false," and any non-`nil` value serves as "true." We often write `false` for `nil`, and `true` for `(nil.nil)`. An example, the following program, `reverse`:

```
read X;
  Y := nil;
  while X do { Y := cons (hd X) Y;  X := tl X };
write Y
```

satisfies $[\![\mathtt{reverse}]\!](\mathtt{a.(b.(c.nil))}) = \mathtt{(c.(b.(a.nil)))}$, or $[\![\mathtt{reverse}]\!]\mathtt{(a\ b\ c)} = \mathtt{(c\ b\ a)}$.

2.1.2 Concrete syntax for `WHILE`-programs.

The column "Concrete syntax" in Figure 2.1 contains a representation of each `WHILE`-program as an element of $\mathbb{D}_A$. Notation: symbols

:=, nil, ;, if, while, var, quote, cons, hd, tl, atom=?

stand for distinct elements of $\mathbb{D}$ (exactly which ones, is not important). Note that with this representation, programs *are* data; no encoding is needed.

2.2 Semantics of WHILE programs

Informally, the net effect of *running a program* p is to compute a partial function $[\![\mathtt{p}]\!] : \mathbb{D} \to \mathbb{D}_\perp$, where $\mathbb{D}_\perp$ abbreviates $\mathbb{D} \cup \{\perp\}$. We write $[\![\mathtt{p}]\!](\mathtt{d}) = \perp$ to mean "the computation by p on d does not terminate."

A formal definition of the semantic function $[\![\mathtt{p}]\!]$ may be found in [15]. Its approach, here simplified to programs containing only one variable X, is to define two relations, $\mathcal{E} \subseteq$ Expression $\times$ $\mathbb{D} \times \mathbb{D}$ and $\vdash \subseteq$ Command $\times$ $\mathbb{D} \times \mathbb{D}$ with these interpretations:

$\mathcal{E}[\![\mathtt{E}]\!]\mathtt{d} = \mathtt{d}'$ Assuming the value of variable X is d, then the value of expression E is d′

$\mathtt{C} \vdash \mathtt{d} \to \mathtt{d}'$ Assuming the initial value of variable X is d, then the execution of command C terminates, with d′ as the final value of X

For brevity, running time is only informally defined. This definition is natural, provided the data-sharing implementation techniques used in Lisp and other functional languages is used.

Definition 2.2 The running time $time_{\mathtt{p}}(\mathtt{d}) \in \{0, 1, 2, \ldots\} \cup \{\perp\}$ is obtained by counting 1 every time any of the following is performed while computing $[\![\mathtt{p}]\!](\mathtt{d})$ as defined in the semantics: a variable or constant reference; an operation hd, tl, cons, or :=; or a test in an if or while command. Its value is $\perp$ if the computation does not terminate.

2.3 Decision problems

Definition 2.3

1. A set $A \subseteq \mathbb{D}$ is WHILE decidable (or *recursive*) iff there is a WHILE program p such that for all $\mathtt{d} \in \mathbb{D}$, $[\![\mathtt{p}]\!](\mathtt{d}) =$ true if $\mathtt{d} \in A$ and $[\![\mathtt{p}]\!](\mathtt{d}) =$ false if $\mathtt{d} \notin A$.

2. A set $A \subseteq \mathbb{D}$ is WHILE semi-decidable (or *recursively enumerable*, abbreviated to r.e.) iff for some WHILE-program p

$$A = \text{domain}([\![\mathtt{p}]\!]) = \{\mathtt{d} \in \mathbb{D} \mid [\![\mathtt{p}]\!](\mathtt{d}) \neq \perp\}$$

Thus a recursive set A's membership question can be answered by a terminating program which, for any input $\mathtt{d} \in \mathbb{D}$, will answer "true" if d lies in A and "false" otherwise. A recursively enumerable set's membership question is answered by a program p that, for any input $\mathtt{d} \in \mathbb{D}$, terminates when d lies in A and loops infinitely when it does not.

These concepts lie at the very core of computability theory. They have many alternative chacterisations; and are independent of the programming languages used to define them.

2.4 Some simple constructions

The following examples illustrate the simplicity of WHILE-program manipulation. All use the concrete syntax of Figure 2.1.

2.4.1 The halting problem

Following is a well-known result: the *undecidability of the halting problem*:

Theorem 2.4 The set $HALT$ is not recursive, where

$$HALT = \{(\mathtt{p}.\mathtt{d}) \mid [\![\mathtt{p}]\!](\mathtt{d}) \neq \perp\}$$

Proof Suppose program q decides $HALT$, so for any WHILE-program p and input d

$$[\![\mathtt{q}]\!](\mathtt{p}.\mathtt{d}) = \begin{cases} \mathtt{true} & \text{if } [\![\mathtt{p}]\!](\mathtt{d}) \neq \bot \\ \mathtt{false} & \text{if } [\![\mathtt{p}]\!](\mathtt{d}) = \bot \end{cases}$$

Now q must have form: `read X; C; write X`. Construct the following program r from q:

```
read X;
X := cons X X;              (*  Does program X stop on input X? *)
C;                          (*  Apply program q to answer this  *)
if X
   then  while X do X := X  (*  Loop if X stops on input X      *)
   else  X := nil;          (*  Terminate if it does not        *)
write X
```

Consider the input X = r. First, suppose that $[\![\mathtt{r}]\!](\mathtt{r}) \neq \bot$. Then control in r's computation on input r must reach the else branch above (else r would loop on r). But then X = false holds after command C, so $[\![\mathtt{r}]\!](\mathtt{r}) = \bot$ by the assumption that q decides the halting problem. This is contradictory. Conclusion: $[\![\mathtt{r}]\!](\mathtt{r}) = \bot$. By similar reasoning, $[\![\mathtt{r}]\!](\mathtt{r}) = \bot$ is also impossible.

The only unjustified assumption was existence of q, so this must be false.

2.4.2 Decidable and semi-decidable sets

Theorem 2.5 If A and $\overline{A}$ are both recursively enumerable, then A is recursive.

Proof idea: Let $A = \text{domain}([\![\mathtt{p}]\!])$ and let $\overline{A} = \text{domain}([\![\mathtt{q}]\!])$. *Decision procedure* for given input d: run p and q alternately, one step at a time until one stops (and one of them must). If p stops first then $\mathtt{d} \in A$, else $\mathtt{d} \notin A$.

Theorem 2.6 The set $HALT$ is recursively enumerable but not recursive. Further, the set $\overline{HALT}$ is not recursively enumerable, where

$$\overline{HALT} = \{(\mathtt{p}.\mathtt{d}) \mid [\![\mathtt{p}]\!](\mathtt{d}) = \bot\}$$

Proof First, note that $HALT = \text{domain}([\![\mathtt{u}]\!])$, and so is recursively enumerable. Thus $\overline{HALT}$ cannot also be recursively enumerable, since Theorem 2.5 would imply that $HALT$ is also recursive.

2.4.3 The *s-m-n* theorem

A *program specialiser* spec is given a subject program p together with part of its input data, s. Its effect is to construct a new program $\mathtt{p}_\mathtt{s} = [\![\mathtt{spec}]\!](\mathtt{p}.\mathtt{s})$ which, when given p's remaining input d, will yield the same result that p would have produced given both inputs.

Definition 2.7 Program spec is a *specialiser* for WHILE-programs if $[\![\mathtt{spec}]\!](\_)$ is total, and if for any $\mathtt{p} \in$ WHILE-*programs* and $\mathtt{s}, \mathtt{d} \in$ WHILE-*data*

$$[\![\mathtt{p}]\!](\mathtt{s}.\mathtt{d}) = [\![[\![\mathtt{spec}]\!](\mathtt{p}.\mathtt{s})]\!](\mathtt{d})$$

Theorem 2.8 (Kleene's *s-m-n* theorem). There is a program specialiser[1] for WHILE-programs.

Proof Program p has form:

[1] Kleene's version: $\phi_p^{m+n}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \phi^n_{s_n^m(p,x_1,\ldots,x_m)}(y_1, \ldots, y_n)$. Definition 2.7 is much simpler, partly because the linear notation $[\![\mathtt{p}]\!](\mathtt{d})$ instead of $\phi^n_\mathtt{p}(\mathtt{d})$ avoids subscripted subscripts. Another reason is the fact that *ID* has a built-in pairing operation. Thus $m = n = 1$ is fully general, so s_n^m for all m, n are not necessary, being replaced by the one function $[\![\mathtt{spec}]\!]$.

```
read X1; Body; write X1
```

Given known input s, construct program $\mathtt{p}_\mathtt{s}$:

```
read X1; X1 := cons s X1; Body; write X1
```

This is obviously (albeit somewhat trivially) correct since $\mathtt{p}_\mathtt{s}$, when given input d, will first assign the value (s.d) to X1, and then apply p to the result. It suffices to see how to construct $\mathtt{p}_\mathtt{s} = [\![\mathtt{spec}]\!](\mathtt{p}.\mathtt{s})$. A program to transform input s and the concrete syntax of program p into $\mathtt{p}_\mathtt{s}$ is easily constructed.

3 Robustness of computability

This section briefly describes some computation models used later in these notes: the GOTO language as a variant of WHILE; the TM language for Turing machines; the CM language for counter machines (and the SRAM language for random access machines is briefly mentioned). Further, we show how to measure the running time and space consumption of their programs. The concepts take on more life for computer scientists if expressed in a context of several programming languages, so we first generalise a bit:

> *Robustness*: all of these models define the same classes of computable functions and decidable sets. Section 3.7 states their equivalence, and proofs may be found in [15]. The reader may wish to skip quickly to Section 3.7 on first reading, returning to the details when or if needed for later sections.

3.1 Programming languages more generally

Definition 3.1 A *programming language* L consists of

1. Two sets, L-*programs* and L-*data*, and two distinct elements $\mathtt{true}, \mathtt{false} \in$ L-*data*.
2. L's *semantic function*: for every $\mathtt{p} \in$L-*programs*, a (partial) input-output function $[\![\mathtt{p}]\!]^{\mathtt{L}}(\_) :$ L-*data* $\to$ L-*data*$_\perp$.
3. L's *running time function*: for every $\mathtt{p} \in$ L-*programs*, a corresponding (partial) time-usage function $time^{\mathtt{L}}_{\mathtt{p}}(\_) :$ L-*data* $\to I\!N \cup \{\perp\}$ such that for any $\mathtt{p} \in$ L-*programs* and $\mathtt{d} \in$ L-*data*, $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}) = \perp$ if and only if $time^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d}) = \perp$

The superscript L will be dropped when L is known from context. In the following we alsways have L-*data* $= I\!D$ or L-*data* $= \{0,1\}^*$

Definition 3.2 L-program p *decides* a subset $A \subseteq$ L-*data* if for any $\mathtt{d} \in$ L-*data*

$$[\![\mathtt{p}]\!](\mathtt{d}) = \begin{cases} \mathtt{true} & \text{if} \quad \mathtt{d} \in A \\ \mathtt{false} & \text{if} \quad \mathtt{d} \in \text{L-}data \setminus A \end{cases}$$

3.2 The GOTO variant of WHILE

The WHILE language has both "structured" syntax and data. This is convenient for programming, but when constructing one program from another it is often convenient to use a lower-level "flow chart" syntax in which a program is a sequence p = 1:I1 2:I2 ... m:Im of labeled instructions, executed serially except as redirected by control transfers. Instructions are limited to the forms X:=nil, X:=Y, X:=cons Y Z, X:=hd Y, X:=tl Y, or if X goto ℓ else ℓ'.

The semantics is natural and so not presented here. Following is a GOTO program to reverse its imput string.[2]

```
0: read X;
1: if X goto 2 else 6
2: Z := hd X;
3: Y := cons Z Y;
4: X := tl X;
5: goto 1;
6: write Y
```

It is easy to see that any WHILE program can be translated into an equivalent GOTO program running at most a constant factor slower (measuring GOTO times by the number of instructions executed). Conversely, any GOTO program can be translated into an equivalent WHILE program running at most a constant factor slower (the factor may depend on the size of the GOTO program).

3.3 Functional programming languages

3.3.1 FUN1, a first-order functional language

This language and its higher-order counterpart FUNHO are functional: there is no assignment operator, and a program is a collection of mutually recursive function definitions.

Definition 3.3 Syntax of FUN1: programs and expressions have forms given by the grammar:

$\mathtt{p} \in$ Program	::=	$\mathtt{def}_1\ \mathtt{def}_2\ \ldots\ \mathtt{def}_m$
$\mathtt{def} \in$ Definition	::=	$\mathtt{f}(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}$
$\mathtt{e} \in$ Expression	::=	$\mathtt{d} \mid \mathtt{x} \mid \mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2 \mid \mathtt{cons}\ \mathtt{e}_1\mathtt{e}_2 \mid \mathtt{hd}\ \mathtt{e} \mid \mathtt{tl}\ \mathtt{e} \mid \mathtt{f}(\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_n)$
$\mathtt{d} \in$ Constant	::=	any element of $\mathbb{D}$
$\mathtt{x} \in$ Parameter	::=	identifier
$\mathtt{f} \in$ FcnName	::=	identifier

The *main function* $\mathtt{f}_1$ of program p is the one in $\mathtt{def}_1$ above; it must have $n = 1$.

Semantics is call-by-value, as one might expect. For readability we often write $\mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2$ for $\mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2$.

The WHILE language corresponds rather precisely to a restriction of FUN1 to *tail recursive.* programs. Briefly: a tail-recursive program is one in which function calls never appear nested inside other calls, or other operations (cons, hd, tl or in if tests). They may however appear in the then or else branches of an if.

Theorem 3.4 For every WHILE-program p there is a tail-recursive FUN1-program q such that $[\![\mathtt{p}]\!]^{\mathtt{WHILE}} = [\![\mathtt{q}]\!]^{\mathtt{FUN1}}$; and conversely.

Proof. To illustrate the idea, we show a tail recursive functional equivalent of the program reverse seen earlier:

```
reverse(X)   =  step(X, nil)
step(X, Y)   =  if X then  aux(cons (hd X) Y, tl X)
                     else  Y
```

Constructions are very straightforward. Given a WHILE-program p, first construct equivalent GOTO program $\mathtt{p}' = \mathtt{1{:}I1\ 2{:}I2\ \ldots\ m{:}Im}$. If $\mathtt{p}'$ has variables $\mathtt{X}_1, \ldots, \mathtt{X}_k$, the following is an equivalent FUN1 program:

[2] Where goto 1 abbreviates if X goto 1 else 1.

```
f(x) = step1(X1,nil,...,nil)

step1 (X1,...,Xk)   = <I1>
  ...
stepm (X1,...,Xk)   = <Im>
stepm+1 (X1,...,Xk) = X1
```

and the expression <I> for GOTO-instruction I is defined as follows:

```
<l:Xj := E>                 = stepl+1 (X1,...,E,...,Xk)

<l:goto l'>                 = stepl' (X1,...,Xk)

<l:if Xj goto l' else l''> = if Xj then stepl'(X1,...,Xk)
                                   else stepl''(X1,...,Xk)
```

The reverse direction is also easy: each tail call in a FUN1 program is translated to a series of assignments followed by a goto.

3.3.2 FUNHO, a higher-order functional language

This includes FUN1, but for convenient extension to higher-order functions uses Haskell's "named combinator" function syntax. The differences from the syntax of FUN1 are few:

Definition 3.5 Syntax: programs and expressions have forms given by the grammar

$\mathtt{def} \in$ Definition	::=	$\mathtt{f}\ \mathtt{x}_1\mathtt{x}_2 \ldots \mathtt{x}_n = \mathtt{e}^{\mathtt{f}}$
$\mathtt{e} \in$ Expression	::=	$\mathtt{d} \mid \mathtt{x} \mid \mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2 \mid \mathtt{cons}\ \mathtt{e}_1\mathtt{e}_2 \mid \mathtt{hd}\ \mathtt{e} \mid \mathtt{tl}\ \mathtt{e} \mid \mathtt{f} \mid \mathtt{e}_1\mathtt{e}_2$
$\mathtt{f} \in$ FcnName	::=	identifier, disjoint from Parameter

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 $\mathit{arity}(\mathtt{f}_1) = 1$. The formalism is well-known to be equivalent to the lambda calculus with explicit binding and recursion operators λ and μ.

Curried function calls. Multiple-argument functions are handled by "currying." To illustrate informally, consider the program

```
twice f zs   =  f (f zs)
append xs ys =  if xs then cons (hd xs) (append (tl xs) ys)
                      else ys
```

Function append behaves as expected if applied to all its arguments, for example, call append xs ys with xs = (1 2) and ys = (3 4) returns (1 2 3 4). An *incomplete* call, however, is also legal (in contrast to FUN1), and defines a function. For instance call (append xs) with xs = (1 2) is legal, and defines the function $f(ys)$ = "append 1 2 to the front of ys," i.e., $f(ys) = (1.(2.ys))$.

Thus the call (twice (append ys) zs) with ys = (1 2) and zs = (3 4) returns $f(f((3\ 4)))$, that is,(1 2 1 2 3 4).

A call f $\mathtt{e}_1$ $\mathtt{e}_2$ $\mathtt{e}_3$ to a three-argument function can be written: (((f $\mathtt{e}_1$) $\mathtt{e}_2$) $\mathtt{e}_3$). In this, f by itself is an expression, whose value is the function f that f defines. Further, (f $\mathtt{e}_1$) is also an expression, whose value is $f(v_1)$ where v_1 is the value of $\mathtt{e}_1$, etc. The idea is that a higher-order value is obtained by an *incomplete function application.*

Operationally, the expression body $\mathtt{e}^{\mathtt{f}}$ that defines f is not activated until all three arguments are present. The process of collecting arguments until a complete list is obtained is formalised explicitly in the *Function call* rules of Figure 3.1.

Expression evaluation

Some axioms

$$\frac{}{\mathtt{p}, env \vdash \mathtt{d} \rightarrow \mathtt{d}} \qquad \frac{}{\mathtt{p}, env \vdash \mathtt{x} \rightarrow env(\mathtt{x})} \qquad \frac{}{\mathtt{p}, env \vdash \mathtt{f} \rightarrow <\mathtt{f}, \varepsilon>}$$

Cons, head, tail

$$\frac{\mathtt{p}, env \vdash \mathtt{e}_1 \rightarrow v_1 \quad \mathtt{p}, env \vdash \mathtt{e}_2 \rightarrow v_2}{\mathtt{p}, env \vdash \mathtt{cons\ e}_1\ \mathtt{e}_2 \rightarrow (v_1.v_2)}$$

$$\frac{\mathtt{p}, env \vdash \mathtt{e} \rightarrow (v_1.v_2)}{\mathtt{p}, env \vdash \mathtt{hd\ e} \rightarrow v_1} \qquad \frac{\mathtt{p}, env \vdash \mathtt{e} \rightarrow \mathtt{nil}}{\mathtt{p}, env \vdash \mathtt{hd\ e} \rightarrow \mathtt{nil}} \qquad \frac{\mathtt{p}, env \vdash \mathtt{e} \rightarrow (v_1.v_2)}{\mathtt{p}, env \vdash \mathtt{tl\ e} \rightarrow v_2} \qquad \frac{\mathtt{p}, env \vdash \mathtt{e} \rightarrow \mathtt{nil}}{\mathtt{p}, env \vdash \mathtt{tl\ e} \rightarrow \mathtt{nil}}$$

Conditional

$$\frac{\mathtt{p}, env \vdash \mathtt{e}_1 \rightarrow \mathtt{nil} \quad \mathtt{p}, env \vdash \mathtt{e}_2 \rightarrow v}{\mathtt{p}, env \vdash \mathtt{if\ e}_1\mathtt{e}_2\mathtt{e}_3 \rightarrow v} \qquad \frac{\mathtt{p}, env \vdash \mathtt{e}_1 \rightarrow (v_1.v_2) \quad \mathtt{p}, env \vdash \mathtt{e}_3 \rightarrow v}{\mathtt{p}, env \vdash \mathtt{if\ e}_1\mathtt{e}_2\mathtt{e}_3 \rightarrow v}$$

Function call

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

$$\frac{}{\mathtt{p} \vdash^{call} <\mathtt{f}, v_1 \ldots v_{i\perp 1}>, v_i \rightarrow <\mathtt{f}, v_1 \ldots v_{i\perp 1} v_i>} \quad (\text{If } i < arity(\mathtt{f}))$$

$$\frac{\mathtt{p}, [\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_i \mapsto v_i] \vdash \mathtt{e}^{\mathtt{f}} \rightarrow w}{\mathtt{p} \vdash^{call} <\mathtt{f}, v_1 \ldots v_{i\perp 1}>, v_i \rightarrow w} \quad (\text{If } i = arity(\mathtt{f}) \text{ and } \mathtt{p} \text{ contains } \mathtt{f\ x}_1\mathtt{x}_2 \ldots \mathtt{x}_i = \mathtt{e}^{\mathtt{f}})$$

Program execution:

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

Figure 3.1: FUNHO *semantics: program execution, expression evaluation.*

Definition 3.6 The FUNHO language has a closure-based call-by-value semantics. Expression evaluation is based on inference rules appearing in Figure 3.1, 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 expressions is $\mathtt{p}, env \vdash \mathtt{e} \rightarrow v$, signifying that expression e in program p evaluates to value v if its free variables have values defined by environment *env*. Environments, values, etc. are defined by

$$\begin{array}{rcl} env \in Env & = & \text{Parameter} \rightarrow Value \\ v \in Value & = & \mathbb{D} \mid Closure \\ cl \in Closure & = & \text{FcnName} \times Value^* \end{array}$$

A "closure" is a device to represent a data value which is a function.[3] In our named combinator syntax, a closure is a function name together with an incomplete parameter list, written $<\mathtt{f}, v_1 \ldots v_i>$.

The form to perform an application $\mathtt{e}_1\ \mathtt{e}_2$ is $\mathtt{p} \vdash^{call} env, cl \rightarrow v$, where expression $\mathtt{e}_1$ evaluates to a closure $cl = <\mathtt{f}, v_1 \ldots v_{i\perp 1}>$, and expression $\mathtt{e}_2$ evaluates to value v. First, the closure is extended by adding v to yield $cl' = <\mathtt{f}, v_1 \ldots v_{i\perp 1} v>$. If this application completes the call of f, i.e., $i = arity(\mathtt{f})$), then the body $\mathtt{e}^{\mathtt{f}}$ in f's definition is evaluated. If not, then cl' is returned as value of $\mathtt{e}_1\ \mathtt{e}_2$.

[3] In λ-calculus implementations a closure consists of a λ-abstraction together with an environment holding the values of all its free variables. The named combinator syntax allows a simpler form to be used.

3.4 Turing machines

By definition these take inputs from the set $\mathtt{TM}\text{-}data = \{0,1\}^*$. Since our aims concern only *decision powers* and not computation of functions, a Turing machine output will be a single bit 0 or 1. (Extension to outputs in $\{0,1\}^*$ is routine, by adding a one-way write-only output tape.)

A tape together with its scanning position will be written as $\mathtt{L}\,\underline{\mathtt{S}}\,\mathtt{R}$, where the underline indicates that S is the scanned symbol. As usual the tape is extensible — a new blank appears when a move is made beyond the end of the tape. A *store* σ describes the contents of the machine's one or several tapes, each with a designated scanned symbol. Instruction formats are described below.

A Turing machine program is a sequence $\mathtt{p} = \mathtt{1{:}I1}\ \mathtt{2{:}I2}\ \ldots\ \ \mathtt{m{:}Im}$. A *total state* is a pair $s = (\ell, \sigma)$ whose first component ℓ is the number of the instruction about to be executed ($m+1$ if it has completed execution) and whose second component is a store.

The semantics of each individual instruction is a state transition relation $s \to s'$, as is usual for Turing machines. Together these form define a function. A *computation* on input $\mathtt{d} \in \{0,1\}^*$ is a sequence $s_0 \to s_1 \to \ldots \to s_n$ where $s_i \to s_{i+1}$ for $0 \le i < n$, and s_0 is the initial state for input d, and s_n has instruction $m+1$ as first component (the "program end").

We define the *time usage* of a Turing machine program p on input d to be

$$time_{\mathtt{p}}^{\mathtt{TM}}(\mathtt{d}) = \begin{cases} n & \text{if } s_0 \to \ldots \to s_n \text{ is p's computation on d} \\ \bot & \text{if p does not terminate on input d} \end{cases}$$

Off-line Turing machines. An off-line Turing machine traditionally has one two-way *read-only input tape*, and one two-way *read-write work tape*. Off-line Turing machine instructions $\mathtt{I}_\ell$ are as follows, where subscript 1 indicates that the input tape 1 is involved; or 2 indicates that work tape 2 is involved. Instruction syntax:

Tape 1:	I	::=	$\mathtt{right}_1 \mid \mathtt{left}_1 \mid \mathtt{if}_1\ \mathtt{S\ goto}\ \ell\ \mathtt{else}\ \ell'$
Tape 2:	I	::=	$\mathtt{right}_2 \mid \mathtt{left}_2 \mid \mathtt{if}_2\ \mathtt{S\ goto}\ \ell\ \mathtt{else}\ \ell' \mid \mathtt{write}_2\ \mathtt{S}$
Symbols:	S	::=	$\mathtt{0} \mid \mathtt{1} \mid \mathtt{B}$
Strings:	L,R	::=	$\varepsilon \mid \mathtt{L\ S}$

A *store* $\sigma = (\mathtt{L}_1\underline{\mathtt{S}}_1\mathtt{R}_1, \mathtt{L}_2\underline{\mathtt{S}}_2\mathtt{R}_2)$ is a pair whose components describe both of the tapes; underlines mark their scanning positions. The *initial store* for input $\mathtt{d} \in \mathbb{D}$ is $\sigma_0 = (\underline{\mathtt{B}}\,\mathtt{d}, \underline{\mathtt{B}})$

Two examples of the state transition relation $s \to s'$, instruction $\mathtt{1{:}}\ \ \mathtt{right}_2$ causes transition from state $(1, \mathtt{B}\underline{\mathtt{1}}\mathtt{0}, \mathtt{B0}\underline{\mathtt{1}}\mathtt{11B})$ to $(2, \mathtt{B}\underline{\mathtt{1}}\mathtt{0}, \mathtt{B01}\underline{\mathtt{1}}\mathtt{1B})$, or from $(1, \mathtt{B}\underline{\mathtt{1}}\mathtt{0}, \mathtt{B0}\underline{\mathtt{1}})$ to $(2, \mathtt{B}\underline{\mathtt{1}}\mathtt{0}, \mathtt{B01}\underline{\mathtt{B}})$. We assume the program never moves right or left beyond a blank B, unless a nonblank symbol has first been written.[4]

The output for input d is defined by:

$$[\![\mathtt{p}]\!](\mathtt{d}) = \begin{cases} \bot & \text{if p has no computation on input d, else} \\ 1 & \text{if p's final state for input d scans worktape symbol 1} \\ 0 & \text{otherwise} \end{cases}$$

We define the *space usage* of a total state $s = (\ell, \mathtt{L}_1\underline{\mathtt{S}}_1\mathtt{R}_1, \mathtt{L}_2\underline{\mathtt{S}}_2\mathtt{R}_2)$ by $|s| = |\mathtt{L}_2\mathtt{S}_2\mathtt{R}_2|$, formally expressing that only the symbols on "work" tape 2 are counted, and not those on tape 1. Finally, we define

$$space_{\mathtt{p}}^{\mathtt{TM}}(\mathtt{d}) = \begin{cases} \max\{|s_i|\} & \text{if } s_0 \to \ldots \to s_n \text{ is p's computation on d} \\ \bot & \text{if p does not terminate on input d} \end{cases}$$

One-tape Turing machines. These are as above, except that only one tape is involved. This holds both input and the final output answer. Details are omitted because of brevity and probable familiarity.

[4] This condition simplifies constructions, and causes no loss of generality in computational power, or increase in time beyond a small constant factor.

3.5 Counter machines

Definition 3.7 A program in the language CM of *counter machines* is a sequence of labeled instructions p = 1:I1 2:I2 ... m:Im of the following forms for $0 < i$ and $0 \leq j$ (so counter C0 is never assigned).

$$\begin{array}{rcl} \mathtt{I} & ::= & \mathtt{Ci := Ci + 1} \mid \mathtt{Ci := Ci - 1} \mid \mathtt{Ci := Cj} \\ & \mid & \mathtt{if\ Cj{=}0\ goto}\ \ell\ \mathtt{else}\ \ell' \mid \mathtt{if\ In_{Cj} = 0\ goto}\ \ell\ \mathtt{else}\ \ell' \end{array}$$

A *total state* has form $(\mathtt{d}, \sigma) \in \{0,1\}^* \times (\mathbb{N} \to \mathbb{N})$ where d is the input data (read-only) and $\sigma(i)$ is the current contents of counter Ci for any $i \in \mathbb{N}$. Input to a counter machine is a string d in $\{0,1\}^*$. Data initialisation sets counter C0 to n, giving the program a way to "know" how long its input is. The counter values σ are initialised to zero except for C0: initially,

$$\sigma_0(\mathtt{d}) = [0 \mapsto |\mathtt{d}|, 1 \mapsto 0, 2 \mapsto 0, \ldots]$$

A state for input d has form $s = (\ell, (\mathtt{d}, \sigma))$, where ℓ is the instruction counter. Input access is by instruction $\mathtt{if\ In_{Ci} = 0\ goto}\ \ell\ \mathtt{else}\ \ell'$. Its effect: If $1 \leq \sigma(i) \leq n$ and $\mathtt{a}_{\sigma(i)} = 0$ then control is transferred to $\mathtt{I}_\ell$, else to $\mathtt{I}_{\ell'}$. The remaining instructions behave as expected from the syntax, except that we define $0 \perp 1 = 0$ to avoid negative integers.

Thus p defines a one-step transition relation $s \to s'$. A *computation* on input $\mathtt{d} \in \{0,1\}^*$ is a sequence $s_0 \to s_1 \to \ldots \to s_n$ where each s_i yields s_{i+1}, and s_0 equals $(1, (\mathtt{d}, \sigma_0(\mathtt{d})))$, and s_n has instruction m+1 as first component (the "program end"). The output for input d is defined by:

$$[\![\mathtt{p}]\!](\mathtt{d}) = \begin{cases} \perp & \text{if p has no computation on input d, else} \\ 1 & \text{if p's final state for input d has 1 in counter C1} \\ 0 & \text{otherwise} \end{cases}$$

Since our aims concern only *decision powers* and not computation of functions, a counter machine output will be a single bit 0 or 1.

We define the *space usage* of a total state $s = (\ell, \sigma)$ by $|s| = \max\{\sigma(0), \sigma(1), \sigma(2), \ldots\}$. This will always be finite. Finally, we define

$$value_{\mathtt{p}}^{\mathtt{CM}}(\mathtt{d}) = \begin{cases} \max\{|s_i|\} & \text{if } s_0 \to \ldots \to s_n \text{ is p's computation on d} \\ \perp & \text{if p does not terminate on input d} \end{cases}$$

A CM program p is $f(n)$-*bounded* if $value_{\mathtt{p}}^{\mathtt{CM}}(\mathtt{d}) \leq f(|\mathtt{d}|)$ for any input d.

3.6 Random access machines

This machine is an extension of the counter machine which more closely resembles current machine languages. Further, WHILE programcan be compiled into random access in a natural way.

The random access machine has a number of storage registers containing natural numbers (zero if uninitialised), and a much richer instruction set than the counter machine. The exact range of instructions allowed differ from one application to another, but nearly always include

1. Copying one register into another.
2. Indirect addressing or indexing, allowing a register whose number has been computed to be fetched from or stored into.
3. Elementary operations on one or more registers, for example adding or subtracting 1, and comparison with zero.

4. Other operations on one or more registers, for example addition, subtraction, multiplication, division, shifiting, or bitwise Boolean operations (where register contents are regarded as binary numbers, i.e. bit sequences).

The *successor random access machine*, SRAM, has only instruction types 1, 2, 3 above. General RAM operations vary within the literature. Although rather realistic in some aspects, the SRAM is, nonetheless, an idealised model with respect to actual machine codes. The reason is that there is no built-in limit to word size or memory address space: it has a potentially infinite number of storage registers, and each may contain an arbitrarily large natural number. Even though any one program can only address a constant number of storage registers directly, indirect addressing allows unboundedly many other registers to be accessed.

The following grammar describes the SRAM instruction syntax.

```
I  ::=  Xi := Xi + 1 | Xi := Xi ∸ 1 | if Xi=0 goto ℓ else ℓ′
     |  Xi := Xj | Xi := <Xj> | <Xi> := Xj
```

While this machine resembles the counter machine, it is more powerful in that it allows programs to fetch values from and store them into *cells with computed addresses*. The intuitive meaning of `Xi := <Xj>` is an *indirect fetch*: register `Xj`'s contents is some number n; and that the contents of register `Xn` are to be copied into register `Xi`. Similarly, the effect of `<Xi> := Xj` is an *indirect store*: register `Xi`'s contents is some number m; and the contents of register `Xj` are to be copied into register `Xm`.

3.7 The Church-Turing thesis: robustness of the concept of computability

A wide variety of computing devices and mathematical formalisms (recursive functions, the lambda calculus, and a great number of machine-like computation models including the Turing machine) have been devised to formalise the concept of "solvable problem" or "computable function."

Invariance of computational power. All these computing devices and mathematical formalisms have turned out to be equivalent in their problem-solving power, leading to the *Church-Turing thesis* that they all capture exactly the informal concept of "effective procedure." Conclusion: for many questions it doesn't really matter which computing formalism is used, so the most convenient for the problem at hand may be chosen.

Definition 3.8 Suppose L-*data* = M-*data*. Language L *can simulate* language M, written as: $\mathtt{L} \preceq \mathtt{M}$, if for every $\mathtt{p} \in$ L-*programs* there is an M-program q such that for all $\mathtt{d} \in$ L-*data* we have

$$[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}) = [\![\mathtt{q}]\!]^{\mathtt{M}}(\mathtt{d})$$

Language L *is equivalent to* language M, written $\mathtt{L} \equiv \mathtt{M}$, if language L and language M can simulate each other, i.e., if $\mathtt{L} \preceq \mathtt{M}$ and $\mathtt{M} \preceq \mathtt{L}$.

Theorem 3.9 WHILE≡GOTO

Traditionally Turing and many other machines take strings as inputs, in contrast to the binary trees accepted by WHILE programs. The following resolves the difference and makes it possible to extend the theorem just stated quite broadly.

A bijection between $\{0,1\}^*$ and a subset of $\mathbb{D}$. As defined, Turing machine inputs are in $\{0,1\}^*$ and WHILE inputs are in $\mathbb{D}$. We resolve this clash by restricting $\mathbb{D}$ to a subset in bijection with $\{0,1\}^*$. Define the coding $c : \{0,1\}^* \to \mathbb{D}$ by $c(\mathtt{a}_1\mathtt{a}_2...\mathtt{a}_n) = (\mathtt{a}'_1\ \mathtt{a}'_2\ \ldots\ \mathtt{a}'_n)$ where $\mathtt{0}' = \mathtt{nil}$ and $\mathtt{1}' = \mathtt{(nil.nil)}$. An example using Lisp list notation:

$$c(\mathtt{001}) = (\mathtt{0'0'1'}) = (\mathtt{nil\ nil\ (nil.nil)}) \in \mathbb{D}$$

Note that $2n+1 \leq |\ c(\mathtt{a}_1\mathtt{a}_2...\mathtt{a}_n)\ | \leq 4n+1$, so sizes differ only by a small constant factor.

Theorem 3.10 ("Robustness of computability") WHILE $\equiv$ GOTO $\equiv$ FUN1 $\equiv$ FUNHO $\equiv$ TM $\equiv$ CM $\equiv$ SRAM, aside from the bijection between $\{0,1\}^*$ and the range of c.

Proof. This is by a series of constructions showing how to simulate a program in one language by one in another language. Details can be found in [15] and [12].

4 Compilation and Interpretation

These concepts provide a natural bridge between the worlds of recursion theory and computer science. Briefly: most of the traditional computability theory constructions are compilations; and the well-known *universal function* is just the function computed by a "self-interpreter."

4.1 Compilation

Suppose we are given three programming languages: a *source language* S, a *target language* T, and an *implementation language* L, and that S-*data* = T-*data*, S-*programs* $\subseteq$ L-*data* and T-*programs* $\subseteq$ L-*data*.

Definition 4.1 An L-program comp is a *compiler* from language S to T if $[\![\mathtt{comp}]\!](\mathtt{p}) \in$ T-*programs* for every $\mathtt{p} \in$ S-*programs*, and for every $\mathtt{d} \in$ S-*data*,

$$[\![\mathtt{p}]\!]^{\mathtt{S}}(\mathtt{d}) = [\![[\![\mathtt{comp}]\!]^{\mathtt{L}}(\mathtt{p})]\!]^{\mathtt{T}}(\mathtt{d})$$

4.2 Interpretation

Self-interpreters, under the name *universal programs*, play (and have played since the 1930's) a central role in both complexity and computability theory. Generalising to several languages, an interpreter $\mathtt{int} \in$ L-programs for a language S has a pair of inputs: a *source program* $\mathtt{p} \in$ S-programs, and the source program's input data $\mathtt{d} \in$ S-*data*.

Definition 4.2 Suppose S-*programs* $\subseteq$ L-*data* = S-*data*. Program int is an *interpreter* for S *written in* L if for every $\mathtt{p} \in$ S-programs and for every $\mathtt{d} \in$ S-*data*,

$$[\![\mathtt{p}]\!]^{\mathtt{S}}(\mathtt{d}) = [\![\mathtt{int}]\!]^{\mathtt{L}}(\mathtt{p.d})$$

4.2.1 An interpreter i for 1-variable WHILE programs

We now give an example interpreter written in language WHILE. This is nearly a self-interpreter, except that for the sake of simplicity we restrict it to programs containing only one variable X. The interpreter, called i, is in Figure 4.1, where the STEP macro is in Figure 4.2. The interpreter will use the concrete syntax of Figure 2.1, together with unique codes to aid evaluation of cons, hd, tl and execution of assignments and while commands.

These are do_cons, do_hd, do_tl, do_asgn, do_while. They may be arbitrarily chosen from $\mathbb{D}$, except that they must be distinct from each other and from :=, ..., atom=? of the "concrete syntax" in Figure 2.1.

Theorem 4.3 `i` as defined in Figure 4.1 is an interpreter written in `WHILE` for the language $\texttt{WHILE}^{1var}$, which is identical to `WHILE` except that programs are restricted to one variable `X`.

```
read PD;                        (* Input is (program .  data) *)
  Cd := cons (hd PD) nil;       (* Control stack = (program.nil) *)
  Val := tl PD;                 (* The value of X = data *)
  Stk := nil;                   (* Computation stack is initially empty *)
  while Cd do STEP;             (* Repeat while control stack nonempty *)
write Val
```

Figure 4.1: Interpreter `i` *for* $\texttt{WHILE}^{1var}$

To aid compactness and readability, the `STEP` part of `i` is given by a set of transitions in Figure 4.2, i.e., rewrite rules `(Cd, s, v)` $\Rightarrow$ `(Cd', s', v')` describing transitions from one state of form

`(Cd, s, v)` = (*Code-stack, Computation-stack, Value*)

to the next state. Expression evaluation is based on the following *net effect property*. Suppose the value of expression `E` is `d`, provided that the current value of variable `X` is `v`. Then (`E.Cd`, `s`, `v`) $\Rightarrow^+$ (`Cd`, `d.s`, `v`) where $\Rightarrow^+$ means "can be rewritten in one or more steps to". A similar net effect property characterises execution of commands.

Blank entries in Figure 4.2 correspond to values that are neither referenced nor changed in a rule. The rules can easily be programmed in the `WHILE` language. For example, the three `while` transitions could be programmed as in Figure 4.3.

4.2.2 Self-interpretation of `WHILE` and `I`

We will call an interpreter for a language `L` which is written in the same language `L` a *self-interpreter* or *universal program* for `L`, and we will often use name `u` for a universal program. By Definition 4.2, `u` must satisfy $[\![\texttt{p}]\!]^{\texttt{L}}(\texttt{d}) = [\![\texttt{u}]\!]^{\texttt{L}}(\texttt{p.d})$ for every `L`-program `p` and `L`-data value `d`.

The interpreter `i` just given is *not* a self-interpreter due to the restriction to just one variable in the interpreted programs. Extension to a full self-interpreter for `WHILE` is a straightforward programming exercise.

Theorem 4.4 There exists a self-interpreter for the `WHILE` language.

4.2.3 The minimal language `I`.

Definition 4.5 The language `I` is identical to `WHILE`, with two exceptions. First, $A = \{\texttt{nil}\}$ so data values have only one atom. Second, programs have only one variable `X`.

The following result will be seen to have interesting complexity consequences.

Theorem 4.6 There exists a self-interpreter `u` for the `I` language.

Proof Interpreter `i` of Theorem 4.3 satisfies $[\![\texttt{i}]\!]^{\texttt{WHILE}}(\texttt{p.d}) = [\![\texttt{p}]\!]^{\texttt{I}}(\texttt{d})$, for any `I`-program `p` and input `d` $\in \mathbb{D}$, Program `i` uses only the one atom `nil`, but has several variables. Now obtain `I`-program `u` from `i` such that $[\![\texttt{u}]\!]^{\texttt{I}}(\texttt{p.d}) = [\![\texttt{p}]\!]^{\texttt{I}}(\texttt{d})$ by packing the several variables of `i` into one using "`cons`".

Cd	Stk	Val	⇒	Cd	Stk	Val
nil			⇒	nil		
(quote D).Cd	S		⇒	Cd	D.S	
(var nil).Cd	S	v	⇒	Cd	v.S	v
(hd E).Cd			⇒	E.do_hd.Cd		
do_hd.Cd	(T.U).S		⇒	Cd	T.S	
do_hd.Cd	nil.S		⇒	Cd	nil.S	
(tl E).Cd			⇒	E.do_tl.Cd		
do_tl.Cd	(T.U).S		⇒	Cd	U.S	
do_tl.Cd	nil.S		⇒	Cd	nil.S	
(cons E1 E2).Cd			⇒	E1.E2.do_cons.Cd		
do_cons.Cd	U.T.S		⇒	Cd	(T.U).S	
(; C1 C2).Cd			⇒	C1.C2.Cd		
(:= X E).Cd			⇒	E.do_asgn.Cd		
do_asgn.Cd	w.S	v	⇒	Cd	S	w
(if E C1 C2).Cd			⇒	E.do_if.C1.C2.Cd		
do_if.C1.C2.Cd	(T.U).S		⇒	C1.Cd	S	
do_if.C1.C2.Cd	nil.S		⇒	C2.Cd	S	
(while E C).Cd			⇒	E.do_while. (while E C).Cd		
do_while. (while E C).Cd	(T.U).S		⇒	C.(while E C).Cd	S	
do_while.C1.Cd	nil.S		⇒	Cd	S	

Figure 4.2: The STEP *macro, expressed by rewriting rules*

5 Applications of the *s-m-n* theorem

5.1 Partial evaluation is program specialisation

The program specialiser of Theorem 2.8 was very simple, and the programs it ouputs are slightly slower than the ones from which they were derived. (This was also true of Kleene's original construction.) On the other hand, program specialisation can be done so as to yield *efficient* specialised programs. This is known in the programming language community as *partial evaluation*; see [16] for a thorough treatment and a large bibliography.

Applications of program specialisation include *compiling* (done by specialising an interpreter to its source program), and *generating compilers from interpreters,* by using the specialiser to specialise itself to a given interpreter. Surprisingly, this can give quite efficient programs as output.

A simple but nontrivial example of partial evaluation Consider Ackermann's function, with program:

```
a(m,n) = if m =? 0 then n+1 else
         if n =? 0 then a(m-1,1)
         else a(m-1,a(m,n-1))
```

Computing `a(2,n)` involves recursive evaluations of `a(m,n)` for `m` = 0, 1 and 2, and various values of `n`. A partial evaluator can evaluate expressions `m=?0` and `m-1`, and function calls of form `a(m-1,...)` can be unfolded. We can now specialise function `a` to the values of `m`, yielding a less general program that is about twice as fast:

```
a2(n) = if n =? 0 then 3 else a1(a2(n-1))
a1(n) = if n =? 0 then 2 else a1(n-1)+1
```

```
if hd hd Cd = while then                    (* Set up iteration *)
   Cd := (cons (hd tl hd Cd) (cons do_while Cd)) else
if hd Cd = do_while then
   if hd Stk then                           (* Do body if test is true *)
   { Cd := cons (hd tl tl tl Cd) (tl Cd);
     Stk := tl Stk } else                   (* Else exit while *)
   { Stk := tl Stk; Cd := tl tl Cd } else ...
```

Figure 4.3: WHILE *code for rewrite rules to implement "*`while`*"*

5.2 Compiling and compiler generation by the Futamura projections

This section shows an application of the *s-m-n* theorem in computer science to compiling and, more generally, to generating program generators. For simplicity we elide the name of the language `L` that is the implementation, input, and output language of the specialiser.

The starting point is an interpreter program `int` for some programming language `S`. By Definition 4.2 $[\![\texttt{source}]\!]^{\texttt{S}}(\texttt{input}) = [\![\texttt{int}]\!](\texttt{source.input})$ for any `S`-program `source` and data $\texttt{input} \in \mathbb{D}$.

First Futamura projection: $\texttt{target} = [\![\texttt{spec}]\!](\texttt{int.source})$. This shows that *a specialiser can compile.* Correctness is to show that `target` is a program in the specialiser's output language which is equivalent to `S`-program `source`, i.e., $[\![\texttt{source}]\!]^{\texttt{S}} = [\![\texttt{target}]\!]$:

$$\begin{array}{lcll} [\![\texttt{source}]\!]^{\texttt{S}}(\texttt{input}) & = & [\![\texttt{int}]\!](\texttt{source.input}) & \text{Definition of interpreter} \\ & = & [\![[\![\texttt{spec}]\!](\texttt{int.source})]\!](\texttt{input}) & \text{Definition of specialiser} \\ & = & [\![\texttt{target}]\!](\texttt{input}) & \text{Definition of } \texttt{target} \end{array}$$

Second Futamura projection: $\texttt{comp} = [\![\texttt{spec}]\!](\texttt{spec.int})$. This shows that *a specialiser can generate a compiler.* Correctness is to show that `comp` is a compiler from `S` to the specialiser's output language. It constructs the just-mentioned target program from the source program:

$$\begin{array}{lcll} \texttt{target} & = & [\![\texttt{spec}]\!](\texttt{int.source}) & \text{Definition of target} \\ & = & [\![[\![\texttt{spec}]\!](\texttt{spec.int})]\!](\texttt{source}) & \text{Definition of specialiser} \\ & = & [\![\texttt{comp}]\!](\texttt{source}) & \text{Definition of } \texttt{comp} \end{array}$$

Third Futamura projection: $\texttt{cogen} = [\![\texttt{spec}]\!](\texttt{spec.spec})$. This shows that *a specialiser can generate a compiler generator.*[5] Correctness is to show that `cogen` constructs the just-mentioned compiler from the interpreter `int`:

$$\begin{array}{lcll} \texttt{comp} & = & [\![\texttt{spec}]\!](\texttt{spec.int}) & \text{Definition of comp} \\ & = & [\![[\![\texttt{spec}]\!](\texttt{spec.spec})]\!](\texttt{int}) & \text{Definition of specialiser} \\ & = & [\![\texttt{cogen}]\!](\texttt{int}) & \text{Definition of } \texttt{cogen} \end{array}$$

Perhaps the most surprising thing about this equational reasoning is that it also works well in practice: A variety of partial evaluators (= program specialisers = programs of *s-m-n* functions) have been constructed, and they give good results in practical applications [16].

[5] A compiler generator transforms an interpreter into a compiler.

Speedups from self-application.

Each of program execution, compilation, compiler generation, and compiler generator generation can be done in two different ways:

$$\begin{array}{lcccl} \texttt{output} & = & [\![\texttt{int}]\!](\texttt{source.input}) & = & [\![\texttt{target}]\!]^{\texttt{S}}(\texttt{input}) \\ \texttt{target} & = & [\![\texttt{spec}]\!](\texttt{int.source}) & = & [\![\texttt{comp}]\!](\texttt{source}) \\ \texttt{comp} & = & [\![\texttt{spec}]\!](\texttt{spec.int}) & = & [\![\texttt{cogen}]\!](\texttt{int}) \\ \texttt{cogen} & = & [\![\texttt{spec}]\!](\texttt{spec.spec}) & = & [\![\texttt{cogen}]\!](\texttt{spec}) \end{array}$$

The exact timings vary according to the design of `spec` and `int`, and with the implementation language `L`. Nonetheless, we have observed in practical computer experiments that *in each case the rightmost run is often about 10 times faster than the leftmost.* Moral: self-application can generate programs that run faster!

6 Gödel's incompleteness theorem

Gödel's theorem is often described (a bit loosely) as "any proof system of a certain minimal complexity is either incomplete or inconsistent." We sidestep the problem of dealing with the multiplicity of concrete and complex proof systems known from mathematical logic by generalisation to an "inference system." n example inference system has already been seen: the `FUNHO` semantics of Figure 3.1.

We then introduce a tiny logical language DL, in which one can make statements about values in $\mathbb{D}$, and prove that *no inference system can generate exactly the set of true statements in* DL. This implies that any inference system which only allows true statements of DL to be proven cannot generate all true statements of DL. In other words: any DL-*consistent* inference system must be DL-*incomplete*.

Conclusion: "the full truth" of DL statements cannot be ascertained by means of axioms and rules of logical deduction.

6.1 Inference systems

Informally, an inference rule R is usually presented in tree form:

$$\frac{P_1(\ldots) \quad \ldots \quad P_k(\ldots)}{P(\ldots)} \quad \text{(rule } R\text{)}$$

with k *premises* $P_1(\ldots)$,..., $P_k(\ldots)$ and one *conclusion* $P(\ldots)$. A rule with no premises ($k = 0$) is called an *axiom*.

Informal example: Let the "append" predicate $A(\texttt{xs ys zs})$, be true for `xs`, `ys`, `zs`$\in \mathbb{D}$ if $\texttt{xs} = (\texttt{x}_1 \ldots \texttt{x}_m), \texttt{ys} = (\texttt{y}_1 \ldots \texttt{y}_n)$ and $\texttt{zs} = (\texttt{x}_1 \ldots \texttt{x}_m \texttt{y}_1 \ldots \texttt{y}_n)$. An Inference system defining this ternary relation on $\mathbb{D}$ has rules:

$$\frac{}{A(\texttt{nil ys ys})} \quad \text{(rule } R_{axiom}\text{)} \qquad\qquad \frac{A(\texttt{xs xs ys})}{A(\texttt{(x.xs) ys (x.zs)})} \quad \text{(rule } R_{step}\text{)}$$

Definition 6.1 An *inference system* $\mathcal{I}$ consists of

1. A finite set $\{P, Q, \ldots, Z\}$ of *predicate names*, and another finite set $\{R_1, R_2, \ldots, R_m\}$ of *inference rules*.

2. For each inference rule R_r, a *type*: $T_r : P_1 \times \ldots \times P_k \to P$ where $P, P_1, \ldots, P_k$ are predicate names and $k \geq 0$.

3. For each inference rule R_r of type $P_1 \times \ldots \times P_k \to P$, a decidable *inference relation*: $I_r \subseteq \mathbb{D}^k \times \mathbb{D}$.

Each inference rule R_r has $k = 0, 1$, or more premises and one comclusion, a rule with 0 premises being called an *axiom*. Inference relation I_r .

Example 6.2 (Inference system for the "append" predicate $A(\mathtt{xs}, \mathtt{xs}, \mathtt{ys})$ described informally above. Following the formalism of Definition 6.1, this can be defined by:

$$\mathcal{I} = (\{A\}, \{R_{axiom}, R_{step}\}, T_{axiom} :\to A, T_{step} : A \to A, I_{axiom} \subseteq \mathbb{D}, I_{step} \subseteq \mathbb{D} \times \mathbb{D})$$

$T_{axiom} :\to A$ indicates that rule R_{axiom} has no premises and an element of predicate A as conclusion. The actual instances of the axiom are given by:

$$I_{axiom} = \{((), (\mathtt{nil\ ys\ ys})) \mid \mathtt{ys} \in \mathbb{D}\}$$

Similarly $T_{step} : A \to A$ says rule R_{step} has one A-premise and one A-conclusion, with instances given by:

$$I_{step} = \{((\mathtt{xs\ ys\ zs}), ((\mathtt{x.xs})\ \mathtt{ys}\ (\mathtt{x.zs}))) \mid \mathtt{x}, \mathtt{xs}, \mathtt{ys}, \mathtt{zs} \in \mathbb{D}\}$$

Definition 6.3 A *proof tree* of an inference system $\mathcal{I}$ is a finite tree, defined as usual: the $n \geq 0$ parents of any proof tree node must be related to their child by one of the inference rules. The set $Thms_P^{\mathcal{I}}$ of all theorems of form $P(\mathtt{d})$, where P is a predicate name, is defined by

$$Thms_P^{\mathcal{I}} = \{\mathtt{d} \mid \mathcal{I} \text{ has a proof tree with root } P(\mathtt{d})\}$$

Theorem 6.4 $Thms_P^{\mathcal{I}}$ is a recursively enumerable set for any inference system $\mathcal{I}$ and predicate name P.

6.2 The logical language DL for $\mathbb{D}$

An abstract syntax of DL is given by a grammar defining *terms*, which stand for values in $\mathbb{D}$, and *statements*, which are assertions about relationships among terms.

Terms:	$\mathtt{T ::= nil} \mid \mathtt{(T.T)} \mid x_0 \mid x_1 \mid \ldots$
Statements:	$\mathtt{S ::= T=T{+}{+}T} \mid \neg\ \mathtt{S} \mid \mathtt{S} \wedge \mathtt{S} \mid \mathtt{S} \vee \mathtt{S} \mid \exists x_i\ \mathtt{S} \mid \forall x_i\ \mathtt{S}$

The symbol ++ stands for the "append" operation on list values. Logical operators $\vee, \Rightarrow$, etc. can be defined as usual, and equality $\mathtt{T} = \mathtt{T}'$ is syntactic sugar for $\mathtt{T} = \mathtt{T}'\mathtt{{+}{+}nil}$. Statements are interpreted in the natural way, for example the relation "x is a sublist of y" could be expressed by:

$$\exists u \exists v \exists w (y = w\mathtt{{+}{+}}v \wedge w = u\mathtt{{+}{+}}x)$$

Definition 6.5 $\mathcal{T}$ is the set of *true closed statements* of DL[6]

Definition 6.6 A predicate $P \subseteq \mathbb{D}^n$ is *representable in* DL if there is a statement $\mathtt{S}(x_1, \ldots, x_n)$ such that

$$P = \{(\mathtt{d}_1, \ldots, \mathtt{d}_n) \in \mathbb{D}^n \mid \mathit{Substitute}(\mathtt{S}, (x_1, \ldots, x_n), (\mathtt{d}_1, \ldots, \mathtt{d}_n)) \in \mathcal{T}\}$$

[6]For technical reasons we assume that the DL statements in $\mathcal{T}$ are encoded as elements of $\mathbb{D}$. Let nil, cons, append, not, and, or, exists, forall be distinct elements in $\mathbb{D}$, and encode variable x_i as (var nil^i). Then, for example, statement $\forall x_0(x_0 = \mathtt{nil{+}{+}}x_0)$ might be encoded as (forall (var nil) (append (var nil) nil (var nil)))

Lemma 6.7 If set $A \subseteq \mathbb{D}^n$ is representable in DL, then so is $\overline{A} = \mathbb{D}^n \setminus A$.

Proof. Suppose DL statement $\mathtt{S}(x_1, \ldots, x_n)$ represent set $A \subseteq \mathbb{D}^n$. Then statement $\neg\, \mathtt{S}(x_1, \ldots, x_n)$ represents $\overline{A} = \mathbb{D}^n \setminus A$.

Theorem 6.8 For any I-program p, the set domain($[\![\mathtt{p}]\!]$) is representable in DL

Proof. We will show that for each I expression E and command C, there exist DL-statements $\mathtt{F_E}(d, d')$ and $\mathtt{G_C}(d, d')$ that represent the binary predicates $\mathcal{E}[\![\mathtt{E}]\!]\mathtt{d} = \mathtt{d}'$ and $\mathtt{C} \vdash \mathtt{d} \to \mathtt{d}'$. This suffices since if p is `read X; C; write X`, then

$$\text{domain}([\![\mathtt{p}]\!]) \text{ is represented by DL statement } S(\mathtt{d}) \equiv \exists d' \mathtt{G_C}(d, d')$$

Expressions. This is by an easy induction on syntax:

$$\begin{array}{lcl}
\mathtt{F_{nil}}(d, d') & \equiv & d' = \mathtt{nil} \\
\mathtt{F_X}(d, d') & \equiv & d' = d \\
\mathtt{F_{cons\,E1\,E2}}(d, d') & \equiv & \exists r \exists s \quad \mathtt{F_{E1}}(d, r) \wedge \mathtt{F_{E2}}(d, s) \wedge d' = (r.s) \\
\mathtt{F_{hd\,E}}(d, d') & \equiv & \exists r \exists s \quad \mathtt{F_E}(d, r) \wedge r = (d'.s) \vee r = \mathtt{nil} \wedge d' = \mathtt{nil} \\
\mathtt{F_{tl\,E}}(d, d') & \equiv & \exists r \exists s \quad \mathtt{F_E}(d, r) \wedge r = (s.d') \vee r = \mathtt{nil} \wedge d' = \mathtt{nil} \\
\mathtt{F_{atom=?\,E1\,E2}}(d, d') & \equiv & (d' = \mathtt{nil} \wedge \mathtt{F_{E1}}(d, \mathtt{nil}) \wedge \mathtt{F_{E2}}(d, \mathtt{nil})) \vee \\
 & & (d' = (\mathtt{nil}.\mathtt{nil}) \wedge \neg(\mathtt{F_{E1}}(d, \mathtt{nil}) \wedge \mathtt{F_{E2}}(d, \mathtt{nil})))
\end{array}$$

Commands. This is also by induction on syntax:

$$\begin{array}{lcl}
\mathtt{G_{X:=E}}(d, d') & \equiv & \mathtt{F_E}(d, d') \\
\mathtt{G_{C1;C2}}(d, d') & \equiv & \exists d''(\mathtt{G_{C1}}(d, d'') \wedge \mathtt{G_{C2}}(d'', d')) \\
\mathtt{G_{while\,E\,do\,C}}(d, d') & \equiv & \exists trace\, \exists fst\, \exists last\, (trace = (d.last) \wedge trace = fst{+}{+}(d'.\mathtt{nil}) \wedge \mathtt{F_E}(d', \mathtt{nil}) \wedge \\
 & & \quad \forall h \forall u \forall v \forall t\, (trace = h{+}{+}(u.(v.t)) \Rightarrow \\
 & & \quad\quad \mathtt{G_C}(u, v) \wedge \exists e \exists f\, (\mathtt{F_E}(u, (e.f)))))
\end{array}$$

Assignment is straightforward, and sequencing $\mathtt{C1;C2} \vdash \mathtt{d} \to \mathtt{d}'$ is represented naturally by an intermediate state $\mathtt{d}''$.

Representation of command `while E do C` is a bit trickier, since its execution may take an unbounded number of steps. The idea is to represent $\mathtt{while\ E\ do\ C} \vdash \mathtt{d} \to \mathtt{d}'$ by a *computation trace*: a sequence $(\mathtt{d}_1 \ldots \mathtt{d}_n)$ where $\mathtt{d} = \mathtt{d}_1, \mathtt{d}' = \mathtt{d}_n$, and $\mathtt{C} \vdash \mathtt{d}_i \to \mathtt{d}_{i+1}$ for $i = 1, 2, \ldots, n \perp 1$.

The construction above uses this idea. The two parts concerning *fst* and *last* ensure that the trace properly begins with $d = \mathtt{d}_1$ and ends with $d' = \mathtt{d}_n$. The remaining part (beginning $\mathtt{F_E}(d', nil)$) checks to see that E evaluates to `false` at the loop's end ($\mathtt{d}' = \mathtt{d}_n$), and that $\mathtt{while\ E\ do\ C} \vdash \mathtt{d}_i \to \mathtt{d}_{i+1}$ holds for every pair $\mathtt{d}_i, \mathtt{d}_{i+1}$ in the trace.

6.3 Proof of Gödel's incompleteness theorem

Theorem 6.9 $\mathcal{T}$ is not recursively enumerable.

Proof. Consider the set $HALT = \text{domain}([\![\mathtt{u}]\!])$ where u is the universal program (self-interpreter) for I programs. By Theorem 6.8 it is representable in DL. By Lemma 6.7, $\overline{HALT}$ is representable by some DL-statement $\mathtt{F}(x)$, so

$$\overline{HALT} = \{\mathtt{v} \mid \mathit{Substitute}(\mathtt{F}, x, \mathtt{v}) \in \mathcal{T}\}$$

Suppose $\mathcal{T}$ were recursively enumerable. Then there must exist a program q such that $\mathcal{T} = \text{domain}([\![\mathtt{q}]\!])$. But then for any input $\mathtt{v} \in \mathbb{D}$, we have

$\mathtt{v} \in \overline{HALT}$ iff $[\![\mathtt{q}]\!](\mathit{Substitute}(\mathtt{F}, x, \mathtt{v})) \neq \perp$

This would imply that $\overline{HALT}$ is recursively enumerable, which is false by Theorem 2.6.

Corollary 6.10 For any inference system $\mathcal{I}$ and predicate name P:

$$\text{If } \mathit{Thms}_P^{\mathcal{I}} \subseteq \mathcal{T} \text{ then } \mathit{Thms}_P^{\mathcal{I}} \neq \mathcal{T}$$

In effect this says that if any inference system proves only true DL statements, then it cannot prove all of them. In other words there is and always will be a difference between *truth* (at least for DL) and *provability* by inference systems. This captures one essential aspect of Gödel's incompleteness theorem. In comparison with the original proof, and others seen in the literature, this one uses surprisingly little technical machinery (though it admittedly builds on the nontrivial difference between recursive and recursively enumerable sets).

Gödel's original work began with a logical system containing Peano arithmetic, and applied diagonalisation to construct a witness: an example of a statement `S` which is true, but which cannot be provable. Gödel's original witness is (intuitively) true since it in effect asserts "there is no proof in this system of `S`." Our version indeed uses diagonalisation, but on `I` programs instead, and to prove that the problem $\overline{HALT}$ is not recursively enumerable.

7 Constant time factors *do* matter

The *Turing machine constant speedup theorem* is traditionally one of the first learned in complexity courses. In effect it says that from any Turing machine program, one may construct an equivalent one that runs twice as fast (asymptotically, and if its running time is superlinear).

Unfortunately, this result gives practically oriented students a bad impression: While the proof is not too complex, what it says is extremely counterintuitive, going against daily programming experience. Further, the construction is useless for speeding up real programs, as it in essence involves changing to a double-density tape. The fact that the theorem is nonetheless taught tends to convey the idea that theory is irrelevant to practice.

The truth of this speedup theorem is an immediate consequence of using an "unfair time measure": one Turing machine state transition is counted as taking one time unit, *regardless of the size of the Turing machine's tape alphabet.*

We show a more satisfying result (at least from a programmer's perspective): a proper hierarchy exists among problems that can be solved by `I` programs in linear time. In particular there exist constants $0 < c < d$ and set A such that the question $\mathtt{x} \in A$? can be decided in time $c \cdot |\mathtt{x}|$ but not in time $d \cdot |\mathtt{x}|$, *regardless of how clever one is* at programming and/or algorithm design. Such an absolute result is rare in computer science, and attracts the students' attention.

This result is false for Turing machines; and its status for the full `WHILE` language is an open question.

7.1 Time-bounded complexity classes

Definition 7.1 Given programming language `L` and a function : $\mathbb{N} \to \mathbb{N}$, we define

$$\begin{array}{lcl} \text{TIME}^{\mathtt{L}}(f) & = & \{A \subseteq \mathbb{D} \mid \begin{array}{l} A \text{ is decided by some } \mathtt{L}\text{-program } \mathtt{p}, \text{ such that} \\ \mathit{time}_{\mathtt{p}}^{\mathtt{L}}(\mathtt{d}) \leq f(|\mathtt{d}|) \text{ for all } \mathtt{d} \in \mathtt{L}\text{-}\mathit{data}\} \end{array} \end{array}$$

Naming convention: the size of the input, e.g., $|\mathtt{d}|$, is called n. If exp is an expression containing n, we write $\text{TIME}^{\mathtt{L}}(exp)$ anstead of $\text{TIME}^{\mathtt{L}}(\lambda n.exp)$.

Theorem 7.2 *Linear-time hierarchy for* `I`*:* There is a b such that for all $a \geq 1$ there is a set $A \subseteq \mathbb{D}$ which is in $\text{TIME}^{\mathtt{I}}(a \cdot b \cdot n)$, but not in $\text{TIME}^{\mathtt{I}}(a \cdot n)$.

The key to this result is the existence of an "efficient" interpreter for I, as seen in Definition 7.3. The proof diagonalises (as for undecidability of the halting problem), using a time-bounded extension of the self-interpreter of Theorem 4.6 to stay within the required time bounds. The current framework makes this substantially simpler than traditional time-hierarchy proofs.

7.2 Interpretation overhead and "efficiency"

Let int be an interpreter for S written in L. Assuming both an L-machine and an S-machine are at one's disposal, interpretation is usually rather slower than direct execution of S-programs. In practice, an interpreter's running time on inputs p and d typically satisfies

$$time^{\mathtt{L}}_{\mathtt{int}}(\mathtt{p.d}) \leq a_{\mathtt{p}} \cdot time^{\mathtt{S}}_{\mathtt{p}}(\mathtt{d})$$

for all d. Here $a_{\mathtt{p}}$ is a "constant" independent of d, but it may depend on the source program p. Often $a_{\mathtt{p}} \doteq c + f(\mathtt{p})$, where constant c represents the time taken for "dispatch on syntax" and recursive calls of the evaluation or command execution functions; and $f(\mathtt{p})$ represents the time for variable access.

Definition 7.3 An interpreter int (for S written in L) is *efficient* if there is a constant a such that for all $\mathtt{d} \in \mathbb{D}$ and every S-program p

$$time^{\mathtt{L}}_{\mathtt{int}}(\mathtt{p.d}) \leq a \cdot time^{\mathtt{S}}_{\mathtt{p}}(\mathtt{d})$$

Constant a is here quantified *before* p, so the slowdown caused by an efficient interpreter is independent of p.

Theorem 7.4 The interpreter u for I written in I from Theorem 4.6 is efficient according to Definition 7.3.

7.3 An efficient timed universal program

Definition 7.5 An I-program tu is an *efficient timed universal program* if there is a constant k such that for all $\mathtt{p} \in$ I-*programs*, $\mathtt{d} \in \mathbb{D}$ and $n \geq 1$:

1. If $time_{\mathtt{p}}(\mathtt{d}) \leq n$ then $[\![\mathtt{tu}]\!](\mathtt{p\ d\ nil}^n) = ([\![\mathtt{p}]\!](\mathtt{d}).\mathtt{nil})$
2. If $time_{\mathtt{p}}(\mathtt{d}) > n$ then $[\![\mathtt{tu}]\!](\mathtt{p\ d\ nil}^n) = \mathtt{nil}$
3. $time_{\mathtt{tu}}(\mathtt{p\ d\ nil}^n) \leq k \cdot \min(n, time_{\mathtt{p}}(\mathtt{d}))$.

The effect of $[\![\mathtt{tu}]\!](\mathtt{p\ d\ nil}^n)$ is to simulate p for $\min(n, time_{\mathtt{p}}(\mathtt{d}))$ steps. If $time_{\mathtt{p}}(\mathtt{d}) \leq n$, that is, p terminates within n steps, then tu produces a non-nil value containing p's result. If not, the value nil is yielded, indicating "time limit exceeded."

Theorem 7.6 There exists an efficient timed universal program tu.

Proof. We first construct an efficient timed interpreter tt for I, in the form of a WHILE-program. The idea is to take the interpreter seen before for one-variable WHILE programs, and add some extra code and an extra input, a *time bound* of the form $\mathtt{nil}^n$ stored in a variable Cntr, so obtaining a program tt.

Every time the simulation of one operation of program input p on data input d is completed, the time bound is decreased by 1. Note the use of the atom encoding function from the proof of Theorem 4.6. Here is tt:

```
read X;                          (* X = (p d nil^n) *)
Cd := cons (hd X) nil;           (* Code to be executed *)
Val := hd (tl X);                (* Initial value of simulated X *)
Cntr := hd (tl (tl X));          (* Time bound *)
Stk := nil;                      (* Computation stack *)
while Cd do
  if Cntr
  then {  if hd (hd Cd) ∈ {quote, var, do_hd, do_tl, do_cons, do_asgn, do_while}
          then Cntr := tl Cntr;
          STEP; X := cons Val nil;}
  else {  Cd := nil; X := nil};
write X
```

Finally, obtain I-program `tu` from `tt` by packing its several variables into one using "`cons`".

7.4 The linear-time hierarchy

This result shows there is a constant b such that for any $a \geq 1$ there is a decision problem which cannot be solved by any I-program that runs in time bounded by $a \cdot n$, *regardless of how clever* one is at programming, or at problem analysis, or both. On the other hand, the problem *can* be solved by an I-program in time $a \cdot b \cdot n$ on inputs of size n. Consequence: given a, there exists an infinite hierarchy

$$\text{TIME}^{\text{I}}(a \cdot n) \subset \text{TIME}^{\text{I}}(a \cdot b \cdot n) \subset \text{TIME}^{\text{I}}(a \cdot b^2 \cdot n) \subset \ldots$$

Proof. of Theorem 7.2. First define program `diag` by Figure 7.1.

```
read X;
Timebound := nil^{a·|X|};
Arg := cons X (cons X (cons Timebound nil));
X := tu Arg;    (* Use tu to run X on X for up to a·|X| steps *)
if hd X then X := false else X := true;
write X
```

Figure 7.1: Diagonalisation program `diag`.

Claim: the set $A = \{\mathtt{d} \mid [\![\mathtt{diag}]\!]^{\text{I}}(\mathtt{d}) = \mathtt{true}\}$ is in $\text{TIME}^{\text{I}}(a \cdot b \cdot n)$ for an appropriate b, but is not in $\text{TIME}^{\text{I}}(a \cdot n)$. Further, b is independent of a.

We first analyse the running time of program `diag` on input `p`. Since a is fixed, $\mathtt{nil}^{a \cdot |\mathtt{d}|}$ can be computed in time $c \cdot a \cdot |\mathtt{d}|$ for some c and any `d`. From Definition 7.5, there exists k such that the timed universal program `tu` of Theorem 7.6 runs in time $time_{\mathtt{tu}}(\mathtt{p\ d\ nil}^n) \leq k \cdot min(n, time_{\mathtt{p}}(\mathtt{d}))$. Thus on input `p`, the command "`X := tu Arg`" takes time at most

$$k \cdot min(a \cdot |\mathtt{p}|, time_{\mathtt{p}}(\mathtt{p})) \leq k \cdot a \cdot |\mathtt{p}|$$

so program `diag` runs in time at most

$$c \cdot a \cdot |\mathtt{p}| + k \cdot a \cdot |\mathtt{p}| + e$$

where c is the constant factor used to compute $a \cdot |\mathtt{X}|$, k is from the timed universal program, and e accounts for the time beyond computing `Timebound` and running `tu`. Now $|\mathtt{p}| \geq 1$ so

$$c \cdot a \cdot |\mathtt{p}| + k \cdot a \cdot |\mathtt{p}| + e \leq a \cdot (c + k + e) \cdot |\mathtt{p}|$$

which implies that $A \in \text{TIME}^{\text{I}}(a \cdot b \cdot n)$ with $b = c + k + e$.

Now suppose for the sake of contradiction that $A \in \text{TIME}^{\text{I}}(a \cdot n)$. Then there exists a program $\mathtt{p}$ which also decides membership in A, and does it quickly, satisfying $time_{\mathtt{p}}(\mathtt{d}) \leq a \cdot |\mathtt{d}|$ for all $\mathtt{d} \in \mathbb{D}$. Consider cases of $[\![\mathtt{p}]\!](\mathtt{p})$. Then $time_{\mathtt{p}}(\mathtt{p}) \leq a \cdot |\mathtt{p}|$ implies that $\mathtt{tu}$ has sufficient time to simulate $\mathtt{p}$ to completion on input $\mathtt{p}$. By Definition 7.5, this implies

$$[\![\mathtt{tu}]\!](\mathtt{p}\ \mathtt{p}\ \mathtt{nil}^{a \cdot |\mathtt{p}|}) = ([\![\mathtt{p}]\!](\mathtt{p})\ \mathtt{nil})$$

If $[\![\mathtt{p}]\!](\mathtt{p})$ is $\mathtt{false}$, then $[\![\mathtt{diag}]\!](\mathtt{p}) = \mathtt{true}$ by construction of $\mathtt{diag}$. If $[\![\mathtt{p}]\!](\mathtt{p})$ is $\mathtt{true}$, then $[\![\mathtt{diag}]\!](\mathtt{p}) = \mathtt{false}$. Both cases contradict the assumption that $\mathtt{p}$ and $\mathtt{diag}$ both decide membership in A. The only unjustified assumption was that $A \in \text{TIME}^{\text{I}}(a \cdot n)$, so this must be false, completing the proof.

This construction has been carried out in detail on the computer by Ben-Amram, who established a stronger result in [5]: that $\text{TIME}^{\text{I}}(232 \cdot a \cdot n)$ properly includes $\text{TIME}^{\text{I}}(a \cdot n)$ for any $a > 1$.

Further, for any non-zero "time constructible" $T(n)$ (using a natural definition), there is a b such that $\text{TIME}^{\text{I}}(b \cdot T(n)) \setminus \text{TIME}^{\text{I}}(T(n))$ is non-empty.

8 Levin's optimal search theorem

A great many familiar problems are searches. Consider the predicate

$$R(\mathcal{F}, \theta) \equiv \text{ truth assignment } \theta \text{ makes formula } \mathcal{F} \text{ true}$$

The problem to find θ (if it exists) when given only $\mathcal{F}$ is the familar and apparently intractable *satisfiability problem*. As is well known, it is much easier to check truth of $R(\mathcal{F}, \theta)$.

Definition 8.1 A *witness function* for a binary predicate $R \subseteq \mathbb{D} \times \mathbb{D}$ is a partial function $f : \mathbb{D} \to \mathbb{D}_{\perp}$ such that:

$$\forall \mathtt{x}(\exists \mathtt{y} \,.\, R(\mathtt{x}, \mathtt{y})) \Rightarrow R(\mathtt{x}, f(\mathtt{x}))$$

A *brute-force search* program for finding a witness immediately comes to mind. Given $x \in \mathbb{D}$ we just enumerate elements $\mathtt{y} \in \mathbb{D}$, checking one after the other until a witness pair $(\mathtt{x}, \mathtt{y}) \in R$ has been found.[7] It is quite obvious that this strategy can yield an extremely inefficient program, since it may waste a lot of time on wrong candidates until it finds a witness. Levin's theorem states a surprising fact: for many interesting problems there is another brute-force search strategy that not only is efficient, but *optimal* up to constant factors. The difference is that Levin's strategy generates and tests not *solutions*, but *programs*.

Theorem 8.2 *Levin's Search Theorem.* Let $R \subseteq \mathbb{D} \times \mathbb{D}$ be a recursively enumerable binary predicate, so $R = \text{dom}([\![\mathtt{r}]\!])$ for some program $\mathtt{r}$. Then there is a `WHILE` program $\mathtt{opt}$ such that $f = [\![\mathtt{opt}]\!]$ is a witness function for R. Further, let $\mathtt{q}$ be *any* program that computes a witness function f for R. Then for all $\mathtt{x}$ such that $(\mathtt{x}, \mathtt{y}) \in R$ for some $\mathtt{y}$:

$$time_{\mathtt{opt}}(\mathtt{x}) \leq a_{\mathtt{q}}(time_{\mathtt{q}}(\mathtt{x}) + time_{\mathtt{r}}(\mathtt{x}.f(\mathtt{x})))$$

where $a_{\mathtt{q}}$ is a constant that depends on $\mathtt{q}$ but not on $\mathtt{x}$. Further, the program $\mathtt{opt}$ can be effectively obtained from $\mathtt{r}$.

[7] If R is decidable, this is straightforward. If semi-decidable but not decidable, a "dovetailing" of computations can be used to test $(\mathtt{x}, \mathtt{d}_0) \in R?$, $(\mathtt{x}, \mathtt{d}_1) \in R?$, ... in parallel for all values $\mathtt{d}_0, \mathtt{d}_1, \mathtt{d}_2, \ldots \in \mathbb{D}$.

Sketch of proof of Levin's theorem. Without loss of generality we assume that when program $\mathtt{r}$ is run with input $\mathtt{(x.y)}$, if $(\mathtt{x},\mathtt{y}) \in R$ it gives $\mathtt{y}$ as output. Otherwise, it loops forever. Enumerate $\mathbb{D} = \{\mathtt{d}_0, \mathtt{d}_1, \mathtt{d}_2, \ldots\}$ effectively (it can be done in constant time per new element). Build program $\mathtt{opt}$ to compute as follows:

1. A "main loop" to generate all finite trees. At each iteration one new tree is added to list $\mathtt{L} = \mathtt{(d}_n \ldots \mathtt{d}_1\mathtt{d}_0\mathtt{)}$. Tree $\mathtt{d}_n$ for $n = 0, 1, 2, \ldots$ will be treated as the command part of the n-th I-program $\mathtt{p}_n$.

2. Iteration n will process programs $\mathtt{p}_k$ for $k = n, n \perp 1, \ldots, 1, 0$ as follows:

   (a) Run $\mathtt{p}_k$ on input $\mathtt{x}$ for a "time budget" of $2^{n \perp k}$ steps.

   (b) If $\mathtt{p}_k$ stops on $\mathtt{x}$ with output $\mathtt{y}$, then run $\mathtt{r}$ on input $\mathtt{(x.y)}$, so $\mathtt{p}_k$ and $\mathtt{r}$ together have been executed for at most $2^{n \perp k}$ steps.

   (c) If $\mathtt{p}_k$ or $\mathtt{r}$ failed to stop, then replace k by $k \perp 1$, double the time budget to $2^{n \perp k+1}$ steps, and reiterate.

3. If running $\mathtt{p}_k$ followed by $\mathtt{r}$ terminates within time budget $2^{n \perp k}$, then output $[\![\mathtt{opt}]\!](\mathtt{x}) = \mathtt{y}$ and stop; else continue with iteration $n + 1$.

Thus the programs are being interpreted concurrently, every one receiving some "interpretation effort." We stop once any one of these programs has both *solved our problem and been checked*, within its given time bounds. Note that $\mathtt{opt}$ will loop in case no witness is found. The following table showing the time budgets of the various runs may aid the reader in following the flow of the construction and seeing its correctness.

The keys to "optimality" of $\mathtt{opt}$ are the efficiency of the self-interpreter STEP operation, plus a policy of allocating time to the concurrent simulations so that the total time will not exceed, by more than a constant factor, the time of the program that finishes first.

Time budget	$\mathtt{p}_0$	$\mathtt{p}_1$	$\mathtt{p}_2$	$\mathtt{p}_3$	$\mathtt{p}_4$	$\mathtt{p}_5$	...
$n = 0$	1	-	-	-	-	-	...
$n = 1$	2	1	-	-	-	-	...
$n = 2$	4	2	1	-	-	-	...
$n = 3$	8	4	2	1	-	-	...
$n = 4$	16	8	4	2	1	-	...
$n = 5$	32	16	8	4	2	1	...
$n = 6$	64	32	16	8	4	2	...
...	...						

Suppose $\mathtt{q} = \mathtt{p}_k$ computes a witness function f. At iteration n, program $\mathtt{p}_k$ and the checker $\mathtt{r}$ are run for $2^{n \perp k}$ steps. Therefore (assuming $R(\mathtt{x}.f(\mathtt{x}))$ is true) the process above will not continue beyond iteration n, where

$$2^{n \perp k} \geq time_{\mathtt{p}_k}(\mathtt{x}) + time_{\mathtt{r}}(\mathtt{x}.f(\mathtt{x}))$$

A straightforward time analysis (summing exponentials) yields

$$time_{\mathtt{opt}}(\mathtt{x}) \leq c2^k(time_{\mathtt{q}}(\mathtt{x}) + time_{\mathtt{r}}(\mathtt{x}.f(\mathtt{x})))$$

as required, where c is not excessively large.

We now estimate the value of the constant factor. Imagine (naturally enough) that the programs $\mathtt{p}_i$ are enumerated in order of increasing length. Then program $\mathtt{q} = \mathtt{p}_k$ would occur at position

near[8] $k = 2^{O(|\mathtt{q}|)}$, where the $O()$ represents a small constant. Thus the constant factor can be bounded from above by $c2^k = c2^{2^{O(|\mathtt{q}|)}}$.

It must be admitted that this constant factor is enormous. The interesting thing is that it exists at all, i.e., that the construction gives, from an asymptotic viewpoint, the best possible result.

9 Overview of complexity theory

Computability theory concerns only *what* is computable, and pays no attention at all to how much time or space is required to carry out a computation. In the real computing world, however, computational resource usage is of primary importance, as it can determine whether or not a problem is solvable at all in practice.

The central computational concepts of complexity theory all involve resources, chiefly time, space, and nondeterminism. Complexity theory has evolved a substantial understanding of just what the *intrinsic complexity* is of many interesting general and practically motivated problems. This is reflected by a well-developed classification system for "how decidable" a problem is. Theorems 7.2 and 8.2 lie in the realm of complexity theory.

Computability theory has similar classification systems, for "how *undecidable*" a problem is, but the subject is beyond the scope of this work.

Complexity theory is mostly about the classification of decision problems (the question $x \in A$? for a set A) by the amount of time, space, etc., required to solve them. Ttime, space, etc., are usually measured as a function of the size $|x|$ of the question instance. Complexity of function computation is similar but a bit more involved (and not treated here).

As was the case for computability, we will not consider finite problems; instead, we study the *asymptotic complexity* of a program solving a program: a function $f : I\!N \to I\!N$ defining how rapidly its resource usage grows, as the size of its input data grows to infinity.

The remainder of this work develops a *hierarchy of robust subclasses* within the class of all decidable sets, and investigates its properties. The significance of the hierarchy is greater because of *representation invariants*: the fact that the placement of a problem within it is in general quite independent of the way the problem is described, for example whether graphs are represented by connection matrices or by adjacency lists.

In some cases we prove *proper containments* between adjacent problem classes in the hierarchy: that a sufficient resource increase will *properly increase* the classes of problems that can be solved. In other cases, questions concerning proper containments are still unsolved, and have been for many years.

In lieue of definitive answers, we will characterise certain problems as *complete* for the class of all problems solvable within given resource bounds. A complete problem is both solvable within the given bounds and, in a precise technical sense, "hardest" among all problems so solvable. Many familiar problems will be seen to be complete for various of these complexity classes.

9.1 Time- and space-bounded classes of decision problems

Definition 9.1 Suppose L is a programming language, and time and space measures $time^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d})$, $space^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d})$ have the property that for any l-program p and L-data d, $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}) = \perp$ iff $time^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d}) = \perp$, and $[\![\mathtt{p}]\!]^{\mathtt{L}}(\mathtt{d}) = \perp$ iff $space^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d}) = \perp$. Let $f : I\!N \to I\!N$ be a nonzero function. Then by definition

$$\text{TIME}^{L}(f) = \{A \subseteq I\!D \mid \exists\ \mathtt{L}\text{-program p (p decides } A\text{, and } time^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d}) \leq f(|\mathtt{d}|) \text{ for all L-data d})\}$$

$$\text{SPACE}^{L}(f) = \{A \subseteq I\!D \mid \exists\ \mathtt{L}\text{-program p (p decides } A\text{, and } space^{\mathtt{L}}_{\mathtt{p}}(\mathtt{d}) \leq f(|\mathtt{d}|) \text{ for all L-data d})\}$$

Further, define

[8] If binary strings are enumerated, then number 2^i has length $i+1$, and it is easy to see $I\!D$ has a similar property.

$$\text{PTIME}^L = \bigcup_{\pi \text{ a polynomial}} \text{TIME}(\pi) \qquad \text{PSPACE}^L = \bigcup_{\pi \text{ a polynomial}} \text{SPACE}(\pi)$$

$$\text{LOGSPACE}^L = \bigcup_{k \geq 0} \text{SPACE}(\lambda n.k \log n)$$

9.2 Invariance of resource-bounded computational power

Resource-bounded problem-solving power is equivalent over a wide spectrum of computational models. Time and space accounting of simulations as in proof of Theorem 3.10 establish that

1. Computability, *up to linear differences in running time*, is equivalent for `WHILE`, `GOTO`, `FUN1` and `FUNHO`.
2. Computability, *up to polynomial differences in running time*, is equivalent for all of: `WHILE`, `GOTO`, `FUN1`, `FUNHO` `SRAM`, and `TM`.
3. Computability, *up to polynomial differences in memory usage*, is equivalent for `CM`, `SRAM`, and `TM`.

This leads to the following "robustness" or invariance result[9], which justifies writing just PTIME:

Theorem 9.2 $\text{PTIME}^{\texttt{TM}} = \text{PTIME}^{\texttt{SRAM}} = \text{PTIME}^{\texttt{WHILE}} = \text{PTIME}^{\texttt{FUN1}} = \text{PTIME}^{\texttt{FUNHO}}$

The class PTIME plays a central role in complexity theory, as it is common to identify "tractable prolem" with "solvable in polynomial time." While this can be argued with it is very natural, and proof that a problem does *not* lie in PTIME is strong evidence that it is indeed intractable.

Similarly, the following result justifies writing just PSPACE:

Theorem 9.3 $\text{PSPACE}^{\texttt{TM}} = \text{PSPACE}^{\texttt{SRAM}} = \text{PSPACE}^{\texttt{CM}}$

Computation with logarithmic space

The class $\text{LOGSPACE}^{\texttt{TM}}$ is the class of problems solvable by offline Turing machines whose space consumption is at most $k \log n$ for inputs of length n. Since the bound makes little sense for other machine types, we henceforth write just LOGSPACE.

LOGSPACE also plays a central role in complexity theory, as it is the beginning of the much-studied hierarchy $\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} \subseteq \ldots$ Further, all the well-known *problem reductions* used to show familiar combinatorial problems complete for these classes can be carried out by logspace-bounded Turing machines.

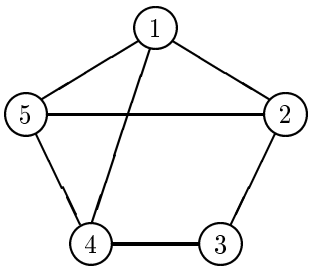
LOGSPACE is in a sense the smallest natural complexity class, because a machine needs storage of at least $O(\log n)$ bits[10] in order to "remember" a position in its input data, or to count a number of input symbols — for instance, to decide whether its input has the form $\texttt{0}^n\texttt{1}^n\texttt{0}^n$.

9.3 Invariance of complexity with respect to problem representations

An important question: can the way a problem is presented significantly affect the complexity of its solution? The *problem representation invariance property* says no:[11]

> All natural problem representations are inter-translatable by algorithms running in logarithmic space, or polynomials of low degree.

[9] Using the bijection to ignore the difference between $\{\texttt{0},\texttt{1}\}^*$ and $\mathbb{N}$.
[10] Here n = input size, i.e., string length or tree size.
[11] This is not a theorem, but an observation based on experience.



Incidence matrix:

	1	2	3	4	5
1	0	1	0	1	1
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	1	1	0	1	0

Figure 9.1: An undirected graph, and its incidence matrix.

For one example, a number n can be presented either in binary notation, or in the much longer unary notation, such as the list form $\texttt{nil}^n$. However unary notation for numbers seems unnatural given that we measure complexity as a function of input length, since unary notation is exponentially more space-consuming than binary notation.

Another example is that a directed or undirected graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$ can be presented in any of several forms:

1. An n by n incidence matrix M with $M_{i,j}$ equal to 1 if $(v_i, v_j) \in E$ and 0 otherwise. Figure 9.1 contains an example.

2. An adjacency list $(u, u', u'', \ldots)$ for each $v \in V$, containing all vertices u for which $(v, u) \in E$. An example is

   $$[1 \mapsto (2, 4, 5), 2 \mapsto (1, 3, 5), 3 \mapsto (2, 4), 4 \mapsto (1, 3, 5), 5 \mapsto (1, 2, 4)].$$

3. A list of all the vertices $v \in V$ and edges $(v_i, v_j) \in E$, in some order.

   Example: $[1, 2, 3, 4, 5]$, $[(1, 2), (2, 3), (3, 4), (4, 5), (5, 1), (1, 4), (2, 5)]$; or

4. In a compressed format in case the graph is known to be sparse, i.e., to have few edges between vertices.

There are also differences in the graph representations, though less dramatic. For example, the incidence matrix is guaranteed to use n^2 bits, but a sparse matrix could be stored in much less space; and even the apparently economical adjacency list form is less than optimal for dense graphs, as adjacency lists can take as many as $O(n^2 \log n)$ bits, assuming vertex indices to be given in binary.

Problem equivalence modulo representation. One example: a k-clique in undirected graph G is a set of k vertices such that G has an edge between every pair in the set. The CLIQUE decision problem is: given (G, k), to decide whether G has at least one k-clique.

It is not known whether the CLIQUE problem is in PTIME or not — and all known algorithms take exponential time in the worst case. However a little thought shows that the choice of representation will not affect its status, since one can convert back and forth among the various graph representations in polynomial time; so existence of a polynomial time CLIQUE algorithm for one representation would immediately imply the same for any of the other representations.

From this viewpoint the most "sensible" problem representations are all equivalent, at least up to polynomial-time computable changes in representation. There are a few exceptions to this rule involving degenerate problem instances, for example extremely sparse graphs, but such exceptions only seem to confirm that the rule holds in general.

9.4 Nondeterminism

Many practically interesting but apparently intractable problems lie in the class NPTIME, a superset of PTIME including, loosely speaking, programs that can "guess," formally called *nondeterministic*. Such programs can solve many challenging search or optimisation problems by a simple-minded and efficient technique of *guessing* a possible solution and then *verifying*, within polynomial time, whether or not the guessed solution is in fact a correct solution. The ability to guess is formally called "nondeterminism."

Time- and space resource-bounded classes have already been defined, so we describe their nondeterministic counterparts quite briefly. Syntactically, nondeterminism can be expressed by adding one new instruction type:

```
choose ℓ or ℓ'      (* Go to either instruction Iℓ or to Iℓ' *)
```

Semantically, the construct is given an *angelic* interpretation: a nondeterministic program correctly solves a set membership problem "$\mathtt{d} \in A$?" if

- $\mathtt{d} \in A$ implies that `p` *can possibly take* one or more guess sequences, leading to the answer `true`; and
- $\mathtt{d} \notin A$ implies that `p` cannot take any *guess sequence at all* that leads to the answer `true`.

Given time resource bound f, we say that `p` *runs in time* f if for all `d`, whenever `p` can yield `true`, then it can do so by a computation with length no longer than $f(|\mathtt{d}|)$. An analogous definition applies to space bounds.

Definition 9.4 Resource-bounded problem classes $\text{NTIME}^{\mathtt{L}}(f)$ and $\text{NSPACE}^{\mathtt{L}}(f)$ are exactly as in Definition 9.1, except that the programs `p` mentioned there are now allowed to be nondeterministic, using the definitions of acceptance and running time just given. Classes NPTIME, NPSPACE and NLOGSPACE are defined analogously.

For practical purposes it is not at all clear *how, or whether*, nondeterministic polynomial-time algorithms can be realised by deterministic polynomial-time computation. This intensely studied problem "PTIME = NPTIME?", often expressed as "P = NP?", has been open for many years. In practice, all solutions to such problems seem to take at least exponential time in worst-case situations. It is particularly frustrating that no one has been able to prove no subexponential worst-case solutions exist.

9.5 A backbone hierarchy of resource-bounded problem classes

The combinations of these resources lead to a widely encompassing "backbone" hierarchy:

Theorem 9.5 LOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ PTIME $\subseteq$ NPTIME $\subseteq$ PSPACE = NPSPACE $\subset$ REC $\subset$ RE

Here LOGSPACE, PTIME and PSPACE denote those problems solvable by deterministic algorithms in time and space, respectively, bounded by polynomial functions of the problem's input size. Classes NLOGSPACE, NPTIME, NPSPACE denote the problems decidable within the same bounds, but by nondeterministic algorithms that are allowed to or "guess"; and REC, RE are the *recursive* and *recursively enumerable* classes of decision problems already studied in Section 2.4.1.

The significance of the hierarchy is that a great number of practically interesting problems (e.g. maze searching, graph coloring, timetabling, regular expression manipulation, context-free grammar properties) can be precisely located at one or another stage in this progression.

Proof overview: To begin with, LOGSPACE $\subseteq$ NLOGSPACE, and PTIME $\subseteq$ NPTIME, and PSPACE $\subseteq$ NPSPACE. These are trivial by definition, since every ordinary deterministic program is also a

nondeterministic program. NPTIME $\subseteq$ NPSPACE is also immediate since a program's space usage cannot exceed its time usage. Further, PSPACE $\subseteq$ REC is immediate by definition, and we have already proven REC $\subset$ RE in Theorem 2.4.

Remaining are: NLOGSPACE $\subseteq$ PTIME and NPSPACE $\subseteq$ PSPACE. These are proven by considering, for a given input `d` and resource-limited program `p`, the graph $G_{\mathtt{p}}(\mathtt{d})$ whose vertices are *configurations* encountered in computations of `p` on `d`, and whose edges $C \to C'$ correspond to single computation steps.[12]

NLOGSPACE $\subseteq$ PTIME is shown by applying a fast algorithm for graph traversal to $G_{\mathtt{p}}(\mathtt{d})$; and NPSPACE $\subseteq$ PSPACE is shown by applying a memory-economical $O((\log n)^2)$ space algorithm for graph traversal to $G_{\mathtt{p}}(\mathtt{d})$. Details may be found in Chapter 23 of [15].

A collection of open problems

A long-standing open problem is whether these "backbone" inclusions are proper. Many researchers think that every inclusion is proper, but proofs have remained elusive. All that is known for sure (by a theorem analogous to 7.2) is that NLOGSPACE $\subset$ PSPACE, a very weak statement. (However, it implies that at least *one* inclusion among NLOGSPACE $\subseteq$ PTIME $\subseteq$ NPTIME $\subseteq$ PSPACE is proper!)

10 Expressive power of some programming languages

The *expressivity* of a programming language `L` can be characterised "extensionally" as the class of all problems that can be solved by `L`-programs. By this measure, the languages `WHILE`, `GOTO`, `FUN1`, `FUNHO`, `TM`, `CM` and `SRAM` are all equally expressive: they can accept all the recursively enumerable sets, and can decide all the recursive sets.

In order to obtain nontrivial results on expressiveness we will look at languages `L` that are less than Turing-complete, and relate the class of `L`-solvable problems to the complexity classes of Section 9. In this section we overview results linking complexity classes to some sublanguages of the programming languages seen earlier.

Following are some of the more striking linkages, where a *read-only* (or *cons-free*) program is one with no constructor operator "`cons`". Full proofs are not given but some key ideas will be sketched.

Theorem 10.1 Set $A \subseteq \{\mathtt{0},\mathtt{1}\}^*$ is in PTIME if and only if it is decidable by a read-only `FUN1` program.

Theorem 10.2 Set $A \subseteq \{\mathtt{0},\mathtt{1}\}^*$ is in LOGSPACE if and only if it is decidable by a tail-recursive read-only `FUN1` program.

Theorem 10.3 Set $A \subseteq \{\mathtt{0},\mathtt{1}\}^*$ is in RE if and only if it is the domain of a read-only `FUNHO` program (untyped).

Applying the construction of Theorem 3.4 to Theorem 10.2, we obtain:

Corollary 10.4 Set $A \subseteq \{\mathtt{0},\mathtt{1}\}^*$ is in LOGSPACE if and only if it is decidable by a read-only `WHILE` program.

Remarks on the LOGSPACE characterisation. Despite the centrality of the class LOGSPACE, its Turing machine definition seems unnatural due to hardware restrictions on the number of tapes, their usage, and the external restriction on the run-time length of the work tape. There is, however, a "folklore theorem" that logspace Turing machines have the same decision power as

[12] Remark: for a deterministic program `p`, graph $G_{\mathtt{p}}(\mathtt{d})$ has maximum node out-degree 1. A nondeterministic program `p` may have many computations on the same input, so nodes in $G_{\mathtt{p}}(\mathtt{d})$ may have greater out-degrees.

certain *read-only* machines: two-way multihead finite automata. These machines can "see but not touch" their input. The read-only `FUN1` and read-only `WHILE` languages naturally formulate just this idea: Theorems 10.2 and 10.4 are a re-expression of the folk theorem.

Remarks on the PTIME characterisation. Theorem 10.1 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 $2^{O(n)}$ 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. It has two parts:

- *Construction 1* 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 read-only program.
- *Construction 2* 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.

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

Overview of proofs. Each of the theorems is shown by a pair of simulations. Proof for the "only if" part begins with a Turing machine that decides/accepts set A. From this, we have to construct a functional program that faithfully simulates the Turing machine.

Proof for the "if" part begins with a functional program that decides/accepts set A. From this, we have to construct a Turing machine that faithfully simulates the functional program, and runs within the specified time or space bounds.

10.1 Turing machine simulation by functional programs

As an intermediate step we show how to simulate a Turing machine by a functional program *equipped with natural numbers*. These numbers will be seen to be have size bound that are functions of tme or space limits on the Turing machine. Finally, it will be shown that the numbers and operations on them can be eliminated, simulating them by read-only functional programs.

Consider a one-tape Turing machine program $\mathtt{tm} = \mathtt{1{:}I_1\ 2{:}I_2\ \ldots m{:}I_m}$. A "configuration" or state of `tm` has form $s = (\ell, \sigma)$ with control point $\ell \in \{1, \ldots, m, m+1\}$, and store of form $\sigma = \ldots \mathtt{BL\underline{S}RB} \ldots$ where $\mathtt{S} \in \{\mathtt{0}, \mathtt{1}, \mathtt{B}\}$ is the scanned symbol, and $\mathtt{L}, \mathtt{R} \in \{\mathtt{0}, \mathtt{1}, \mathtt{B}\}^*$ are the tape portions lying to its left and right

10.1.1 Backward Turing machine simulation

Lemma 10.5 Let $\mathtt{tm} = \mathtt{1{:}I_1\ \ 2{:}I_2\ \ \ldots m{:}I_m}$ be a one-tape TM-program with input $\mathtt{as} = \mathtt{a}_1 \ldots \mathtt{a}_n \in \{0,1\}^*$. Let $(\ell_1, \sigma_1) \to \ldots \to (\ell_t, \sigma_t) \to \ldots$ be the (finite or infinite) computation of tm on as, where $\ell_1 = 1$ and the initial tape is $\sigma_1 = \underline{\mathtt{B}}\,\mathtt{a}_1 \ldots \mathtt{a}_n$. Define functions

$$\mathtt{Tape} : \mathbb{Z} \times \mathbb{N} \to \{0, 1, \mathtt{B}\}, \mathtt{Label} : \mathbb{N} \to \{1, \ldots, m, m+1\}$$

as follows for any time and tape positions $t \geq 0, i \in \mathbb{Z}$:

$$\begin{array}{lcll} \mathtt{Tape}(i,t) & = & \mathtt{b}_i & \text{if } s_t = (\ell_t, \sigma_t) \text{ and } \sigma_t = \ldots \mathtt{b}_{\perp 2} \mathtt{b}_{\perp 1}\, \underline{\mathtt{b}_0}\, \mathtt{b}_1\, \mathtt{b}_2\, \mathtt{b}_3 \ldots \\ \mathtt{Label}(t) & = & \ell_t & \text{if } s_t = (\ell_t, \sigma_t) \end{array}$$

Then the equations in Figure 10.1 hold for all $t, i \in \mathbb{Z}$ such that $0 \leq t < time_{\mathtt{tm}}(\mathtt{as})$.

At time $t = 0$:

$$\mathtt{Label}(0) = 1 \quad \text{At start: about to execute instruction 1}$$

$$\mathtt{Tape}(i,0) = \begin{cases} \mathtt{a}_i & \text{if } 1 \leq i \leq n \\ \mathtt{B} & \text{if } i \leq 0 \text{ or } i > n \end{cases}$$

At time $t > 0$:

$$\mathtt{Label}(t) = \begin{cases} \ell' & \text{if } \mathtt{I}_{\mathtt{Label}(t \perp 1)} = \mathtt{if\ S\ goto}\ \ell'\ \mathtt{else}\ \ell'' \\ & \quad \text{and } \mathtt{Tape}(0, t \perp 1) = \mathtt{S} \\ \ell'' & \text{if } \mathtt{I}_{\mathtt{Label}(t \perp 1)} = \mathtt{if\ S\ goto}\ \ell'\ \mathtt{else}\ \ell'' \\ & \quad \text{and } \mathtt{Tape}(0, t \perp 1) \neq \mathtt{S} \\ \mathtt{Label}(t \perp 1) + 1 & \text{otherwise} \end{cases}$$

$$\mathtt{Tape}(i,t) = \begin{cases} \mathtt{Tape}(i+1, t \perp 1) & \text{if } \mathtt{I}_{\mathtt{Label}(t \perp 1)} = \mathtt{right} \\ \mathtt{Tape}(i \perp 1, t \perp 1) & \text{if } \mathtt{I}_{\mathtt{Label}(t \perp 1)} = \mathtt{left} \\ \mathtt{S} & \text{if } \mathtt{I}_{\mathtt{Label}(t \perp 1)} = \mathtt{write\ S} \text{ and } i = 0 \\ \mathtt{Tape}(i, t \perp 1) & \text{otherwise} \end{cases}$$

Figure 10.1: Relations between the values of Tape *and* Label.

Backward simulation by a functional program with numbers

The equations of Figure Figure 10.1 can be used to construct a functional program simulating Turing program tm. We use a Haskell-like syntax.

```
Run(d) = if Tape(0, Runtime(0)) = 1 then True else False   where

   Runtime(t) =  if Label(t) = m+1 then t else Runtime(t+1)

   Label(t)   =  if t=0 then 1 else case Label(t-1) of ...

   Tape(i,t)  =  if t=0 then ... else ...
```

The missing parts ... are code for the relations seen in Figure 10.1. The calls to Label and Tape terminate since t decreases until it reaches 0. If tm terminates on input as, the functional program will too, since instruction m+1 will eventually be reached, so the Runtime also terminates.

Lemma 10.6 A Turing machine program `tm` can be simulated by a read-only number program. Further, if `tm` runs in time $T(n)$ on inputs of length n, then the natural numbers used by $\texttt{Tr}_{\texttt{tm}}$ are at largest $T(n)$.

10.1.2 Forward Turing machine simulation

A tape containing $\ldots \texttt{b}_{\perp 2}\texttt{b}_{\perp 1}\underline{\texttt{b}_0}\,\texttt{b}_1\,\texttt{b}_2\,\texttt{b}_3\ \ldots$ can be represented by a pair of numbers l, r, where

- $l = \sum_i^\infty 2^i \beta_{\perp i}$ is the value of string $\ldots \texttt{b}_{\perp 2}\texttt{b}_{\perp 1}\texttt{b}_0$ *encoded as a base 3 number*, so $\beta = 0, 1, 2$ for symbols `b` =`B`, `0` and `1`, respectively).
- $r = \sum_i^\infty 2^i \beta_i$ is the value of string $\texttt{b}_0\texttt{b}_1\,\texttt{b}_2\,\texttt{b}_3\ \ldots$, also as a base 3 number. (Note the order reversal, and the fact that blanks on left or right end of the tape do not contribute to l, r.)

The simulation is carried out by the program $\texttt{Tr}_{\texttt{tm}}$ of Figure 10.2, 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.

```
runTR(as)                = aux(0,1,as)

aux (s,val,[])           = execute_1(s, val)
aux (s,val,(False:as))   = aux (s+1*val,3*val,as)
aux (s,val,(True:as))    = aux (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 $\ell = 1, 2, \ldots, m$, $\texttt{SIMULATE}_\ell$ is defined as follows, where $\overline{\texttt{S}} = 0, 1, 2$, resp., if `S` = `B`, `0` or `1`:

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

Figure 10.2: Program $\texttt{Tr}_{\texttt{tm}}$*: tail recursive Turing machine simulation*

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

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

Lemma 10.7 A Turing machine program `tm` can be simulated by a read-only tail-recursive 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 $\texttt{Tr}_{\texttt{tm}}$ are at largest $3^{S(n)}$.

An extension of this construction to two-tape offline Turing machines yields the following result. The polynomial comes from the fact that $3^{k \log n} = n^{k \log 3}$.

Lemma 10.8 Suppose Turing machine program `tm` runs in space $k \log n$ on inputs of length n. Then `tm` can be simulated by a read-only tail-recursive number program $\mathtt{Tr}_{\mathtt{tm}}$. Further, the natural numbers used by $\mathtt{Tr}_{\mathtt{tm}}$ are bounded by a polynomial in n.

10.1.3 Getting rid of the numbers

Recall that the n appearing above is the length of the input, which is a list $\mathtt{as} = (\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n)$.

Lemma 10.9 Suppose read-only number program `p` has numbers bounded by a polynomial in n on inputs of legth n. Then there exists a read-only program `q` without numbers such that $[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$. Further, if `q` is tail-recursive if `p` is.

Proof. The idea is to represent a number $i \leq n$ by a suffix of the input `as`. Claim: the following read-only functional program can (in effect) increment and decrement a counter of size up to n, and test for zero.

```
min(as)      =  nil
max(as)      =  as
decr(i,as)   =  tl i
zero?(i,as)  =  (i=nil)
incr(i,as)   =  aux1(i,as,as)
aux1(i,j,as) =  if i=nil then aux2(tl j,as,as)
                         else aux1(tl i, tl j, as)
aux2(i,j,as) =  if i=nil then j
                         else aux2(tl i, tl j, as)
```

Idea: represent number $i \leq n$ by a length-i suffix `i` of $\mathtt{as} = (\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n)$. Then `tl i` represents $\max(i \perp 1, 0)$, and the zero test is then simply a test on whether `i` is the empty list. The code for `incr` is trickier but exploits the fact that $i \leq n$. In effect it calls $\mathtt{aux1}(0, n, n)$, which counts its i, j arguments down until $i = 0, j = n \perp i$, and then calls $\mathtt{aux2}(n \perp i \perp 1, n, n)$. Then `aux2` counts its first and second arguments down together until the first reaches 0, at which time the second is $n \perp (n \perp i \perp 1) = i + 1$.

This "library" allows simulation of any read-only number program `p` whose numbers are bounded by n. If a program's numbers are bounded by, say, n^2, this can be simulated as a pair of digits, each ranging from 0 to n. This generalizes to any polynomial.

Remark: these functions are all tail-recursive.

This plus Lemma 10.6 complete the "only if" part of Theorem 10.1. This plus Lemma 10.8 complete the "only if" part of Theorem 10.2.

10.2 Simulating read-only programs by Turing machines

The "if" parts of Theorems 10.1 and 10.2 require simulating read-only programs by Turing machines.

10.2.1 Relating GOTO and Turing machine states.

Given a `GOTO` program, a Turing machine simulating it will store on its work tape the pointer values of all the `GOTO` variables. Conversely, given a Turing program, a `GOTO` program simulating it will encode, by means of pointer values, the current contents of the Turing machine's work tape. We first analyse their numbers of states.

Off-line Turing machine: Let tm have m instructions, work tape storage bound $k \log n$, and input $\mathtt{a}_1\mathtt{a}_1 \ldots \mathtt{a}_n \in \{0,1\}^*$. Then tm can enter at most

$$m(n+2)3^{k \log n} = m(n+2)n^{k \log 3}$$

different total states. For fixed tm this number is $O(n^{1+k \log 3})$.

Corollary 10.10 LOGSPACE $\subseteq$ PTIME.

Read-only GOTO program: Let q have m' instructions, k' variables, and input list $(\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n) \in \mathbb{D}_{01}$. During the computation every q-variable X value must be a pointer to one of:

1. The root of a suffix $(\mathtt{a}_i\mathtt{a}_{i+1} \ldots \mathtt{a}_n)$ at a position i along the "spine" of the input list; or
2. The root of (nil.nil), the coding $c(1)$ of some $a_i = 1$; or
3. The atom nil.[13]

Thus each variable can take on at most $n+2$ values, so the number of program total states is bounded by: $m'(n+2)^{k'}$. This is also a polynomial in n and thus a polynomial in $|\ (\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n)\ |$, giving hope for Theorem 10.2 since the programs of these two types have comparable numbers of states for a given input.

If part of Corollary 10.4. Let q = 1:I1 2:I2... m:Im be a cons-free GOTO program. Its instructions are of the form X := nil, X := Y, X := hd Y, X := tl Y, or if X goto ℓ else ℓ'.

The input to q is a list $(\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n) \in \mathbb{D}_{01}$ corresponding to string $\mathtt{a}_1\mathtt{a}_2 \ldots \mathtt{a}_n$ in $\{0,1\}^*$. Possible variable values in any q state have been analysed in three cases above. Construct an off-line Turing machine p to simulate q, as follows.[14] The idea is to represent each variable X of q by its *position* and its *tag:* numbers (p_X, t_X) with values $(i,0)$ in case 1, and $(0,1)$ in case 2, and $(0,0)$ in case 3. Turing machine program p stores each pair (p_X, t_X) on its work tape in binary, thus using at most $1 + \lceil \log n \rceil$ bits for each of q's variables. GOTO command I is simulated by Turing commands achieving the following effects:

GOTO command I	Effect of corresponding Turing code
X := nil	$(p_X, t_X) := (0,0)$
X := Y	$(p_X, t_X) := (p_Y, t_Y)$
X := hd Y	(p_X, t_X):= if $p_Y > 0 \wedge \mathtt{a}_{p_Y} = 1$ then$(0,1)$else$(0,0)$
X := tl Y	(p_X, t_X):= if $p_Y > 0$ then $(p_Y \perp 1, t_Y)$ else $(0,0)$
if X goto ℓ else ℓ'	if $t_X = 1 \vee (p_X > 0 \wedge \mathtt{a}_{p_X} = 1)$ then goto ℓ else ℓ'

All is straightforward Turing programming; the test "$\mathtt{a}_{p_Y} = 1$" is done by scanning the Turing input tape $\mathtt{a}_1\mathtt{a}_1 \ldots \mathtt{a}_n$ left to right, until p_Y symbols have been seen. It is easy to see that this is a faithful simulation that preserves the representation of the variables in GOTO program q.

If part of Theorem 10.1. Suppose A is decided by a read-only FUN1 program p, and that it is given input as $= \mathtt{a}_1 \ldots \mathtt{a}_n \in \{0,1\}^*$, fixed in the rest of this proof. Consider a definition

```
f(x1,...,xk) = exp
```

[13] A value of nil can arise 3 ways, but the effects while executing p are the same: either it is the coding of some $a_i = 0$; or the nil at the end of the input list; or it is the head or tail of $c(1) =$ (nil.nil).

[14] We assume $n > 0$; special case code gives the correct answer for $n = 0$.

Analogous to the preceding proof. Let V denote the set of possible values of any parameter of `f`. A value can only be a pointer to a suffix of `as`, or to `(nil.nil)` or `nil`, so V has $n+2$ elementst. Thus there exist at most $(n+2)^k$ possible argument tuples for `f`.

We sketch a "memoising" implementation: for each such function `f`, the simulator maintains a table $mfg(\texttt{f}) \subseteq V^k \times (V \cup \{\bullet\})$. The program is executed from the start, but the tables $mfg(\texttt{f})$ for every `f` are maintained (and exploited). Initially, all are empty except for the initial function, containing $mfg(\texttt{f}_1) = ((\texttt{as}), \bullet)$.

Each time a function call $\texttt{f(e}_1 \ldots \texttt{e}_k\texttt{)}$ occurs, the parameter values $(v_1, \ldots, v_k)$ are obtained. If a pair $((v_1, \ldots, v_k), v) \in mfg(\texttt{f})$ with $v \neq \bullet$, then `f` has already been applied to this argument tuple, and simulation continues with value v. If not, then `f` is executed in the normal way, terminating with a value v. Before continuing, tuple $((v_1, \ldots, v_k), v)$ is added to $mfg(\texttt{f})$.

It is easy to see that no function call is simulated more than once. A straightforward time analysis reveals that the tables contain at most polynomially many elements; and that the total simulation time is also polynomially bounded.

10.3 Higher-order functions

First, Theorem 10.3 is just a restatement of the familiar result that expressions in the untyped lambda-calculus can compute arbitrary recursive functions. The situation is quite different, however, when *typed* functional programs are considered. Before stating the results (from the paper [12]) we define a simple monomorphic type system for `FUNHO`.

10.3.1 Types

Each parameter and function in a program is assigned a type (nonpolymorphic). Each type τ denotes a set of values $[\![\tau]\!]$. Type $\bullet$ denotes $[\![\bullet]\!] = \mathbb{D}$, and type $\tau \to \tau'$ denotes $\{f : [\![\tau]\!] \to [\![\tau]\!]'\}$, all functions with one argument of type τ and one result of type τ'. Multiple-argument functions are handled by "currying," for example $f : \tau_1 \to (\tau_2 \to \tau_3)$ is regarded as a function of two arguments, of types τ_1 and τ_2, and result type τ_3.

Judgement $\texttt{e}::\tau$, signifying that *expression* `e` *has type* τ is defined in Figure 10.3. A fully type-annotated function definition has form

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

A program must be well-typed to be syntactically legal. Henceforth, all programs are assumed to be fully annotated. For readability type superscripts are omitted when clear from context.

$$\frac{}{\texttt{d} :: \bullet} \qquad \frac{}{\texttt{x}^{\tau} :: \tau} \qquad \frac{}{\texttt{f}^{\tau} :: \tau} \qquad \frac{\texttt{e}_1 :: \bullet \quad \texttt{e}_2 :: \tau \quad \texttt{e}_3 :: \tau}{\texttt{if e}_1\texttt{e}_2\texttt{e}_3 :: \tau}$$

$$\frac{\texttt{e}_1 :: \bullet \quad \texttt{e}_2 :: \bullet}{\texttt{cons e}_1\ \texttt{e}_2 :: \bullet} \qquad \frac{\texttt{e} :: \bullet}{\texttt{hd e} :: \bullet} \qquad \frac{\texttt{e} :: \bullet}{\texttt{tl e} :: \bullet} \qquad \frac{\texttt{e}_1 :: \tau \to \tau' \quad \texttt{e}_2 :: \tau}{\texttt{e}_1\texttt{e}_2 :: \tau'}$$

Figure 10.3: Expression types

Definition 10.11 The *order* of a type is defined by $\text{order}(\bullet) = = 0$; and $\text{order}(\tau \to \tau') = \max(1 + \text{order}(\tau), \text{order}(\tau'))$.

Definition 10.12 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.

Remark: A "first-order program" is one that has data order 0 (it is the functions defined in the program that have order 1).

10.3.2 The expressive power of higher-order types

We precisely characterize, in terms of complexity classes, the effects on expressive power of various combinations of three possible limits on programs' *operations on data:* 1) constructors and destructors; 2) the order of their *data values:* 0, 1, or higher; and 3) their *control structures*: general recursion, tail recursion, primitive recursion. The links are summed up in the table of Figure 10.4, and confirm programmers' intuitions that higher-order types indeed give a greater problem-solving ability.

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 10.4 shows the effect of higher-order types on the computing power of programs of type $\bullet \to \bullet$. 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**
RO, untyped	–	–	–	–	...	REC.ENUM
RW, typed	REC.ENUM	REC.ENUM	REC.ENUM	REC.ENUM	...	REC.ENUM
RW, fold only	PRIM.REC	PRIM^1REC	PRIM^2REC	PRIM^3REC	...	(System T)
RO, typed	PTIME	EXPTIME	EXP^2TIME	EXP^3TIME	...	ELEMENTARY
RO tail recursive	LOGSPACE	PSPACE	EXPSPACE	EXP^2SPACE	...	ELEMENTARY
RO, fold only	LOGSPACE	PTIME	PSPACE	EXPTIME	...	ELEMENTARY

Figure 10.4: Expressivity of combinations of data orders and control. RO *= read-only = cons-free.*

Interpretation of the results. Figure 10.4 shows the effect of higher-order types on the computing power of various restricted program classes. Some of the table entries have analogs, more or less close, in the existing literature (see below). The formulations, definitions and contructions using functional programming are our own, and row 5, on higher-order tail recursive programs, seems to be new.

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.

Rows 1, 2: These program classes are all Turing complete. Consequently they can accept exactly the recursively enumerable subsets of $\{0,1\}^*$.

Row 3: These programs have unlimited data operations and types, but control is limited to primitive recursion, familiar to functional programmers under the name "fold right".[15] 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 [6], [22]. 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 4: 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 5 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.)

Row 6 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 4 equals the union for row 5 and for row 6. 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 these results. The results in Rows 1 and 2 are classical, and not repeated here. We prove the results in rows 4 and 5 of Figure 10.4. The results in row 5 appear to be new; and the results in row 4, while in a sense anticipated by [7], are here proven for the first time in a programming language context. The results in Row 6 are obtained from [8] and [9] by re-interpreting results from finite model theory.

On expressivity. 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 10.4 the answer is "no" for unrestricted programs, and "yes" for all the restricted languages we consider.

[15] Kleene's definition of primitive recursion is a bit more general than "fold right," but is easily programmed using fold right and composition. See [10] for details.

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 4, 5) 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 4, 6): *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.

11 Complete problems

An old slogan: "If you can't solve problems, then at least you can classify them."

Complete problems for the problem classes in the hierarchy. In spite of the many unresolved questions concerning proper containments in the "backbone," a great many problems have been proven to be *complete* for the various classes. If such a problem X is complete for class $\mathcal{C}$, then X is "hardest" for that class in the sense that if it lay within the next smaller class (call it $\mathcal{B}$ with $\mathcal{B} \subseteq \mathcal{C}$), then *every* problem in class $\mathcal{C}$ would also be in class $\mathcal{B}$, i.e. the hierarchy would "collapse" there, giving $\mathcal{B} = \mathcal{C}$. Complete problems are known to exist and will be constructed for every class in the "backbone" except for LOGSPACE (since no smaller class is present) and REC (for more subtle reasons.)

11.1 Reduction of one problem to another

The questions concerning concerning proper containments within

$$\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} = \text{NPSPACE}$$

are apparently quite difficult, since they have remained open since the 1970s in spite of many reseachers' best efforts to solve them. This has led to an alternative approach: to define *complexity comparison* relations $\leq$, also called *reductions*, between decision problems. Different relations $\leq$ will be appropriate for different complexity classes.

The statement $A \leq B$ can be interpreted as "problem A is no more difficult to solve than problem B," or even better: "given a way to solve B, a way to solve A can be found." One application is to prove a problem undecidable by reducing the halting problem to it. Intuitively: if $HALT$ is thought of as having infinite complexity, proving $HALT \leq B$ shows that B also has infinite complexity.

Further, we can use this idea to break a problem class such as RE or NPTIME into *equivalence subclasses* by defining A and B to be of equivalent complexity if $A \leq B$ and $B \leq A$.

The idea of reduction of one problem to another has been studied for many years, for example quite early in Mathematical Logic as a tool for comparing the complexity of two different unsolvable problems or undecidable sets. Many ways have been devised to reduce one problem to another since Emil Post's pathbreaking work in 1944.

11.2 Many-one reductions

Reduction $A \leq B$ where (say) $A, B \subseteq \mathbb{ID}$ can be defined in several ways.

Definition 11.1 Let $A, B \subseteq \mathbb{ID}$. Then A is *many-one* reducible to B by total function $f : \mathbb{ID} \to \mathbb{ID}$ if for any $\mathtt{d} \in \mathbb{ID}$ we have

$$\mathtt{d} \in A \text{ if and only if } f(\mathtt{d}) \in B$$

Clearly, if f is *computable* then an algorithm to decide membership in B can be used to decide membership in A. Further, if f is *efficiently* computable, then existence of an efficient B algorithm implies existence of an efficient A algorithm.

For proofs of undecidability, the only essential requirement is that the reduction be computable. Complexity classifications naturally involve bounds on the complexity of the questions that can be asked, for example of the function f used for many-one reducibility. In order to study, say, the class NPTIME, it is natural to limit one's self to reductions that can be computed by deterministic algorithms in polynomial time. Complexity comparison is thus almost always via *resource-bounded reduction* of one problem to another: $A \leq B$ means that one can efficiently transform an algorithm that solves B within given resource bounds into an algorithm that solves A within similar resource bounds. Some instances:

$\leq_{recursive}$	f must be a total computable function
$\leq_{ptime}$	f must be computable in polynomial time
$\leq_{logspace}$	f must be computable in logarithmic space

Some interesting facts, proven in the book, lie at the core of complexity theory:

1. Each of the several complexity classes $\mathcal{C}$ in the backbone hierarchy above LOGSPACE possesses *complete problems*. Such a problem (call it H) lies in class $\mathcal{C}$, and is "hardest" for it in the sense that $A \leq H$ for each problem A in $\mathcal{C}$. Class $\mathcal{C}$ may have many hard problems.

2. A complete problem H for class $\mathcal{D}$ has the property that if $H \in \mathcal{C}$ for a lower class $\mathcal{C}$ in the hierarchy, then $\mathcal{C} = \mathcal{D}$: the two classes are identical. Informally said, the hierarchy *collapses* at that point.

3. Even more interesting: Many *natural and practically motivated* problems have been proven to be complete for one or another complexity class $\mathcal{C}$.

11.3 Three example problems

Definition 11.2

1. A k-clique in undirected graph G is a set of k vertices such that G has an edge between every pair in the set. Figure 9.1 shows a graph G containing two 3-cliques: one with vertices 1, 2, 5 and another with vertices 1, 4, 5.

2. A boolean expression $\mathcal{F}$ is *closed* if it has no variables. If closed, $\mathcal{F}$ can be *evaluated* by the familiar rules such as $true \wedge false = false$.

3. A *truth assignment* for $\mathcal{F}$ is a function θ mapping variables to truth values such that $\theta(\mathcal{F})$ is a closed boolean expression. $\mathcal{F}$ is *satisfiable* if it evaluates to *true* for some truth assignment θ. $\mathcal{F}$ is in *conjunctive normal form*, abbreviated CNF, if it is a conjunction of disjunctions of variables or their negations.

4. By definition SAT $= \{\mathcal{F} \mid \mathcal{F}$ is a satisfiable boolean CNF expression.$\}$

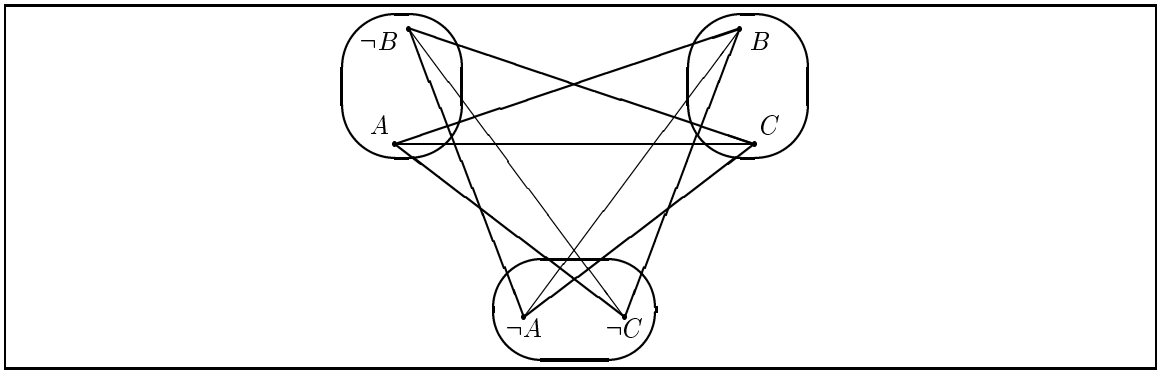


Figure 11.1: The graph $f((A \vee \neg B) \wedge (B \vee C) \wedge (\neg A \vee \neg C))$.

For an example of a satisfiable formula, the CNF expression

$$(A \vee \neg B) \wedge (B \vee C) \wedge (\neg A \vee \neg C)$$

is satisfied by truth assignment $\theta = [A \mapsto false, B \mapsto false, C \mapsto true]$.

Three combinatorial decision problems. Following are three typical and interesting problems which will serve to illustrate several points. In particular, each will be seen to be complete, i.e. hardest, problems among all those solvable in a nondeterministic time or space class. The problems:

GAP	=	{ G	|	directed graph $G = (V, E, v_0, v_{end})$ has a path from vertex v_0 to v_{end} }
CLIQUE	=	{ (G, k)	|	undirected graph G has a k-clique }
SAT	=	{ $\mathcal{F}$	|	$\mathcal{F}$ is a satisfiable boolean CNF expression }

Theorem 11.3 GAP is complete for the class NLOGSPACE with respect to $\leq_{logspace}$ reductions, where:

$$\text{GAP} = \{\ G = (V, E, v_0, v_{end}) \mid \text{graph } G \text{ has a path from vertex } v_0 \text{ to } v_{end}\ \}$$

Theorem 11.4 *HALT* is complete for the class RE with respect to $\leq_{recursive}$ reductions, where:

$$\mathit{HALT} = \{(\mathtt{p.d}) \mid \mathtt{p} \text{ is a GOTO-program and } [\![\mathtt{p}]\!](\mathtt{d}) \neq \bot\}$$

11.4 Reducing SAT to CLIQUE in polynomial time

Many superficially quite different problems turn out to be "sisters under the skin," in the sense that each can be efficiently reduced to the other. We show by informal example that SAT $\leq_{ptime}$ CLIQUE. This means that there is a polynomial time computable function f which, when given any CNF boolean expression $\mathcal{F}$, will yield a pair $f(\mathcal{F}) = (G, k)$ such that graph G has a k-clique if and only if $\mathcal{F}$ is a satisfiable expression.

This implies that CLIQUE is at least as hard to solve as SAT in polynomial time: given a polynomial time algorithm p to solve CLIQUE, one could answer the question "is $\mathcal{F}$ satisfiable?" by first computing $f(\mathcal{F})$ and then running p on the result.

Construction. Given a conjunctive normal form boolean expression $\mathcal{F} = C_1 \wedge \ldots \wedge C_k$, construct a graph $f(\mathcal{F}) = (G, k)$ where graph $G = (V, E)$ and

1. V = the set of occurrences of literals in $\mathcal{F}$
2. $E = \{(a, b) \mid a$ and b are not in the same conjunct of $\mathcal{F}$, and neither is the negation of the other$\}$

For an instance, the expression

$$(A \vee \neg B) \wedge (B \vee C) \wedge (\neg A \vee \neg C)$$

would give graph $f(\mathcal{F})$ as in Figure 11.1. The expression $\mathcal{F}$ is satisfied by truth assignment $[A \mapsto false, B \mapsto false, C \mapsto true]$, which corresponds to the 3-clique $\{\neg A, \neg B, C\}$. More generally, if $\mathcal{F}$ has n conjuncts, there will be one n-clique in $f(\mathcal{F})$ for every truth assignment that satisfies $\mathcal{F}$, and these will be the only n-cliques in $f(\mathcal{F})$.

It is also possible to show that CLIQUE $\underset{ptime}{\leq}$ SAT, but by a less straightforward construction. Consequence: SAT $\in$ PTIME if and only if CLIQUE $\in$ PTIME.

11.5 Complexity of problems about Boolean programs

Theorem 11.5 SAT is complete for the class NPTIME with respect to $\leq_{ptime}$ reductions.

Proof is by reduction. The key idea (due to Cook [4]) is to construct, from a nondeterministic polynomial-time-bounded Turing program `p` and input `d`, a CNF formula $\mathcal{F}$ that is satisfiable if and only if `p` has an accepting computation on input `d`.

The breakthrough in Cook's proof was to see how to relate Turing machine computations with boolean expressions, so the existence of a choice sequence leading the Turing machine to accept its input corresponds to the existence of variables making the constructed boolean expression true. As a stepping-stone, we show how to reduce Turing machine computations to computations by *boolean programs*.

Program analysis is in general undecidable due the halting problem's recursive unsolvability, and Rice's general result that all nontrivial extensional program properties are undecidable. On the other hand, *finite-memory* programs [18, 19] do have decidable properties since their entire state spaces can be computed. A series of theorems relating well-known complexity classes to finite program analysis appears in [15].

Definition 11.6 (The language `BOOLE`) A *boolean program* is an input-free program $\texttt{p} = \texttt{1:I}_1 \ldots \texttt{m:I}_m$ where each instruction `I` and expression `E` is of form given by:

`I` ::= `X := E` | $\texttt{I}_1$`; `$\texttt{I}_2$ | `goto` ℓ | `if E then` $\texttt{I}_1$ `else` $\texttt{I}_2$

`E` ::= `X` | `true` | `false` | $\texttt{E}_1 \vee \texttt{E}_2$ | $\texttt{E}_1 \wedge \texttt{E}_2$ | $\neg$ `E` | $\texttt{E}_1 \Rightarrow \texttt{E}_2$ | $\texttt{E}_1 \Leftrightarrow \texttt{E}_2$

`X` ::= `X0` | `X1` | ...

Semantics is what one expects, not detailed here.

Theorem 11.7 The problem of deciding membership in the following subsets of `BOOLE`-*programs* is characterised as follows:

1. { `p` $\in$ `BOOLE`-*programs* | $[\![\texttt{p}]\!] =$ `true` }: complete for PSPACE (deterministic or nondeterministic polynomial space)

2. {goto-free $\mathtt{p} \in$ BOOLE-*programs* | $[\![\mathtt{p}]\!] = \mathtt{true}$ }: complete for PTIME (deterministic polynomial time)

3. {goto-free $\mathtt{p} \in$ BOOLE-*programs* | $[\![\mathtt{q;p}]\!] = \mathtt{true}$ for some $\mathtt{q}$ }: complete for NPTIME (nondeterministic polynomial time)

11.6 Reducing a PSPACE problem to acceptance by a boolean program

We now show Part 1 of Theorem 11.7. Minor variations on the construction suffice for Parts 2 and 3, as well.

Lemma 11.8 Let Turing machine program $\mathtt{p}$ run in polynomial space on any input of length n. Then there is a function

$$f : \{0,1\}^* \to \{\text{BOOLE programs}\}$$

such that for any input $\mathtt{d} = \mathtt{a}_1 \ldots \mathtt{a}_n$ (each $\mathtt{a}_i \in \{0,1\}$) we have

$$\mathtt{p} \text{ accepts } \mathtt{d} \text{ if and only if } [\![f(\mathtt{d})]\!]^{\mathtt{BOOLE}} = \mathtt{true}$$

Further, $f(\mathtt{d})$ is computable in space $O(\log |\mathtt{d}|)$.

Proof. Let $\mathtt{p} = \mathtt{1{:}I}_1 \ \ldots \ \ \mathtt{m{:}I}_m$. Without loss of generality assume that $\mathtt{d} \in A$ if and only if $\mathtt{p}$ has a computation that terminates at program control point $\mathtt{m{+}1}$. By assumption there is a polynomial $\pi(n)$ bounding from above the run time of $\mathtt{p}$ on inputs of length n.

We show how, given $\mathtt{p}$, to construct from $\mathtt{d}$, a Boolean program $\mathtt{q} = f(\mathtt{d})$ as desired. The construction is closely analogous to that used in Theorem 10.5, "specialised" to the known data input and space bound. Let $\mathtt{d} = \mathtt{a}_1 \ldots \mathtt{a}_n$ with each $\mathtt{a}_i \in \{0,1\}$.

Boolean variables. Program $\mathtt{q} = f(\mathtt{d})$ has a total of $(m+1)+6\pi(n)$ boolean variables, grouped as follows. Recall that any uninitialized variable has start value `false`.

Name	Intended interpretation: `true` iff	Index range
$\mathtt{L}_\ell$	Instruction ℓ of $\mathtt{p}$ is about to be simulated	$1 \le \ell \le m+1$
$\mathtt{T}_i^a$	TM square i (to left) holds symbol $a \in \{\mathtt{B},0,1\}$	$\perp\pi(n) \le i \le \pi(n)$

Relation to variables of Figure 10.1. First, the $\mathtt{q}$ variables are not indexed by time parameter t: the current time will be implicit in $\mathtt{q}$'s current control point. Omitting this gives $\mathtt{Label} : \{1,\ldots,m,m+1\}$, which can be represented by $m+1$ boolean variables $\mathtt{L}_1,\ldots,\mathtt{L}_{m+1}$. Similarly, Second, $\mathtt{Tape} : \mathbb{Z} \times \mathbb{N} \to \{0,1,\mathtt{B}\}$ is reduced to a function $\mathtt{T} : \mathbb{Z} \to \{0,1,\mathtt{B}\}$. A Boolean program is desired, so this is further split into three Boolean variables $\mathtt{T}^0, \mathtt{T}^1, \mathtt{T}^{\mathtt{B}} : \mathbb{Z} \to \{\mathtt{true},\mathtt{false}\}$. Since a computation with time bounded by polynomial $\pi(n)$ will satisfy $\perp\pi(n) \le i \le \pi(n)$, we represent each function $\mathtt{T}(i)$ by $6\pi(n)$ Boolean variables $\mathtt{T}_i^{\mathtt{a}}$. The intended relation is thus:

$$\mathtt{L}_\ell = \mathtt{true} \quad \text{iff} \quad \mathtt{Label}(t) = \ell$$
$$\mathtt{T}_i^{\mathtt{a}} = \mathtt{true} \quad \text{iff} \quad \mathtt{Tape}(i,t) = \mathtt{a}$$

```
L_1 := true;                                  Start simulation:  Turing instruction 1

    Assignments for initial tape contents:  blanks left and right of input

T_{⊥π(n)} :≡B; ... T_0 :≡ B;  T_1 :≡ a_1; ... T_n :≡ a_n;  T_{n+1} :≡ B; ... T_{π(n)} :≡B;

                        Simulate Turing machine instructions

while not L_{m+1} do STEP;   Answer := true
```

Figure 11.2: Turing simulation by Boolean program $\mathtt{q} = f(\mathtt{d})$.

Structure of program $\mathtt{q} = f(\mathtt{d})$**.** For brevity we use two abbreviations for multiple assignment ($\mathtt{a} \in \{0, 1, B\}$):

$\mathtt{T}_i :\equiv \mathtt{a}$; stands for $\mathtt{T}_i^0$:=false; $\mathtt{T}_i^1$:=false; $\mathtt{T}_i^B$:=false; $\mathtt{T}_i^a$:=true;
$\mathtt{T}_i :\equiv \mathtt{T}_j$; stands for $\mathtt{T}_i^0$:=$\mathtt{T}_j^0$; $\mathtt{T}_i^1$:=$\mathtt{T}_j^1$; $\mathtt{T}_i^B$:=$\mathtt{T}_j^B$;

In Figure 11.2, STEP is a sequence of Boole instructions simulating the effect of one Turing machine computational step. It does a "case on instruction" to select the appropriate instruction to be simulated in p. STEP is defined by:

$$\mathtt{STEP} = \mathtt{if}\ \mathtt{L}_1\mathtt{then}\ \overline{\mathtt{I}}_1;\ \mathtt{if}\ \mathtt{L}_2\mathtt{then}\ \overline{\mathtt{I}}_2; \ldots ;\ \mathtt{if}\ \mathtt{L}_m\mathtt{then}\ \overline{\mathtt{I}}_m;$$

Finally, Figure 11.3 defines the simulation of the individual instructions.

Lemma 11.9 Let $1 \le t \le \pi(n)$, and consider the state of Boolean program q just before executing the instructions in STEP. Then $\mathtt{L}_\ell$ will be true for exactly one ℓ, and for each i with $\perp\pi(n) \le i \le \pi(n)$, and Boolean variable $\mathtt{T}_i^a$ will be true for exactly one a.

Proof. This is immediate for $t = 1$ by the way q was constructed. Further, examination of the cases in Figure 11.3 shows that these properties are preserved in going from t to $t + 1$.

Lemma 11.10 At the end of execution, q will assign true to Answer if and only if p accepts d.

Proof. This is straightforward. "Only if" is by induction on Turing machine computation length, and "if" is by induction on the length of q.

Lemma 11.11 $\mathtt{q} = f(\mathtt{d})$ is constructible from d in space $O(\log |\mathtt{d}|)$.

Proof. Function f can be computed by one loop to generate the loop initialisations in q, and one loop over $t = 1, \ldots, \pi(n)$. Inside this loop for STEP with $\ell = 1, \ldots, m$, one more loop for $\perp\pi(n) \le i \le \pi(n)$ generates code for each Turing machine instruction type.

We now show Part 2 of Theorem 11.7.

Lemma 11.12 Let Turing machine program p run in polynomial time $\pi(n)$ on any input of length n. Then there is a function

$$f : \{0, 1\}^* \to \{\text{goto-free BOOLE programs}\}$$

Turing $\mathtt{I}_\ell$	Boolean $\overline{\mathtt{I}}_\ell$
`right`	$\mathtt{T}_{\perp\pi(n)} :\equiv \mathtt{T}_{\perp\pi(n)+1}; \ldots \mathtt{T}_0 :\equiv \mathtt{T}_1; \ldots; \mathtt{T}_{\pi(n)\perp 1} :\equiv \mathtt{T}_{\pi(n)};\ \mathtt{T}_{\pi(n)} :\equiv \mathtt{B};$ $\mathtt{L}_\ell := \mathtt{false};\ \mathtt{L}_{\ell+1} := \mathtt{true};$
`left`	$\mathtt{T}_{\pi(n)} :\equiv \mathtt{T}_{\pi(n)\perp 1}; \ldots \mathtt{T}_1 :\equiv \mathtt{T}_0; \ldots; \mathtt{T}_{\perp\pi(n)+1} :\equiv \mathtt{T}_{\perp\pi(n)};\ \mathtt{T}_{\perp\pi(n)} :\equiv \mathtt{B};$ $\mathtt{L}_\ell := \mathtt{false};\ \mathtt{L}_{\ell+1} := \mathtt{true};$
`write a`	$\mathtt{T}_0 :\equiv \mathtt{a};\ \mathtt{L}_\ell := \mathtt{false};\ \mathtt{L}_{\ell+1} := \mathtt{true};$
`if a goto` ℓ' `else` ℓ''	$\mathtt{L}_\ell := \mathtt{false};$ $\mathtt{L}'_\ell := \mathtt{T}^{\mathtt{a}}_0;\ \mathtt{L}_{\ell''} := \mathtt{not}\ \mathtt{T}^{\mathtt{a}}_0;$

Figure 11.3: Turing instructions simulated by sequences of Boolean instructions.

such that for any input $\mathtt{d} = \mathtt{a}_1 \ldots \mathtt{a}_n$ (each $\mathtt{a}_i \in \{\mathtt{0},\mathtt{1}\}$) we have

$\mathtt{p}$ accepts $\mathtt{d}$ if and only if $[\![f(\mathtt{d})]\!]^{\mathtt{BOOLE}} = \mathtt{true}$

Further, $f(\mathtt{d})$ is computable in space $O(\log|\mathtt{d}|)$.

Proof. Define $\mathtt{q} = f(\mathtt{d})$ as in Figure 11.2, but with one change: replace the final line

$$\mathtt{while\ not\ L}_{\mathtt{m}+1}\ \mathtt{do\ STEP;}\quad \mathtt{Answer := true}$$

by

$$\mathtt{STEP}^{\pi(n)};\quad \mathtt{Answer := true}$$

Clearly $\mathtt{q}$ is free of `goto`'s, and it is easy to see that it yields `true` if and only if Turing program $\mathtt{p}$ accepts input $\mathtt{d}$.

We now show Part 3 of Theorem 11.7.

Lemma 11.13 Let nondeterministic Turing machine program $\mathtt{p}$ run in polynomial time $\pi(n)$ on inputs of length n. Then there is a function

$g : \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{BOOLE}\ \text{programs}\}$

such that for any input $\mathtt{d} = \mathtt{a}_1 \ldots \mathtt{a}_n$ (each $\mathtt{a}_i \in \{\mathtt{0},\mathtt{1}\}$) we have

$\mathtt{p}$ accepts $\mathtt{d}$ if and only if for some Boolean program $\mathtt{b}$, $[\![\mathtt{b};f(\mathtt{d})]\!]^{\mathtt{BOOLE}} = \mathtt{true}$

Further, $g(\mathtt{d})$ is computable in space $O(\log|\mathtt{d}|)$.

Proof. The construction of Figure 11.3 needs extension, because nondeterministic program $\mathtt{p}$ may have the choice instruction `choose` ℓ' `or` ℓ''. Extending Figure 11.3 as follows.

Simulate instruction `choose` ℓ' `or` ℓ'' by Boolean instructions:

$$\mathtt{L}_\ell := \text{false; if Oracle}_t \text{ then } \mathtt{L}_{\ell'} := \text{true else } \mathtt{L}_{\ell''} := \text{true;}$$

Define $\mathtt{q} = g(\mathtt{d}) = \mathtt{Init};f(\mathtt{d})$ where $f(\mathtt{d})$ is as in Lemma 11.13, and `Init;` initialises to `false` all variables in $f(\mathtt{d})$ *except* the oracle variables $\mathtt{Oracle}_t$.

Now Turing program $\mathtt{p}$ accepts input $\mathtt{d}$ if and only if there exists a sequence of choices leading to success. Clearly this will be true if and only if for some assignemt $\mathtt{b}$ of truth values to the oracle variables, $\mathtt{b};\mathtt{Init};f(\mathtt{d})$ yields `true`.

12 Related Work

The book [20] by Kfoury, Moll and Arbib has similar aims, but [15] goes further in two respects: it covers complexity theory as well as computability; and it demonstrates the advantages that come from structuring *both programs and data*. [20] deals with structured programs, but uses the natural numbers as data.

Paul Voda is redeveloping recursion theory on the basis of Lisp-like data structures. A book is forthcoming; and [21] is a recent article.

References

[1] N. Andersen and N. D. Jones, Generalizing Cook's construction to imperative stack programs, Lecture Notes in Computer Science 812, pp. 1-18, Springer-Verlag, 1994.

[2] S. A. Cook, Linear-time simulation of deterministic two-way pushdown automata, *Information Processing* (IFIP) 71, C.V. Freiman, (ed.), North-Holland, pp. 75–80, 1971.

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

[4] S. A. Cook, The complexity of theorem-proving procedures, *Proceedings Third Symposium on the Theory of Computing*, pp. 151-158, ACM Press, 1971.

[5] A. Ben-Amram and N Jones. A precise version of a time hierarchy theorem. Fundamenta Informaticae, vol. 38, pp. 1-15. 1999.

[6] 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.

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

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

[9] 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.

[10] Hutton, G. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* **1** (1):1-17 (1953).

[11] N. D. Jones, Space-bounded reducibility among combinatorial problems, *Journal of Computer and System Science*, vol. 11, pp. 68–85, 1975.

[12] N. D. Jones, The Expressive Power of Higher-order Types or, Life without CONS, *Journal of Functional Programming* accepted for publication, 2000.

[13] N. D. Jones, A note on linear-time simulation of deterministic two-way pushdown automata, *Information Processing Letters* vol. 6, pp. 110-112, 1977.

[14] N. D. Jones. Constant time factors *do* matter. In Steven Homer, editor, *STOC '93. Symposium on Theory of Computing*, pages 602–611. ACM Press, 1993.

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

[16] N.D. Jones, C. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[17] Neil D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 1998.

[18] N. D. Jones and S. Muchnick. Even simple programs are hard to analyze. *Journal of the Association for Computing Machinery*, 24(2):338–350, 1977.

[19] N. D. Jones and S. Muchnick. Complexity of finite memory programs with recursion. *Journal of the Association for Computing Machinery*, 25(2):312–321, 1978.

[20] A.J Kfoury, R.N. Moll, and M.A. Arbib. *A Programming Approach to Computability*. Texts and monographs in Computer Science. Springer-Verlag, 1982.

[21] Paul Voda. Subrecursion as Basis for a Feasible Programming language. In Steven Homer, editor, *Logic in Comp. Science September 94*. Lecture Notes in Computer Science 933, Springer Verlag 1995.

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

Contents