Chapter 3

Exceptions and Functions

The goal of this chapter is to extend the basic ML-style language of the previous chapter towards more structured programming:

- exception handling,
- function calls.

3.1 Programs with Exceptions

Each time one wants to add a new kind of statement in the language, the set of rules for Hoare logic and WP calculi must be augmented. In this section, we want to support exceptions, which demands to generalize the notion of triple itself.

3.1.1 Syntax

The exceptions are given by a set of identifiers exn. The grammar of expressions is enriched with two new statements:

$$e \quad ::= \quad \texttt{raise}\ exn \mid \texttt{try}\ e\ \texttt{with}\ exn \Rightarrow e$$

3.1.2 Operational Semantics

Previously, a value represented the end of an execution. With exceptions, a value represents the *normal* end of an execution, while the statements `raise` exn represent *exceptional* ends.

The former small-step rules are kept the same, and we add the following new ones.

$$\frac{}{\Sigma, \Pi, (\texttt{let}\ x = \texttt{raise}\ exn\ \texttt{in}\ e) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ exn}$$

$$\frac{}{\Sigma, \Pi, (\texttt{try}\ v\ \texttt{with}\ exn \Rightarrow e') \rightsquigarrow \Sigma, \Pi, v}$$

$$\frac{}{\Sigma, \Pi, (\texttt{try}\ \texttt{raise}\ exn\ \texttt{with}\ exn \Rightarrow e) \rightsquigarrow \Sigma, \Pi, e}$$

$$\frac{exn \neq exn'}{\Sigma, \Pi, (\texttt{try}\ \texttt{raise}\ exn\ \texttt{with}\ exn' \Rightarrow e) \rightsquigarrow \Sigma, \Pi, \texttt{raise}\ exn}$$

$$\frac{\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'}{\Sigma, \Pi, (\texttt{try}\ e\ \texttt{with}\ exn \Rightarrow e'') \rightsquigarrow \Sigma', \Pi', (\texttt{try}\ e'\ \texttt{with}\ exn \Rightarrow e'')}$$

3.1.3 Hoare Triples and WP Rules

The notation for Hoare triple must be modified to take into account *exceptional post-conditions*. It now has the form $\{P\}e\{Q \mid exn_i \Rightarrow R_i\}$ and its validity has the following meaning. If e is executed in a state where P holds, then it does not block and

- if it terminates normally with some value v in some state Σ, then $Q[\mathsf{result} \leftarrow v]$ holds in Σ;
- if it terminates with some exception exn in some state Σ, then there is some i such that $exn = exn_i$ and R_i holds in Σ.

Note that this definition implies that if e terminates with some exception exn not in the set $\{exn_i\}$, then the triple is not valid.

The WP function is also modified to take the set of exceptional post-conditions as argument: $\mathrm{WP}(e, Q, exn_i \Rightarrow R_i)$. Note that, if an exception is not in the set $\{exn_i\}$, the associated post-condition is implicitly set to false.

$$
\begin{aligned}
\mathrm{WP}(\texttt{raise}\ exn_k, Q, exn_i \Rightarrow R_i) &= R_k \\
\mathrm{WP}(x := t, Q, exn_i \Rightarrow R_i) &= Q[\mathsf{result} \leftarrow (), x \leftarrow t] \\
\mathrm{WP}(\texttt{assert}\ R, Q, exn_i \Rightarrow R_i) &= R \wedge Q \\
\mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, Q, exn_i \Rightarrow R_i) &= \\
&\quad \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, exn_i \Rightarrow R_i)[\mathsf{result} \leftarrow x], exn_i \Rightarrow R_i) \\
\mathrm{WP}(\texttt{if}\ t\ \texttt{then}\ e_1\ \texttt{else}\ e_2, Q, exn_i \Rightarrow R_i) &= \\
&\quad \texttt{if}\ t\ \texttt{then}\ \mathrm{WP}(e_1, Q, exn_i \Rightarrow R_i)\ \texttt{else}\ \mathrm{WP}(e_2, Q, exn_i \Rightarrow R_i) \\
\mathrm{WP}\left(\begin{array}{l}\texttt{while}\ c\ \texttt{invariant}\ I \\ \texttt{variant}\ v, \prec\ \texttt{do}\ e\end{array}, Q, exn_i \Rightarrow R_i\right) &= I \wedge \forall v_1, \ldots, v_k, \\
&\quad (I \wedge \texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(L : e, I \wedge v \prec v@L, exn_i \Rightarrow R_i)\ \texttt{else}\ Q)[w_i \leftarrow v_i] \\
&\quad \text{where } w_1, \ldots, w_k \text{ is the set of assigned variables in} \\
&\quad \text{expressions, } v_1, \ldots, v_k \text{ are fresh logic variables,} \\
&\quad \text{and } L \text{ is a fresh label.} \\
\mathrm{WP}((\texttt{try}\ e_1\ \texttt{with}\ exn \Rightarrow e_2), Q, exn_i \Rightarrow R_i) &= \\
&\quad \mathrm{WP}\left(e_1, Q, \left\{\begin{array}{l} exn \Rightarrow \mathrm{WP}(e_2, Q, exn_i \Rightarrow R_i) \\ exn_i \backslash exn \Rightarrow R_i \end{array}\right.\right)
\end{aligned}
$$

3.2 Functions, Modular Verification

Non-trivial programs are structured into sub-programs (functions, procedures, etc.) and at a higher level into modules. Visibility rules (local variables, local procedures, etc.) allow the programmer to hide implementation details from the callers of a sub-program.

When proving such a structured program, one naturally expects a *modular* proof that follows the modular structure of the program. Thus, reasoning on a sub-program call should consider an *abstract* view of the behavior of that sub-program: a specification of what it does, without telling how it does it. The notation of Hoare triples is a good candidate for an abstraction of a given sub-program. In the literature, it is known as the *subprogram contract*, a term that was first used in the context of the Eiffel language [2].

We now add to the language the notion of functions and function calls. These functions have parameters that are seen as immutable local variables. Procedures will just be a special case of functions, which return values will not carry any meaningful information, that is, they will be of type `unit`.

3.2.1 Syntax

The syntax of programs is given as follows. A program is a sequence of declarations. Each declaration is either the declaration of a global variable, whose type is a reference, or the declaration of a function. A function is declared with a possibly empty list of parameters, whose types are base types (not reference), a return type, a contract, and a body. The contract is made of the precondition, the list of global variables possibly modified, and the postcondition.

$$
\begin{array}{rcl}
prog & ::= & decl^* \\
decl & ::= & vardecl \mid fundecl \\
vardecl & ::= & \mathsf{val}\ id : \mathsf{ref}\ basetype \\
fundecl & ::= & \mathsf{function}\ id(\ (param,)^*\ ) {:} basetype\ contract\ \mathsf{body}\ e \\
param & ::= & id : basetype \\
contract & ::= & \mathsf{requires}\ t\ \mathsf{writes}\ (id,)^*\ \mathsf{ensures}\ t
\end{array}
$$

The reserved label *Old* is allowed in the postcondition of the contract, and also the result keyword to denote the returned value.

The syntax of expressions is augmented with function call. As with binary operators before, we avoid issues with side effects and evaluation order in function arguments by forcing them to be values (e.g. constants or let-binding immutable variables).

$$e ::= id(\ (t,)^*\ )$$

Example 3.2.1 *The following is a program where our ISQRT example is now a procedure that takes its argument as a parameter. It also contains a simple test procedure using a global variable.*

```
function isqrt(x:int): int
  requires x ≥ 0
  ensures result ≥ 0 ∧ sqr(result) ≤ x < sqr(result + 1)
body
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done;
  res

val res : ref int

procedure test()
  requires true
  writes res
  ensures res = 6
body
  res := isqrt(42)
```

3.2.2 Typing Rules for Functions

The definition of a function f has the following form.

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires *Pre*
  writes $\vec{w}$
  ensures *Post*
  body *Body*

The rule expressing that such a declaration d is well-formed is

$$\frac{\Gamma' = \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma \qquad \Gamma' \vdash \mathit{Pre}, \mathit{Post} : \mathit{formula} \qquad \vec{w} \subseteq \Gamma \qquad \Gamma' \vdash \mathit{Body} : \tau}{\Gamma \vdash d : \mathit{wf}}$$

where Γ contains the declaration of global references. Notice that *Post* does not have access to local variables that are declared inside the body.

The typing rule for a call to f is

$$\frac{\Gamma \vdash t_i : \tau_i}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau}$$

Notice that we allow recursive calls. We even allow mutually recursive functions.

3.2.3 Operational Semantics

In order to define the small-step semantics of a function call, we need a new pseudo-operation `return` which carries the local state of the caller during the call and restores it at the end. It also checks that a given property P holds for the returned value.

$$\frac{\Sigma, \Pi \models P[\mathsf{result} \leftarrow t]}{\Sigma, \Pi, (\texttt{return}\ t, P, \Pi') \rightsquigarrow \Sigma, \Pi', [\![t]\!]_{\Sigma,\Pi}}$$

The semantics of the call itself follows.

$$\frac{\Pi' = \{x_i \mapsto [\![t_i]\!]_{\Sigma,\Pi}\} \qquad \Sigma, \Pi' \models \mathit{Pre}}{\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi', (\mathit{Old} : \texttt{let}\ \xi = \mathit{Body}\ \texttt{in return}\ \xi, \mathit{Post}, \Pi)}$$

with ξ a fresh identifier. Note that the *Old* label above is the one used in *Post*, but it should be assumed to be fresh for all other purposes, that is, no label defined during the evaluation of *Body* will conflict with it. This special label is provided so that the postcondition can mention the values the variables had at the start of the function.

Basically, the execution of a function call starts by checking that the precondition holds. It then executes the body in a fresh local state that only associates the argument variables to the passed values. Finally, the postcondition is checked and the local state and the function result is returned.

Notice that, as in the previous chapter, we define a blocking semantics for annotations (the pre- and the postcondition): the execution blocks if any annotation does not hold.

Example 3.2.2 *Here is a toy procedure that increments the global variable* res *by a given amount.*

```
val res: ref int

procedure incr(x:int)
  requires true
  writes res
  ensures res = res@Old + x
body
  res := res + x
```

3.2.4 Weakest Preconditions for Function Calls

A simple idea to prove a function call in Hoare logic would be to replace the call by the body of the function. This is a bad idea, as it would mean we would reprove the safety of the body each time the function is called. What we want to do is to prove once and for all that the body of a function satisfies its contract, and then use only the contract when reasoning on function calls. This is a *modular* approach. Moreover, it makes it possible to reason about programs that call functions which body has been abstracted away.

The WP rule that expresses this idea is the following:

$$\mathrm{WP}(f(t_1,\ldots,t_n),Q) = \mathit{Pre}[x_i \leftarrow t_i] \wedge \forall \vec{v},\ (\mathit{Post}[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@\mathit{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j])$$

with $\vec{w}$ the set of variables written by the function body. Reminder: Argument terms t_i are assumed not to contain any mutable variables, so that their values for *Pre* and *Post* are equal.

3.2.5 Soundness

The soundness theorem must be stated in a slightly different way than in the previous chapter: we need to express the idea of modularity of proofs. This is done by stating the following global hypothesis: for each function declared as

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires *Pre*
  writes $\vec{w}$
  ensures *Post*
  body *Body*

we assume that

1. the variables assigned in *Body* belong to $\vec{w}$,
2. $\models \mathit{Pre} \Rightarrow \mathrm{WP}(\mathit{Body}, \mathit{Post})[w_i@\mathit{Old} \leftarrow w_i]$.

Theorem 3.2.3 *Assuming the global hypothesis above holds, then the WP rules are sound.*

Proof. First of all, we have to give the weakest precondition of the `return` statement we introduced to describe the operational semantics of function calls. Note that it will never be part of an actual WP computation, since `return` is not part of the user language. This rule is needed for this proof of soundness only.

$$\mathrm{WP}((\texttt{return}\ t, P, \Pi), Q) = P[\mathsf{result} \leftarrow t] \wedge Q[\mathsf{result} \leftarrow t, \ell_i \leftarrow [\![\ell_i]\!]_\Pi]$$

with ℓ_i representing local variables of Q.

For our WP rules to be sound, we have to prove two properties: progress and preservation by reduction.

1. If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ and e is not a value then there exists Σ', Π', e' such that $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$.
2. If $\Sigma, \Pi \models \mathrm{WP}(e, Q)$ and $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$ then $\Sigma', \Pi' \models \mathrm{WP}(e', Q)$.

In the previous chapter, progress was proved by structural induction on all the expressions of the language (except for function call and `return`, which were added in this chapter). Preservation by reduction was proved by predicate induction on the operational semantics for all the reduction rules (except the two new rules added for function call and `return`). So we only have to handle these four new cases to complete the previous proofs.

Progress for `return`. Let us assume

$$\begin{aligned}\Sigma, \Pi &\models \mathrm{WP}((\texttt{return}\, t, P, \Pi'), Q)\\ &\models P[\mathsf{result} \leftarrow t] \wedge Q[\mathsf{result} \leftarrow t, \ell_i \leftarrow [\![\ell_i]\!]_{\Pi'}]\\ &\models P[\mathsf{result} \leftarrow t]\end{aligned}$$

which is exactly the hypothesis of the reduction rule for `return`. So this expression progresses.

Preservation by reduction for `return`. Let us assume

$$\begin{aligned}\Sigma, \Pi &\models \mathrm{WP}((\texttt{return}\, t, P, \Pi'), Q)\\ &\models P[\mathsf{result} \leftarrow t] \wedge Q[\mathsf{result} \leftarrow t, \ell_i \leftarrow [\![\ell_i]\!]_{\Pi'}]\\ &\models Q[\mathsf{result} \leftarrow [\![t]\!]_{\Sigma,\Pi}, \ell_i \leftarrow [\![\ell_i]\!]_{\Pi'}]\end{aligned}$$

Since $Q[\mathsf{result} \leftarrow [\![t]\!]_{\Sigma,\Pi}]$ contains no local variables except for the ℓ_i, it holds in state Σ, Π'. Thus $\Sigma, \Pi' \models \mathrm{WP}([\![t]\!]_{\Sigma,\Pi}, Q)$. So WP is preserved for this reduction rule.

Progress for function call. Let us assume

$$\begin{aligned}\Sigma, \Pi &\models \mathrm{WP}(f(t_1, \cdots, t_n), Q)\\ &\models \mathit{Pre}[x_i \leftarrow t_i] \wedge \forall \vec{v},\ (\mathit{Post}[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@\mathit{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j])\\ &\models \mathit{Pre}[x_i \leftarrow [\![t_i]\!]_{\Sigma,\Pi}]\end{aligned}$$

Since *Pre* contains no local variables except for the x_i, it holds in state Σ, Π', with $\Pi' = \{x_i \mapsto [\![t_i]\!]_{\Sigma,\Pi}\}$. This is the hypothesis of the reduction rule for function call, so the expression progresses.

Preservation by reduction for function call. Let us assume

$$\begin{aligned}\Sigma, \Pi &\models \mathrm{WP}(f(t_1, \cdots, t_n), Q)\\ &\models \mathit{Pre}[x_i \leftarrow t_i] \wedge \forall \vec{v},\ (\mathit{Post}[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@\mathit{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j])\end{aligned}$$

We have to prove

$$\begin{aligned}\Sigma, \Pi' &\models \mathrm{WP}((\mathit{Old} : \texttt{let}\, \xi = \mathit{Body}\, \texttt{in return}\, \xi, \mathit{Post}, \Pi), Q)\\ &\models \mathrm{WP}((\texttt{let}\, \xi = \mathit{Body}\, \texttt{in return}\, \xi, \mathit{Post}, \Pi), Q)[w_j@\mathit{Old} \leftarrow w_j]\\ &\models \mathrm{WP}(\mathit{Body}, \mathrm{WP}(\texttt{return}\, \xi, \mathit{Post}, \Pi, Q)[\xi \leftarrow \mathsf{result}])[w_j@\mathit{Old} \leftarrow w_j]\\ &\models \mathrm{WP}(\mathit{Body}, (\mathit{Post}[\mathsf{result} \leftarrow \xi] \wedge Q[\mathsf{result} \leftarrow \xi, \ell_i \leftarrow [\![\ell_i]\!]_\Pi])[\xi \leftarrow \mathsf{result}])[w_j@\mathit{Old} \leftarrow w_j]\\ &\models \mathrm{WP}(\mathit{Body}, (\mathit{Post} \wedge Q[\ell_i \leftarrow [\![\ell_i]\!]_\Pi]))[w_j@\mathit{Old} \leftarrow w_j]\\ &\models \mathrm{WP}(\mathit{Body}, \mathit{Post})[w_j@\mathit{Old} \leftarrow w_j] \wedge \mathrm{WP}(\mathit{Body}, Q[\ell_i \leftarrow [\![\ell_i]\!]_\Pi])[w_j@\mathit{Old} \leftarrow w_j]\end{aligned}$$

The last line was obtained by applying Lemma 2.3.2 about conjunctions and WP. Note that the substitution $w_j@\mathit{Old} \leftarrow w_j$ does not hinder this application, as we could have chosen a global state $\Sigma' = \Sigma\{(w_j, \mathit{Old}) \mapsto [\![w_j]\!]_\Sigma\}$.

The left-hand side of the conjunction is a consequence of the global hypothesis, since *Pre* holds in Σ, Π' (see progress above). The right-hand side amounts to proving that $\mathrm{WP}(\mathit{Body}, Q)$ holds in state Σ', Π'' with $\Pi'' = \Pi'\{\ell_i \mapsto [\![\ell_i]\!]_\Pi\}$. We apply Lemma 2.3.3 about WP and monotonicity, which requires to prove

$$\Sigma', \Pi'' \models \mathrm{WP}(\mathit{Body}, \mathit{Post}) \wedge \forall \vec{v},\ (\mathit{Post} \Rightarrow Q)[w_j \leftarrow v_j]$$

The left-hand side can be expressed as $\Sigma, \Pi' \models \mathrm{WP}(\mathit{Body}, \mathit{Post})[w_j \leftarrow w_j@\mathit{Old}]$, since none of the ℓ_i occurs in it. So we have already proved that it holds.

The right-hand side can be expressed as $\Sigma, \Pi \models \forall \vec{v},\ (\mathit{Post}[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@\mathit{Old} \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j])$, which is the hypothesis for proving the preservation.

3.3 Functions Throwing Exceptions

If a function may raise an exception, then this should be mentioned in its contract. A generalized contract has the form

requires *Pre*
writes $\vec{w}$
raises $E_1 \cdots E_n$
ensures *Post* | $E_1 \to$ *Post*$_1$ | $\cdots$ | $E_n \to$ *Post*$_n$

It states that the exceptions $E_1 \cdots E_n$ may be raised by the function, and no others. Moreover, if it terminates normally then the formula *Post* holds; if it terminates in exception E_i then the formula *Post*$_i$ holds.

The rule for WP is extended to exceptional cases in a natural way:

$$\begin{aligned}\mathrm{WP}(f(t_1,\ldots,t_n), Q, E_k \Rightarrow R_k) = \mathit{Pre}[x_i \leftarrow t_i] \wedge \forall \vec{v},\\ (\mathit{Post}[x_i \leftarrow t_i, w_j \leftarrow v_j] \Rightarrow Q[w_j \leftarrow v_j]) \wedge\\ \textstyle\bigwedge_k (\mathit{Post}_k[x_i \leftarrow t_i, w_j \leftarrow v_j] \Rightarrow R_k[w_j \leftarrow v_j])\end{aligned}$$

Example 3.3.1 *The following program is a "defensive" variant of our ISQRT example. Instead of requiring a non-negative argument, and returning a result rounded down, it raises an exception when the argument is negative or it is not a perfect square.*

```
exception NotSquare

function isqrt(x:int): int
  requires true
  raises NotSquare
  ensures result ≥ 0 ∧ sqr(result) = x
        | NotSquare → forall n:int. sqr(n) ≠ x
body
  if x < 0 then raise NotSquare;
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1; sum := sum + 2 * res + 1
  done;
  if res * res ≠ x then raise NotSquare;
  res
```

3.4 Recursive Functions and Termination

For recursive procedures, in a similar way as for loops, we need to add a variant to exhibit a measure that decreases between recursive calls. We allow to add a *variant clause* in the contracts, of the form

variant v for $\prec$

where v is a term of some type τ and $\prec$ is a well-founded relation on values of type τ.

The formula for the weakest precondition of a recursive call is

$$\mathrm{WP}(f(t_1,\ldots,t_n), Q, E_i \Rightarrow Q_i) = \mathit{Pre}[x_i \leftarrow t_i] \wedge v[x_i \leftarrow t_i] \prec v@\mathit{Init} \wedge \forall \vec{y}, \ldots$$

where $Init$ is a label placed at the beginning of the body of the procedure. It thus states that the expression v, instantiated with the arguments of the call, is smaller than it was at the entrance to the procedure.

The same kind of WP rule is able to handle the case of mutually recursive functions. If two functions $f(\vec{x})$ and $g(\vec{y})$ may call each other, then each of them should be given its own variant v_f (resp. v_g) in their contract, but with the same well-founded ordering $\prec$. Then, when f calls $g(\vec{t})$ the WP should include

$$v_g[\vec{y} \leftarrow \vec{t}] \prec v_f@Init.$$

and symmetrically when g calls f

It generalizes naturally to any number of functions that can call each other recursively.

Example 3.4.1 *The following is an implementation of the factorial function. For simplicity we do not give any functional property in postcondition.*

```
function fact(x:int): int
  requires x ≥ 0
  variant x
  ensures true
body
  if x = 0 then 1 else fact(x-1) * x
```

We do not precise the ordering since we use the default one introduced in Chapter 1. The WP at the recursive call is

$$(x \geq 0)[x \leftarrow x - 1] \wedge x[x \leftarrow x - 1] \prec x@Init$$

which reduces to

$$x - 1 \geq 0 \wedge x - 1 < x \wedge x \geq 0$$

which is valid under the premises $x \geq 0$ (the precondition) and $x \neq 0$ (the negation of the condition of the if).

3.5 Exercises

Exercise 3.5.1 *Using the* `incr` *procedure of Example 3.2.2, prove the test*

```
procedure test()
  requires res = 36
  writes res
  ensures res = 42
body
  incr(6)
```

using WP.

Exercise 3.5.2 *The McCarthy's 91 function [1] is defined on non-negative integers by the recursive equation*

$$f91(n) = \texttt{if } n \leq 100 \texttt{ then } f91(f91(n + 11)) \texttt{ else } n - 10$$

The following is a canvas for a function that computes $f91(n)$.

```
function f91 (n:int) : int
  requires ?
  variant ?
  writes ?
  ensures ?
body
  if n ≤ 100 then f91 (f91 (n + 11))
  else n - 10
```

1. *Fill-in the contract in order to prove the correctness of this function.*
2. *Would it be possible to prove the total correctness using only true as postcondition?*

Bibliography

[1] Z. Manna and J. McCarthy. Properties of programs and partial function logic. In *Machine Intelligence*, volume 5, pages 79–98, 1970.

[2] B. Meyer. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, 1992.