Propagation Logic & Exception Handling

Dewayne E Perry
ENS 623A
Office Hours: T/Th 11:00-12:00
perry @ ece.utexas.edu
www.ece.utexas.edu/~perry/education/382v-s06/

One Direction of Inscape Research

- ➔ Make practical use of specification and verification technology
  - ↳ Use the module interface specifications in the construction of software systems
  - ↳ Implement a semantic interconnection model that records unit, syntactic and semantic dependencies as the basis for evolution
  - ↳ Make practical trade-offs between logic and analysis
    - ➢ how weak a logic?
    - ➢ how strong a form of consistency checking?
- ➔ Instress — the module interface specification language
  - ↳ Specification logic (SL), C syntax for declarations
  - ↳ SL annotations for data object properties and operation interface behavior
- ➔ Inscape — using the specifications
  - ↳ Data object/operation specification instantiation
    - ➢ as the basis for the semantic interconnection and propagation
    - ➢ as a result of variable names and operation arguments
  - ↳ Propagation logic (PL)
    - ➢ construct interfaces for *sequence*, *selection*, *iteration*
    - ➢ construct interfaces for implementations of operations

---

Goals of Analysis

- ➔ Incremental Analysis
  - ↳ Use specifications as a bootstrapping mechanism — ie, assume that they are correct and use them
  - ↳ An operation is the gross grain of incremental analysis (the collection of operations individually analyzed forms the analysis of a module)
  - ↳ A statement is the fine grain of incremental analysis (the goal is to be able to consider each statement independent of its surrounding context)
- ➔ Basic Rule:
  - ↳ *Every precondition and obligation must either be satisfied within the implementation or propagated to its interface.*

Some Logical Intuition

- ➔ Separate *consistency* from *propagation*
  - ↳ Consistency depends on trade-offs about how to manipulate the specification logic
  - ↳ Propagation depends on trade-offs bout how to statically represent a possible-execution tree as a *single thread of knowledge*
- ➔ Temptation to use *or*
  - ↳ In reducing a possible-execution tree to a static (sequential) directed graph, how do you handle the results of joining two paths?
  - ↳ Say *P* is true in one path, *Q* in the other — one is tempted to describe the result as *P or Q*
  - ↳ However, if *not P* later becomes true, the inference of *Q* is not a valid one, because it might well have been the path that produced *P* that was actually executed.

Some More Intuition

- Why we can treat each statement/operation as an indivisible unit
  - Intuitive picture of preconditions and postconditions
    - postconditions "sink down" through an implementation
    - preconditions "float up" through an implementation "looking" for satisfaction
  - At the point where a precondition P occurs, either P is known to be true or false, or it is not known whether P is true or false
    - if P is true, then the precondition is satisfied (and it does not matter where in the preceding sequence it became true)
    - if P is false, P cannot be propagated to the interface and hence there is a problem with the implementation (it also does not matter where P became false, except to provide a range in which to fix the problem)
    - if P is unknown, then it is unknown "in" all of the preceding statements as well
  - Unfortunately, obligations are not quite so tractable

Inscape's Propagation Logic (PL)

- PL is a proposition calculus (SL is a predicate calculus with quantification) in which
  - A proposition P is either *true* (P) or *false* (¬P)
  - The state of a proposition is either
    - *unknown P* is not known to be either true or false in any execution path
    - *known P* is known to be true in all execution paths
    - *possible P* is known to be true in at least one execution path
  - There is one sentence connective, and
  - There are the inference rules
    - *+seq* a sequential addition or join based on the state of the propositions
    - *+par* a parallel addition or join based on the state of the propositions
    - *+con* a sequential addition or join based on the consistency of the propositions

---

Inscape's Propagation Logic (PL)

- Also used in the description of propagation are
  - Set theory operations: ∊, ∪, ∩, ⊂, ⊆, =, and −
  - Operations from SL:
    - *consistent* P is consistent with Q
    - *isknown* P is known in Q

The Definition of +seq

- P1 +seq P2 is defined as follows

P1	P2	Result
*P	known P	known P
*P	known ¬P	known ¬P
*P	unknown P, ¬P	*P
known P	possible P	known P
known P	possible ¬P	possible P, possible ¬P
known P	possible ¬P	possible P, possible ¬P
possible P	possible ¬P	possible P, possible ¬P
possible P	possible P	possible P
unknown P	possible P	possible P
unknown P	possible ¬P	possible ¬P

- Note: P +seq Q is not symmetric
- The state of P2 supercedes the state of P1.
  - whatever is known in P2 supplants that of P1.
  - whatever in P1 is unknown in P2 retains its state from P1
  - what is possible in P2 remains so in the result, but may also reduce what is know in P1 to a possible in the result
  - except where it was known in P1

The Definition of +par

➔ **P1 +par P2** is symmetric and defined as follows

P1	P2	Result
known P (¬P)	known P (¬P)	known P (¬P)
known P (¬P)	unknown P (¬P)	possible P (¬P)
known P (¬P)	possible P (¬P)	possible P (¬P)
known P	known ¬P	possible P, possible ¬P
known P	possible ¬P	possible P, possible ¬P
unknown P, ¬P	unknown P, ¬P	unknown P, ¬P
unknown P (¬P)	possible P (¬P)	possible P (¬P)
possible P	possible ¬P	possible P, possible ¬P

➔ Only what is known (unknown) to be true in both parts is known (unknown) to be true in a parallel join of the two parts. What is known in only one part becomes possible in the result.

The Definition of +con and -con

➔ **P1 +con P2** is defined as

{p1...pk ∊ P1 | p1...pk are consistent P2} ∪ P2

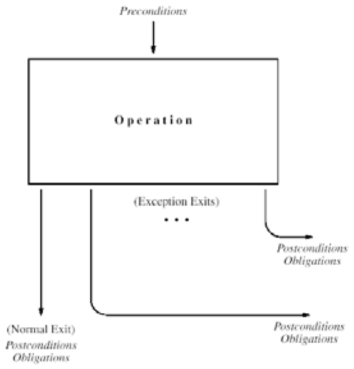
that is, the result of +con is P2 plus those propositions in P1 that are consistent with P2

➔ **P1 -con P2** is defined as

P1 - (P1 +con P2)

that is, the result of -con are those propositions in P1 that are inconsistent with P2

The Basis for Propagation



Reasoning about Sequences

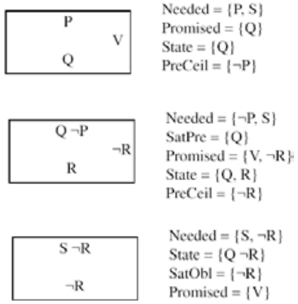
➔ The following sets are used for reasoning about each Sequent S_i in the Sequence $S = S_1 .. S_n$

Pre$_i$	the set of preconditions for sequent$_i$
Post$_i$	the set of postconditions for sequent$_i$
Obl$_i$	the set of obligations for sequent$_i$
PreCeil$_i$	the preconditions ceilinged by sequent$_i$
OblFloor$_i$	the obligations floored by sequent$_i$
State$_i$	the current state after sequent$_i$
Promised$_i$	the set of obligations outstanding after sequent$_i$
Needed$_i$	the accumulated set of unsatisfied preconditions up to and including sequent$_i$ (from sequent$_n$)
SatPre$_i$	the satisfied preconditions for sequent$_i$
UnsatPre$_i$	the unsatisfied preconditions for sequent$_i$
SatObl$_i$	the satisfied obligations for sequent$_i$

Sequents

- ➔ For the Sequence S there is an initial State_0 and Promised_0
- ➔ For each Sequent Si in Sequence $S = S_1 .. S_n$
  - $\text{State}_i = \text{State}_{i-1}$ +seq Post_i
  - $\text{SatPre}_i = \{ P \in \text{Pre}_i \mid P$ is satisfied by $\text{State}_{i-1} \}$
  - $\text{UnsatPre}_i = \text{Pre}_i - \text{SatPre}_i$
  - $\text{Needed}_i = (\text{Needed}_{i+1} - \text{PreCeil}_i) \cup \text{UnsatPre}_i$
  - $\text{PreCeil}_i = (\text{Needed}_{i+1}$ -con $\text{Post}_i) \cup (\text{Needed}_{i+1}$ -con $\text{Pre}_i)$
  - $\text{SatObl}_i = \{ O \in \text{Obl}_i \mid O$ is satisfied by $\text{State}_i \}$
  - $\text{Promised}_i = (\text{Promised}_{i-1} - \text{OblFloor}_i) \cup \text{UnsatObl}_i$
  - $\text{OblFloor}_i = \text{Promised}_{i-1}$ -con Obl_i

Sequence Example



Needed = {P, S}
Promised = {Q}
State = {Q}
PreCeil = {¬P}

Needed = {¬P, S}
SatPre = {Q}
Promised = {V, ¬R}
State = {Q, R}
PreCeil = {¬R}

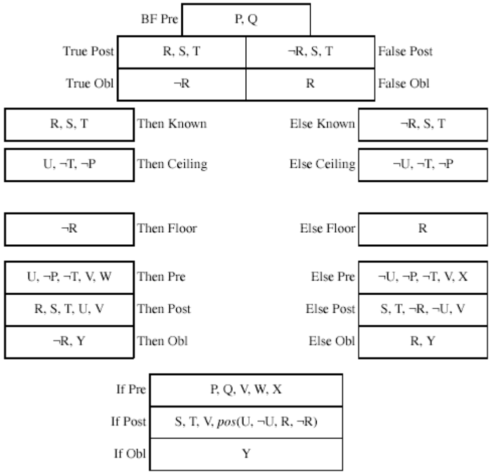
Needed = {S, ¬R}
State = {Q ¬R}
SatObl = {¬R}
Promised = {V}

Sequence

- ➔ Let the Sequence $S = S_1 .. S_n$ where State_0 and Promised_0 have been initialized according to the context of the sequence.
- ➔ The interface for S is propagated as follows:
  - ↳ $S.\text{Pre} = \text{Needed}_1$
  - ↳ $S.\text{Post} = \text{State}_n$
  - ↳ $S.\text{Obl} = \text{Promised}_n$
- ➔ The content of S.PreCeil and S.OblFloor may be amended according to the context of the use of the sequence S

Selection (IF)

- ➔ Let the Selection Statement S consist of BE = the boolean expression, T = the then sequence, and E = the else sequence where
  - $T.\text{State}_0 = \text{BE.True.Post}$
  - $E.\text{State}_0 = \text{BE.False.Post}$
  - $T.\text{Promised}_0 = \text{BE.True.Obl}$
  - $E.\text{Promised}_0 = \text{BE.False.Obl}$
- ➔ The interface for S is propagated as follows:
  - $S.\text{Pre} = \text{BE.Pre} \cup (T.\text{Pre} - T.\text{PreCeil}) \cup (E.\text{Pre} - E.\text{PreCeil})$
  - $S.\text{Post} = T.\text{Post}$ +par $E.\text{Post}$
  - $S.\text{Obl} = T.\text{Obl} \cap E.\text{Obl}$
- ➔ The state of the selection statement S is amended as follows:
  - $T.\text{OblFloor} = T.\text{Obl} - S.\text{Obl}$
  - $E.\text{OblFloor} = E.\text{Obl} - S.\text{Obl}$
  - $T.\text{PreCeil} = (T.\text{Pre}$ -con $\text{BE.True.Post}) \cup (T.\text{Pre}$ -con $E.\text{Pre}) \cup (T.\text{Pre}$ -con $B.\text{Pre})$
  - $E.\text{PreCeil} = (E.\text{Pre}$ -con $\text{BE.False.Post}) \cup (E.\text{Pre}$ -con $T.\text{Pre}) \cup (E.\text{Pre}$ -con $B.\text{Pre})$



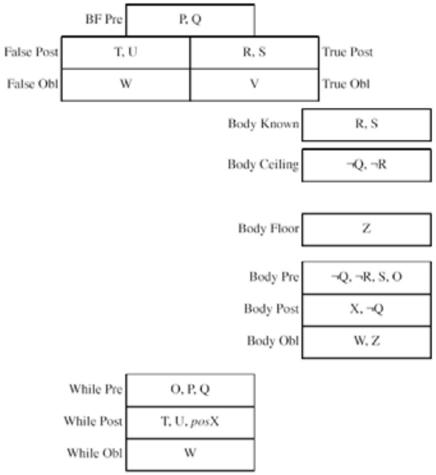
BF Pre	P, Q		
True Post	R, S, T	¬R, S, T	False Post
True Obl	¬R	R	False Obl

R, S, T	Then Known	Else Known	¬R, S, T
U, ¬T, ¬P	Then Ceiling	Else Ceiling	¬U, ¬T, ¬P
¬R	Then Floor	Else Floor	R
U, ¬P, ¬T, V, W	Then Pre	Else Pre	¬U, ¬P, ¬T, V, X
R, S, T, U, V	Then Post	Else Post	S, T, ¬R, ¬U, V
¬R, Y	Then Obl	Else Obl	R, Y

If Pre	P, Q, V, W, X
If Post	S, T, V, *pos*(U, ¬U, R, ¬R)
If Obl	Y

Iteration [While]

- ➔ Let the Iteration Statement W consist of BE = the boolean expression, and B = the loop body (a sequence) where
  - $B.State_0 = BE.True.Post$
  - $B.Promised_0 = BE.True.Obl$
- ➔ The interface for I is propagated as follows:
  - $W.Pre = BE.Pre \cup (B.Pre - B.PreCeil)$
  - $W.Post = (B.Post +par \emptyset) +seq BE.False.Post$
  - $W.Obl = BE.False.Obl$
- ➔ The state of the loop body B is amended as follows:
  - $B.OblFloor = B.Obl$
  - $B.PreCeil = (B.Pre -con BE.True.Post) \cup (B.Pre -con BE.Pre) \cup (B.Pre -con B.Post)$
- ➔ There is an Error when $W.Pre -con B.Post \neq \emptyset$



	BF Pre	P, Q	
False Post	T, U	R, S	True Post
False Obl	W	V	True Obl

Body Known	R, S
Body Ceiling	¬Q, ¬R
Body Floor	Z
Body Pre	¬Q, ¬R, S, O
Body Post	X, ¬Q
Body Obl	W, Z

While Pre	O, P, Q
While Post	T, U, *pos*X
While Obl	W

Iteration [Repeat]

- ➔ Let the Iteration Statement R consist of BE = the boolean expression, and B = the loop body (a sequence) where
  - $B.State_0 = BE.True.Post$
  - $B.Promised_0 = BE.True.Obl$
- ➔ The interface for I is propagated as follows:
  - $R.Pre = BE.Pre \cup (B.Pre - B.PreCeil)$
  - $R.Post = B.Post +seq BE.False.Post$
  - $R.Obl = B.Obl +seq BE.False.Obl$
- ➔ The state of the loop body B is amended as follows:
  - $B.OblFloor = B.Obl - I.Obl$
  - $B.PreCeil = (B.Pre -con BE.True.Post) \cup (B.Pre -con BE.Pre) \cup (B.Pre -con B.Post)$
- ➔ There is an Error when $R.Pre -con B.Post \neq \emptyset$

Operation

- ➔ • Let the Operation O have an implementation sequence S = S_1 .. S_n where S.$State_0$ = and S.$Promised_0$ = ∅
- ➔ The interface for O is propagated as follows:

  O.Pre = S.Pre - { P | P refers to local variables }
  O.Post = S.Post - { P | P refers to local variables }
  O.Obl = S.Obl - { P | P refers to local variables }

- ➔ The state of the sequence S is amended as follows:

  S.PreCeil = S.Pre - O.Pre
  S.OblFloor = S.Obl - O.Obl

Completeness & Correctness

- ➔ An implementation I = sequence S = S_1 .. S_n for an operation O is complete if and only if
  - ↳ Every precondition P has either been satisfied or is in the interface of O. That is, all precondition ceilings (recursively) in S are empty
  - ↳ Every obligation O has either been satisfied or is in the interface of O. That is, all obligation floors (recursively) in S are empty.
  - ↳ There are no iteration errors (that is, I.Pre -con B.Post = ∅)
- ➔ A propagated interface PI for operation O is correct with respect to the specified interface SI for operation O if and only if
  - ↳ the implementation I for operation O is complete
  - ↳ the interfaces PI and SI are identical
    - ➢ PI.Pre = SI.Pre
    - ➢ PI.Post = SI.Post
    - ➢ PI.Obl = SI.Obl
  - ↳ Note: Redefinition of the propagated interface may be needed to cast it in terms of the specified interface.

Conclusions

- ➔ Have the building blocks for the synthesizing the interfaces for complex
- ➔ language statements
- ➔ Mechanisms are in place for extension to handling exceptions
- ➔ Working on the semantics of assignment: partly automatic, partly interactive
- ➔ Punt on expressions: interact with the programmer encapsulation in functions is a way out
- ➔ Working on the specification logic (SL) and how consistency determination can be strengthened and made efficient.

Error Handling

Error Handling

- 50 - 70% of code in a large system is error handling code
- Error handling code is a weak link
  - no theory, little methodolgy
  - the errors are not well understood
  - the error handling code is not well-tested
- 20% of interface errors
  - Based on MRs (modification reports) from the third release of UNIX RTR.
    - An Empirical Study of Software Interface Faults, New Directions In Computing, IEEE, Trondheim, Norway, August 1985.
    - An Empirical Study of Software Interface Faults — An Update, (July 1986) to be presented at HICSS-20 Software Track - Code Analysis and Maintainability Session, January 1987, Kona, Hawaii.
  - 68.6% of all the MRs represented interface errors

Extensions to Hoare Specifications

- Hoare specifications provide the following paradigm for describing programs:
  - Preconditions { Program } Postconditions
- The Inscape paradigm of specification captures the notions of exceptions and obligations with the following single entrance, multi-exit specification:

  Preconditions { Program }
  Postconditions, Obligations
  . . .
  Postconditions, Obligations
- Instress distinguishes three kinds of preconditions: *assumed*, *validated*, and *dependent* preconditions.

Exception Result Specifications

- Explicit specification of the meaning
  - of partial results
  - of side-effects
- Explicit specification of minimal handling requirements
  - satisfying obligations
  - undoing undesired side-effects
- Possible specification of useful pragmatic information
  - about the severity of the exception
  - about possible recovery operations or techniques

Kinds of Assumptions

- Hoare states in "Programs are Predicates" (in Mathematical Logic and Programming Languages, Prentice-Hall, 1985):
  - *If the assumptions are falsified, the product may break, and its subsequent (but not its previous) behaviour may be wholly arbitrary. Even if it seems to work for a while, it is completely worthless, unreliable, and even dangerous.*
- Instress distinguishes three kinds of preconditions to indicate the different kinds of effects that these preconditions may have.
  - *Assumed* preconditions are those which are assumed to be true. Their falsification may indeed produce arbitrary behavior.
  - *Validated* preconditions are those which are tested for. Their falsification results in predictable results.
  - *Dependent* preconditions are those whose truth is dependent upon system circumstances. While the truth or falseness of the these conditions is unpredictable, the behavior of the program itself is predictable.

Assumed : ValidFilePtr (FP)
Validated : FileOpen (FP)
LegalRecordNr (R)
RecordExists (R)
Dependent : RecordReadable (R)
RecordConsistent (R)

ReadRecord (FP, R, &L, &Bufptr)

Postconditions : ValidFilePtr (FP)
FileOpen (FP)
LegalRecordNr (R)
RecordExists (R)
Was (RecordReadable (R))
Was (RecordConsistent (R))
Allocated (*Bufptr)
0 <= L <= AllocatedSize (*Bufptr)
RecordIn (*Bufptr)

Obligations : Deallocated (*Bufptr)

Exception: IllegalRecordNr (R)

Postconditions: ValidFilePtr (FP)
FileOpen (FP)
Not (LegalRecordNr (R))

Obligations: <none>

Recovery: Use a legal record number

Exception: RecordInconsistent (R)

Postconditions: ValidFilePtr (FP)
FileOpen (FP)
LegalRecordNr (R)
RecordExists (R)
Was (RecordReadable (R))
Not (RecordConsistent (R))
Allocated (*Bufptr)
0 <= L <= Allocated (*Bufptr)
RecordIn (*Bufptr)

Obligations: Deallocated (*Bufptr)

Recovery: ReconstructRecord (Bufptr)

Formalization of Exception Handling

- ➔ *precluded* -The associated precondition has been satisfied — there is no need to handle the exception.
- ➔ *pruned* - A refusal to handle the exception. The associated preconditions become *assumed* preconditions that must be satisfied.
- ➔ *reported* - The exception is propagated, possibly with some repair, to the interface.
- ➔ *recovered* - The exception is handled by retrying the operation that caused the exception. There may be some repair to increase the likelihood of success.
- ➔ *repaired* - The results of the exception are fixed, or compensated for, in some fashion — for example, fixed to match the successful results. Control flow then proceeds in the same fashion as the successful case.
- ➔ *ignored* - The results of the exception are satisfactory. The exception is treated as a successful result.

Examples of Exception Handling

- If *RecordExists(R)* is known to be true, then the handling of *RecordNonexistent(R)* is precluded.
- If it is not known to be true, the most obvious course is to report (ie, propagate) it to the caller.
- One possible way to handle the *RecordInconsistent(R)* exception is to fix the record (use the repair handling form) by means of the recovery routine and then continue.
- If the damage to the record is unimportant, then ignore the exception and treat the result as successful.

Exception:	RecordNonexistent (R)
Postconditions:	ValidFilePtr (FP) FileOpen (FP) LegalRecordNr (R) Not RecordExists (R)
Obligations:	<none>
Recovery:	Try a different record number

Exception:	I/O-Error (R)
Postconditions:	ValidFilePtr (FP) FileOpen (FP) LegalRecordNr (R) RecordExists (R) Not (RecordReadable (R)) or Not (RecordConsistent (R)) Allocated (*Bufptr) 0 <= L <= Allocated (*Bufptr) RecordIn (*Bufptr)
Obligations:	Deallocated (*Bufptr)
Recovery:	ReconstructRecord (Bufptr)

Summary

- Forward Error Recovery
- Instress
  - explicit specification of each exception
  - explicit specification of minimal handling
  - pragmatic information about severity and recovery
  - method for determining exceptions
- Inform/Infuse
  - formalization of exception handling
  - enforced handling of exceptions
  - knowledgeable about relations between preconditions and exceptions
  - automatic construction of coherent interface (with exceptions) from the implementation