# Lecture 13
# Abstraction in Z

**David Garlan**

**Carnegie Mellon University**

**Models of Software Systems**

**15-671**

**Fall 1994**

# This Lecture

- Establishing properties about a model
  - > Properties of the state space
    - » e.g., initial state space is not empty
  - > Properties of operations
    - » values of unspecified variables
    - » preconditions of operations
- Relating models: refinement
  - > State
    - » Abstraction relations
    - » Initial states property
  - > Operations
    - » Applicability
    - » Correctness

# You Should Know

- How to state the Initialization Theorem.
- How to apply the 1-point rule.
- How to write down the formula for calculating the pre-condition of an operation.
- How to write down an abstraction relation as a schema.
- What refinement means
  - whether one relation refines another
  - what you need to prove for schemas

# Informal versus Formal Reasoning

- **Reasoning is used to increase our confidence that we have defined an appropriate model.**
  - Does my model have the desired properties?
- **Thus far (in our treatment of Z) our reasoning process has been largely informal.**
  - Not clear what we can assume when we prove something.
  - Not clear what is worth proving.
- **We will now look at a (somewhat) more formal approach.**

# Formal Proofs using Z

**The general form of a proof in any mathematical system is:**

**Hypotheses |- Conclusions**

**In Z the Hypotheses typically consist of:**

- **Declarations -- which introduce variables**
- **Predicates - which constrain the values of of those variables**

**The Conclusion is a *Predicate* that must be shown to follow as a consequence of the hypotheses.**

# Schemas as Hypotheses

Since a schema introduces variables and predicates, we may use them in our hypothesis:

$$\text{BirthdayBook} \vdash (\text{dom birthday}) \setminus \text{known} = \varnothing$$

$$\text{AddBirthday} \vdash \text{known}' = \text{known} \cup \{\text{name?}\}$$

Global (axiomatic) declarations are implicitly included:

$$\left|\ \text{maxsize}: \mathbb{N}\right.$$

$$\text{AddBirthday}; \#\text{known} < \text{maxsize}$$
$$\vdash \#\text{known}' \leq \text{maxsize}$$

# Schemas as Hypotheses (cont)

**So, a theorem of the form:**

**BirthdayBook |- (dom birthday) \ known = $\varnothing$**

**actually means:**

**known: $\mathbb{P}$ NAME; birthday: NAME $\rightarrow$ DATE**

**| known = *dom* birthday**

**|-**

**(dom birthday) \ known = $\varnothing$**

**To prove this we would argue as follows:**

$$
\begin{array}{lll}
\multicolumn{2}{l}{(\mathrm{dom}\ birthday) \setminus known} & \\
\quad = & known \setminus known & [\text{hypothesis}] \\
\quad = & \emptyset & [\text{property of } \setminus\ ]
\end{array}
$$

# Schemas as Conclusions

**Formally, a conclusion must be a predicate.**

**We can use a schema in the place of a conclusion**

- **The predicate of the schema becomes the conclusion**
- **Provided that the variables are all within scope**

**<u>Examples</u>**

**InitTN $\vdash$ EfficientTN**

**name? $\notin$ known; (AddBirthday $;$ DeleteBirthday)**

**$\vdash$ $\Xi$ BirthdayBook**

# Example

$$
\begin{array}{|l}
\hline
\textbf{NonEmptyBB} \\
\text{BirthdayBook} \\
\hline
\text{known} \neq \emptyset \\
\hline
\end{array}
$$

**AddBirthday |- NonEmptyBB'** $\equiv$

known, known': $\mathbb{P}$ NAME;
birthday, birthday': NAME $\rightarrow$ DATE;
name?: NAME; date?: DATE
| known = *dom* birthday $\wedge$ known' = *dom* birthday'
name? $\notin$ known
birthday' = birthday $\cup$ { (name?, date?) }
|-
known' $\neq \emptyset$

# The Initialization Theorem

**Given a description of the initial state of a system, we would like to guarantee that at least one such state exists.**

**Suppose <u>InitState</u> is a schema describing the initial state of a system whose state is defined by schema <u>State</u>**

**Formally, we need to prove:**

$$\vdash \exists \; State' \bullet InitState$$

(note in this case we have no hypotheses)

# Example

**InitBirthdayBook**

**BirthdayBook'**

---

**known' = $\emptyset$**

**birthday' = $\emptyset$**

**We mustshow**

$\vdash$ $\exists$ **BirthdayBook' • InitBirthBook**

# Example (cont)

$\vdash \exists$ BirthdayBook' • InitBirthBook

$\equiv$

$\vdash \exists$ known': $\mathbb{P}$ NAME; birthday': NAME $\rightarrow$ DATE

| known' = *dom* birthday'

• known' = $\varnothing \wedge$ birthday' = $\varnothing$

**Informally we might argue this as follows:**

– Take known' and birthday' to be $\varnothing$.

– For these choices the state invariant known' = *dom* birthday' is satisified.

– This demonstates the existence of an appropriate known' and birthday'.

# The One-Point Rule

In a more formal proof we might prove the theorem by simplifying the above to: |- true.

To do this we would have to eliminate the $\exists$ .

A key tool for doing this is the <u>One Point Rule</u>

$$\frac{\exists\, x: S \bullet (p \wedge x = t)}{p[t/x]} \qquad \text{(x is not free in t)}$$

i.e., if you know the value of a variable, you can substitute the value for the variable and eliminate it from the existential quantification.

# Two Useful Identities

1. $\forall x: T \mid P(x) \bullet Q(x)$

$\equiv$

$\forall x: T \bullet P(x) \Rightarrow Q(x)$

2. $\exists x: T \mid P(x) \bullet Q(x)$

$\equiv$

$\exists x: T \bullet P(x) \land Q(x)$

# Example

$\exists$ **known': $\mathbb{P}$ NAME; birthday': NAME $\rightarrow$ DATE**

  $|$ **known' = *dom* birthday'**

  • **known' = $\varnothing$ $\wedge$ birthday' = $\varnothing$**

$\equiv$ [logic]

$\exists$ **known': $\mathbb{P}$ NAME; birthday': NAME $\rightarrow$ DATE**

  • **known' = *dom* birthday' $\wedge$ known' = $\varnothing$ $\wedge$ birthday' = $\varnothing$**

$\equiv$ [1 point rule -- known' = $\varnothing$]

$\exists$ **birthday': NAME $\rightarrow$ DATE**

  • **$\varnothing$ = *dom* birthday' $\wedge$ birthday' = $\varnothing$**

$\equiv$ [1 point rule -- birthday' = $\varnothing$]

**($\varnothing$ = dom $\varnothing$)**

$\equiv$ [property of dom]

**true**

# Precondition Calculation

- When an operation is defined it is important to know in which states it can be applied.
- Other cases can then be identified and error messages supplied.
- The "precondition" tells us the starting states from which the operation could be successfully applied
- Informally the precondition is obtained by looking at the part of the operation's predicate that contains only inputs and "before" variables.

# Example

**AddBirthday**

$\Delta$ **BirthdayBook**

**name?: NAME**

**date?: DATE**

---

**name?** $\notin$ **known**

**birthday' = birthday** $\cup$ **{ name? ¢ date? }**

**has precondition: name?** $\notin$ **known**

# Preconditions (cont)

**More formally we would like a rule for determining the precondition of a schema.**

**We do this by "hiding" the primed and output variables.**

**If Op is an operation on state State with outputs Outs!, then we define pre Op (the precondition of Op) to be the schema:**

$$\exists\ State'; Out! \bullet Op$$

**Formally this is the schema**

```
┌─pre Op ──────────────────────────────
│ State
│ Ins?
├──────────────────
│ ∃ State'; Out!  • op-predicate
└──────────────────────────────────────────
```

# Example

**pre AddBirthday**

BirthdayBook
name?: NAME
date?: DATE

---

$\exists$ BirthdayBook' $\bullet$

name? $\notin$ known $\wedge$

birthday' = birthday $\cup$ { (name?, date?) }

# Example (cont)

$\exists$ **known': $\mathbb{P}$ NAME; birthday': NAME $\rightarrow$ DATE**

- **known' = *dom* birthday' $\wedge$ name? $\notin$ known $\wedge$ birthday' = birthday $\cup$ {(name?, date?)}**

$\equiv$ [1 point rule -- known' ]

$\exists$ **birthday': NAME $\rightarrow$ DATE**

- **name? $\notin$ known $\wedge$ birthday' = birthday $\cup$ {(name?, date?)}**

$\equiv$ [1 point rule -- birthday' ]

**name? $\notin$ known**

# General Reasoning

**Other useful things can be proved, but these typically depend on the application.**

**<u>Example</u>**

**name? $\notin$ known; AddBirthday ; DeleteBirthday**

**• $\Xi$ BirthdayBook**

(question: why is the first hypothesis needed?)

# Implementation/Refinement

- Qn: Given two state machines, what does it mean to say one implements another?
- Ans: The second machine should do at least what the first machine does.
- Note: It can do more, and if the abstract machine leaves some options open, then it can chose one of those options.

# Methods of Refinement in Z

- **Basic problem: How get from a specification to a program?**
- **No single method**
- **Usually development is divided into two parts**

  **<u>Part1</u>:**

  **repeat two step process of**

  – data refinement

  – operation refinement

  **until "sufficiently close" to a programming language**

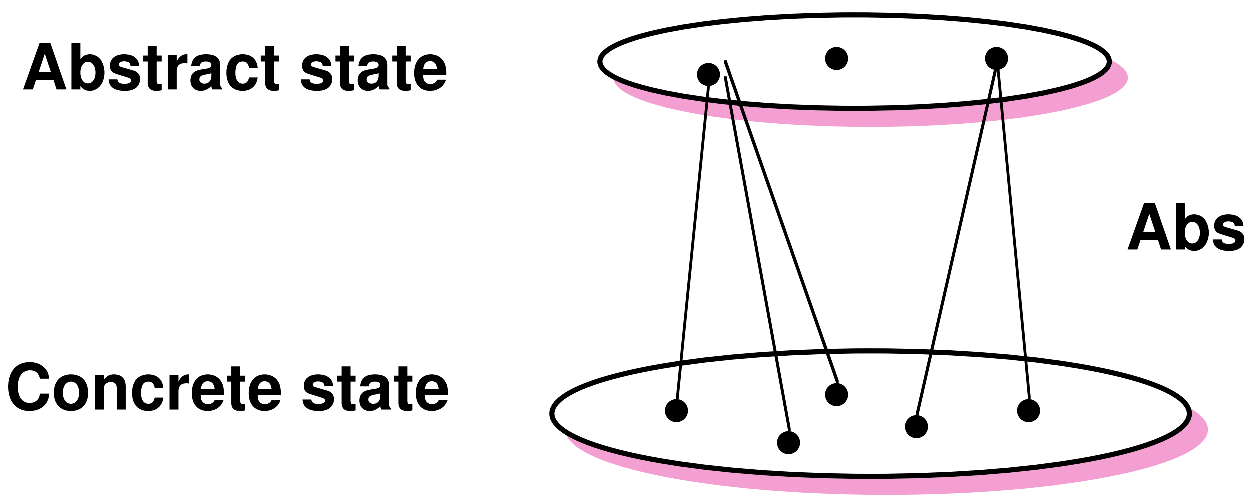
  **<u>Part 2</u>:**

  **"associate" code with the specification**

# Relating Two State Machines

In general, two machines may have different state spaces :

Our first task is therefore to produce an abstraction relation from the "concrete" machine to the "abstract" machine (Abs).



Then for each state transition, T, establish the correspondence.

T-abs (Abs(state-con)) = Abs(T-con(state-con)

$$\begin{tikzcd} \text{state-abs} \arrow[r, "\text{T-abs}"] & \text{state-abs}' \\ \text{state-con} \arrow[u, "\text{Abs}"] \arrow[r, Rightarrow, "\text{T-con}"'] & \text{state-con}' \arrow[u, Rightarrow, "\text{Abs}"'] \end{tikzcd}$$

# Observations

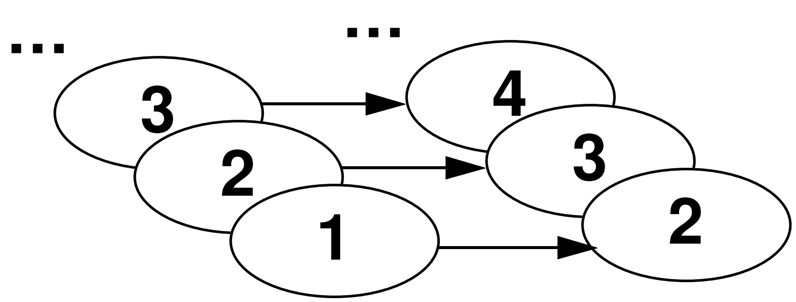
- **Abs usually associates many concrete states with each abstract state.**

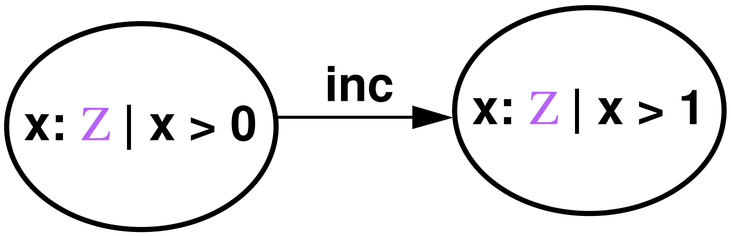
  (eg., set represented by a sequence)
- **Abs is usually a function: each concrete state has at most one abstract state.**

  (If not, what does this say about the abstract state model?)
- **Abs is sometimes called a "retrieve" function.**
- **Must also account for initial states.**
- **Concrete operations**
  - must be defined for all the corresponding states for which the abstract operation is defined (applicability)
  - must produce appropriate values (correctness)

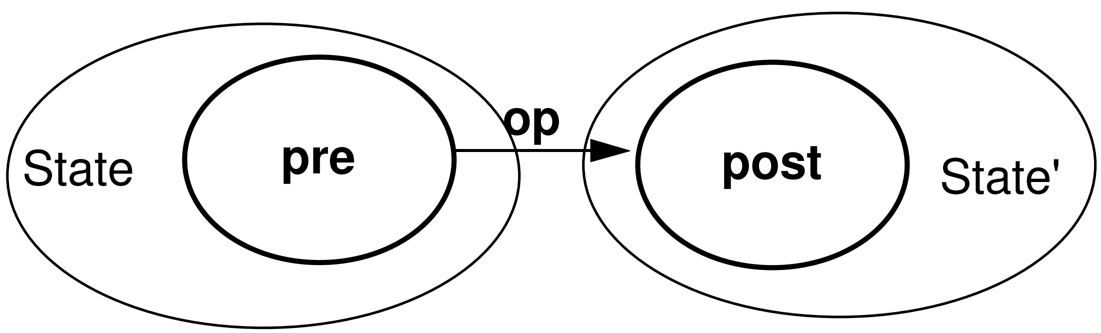
# Recall

- For state machines with large state spaces diagramatic techniques are inadequate.
- Hence, we often describe a *collection* of states by indicating some property that they all have.
- Many transitions on many states can be collapsed into a single (relational) description in this way.
- Example:



- In Z we first describe the global domain of discourse as a set of states (defined by a schema) – the state space.
- Then we define the operations in terms of the transition relationships between subsets of these states.
- For each operation:
  - The pre-condition is a predicate that determines the legal starting states.
  - The post-condition is a predicate that determines the resulting states.



# Data Refinement in Z

**BirthdayBook**

known: $\mathbb{P}$ NAME

birthday: NAME $\rightarrow$ DATE

---

known = *dom* birthday

**BirthdayBook1**

names: $\mathbb{N}_1 \rightarrow$ NAME

dates: $\mathbb{N}_1 \rightarrow$ DATE

size: $\mathbb{N}$

array[1.. ] of ...

---

$\forall$ i, j: 1 .. size • i $\neq$ j $\Rightarrow$ names(i) $\neq$ names(j)

# Abstraction Function

**Abs**

BirthdayBook

BirthdayBook1

---

known = {i: 1.. size • names(i)}

$\forall$ i, j: 1 .. size • birthday(names(i)) = dates(i))

# The Initial States Theorem

- **Must show that every initial concrete state corresponds to an initial abstract state.**
- **This is the theorem:**

$$\text{InitConcrete} \wedge \text{Abs}' \Rightarrow \text{InitAbstract}$$

- **Example:**

**InitBirthBook**

BirthdayBook'

---

known' = $\varnothing$

birthday' = $\varnothing$

**InitBirthBook1**

BirthdayBook1'

---

size' = 0

names' = dates' = <>

**To prove**

**InitConcrete $\wedge$ Abs' |- InitAbstract**

**we would argue:**

**known**

**= {i: 1.. size • names(i)}    [Abs']**

**= { i: 1.. 0 • names(i)}    [size = 0]**

**= $\varnothing$**

**and similarly we could show**

**birthday = $\varnothing$**

# Operation Refinement

- **When a concrete operation Op2 refines an abstract operation Op1, we write: Op1 Ê Op2**
- **To show that Op2 is a refinement of Op2:**

  **1. <u>Applicability</u>**

  **Whenever Op1 is applicable so is Op2.**

  **2. <u>Correctness</u>**

  **When Op1 is applicable but Op2 is applied, then the result is consistent with Op1.**

# Relations

- For the case of relations R and S, R Ê S means:
  1. dom R ≤ dom S [applicability]
  2. ((dom R) ` S) ≤ R [correctness]
- Example

  R = {Todd ¢ blue, Todd ¢ red, Jim ¢ green, Jim ¢ red}

  S = {Todd ¢ blue, Jim ¢ green, Hillary ¢ red}

R Ê S because

1. {Todd, Jim} ≤ {Todd, Jim, Hillary}
2. {Todd ¢ blue, Jim ¢ green} ≤ R

# Schemas

- **For the case of schemas, R and S, that represent operations on the same state, R Ê S means:**

  **1. pre R ‹ pre S** [applicability]

  **2. (pre R $\wedge$ S) ‹ R** [correctness]

# General Case

- **In the general case, the operations apply to different states (concrete and abstract).**
- **In this case Op-abs Ê Op-conc means:**

  **1. pre Op-abs $\wedge$ Abs ‹ pre Op-conc** [applicability]

  **2. (pre Op-abs $\wedge$ Abs $\wedge$ Op-conc)**

  **‹ $\exists$ State' • Op-abs $\wedge$ Abs'** [correctness]

- **Sometimes 2 is written**

  **pre Op-abs $\wedge$ Abs $\wedge$ Op-conc $\wedge$ Abs'**

  **‹ Op-abs**

# Example

**AddBirthday**

$\Delta$ BirthdayBook

name?: NAME

date?: DATE

---

name? $\notin$ known

birthday' = birthday $\cup$ { name? ¢ date? }

**AddBirthday1**

Δ BirthdayBook1

name?: NAME

date?: DATE

---

∀ i: 1 .. size • names? ≠ names(i)

size + 1 = size'

names' = names Õ {size' ¢ name?}

dates' = dates Õ {size ¢ date?}

# Applicability (informal)

- Must show that whenever AddBirthday can be legally applied to an abstract state, AddBirthday1 can be legally applied to any corresponding concrete state.
- The precondition of AddBirthday1 is

  $\forall$ i: 1 .. size • name? $\neq$ names(i)

- Abs tells us that this precondition is equivalent to saying

  name? $\notin$ known

- But this is precisely the precondition of AddBirthday

# Correctness (informal)

- **We must show that the concrete operation is consistent with the postcondition of AddBirthday:**

  **birthday' = birthday ∪ { name? ¢ date? }**

- **The proof goes as follows:**

  **1. We show that the function represented by the two arrays after the operation has the same domain as birthday’ (as viewed through Abs).**

  **2. Then we show that the ranges are the same.**