CS63Z
Formal Methods in Software Engineering

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Practical Matters

Place and Time:

- Room IFLT (for now. . .)
- Monday, Tuesday, Thursday 6.00 p.m.–9.00 p.m.
- Later, we will get some lab time as well

Course web page

- `http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html`
- Lecture notes (updated continously)
- Some links to external resources

What are "Formal Methods"?

Approach to develop high-quality systems

Develop a clear, formal, unambiguous, abstract model

- Ensure model actually reflects requirements

Use formal reasoning to ensure the model has required properties

- Safety properties: State X never reached
- Lifeness properties: State X eventually reached
- Fairness properties: In infinite time, X is reached infinitely often
- Partial correctness: If system terminates on input X, output is $f(X)$
- ...

Implement model

- Use proof/synthesis to ensure correct translation
- Ensure abstraction does not break (i.e. integer numbers vs. machine words, unlimited memory, exact floating point arithmetic)

Course Overview

Introduction

Mathematical Foundations

- Naive set theory
- Propositional logic
- First-order logic

Reasoning about programs

- An informal look at invariants
- Hoare-Logic
- Termination

Formal specifications

- The Z notation
- Reasoning about specifications
- Refining specifications into (Python) code

Introduction

"Formal Methods" is an umbrella term

- Formal specification (Z, B)
- (Software) model checking
- Program verification
- Programming language semantics
- Software/System modelling (Finite state machines, Petri-Nets . . .)
- . . .

Common properties:

- Logic-based (or translatable into logic)
- Allow assurance of (some) software properties

Aim:

- Unambiguous specifications
- Proven correct programs
- Happy users

. . . we are far from that aim ;-)

Formal Methods for (Digital) Hardware Design

Formal methods are standard for modern hardware design!

- Model checking
- Simulation
- Theorem proving
- Equivalence checking
- Synthesis of designs
- . . .

Large industry:

- DATE conference (industrial)
- Large companies: Cadence systems (Verilog), Mentor Graphics
- Startups: Safelogic A.B., Prover Technologies
- . . . much CAD (Computer Aided Design) is FM for Hardware

Large and active research community

Result: Few bugs in delivered hardware

Hardware Quality is (relatively) high

Few serious hardware bugs

Example: Pentium FDIV bug caused major outrage

- Relatively rarely encountered
- Minor deviation from correct results
- Intel replaced processor for free

Maybe 10 known bugs in Pentium family

- Most rarely encountered
- Most have easy workarounds

Hardware Quality is (relatively) high

Few serious hardware bugs

Example: Pentium FDIV bug caused major outrage

- Relatively rarely encountered
- Minor deviation from correct results
- Intel replaced processor for free

Maybe 10 known bugs in Pentium family

- Most rarely encountered
- Most have easy workarounds

Comparison: Nobody even bothers reporting most Windows bugs

- Security issues cause weekly outbreaks of worms/viruses
- "Blue Screen of Death" has become part of our language
- Other developers are not much better (My computer runs MaxOS-X 10.3.9)

Obvious solutions: Adopt suitable formal methods for Software Engineering

State of the Art in Software Engineering

Formal methods rarely used

- Mostly in saftety-critical applications
- Example: Microsoft Driver Verification Program
- Example: NASA space probe control software

FM industry just starting:

- Praxis (Z)
- Escher Technologies (Perfect Developer)

But: Long research tradition

- Floyd/Hoare style logics start in 1960s
- Z development starts in 70s
- Useful tool support starts in 90s

Why are Formal Methods rarely used in Software Development?

Why are Formal Methods rarely used in Software Development?

Basic software development is cheap

- Minimal investment: One PC, one copy of Linux/gcc
- Production cost for a new (corrected?) version is negligible (a few minutes of compilation)
- Distribution cost is cheap (ship a CD/free download)

Customers are more tolerant of bad software

- Few people expect correct software: "I had to stay late, Word ate my document" is frequently used excuse
- Software is intangible

Reasoning about software is hard

- Formal methods add massive overhead
- Training for formal methods is expensive/time consuming

Why are Formal Methods rarly used in Software Development?

Basic software development is cheap

- Minimal investment: One PC, one copy of Linux/gcc
- Production cost for a new (fixed?) version is negligible (a few minutes of compilation)
- Distribution cost is cheap (ship a CD/free download)

Customers are more tolerant of bad software

- Few people expect correct software: "I had to stay late, Word ate my document" is frequently used excuse
- Software is intangible

Reasoning about software is hard

- Formal methods add massive overhead
- Training for formal methods is expensive/time consuming

$\Longrightarrow$ Standard practice to achive Software Quality is Code-Test-Debug cycle. . .

Why are Formal Methods rarly used in Software Development?

Basic software development is cheap

- Minimal investment: One PC, one copy of Linux/gcc
- Production cost for a new (fixed?) version is negligible (a few minutes of compilation)
- Distribution cost is cheap (ship a CD/free download)

Customers are more tolerant of bad software

- Few people expect correct software: "I had to stay late, Word ate my document" is frequently used excuse
- Software is intangible

Reasoning about software is hard

- Formal methods add massive overhead
- Training for formal methods is expensive/time consuming

$\Longrightarrow$ Standard practice to achive Software Quality is Code-Test-Debug cycle. . . often with the customer in the loop

Software complexity vs. Hardware complexity

Hardware:

- Millions of transistors ($\approx$ 50 million for P4)
- Most of the structure is highly regular:
  - Caches
  - Register files
- ALU well understood
- Control logic complex, but small
- A processor is a finite state machine

Software:

- Often millions of lines of code ($\approx$ 50 million for Windows NT)
- Highly irregular (no two functions are the same)
- Often conceptually infinitely many states

$\Longrightarrow$ Reasoning about software is harder

$\Longrightarrow$ But: Debugging software is easier

Software Engineering Processes

Traditional Processes (e.g. Waterfall):

1. Requirements Analysis
2. Specification
3. Design and Architecture
4. Coding
5. Testing
6. Documentation
7. Maintenance

Agile Development:

- Prototyping (Covers 1-4, 6)
- Testing
- Refactoring (Covers 3,4, 7)

FM and Traditional Process Steps (1)

Requirements analysis may fail to identify all requiremtents

- Formal Methods cannot help much here...

Specifications may be ambiguous and/or not reflect requirements

- Formal specifications can be unambiguous
- Formal specifications can be checked for consistency
- Formal specifications can be used to deduce (unexpected?) properties

Bad design and Architecture

- Mostly a matter of taste and experience. . .

FM and Traditional Process Steps (2)

Coding may fail to implement the specifications

- Program verification (large subfield of FM) can help
- Program synthesis/transformation may generate code directly from specifications

Testing cannot (well, rarely) estabish absence of bugs!

- Only existance of bugs can be detected
- Again, verification can guarantee the absence of bugs (or at least some classes of bugs)

Software maintenance often involves large-scale systematic changes

- Formal methods can assure equivalence of changed code
- Formal methods can automate changes

Example: Ambiguous Requirements Analysis/Specification

Spacecraft developed by two teams of engineers at NASA JPL and Lockheed Martin

- NASA uses metric unit system
- Lockheed Martin uses imperial unit system

Nobody noticed the discrepancy!

Result: Mars Climate Orbiter crashed into Mars on September 23rd, 1999
$\Longrightarrow$ 125 million US$ lost
$\Longrightarrow$ 10 month journey to Mars wasted
$\Longrightarrow$ Mission partially compromised (back-up space probes to the rescue...)

Example: Ambiguous Requirements Analysis/Specification

Spacecraft developed by two teams of engineers at NASA JPL and Lockheed Martin

- NASA uses metric unit system
- Lockheed Martin uses imperial unit system

Nobody noticed the discrepancy!

Result: Mars Climate Orbiter crashed into Mars September 23rd, 1999
$\Longrightarrow$ 125 million US$ lost
$\Longrightarrow$ 10 month journey to Mars wasted
$\Longrightarrow$ Mission partially compromised (back-up space probes to the rescue...)
$\Longrightarrow$ Possibly 250.000 dead Martians

Example: Maintainance/Porting Error

ESA had a successful space program

- Ariane IV one of the most successful commercial satellite launch platforms

Ariane V should be bigger, better, faster version

- To safe development costs, some code from Ariane IV was reused
- Development took 10 years and EUR 8 billion anyways

In the code, 64 bit floating point representation of speed was cast into 16 bit integer value

- Ok for Ariane IV
- Ariane V's higher speed causes overflow condition
  - Correctly trapped by software
  - . . . but no trap handler (efficiency reasons, "it never happens anyways")
  - . . . hence software crash

Result: First Ariane V rocket out of control, destroyed by safety mechanism (June 4th, 1996)

Example: Design/Coding Error

Therac-25 was a combined electron/X-ray medical radiation device (1985-1988)

- Able to produce high energy (5-25 MeV) electron beams, using a beam spreader to control intensity
- Also able to produce 25MeV X-rays (by placing a target and a beam spreader into a much amplified electron beam)
- Used in radiation therapy for cancer patients

Software was expected to assure that electron beam intensity, target, and beam spreader can only work in safe conditions

However, race conditions in the code allowed unsafe states to be reached:

- High-energy, high intensity electron beam without target/beam spreader
- Race condition never detected in testing!

Result:

- Several people died of radiation burns, several more injured
- Lawsuits settled out of court.

Testing Breakdown: Array initialization

Example from my own code: Polymorphic, dynamic arrays in C

- Support storage of (long) integers or pointers
- Grow dynamically as needed, new elements initialized to 0/NULL

Base type:

```
/* Trick the stupid type concept for polymorphic indices (hashes,
   trees) with int/pointer type. */

typedef union int_or_p
{
   long i_val;
   void *p_val;
}IntOrP;
```

Testing Breakdown: Array initialization (contd.)

Array type:

```
typedef struct pdarraycell
{
   bool   integer;  /* if true, elements are (long) ints */
   long   size;     /* How large so far?    */
   long   grow;     /* How is growing handled? */
   IntOrP *array;   /* The data itself          */
}PDArrayCell, *PDArray_p;
```

Testing Breakdown: Array initialization (contd.)

Initialization code for growing array:

```
...
for(i=old_size; i<array->size; i++)
{
   if(array->integer)
   {
      array->array[i].p_val = NULL;
   }
   else
   {
      array->array[i].i_val = 0;
   }
}
...
```

Testing Breakdown: Array initialization (contd.)

Initialization code for growing array:

```
...
for(i=old_size; i<array->size; i++)
{
   if(array->integer)
   {
      array->array[i].p_val = NULL;
   }
   else
   {
      array->array[i].i_val = 0;
   }
}
...
```

Who finds the error?

Testing Breakdown: Array initialization (contd.)

Effect:

- Integer elements get initialized as NULL pointers
- Pointer elements get initialized as 0 long integers

but. . .

Testing Breakdown: Array initialization (contd.)

Effect:

- Integer elements get initialized as NULL pointers
- Pointer elements get initialized as 0 long integers

but. . .

- On common platforms, NULL and (long)0 have the same machine representation
  - 32-bit UNIX dialects: Both are 32 bits of value 0
  - 64-bit UNIX dialects: Both 64 are bits of value 0
  - 32-bit Windows (*barf*): Both 32 are bits of value 0

  $\Longrightarrow$ Programmer fault never manifested as software failure!

Bug undetected between 1998 and 2005 in heavily excercised code!

Testing Breakdown: Array initialization (contd.)

Effect:

- Integer elements get initialized as NULL pointers
- Pointer elements get initialized as 0 long integers

but. . .

- On common platforms, NULL and (long)0 have the same machine representation
  - 32-bit UNIX dialects: Both are 32 bits of value 0
  - 64-bit UNIX dialects: Both 64 are bits of value 0
  - 32-bit Windows (*barf*): Both 32 are bits of value 0

  $\Longrightarrow$ Programmer fault never manifested as software failure!

Bug undetected between 1998 and 2005 in heavily excercised code!

- In 2005, code was compiled on 64-bit Windows
  - long is 32 bit
  - Pointers are 64 bit
- Programmer wears brown paper bag over his head ;-)

Excercises

Answer these questions:

Why is comprehensive testing usually impossible?

Name three difficulties with formal specifications!

List 5 examples of software where formal methods make sense and 5 where they are less useful!

Preview for Wednesday

Some set theory and discrete math

- Sets, set properties
- Relations and functions
- If time permits: Propositional logic

CS63Z
Formal Methods in Software Engineering

Mathematical Foundations

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Practical Matters

Place and Time:

- Room IFLT (for now. . .)
- Monday, Tuesdays, Thursdays, 6.00 p.m.–9.00 p.m.
- Later, we will get some lab time as well

Course web page

- `http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html`
- Lecture notes (updated continously)
- Some links to external resources

Naive set theory

Foundation of many formal methods (and much of modern mathematics)

Definition: A set is a well-defined collection of objects considered as a whole. The objects of a set are called elements. Sets are conventionally denoted with capital letters, A, B, C, etc. Two sets A and B are said to be equal, written $A = B$, if they have the same elements.

We write

- $x \in A$ if x is an element of A
- $x \notin A$ if x is not an element of A

Definition:

- A is a subset of B ($A \subseteq B$), if for all $x \in A : x \in B$
- A is a proper subset of B ($A \subset B$), if $A \subseteq B$ and $A \neq B$

Writing Sets

Informal set descriptions:

- Let A be the set of all even numbers
- Let B be the set of all students passing this class
- Can be dangerous (ambiguity/contradictions)

Sets can be written as explicit enumerations:

- $A = \{1, 2, 3, 4\}$
- $B = \{4, 4, 4, 4, 3, 3, 3, 2, 2, 1\}$ (Note: $A = B$!)
- $C = \{1, 3, 5, 7, \ldots, 211\}$

Sets described by comprehensions:

- $A = \{x \mid x \text{ is a blue cat}\}$
- $B = \{x \in \mathbb{N} \mid x \text{ is even}\}$
- $C = \{2x \mid x \in \mathbb{N}\}$ (Note: $B = C$)

Some Important Sets

The empty set:

- $\varnothing = \{\}$ contains no elements

Sets of numbers:

- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$: Natural numbers
- $\mathbb{N}_1 = \{1, 2, 3, \ldots\}$: Positive integer (natural) numbers
- $\mathbb{Z} = \{0, 1, -1, 2, -2, 3, -3 \ldots\}$: Integers numbers
- $\mathbb{R}$: Real numbers

Finite and Infinite Sets

Sets can be finite or infinite

- Examples:

Finite and Infinite Sets

Sets can be finite or infinite

- Examples: $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ are all infinite
- $A = \{x \mid x \text{ is a blue cat}\}$ is (probably) finite
- $\varnothing$ is finite

Finite and Infinite Sets

Sets can be finite or infinite

- Examples: $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ are all infinite
- $A = \{x \mid x \text{ is a blue cat}\}$ is (probably) finite
- $\varnothing$ is finite

Definition:

- $\mathbb{F}$ is the (infinite ;-) set of finite sets
- $\mathbb{F}_1$ is the set of non-empty finite sets

Definition: If $A \in \mathbb{F}$, then the cardinality of A is the number of elements in A

- Traditional: $\mid A \mid$ is the cardinality of A
- Z notation: $\#A$ is the cardinality of A

Examples:

- $\#\varnothing = ?$

Finite and Infinite Sets

Sets can be finite or infinite

- Examples: $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ are all infinite
- $A = \{x \mid x \text{ is a blue cat}\}$ is (probably) finite
- $\varnothing$ is finite

Definition:

- $\mathbb{F}$ is the (infinite ;-) set of finite sets
- $\mathbb{F}_1$ is the set of non-empty finite sets

Definition: If $A \in \mathbb{F}$, then the cardinality of A is the number of elements in A

- Traditional: $\mid A \mid$ is the cardinality of A
- Z notation: $\#A$ is the cardinality of A

Examples:

- $\#\varnothing = 0$
- $\#\{\text{red}, \text{green}, \text{blue}\} = ?$

Finite and Infinite Sets

Sets can be finite or infinite

- Examples: $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ are all infinite
- $A = \{x \mid x \text{ is a blue cat}\}$ is (probably) finite
- $\varnothing$ is finite

Definition:

- $\mathbb{F}$ is the (infinite ;-) set of finite sets
- $\mathbb{F}_1$ is the set of non-empty finite sets

Definition: If $A \in \mathbb{F}$, then the cardinality of A is the number of elements in A

- Traditional: $\mid A \mid$ is the cardinality of A
- Z notation: $\#A$ is the cardinality of A

Examples:

- $\#\varnothing = 0$
- $\#\{\text{red}, \text{green}, \text{blue}\} = 3$
- $\#\{1, 2, 3, 3, 3, 4\} = ?$

Finite and Infinite Sets

Sets can be finite or infinite

- Examples: $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}$ are all infinite
- $A = \{x \mid x \text{ is a blue cat}\}$ is (probably) finite
- $\varnothing$ is finite

Definition:

- $\mathbb{F}$ is the (infinite ;-) set of finite sets
- $\mathbb{F}_1$ is the set of non-empty finite sets

Definition: If $A \in \mathbb{F}$, then the cardinality of A is the number of elements in A

- Traditional: $\mid A \mid$ is the cardinality of A
- Z notation: $\#A$ is the cardinality of A

Examples:

- $\#\varnothing = 0$
- $\#\{\text{red}, \text{green}, \text{blue}\} = 3$
- $\#\{1, 2, 3, 3, 3, 4\} = 4$

Set Operations

Let A, B, C be sets

$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$ is the union of A and B

- $A \cup B = B \cup A$ ($\cup$ is commutative)
- $A \cup (B \cup C) = (A \cup B) \cup C$ ($\cup$ is associative)
- $A \cup A = A$ ($\cup$ is idempotent)
- More general: $B \subseteq A$ implies $A \cup B = A$

$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$ is the intersection of A and B

- $\cap$ is associative, commutative, idempotent
- $B \subseteq A$ implies $A \cap B = B$

$A - B = \{x \mid x \in A, x \notin B\}$ is the (set-theoretic) difference of A and B

- $A - A = \varnothing$
- $A - \varnothing = A$

$C = A \uplus B$ is called the disjoint union of A and B, if $C = A \cup B$ and $A \cap B = \varnothing$

Power Sets

Definition: Let A be a set. The power set $\mathbb{P}\,A$ is the set of all subsets of A, i.e. $\mathbb{P}\,A = \{B \mid B \subseteq A\}$

- $\mathbb{P}_1\,A$ is the set of non-empty subsets of A.

Examples:

- $\mathbb{P}\,\varnothing = ?$

Power Sets

Definition: Let A be a set. The power set $\mathbb{P}\,A$ is the set of all subsets of A, i.e. $\mathbb{P}\,A = \{B \mid B \subseteq A\}$

- $\mathbb{P}_1\,A$ is the set of non-empty subsets of A.

Examples:

- $\mathbb{P}\,\varnothing = \{\varnothing\}$
- $\mathbb{P}\{1,2,3\} = ?$

Power Sets

Definition: Let A be a set. The power set $\mathbb{P}\,A$ is the set of all subsets of A, i.e. $\mathbb{P}\,A = \{B \mid B \subseteq A\}$

- $\mathbb{P}_1\,A$ is the set of non-empty subsets of A.

Examples:

- $\mathbb{P}\,\varnothing = \{\varnothing\}$
- $\mathbb{P}\{1,2,3\} = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$
- $\mathbb{P}_1\{1,2\} = ?$

Power Sets

Definition: Let A be a set. The power set $\mathbb{P}\,A$ is the set of all subsets of A, i.e. $\mathbb{P}\,A = \{B \mid B \subseteq A\}$

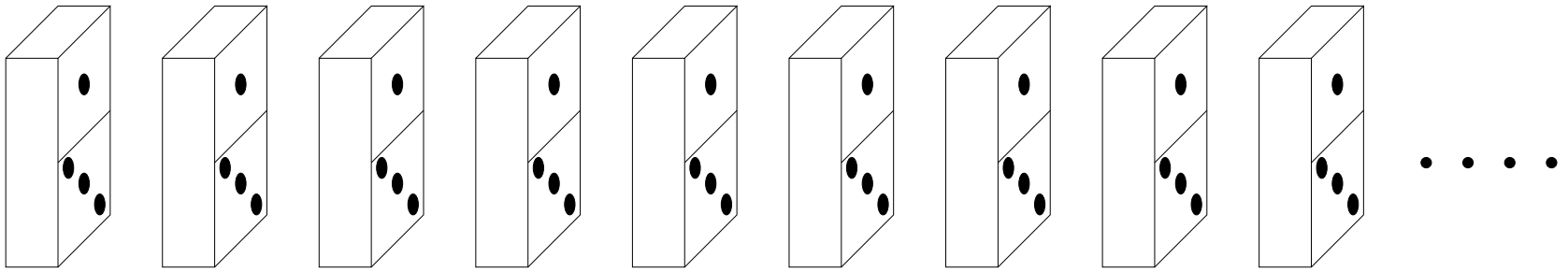
- $\mathbb{P}_1\,A$ is the set of non-empty subsets of A.

Examples:

- $\mathbb{P}\,\varnothing = \{\varnothing\}$
- $\mathbb{P}\{1,2,3\} = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$
- $\mathbb{P}_1\{1,2\} = \{\{1\}, \{2\}, \{1,2\}\}$

Warning: Enumerating large powersets may leave you tongue-tied!

Mathematical Induction



Induction is a vital proof technique for showing. . .

- properties for all natural numbers
- properties for all elements on a recursively defined set
- most generally, properties for all elements of any well-ordered set (a set with a well-founded partial ordering)

Simple case: Complete induction over the natural numbers

- To show property $P(n)$ holds for all $n \in n$, show:
  - $P(0)$
  - $P(n)$ implies $P(n+1)$
- Like dominoes, the rest follows automatically

A (Classical) Example

Claim: For all $n \in \mathbb{N} : \Sigma_{i=0}^{n} = \frac{n(n+1)}{2}$

Proof: By induction over n

Base case: $n = 0$. $\Sigma_{i=0}^{0} = 0 = \frac{0(0+1)}{2}$

Induction hypothesis: $\Sigma_{i=0}^{n} = \frac{n(n+1)}{2}$

Induction step: To show: The induction hypothesis implies $\Sigma_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$

$\Sigma_{i=0}^{n+1} i = \Sigma_{i=0}^{n} i + (n+1) =_{(IH)} \frac{n(n+1)}{2} + (n+1)$

$= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{n(n+1)+2(n+1)}{2}$

$= \frac{(n+2)(n+1)}{2} = \frac{(n+1)(n+2)}{2} = \frac{(n+1)((n+1)+1)}{2}$ □

Cardinality of Powersets

Theorem: Let $A \in \mathbb{F}$. Then $\#(\mathbb{P}\, A) = 2^{\#A}$

Proof: Induction over $\#A$.

Base case: $\#A = 0$, i.e. $A = \varnothing$. Then $\mathbb{P}\, A = \{\varnothing\}$ and $\#(\mathbb{P}\, A) = 1 = 2^0 = 2^{\#A}$.

Assumption: If $\#A \leq n$, then $\#(\mathbb{P}\, A) = 2^{\#A}$

Induction step: Let $\#A' = n + 1$.
Then we can write $A' = A \uplus \{e\}$ for some A with $\#A = n$ and some element e.
Each subset of A' either contains e, or it is a subset of A.
Hence $\mathbb{P}\, A' = \mathbb{P}\, A \cup \{B \cup \{e\} \mid B \in \mathbb{P}\, A\}$.
Now $\#\{B \cup \{e\} \mid B \in \mathbb{P}\, A\} = \#(\mathbb{P}\, A)$ and both sets are disjoint.
Hence $\#(\mathbb{P}\, A') = \#(\mathbb{P}\, A) + \#(\mathbb{P}\, A) = 2\#(\mathbb{P}\, A) =$ (by assumption) $2 \times 2^{\#(\mathbb{P}\, A)} = 2^{\#(\mathbb{P}\, A)+1} = 2^{\#(\mathbb{P}\, A')}$

□

Carthesian Products and Tuples

Definition:

Let A, B be sets. $A \times B = \{(a, b) \mid a \in A, b \in B\}$ is called the (binary) carthesian product of A and B.

More generally, let $A_1, \ldots, A_n$ be sets. Then $A_1 \times \ldots \times A_n = \{(a_1, ldots, a_n \mid a_i \in A_i for 1 \leq i \leq n\}$ is the carthesian product of $A_1, \ldots, A_n$.

The elements of binary cathesian products are called (binary) tuples

The elements of the carthesian product of n sets are called $n - tuples$.

Examples:

- $\{1, 2, 3\} \times \{a, b\} =?$

Carthesian Products and Tuples

Definition:

Let A, B be sets. $A \times B = \{(a, b) \mid a \in A, b \in B\}$ is called the (binary) carthesian product of A and B.

More generally, let $A_1, \ldots, A_n$ be sets. Then $A_1 \times \ldots \times A_n = \{(a_1, ldots, a_n \mid a_i \in A_i for 1 \leq i \leq n\}$ is the carthesian product of $A_1, \ldots, A_n$.

The elements of binary cathesian products are called (binary) tuples

The elements of the carthesian product of n sets are called $n - tuples$.

Examples:

- $\{1, 2, 3\} \times \{a, b\} = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$
- $\mathbb{N} \times \varnothing = ?$

Carthesian Products and Tuples

Definition:

Let A, B be sets. $A \times B = \{(a, b) \mid a \in A, b \in B\}$ is called the (binary) carthesian product of A and B.

More generally, let $A_1, \ldots, A_n$ be sets. Then $A_1 \times \ldots \times A_n = \{(a_1, ldots, a_n \mid a_i \in A_i for 1 \leq i \leq n\}$ is the carthesian product of $A_1, \ldots, A_n$.

The elements of binary cathesian products are called (binary) tuples

The elements of the carthesian product of n sets are called $n - tuples$.

Examples:

- $\{1, 2, 3\} \times \{a, b\} = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$
- $\mathbb{N} \times \varnothing = \varnothing$

Relations

Definition: Let A, B be sets. $R \in \mathbb{P}(A \times B)$ is called a binary relation over A and B.

- Frequent case: $A = B$ (homogenous relations)
- In Z, we also write $A \leftrightarrow B$ instead of $\mathbb{P}(A \times B)$ for the set of all relations over A and B.
- Similarly, we write $R : A \leftrightarrow B$ instead of $R \in \mathbb{P}(A \times B)$
- We often write $R(a, b)$ for $(a, b) \in R$
- Many special relations are written infix ($a > b, a = b, \ldots$)

Definition: If R is a relation, $R^{-1} = \{(b, a) \mid R(a, b)\}$ is called the inverse relation of R.

Examples:

- $>$ (over $\mathbb{N}$) is a relation over $\mathbb{N}, \mathbb{N}$. $(4, 1) \in >$ (more familar: $4 > 1$)
- $>^{-1}$ is $<$
- $\{(a, a) \mid a \in A\} \in A \leftrightarrow A$ is the equality relation on A ($=$), and $=^{-1} ==$

Relational Chains

Definition: Let A be a set and let $R : A \leftrightarrow A$ be a binary relation over A. Let $a_1, a_2, \ldots \in A$ be elements. The sequence $a_1, a_2, \ldots$ is called an R-chain, if $R(a_i, a_{i+1})$ holds for all $i \in \{1, \ldots\}$.

- Remark: Relational chains can be finite or infinite (in the finite case, assume a suitable limit to i)
- We write many predefined relation symbols in infix notation, and can do so in relational chains: $5 > 4 > 3 > 2 > 1$ is a $>$-chain

If $a_1, \ldots, a_n$ is a finite R-chain, then the length of the chain is the number of steps in it $(n - 1)$

Properties of Relations

Definition: Let $R \in A \leftrightarrow A$ be a homogenous relation over A

R is called reflexive, if $R(a, a)$ for all $a \in A$.

R is called symmetric, if $R(a, b)$ implies $R(b, a)$ for all $a, b \in A$.

R is called transitive, if $R(a, b), R(b, c)$ implies $R(a, c)$ for all $a, b, c \in A$

A relation that is reflexive, symmetric and transitive is called an equivalence relation.

Thare are two different, but related versions of antisymmetry around:

1. R is called (strictly) antisymmetric, if $R(a, b)$ implies $(b, a) \notin R$ for all $a, b \in A$.
2. R is called (weakly) antisymmetric, if for all $a, b \in A : R(a, b)$ and $R(b, a)$ implies $a = b$

R is called antireflexive, if for no $a \in A : R(a, a)$.

Partial Orderings

Definition (1): A relation $>: A \leftrightarrow A$ is called a (strict) partial ordering on A if:

1. $>$ is (strongly) antisymmetric
2. $>$ is transitiv
3. ($>$ is antireflexive) (follows from (strong) antisymmetry)

Definition (2): A relation $\geq: A \leftrightarrow A$ is called a partial ordering on A if:

1. $\geq$ is (weakly) antisymmetric
2. $\geq$ is transitiv
3. $\geq$ is reflexive

$(\geq - =)$ is called the strict part of $\geq$ and is a partial ordering in the first sense

$> \cup =$ is a partial ordering in the second sense.

Termination

Definition: A (strict) partial ordering $>$ on A is called terminating, Noetherian, well-founded, if there is no infinite $>$-chain

- We customarily say "No infinitely descending chain", implicitely assuming that $a > b$ means that a is greater than b.
- We often say that an ordering $\geq$ terminates, if its strict part does.

We sometimes also speak of the termination of arbitrary relations. R terminates, if there is no infinite R-chain

Examples

Let A be a set

The empty relation is $\varnothing : A \leftrightarrow A$.

▶

The identity relation $\mathrm{id} : A \leftrightarrow A$ is defined by $\mathrm{id} = \{(a, a) \mid a \in A\}$

▶

▶

The universal relation is defined by $\{(a, b) \mid a, b \in A\}$ (otherwise known as $A \times A$)

▶

Consider $R_1 : \mathbb{N} \leftrightarrow \mathbb{N}$ defined by $R_1 = \{(a, 2n + a) \mid a, n \in \mathbb{N}\} \cup \{(2n + a, a) \mid a, n \in \mathbb{N}\}$

▶

Examples

Let A be a set

The empty relation is $\varnothing : A \leftrightarrow A$.

- The empty relation is a strict, well-founded partial relation

The identity relation $\mathrm{id} : A \leftrightarrow A$ is defined by $\mathrm{id} = \{(a, a) \mid a \in A\}$

- id is an equivalence relation and a non-strict partial ordering.
- The strict part is the empty relation (and well-founded)

The universal relation is defined by $\{(a, b) \mid a, b \in A\}$

- The universal relation is an equivalence relation

Consider $R_1 : \mathbb{N} \leftrightarrow \mathbb{N}$ defined by $R_1 = \{(a, 2n + a) \mid a, n \in \mathbb{N}\} \cup \{(2n + a, a) \mid a, n \in \mathbb{N}\}$

- R_1 relates all even numbers and all odd numbers with each other, respectively
- R_1 is an equivalence relation
- We say: The two equivalence classes are the even and the odd numbers

Closures

Given a relation $R : A \leftrightarrow A$, we can extend it to have the desired properties:

The transitive closure of R, denoted R^+, is the smallest transitive relation with $R \subseteq R^+$

The transitive, reflexive closure, of R, denoted R^*, is $R^+ \cup \mathrm{id}$

The reflexive, symmetric and transitive closure of R, $(R \cup R^{-1})^*$ is the smallest equivalence relation containing R

Functions

Definition: Let A, B be sets and $f : A \leftrightarrow B$ a relation.

f is called a (partial) function if $f(a, b)$ and $f(a, c)$ implies $b = c$. In that case, we write $f : A \nrightarrow B$ and $f(a) = b$.

$\mathrm{dom}\, f = \{a \mid (a, b) \in f\}$ is called the domain of f

$\mathrm{ran}\, f = \{b \mid (a, b) \in f\}$ is called the range of f

If $\mathrm{dom}\, f = A$, then f is called a total function, and we write $f : A \rightarrow B$

If $\mathrm{ran}\, f = b$, then f is called surjective, and we write $f : A \twoheadrightarrow B$ or $f : A \rightarrow\!\!\!\!\!\twoheadrightarrow B$ (if f is total)

If $f(a) = c$ and $f(b) = c$ imply $a = b$, then f is called injective.

If f is both injective and surjective, it is called bijective or a one-to-one mapping.

Excercises

1.1 Show: If $A, B \in \mathbb{F}$, then $\#(A \times B) = \#A \times \#B$. Hint: Use induction.

1.2 Show: If f is an injective function, then f^{-1} is a function.

Excercises are two marks each (out of 8 for all coursework) and are due on Thursday, June 30th.

CS63Z
Formal Methods in Software Engineering

Propositional Logic

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Propositional Logic

Formalization of reasoning on simple propositions

- Goes back to ancient Greek philosophers thinking about thinking

Propositions are atomar

- "The grass is green"
- "The student is smart"

Propositions can be true or false

Deals with arguments, implications, . . . , but abstracts from the meaning of propositions

- You can analyse the soundness of an argument
- You cannot (always) establish the truth if you cannot independently verify the assumptions

Example

Consider the statement "If it rains, the grass is green"

- There are two elementary propositions: "it rains" and "the grass is green"
- There is a relationship between the two: It asserts that whenever the first is true, so is the second

In propositional logic, the propositions are atomar

The relationship is modelled by a logical operator

In this case: $(it\_rains \Rightarrow grass\_is\_green)$

Depending on which truth values we assign to the propositions, this formula can be true or false

- We are not forced to assume it_rains is true today ;-)
- We can reason about the validity of the statement for all cases
- It can be true, but does not have to be true

Example (continued)

Some statements are always true or false:

- $(it\_rains \Rightarrow it\_rains)$
- $(grass\_is\_green \wedge (\neg\, grass\_is\_green))$

Note the implicit assumption:

- Something is either true or false (*Tertium non datur*, "there is no third possibility")

The propositional logic world is black/white ;-)

Language/Syntax of Propositional Logic

Propositional logical formula are words over an alphabet containing:

- Propositions (also called propositional atoms, propositional variables. . .)
- Logical operators (also called junctors)
- Parentheses

Definition: Let $\Sigma = V \cup o \cup \{(,)\}$ be an alphabet, where V is a enumerable set of propositions and $o = \{\neg\ , \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ is a set of operators. The set of well-formed propositional formulae ($wfpf$) is defined as follows:

- Every proposition is a well-formed propositional formula ($V \subset wfpf$) (called atomic).
- If $A, B \in wfpf$, then:
  1. $(\neg\ A) \in wfpf$
  2. $(A * B) \in wfpf$ for all $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.
  3. $wfpf$ is the smallest set that fullfils 1) and 2).

If the context is clear, we just say "formula"

Examples

Assume $V = \{p_1, p_2, \ldots\}$

$A_0 = (p_1 \Leftrightarrow (\neg\ p_2))$ is a well-formed propositional formula

- $p_1, p2 \in wfpf$ by definition
- $(\neg\ p_2) \in wfpf$ by rule 1
- $(p_1 \Leftrightarrow (\neg\ p_2)) \in wfpf$ by rule 2

$A_1 = (p_1 \wedge\Rightarrow\ p_1) \notin wfpf$

- $A_1 \notin V$
- A_1 is not generated by rule 1, as the second sign is not $\neg$
- Assume A_1 is generated by rule 2. The either $p_1 \wedge$ has to be a formula, or $\Rightarrow p1$ has to be a formula.
  - $p_1 \wedge$ is not a formula, as it is not atomic and does not start with a parenthesis
  - The same holds for p_2.

$A_2 = (p_1 \wedge (p_2 \wedge (\neg\ p_3)))$

$A_3 = (p_1 \vee (p_2 \vee \neg\ p3))$

Structural Induction

We show properties of well-formed formulas by structural induction

- Base case: Claim holds for propositions
- Assumption: Claim holds for $A, B \in wfpf$
- Induction step: Claim holds for $(\neg A), (A * B)$ for $* \in o$

Example: All formulas $C \in wfpf$ are either atomic or start in '(' and ends with ')'

- Base case: $A \in V$ is atomic, hence the claim holds
- Assumption: Claim holds for $A, B \in wfpf$
- Step:
  - $(\neg A)$ starts with '(' and ends with ')'
  - $(A * B)$ starts with '(' and ends with ')'
- Hence, the claim holds for all formulae $\square$

Semantics

Definition: Let $\mathbb{B} = \{\mathrm{true}, \mathrm{false}\}$ be a set of truth values.

- We assume $\overline{\mathrm{true}} = \mathrm{false}$ and $\overline{\mathrm{false}} = \mathrm{true}$.
- A propositional interpretation is a function $I : V \to \mathbb{B}$ assigning truth values to the propositions
- It is continued to a function $I : wfpf \to \mathbb{B}$ as follows:
  * $I((\neg A)) = \overline{I(A)}$
  * $I((A \vee B)) = \mathrm{true}$ if $I(A) = \mathrm{true}$ or $I(B) = \mathrm{true}$, false otherwise
  * $I((A \wedge B)) = \mathrm{true}$ if $I(A) = \mathrm{true}$ and $I(B) = \mathrm{true}$, false otherwise
  * $I((A \Rightarrow B)) = \mathrm{true}$ if $I(B) = \mathrm{true}$ or $I(A) = \mathrm{false}$
  * $I((A \Leftrightarrow B)) = \mathrm{true}$ if $I(A) = I(B)$, false otherwise.

An interpretation of the variables uniquely determines the interpretation of the formula

Examples

Note: We will implicitely assume a suitable V from now on

$A = (it\_rains \Rightarrow grass\_is\_green)$

- There are 4 possible interpretations:

Interpretation	it_rains	$grass\_is\_green)$
I_0	false	false
I_1	false	true
I_2	true	false
I_3	true	true

What is $I_0(A)$, $I_1(A)$, $I_2(A)$, $I_3(A)$?

Examples

Note: We will implicitely assume a suitable V from now on

$A = (it\_rains \Rightarrow grass\_is\_green)$

- There are 4 possible interpretations:

Interpretation	it_rains	$grass\_is\_green)$
I_0	false	false
I_1	false	true
I_2	true	false
I_3	true	true

What is $I_0(A)$, $I_1(A)$, $I_2(A)$, $I_3(A)$?

- $I_0(A) = \text{true}$
- $I_1(A) = \text{true}$
- $I_2(A) = \text{false}$
- $I_3(A) = \text{true}$

Truth Tables

We can extend the table above into a truth table, showing for each interpretations which value the formula will take:

$I(it\_rains)$	$I(grass\_is\_green)$	$I(A)$
false	false	true
false	true	true
true	false	false
true	true	true

Truth tables are a useful instrument to keep track of the values a formula can take!

Truth tables can also be used to specify the behaviour of operators.

Truth Tables for Standard Operators

$I(A)$	$I((\neg\ A))$
false	true
true	false

$I(A)$	$I(B)$	$I((A \wedge B))$
false	false	false
false	true	false
true	false	false
true	true	true

$I(A)$	$I(B)$	$I((A \vee B))$
false	false	false
false	true	true
true	false	true
true	true	true

$I(A)$	$I(B)$	$I((A \Rightarrow B))$
false	false	true
false	true	true
true	false	false
true	true	true

$I(A)$	$I(B)$	$I((A \Leftrightarrow B))$
false	false	true
false	true	false
true	false	false
true	true	true

One More Example

$f = ((a \wedge b) \Leftrightarrow ((\neg\, a) \vee (\neg\, c)))$

Consider $I(a) = \text{true},\ I(b) = \text{false},\ I(c) = \text{true}$

▶

Consider $I(a) = \text{true},\ I(b) = \text{true},\ I(c) = \text{true}$

▶

One More Example

$f = ((a \wedge b) \Leftrightarrow ((\neg\ a) \vee (\neg\ c)))$

Consider $I(a) = \text{true},\ I(b) = \text{false},\ I(c) = \text{true}$

- $I(f) = \text{true}$

Consider $I(a) = \text{true},\ I(b) = \text{true},\ I(c) = \text{true}$

- $I(f) = \text{false}$

Models and Satisfiability

Definition: Let f be a formula, F be a set of formulae and let I be an interpretation of F

- I satisfies f, if $I(f) = \text{true}$
- In that case, I is called a model of f

We extend this to sets of formulae:

- I satisfies F, if $I(f) = \text{true}$ for all $f \in F$
- In that case, I is called a model of F

We can now distinguish different types of formulae:

- A formula that has at least one model is called satisfiable
- A formula that has no model is called unsatisfiable or contradictory
- A formula for which every interpretation is a model is called valid or a tautology
  In that case, we write $\models f$
- A formula that has at lest one non-model is called invalid

Examples

Examples

Valid:

- $(a \Rightarrow a)$
- $(a \Rightarrow (b \Rightarrow a))$

Invalid, but satisfiable:

- a
- $(a \wedge (b \Leftrightarrow a))$

Unsatisfiable:

- $(a \wedge (\neg\ a))$
- $((a \wedge b) \Leftrightarrow ((\neg\ a) \vee (\neg\ b)))$

Some Laws

Let f be a formula.

If f is valid, $(\neg f)$ is unsatisfiable

If f unsatisfiable, $(\neg f)$ is valid

If f is satisfiable, but invalid, $(\neg f)$ is satisfiable (but invalid)

If $F = \{f_1, f_2, \ldots, f_n\}$, then I is a model of F if I is a model of $(f_1 \wedge (f_2 \wedge \ldots f_n))$

Logical Consequence

Definition: Assume a set F of formulae (the axioms) and a formula f (the conjecture)

- f is called a logical consequence of F, if every model of F is a model of f

In other word, the truth of the formulae in F implies the truth of f

If F implies f, we write $F \models f$

We also say f is a theorem of F

Some Examples

$\{a\} \models (a \vee b)$

$\{a, b\} \models (a \wedge b)$

$\{(a \wedge b)\} \models a$

$\{(a \wedge b)\} \models b$

$\{a, (a \Rightarrow b)\} \models b$ (Modus Ponens)

Deduction

We often are interested in testing if f is a consequence of F

The deduction theorem allows us to reduce this to the test of validity of a single formula:

Theorem: Let F be a set of formulae and a, f be formulae.

$F \cup \{a\} \models f$ if and only if $F \models (a \Rightarrow f)$

We can apply this law repeatedely until F is empty

- We reduce $\{f_1, \ldots f_n\} \models f$ to $\models (f_1 \Rightarrow \ldots (f_n \Rightarrow f))$

Thus, we can reduce theorem proving to validity checking!

Conventions for Writing Propositional Formulae

Since parenthesizing all the formulas can become painful (and error-prone), we use the following conventions:

- The outermost pair of parenthesis can always be dropped
- The operators are assigned a precedence: $\neg \; > \wedge > \vee > \Rightarrow > \Leftrightarrow$
- Operators with a higher precedence bind stronger then operators with a lower precedence
- Parenthesis around formulas which are already clearly defined by precedence can be dropped

Parentheses around associative and commutative $\wedge$ and $\vee$ sequences can be dropped:

- We assume they associate to the left: $a \wedge b \wedge c = ((a \wedge b) \wedge c)$

This does not change the formal definition of well-formed propositional formulae!

- Only shorthand for the "real" representation

Examples

$$a \wedge b \vee c \Leftrightarrow a \Rightarrow a = (((a \wedge b) \vee c) \Leftrightarrow (a \Rightarrow a))$$

$$(a \vee b \vee \neg\, c \vee d) \wedge (\neg\, b \vee \neg\, c) \wedge (\neg\, b) =$$
$$(((((a \vee b) \vee (\neg\, c)) \vee d) \wedge ((\neg\, b) \vee (\neg\, c))) \wedge (\neg\, b))$$

- This example is in clause normal form (CNF), which is much easier to see in the sparingly parenthesized version

$$a \wedge b \wedge c \Rightarrow d \vee e = (((a \wedge b) \wedge c) \Rightarrow (d \vee e))$$

- This a clause in implicational form, equivalent to $\neg\, a \vee \neg\, b \vee \neg\, c \vee d \vee e$

Literature Suggestions

There are hundreds of books on propositional (and first-order) logic. I like: C. Chang and R.C. Lee: "Symbolic Logic and Mechanical Theorem Proving", Academic Press, 1973

On functions and relations, any discrete math or algebra book should cover most. The only English language book I have read is: Seymour Lipschutz: "Schaum's Outline of Linear Algebra" (3rd edition), McGraw-Hill, 2000 – I liked the second edition, but trust this is as good

Much of this is also available in the first few chapters of my Ph.D. Thesis, "Learning Search Control Knowledge for Equational Deduction" (not very pedagogical, but freely downloadable at `http://www4.in.tum.de/~schulz/bibliography.html`)

For Python, I recommend: Guido Van Rossum (edited by Fred L. Drake, Jr.): "Python Tutorial" and "Python Library Reference". Both are freely available in various forms from `http://www.python.org`

Excercise

Voluntarily, no marks. Will be discussed next lecture.

Show: Every well-formed propositional formula contains the same number of junctors and opening parentheses.

Prove the Modus Ponens rule

Exercise I

Claim: Every well-formed propositional formula contains the same number of junctors and opening parentheses. We write $|\ f\ |_{\{c\}}$ to denote the number of occurances of a character from c in f. So we have to show: $|\ f\ |_{\{(\}} = |\ f\ |_o$ (where $o = \{\neg\ , \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ is the set of junctors).

Proof: By structural induction

- Base case: $f \in V$. Then $|\ f\ |_{\{(\}} = 0 = |\ f\ |_o$.
- Induction assumption: Claim holds for $a, b \in wfpf$.
- Induction step:
  Case 1: $f = (\neg\ a)$. Then $|\ f\ |_{\{(\}} = |\ a\ |_{\{(\}} + 1 = |\ a\ |_o + 1 = |\ f\ |_o$
  Case 2: $f = (a * b)$ with $* \in o$.
  Then $|\ f\ |_{\{(\}} = |\ a\ |_{\{(\}} + |\ b\ |_{\{(\}} + 1 = |\ a\ |_o + |\ b\ |_o + 1 = |\ f\ |_o$.

□

Exercise II

Claim: Assume $a, b \in wfpf$. Then $\{a, (a \Rightarrow b)\} \models b$ (the $Modus\ Ponens$)

Proof: To show: Any interpretation that satisfies $\{a, (a \Rightarrow b)\}$ also satisfies b. Let I be an arbitrary interpretation.

Case 1: $I(a) = \text{false}$. Then I is not a model of $\{a, (a \Rightarrow b)\}$, hence we have nothing to show

Case 2: $I(a) = \text{true}$. We further distinguish based on $I(b)$:

Case 2a : $I(b) = \text{false}$. Then $I((a \Rightarrow b)) = \text{false}$ and hence I is not a model of $\{a, (a \Rightarrow b)\}$.

Case 2b : $I(b) = true$. Then I is a model of b.

Hence, I is a model of b whenever I is a model of $\{a, (a \Rightarrow b)\}$, and thus b is a logical consequence of $\{a, (a \Rightarrow b)\}$.

□

CS63Z
Formal Methods in Software Engineering

Python

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Introduction

Python is an object-oriented/imperative, interpreted, elegant high-level programming language

It borrows concepts from

- C/C++ (expression syntax, control flow, classes)
- LISP (dynamic typic, interactive interpreter, lists, lambda-expressions and maps)
- BASIC (string processing, simple syntax)
- Modula-3 (modules, classes and inheritance)
- Perl (regexp support and text processing)
- . . .

Python supports both object-orientend programming and a module concept

- The object model allows for multiple inheritance and is implemented using name spaces
- The module concept is a very simple one module – one file model, but allows for separate importing of names

Python for Implementing Formal Specifications

Very clear syntax

- Block structure defined by indention
- By far the most readable language I know
- Supports embedded documentation and automatic generation of interface documentation from documented code

Nice semantic support

- Supports assertions for ensuring pre- and postconditions
- Strictly (but dynamically) typed (polymorphism is easy)

Built-in datatypes are excellent match for formal concepts

- Built-in set datatype
- Comprehensions: `Set([a**2 for a in [1,2,3,4]])` is the set $\{1, 4, 9, 16\}$
- Also: Tuples, lists, hashes,. . .

Result: Many formal specifications can be mapped rather directly to a Python prototype

More on Python

Python is interpreted

- Interpreter can be used interactively for development (shorter debugging cycles)
- Modules can be byte compiled for faster loading
- Speed is so-so

Python is embeddable and extensible

- Can be used as extension or scripting language for other programs
- It is easy to add C functions for speed-critical operations

Python is actually used a lot (and usage is increasing):

- It's behind much of Google
- Many open-source projects use Python (PIL, fetchmailconfig, . . .)
- . . .

Python has an excellent web site with lots of free documentation at `http://www.python.org/`

Hello World

```
#!/usr/bin/env python
"""
This is the DocString for 'hello.py', a
program that prints 'Hello, World!'
"""
print "Hello, World!"
```

Specify interpreter for UNIX

Starts multi-line verbatim string

Actual code

To run the program:

- Create a file `hello.py` with the above content (using your text editor of choice)
- Make it executable: `chmod ugo+x hello.py`
- Run it: `./hello.py`
- If you want to terminate a program that wont, type `[C-c]` (Hit the Ctrl key and the C key at the same time)

Interactive Python Use

To get to the python prompt, just type python

```
schulz@leonardo 3:57am [PYTHON] python
Python 2.3.5 (#1, Apr 12 2005, 18:47:14)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1671)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can now start typing Python code at the prompt:

```
>>> i = 0
>>> while i < 3:
...     print "Hello, World!"
...     i = i+1
...
Hello World
Hello World
Hello World
>>>
```

Python Program Structure

A Python program is a sequence of statements

- Statements may return values
- Statements may introduce new names and bindings
- In particular, statements can create functions/classes (and name them)

Name spaces are a very central concept in Python!

- Name spaces are used for scoping of names
- Each file has its own global namespace
- Each function creates its own local namespace
- Bindings for names are searched first in local name spaces, then in the global one
- In simple interactive use, every name we create (e.g. by using it at the left hand side of an assignment) is in the global namespace

Comments in Python start with a # and continue to the end of line

```
>>> i = 10 # This is a useless comment
```

Python Data Types I

Python is dynamically typed:

- The type is not a property of the variable, but of the value!
- The same variable can hold values of different type
- We can query the type of a value using `type()`:

```
>>> i=10
>>> print i
10
>>> i="Hello"
>>> print i
Hello
>>> type(i)
<type 'string'>
```

Python Data Types II

Built-in types include:

- Numbers (Integer, floating point, complex)
  - Integers are arbitrary sized
  - Floats are machine floats
- Strings (characters are just strings of lenght 1)
- (Polymorphic) Lists (doubling as arrays)
- Tuples
- Dictionaries (associative arrays or hashes)
- Sets (for Python 2.4 and greater - in 2.3 a library provides them)
- Objects
- Functions
- . . .

Libraries provide more

- Regular expressions
- . . .

Python As a Calculator

Pythons expression syntax is mostly stolen from C

We can type expressions at the Python prompt and get them evaluated:

```
>>> 10 / 3
3
>>> 10 / 3.0
3.33333333333
>>> 10 + 3 *4
22
>>>
>>> (1+5j) + 3j
(1+8j)
```

Assignment works just like in C:

```
>>> a = 20
>>> b = 44
>>> a * b
880
```

Some Python Statements

Printing

- `print` is a build-in command that can be used to print all Python data types
- Multiple arguments are separated by commata, and are printed separated by a single space
- By default, `print` will print a newline. An empty last argument (i.e. a comma as the last character of the statement) will supress this

Assignments

- Assignments use the single equality sign =
- Unlike C, we can assign many values in parallel: `(a,b)=(b,a)`
- If a variable named as an `lvalue` does not exist in the current namespace, it will be created

Expressions

- Expressions can be used as statements
- In interactive use, the value of an expression will be printed back

Example

```
>>> print "Hello ";print "World"
Hello
World
>>> print "Hello ",;print "World"
Hello  World
>>> a = 10
>>> b = 20
>>> print a,b
10 20
>>> (a,b)=(b,a)
>>> print a,b
20 10
>>> 32+b
42
>>>
```

Python Program Structure and Statement Blocks

A single file Python program is a sequence of statements

- Multiple statements can be written on a line, separated by ;
- Hower, normally a single statement is written on a line, terminated by the end of line

Compound statements or block of statements are denoted by indention

- Compound statements are only allowed as part of certain statements (loops, conditionals, function or class definitons, . . .)
- The controlling part of that statement ends in a colon :
- The first line of the compound statement must be indented by an arbitrary amount of white space relative to the context
- All further statements share this indention (but sub-statements may be indented further, giving a recursive block structure as in C)
- The end of the compound statement is denoted by a statement indented to a the same level as any containing context

Statements in Python do not generally return a value!

Example

```
a = 10
if a==10:
   print "The world seems consistent"
   if 17==3:
      print "I'm surprised!"
      a = 13
   print a
   a = 11

print a

The world seems consistent
10
11
```

Flow Control: `if`

`if` is used for conditional execution

Example:

```
if a==10:
   print "A = 10"
elif a==11:
   print "A = 11"
else:
   print "A has unrecognized value"
```

An expression is considered `true`, unless:

- It has the numerical value 0 (as in C)
- It's value is a sequence object of lenght 0 (i.e. an empty list)
- It is Boolean `False`
- It's the special value `None`

Flow Control: `while`

The syntax of `while` is analogous to that of `if`:

```
while <expr>:
    <statment1>
    <statment2>
    ...
```

As in C, `break` and `continue` are supported

- `break` will leave the loop early
- `continue` will immediately return to the top

However, the `while` loop also supports a `else` clause:

- The else clause is executed, if the loop terminates because the condition is false
- It's not executed if the loop is left because of a break!

Example

```
i=1
res=1
while i<=10:
   res = res*i
   i=i+1
else:
    print "Leaving loop"

print "Factorial of 10:", res
```

Function Definitions

Functions are defined using the `def` statement

`def` is followed by a name, a list of formal parameters, and a colon (making up the header), followed by the body (a compound statement)

```
def <name>(<arg1>,<arg2>,...):
    <statement1>
    <statement2>
    ...
```

The function activation ends if a `return` statement is executed

- In that case, the value of the argument of the return is returned as the value of the function call
- Otherwise, execution of the function ends at the end of the body
- In that case, and if `return` is used without argument, the function returns the special value `None`

Example

```
def yes(string,number):
   i=0
   res = ""
   while i<number:
      res = res + string
      i = i+1
   return res

yes("Yes",10)
'YesYesYesYesYesYesYesYesYesYes'
```

CS63Z
Formal Methods in Software Engineering

Interlude: Using Linux on the Lab Machines

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Unix Shell in a Nutshell

Basic interaction: You type a command, the computer obeys

- Most commands take file names as arguments
- Most commands support one or more options, typically of the form `-l` (a dash followed by a letter)

`[C-c]` (`^C`, `CTRL+c`) terminates most programs

Important commands:

- `emacs &` – Start the Emacs editor in the background
- `ls` – List files in currend directory (folder)
- `ls -l` – List files with extra information
- `rm file` – Remove file
- `rm -r dir` – Remove directory and contents
- `chmod ugo+x file` – Make file executable
- `mkdir dir` – Make a new directory
- `logout` – Log out
- `cd dir` – Change to directory dir
- `cd ..` – Change to enclosing directory
- `man command` – Display manual page

Emacs for Everyone

Getting into it: `emacs file` or just `emacs &` (remark: Normally, **emacs** is only started once, and you visit different files from within the editor. Emacs can work on many files at once)

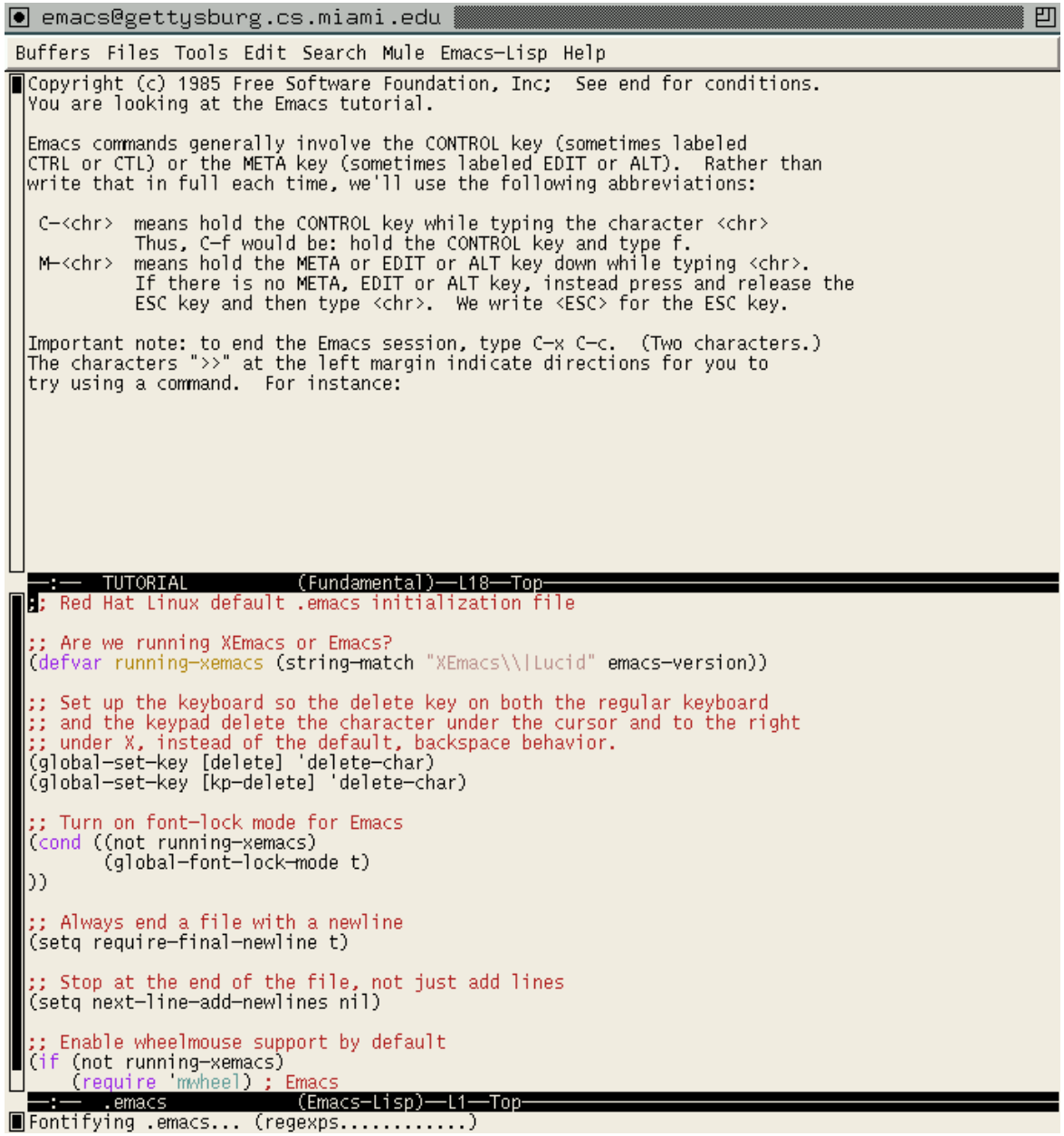
Emacs is extremely configurable and extendable:

- Special modes support nearly all programming languages (Python included)
  - Indentation
  - Compilation/Error correcting
  - Debugging

An Emacs window normally has different sub-regions:

- Menu bar (operate with a mouse, many frequently used commands)
- One or more text windows, each displaying a buffer (a text editing area)
- One mode line for each text window, displaying various pieces of information
- Finally, the mini-buffer for typing complex commands and dialogs

Emacs for Everyone II



Emacs for Everyone III

Commands are typed by using **[CRTL]** or **[ALT]** in combination with normal keys. We write e.g. **[C-a]** or **[M-a]** to denote **[a]** pressed with**[CRTL]** or **[ALT]** (**M** for meta). **[C-h t]** is **[C-h]** followed by plain **[t]**.

Key(s)	What it does
[C-h t]	Enter the emacs tutorial
[C-x C-c]	Leave emacs
Cursor keys	Move around
[C-x C-f]	Open a new file (*)
[C-x C-s]	Save current file
[C-x s]	Save all changed files (*)
[M-x]	Call arbitrary LISP function by name (*)
[C-s]	Incremental search (try it!) (*)

Entries marked with (*) will ask for additional information in the mini-buffer

Exercises

Experiment with bash command line editing and history

Read the **emacs** tutorial

CS63Z
Formal Methods in Software Engineering

Python-II

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Python Type System

The Python type system is dynamic

- Type is a property of the value, not the variable

Every Python value is an object (though not necessarily an instance of a class - build-in types are not typically classes)

Typical object properties:

- Identity: Each object has an identity (e.g. it's address in memory). `id(t)` returns the a representation of the identity of `t`.
- Type: Each object has a type (which tells us how to interprete the bit pattern in memory). `type(t)` returns the type of `t`
- The value of an object is what we are normaly concerned about. Values may be immutable (e.g. the value of the number 1) or mutable (e.g. the value of a list, to which we can add elements), depending on the type of the object
- Objects may have methods and attributes associated with it

Common Operations

Some operations/relations are defined on all objects

Equality: `a == b`

- Does the right thing
- Numbers are equal if they are of the same numerical value (in particular, `1==1.0` and `1==1+0j`
- Strings are equal if they are equal as charecter sequences
- More generally, container types are equal if they have the same elements
- Class instances are equal, if they have the same address in memory

Inequality: `a!=b`

Comparison: `a>b`, `a<b`, `a>=b`, `a<=b`

- Defined between all objects
- On numbers, corresponds standard $>$
- On strings, compares lexicographically
- For objects of different type, arbitrary, but well-defined: `1 > len == True`

Basic Built-In Types

Python implements a number of predefined types. Some important ones are:

- **None:** A type with the single value `None`
- **Numbers**
  - Integers
  - Floating point numbers
  - Complex numbers
- **Sequences**
  - Strings (over ASCII characters) (written with single or double quotes: `"abc" == 'abc'`)
  - Tuples (over arbitrary values)
  - Lists (of arbitrary values)
- **Dictionaries** (associative arrays)
- **Sets** (of arbitrary elements)

Constructing and Converting

Integer numbers — Constructor `int()`

- Accepts reals (rounds down) and strings

Real numbers — Constructor `float()`

- Accepts integers and strings

Strings — Constructor `str()`

- Accepts any object
- If object has a method `__str__(self)`, uses the result (which should be of type string)
- Otherwise, uses built-in rules to generate some reasonably useful representation
- Strings are a basic,built-in type, but not atomic (see below)
- Strings are immutable (cannot be changed)

Examples

Frequent case: Convert (input) string to a number:

- `int("123")` = 123
- `float("123")` = 123.0 (notice silent conversion to float)
- `float("0.2")` = 0.20000000000000001
- Notice: String has to be convertable to target type:

```
>>> int("123.0")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 123.0
```

Number constructors are liberal with white space:

- `int("  123\n")` = 123

Convert everything to string:

- `str(123)` = `'123'`
- `str(len)` = `'<built-in function len>'`

Sequence Types

Sequence types represent ordered collections of objects

- Can be mutable or immutable

Common functions for all sequence types:

- `len(s)` – number of elements in `s`
- `s[i]` – ith element of `s`
  - Counting starts at 0
  - Illegal index raises an `IndexError` exception
  - Convenient: Indexing with negative numbers starts at end (`s[-1]` is the last element of `s`)
- `s[i:j]` – subsequence from ith element (inclusive) to jth element(exclusive (`!`))
  - Also allows negative indices
  - We can leave of values: `s[:-1]` is `s` without the last element, `s[:]` is `s`
- `e in s` – true, if `e` is an element of `s`
  - Also: `e not in s`

Strings

Strings are immutable sequences of ASCII characters (note: Python supports UNICODE strings as a separate but essentially similar data type)

String literals can be written in many different ways:

- Plain strings, enclosed in double or single quotes (`"Hi"`, `'You'`). Escape sequences like `'\n'` will be expanded in a manner similar to C
- Raw strings: `r"Hallo\n"`: No escape seqences are processed
- Tripple quoted strings can span multiple lines, literal newlines are part of the string

While strings are immutable, we can create new strings easily and efficiently

- Slicing allows us to extract substrings
- `'+'` can be used to concatenate strings

String Examples

```
>>> a ="""
... This is a multi-line string
... Here is line 3, lines 1 and 4 are empty!
... """
...
>>> a
'\012This is a multi-line string\012Here is line 3, lines 1 and 4 are empty!\012'
>>> print a

This is a multi-line string
Here is line 3, lines 1 and 4 are empty!

>>> b = a[1:28]+" "+a[29:69]
>>> b
'This is a multi-line string Here is line 3, lines 1 and 4 are empty!'
>>> c="\\\\\\\\"
>>> d=r"\\\\\\\\"
>>> print c
\\\\
>>> print d
\\\\\\\\
```

Tuples

Tuples group 0 or more values (typically 2 or more values) together

Tuples are immutable sequences

Writing tuples:

- Empty tuple: `()`
- Single element tuples: `1,` or `(1,)`
- Multi-element tuples: `1,2,3` or `(1, 2, 3)`
- Note: Sometimes the round brackets are necessary:
  - `(1, 2,3) != (1, (2,3))`
  - It's good style to always write them

Tuple elements and slices are accessed as for all sequences:

- `(1,2,3)[1] == 2`
- `(1,2,3)[1:3] == (2,3)`

Uses for Tuples:

Returning multiple values from a function

Representing relations (as sets of tuples) ;-)

Parallel assignment works via tuples: `a,b,c = 1,2,3`

. . .

Lists

Lists are mutable ordered collections of arbitrary objects

- List literals are written by enclosing a comma-separated list of elements in square braces (`[1,2,3]`, an empty list is written `[]`)
- We can also create lists from any other sequence type using the `list()` function:

```
>>> list("Hallo")
['H', 'a', 'l', 'l', 'o']
```

Frequent list operations:

- Indexing and slicing for the extraction of elements. Note that you can also assign values to slices and elements to change the list!
- Deleting elements with the build-in function `del()`:

```
>>> a = ["a", "b", "c"]
>>> del(a[1])
>>> a
['a', 'c']
```

List Operations

The list data type has a number of useful methods (functions associated with an object). Note that these functions destructively change the list object (i.e. they do not compute a new list and return the value, but change the old list)

- `l.append(x)` will append the element `x` at the end of `l`
- `l.extend(l1)` will append all the elements in `l1` to the end of `l`
- `l.insert(i,x)`: Insert at position `i` in the list (shifting all elements with indices >=`i` by one)
- `l.remove(x)`: Remove first occurrence of `x` from the list
- `l.pop()`: Remove the last element from the list and return it
- `l.pop(i)`: Remove the element with index `i` from the list and return it
- `l.sort()`: Sort the list (assuming there is a > relation
- `l.reverse()`: Reverse the list

List Examples

```
>>> a = []
>>> b = ["a", "b", "c"]
>>> a.append(1)
>>> a.append(2)
>>> a.append(3)
>>> a
[1, 2, 3]
>>> a.extend(b)
>>> b
['a', 'b', 'c']
>>> a
[1, 2, 3, 'a', 'b', 'c']
>>> a.reverse()
>>> a
['c', 'b', 'a', 3, 2, 1]
>>> a.pop()
1
>>> a.pop()
2
>>> a
['c', 'b', 'a', 3]
```

Making Lists of Numbers

The `range()` function creates lists of integers. It can be used with one, two, or three integer arguments

`range(arg1, arg2)` generates a list of integer, starting at `arg1`, up to (but not including) `arg2`:

```
>>> range(10, 20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(10,0)
[]
```

`range(arg)` (single argument) is equivalent to `range(0,arg)`:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The third argument can be used to specify a `step size`:

```
>>> range(0,20,3)
[0, 3, 6, 9, 12, 15, 18]
>>> range(20,10,-1)
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
```

Sets

Sets are unordered collections of unique immutable elements

- Sets are mutable (but there is a FrozenSet type that is immutable – in Python 2.4, you have to explicitely construct FrozenSets if you want them as set elements)
- No sequence is defined (no indexing or slicing)
- No element can be more than once in a set

Sets are created by calling the `Set()` constructor

- Default set is empty
- Optional argument: Any object that supports iteration (elements yielded by iteration become members of sets)

Set support built-in in Python 2.4.X, implemented as a library in Python 2.3

- In either case, you need to import support for sets:

```
from sets import *
```

Non-Updating Set Operations

These operations leave the original set(s) unaffected

- If necessary, a new set is created

Most ordinary set operations are implemented in Python:

- Cardinality (member of elements) ($\#A$): `len(A)`
- Membership test: $a \in A$ iff `a in A`
- Non-membership test: $a \notin A$ iff `a not in A`
- Subset relation: $A \subseteq B$ iff `A <= B` (similar for >=)
  - Alternative: `A.issubset(B)`, `A.issuperset(B)`
  - Notice: `A==B` is equivalent to `A>=B and B<=A`
- Union: $A \cup B$ corresponds to `A | B` and `A.union(B)`
- Intersection: $A \cap B$ corresponds to `A & B` and `A.intersection(B)`
- Difference: $A - B$ corresponds to `A - B` and `A.difference(B)`

Updating Set Operations

Often, it is much more efficient to modify an existing set!

- $A = A \cup B$: `A.update(B)`
- $A = A \cap B$: `A.intersection_update(B)`
- `...`

Also for individual elements:

- `A.add(x)`: Add x into A
- `A.remove(x)`: Remove x from A, complain (via `KeyError` exception) if $x \notin A$
- `A.discard(x)`: Remove x from A if $x \in A$, do nothing otherwise

Iteration

All sequences and sets support iteration over elements with `for`!

```
>>> a = Set([2,1,1,3])
>>> for i in a:
...     print i
...
1
2
3
>>> for i in "Hello":
...     print i
...
H
e
l
l
o
```

List Comprehensions and Sets

Iteration can also be used for list comprehensions

- Syntax: Expression, followed by at least one `for` clause, followed by any number of `for` and `if` clauses, enclosed in square braces
- Example:

```
>>> l = [(a, a+2) for a in range(30) if a % 3 == 0]
>>> l
[(0, 2), (3, 5), (6, 8), (9, 11), (12, 14), (15, 17), (18, 20)
>>> [t[0]+t[1] for t in l]
[2, 8, 14, 20, 26, 32, 38, 44, 50, 56]
```

To use this as set comprehensions, just use the `Set()` constructor:

-
```
>>> Set([ a**b for a in [1,2,3,4] for b in [1,2,3]])
Set([64, 1, 2, 3, 4, 8, 9, 16, 27])
```

Dictionaries

Dictionaries are another mutable compound data type

- The allow the association of keys and values
- Other names: Associative arrays (AWK), Hashes (Perl)

Dictionaries act like arrays where any immutable object can be used as an index

- Written representation: List of `key:value` pairs in curly braces:

```
a ={} # Empty dictionary
b = {1:"Neutral element", 2:"Smallest prime", 3:"Prime",
     4:"Composite, 2*2"}
```

Dictionaries support iteration and `in`

- Both are applied to the keys (not to the values or the key/value tuples)

Dictionary Operations

Assume `d` is a dictionary

- `d[key] = value` adds the pair `key:value` to the dictionary (if an old value for `key` existed, it is forgotten
- `del(d[key])` deletes the element with key `key`
- `d.has_key(key)` returns true, if `key` is a in `d` (equivalent: `key in d`)
- `d.keys()` returns a list of all keys in `d` (equivalent: `[key for key in d]`

Example:

```
b = {1:"Neutral element", 2:"Smallest prime", 3:"Prime",
     4:"Composite, 2*2"}
for i in range(1000):
   if b.has_key(i):
      print i, ":", b[i]
   else:
      print "No information about ",i
```

Excursus: Accessing the Command Line

A running UNIX program has access to its command line arguments

In Python, this is done by accessing a predefined variable `argv` in the `sys` module:

```
#!/usr/bin/env python

import sys

print sys.argv
```

- `sys.argv` is a list of strings
- Represents the whole command line (i.e.) first element is the name of the program

Files and I/O

Scripting languages often have very convenient text I/O functions – Python is no exception

I/O is performed on file objects

File objects are created with `open(Name,Mode)`

- `Name` is a string containing a valid filename
- `Mode` is a string containing the mode (e.g. reading (`"r"`) or writing (`("w")`))

If `f` is a file object, the following methods are defined:

- `f.readline()`: Read a single line and return it (including `'\n'`). Return empty string on EOF
- `f.readlines()`: Read the whole file into a list of strings, one line per element
- `f.read()` reads the whole file into a single string
- `f.read(bytes)` reads up to `bytes` into a string

Predefined File Objects

UNIX defines three I/O streams for each running program:

- `stdin` for input
  - Usually connected to the keyboard
- `stdout` for normal output
  - Normally connected to the terminal
  - This is where `print` prints to
- `stderr` for error messages and other unusual output
  - Normally connected to the terminal

In Python, these are available in the module `sys` as `sys.stdin`, `sys.stdout`, `sys.stderr`

```
import sys
a = sys.stdin.readline()
```

Example: Sorting Files

```
#!/usr/bin/env python

import sys

if len(sys.argv) == 1:
    l = sys.stdin.readlines()
else:
    l = []
    for name in sys.argv[1:]:
        f = open(name,"r")
        l.extend(f.readlines())
        f.close()

l.sort()

for line in l:
    print line,
```

Objects and Classes

Objects combine data and operations in one entity

- Consequently, they contain functions and plain variables as elements

Common terminology:

- Elements of an object are called members
- Functions are called member functions or methods
- Variables are called data members or attributes

In Python, all members are public

- Public members can be accessed from anywhere in the program

(Most) objects are instances of a class

Classes

Python implements classes as separate name spaces, using the class keyword

Syntax:

```
class NewClassName(base_classes):
   <StatementBlock>
```

All statements in StatementBlock are executed in a new namespace and create local names

- Assignments create data members
- `def` statements create member functions

The new class inherits members of the base classes

- It is generally recommended to inherit a built-in class or `object`

Member Functions in Python

Member functions in Python are defined in the scope of a class definiton

- They have no special access to object members
- However, they are always called with a first argument that represents the object, and they can use this argument to manipulate object members

Example:

```
class SimpleString(object):
   str = "Default String"
   def QueryString(self):
      return self.str

   def SetString(self, arg):
      self.str = arg
```

Instances and Methods

Instances of a class are created by treating the class name as a function without arguments:

```
>>> a = SimpleString()
>>> print a
<__main__.SimpleString instance at 80b25d0>
```

Unless explicitely changed, instances inherit the values of class variables:

```
a.str
'Default String'
```

Methods in an instance are called as members, but without the initial argument (`self`), which is supplied by the Python system:

```
>>> a.SetString("New String")
>>> a.str
'New String'
>>> a.QueryString()
'New String'
```

Contructors and Destructors

Most often, we want to create an object in a well-defined initial state

We also may need to perform clean-up action if an object gets destroyed (e.g. close a file)

Constructors are functions that are automatically called when an object is created

- In Python, a method called `__init__(self)` acts as a constructor
- `__init__()` can also take additional arguments (which have to be added to the class creation call

Destructors are called when an object is destructed (note that this acutally only happens when the object is garbage collected!)

- The destructor is called `__del__(self)`

Note: Constructors and (especially) destructors are a lot more important in languages with explicit memory handling (e.g. C++)!

Example: Stacks as Objects

```
class Stack(object):
   StackEmpty = "StackEmpty"

   def __init__(self):
      self.stack = []

   def __del__(self):
      if not self.Empty():
         print "Warning: Non-Empty stack destroyed"

   def Push(self,element):
      self.stack.append(element)

   def Empty(self):
      return len(self.stack)==0

   def Pop(self):
      if(self.Empty()):
         raise Stack.StackEmpty
      return self.stack.pop()
```

Example Continued

```
>>> a = Stack()
>>> b = Stack()
>>> a.Push(10)
>>> b.Push(20)
>>> a.stack
[10]
>>> b.stack
[20]
>>> a.Push(11)
>>> a.Pop()
11
>>> a.Pop()
10
>>> try:
...   a.Pop()
... except Stack.StackEmpty:
...    print "Caught Exception"
...
Caught Exception
>>> a = 10 # a will be garbage-collected, but is empty
>>> b = 10 # b will be garbage-collected
Warning: Non-Empty stack destroyed
```

Mapping Formal Concepts to Python

Sets , tuples

- Built-in

(Finite) Cross products

- Via comprehensions: `Set([(a,b) for a in A for b in B])`

Finite Relations

- Relations are just sets

Finite Functions

- Either as relations, or, more efficiently, as dictionaries

Infinite cardinality functions/relations

- By building appropriate classes

Lab Notes

Make sure your computer has Python 2.4 installed

- Type `python2.4` at the shell prompt
- If you don't find it, tell me (so I can tell the sysadmin) and choose another computer

The first lines of your program should be:

```
#!/usr/bin/env python2.4

from sets import *
```

Don't call you program `sets.py`

- . . . or Python will try to import the `sets` module from your file. . .

Lab Exercises

Let S be a finite set and A be a binary relation over S (set of 2-tuples). Write Python functions that

- Compute the reflexive closure of A
- Compute the inverse, A^{-1}
- Compute the transitive closure of A
- Compute the reflexive, transitive and symmetric closure of A.

Write a Python function that computes the power set of a finite set

Example Solution to first Lab Exercise

Let S be a finite set and A be a binary relation over S (set of 2-tuples). Write a Python function that computes the reflexive closure of A

- Definition: A binary relation $A : S \leftrightarrow S$ is a set of tuples over a, i.e. $A \subseteq S \times S$ (equivalent: $A \in \mathbb{P}(S \times S)$)
- Definition: The reflexive closure of A (written A^0) is the smallest relation that includes A and is reflexive
  - So: $A \subseteq A^0$ (required)
  - So: $\mathrm{id} \subseteq A^0$ (remember: $id = \{(a, a) \mid a \in S\}$)
  - $A \cup \mathrm{id}$ is reflexive, and hence $A^0 = A \cup \mathrm{id}$

Implementation is straightforward (due to Pythons set data type):

- Function takes A and S as parameters
- Compute $\mathrm{id} : A \leftrightarrow A$
- Compute $A^0 = A \cup \mathrm{id}$
- Return the result

Actual Solution Code

```
#!/usr/bin/env python2.4

from sets import *

def reflexive_closure(relation, base_set):
    """
    Given a binary relation (a set of 2-tuples over a base set),
    return the reflexive closure  of it.
    """
    return relation.union(Set([(a,a) for a in base_set]))

#Some testing code:
example_set = Set(range(10))
test_rel1 = Set([(0,1), (1,2), (2,3)])
test_rel2 = Set([(0,0), (6,7), (7,8), (3,1)])

print "Relation 1:", test_rel1
print "Reflexive closure: ", reflexive_closure(test_rel1, example_set)

print "Relation 2:", test_rel2
print "Reflexive closure: ", reflexive_closure(test_rel2, example_set)
```

Programming Assignment (1)

Write a propositional validity checker in Python (4 marks)

- The program should read a well-formed propositional formula from a file given on the command line
- If that formula is valid (i.e. if every interpretation is a model), it should print:
  `TSTP exit status: Tautology`
- Otherwise, it should print
  `TSTP exit status: Countersatisfiable`

Individual steps:

- Design and implement a suitable datatype for repesenting formulae (include at least a parser and a pretty-printer) (1 mark)
- Design and implement a suitable data type for interpretations and a function that evaluates a formula under an interpretation (1 mark)
- Write a function that enumerates all possible interpretations (1 mark)
- Integrate everything into a working system (1 mark)

Document and comment your work. Quality (not length) of comments and documentation will influence grading!

Programming Assignment (2)

The concrete language for formula is given by the following BNF:

```
<proposition> ::= [a-zA-Z][A-Za-z0-9_]*
<binop>       ::= '&' | '|' | '=>' | '<=>'
<formula>     ::= <proposition> |
                  '(' '~' <formula> ')' |
                  '(' <formula> <binop> <formula> ')'
```

Some valid examples:

```
(joe => (bob & carrol)) /* Junk comment */

(~karl)

(this <=> (goes | ((~deep ) => or)))
```

The course web page has a suitable lexical analyser (which also handle white space and comments), an simple example of a tool doing lexical analysis, and an example of a simple recursive descent parser for the language.

Programming Assignment (3)

Note that we use the following symbols/strings for junktors:

Concrete language	English	Math
&	Logical and	$\wedge$
|	Logical or	$\vee$
~	Logical not	$\neg$
=>	Implication	$\Rightarrow$
<=>	Equivalence	$\Leftrightarrow$

If you come up with a smarter way than enumerating all interpretations, you will get full marks (assuming adequate documentation and quality)

I'm available for discussing preliminary versions (I will only grade the final version, not intermediate code you show me)

Due date: Tuesday, July 12th (but I suggest you start early!)

CS63Z
Formal Methods in Software Engineering

The Meaning of Life
(or at least Programs)

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

What is the Meaning of this Program?

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)
    return (z,s)
```

Hint

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)
    return (z,s)
```

```
mysterious(10,3)  : (3, 1)
mysterious(12,3)  : (4, 0)
mysterious(12,4)  : (3, 0)
mysterious(113,7) : (16, 1)
mysterious(128,2) : (64, 0)
```

Guess

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)
    return (z,s)
```

Program computes $(\frac{x}{y}, x\%y)$

- Integer division with rest

But:

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)

    return (z,s)


mysterious(-1,3)  : (0, -1)
mysterious(-100,3): (0, -100)
```

But:

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)

    return (z,s)


mysterious(-1,3)  : (0, -1)
mysterious(-100,3): (0, -100)
mysterious(100,-3):
```

But:

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)

    return (z,s)
```

```
mysterious(-1,3)  : (0, -1)
mysterious(-100,3): (0, -100)
mysterious(100,-3): <nothing happens for a long time>
```

But:

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)

    return (z,s)
```

```
mysterious(-1,3)  : (0, -1)
mysterious(-100,3): (0, -100)
mysterious(100,-3): <nothing happens for a long time>
mysterious(7,0)   :
```

But:

```
def mysterious(x,y):
    z = 0
    s = x
    while s >= y:
        assert x == z*y+s
        (z,s) = (z+1, s-y)

    return (z,s)
```

```
mysterious(-1,3)  : (0, -1)
mysterious(-100,3): (0, -100)
mysterious(100,-3): <nothing happens for a long time>
mysterious(7,0)   : <nothing happens for a very long time>
```

Observations

"Correctness" depends on proper input

- Program correctly computes integer division and rest
- . . . but only if input obeys certain preconditions

"Meaning" of the program is a partial function

- Maps input states (variables) to output states (variables)
- Program may fail to terminate on certain inputs

We can express (parts of) the intended meaning by pre- and postconditons

- Postconditions are predicates which have to be true at the end of the program
- Again, correctness depends on postconditions
- Every program is correct with precondition false and postcondition true

Python Excursus: `assert`

Assertions are statements of neccesarily true predicates on the state of the program

- Preconditions are one example

Python (and other languages) support assertions at run-time:

- `assert <expr>` will check if `<expr>` evaluates to `True`
- If yes, nothing happens
- If no, raises `AssertionError` (which, by default, aborts program)

Assertions are important for reasoning about programs

`assert` allows us to explicitely state assertions

- Also supports testing
- Extensive use of `assert` is good defensive programming

Example with pre- and postconditions

```
def mysterious(x,y):
    assert type(x) == int and x>=0
    assert type(y) == int and y>0
    z = 0
    s = x
    while s >= y:
        (z,s) = (z+1, s-y)

    assert (x == z*y+s) and (s >= 0) and (s<y)
    return (z,s)
```

Claims: If the preconditions are true. . .

- the program always terminates
- at the end, the postcondition is true
- (which implies it does the division/remainder)

But can we prove it?

Termination

Can we show that the division program always terminates if the precondition is met?

- It is well known that in program termination is, in general, undecidable (this is the famous Halting Problem)
- However, for many concrete programs, it is possible!

Termination

Can we show that the division program always terminates if the precondition is met?

- It is well known that in program termination is, in general, undecidable (this is the famous Halting Problem)
- However, for many concrete programs, it is possible!

```
def always_terminates(x):
    y = x
    return y
```

Question: Is it possible for this program?

Termination

Can we show that the division program always terminates if the precondition is met?

Termination

Can we show that the division program always terminats if the precondition is met?

The only important question is if the `while` loop terminates

- All other statements in the program terminate obviously
- Finite sequence of terminating statements also terminates

Argument for the `while`:

- `y` is a positive integer and not modified in the program
- `s` is initialized as a positive integer
- Each iteration of the loop, `s` is decreased by `y`
- `s` can only be decreased a finite number of times before it becomes smaller than or equal to y
- Then the condition of the while loop fails and the program terminates

The Loop Invariant

```
def mysterious(x,y):
    assert type(x) == int and x>=0
    assert type(y) == int and y>0
    z = 0
    s = x
    while s >= y:
        assert (x == z*y+s) and (s >= 0) # Loop invariant!
        (z,s) = (z+1, s-y)

    assert (x == z*y+s) and (s >= 0) and (s < y)
    return (z,s)
```

Claims: If the preconditions are true. . .

- the program always terminates
- at the end, the postcondition is true
- (which implies it does the division/remainder)

But can we prove it?

Loop Invariants

A loop invariant is a predicate on the state of a program

- It is associated with a single loop
- It always has to hold at a given point in the loop (i.e. it is invariant over the number of executions of the loop)
- Typically, we specify a loop invariant at the top of a loop

Finding good loop invariants is a critical skills

- Often necessary for proving correctness
- Generally extremely useful for reasoning about programs
- Also helps in developing algorithms!

If `while(C):assert I;<body>` terminates, we can deduce:

- $\neg C$ (the negation of the controling expression)
- I (the invariant still holds)

Showing the Invariant

```
def mysterious(x,y):
    assert type(x) == int and x>=0
    assert type(y) == int and y>0
    z = 0
    s = x

    while s >= y:
        assert (x == z*y+s) and (s >= 0) # Loop invariant!
        (z,s) = (z+1, s-y)

    assert (x == z*y+s) and (s >= 0) and (s < y)
    return (z,s)
```

Idea: Induction over number of executions of loop

- Base case: First execution
- Step: Show that if the invariant and the loop guard both hold, then the invariant will hold after execution of the loop body

Writing Convention

We often have to reason about the value of a variable or the state of a program

- . . . before a certain operation (i.e. for preconditions)
- . . . after this operation

To avoid inconvenient (and error-prone) verbiage, we adopt the following conventions:

- Plain variable name ($x, y, \ldots$) represent the state of variable before an operation
- Primed variable names ($x', y', \ldots$) represent the state after an operation

Example:

```
x = 2*y
y = x+3
```

- For this fragment of code, we can e.g. say:
  * $x' = 2 * y$
  * $y' = 2 * y + 3$

Writing Convention (contd.)

Note that I usually write variable names:

- $like\_this$ in mathematical expression
- `like_this` in programs
- This is purely for convenience

Showing the Invariant: Base Case

```
def mysterious(x,y):
    assert type(x) == int and x>=0
    assert type(y) == int and y>0
    z = 0
    s = x
    assert z = 0 and s = x and s >= 0 and (other preconditions)
    while s >= y:
        assert (x == z*y+s) and (s >= 0) # Loop invariant!
        (z,s) = (z+1, s-y)

    assert (x == z*y+s) and (s >= 0) and (s < y)
    return (z,s)
```

As $z = 0$ and $s = x$, we can conclude $x = z * y + s$

As $s = x$ $x >= 0$ (precondition), we can can conclude $s >= 0$

Hence, the loop invariant holds on first entry of the loop

Showing the Invariant: Step Case

```
assert z = 0 and s = x and s >= 0 and (other preconditions)
while s >= y:
    assert (x == z*y+s) and (s >= 0) # Loop invariant!
    (z,s) = (z+1, s-y)
```

At the head of the loop we know:

- $s >= y$ (else we would not enter the loop)
- $(x == z * y + s)$ and $(s >= 0)$ (the invariant, by assumption)

We have to show: $(x' == z' * y' + s') and (s' >= 0)$ holds after execution of the loop body

- x does not change in the loop, i.e. $x' = x$
- Similarly, $y' = y$
- $z' = z + 1$ and $s' = s - y$
- $s >= y$ and hence $s - y = s' \geq 0$
- $z' * y' + s' = (z + 1) * y' + s' = (z + 1) * y + (s - y) = z * y + y + s - y = z * y + s = x = x'$

So the invariant indeed is invariant ;-)

One more Example (Variant 1)

```
def ggt1(x0,y0):
    if y0 > x0:
        (x,y) = (y0, x0)
    else:
        (x,y) = (x0, y0)

    while y != 0:
        if x>y:
            (x,y) = (x-y, y)
        else:
            (x,y) = (x, y-x)
    return x
```

Variant 2

```
def ggt2(x0,y0):
    (x,y) = (x0, y0)
    while x != y:
        if x>y:
            (x,y) = (x-y, y)
        else:
            (x,y) = (x, y-x)
    return x
```

Questions (and Answers)

Are the two variants equivalent?

- No, counterexample (10,0)

What are they supposed to do?

- Compute the greatest common divisor of two (non-negative/positive) numbers

Does it always terminate?

- No.

See next few slides for the rest...

What are reasonable preconditions?

Given reasonable preconditions, what do they do?

- Give a good invariant!
- Prove the invariant
- Prove your idea about the meaning of the programs

Preconditions for `ggt1()`

The Greates Common Divisor, $gcd(x0, y0)$, is only well-defined if:

- $x0, y0 \in \mathbb{N}$
- Either $x0 > 0$ or $y0 > 0$
- We believe that these conditions are sufficient to show that the program computes the GCD of its arguments
- Hence:

```
assert type(x0)==int and x0>=0 and type(y0)==int
       and y0>=0 and (x0>0 or y0>0)
```

- For simplicity, we will from now on assume that all variables are integer unless otherwise specified

What can we conclude just before the `while` loop?

- Case 1: $y0 > x0$. Then $x = y0$ and $y = x0$, hence $y \geq 0$ and $x > 0$ and $x > y$
- Case 2: $y0 \leq x0$. Then $x = x0$ and $y = y0$. Hence $y \geq 0$ and $x \geq y$ and $x > 0$ ($x0$ and $y0$ can't be both 0 by precondition).
- Common in both cases: $y \geq 0$, $x > 0$, $x \geq y$
- So we can assert this

Preconditions for `ggt1()`

```
def ggt1(x0,y0):
    assert x0>=0 and y0>=0 and (x>0  or y > 0) # (A1) Int implicit
    if y0 > x0:
        (x,y) = (y0, x0)
    else:
        (x,y) = (x0, y0)

    assert y>=0 and x>0 and x>=y # (A2)

    while y != 0:
        if x>y:
            (x,y) = (x-y, y)
        else:
            (x,y) = (x, y-x)
    return x
```

Termination (1)

We will now show termination of `ggt1(x0,y0)` for the case that the precondition (A1) holds

Idea: We will show:

- $x + y$ is decreased on each loop traversal
- $x + y$ never becomes negative
- Therefore, there can be only finitely many loop traversals ($>: \mathbb{N} \leftrightarrow \mathbb{N}$ is well-founded)

In order to show the above, we first show the following loop invariant: $x > 0$ and $y \geq 0$

- Base case: Follows from assertion (A2), proven above
- Step case: We know $x > 0$ and $y \geq 0$ (loop invariant) and $y! = 0$ (otherwise we would not enter the loop). We have to show: $x' > 0$ and $y' \geq 0$.
  - Case 1: $x > y$. Then $x' = x - y > 0$. $y' = y \geq 0$ by assumption.
  - Case 2: $x \leq y$. Then $x' = x > 0$ by assumption. $y' = y - x \geq 0$.
  - Thus, the invariant holds

ggt1() with invariant for termination proof

```
def ggt1(x0,y0):
    assert x0>=0 and y0>=0 and (x>0  or y > 0) # (A1) Int implicit
    if y0 > x0:
        (x,y) = (y0, x0)
    else:
        (x,y) = (x0, y0)

    assert y>=0 and x>0 and x>=y # (A2)

    while y != 0:
        assert x>0 and y >= 0 # (A3)
        if x>y:
            (x,y) = (x-y, y)
        else:
            (x,y) = (x, y-x)
    return x
```

Termination (2)

With the invariant, we can now show termination

Claim 1: In the main loop the value of $x + y$ decreases every time (i.e.: $x + y > x' + y'$)

Proof:

- Case 1: $x > y$. Then $x' = x - y$ and $y' = y$. But we know $y \geq 0$ (Invariant, (A4)) and $y \neq 0$ (otherwise we would not enter the loop body). Thus $x - y < x$ and hence $x' < x$. This yields $x' + y' < x + y$.
- Case 2: $x <= y$. Then $x' = x$ and $y' = y - x$. By invariant $x > 0$. This yields the result.

Claim 2: In the loop, $x + y$ is never less then 0.

Proof: Follows directly from the invariant

Claim 1 and 2 imply that the loop can only be executed a finite number of times. Hence the program must terminate.

Correctness

We have termination of `ggt1()` nailed down

We have not yet shown that the prgramm acually computes the GCD

Suggestion: We might try to show that $gcd(x, y) == gcd(x0, y0)$ is an invariant

Some Useful Properties of GCD's

Simple laws

- $gcd(x, y) = gcd(y, x)$
- $gcd(x, x) = x$ if $x > 0$
- $gcd(x, 0) = x$ if $x > 0$
  (since we allow that as input, we might as well define it that way)
- But: $gcd(0, 0)$ is undefined!

Crucial (?) GCD Property

If $x > y$, then $gcd(x, y) = gcd(x - y, y)$

- Proof: If $z = gcd(x, y)$, then z divides both x and y.
  Hence we can write $y = z * k$ and $x = z * (k + l)$ with $l > 0$. Thus:

$$\begin{aligned} x - y &= z * (k + l) - z * k \\ &= z * l \end{aligned}$$

  Hence z also divides $x - y$.
  Assume z is not the GCD of $x - y$ and y. Then there exists a $u > z$ that divides $x - y$ and y. Then:

$$\begin{aligned} x - y &= k_0 * u && \text{for some } k_0 \in \mathbb{N} \text{ (since u divides } x - y) \\ y &= l_0 * u && \text{for some } l_0 \in \mathbb{N} \text{ (since u divides } y) \\ x &= (k_0 + l_0) * u && \text{(add the preceeding two lines)} \end{aligned}$$

  Hence u divides both x and y, and $u > z$. Thus z is not the GCD of x and y. This contradicts our premise. Hence the assumption is wrong and there is no such u. Therefore, z is indeed GCD of $x - y$ and y.

Now let's Proof Correctness of ggt1(x,y)

```
def ggt1(x0,y0):
    assert x0>=0 and y0>=0 and (x>0  or y > 0) # (A1) Int implicit
    if y0 > x0:
        (x,y) = (y0, x0)
    else:
        (x,y) = (x0, y0)

    assert y>=0 and x>0 and x>=y # (A2)

    while y != 0:
        assert x>0 and y >= 0 # (A3)
        assert gcd(x,y) == gcd(x0, y0) # (A4) To show!
        if x>y:
            (x,y) = (x-y, y)
        else:
            (x,y) = (x, y-x)

    assert x == gcd(x0, y0) # (A5) To show!
    return x
```

One more version of the assertions

```
def ggt1(x0,y0):
    assert x0>=0 and y0>=0 and (x>0  or y > 0) # (A1) Int implicit
    if y0 > x0:
        (x,y) = (y0, x0)
    else:
        (x,y) = (x0, y0)

    assert y>=0 and x>0 and x>=y # (A2)
    while y != 0:
        assert x>0 and y >= 0 # (A3)
        assert gcd(x,y) == gcd(x0, y0) # (A4) To show!
        if x>y:
            (x,y) = (x-y, y)
        else:
            (x,y) = (x, y-x)

    assert gcd(x0,y0) == gcd(x,y) and y=0 # (A6) From (A4), loop exit
    assert x == gcd(x0, y0) # (A5) From (A6) and GCD laws
    return x
```

Some Observations

Now not all of our assertions are necessarily proper Python

- (A4), (A5) and (A6) use `gcd()` as a primitive
- We can prove assertions that we cannot even write in Python (and hence whose violation we could never find by testing)

To show program properties, we used:

- Mathematical laws (GCD, addition, . . .)
- Logic
- Assumptions about the behaviour of the program
  - Executing `x=10` implies that $x == 10$ afterwards ;-)

We showed correctness with respect to preconditions and postconditions

- "Correctness" only makes sense if we know what the program is supposed to do

We showed termination with respect to preconditions

- Termination not guaranteed for nonsensical input

CS63Z
Formal Methods in Software Engineering

Solution to Programming Exercises

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Reflexive Closure

```
def reflexive_closure(relation, base_set):
    """
    Given a binary relation (a set of 2-tuples over a base set),
    return the reflexive closure  of it.
    """
    return relation.union(Set([(a,a) for a in base_set]))
```

Inverse Relation

```
def inverse(relation):
    """
    Given a relation, compute its inverse.
    """
    return(Set([(b,a) for (a,b) in relation]))
```

Transitive Closure

```
def transitive_closure(relation):
    """
    Given a binary relation, compute its transitive closure.
    """
    res = Set()
    missing = relation
    while missing:
        res.update(missing)
        missing = Set([(a,d) for (a,b) in res \
                             for (c,d) in res \
                             if b==c and (a,d) not in res])
    return res
```

Reflexive, Symmetric, Transitive Closure

```
def ref_sym_trans_closure(relation, base_set):
    """
    Given a binary relation over a base set, return the reflexive,
    transitive and symmetric closure of it.
    """
    return transitive_closure(reflexive_closure(\
        relation|inverse(relation),base_set))
```

Power Set

```
def power_set(set):
    """
    Given a set, return its power set.
    """
    res = Set([frozenset()]) # Power set of the empty set
    for e in set:
        new_subsets = Set([ i | frozenset([e]) for i in res])
        res.update(new_subsets)
    return res
```

Note that it returns a set of `frozenset` (Python 2.4) or `ImutableSet`

- Plain sets cannot be in sets
- Note that Python 2.4 requires the `frozenset()` constructors in lines 4 (and it is better to also have it in line 6)

CS63Z
Formal Methods in Software Engineering

Exercise Solutions

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Excercise Solutions

1.1 Show: If $A, B \in \mathbb{F}$, then $\#(A \times B) = \#A \times \#B$.
Proof: Let $A \in \mathbb{F}$ be an arbitrary finite set. We show the conjecture by induction over $\#B$.

Base case: $\#B = 0$. Then $B = \varnothing$. $A \times \varnothing = \{(a, b) \mid a \in a, b \in b\} = \varnothing$.
Hence $\#(A \times B) = 0 = \#A \times 0 = \#A \times \#B$.

Induction hypothesis: For $A, B \in \mathbb{F}$ with $\#B = n$: $\#(A \times B) = \#A \times \#B$.

Remark: If your base case is $\#A = 0$ and $\#B = 0$, then you cannot assume the hypothesis for arbitrary A and B, but only for the case $\#A = \#B$

- Not sufficient for the general case!

Excercise Solutions

1.1 Induction step: Let $\#B' = n + 1$. Then we can write $B' = B \uplus \{e\}$ and

$$
\begin{aligned}
&\#(A \times B') \\
&\quad = \#((A \times B) \uplus (A \times \{e\})) && \text{The new tuples are exactly those having} \\
&& & \text{the new element as the second member} \\
&\quad = \#(A \times B) + \#(A \times \{e\}) && \text{since both sets are disjoint} \\
&\quad = \#A \times \#B + \#(A \times \{e\}) && \text{by induction hypothesis} \\
&\quad = \#A \times \#B + \#A && \text{(each element in } A \text{ combines with } e\text{)} \\
&\quad = \#A \times (\#B + 1) && \text{(arithmetic)} \\
&\quad = \#A \times \#B' && \text{since } B' = B \uplus \{e\}
\end{aligned}
$$

□

Excercise Solutions

1.2 Show: If f is an injective function, then f^{-1} is a function.

Proof: Let $f : A \rightarrowtail B$ be an injecive function. By definition, f^{-1} is a relation $f^{-1} : B \leftrightarrow A$. It remains to be shown that for all $b \in B, a, c \in A$: If $(b, a) \in f^{-1}$ and $(b, c) \in f^{-1}$, then $a = c$ (i.e. f^{-1} relates each element of B to at most one element of A).

So let $b \in B, a, c \in A$ and assume $(b, a), (b, c) \in f^{-1}$. But then $(a, b), (c, b)$ in f. Since f is injective, $a = c$.

□

Literature Suggestions

Theoretical Background:

- J.W. de Bakker: "Mathematical Theory of Program Correctness", Prentice-Hall, 1980
- J. Loeckxs and K. Sieber: "The Foundations of Program Verification (2nd ed.)", Teubner, 1987

Recent Application and Systems Papers:

- M. Whalen and J. Schumann and B. Fischer: "Synthesizing Certified Code". In: *FME 2002: Formal Methods - Getting IT Right (Proc. FME 2002, Copenhagen, Denmark)*, Springer LNCS 2391, 2002
- Crocker, D.: "Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement". In: *Tool Exhibition Notes, 12th International FME Symposium, Pisa*, 2003

Other:

- Jim Woodcock: "Dependable Systems Evolution", in: Tony Hoare and Robin Milner, *Grand Challenges in Computing – Research*, The British Computer Society, 2004

CS63Z
Formal Methods in Software Engineering

Interlude: Test Exam

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Foundations

Let $f : A \nrightarrow B$ be a surjective function. Which of the following statements are true? (0.25 Marks each)

1. $\mathrm{dom}\, f = A$ No (f may be non-total)
2. $\mathrm{dom}\, f = B$ No (Nonsense)
3. $\mathrm{ran}\, f = B$ Yes (f is surjective)
4. f^{-1} is a binary relation Yes (any relation has an inverse relation)

Let $succ : \mathbb{Z} \leftrightarrow \mathbb{Z}$ be defined by $succ = \{(a, a-1) \mid a \in \mathbb{N}_1\} \cup \{(-a, -a-1) \mid a \in \mathbb{N}_1\}$.

- List 5 elements of $succ$ (0.5 Marks) (5,4), (4,3), (3,2), (2,1), (-4, -5)
- Prove or disprove: $succ$ is a partial ordering. (0.5 Marks) It's not, as it is not transitive (e.g. $(3,2), (2,1) \in succ$, but $(3,1) \notin succ$)

Propositional Logic

Assume $F \subset wfpf, f \in wfpf$.

- Define: f is a logical consequence of F (0.25 Marks) f is a logical consequence of F if every model of F also is a model of f
- Define: f is a tautology (0.25 Marks) f is a tautology if every interpretation is a model of f

Assume $F = \{((a \wedge b) \vee (a \wedge c)), (b \Rightarrow (\neg b))\}$ and $f = ((b \vee c) \Leftrightarrow (b \Rightarrow c))$.

- Prove or disprove: $F \models f$ (0.5 Marks)
  Consider all interpretations:

a	b	c	$((a \wedge b) \vee (a \wedge c))$	$(b \Rightarrow (\neg b))$	Model F?	f
false	false	false	false	true	No	
false	false	true	false	true	No	
false	true	false	false	false	No	
false	true	true	false	false	No	
true	false	false	false	true	No	
true	false	true	true	true	Yes	true
true	true	false	true	false	No	
true	true	true	true	false	No	

Since any model of F also is a model of f, f is a logical consequence of F.

Two formulas f and f' are equivalent, if $I(f) = I(f')$ for all interpretations I. Show: For any formula f there is an equivalent formula f which does not contain $\Rightarrow$. Hint: Use structural induction. (1 Mark)

Proof: By structural induction

- Base case: $f \in V$ is a proposition. Then $f' = f$ is equivalent to f and does not contain $\Rightarrow$. Hence the claim holds for atomic formulas.
- Induction assumption: For $f_1, f_2 \in wfpf$ exist $f_1', f_2' \in wfpf$, such that f_1 is equivalent to f_1', f_2 is equivalent to f_2', and f_1', f_2' do not contain $\Rightarrow$.
- Induction step:
  Case 1: $f = (\neg f_1)$. Then $f' = (\neg f_1')$ does not contain $\Rightarrow$ (by Induction assumption and construction). We claim f' is equivalent to f. Consider an arbitrary interpretation I. By assumption, $I(f_1) = I(f_1')$ (as f_1, f_1' are equivalent). But then $I(f) = I((\neg f_1)) = I((\neg f_1')) = I(f')$. Thus, f and f' are indeed equivalent.
  Case 2: $f = (f_1 * f_2), * \in \{\wedge, \vee, \Leftrightarrow\}$. Then $f' = (f_1' * f_2')$ is $\Rightarrow$-free by assumption and construction. We claim f' is equivalent to f. Consider an arbitrary interpretation I. By assumption, $I(f_1) = I(f_1')$ and $I(f_2) = I(f_2')$ (as f_1, f_1' and f_2, f_2' are equivalent). But then: $I(f) = I((f_1 * f_2)) = I((f_1' * f_2')) = I(f')$ as above. Thus, f and f' are indeed equivalent.

Case 3: $f = (f_1 \Rightarrow f_2)$. Then $f' = ((\neg f_1') \vee f_2')$ is $\Rightarrow$-free by assumption and construction. We claim f' is equivalent to f. Consider an arbitrary interpretation I.

Case 3.1: $I(f_1) = \text{true}$. Then $I(f) = I(f_2)$. By induction assumption, $I(f_1') = \text{true}$, and hence $I((\neg f_1') = \text{false}$. Thus, $I(f') = I(f_2')$. Hence (using the induction assumption): $I(f) = I(f_2) = I(f_2') = I(f')$.

Case 3.2: $I(f_1) = \text{false}$. Then $I(f) = \text{true}$. By assumption, $I(f_1') = \text{false}$ and thus $I((\neg f_1') = \text{true}$. But then $I(((\neg f_1') \vee f_2')) = \text{true} = I(f)$.

Since $I(f) = I(f')$ for all interpretations I, case 3 is holds, hence the induction step holds, and thus the original claim holds. $\square$

Well-founded Orderings

Which of the following relations is well-founded? If the answer is no, give an example of an infinitely descending chain. Note that $>$ is used for the normal ordering on $\mathbb{N}$ and $\mathbb{Z}$. (0.5 Marks each)

1. $\succ_0$: $(\mathbb{N} \times \mathbb{N}) \leftrightarrow (\mathbb{N} \times \mathbb{N})$ defined by $(a, b) \succ_0 (a', b')$ if and only if $(a + b) > (a' + b')$

2. $\succ_1$: $(\mathbb{Z} \times \mathbb{Z}) \leftrightarrow (\mathbb{Z} \times \mathbb{Z})$ defined by $(a, b) \succ_1 (a', b')$ if and only if $(a + b) > (a' + b')$

3. $\succ_2$: $(\mathbb{Z} \times \mathbb{N}) \leftrightarrow (\mathbb{Z} \times \mathbb{N})$ defined by $(a, b) \succ_2 (a', b')$ if and only if $a = a'$ and $b > b'$.

4. $\succ_3$: $(\mathbb{N} \times \mathbb{N}) \leftrightarrow (\mathbb{N} \times \mathbb{N})$ defined by $(a, b) \succ_3 (a', b')$ if and only if $a > a'$ or $a = a'$ and $b > b'$.

Termination

Consider the following Python function. Implicitly assume all variables are integers.

```
def countdown(x):
    y = 0
    while x > 1:
        if x % 2 == 0: # % is division rest (modulus)
            x = x/2
        else:
            x=x-1
        y = y+1
    return y
```

What is the value of (0.25 Marks each)

- `countdown(3)`
- `countdown(6)`
- `countdown(7)`
- `countdown(8)`

Prove that `countdown()` terminates with respect to precondition $x \in \mathbb{Z}$. Hint: Consider the cases $x > 0$ and $x \leq 0$ separately. (1 Mark)

Program Correctness

Consider the following Python function. Implicitly assume all variables are integers.

```
def power(x,y):
    assert x>0 and y>0 # (Pre)
    res = 1
    i = 0
    while i<y:
        assert res == x**i # (Inv),  ** is exponentiation!
        res = res * x
        i = i+1
    assert res == x**y # (Post)
    return res
```

Show the following:

If the precondition (Pre) is true, the line labeled (Inv) is indeed an invariant. (0.5 Marks)

Program Correctness (contd.)

```
def power(x,y):
    assert x>0 and y>0 # (Pre)
    res = 1
    i = 0
    while i<y:
        assert res == x**i # (Inv),  ** is exponentiation!
        res = res * x
        i = i+1
    assert res == x**y # (Post)
    return res
```

Show the following:

The function is partially correct with respect to the precondition (Pre) and prostcondition (Post), i.e. it correctly computes the value x^y for positive integers x and y (1.5 Marks)

CS63Z
Formal Methods in Software Engineering

States, Transitions, and First Order Logic

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Submitting the Programming Assignment

Please email the programming assignment to me, incuding all necessary files as plain text attachments.

- My most reliable and fast email address is `schulz@in.tum.de`
- (`schulz@eprover.org`) also works

I will acknowledge receipt as soon as possible

- If you submit before Thursday afternoon, by Thursday evening
- If you submit later, I'll acknowledge possibly on Sunday, but more likely on Monday
- I'll live by Central European Summer Time, i.e. I'll be 7 hours behind you...
- If you do not receive confirmation by Tuesday, please mail me again

Contacting me between Thursday and Sunday is likely futile

- I'll be in various planes and airports

Formalizing It

We will now introduce definitions that allow us to more precisely reason about programs

- States capture the values of variables
- Program configurations describe the full state of execution
- The Semantics of a program is a partial function on states
- Assertions are predicates (i.e. unary relations) on the state space
- Partial and total correctness are properties of a program with respect to pre- and postconditions

Later on, we will need first order predicate logic to describe

- Semantics (as operational and axiomatic semantics)
- Assertions (as formulas)

The State Space

Let D be the set of all possible values for variables (called the universe)

Let P be a program and let $Var(P)$ be the set of variables in P

Then a state of the program is a (partial) function $\sigma : Var(P) \nrightarrow D$ assigning values to variables

- We often write $\sigma = \{x \mapsto 10, y \mapsto 11 \ldots\}$
- We write $\sigma(z) \uparrow$ if $\sigma(z)$ is undefined, i.e. if there is no $d \in D$ such that $(z, d) \in \sigma$.

The set of all possible states $\Sigma = \{\sigma \mid \sigma \text{ is a state}\}$ is called the state space

Remarks:

- In practice, $D = D_1 \cup D_2 \ldots$, where the D_i represent different types
- A state describes the values of the variables, not the complete state of the progam (we still need to know which line is next to be executed)
- Σ depends on U and $Var(P)$. We usually assume an approriate U and P

The Meaning of a Program

Definition: Let P be a program, D be a universe, and Σ the state space of P. The semantics of P is a partial function $M(P) : \Sigma \nrightarrow \Sigma$ defined by:

$$M(P)(\sigma) = \begin{cases} \sigma' & \text{if } P \text{ terminates on input } \sigma \text{ and the final state is } \sigma' \\ \uparrow & \text{otherwise} \end{cases}$$

Notice:

- $\sigma : V \nrightarrow D$ (a state) is a function mapping variables to values (elements of the universe)
- $M(P) : \Sigma \nrightarrow \Sigma$ (the semantics of P)is an function mapping states to states
- $M : \{P \mid P \text{ is a program}\} \rightarrow (\Sigma \nrightarrow \Sigma)$ is a function mapping programs to meanings, so it is. . .
  - a total function mapping programs (texts) to partial functions (the semantics), where. . .
  - the semantics are another partial function mapping partial functions (states) to partial functions (states) and . . .
  - and each state is a partial function mapping elements of the unviverse to elements of the universe

Examples

```
def factorial(x):
    fac = 1
    sum = 0
    while x > 0:
        fac = fac*x
        sum = sum+x
        x = x-1
    return (fac, sum)
```

Examples

```
def factorial(x):
    fac = 1
    sum = 0
    while x > 0:
        fac = fac*x
        sum = sum+x
        x = x-1
    return (fac, sum)
```

Assume the funtion has been called as `factorial(3)`

- The initial state is $\sigma_0 = \{x \mapsto 3\}$
- The state at the first execution of while is $\sigma_2 = \{x \mapsto 3, fac \mapsto 1, sum \mapsto 0\}$
- Next time, the state at the while is $\sigma_6 = \{x \mapsto 2, fac \mapsto 3, sum \mapsto 3\}$
- The final state is $\sigma_{10} = \{x \mapsto 0, fac \mapsto 6, sum \mapsto 6\}$
- Note: Initialization is incomplete
- Note: Return just projects parts of the final state

Partial and Total Correctness

Let φ and ψ be assertions over states $\sigma \in \Sigma$

- If φ is true before P executes (i.e. for the input state), and ψ is true after P executes (for the final state), we say that P is partially correct with respect to precondition φ and postcondition ψ. We write:

$$\{\varphi\}P\{\psi\}$$

- If P terminates whenever φ holds before execution, and ψ holds when P stops, then we say that P is totally correct. We write

$$[\varphi]P[\psi]$$

First-Order Logic

We have, so far, used semi-formal arguments to reason about programs

To get the full benefit, we need fully formalized arguments

- Reduces risk of error
- Allows automatation

Propositional logic allows us to reason about propositions (fixed assumptions)

Reasoning about program requires us to talk about functions, relations, and elements

- Propositional logic is insufficient!

First-order logic extends propositional logic to deal with elements, relations and functions!

First-Order Logic: Introduction

First-order formula contain:

- First-order Terms, build from function symbols and (individual) variables, representing elements of the universe
- First-Order Atoms, build from a predicate symbol and terms, representing relations between elements
- Quantors ($\forall, \exists$), describing how to handle formulas with variables
- Boolean Operators, as in propositional logic, for combining formulae

Example: $\forall\, X \bullet (human(X) \Rightarrow (human(father(X)) \wedge mortal(X)))$

English: For all things X: If X is human, than the father of X is human, and X is mortal.

Colloquial: Humans have human fathers and eventually die.

First-Order Logic: Example

$$\forall\, X \bullet (human(X) \Rightarrow (human(father(X)) \wedge mortal(X)))$$

X is a individual variable that can range over the universe D of all values

$father$ is a function symbol (not a function!) (with the intended meaning of giving the father of an element)

$human$ is a predicate symbols (intended to represent a predicate evaluating to true if and only if its argument is human)

$\wedge$ and $\Rightarrow$ are Boolean operators with the same meaning as in propositional logic

$\forall$ is the universal quantor, making the formula true if its second argument is true for all elements of the universe

First-Order Logic: Syntax (1)

Definition: A signature is tuple $sig = (P, F, V, ar)$ where:

- P is a finite or enumerable set of predicate symbols
- F is a finite or enumerable set of function symbols
- V is an enumerable set of (individual) variables
- $ar : F \cup V \rightarrow \mathbb{N}$ is a function assigning a arity to each function and predicate symbol. If $ar(f) = n$ for some $f \in F \cup V$, we write $f\ |_n$ as shorthand

Let $sig = (P, F, V, ar)$ be a signature. The set $Term(F, V)$ is defined as follows:

- $X \in V$ is a term (all variables are terms)
- If $f\ |_n \in F$ and $t_1, \ldots, t_n \in Term(F, V)$, then $f(t_1, \ldots, t_n) \in Term(F, V)$
  - Note the special case of $f\ |_0$. In that case, we call f a constant and normally omit the parentheses.
- Nothing else is a first-order term.

First-Order Logic: Syntax (2)

Let $sig = (P, F, V, ar)$ be a signature. Assume $p \mid_n \in P$ and $t_1, \ldots, t_n \in Term(F, V)$. Then $p(t_1, \ldots, t_n)$ is an atom (over sig)

- Note: The special case $p \mid_0 \in P$ creates a propositional variable or just proposition. We again omit the parentheses. Yes, this is the same thing as in propositional logic!
- The set of all atoms is called $Atom(P, F, V)$.

Let $o = \{\neg\ , \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ be the usual set of logical operators. The set $Form(P, F, V)$ of well-formed first-order formulae (over sig) is defined as follows:

- $Atom(P, F, V) \subset Form(P, F, V)$ (Atoms are formulas)
- If $f_1, f_2 \in Form(P, F, V), X \in V$, then:
  - $(\neg\, f_1) \in Form(P, F, V)$
  - $(f_1 * f2) \in Form(P, F, V)$ for $* \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
  - $(QX \bullet f_1) \in Form(P, F, V)$ for $Q \in \{\forall, \exists\}$
- Nothing else is a formula

Remarks

Note that a first-order formula is a word over the alphabet $P \cup F \cup V \cup o \cup \{(,), \forall, \exists, \bullet\}$

The definiton of formula so far is a purely syntactic one!

We also call the set of all formulas over sig the (first order) language of sig or $L(sig)$

Example Revisited

$$f = \forall\, X \bullet (human(X) \Rightarrow (human(father(X)) \wedge mortal(X)))$$

What is a possible signature for f?

- $P =?$
- $F =?$
- $V =?$
- $ar =?$

What are the terms?

-

What are the atoms?

-

Example Revisited

$$f = \forall\, X \bullet (human(X) \Rightarrow (human(father(X)) \wedge mortal(X)))$$

What is a signature for f?

- $P = \{human\}$
- $F = \{father\}$
- $V = \{X, \ldots\}$
- $ar = \{human \mapsto 1, father \mapsto 1, mortal \mapsto 1\}$

What are the terms in f?

- $X, father(X)$

What are the atoms in f?

- $human(X), human(father(X)), mortal(X)$

Further Definitions

The set of ground terms is the set $Term(F, \varnothing)$ of variable-free terms

The set of ground atoms is the set $Atom(P, F, \varnothing)$ of variable-free atoms

The set of ground formulas is the set $Form(P, F, \varnothing)$ of variable-free formulas (which necessarily also does not contain quantors)

Lemma: $Form(P, \varnothing, \varnothing)$ is exactly the set of well-formed propositional formulae with propositions from P.

- Since there are no terms, only atoms with 0 arguments can be build
- Since there are no variables, quantors cannot occur
- What remains are exactly the well-formed propositional formulae!

Bound and Free Variable

Assume F is of the form $(QX \bullet F')$ for $Q \in \{\forall, \exists\}$

- F' is the scope of the quantor Q
- All occurences of X in F are bound
- A variable that is not bound by any quantor is called free

A formula without free variables is called a sentence or a closed formula

- Very many applications only deal with sentences
- For program verification, we need free variables

More Examples

Let $F = \{a\ |_0, f\ |_1\}$. What is the set $Term(F, \varnothing)$?

-

Let $F = ((\exists\, X \bullet p(X, Y)) \wedge p(Y, X))$

- What are the free variables of F?
- What are the bound variables of F?

Some more examples of first-order formulae:

- $((\forall\, X \bullet p(X)) \Rightarrow (\exists\, X \bullet p(x)))$
- $((\exists\, X \bullet p(X)) \Rightarrow (\forall\, X \bullet p(x)))$
- $(\forall\, X \bullet (\exists\, Y \bullet (p(X) \Rightarrow p(Y))))$

More Examples

Let $F = \{a\ |_0, f\ |_1\}$. What is the set $Term(F, \varnothing)$?

- $Term(F, \varnothing) = \{a, f(a), f(f(a)), \ldots\}$

Let $F = ((\exists\, X \bullet p(X, Y)) \wedge p(Y, X))$

- What are the free variables of F?
  X and Y
- What are the bound variables of F?
  Just X. Note X is both free and bound in F!

Some more (interesting) examples of first-order formulae:

- $((\forall\, X \bullet p(X)) \Rightarrow (\exists\, X \bullet p(X)))$
- $((\exists\, X \bullet p(X)) \Rightarrow (\forall\, X \bullet p(X)))$
- $(\forall\, X \bullet (\exists\, Y \bullet (p(X) \Rightarrow p(Y))))$

Remarks

We have no particular constraints on the syntactic form of function symbols and predicate symbols!

- Often, we will use $+, -, *, \ldots$ as function symbols
- Similarly, $=, >, \geq$ will occur as predicate symbols
- If convenient, we will use these symbols as infix

We will often abbreviate a sequence of quantors:
$\forall\, X, Y, Z \bullet f$ is short for $\forall\, X \bullet (\forall\, Y \bullet (\forall\, Z \bullet f))$

As with propositional logic, we will sometimes omit redundant parenthesss

- Use the conventions from page 85
- The quantors have highest precedence
- Example: $\forall\, X\, \exists\, Y \bullet X > Y \vee X < Y$
- Formally: $((\forall\, X \bullet (\exists\, Y \bullet > (X, Y))) \vee (< (X, Y)))$

Very often, we will assume two reserved propositions T and F (always interpreted as true and false)

Meaning of Formulas

As for propositional logic, formulas can evaluate to true and false

However, we need an expanded definition of interpretation

- We need to decide on a universe of possible values
- Function symbols are mapped to functions
- Predicate symbols are mapped to relations
- Free variables need to be assigned a value from the universe

Example: Evaluate

$$(\forall\, X \bullet (X \geq Y))$$

- Is $\geq$ really "greater than or equal"?
- If yes: Universe $\mathbb{N}$?
- If yes: Universe $\mathbb{Z}$?

First-Order Interpretations

Definition: A first-order interpretation is a tuple $I = (D, I_P, I_F, I_V)$ where:

- $D \neq \varnothing$ is the universe of possible values
- $I_P : P \to \mathbb{P}(D^*)$ maps each n-ary predicate symbol to an n-ary relation over D
- $I_F : F \to (D^* \to D)$ maps each n-ary function symbol to a function from D^n to D
- $I_V : V \to D$ maps each variable to a value from D

Example

Assume $sig = (\{>|_2, =|_2\}, \{+|_2, *|_2\}, \{X, Y, \ldots,\}, ar)$ is a signature

We define I as follows:

- $D = \mathbb{N}$
- $I_P(>) =>: \mathbb{N} \leftrightarrow \mathbb{N}$ (the predicate symbol $>$ is interpreted by the relation $>$ on $\mathbb{N}$
- $I_P(=) = \{(i, i) \mid i \in \mathbb{N}\}$ ($=$ is interpreted as the identity relation)
- $I_F(+) = + : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ (the function symbol $+$ is interpreted by the function $+$ that maps pairs of numbers to numbers
- $I_F(*) = * : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}$ ($*$ is interpreted as multiplication)
- $I_V(X) = 1, I_V(Y) = 3$
- $I_V(Z) = 0$ for all other $Z \in V$

Then $I = (D, I_P, I_F, I_V)$ is an interpretation of $L(sig)$

But what does it mean for formulas?

First Order Semantics (1)

An interpretation so far only assigns meaning to the elements of sig

We will now extend this to composite terms, atoms, and formulas

Assume $sig = (P, F, V, ar)$ and $I = (D, I_P, I_F, I_V)$ as before

Then terms are interpreted as follows:

- $I(X) = I_V(X)$ for all $X \in V$
- $I(f(t_1, \ldots, t_2)) = I_F(f)(I(t_1), \ldots, I(t_n))$
  - Note: $I_F(f)$ is a function of the proper type!
  - As a consequence, each term is mapped to an element of the universe!

Atoms are interpreted as follows:

$$I(p(t_1, \ldots, t_n)) = \begin{cases} \text{true} & \text{if } (I(t_1), \ldots, I(t_n)) \in I_P(p) \\ \text{false} & \text{otherwise} \end{cases}$$

First Order Semantics (2)

We will now extend this from atoms to formulas. To handle quantors, we need the following defintion:

Definition: Let $I_V : V \to D$ be a function assigning values to variables and assume $X \in V$ and $d \in D$. Then we define $I_V^{(X,d)} : V \to D$ as follows:

$$I_V^{(X,d)}(Y) = \begin{cases} d & \text{if } X = Y \\ I_V(Y) & \text{otherwise} \end{cases}$$

In other words, $I_V^{(X,d)}$ maps X to $d \in D$, and maps all other variables like I_V.

Assume $I = (D, I_P, I_F, I_V)$. Then we define $I^{(X,d)} = (D, I_P, I_F, I_V^{(X,d)})$

First Order Semantics (3)

- Assume $sig = (P, F, V, ar)$ and $I = (D, I_P, I_F, I_V)$. We know how to evaluate terms and atoms.
- Assume $f_1, f_2 \in Form(P, F, V)$ and $X \in V$

$$I((\neg f_1)) = \begin{cases} \text{true} & \text{if } I(f_1) = \text{false} \\ \text{false} & \text{otherwise} \end{cases}$$

- Similarly, $I(f_1 \wedge f_2), I(f_1 \vee f_2), I(f_1 \Rightarrow f_2), I(f_1 \Leftrightarrow f_2)$ are defined as for the propositional case
- $$I((\forall X \bullet f_1)) = \begin{cases} \text{true} & \text{if } I^{(X,d)}(f_1) = \text{true for all } d \in D \\ \text{false} & \text{otherwise} \end{cases}$$
- $$I((\exists X \bullet f_1)) = \begin{cases} \text{true} & \text{if } I^{(X,d)}(f_1) = \text{true for at least one } d \in D \\ \text{false} & \text{otherwise} \end{cases}$$

This recursively defines the value of any formula!

Examples

Consider $sig = (\{>|_2, =|_2\}, \{+\ |_2, *\ |_2, 0\ |_0, 1\ |_0\}, \{X, Y, Z \ldots\}, ar)$

Consider I with:

- $D = \mathbb{Z}$
- $I(>) = \ >: \mathbb{Z} \leftrightarrow \mathbb{Z}$ (normal $>$ on $\mathbb{N}$)
- $I(=) = \mathrm{id}$ (Identity)
- $I(0) = 0$, $I(1) = 1$
- $I(+)(x, y) = x + y$
- $I(*)(x, y) = x * y$
- $I(X) = 10$

Consider the following terms and formulas:

- $I(((0 + 1) + 1) = 2$
- $I((((0 + 1) + 1) > X)) = \mathrm{false}$
- $I((((0 + 1) + 1) > 0)) = \mathrm{true}$
- $I(((X + 1) + 1) = 12$
- $(\forall\, X \bullet (((X > 0) \vee (X = 0)) \vee (0 > X))) = \mathrm{true}$

Interpretations and Models

Assume:

- sig is a signature as usual
- $f \in Form(P, F, V)$ and $F \subseteq Form(P, F, V)$
- I is an interpretation of f

As in propositional logic:

- I is called a model of f, if $I(f) = \mathrm{true}$. We write: $\models_I f$
- I is called a model of F, if $I(f) = \mathrm{true}$ for all $f \in F$. We write: $\models_I F$

If $I(f) = \mathrm{true}$ for all interpretations I, then f is called a tautology

- We write: $\models f$
- The set of all tautologies (for a signature) is called $Taut(P, F, V)$ (or just $Taut$ if we are lazy)

If $I(f) = \mathrm{false}$ for all interpretations I, then f is called unsatisfiable

Trick Questions

Is there a model for $Form(P, F, V)$?

Is there a model for $Taut(P, F, V)$?

How many interpretations are there for $(\forall\, X \bullet p(X))$?

Trick Questions

Is there a model for $Form(P, F, V)$? No, since e.g. $p, (\neg p) \in Form(P, F, V)$

Is there a model for $Taut(P, F, V)$? Yes, any interpretation is a model

How many interpretation are there for $(\forall X \bullet p(X))$?

- There are infinitely many interpretation for (most) first order languages
- As a consequence, we cannot detect tautologies by enumerating all interpretations
- As a (more complex) consequence: We cannot in general decide if a formula is a tautology (or unsatisfiable) at all!

First-Order Logic is Undecidable!

- We can still prove any valid theorem...
- But we cannot always refute wrong ones!

Examples

Consider $sig = (\{>|_2, =|_2\}, \{+\,|_2, *\,|_2, 0\,|_0, 1\,|_0\}, \{X, Y, Z \ldots\}, ar)$

1. $(\forall\, X, Y \bullet (X + Y = Y + X)))$
2. $(\forall\, X, Y, Z \bullet (X * (Y + Z)) = (X * Y) + (X * Z))$
3. $(\forall\, X \bullet (\exists\, Y \bullet X + Y = 0))$
4. $(\forall\, X \bullet (X > 0 \vee (\exists\, Y \bullet X + Y = 0)))$

Consider I_1 with:

- $D = \{\top, \bot\}$
- $I(>) = \{(\top, \bot)\}$
- $I(=) = \mathrm{id}$
- $I(0) = \bot$
- $I(1) = \top$
- $I(+)$ given by

X \ Y	$\bot$	$\top$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

$I(*)$ given by

X \ Y	$\bot$	$\top$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$

Examples

Consider $sig = (\{>|_2, =|_2\}, \{+\,|_2, *\,|_2, 0\,|_0, 1\,|_0\}, \{X, Y, Z \ldots\}, ar)$

1. $(\forall\, X, Y \bullet (X + Y = Y + X)))$
2. $(\forall\, X, Y, Z \bullet (X * (Y + Z)) = (X * Y) + (X * Z))$
3. $(\forall\, X \bullet (\exists\, Y \bullet X + Y = 0))$
4. $(\forall\, X \bullet (X > 0 \lor (\exists\, Y \bullet X + Y = 0)))$

Consider I_2 with:

- $D = \mathbb{N}$
- $I(>) = \ >: \mathbb{N} \leftrightarrow \mathbb{N}$ (normal $>$ on $\mathbb{N}$)
- $I(=) = \mathrm{id}$ (Identity)
- $I(0) = 0$
- $I(1) = 1$
- $I(+)(x, y) = x * y$
- $I(*)(x, y) = x + y$

Examples

Consider $sig = (\{>|_2, =|_2\}, \{+\,|_2, *\,|_2, 0\,|_0, 1\,|_0\}, \{X, Y, Z \ldots\}, ar)$

1. $(\forall\, X, Y \bullet (X + Y = Y + X)))$
2. $(\forall\, X, Y, Z \bullet (X * (Y + Z)) = (X * Y) + (X * Z))$
3. $(\forall\, X \bullet (\exists\, Y \bullet X + Y = 0))$
4. $(\forall\, X \bullet (X > 0 \vee (\exists\, Y \bullet X + Y = 0)))$

Consider I_3 with:

- $D = \mathbb{Z}$
- $I(>) = \ >: \mathbb{Z} \leftrightarrow \mathbb{Z}$ (normal $>$ on $\mathbb{N}$)
- $I(=) = \mathrm{id}$ (Identity)
- $I(0) = 0$
- $I(1) = 1$
- $I(+)(x, y) = x + y$
- $I(*)(x, y) = x * y$

Examples

Consider $sig = (\{>|_2, =|_2\}, \{+\,|_2, *\,|_2, 0\,|_0, 1\,|_0\}, \{X, Y, Z \ldots\}, ar)$

1. $(\forall X, Y \bullet (X + Y = Y + X)))$
2. $(\forall X, Y, Z \bullet (X * (Y + Z)) = (X * Y) + (X * Z))$
3. $(\forall X \bullet (\exists Y \bullet X + Y = 0))$
4. $(\forall X \bullet (X > 0 \vee (\exists Y \bullet X + Y = 0)))$

Consider I_4 with:

- $D = \{0, 1, 2, 3\}$
- $I(>) = \ >: \mathbb{N} \leftrightarrow \mathbb{N} \cap (D \times D)$ (normal $>$)
- $I(=) = \mathrm{id}$ (Identity)
- $I(0) = 0$
- $I(1) = 1$
- $I(+)(x, y)$:

X \ Y	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$I(*)(x, y)$:

X \ Y	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Remarks

The value of a formula f is fully determined by

- Deciding on a universe
- Interpreting the function symbols
- Interpreting the predicate symbols
- Assigning values to the free variables in f

In particular, for closed formulas (sentences), we don't need I_V at all

- We can often ignore the I_V component of I

Logical Consequence

Assume $F \subseteq Form(P, F, V)$ and $f \in Form(P, F, V)$

- f is called a logical consequence of F, if every for every model I of F, $I(f) = \text{true}$.w

Logical Theories

A theory is a set of formula that is closed under consequences

Formal Definition: Let $T \subseteq Form(P, F, V)$ be a set of first-order formulas. T is called a theory, if $T \models f$ implies $f \in T$.

Examples:

- $Taut(P, F, V)$ is a theory
- $Form(P, F, V)$ is a theory
- $\varnothing$ is not a theory, as $\varnothing \models f$ for all $f \in Taut(P, F, V)$
- By the same reasoning: If T is a theory, then $Taut(P, F, V) \subseteq T$ (the tautologies are part of all theories)

If F is a set of axioms, then $Th(F) = \{f \mid F \models f\}$ is a theory

CS63Z
Formal Methods in Software Engineering

Introduction to Z

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Introduction

We have discussed how to

- Figure out the meaning of programs
- Pre- and postconditions
- Invariants
- How to show that programs behave correctly with respect to their pre- and postconditions

Now: How do we arrive at those conditions?

- Formal specification of systems
- Describing desired properties by . . .
  - State space
  - Pre-and postconditions
- Refining specifications into code
  - Choosing concrete implementations of abstract properties
  - (If necessary:) Show agreement of implementation with specification

The Z Notation

Systematic way of formally describing systems using Schemas

Descriptions are abstract and unambiguous

- Uses Set theory, types, predicate logic

Organized behaviour by:

- Describing possible states
- Describing initial states
- Describing state changes
- Describing state queries

Developed at Oxford University by Jean-Raymond Abrial & Collegues

Some tool support for Z exists

- Type checking (make sure specifications are well-typed)
- Typesetting (make sure specifications are readable ;-)

Example Application

Standard Example from the Z reference manual: The Birthday Book

- Stores a list of names with associated birthdays
- Operations for:
  - Add new people
  - Query for a given persons birthday
  - Query for all birthdays on a given date

Types in Z

Types allow us to distinguish different classes of objects

In Z, a type is an expression refering to a set (the value of the type)

- For convenience, we usually say that a member of the value of a type is "of type X"

There are:

- Basic types (assumed to be known)
- Set types (values are sets of elements of a known type)
- Tuple type (values are tuples combining elements of different known types)
- Schema types (values are bindings associating names with values of a certain types)

Birthday Book Basic Types

For the birthday book, we have to deal with two basic types:

- Names of people
- Birthday dates (day and month)

For the formal specification we can ignore properties of names and dates

- Need only generic properties of all objects
- This abstraction makes the specification more compact

To introduce the two types, we just specify them:

$$[NAME, DATE].$$

The Birthday Book State Space

The major purpose of the birthday book is to record a mapping from people to names

- This is properly modelled as a function $birthday : NAME \pfun DATE$

We also need to access the set of all known names frequently, so we want this set as an explicit entity

- Its value will be a set of names (i.e. an element of the powerset of $NAME$
- So we declare it as $known : \mathbb{P}\, NAME$

We encode the relationship between $birthday$ and $known$ as a invariant of the system:

- $\mathrm{dom}\, birthday = known$
- This has to hold in every legal state of the system!

In Z, we declare this as a Schema

Base Schema for the Birthday Book

$$
\begin{array}{|l}
\hline
BirthdayBook \\
known : \mathbb{P}\, NAME \\
birthday : NAME \pfun DATE \\
\hline
known = \mathrm{dom}\, birthday \\
\hline
\end{array}
$$

Schema with Explanations

$BirthdayBook$ — Gives the name of the schema

$known : \mathbb{P}\, NAME$ — Variable declarations go above the dividing line

$birthday : NAME \pfun DATE$

$known = \mathrm{dom}\, birthday$ — Conditions go beneath the line

Variable declarations introduce names

- The types form one type of constraint for the possible states
- A variable can only take values corresponding to its type

The condition part of the schema further constrains the possible states

- Note: This is not an assignment
- It is one example of a condition, generally specified in (interpreted) first order predicate logic

Decorations

Given a schema, Z allows us to decorate it

- Most common decoration: '
- A decorated schema has all the same components as the original, but all the names are decorated in the same way as the schema

Example: $BirthdayBook'$ is:

$$
\begin{array}{|l}
\hline
BirthdayBook' \\
known' : \mathbb{P}\, NAME \\
birthday' : NAME \pfun DATE \\
\hline
known' = \mathrm{dom}\, birthday' \\
\hline
\end{array}
$$

As before, the plain and '-ed versions are used to refer to the state before and after an operation

- Used in specifying operations on data types

Adding people to the Birthday Book

A modifying operation describes the relationship of two states, one before and one after the operation

- For the before state we use plain names
- For the after state, we use '-ed names
- The operation is described solely by specifying how the before and after states correllate!

Of course, many operations need input- and output data

- $name?$ is an input variable
- $name!$ is an output variable

We can now describe the operation that adds a person to the birthday book!

Modification: The Cumbersome Way

AddBirthday

$$known : \mathbb{P}\, NAME$$
$$known' : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$date? : DATE$$

$$known = \mathrm{dom}\, birthday$$
$$known' = \mathrm{dom}\, birthday'$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Modification: The Cumbersome Way

$AddBirthday$ Name of the Schema

$known : \mathbb{P}\, NAME$ Explicit inclusion of all declarations
$known' : \mathbb{P}\, NAME$
$birthday : NAME \pfun DATE$
$birthday' : NAME \pfun DATE$
$name? : NAME$
$date? : DATE$

$known = \mathrm{dom}\, birthday$ Invariants included by hand
$known' = \mathrm{dom}\, birthday'$
$name? \notin known$ Predondition
$birthday' = birthday \cup \{name? \mapsto date?\}$ Postcondition

Somewhat Better: Using Scheme Inclusion

AddBirthday

$BirthdayBook$ Inherit all of BirthdayBook

$BirthdayBook'$ Inherit all of BirthdayBook'

$name? : NAME$

$date? : DATE$

Notice that $known = \operatorname{dom} birthday$ and $known' = \operatorname{dom} birthday'$ are inherited!

$name? \notin known$

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

Advantages:

- We cannot make stupid mistakes
- Much less typing (especially for large schemas)

Disadvantage:

- It is just another Schema
- How do we know it modifies a $BirthdayBook$?

Elegant Version: The Δ-Convention

$AddBirthday$

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$
$birthday' = birthday \cup \{name? \mapsto date?\}$

$\Delta - SchemaName$ is implicit used to:

- Include $SchemaName$ and $SchemaName'$
- Signal to the reader that this modifies the state

This is only a writing convention!

Querying State Properties

Some operations do not change the state of a system, but just query it

Examples in our application:

- Given a person, find the birthday
- Given a date, find all persons

Specifiying these by hand is particulary cumbersome!

Finding a Birthday the hard way!

$FindBirthday$

$$known : \mathbb{P}\, NAME$$
$$known' : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$birthday' : NAME \pfun DATE$$
$$name? : NAME$$
$$date! : DATE$$

$$known = known'$$
$$birthday = birthday'$$
$$name? \in known$$
$$date! = birthday(name?)$$

Note: This is an operation, so:

- We have the before and after variables!
- We have to make clear that nothing changes!

Ξ-convention to the rescue...

FindBirthday

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

$\Xi SchemaName$ is used for operations that do not change the state

- Automatically includes $SchemaName$ and $SchemaName$
- Automatically creates the frame axioms (before $=$ after) for all variables in $SchemaName$

Notice: We have only specified $FindBirthday$ for known persons!

Finally: Never forget a Birthday Again

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$cards! = \{n \in known \mid birthday(n) = today?\}$$

Notice:

- No precondition (which means the precondition is true)
- No changes to the state

Ooops....how do we get a valid initial state?

InitBirthdayBook

$BirthdayBook$

$known = \varnothing$

We do not explicitely specify that $birthday$ is empty too

- Is implied by the invariant and the condition in InitBirthdayBook

Refining the Design into Code

To implement the Specification into working Code, we have to decide:

- How do we repesent the complete system?
  - A BirthdayBook is a stand-alone program with data
  - A BirthdayBook is represented by some variables, the operations by function
  - A BirthdayBook is an object
- How do we instantiate the abstract data types?
  - What is a $NAME$?
  - What is a $DATE$?
  - How do we represent the mapping $birthday$?
  - How do we represent $known$?
- Finally, how do we implement the operations?

Refining the Design into Code

To implement the Specification into working Code, we have to decide:

- How do we repesent the complete system?
  - A BirthdayBook is a stand-alone program with data
  - A BirthdayBook is represented by some variables, the operations by function
  - A BirthdayBook is an object
- How do we instantiate the abstract data types?
  - What is a $NAME$?
  - What is a $DATE$?
  - How do we represent the mapping $birthday$?
  - How do we represent $known$?
- Finally, how do we implement the operations?

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.–Fred Brooks (modernized)

Refining the Design into Code

To implement the Specification into working Code, we have to decide:

- How do we repesent the complete system?
  - A BirthdayBook is a stand-alone program with data
  - A BirthdayBook is represented by some variables, the operations by function
  - A BirthdayBook is an object
- How do we instantiate the abstract data types?
  - What is a $NAME$?
  - What is a $DATE$?
  - How do we represent the mapping $birthday$?
  - How do we represent $known$?
- Finally, how do we implement the operations?

Show me your flowcharts and conceal your table, and I shall continue to be mystified. Show me yourtables, and I won't usually need your flowcharts; it'll be obvious.–Fred Brooks, "The Mythical Man-Month"

Decisions

We just build a prototype!

- Names are strings (reasonable anyways)
- Dates are strings (no hassle with formats, etc...)

The mapping is realized as a Python dictionary

- Maps keys to values, i.e. ideally suited
- Dictionaries allow access to the set of their keys – we get $known$ for free!

The Creation and Initialization

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

$$known = \mathrm{dom}\, birthday$$

InitBirthdayBook

$$BirthdayBook$$

$$known = \varnothing$$

Python

```
class birthday_book(object):
    """
    A simple birthday book associating names with birthdays.
    """

    def __init__(self):
        """
        Initialize the birthday book as empty. Note that the
        specification has two names: 'known' (the set of people in the
        book) and 'birthday' (the mapping from people to birthdays).

        Our implementation automatically supports 'known' as the set
        of keys in the dictionary that maps people to birthdays, so
        the invariant is guaranteed. That is a big bonus of using a
        suitable language!

        We add the assertion just for documentation of the invariant!
        """
        self.birthday = {}
        assert self.birthday.keys() == []
```

A Note on Errors

We only specified behaviour for the non-error case

Our implementation will check conditions and raise an appropriate error if it is not verified.

Hence the following extra definitons:

```
class birthday_book(object):
    ...

    result_already_known = "Tried to add person already known"
    result_not_known = "Queried birthday of unknown person"
```

Adding People

AddBirthday

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

$name? \notin known$

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

Python

```
class birthday_book(object):
    ...

    def add_birthday(self, name, date):
        """
        Add a person (with birthday) to the book. Raise an error if
        the person is known.
        """
        if name in self.birthday.keys():
            raise birthday_book.result_already_known
        assert name not in self.birthday.keys() # Precondition
        self.birthday[name] = date
        # Post-condition is trivial by construction
```

Finding the Birthday

FindBirthday

$\Xi BirthdayBook$

$name? : NAME$

$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

```
class birthday_book(object):
    ...
    def find_birthday(self, name):
        """
        Return a persons birthday.
        """
        if name not in self.birthday.keys():
            raise birthday_book.result_unknown

        assert name in self.birthday.keys() # Precondition
        return self.birthday[name]
```

Getting Reminders

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$cards! = \{n \in known \mid birthday(n) = today?\}$$

```
class birthday_book(object):
    ...
    def remind(self, date):
        """
        Return a list of all people that have birthday on a given
        date.
        """
        # No precondition (besides the implicit invariant)
        cards = [ person for person in self.birthday.keys() \
                  if self.find_birthday(person)==date]
        # Postcondition true by construction - we just wrote down the
        # mathematical description. Behold the power  of Python!
        return cards
```

Infrastructure. . .

```
class birthday_book(object):
    ...
    def __str__(self):
        """
        Just for convenience: Make it printable.
        """
        res = "{"
        sep = ""
        for i in self.birthday.keys():
            res = res+sep+i+"->"+self.birthday[i]
            sep = " , "
        res = res + "}"
        return res
```

Testing

```
book = birthday_book()
print book

book.add_birthday("Stephan", "27-May")
print book
book.add_birthday("Janine", "9-Aug")
book.add_birthday("William II of Orange", "27-May")
book.add_birthday("Henry Kissinger", "27-May")
book.add_birthday("Jamie Olivier", "27-May")
book.add_birthday("Cornelius Vanderbilt", "27-May")
book.add_birthday("Albert Einstein", "14-Mar")
book.add_birthday("Hugo Black", "27-Feb")

print book
print "Stephan's Birthday:", book.find_birthday("Stephan")
print "Birthday on Dec. 24th:", book.remind("24-Dec")
print "Birthdays on May 27th:", book.remind("27-May")
```

Output

```
{}
{Stephan->27-May}
{William II of Orange->27-May , Hugo Black->27-Feb , Cornelius Vanderbilt->27-May ,
Stephan's Birthday: 27-May
Birthday on Dec. 24th: []
Birthdays on May 27th: ['William II of Orange', 'Cornelius Vanderbilt', 'Stephan', '
```

Exercises

Specify a simple shop inventory class in Z

- For each item, store number of items in stock
- Support operations for:
  - Restocking (adding items)
  - Delivering (removing items)
  - Getting the number of items in stock
  - Warning about all items that are in low supply (where "low" is a user-specified number)‘

Cast your specification into a Python class (or implement it in any other language of your choice)

Suggestions for Work

Most important topics have been touched upon in either the excercises or the test exam

Proofs of correctness and termination are central

I'm interested in your knowledge, but even more in your skills

Good Luck!

CS63Z
Formal Methods in Software Engineering

Recipes

Stephan Schulz

Mona Institute of Applied Science

University of the West Indies

schulz@eprover.org

http://www4.in.tum.de/~schulz/TEACHING/CS63Z-2005.html

Program Termination (1)

To show termination of a (non-recursive) program, show termination of each loop

- In many languages, only `while` loops are relevant (others terminate by construction)

To show termination of a loop:

- Find a mapping $W : \Sigma \nrightarrow A$ (from the state space to some set A) such that:
  - There is a well-founded partial ordering $\succ \subset (A \times A)$ (then we call $(A, \succ)$ a well-founded set)
  - If σ is the state before execution of the loop body, and σ' is the state after execution of the loop body, then $W(\sigma) \succ W(\sigma')$
- Since every execution of the loop body decreases $W(\sigma)$ with respect to $\succ$ and $\succ$ is terminating, the program eventually has to stop (otherwise $W(\sigma) \succ W(\sigma') \succ W(\sigma'') \succ \ldots$ would be an infinitely descending $\succ$-chain, violating the premise that $\succ$ is well-founded)

Program Termination (2)

Choosing W and A:

- Often, W is a simple projection of a variable or variable, i.e. $W(\sigma) = \sigma(x)$ or $W(\sigma) = (\sigma(x), \sigma(y), \sigma(z))$
- Often, W is a linear combination of two variables, e.g. $W(\sigma) = \sigma(x) + 2\sigma(y)$
- Often A is $\mathbb{N}$
- Often A is $\mathbb{Z}_{\geq k} = \{x \in \mathbb{Z} \mid x \geq k\}$ for some k. In that case, you often have to separately show:
  - $W(\sigma)$ decreases
  - $W(\sigma)$ can never decrease below k
- Often A is the carthesian product of $\mathbb{N}$ and $\mathbb{Z}_{\geq k}$ (often one set for one variable)

To show that W decreases, but may not decrease below a lower bound, you sometimes need to show an invariant and you may need to use the precondition

Often W and A are only mentioned implicitely, because they are very simple

Program Termination (trivial example)

Assume all variables are integer.

```
def terminates(x):
    while x > 0:
        x = x - 1
```

Claim: `terminates()` terminates

- Case 1: $x \leq 0$. Then the loop terminates directly
- Case 2: $x > 0$. Choose $(A, \succ) = (\mathbb{N}, >)$ and $W(\sigma) = \sigma(x)$. The value of x decreases every time the loop body is executed. x can never become smaller than 0 (i.e. leave $\mathbb{N}$), as a) the loop is only entered for $x > 0$ and b) x is decreased by 1 every time. Hence the program has to terminate.