~~Thesis Defense:~~ Incremental Validation of Formal Specifications

Paul Corwin

May 12, 2009

You can leave this slide out.

Committee Members

Name	**Role**
Dr. Gene Fisher	Advisor & Committee Chair
Dr. David Janzen	Committee Member
Dr. Clark Turner	Committee Member

You can leave this slide out.

Incremental Validation of Formal Specifications

- Presents a tool for the mechanical validation of formal software specifications
- Novel approach to incremental validation
- Form of “light-weight” model checking
- Part of FMSL: a formal modeling and specification language
- Small-scale aspects of a specification are validated; step-wise refinement
- Presents example that’s been used in software engineering courses for years
- Use of the tool led to discovery of a specification flaw

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demo~~nstration~~ of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demo~~nstration~~ of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Chapter 1: Introduction

- ~~Software engineering~~ is error-prone and expensive
- Early detection of errors is beneficial
- Thesis focuses on early error detection during formal specification
- ~~Tool-supported technique: FMSL~~
- ~~Added~~ executability
  - Standard functional evaluation
  - Boolean expressions containing universal and existential quantifiers, bounded and unbounded

The Problem

- How to validate a ~~formal~~ model-based speci~~fication~~?
- ~~Need for tools and methods that expose errors, misunderstood properties, improperly stated behaviors~~
- ~~FMSL~~ model behavior defined with Boolean preconditions and postconditions ~~on operations~~
- Evaluating quantifier ~~expressions~~ of particular interest

You can leave this slide out.

Thesis Aims

- Provide a means to validate formal specifications in a straightforward manner
- Demonstrate practicality in an instructional context

FMSL

- ~~Was a predicative specification language~~
- Had type checker
  - Performs syntactic and semantic analysis
  - Comparable to compilers of strongly typed programming languages
- Added executability
- Focused on operation validation

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Chapter 2: Background and Related Work

- Formal methods and related topics
- “Lightweight” formal methods
- Relevant specification languages and model checkers

Formal Methods: What Are They?

- Processes that exploit the power of mathematical notation and proofs
- Express system and software properties
- Help establish whether a specification satisfies certain properties or meets certain behavioral constraints

Formal Methods: Downsides

- Played “insignificant” role in software engineering in last 30 years [Glass]
- Rarely used, high barrier of entry [Heitmeyer]
- Few people understand what ~~formal methods~~ are or how to use them [Bowen et al.]
- High up front cost [Larsen et al.]

Formal Methods: Upsides

- Can be used during requirements development, specification, design, and implementation
- Practical means of showing absence of undesired behavior [Kuhn]
- Helps users better understand a system
  - ~~Formality prompts engineers to raise questions [Easterbrook et al.]~~
  - ~~Forces early, serious consideration of design issues [Jackson et al.]~~
  - ~~Abstraction can mask complexities [Agerholm et al.]~~
- Cost savings achieved [Larsen et al.]

"Heavyweight" Formal Methods: Model Checkers and Theorem Provers

- Model Checking
  - ~~Formal technique~~ based on state exploration; ~~purpose is to evaluate particular properties~~ [Chan]
  - ~~Often~~ involves search for a counter-example [Kuhn et al.]
- Theorem Provers
  - Assist ~~the user~~ in constructing proofs [Kuhn et al.]
  - May require expert users
  - Can lead to slower product design cycle [Kurshan]

~~Formal Methods:~~ Can Be Used on Individual System Parts

- Can be overkill for a system in its entirety [Bowen]
- Sometimes only ~~system~~ parts would benefit from ~~formal methods~~ [Agerholm et al.]
- Requires consideration to determine where formal methods use makes sense [Larsen et al.]
- Formalized parts can be re-used ~~in other projects~~ [Kuhn et al.]
- ~~Promotes code re-use [Jackson et al.]~~

Lightweight Formal Methods

- ~~Formal technique that~~ falls short of complete verification
- May not require that ~~the~~ user be trained ~~in advanced mathematics or sophisticated proof strategies [Heitmeyer]~~
- ~~May use formal notations~~
- Can be more practical and cost effective than "heavyweight" ~~formal methods [Jackson]~~

A Lightweight Technique: Simulation

- Animates a model by examining a small subspace of possible states ~~[Jackson et al.]~~
- May immediately expose mistakes
- Provides ability to test ~~functional requirements of interest~~ early

Test-Driven Development and Simulation

- ~~Calls for programmers to~~ write ~~low-level~~ functional tests before ~~beginning the~~ implementation
  - Improves productivity [Erdogmus]
  - Leads to less complex, ~~but~~ highly tested code [Janzen et al.]
- ~~Evaluating claims early on~~ improves understanding [Myers]
- ~~Whether generated manually or automatically~~, tests used for simulation ~~purposes~~ can be re-used against the implementation

Existing Model Checking Tools and Formal Specification Languages

- Verisoft
- Symbolic Model Verifier (SMV)
- Java Modeling Language (JML)
- Korat
- Object Constraint Language (OCL)
- Object-oriented Specification Language (OOSPEC)
- ASLAN
- Aslantest

See thesis paper for more details

Empirical Successes with Formal Methods

- BASE: A Trusted Gateway
- Miami University of Ohio: OOD Course
- NASA: Lightweight Formal Methods study

You can leave this slide out.

BASE: A Trusted Gateway

- Had formal methods and non-formal methods (control) groups develop a trusted gateway
- The formal methods group uncovered a hole in the requirements; the control team did not
- The formal methods group's software passed more tests than the control methods group's software
- The formal methods group's software performed fourteen times faster than the control group's software

You can leave this slide out.

Miami University of Ohio: OOD Course

- Had a formal methods group of students and a control group of students design and implement a common elevator project
- The formal methods teams followed more rigorous design processes and had relatively better designs
- 100% of the formal methods teams' implementations passed the tests; only 45.5% of the control group teams' implementations passed

You can leave this slide out.

NASA: Lightweight Formal Methods

- Observed effects of implementing light-weight formal methods in certain NASA programs
- Easterbrook et al. concluded that the application of formal methods early on added value
- Helped uncover ambiguities, inconsistencies, missing assumptions, logic errors, and more
- Observed that the development team was more receptive to fixing the problems, as they were discovered early

Presentation Outline

Chapter 3: Demonstration of Tool Capabilities

- Simple illustrative example
- Objects and operations

```
object PersonList
    components: Person*;
    description: (*
        A PersonList contains zero or more Person records.
    *);
end PersonList;

object Person
    components: firstName:Name and lastName:Name and age:Age;
    description: (*
        A Person has a first name, last name, and age.
    *);
end Person;

object Name = string;
object Age = integer;

operation Add
    inputs: p:Person, pl:PersonList;
    outputs: pl':PersonList;
    precondition: not (p in pl);
    postcondition: p in pl';
    description: (*
        Add a person to a list, if that person is not already in the
        list.
    *);
end Add;
```

How Does One Validate That It Is Correct?

- Static correctness validated by ~~FMSL~~ type checker
  - ~~Syntactic and semantic analysis~~
- Focus of ~~this~~ thesis is ~~determining the~~ dynamic correctness

You can leave this slide out.

Person Definitions with Add

```
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];

> Add(p, pl);          -- invoke Add operation
```

~~Person Definitions with Add~~

```
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];

> Add(p, pl);          -- invoke Add operation
```

- What value does the invocation ~~of~~ ~~`Add(p, pl)`~~ produce?
- `nil` ~~-- the empty value for any type of object~~
- Same value produced for any ~~inputs, since~~ ~~`Add` is defined only with a precondition and postcondition~~

Evaluate Add’s Precondition and Postcondition

```
operation Add
    inputs: p:Person, pl:PersonList;
    outputs: pl':PersonList;
    precondition: not (p in pl);
    postcondition: p in pl';
    description: (*
        Add a person to a list, if that person is not already in
        the list.
    *);
end Add;
```

Evaluate Add’s Precondition and Postcondition

```
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];
```

```
    precondition: not (p in pl);
    postcondition: p in pl';
```

```
> p in pl;                          -- should be false
> not (p in pl);                    -- should be true
> not (p in pl');                   -- should be false

> p in pl’;                         -- should be true
```

Validation Invocation

- Given inputs `p` and `p1`, expected output `p1'`, what are the values of the `Add` precondition and postcondition?

```
> Add(p, p1) ?-> p1';
```

Validation Invocation

- Given inputs `p` and `p1`, expected output `p1'`, what are the values of the `Add` precondition and postcondition?

```
> Add(p, p1) ?-> p1';
```

```
{ true, true }
```

Validation Invocation: Counter Example

```
> Add(p, p1) ?-> p1;
```

Validation Invocation: Counter Example

```
> Add(p, p1) ?-> p1;
```

```
{ true, false }
```

Expression Evaluation in FMSL

- Entails invoking an operator or operation and returning the calculated result
- Collection of built-in Boolean, arithmetic, tuple, and list expressions

Boolean Expression Examples

```
(*
 * Declare short value names for true and false.
 *)
val t:boolean = true;
val f:boolean = false;

(*
 * Boolean operator examples
 *)
> not t;                              -- evaluates t false
> t and f;                            -- evaluates t false
> t or f;                             -- evaluates to true
> t xor f;                            -- evaluates to true
> t => f;                             -- evaluates to false
> t <=> f;                            -- evaluates to true
```

Arithmetic Expression Example

```
 (*
  * Declare and assign values to x, y
  *)
val x:real = 3.141592654;
val y:real = 2.718281828;

 (*
  * Evaluate x divided by y and output the result
  *)
> x / y;
```

```
Output:

1.15573
```

Quantifier Evaluation

- Quantifiers: Boolean-valued expressions that evaluate a quantified sub-expression multiple times
- Universal (`forall`) and existential (`exists`) forms of quantification
- Bounded and unbounded quantifiers supported by FMSL

Universal Quantification

- Has the general form:

```
forall (x:t) predicate
```

- Read as "for all values *x* of type *t*, *predicate* is true"
- Other extended forms:

```
forall (x:t | p1) p2
forall (x in l) p
```

Existential Quantification

- Has the general form:

```
exists (x:t) predicate
```

- Read as "there exists a value *x* of type *t* such that *predicate* is true"
- Other extended forms:

```
exists (x:t | p1) p2
exists (x in l) p
```

Bounded Quantifier

```
 (*
  * Declare an IntList object type and an IntList value
  *)
obj IntList = integer*;
val list:IntList = [ 1, 1, 2, 3, 5 ];

 (*
  * Evaluate: all the integer elements within list are positive.
  *)
> "Expected: true";
> forall (i in list) i > 0;
```

Unbounded Quantifier

```
object Person
    components: firstName:Name and lastName:Name and age:Age;
    description: (*
        A Person has a first name, last name, and age.
    *);
end Person;
```

```
(*
 * Create values p1 and p2, which puts them in the Person value
 * Universe.
 *)
val p1:Person = {"Alan", "Turing", 97};
val p2:Person = {"Arnold", "Schwarzenegger", 61};

> forall (p:Person) p.lastName != nil;       -- evaluates to true
```

User Database Specification Example

- Pedagogical example for a distributed calendaring application
- Used for undergrad instruction at Cal Poly (CSC 308 – Gene Fisher)

```
object UserDB
    components: UserRecord*;
    operations: AddUser, FindUser, ChangeUser, DeleteUser;
    description: (*
        UserDB is the repository of registered user information.
    *);
end UserDB;

object UserRecord
    components: name:Name and id:Id and email:EmailAddress and
        phone:PhoneNumber;
    description: (*
        A UserRecord is the information stored about a registered user of the
        Calendar Tool.  The Name component is the user's real-world name.  The
        Id is the unique identifier by which the user is known to the Calendar
        Tool.  The EmailAddress is the electronic mail address used by the
        Calendar Tool to contact the user when necessary.  The PhoneNumber is
        for information purposes; it is not used by the Calendar Tool for
        contacting the user.
    *);
end User;

object Name = string;
object Id = string;
object EmailAddress = string;
object PhoneNumber = area:Area and num:Number;
object Area = integer;
object Number = integer;
```

AddUser Operation

```
operation AddUser
    ~~inputs: udb:UserDB, ur:UserRecord;~~
    ~~outputs: udb':UserDB;~~

    precondition:
        (*
         * The id of the given user record must be unique and less
         * than or equal to 8 characters; the email address must be
         * non-empty; the phone area code and number must be 3 and
         * 7 digits, respectively.
         *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *);

    description: (* As above *);

end AddUser;
```

```
(*
 * Create some testing values.
 *)
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val udb = [ur1, ur2];
val udb_added = udb + ur3;

> AddUser(udb,ur3)?->(udb_added);


Output:

{ true, nil }
```

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition: (* Coming soon. *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        ur in udb';

end AddUser;
```

```
(*
 * Create some testing values.
 *)
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val udb = [ur1, ur2];
val udb_added = udb + ur3;

> AddUser(udb,ur3)?->(udb_added);


Output:

{ true, true }
```

Fundamental Question

- Are the ~~preconditions and post~~conditions strong enough?
- In the AddUser example, ~~the~~ precondition is ~~non-existent and thus it's~~ maximally weak
- To test postcondition strength, ~~we can~~ use the validation operator to run ~~some~~ example ~~inputs and outputs against AddUser~~

```
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val ur4 = {"Extra", "4", nil, nil};
val udb = [ur1, ur2];

~~(*~~
 ~~* A database value representing a spurious addition having been~~
 ~~* made.~~
 ~~*)~~
val udb_spurious_addition = udb + ur3 + ur4;

~~(*~~
 ~~* A database value representing a spurious deletion having been made.~~
 ~~*)~~
val udb_spurious_deletion = udb + ur3 - ur2;

> AddUser(udb,ur3)?->(udb_spurious_addition);

> AddUser(udb,ur3)?->(udb_spurious_deletion);
```

```
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val ur4 = {"Extra", "4", nil, nil};
val udb = [ur1, ur2];

~~(*~~
 ~~* A database value representing a spurious addition having been~~
 ~~* made.~~
 ~~*)~~
val udb_spurious_addition = udb + ur3 + ur4;

~~(*~~
 ~~* A database value representing a spurious deletion having been made.~~
 ~~*)~~
val udb_spurious_deletion = udb + ur3 - ur2;

> AddUser(udb,ur3)?->(udb_spurious_addition);

> AddUser(udb,ur3)?->(udb_spurious_deletion);
```

```
Output:
{ true, true }
{ true, true }
```

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        ~~(*~~
         ~~* The given user record is in the output UserDB.~~
         ~~*)~~
        (ur in udb')

            and

        ~~(*~~
         ~~* All the other records in the output db are those from the~~
         ~~* input db, and only those.~~
         ~~*)~~
        forall (ur':UserRecord | ur' != ur)
            if (ur' in udb)
            then (ur' in udb')
            else not (ur' in udb');
end AddUser;
```

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        ~~(*~~
         ~~* The given user record is in the output UserDB.~~
         ~~*)~~
        (ur in udb')

            and

        ~~(*~~
         ~~* All the other records in the output db are those from the~~
         ~~* input db, and only those.~~
         ~~*)~~
        forall (ur':UserRecord | ur' != ur)
            if (ur' in udb)
            then (ur' in udb')
            else not (ur' in udb');
end AddUser;
```

```
Output:

{ true, false }

{ true, false }
```

Constructive Postcondition

- Constructive operations perform an actual constructive calculation
- Analytic operations evaluate Boolean expressions about the arguments

Constructive Postcondition

- Constructive operations perform an actual constructive calculation
- Analytic operations evaluate Boolean expressions about the arguments

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        udb' = udb + ur;
end AddUser;
```

FindUserByName Operation

```
operation FindUserByName
    inputs: udb:UserDB, name:Name;
    outputs: ur':UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * A record is in the output list if and only if it is in
         * the input UserDB and the record name equals the Name
         * being searched for
         *);

    description: (*
        Find a user or users by real-world name. If more than one is
        found, output list is sorted by id.
    *);
end FindUserByName;
```

```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

Validation Operator Invocation Results: English Comments

```
"What happens if there are unique, unsorted records?"
{ true, nil }
"What happens if there are unique, sorted records?"
{ true, nil }
"What happens if there are non-unique, unsorted records?"
{ true, nil }
"What happens if there are non-unique, sorted records?"
{ true, nil }
```

FindUserByName: Basic Logic

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n));

    description: (*
        Find a user or users by real-world name.  If more than one is
        found, the output list is sorted by id.
    *);
end FindUserByName;
```

```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

Validation Operator Invocation Results: Basic Logic

```
"What happens if there are unique, unsorted records?"
{ true, true }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, true }
"What happens if there are non-unique, sorted records?"
{ true, true }
```

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and
        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id <= url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one
        is found, the output list is sorted by id.
    *);
end FindUserByName;
```

```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

Validation Operator Invocation Results: Basic with Sort Constraint

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, true }
```

```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and
        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id < url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one is
        found, the output list is sorted by id.
    *);
end FindUserByName;
```

Validation Operator Invocation Results: Strengthened Logic

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, false }
```

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and
        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id < url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one is
        found, the output list is sorted by id.
    *);
end FindUserByName;
```

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;
    postcondition:
        RecordsFound(udb,n,url)
            and
        SortedById(url);
end FindUserByName;

function RecordsFound(udb:UserDB, n:Name, url:UserRecord*) =
    (*
     * The output list consists of all records of the given name in
     * the input db.
     *)
    (forall (ur' in url)
       (ur' in udb)
          and
       (ur'.name = n));

function SortedById(url:UserRecord*) =
    (*
     * The output list is sorted alphabetically by id.
     *)
        (if (#url > 1) then
            (forall (i in [1..(#url - 1)])
               url[i].id < url[i+1].id)
         else true);
```

Validation Operator Invocation Results: Auxiliary Functions

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, false }
```

Translating User-level Requirements into Boolean Logic

1. ~~There is no user record in the input database with the same id as the record to be added~~
2. ~~The id of an added user record cannot be empty and must be no more than 8 characters in length~~
3. ~~If the area code and phone number are present, they must be 3 digits and 7 digits respectively~~

```
operation AddUser
    ~~inputs: udb:UserDB, ur:UserRecord;~~
    ~~outputs: udb':UserDB;~~
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (ur in udb))

            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    ~~postcondition: (* Same as above *);~~
~~end AddUser;~~
```

You can leave this slide out.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (ur in udb))

            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

Validation Operator Invocation #1

```
val phone:PhoneNumber = {805, 5551212};
val email:EmailAddress = "pcorwin@calpoly.edu";
val ur:UserRecord = {"Corwin", "1", email, phone};
val ur_duplicate_id:UserRecord = {"Fisher", "1", email, phone};
val udb:UserDB = [];
val udb_added:UserDB = [ur];

> AddUser(udb_added, ur_duplicate_id) ?-> (udb_added);
```

Requirement Translation Flaw #1

- There is no user record in the input database with the same id as the record to be added

```
Flawed:

 (not (ur in udb))
```

```
Correct:

 (not (exists (ur' in udb) ur'.id = ur.id))
```

You can leave this slide out.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))
            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

Validation Operator Invocation #2

```
val ur_empty_id:UserRecord = {"Corwin", nil, email, phone};

> AddUser(udb, ur_empty_id) ?-> (udb);
```

Requirement Translation Flaw #2

- The id of an added user record cannot be empty and must be no more than 8 characters in length

```
Flawed:

(#(ur.id) <= 8)
```

```
Correct:

(ur.id != nil) and (#(ur.id) <= 8)
```

You can leave this slide out.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))
            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)
            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

Validation Operator Invocation #2

```
val ur_empty_phone:UserRecord = {"Corwin", "1", email, nil};

> AddUser(udb, ur_empty_phone)?->(udb);
```

Requirement Translation Flaw #3

- If the area code and phone number are present, they must be 3 digits and 7 digits respectively

```
Flawed:

(#(ur.phone.area) = 3) and
(#(ur.phone.num) = 7));
```

```
Correct:

(if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
(if (ur.phone.num != nil) then (#(ur.phone.num) = 7));
```

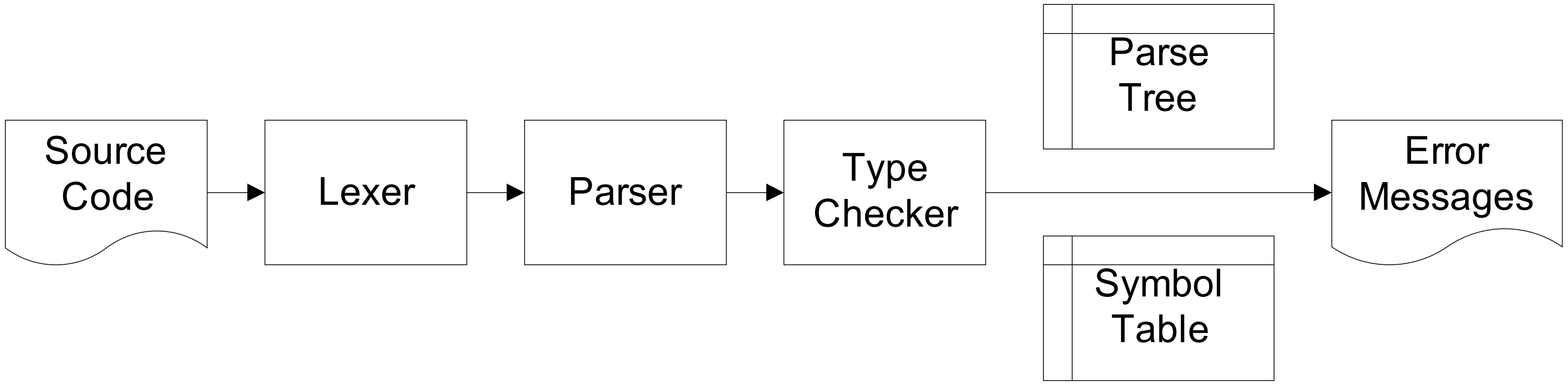
You can leave this slide out.

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))

            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
        (if (ur.phone.num != nil) then (#(ur.phone.num) = 7));

    postcondition: (* Same as above *);
end AddUser;
```

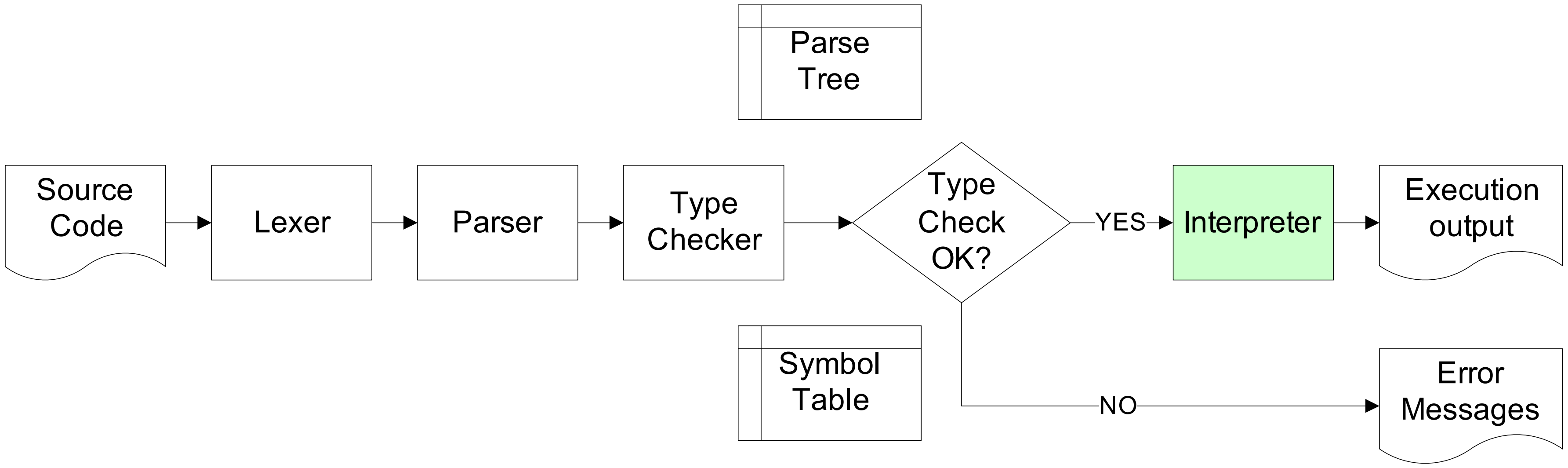
Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

FMSL Pre-Thesis



FMSL Post-Thesis



New to FMSL: The Functional Interpreter

- Expression evaluation
- Function / operation evaluation
- Execution of preconditions and postconditions
- Quantifier evaluation
- Value universe
- Validation operator

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Basic Object Types

- boolean
- integer
- real
- string
- nil

Basic Operators: booleans

Operator	**Description**	**Returns**
`not`	negation	`boolean`
`and`	conjunction	`boolean`
`or`	disjunction	`boolean`
`xor`	exclusive disjunction	`boolean`
`=>`	implication	`boolean`
`<=>`	two-way implication; if and only if	`boolean`
`if b1 then b2` where `b1`, `b2` are Boolean expressions	conditional	`boolean`
`if b1 then b2 else b3` where `b1`, `b2`, `b3` are Boolean expressions	conditional with else	`boolean`

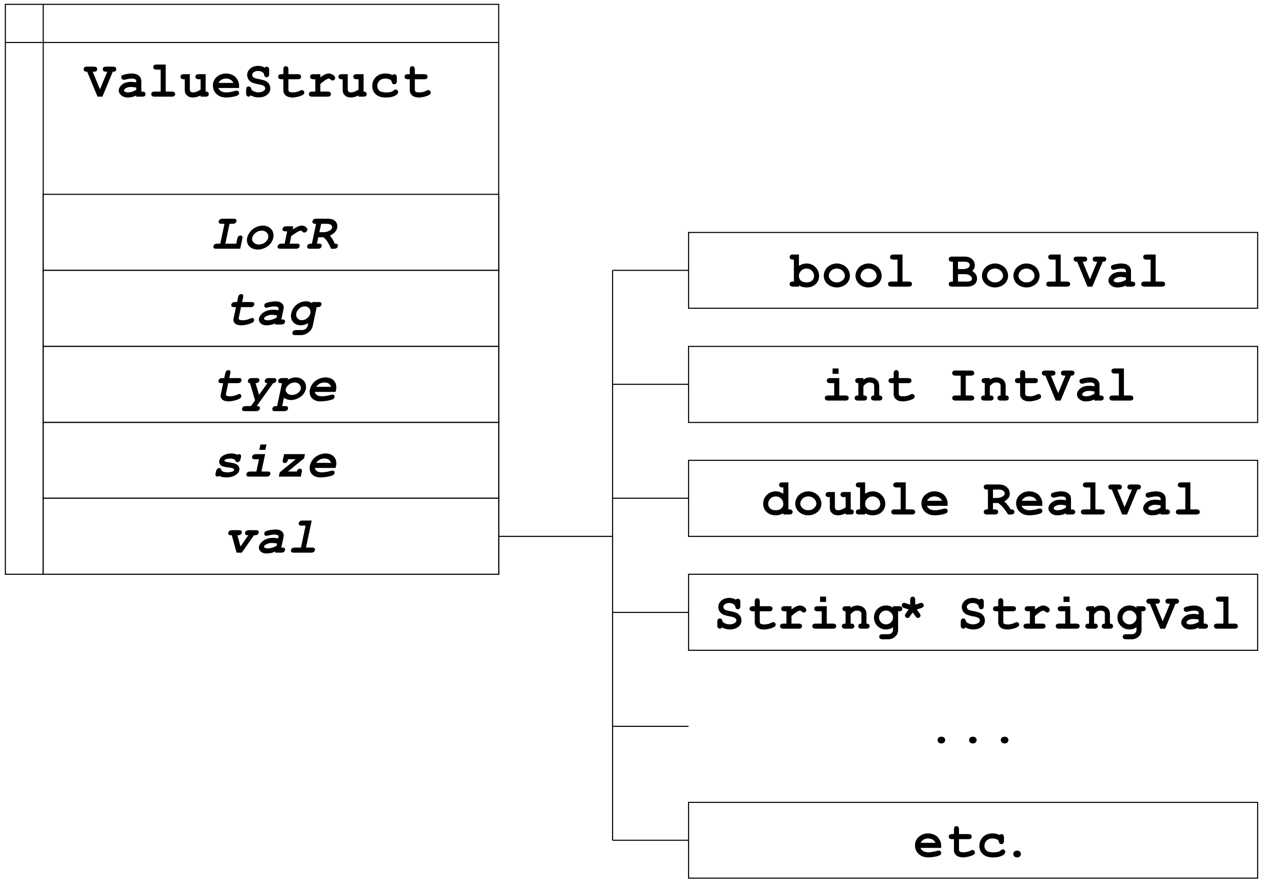
Basic Operators: integers and reals

Operator	**Description**	**Returns**
`+`	Addition	`integer` or `real`
`-`	Subtraction	`integer` or `real`
`*`	multiplication	`integer` or `real`
`/`	Division	`integer` or `real`
`mod`	Modulus	`integer`
`+ (unary)`	returns 1*the number	`integer` or `real`
`- (unary)`	returns -1*the number	`integer` or `real`
`=`	Equality	`boolean`
`!=`	Inequality	`boolean`
`>`	greater than	`boolean`
`<`	less than	`boolean`
`>=`	greater than or equal to	`boolean`
`<=`	less than or equal to	`boolean`

Basic Operators: strings

Operator	**Description**	**Returns**
`=`	equality	`boolean`
`!=`	inequality	`boolean`
`#`	string length	`integer`
`in`	membership test	`boolean`
`+`	concatenation	`string`
`[n]`	single character selection	`string`
`[m .. n]`	range / substring selection	`string`

Internal Representation of Values



Division Operator Example

```
 (*
  * Declare and assign values to x, y
  *)
val x:real = 3.141592654;
val y:real = 2.718281828;

 (*
  * Evaluate x divided by y and output the result
  *)
> x / y;
```

- Determine that the expression involves a binary operator
- Determine the operator (/)
- Call and return the result of the function that performs the division, passing in as parameters the `ValueStruct`s corresponding to the x and y operands

```
ValueStruct doRealDiv(ValueStruct v1, ValueStruct v2, nodep t) {
    /*
     * Propagate null value if either is operand is null.
     */
    if ((v1 == null) or (v2 == null))
        return null;
    /*
     * Handled the overload for real or integer operands.
     */
    switch (v1->tag) {
        case RealTag:
            if (v2->tag == IntTag) {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.IntVal;
            }
            else {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.RealVal;
            }
            free(v2);
            return v1;
        case IntTag:
            if (v2->tag == RealTag) {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.RealVal = v1->val.IntVal / v2->val.RealVal;
                v1->tag = RealTag;
            }
            else {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.IntVal = v1->val.IntVal / v2->val.IntVal;
            }
            free(v2);
            return v1;
    }
}
```

Complex Structures

- Lists
  - Homogeneous
  - Hold zero or more object values
  - Analogous to an array with no predetermined, fixed size
- Tuples
  - Heterogeneous
  - Hold fixed number of components of specific object types
  - Analogous to a C struct

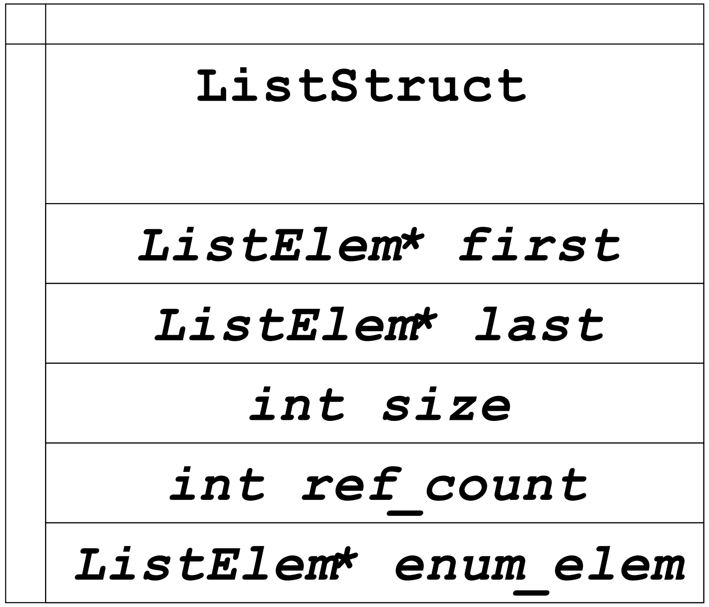
Basic Operators: lists

Operator	Description	Returns
`=`	equality	`boolean`
`!=`	inequality	`boolean`
`in`	membership	`boolean`
`#`	element count	`integer`
`+`	concatenation	`list type`
`-`	deletion from list	`list type`
`[n]`	element selection	`list type`
`[m .. n]`	range selection	`list type`

Basic Operators: tuples

Operator	**Description**	**Returns**
`=`	Equality	`boolean`
`!=`	inequality	`boolean`
`.`	field access	`any field type`

ListStruct Definition



List Range Selection Example

Code listing:

```
(*
* Declare the IntegerList type
*)
object IntegerList = integer*;

(*
* Declare an intlist value
*)
val intlist:IntegerList = [1,1,2,3,5,3+5];

(*
* Select the subcomponents at indexes 3, 4, and 5.
*)
> intlist[3..5];
```

Output:

```
[ 2, 3, 5 ]
```

List Range Selection Example

- Determine that the expression involves a ternary operator with three operands: list, lower bound, upper bound
- Determine the operator ([] – list range selection)
- Call and return the result of the function that performs the list range selection, passing in as parameters the `ValueStruct`s corresponding to the list, lower bound, and upper bound operands

```
ValueStruct doArraySliceRef(v1, v2, v3)
    ValueStruct v1;
    ValueStruct v2;
    ValueStruct v3;
{
    ValueStruct result;
    int i;
    /* start building the new list */
    result = MakeVal(RVAL, v1->type);
    if (v1->tag == ListTag) {
        result->val.ListVal = NewList();
        /*
         * loop through from lower .. upper and add the elements
         * to result.
         */
        for (i = v2->val.IntVal; i <= v3->val.IntVal; i++) {
            PutList(result->val.ListVal,
                    GetListNth(v1->val.ListVal, i));
        }
    }
    else if (v1->tag == StringTag) {
        result->val.StringVal =
            (String *)SubString(v1->val.StringVal,
                                v2->val.IntVal,
                                v3->val.IntVal);
    }
    return result;
}
```

Tuple Field Access Example

Code listing:

```
(*
* Declare p, a person variable
*)
val p:Person = {"Arnold", "Schwarzenegger", 61};

(*
* Access p's last name field
*)
> p.lastName;
```

Output:

```
"Schwarzenegger"
```

- Determine that the expression involves a binary operator with two operands: the tuple name and the tuple field name
- Determine the operator (. – field access)
- Call and return the result of the function that performs the field access, passing in as parameters the `ValueStruct`s corresponding to the tuple and the tuple field name

```
ValueStruct RecordRef(desig, field)
    ValueStruct desig;  /* L-value for the left operand. */
    nodep field;        /* Ident for the right operand. */
{
    ValueStruct valueField,
        tuple,
        newDesig;
    SymtabEntry *f;
    int n;
    TypeStruct type = ResolveIdentType(desig->type, null, false),
        fieldType;

    /*
     * Deal with nil desig, i.e., just return it as is.
     */
    if (isNilValue(desig)) {
        return desig;
    }
```

```
/*
 * coming in, desig->LVal should point to the ValueStruct of the
 * struct
 */
if (field->header.name == Yident) {
    f = LookupIn(field->components.atom.val.text,
                    type->components.type.kind.record.fieldstab);
    fieldType = ResolveIdentType(f->Type, null, false);
}
else {
    f = null;
    n = field->components.atom.val.integer;
    fieldType = ResolveIdentType(
        GetNthField(type->components.type.kind.record.fields, n)->
            components.decl.kind.field.type, null, false);
}
```

```
    if (desig->LorR == LVAL) {
        tuple = (ValueStruct)*(desig->val.LVal);
    }
    else {
        tuple = desig;
    }

    /* Note: Lists are 1-indexed */
    valueField = (ValueStruct)GetListNth(tuple->val.StructVal,
        f ? f->Info.Var.Offset + 1 : n);
    /*
     * if we have valueField filled in, use its type... otherwise use the
     * fieldType
     */
    if (!valueField) {
        newDesig = MakeVal(LVAL, fieldType);
    }
    else {
        newDesig = MakeVal(LVAL, valueField->type);
    }
    newDesig->val.LVal = (ValueStruct *) malloc(sizeof(Value **));
    *(newDesig->val.LVal) = valueField;

    return newDesig;
}
```

Operation Invocation

Code listing:

```
operation Cube (x:integer) = x * x * x;

> Cube(2);
> Cube(5);
```

Output:

```
8
125
```

Validation Operator Invocation

- Recall that generically, the validation operator usage is:

```
operation_name(input argument list) ?-> (output argument list)
```

- The result is a two-tuple that contains boolean values
  - The first corresponds to precondition evaluation
  - The second corresponds to postcondition evaluation

Validation Result Indications

Tuple Returned	**Indication**
`{ nil, nil }`	execution failure in the precondition; postcondition evaluation not attempted
`{ false, nil }`	precondition evaluation failed; postcondition evaluation not attempted
`{ true, nil }`	precondition evaluation passed; no postcondition specified or there was an execution failure in the postcondition
`{ true, false }`	Precondition evaluation passed; postcondition evaluation failed
`{ true, true }`	Both precondition and postcondition evaluation passed

Validation Result Significances

Tuple Returned	Significance
`{ nil, nil }`	The precondition may be specified incorrectly since a run-time / execution error was detected during precondition execution
`{ false, nil }`	Test values for inputs were invalid or the precondition was specified incorrectly
`{ true, nil }`	Test values for inputs were valid, but the postcondition either wasn’t specified or it may be specified incorrectly since a run-time / execution error was detected during postcondition execution
`{ true, false }`	Test values for inputs were valid, but the output values were invalid or the postcondition was specified incorrectly
`{ true, true }`	Test values for both inputs and outputs agreed with both the precondition and postcondition

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Chapter 6: Quantifier Execution

- FMSL supports quantifiers that are
  - Bounded or unbounded
  - Of universal (forall) or existential (exists) forms

Quantifier Syntax

Syntax	**Quantifier Type**	**Reads Like …**
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true

Bounded Quantifier

```
 (*
* Declare an IntList object type and an IntList value
*)
obj IntList = integer*;
val list:IntList = [ 1, 1, 2, 3, 5 ];

 (*
* Evaluate: all the integer elements within list are positive.
*)
> forall (i in list) i > 0;               -- evaluates to true
```

Unbounded Quantifier

```
obj Person = name:Name and age:Age;
obj Name = string;
obj Age = integer;

> forall (p:Person) p.age >= 21;
```

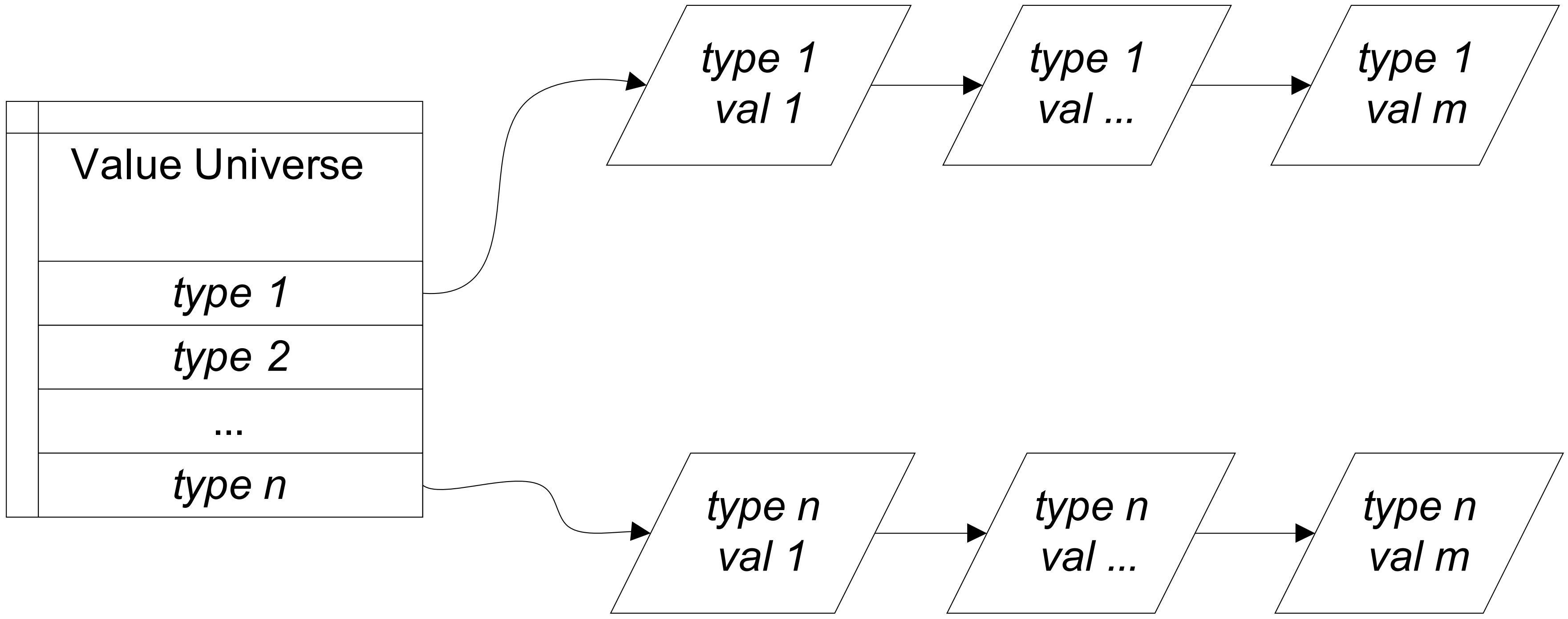
Other Methods of Unbounded Quantifier Execution

- Aslantest
  - When it cannot automatically reduce an expression to true or false, it suspends execution and prompts for user input
- Jahob
  - pickAny: picks an arbitrary value, optionally bounded by lemmas that the user can input, that is placed into the unbounded quantifier
- Executable Z
  - Treats unbounded quantification as a source of non-executability, so such statements are treated as compiler assumptions (and not executable statements)

Unbounded Quantification in FMSL: Value Universe

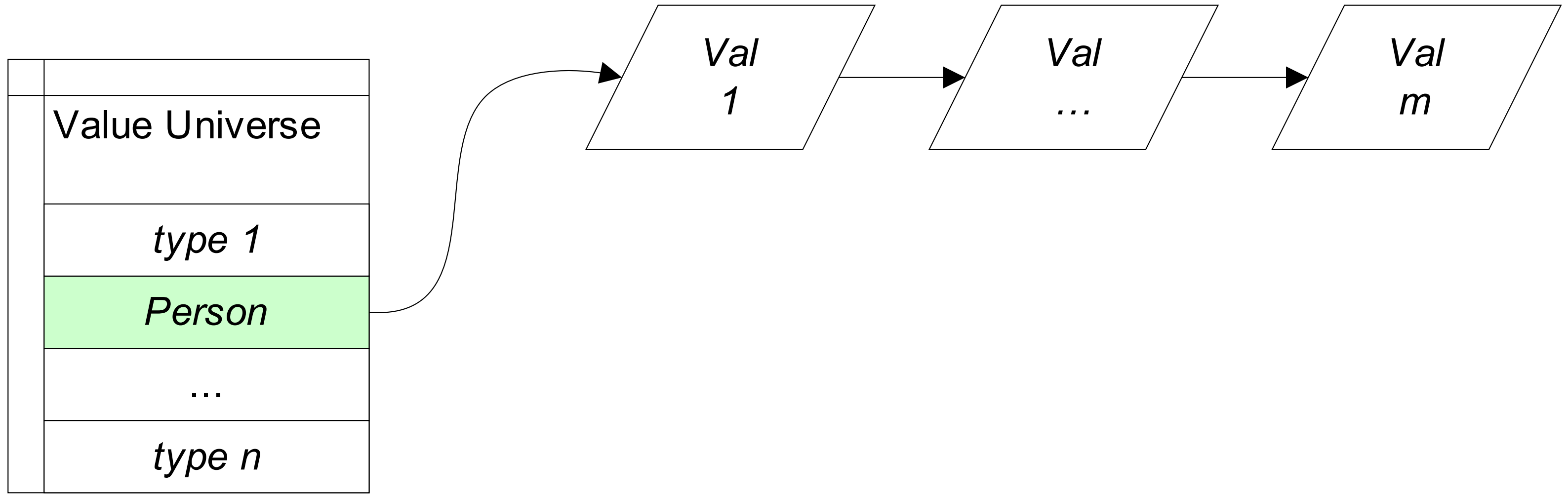
- A discrete pool of values, indexed by type, that supply meaningful values to unbounded quantifier predicates
- Can contain values of any value type, from simple atomic types to complex types like lists and tuples
- Grows incrementally as values appear during specification execution
- With repeatability in mind, values added primarily in contexts where values cannot be mutated
  - Let expressions
  - Parameter binding
  - List construction
- By default does not contain duplicates

Value Universe Structure

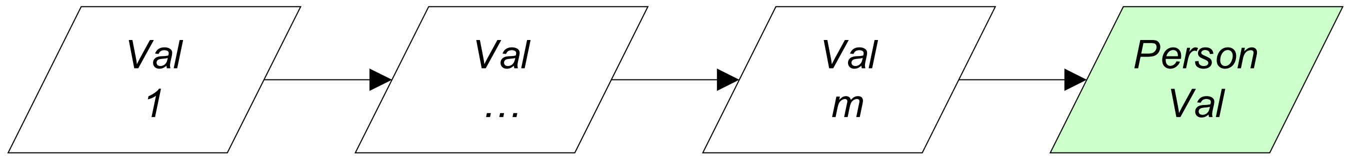


Value Universe Add

1.



2.



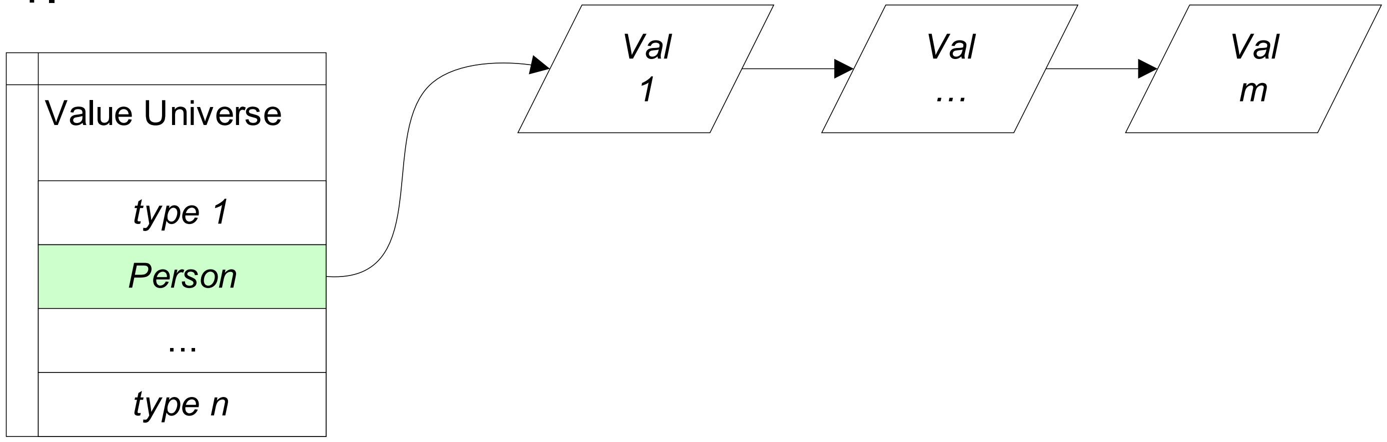
Quantifier Syntax

Syntax	**Quantifier Type**	**Reads Like …**
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true

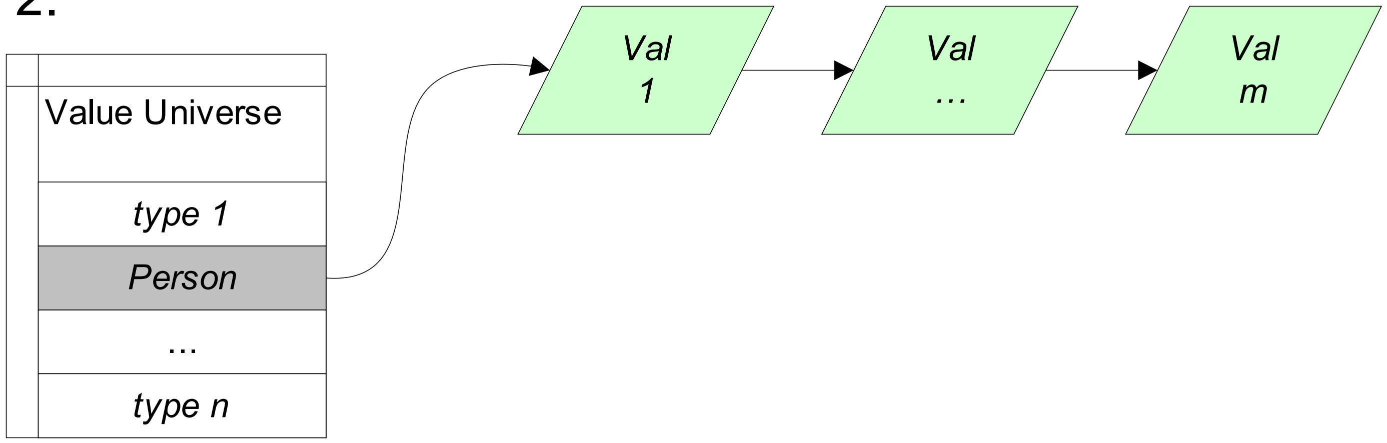
Unbounded Existential Quantifier

```
(*
 * Perform lets with p1, p2 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);

> "Expected: false";
> exists (p:Person) p.lastName = nil;

(*
 * Since p3, with a nil last name, has been introduced
 * then we expect true below.
 *)

> (let p3:Person = {"Charles", nil, 218};);
> "Expected: true";
> exists (p:Person) p.lastName = nil;
```

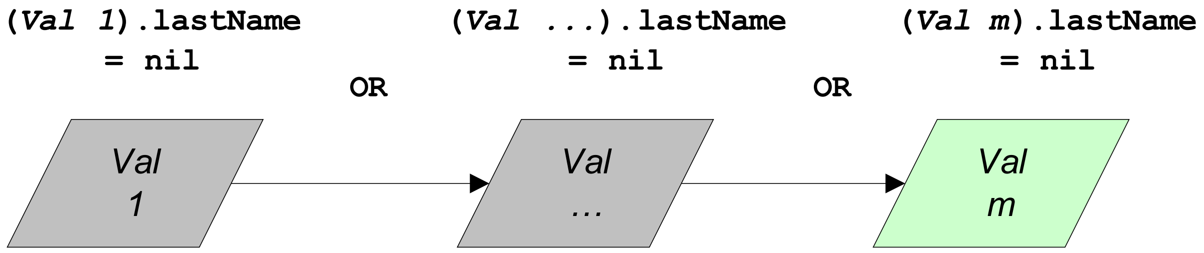
1.



2.



3.



```
(Val 1).lastName        (Val ...).lastName        (Val m).lastName
    = nil                     = nil                     = nil
```

Unbounded Existential Quantifier

```
(*
 * Perform lets with p1, p2 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);

> "Expected: false";
> exists (p:Person) p.lastName = nil;

(*
 * Since p3, with a nil last name, has been introduced
 * then we expect true below.
 *)

> (let p3:Person = {"Charles", nil, 218};);
> "Expected: true";
> exists (p:Person) p.lastName = nil;
```

Evaluates to false

Evaluates to true

Quantifier Syntax

Syntax	**Quantifier Type**	**Reads Like …**
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true

Unbounded Universal Quantifier (with suchthat)

```
(*
 * Perform lets with p1, p2, p3 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);
> (let p3:Person = {"Charles", nil, 218};);

(*
 * Evaluate: for all Person objects such that p.lastName is
 * not nil, the last name length is at least 6 characters
 *)
> "Expected: true";
> forall (p:Person | p.lastName != nil) #p.lastName >= 6;
```

Unbounded Universal Quantifier (with suchthat)

```
(*
 * Perform lets with p1, p2, p3 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);
> (let p3:Person = {"Charles", nil, 218};);

(*
 * Evaluate: for all Person objects such that p.lastName is
 * not nil, the last name length is at least 6 characters
 *)
> "Expected: true";
> forall (p:Person | p.lastName != nil) #p.lastName >= 6;
```

Evaluates to true

Presentation Outline

Summary of Contributions

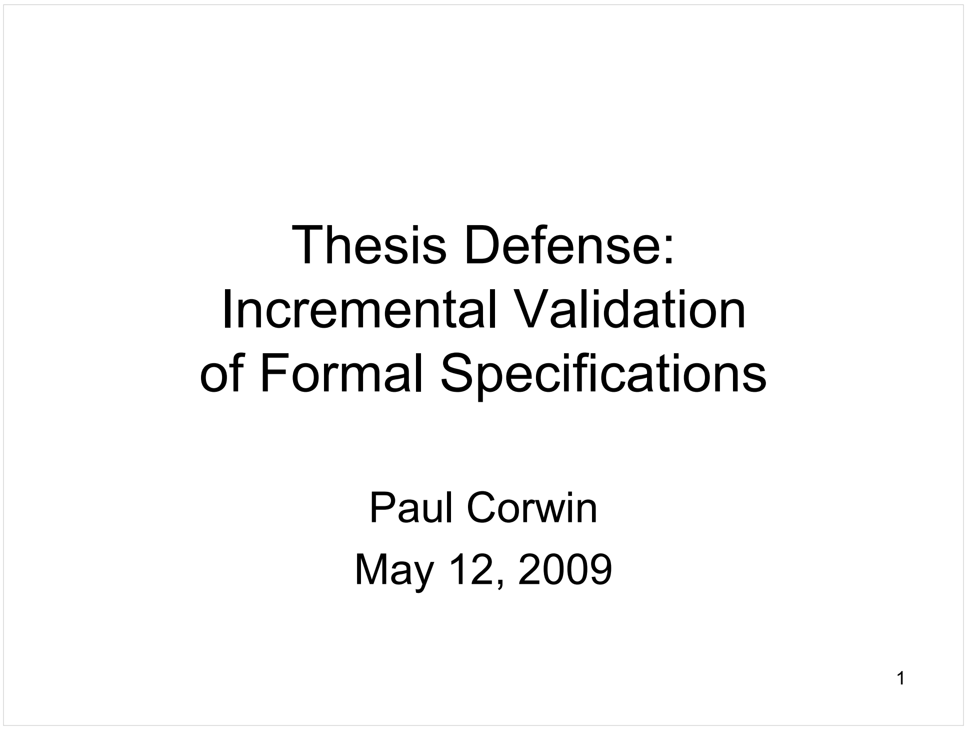
1. The design and implementation of a functional interpreter for a formal specification language, ~~rendering the language executable for the first time~~
2. The design and implementation of a ~~novel technique to execute purely predicative specifications, using~~ validation ~~operator~~ invocations
3. Demonstration of ~~how the~~ execution capabilities ~~can be used to validate formal specifications~~
4. ~~A thorough~~ discussion of how ~~the~~ specification execution capabilities fit into the realm of light-weight and heavy-weight formal methods

Future Work

- UML to FMSL tool
- Test case generator
- GUI front end
- Improve value universe performance
- Garbage collector
- End-user studies

Questions?

The End



Thesis Defense: Incremental Validation of Formal Specifications

Paul Corwin

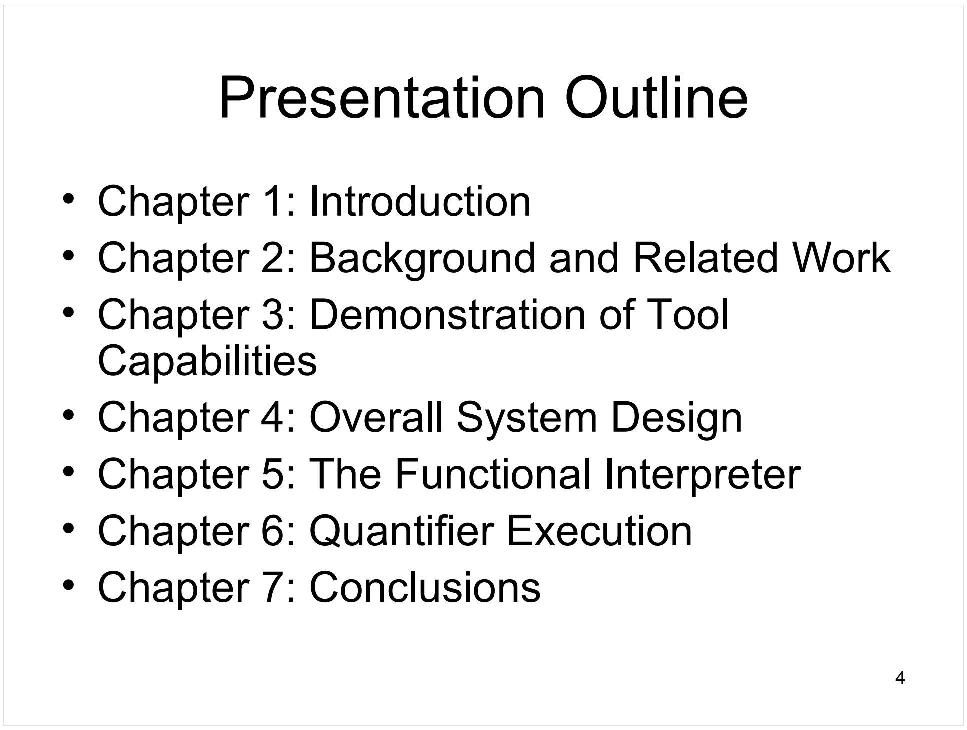
May 12, 2009

Committee Members

Name	Role
Dr. Gene Fisher	Advisor & Committee Chair
Dr. David Janzen	Committee Member
Dr. Clark Turner	Committee Member

Incremental Validation of Formal Specifications

- Presents a tool for the mechanical validation of formal software specifications
- Novel approach to incremental validation
- Form of “light-weight” model checking
- Part of FMSL: a formal modeling and specification language
- Small-scale aspects of a specification are validated; step-wise refinement
- Presents example that’s been used in software engineering courses for years
- Use of the tool led to discovery of a specification flaw



Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Chapter 1: Introduction

- Software engineering is error-prone and expensive
- Early detection of errors is beneficial
- Thesis focuses on early error detection during formal specification
- Tool-supported technique: FMSL
- Added executability
  - Standard functional evaluation
  - Boolean expressions containing universal and existential quantifiers, bounded and unbounded

The Problem

- How to validate a formal model-based specification?
- Need for tools and methods that expose errors, misunderstood properties, improperly stated behaviors
- FMSL model behavior defined with Boolean preconditions and postconditions on operations
- Evaluating quantifier expressions of particular interest

Thesis Aims

- Provide a means to validate formal specifications in a straightforward manner
- Demonstrate practicality in an instructional context

FMSL

- Was a predicative specification language
- Had type checker
  - Performs syntactic and semantic analysis
  - Comparable to compilers of strongly typed programming languages
- Added executability
- Focused on operation validation

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Chapter 2: Background and Related Work

- Formal methods and related topics
- “Lightweight” formal methods
- Relevant specification languages and model checkers

Formal Methods: What Are They?

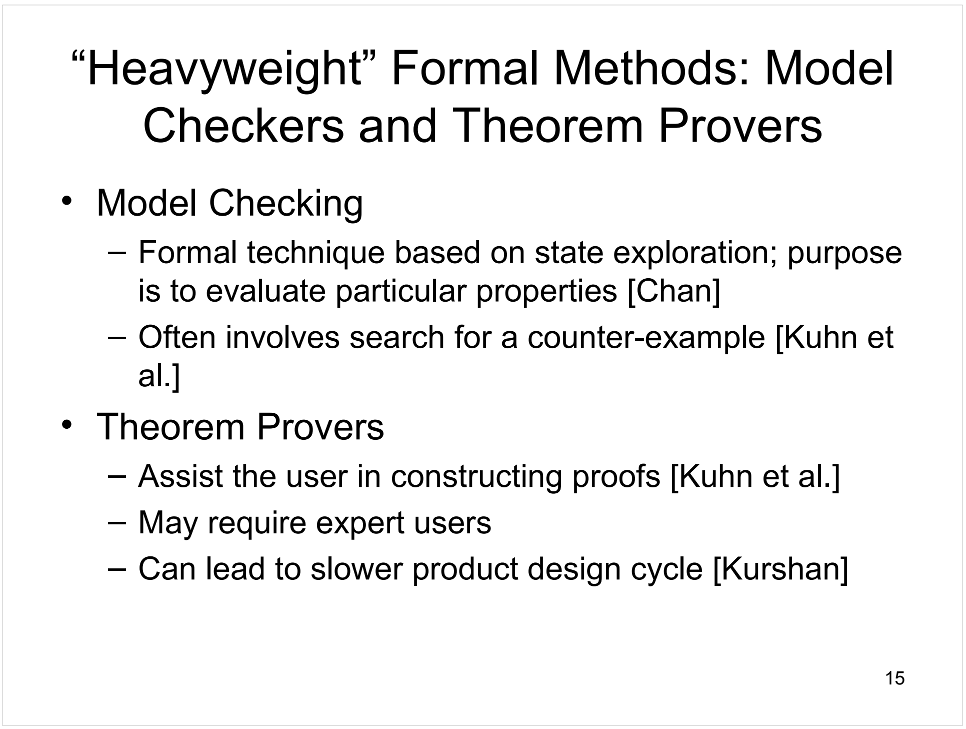
- Processes that exploit the power of mathematical notation and proofs
- Express system and software properties
- Help establish whether a specification satisfies certain properties or meets certain behavioral constraints

Formal Methods: Downsides

- Played “insignificant” role in software engineering in last 30 years [Glass]
- Rarely used, high barrier of entry [Heitmeyer]
- Few people understand what formal methods are or how to use them [Bowen et al.]
- High up front cost [Larsen et al.]

Formal Methods: Upsides

- Can be used during requirements development, specification, design, and implementation
- Practical means of showing absence of undesired behavior [Kuhn]
- Helps users better understand a system
  - Formality prompts engineers to raise questions [Easterbrook et al.]
  - Forces early, serious consideration of design issues [Jackson et al.]
  - Abstraction can mask complexities [Agerholm et al.]
- Cost savings achieved [Larsen et al.]



“Heavyweight” Formal Methods: Model Checkers and Theorem Provers

- Model Checking
  - Formal technique based on state exploration; purpose is to evaluate particular properties [Chan]
  - Often involves search for a counter-example [Kuhn et al.]
- Theorem Provers
  - Assist the user in constructing proofs [Kuhn et al.]
  - May require expert users
  - Can lead to slower product design cycle [Kurshan]

Model checking challenges -

•Requires specialized expertise

•Can be costly

•State explosion

  •So many variables that the model size “explodes” exponentially

  •Not enough computing resources to carry out the state exploration algorithm

Formal Methods: Can Be Used on Individual System Parts

- Can be overkill for a system in its entirety [Bowen]
- Sometimes only system parts would benefit from formal methods [Agerholm et al.]
- Requires consideration to determine where formal methods use makes sense [Larsen et al.]
- Formalized parts can be re-used in other projects [Kuhn et al.]
- Promotes code re-use [Jackson et al.]

Lightweight Formal Methods

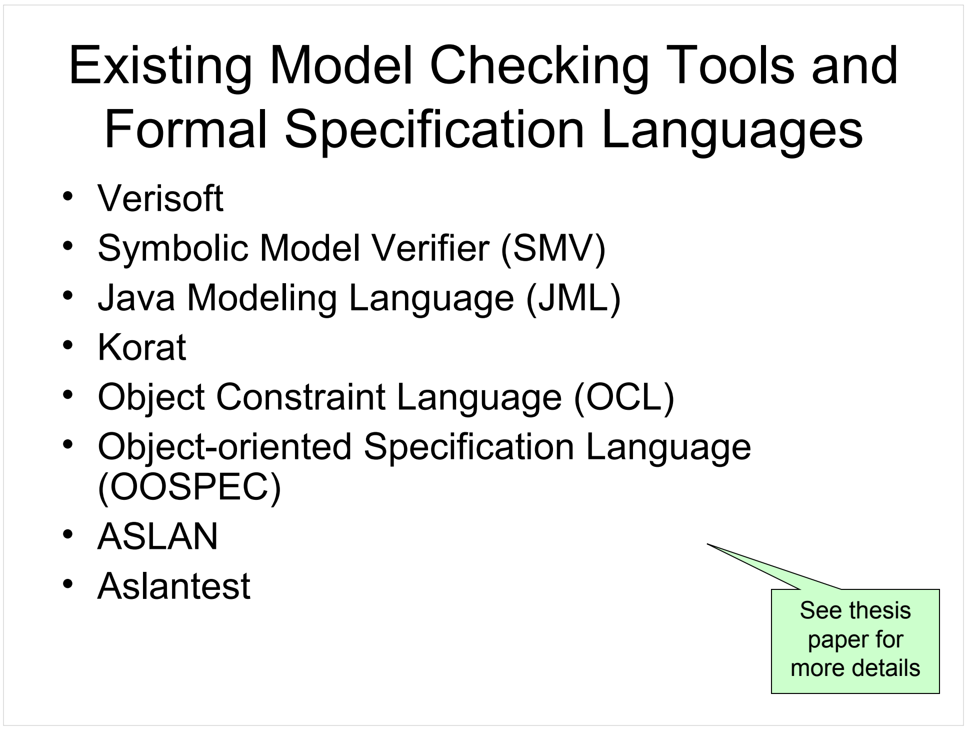
- Formal technique that falls short of complete verification
- May not require that the user be trained in advanced mathematics or sophisticated proof strategies [Heitmeyer]
- May use formal notations
- Can be more practical and cost effective than “heavyweight” formal methods [Jackson]

A Lightweight Technique: Simulation

- Animates a model by examining a small subspace of possible states [Jackson et al.]
- May immediately expose mistakes
- Provides ability to test functional requirements of interest early

Test-Driven Development and Simulation

- Calls for programmers to write low-level functional tests before beginning the implementation
  - Improves productivity [Erdogmus]
  - Leads to less complex, but highly tested code [Janzen et al.]
- Evaluating claims early on improves understanding [Myers]
- Whether generated manually or automatically, tests used for simulation purposes can be re-used against the implementation



Existing Model Checking Tools and Formal Specification Languages

- Verisoft
- Symbolic Model Verifier (SMV)
- Java Modeling Language (JML)
- Korat
- Object Constraint Language (OCL)
- Object-oriented Specification Language (OOSPEC)
- ASLAN
- Aslantest

See thesis paper for more details

Verisoft: Model checking tool that analyzes a system implementation

SMV: Heavy-weight model checking tool that uses a diagram-based algorithm to search for counterexamples, with formal semantics

JML: used for specifying java modules

Korat: Automatically generates test cases for JML specifications

OCL: part of UML, Must be accompanied by visual UML diagram

OOSPEC: Used to introduce formal methods to undergrad students, executable, influenced by languages that utilize set-theoretic notation

ASLAN: Supports system specification through definition of states and state transitions, users define entry and exit criteria

Aslantest: executes ASLAN specifications and supports symbolic checking as well as checking through individual test cases

Empirical Successes with Formal Methods

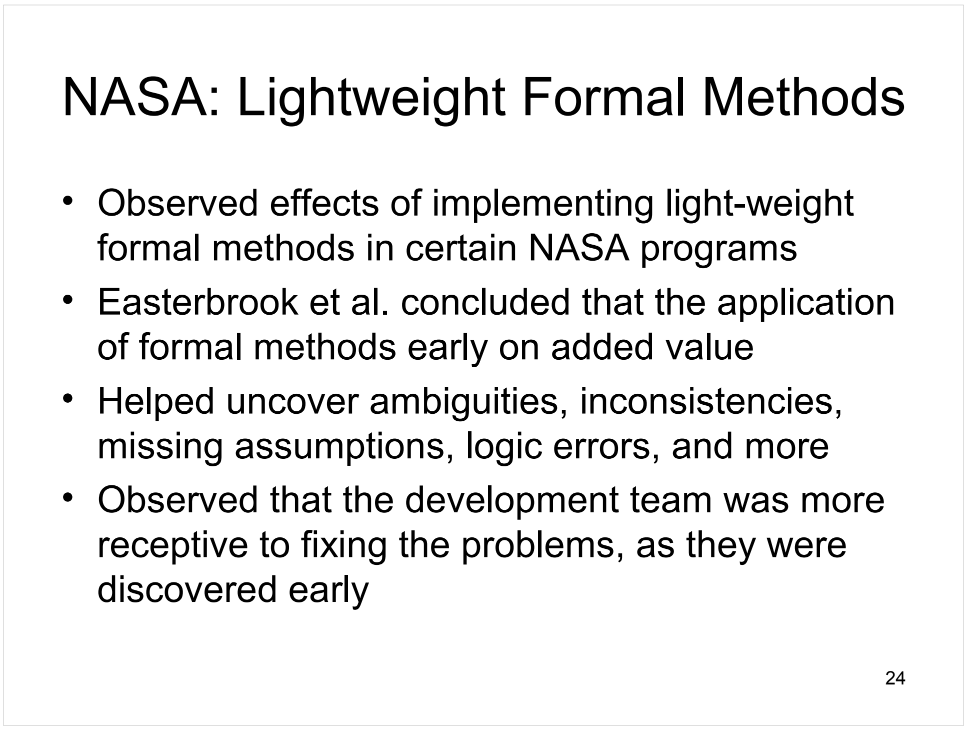
- BASE: A Trusted Gateway
- Miami University of Ohio: OOD Course
- NASA: Lightweight Formal Methods study

BASE: A Trusted Gateway

- Had formal methods and non-formal methods (control) groups develop a trusted gateway
- The formal methods group uncovered a hole in the requirements; the control team did not
- The formal methods group's software passed more tests than the control methods group's software
- The formal methods group's software performed fourteen times faster than the control group's software

Miami University of Ohio: OOD Course

- Had a formal methods group of students and a control group of students design and implement a common elevator project
- The formal methods teams followed more rigorous design processes and had relatively better designs
- 100% of the formal methods teams' implementations passed the tests; only 45.5% of the control group teams' implementations passed



NASA: Lightweight Formal Methods

- Observed effects of implementing light-weight formal methods in certain NASA programs
- Easterbrook et al. concluded that the application of formal methods early on added value
- Helped uncover ambiguities, inconsistencies, missing assumptions, logic errors, and more
- Observed that the development team was more receptive to fixing the problems, as they were discovered early

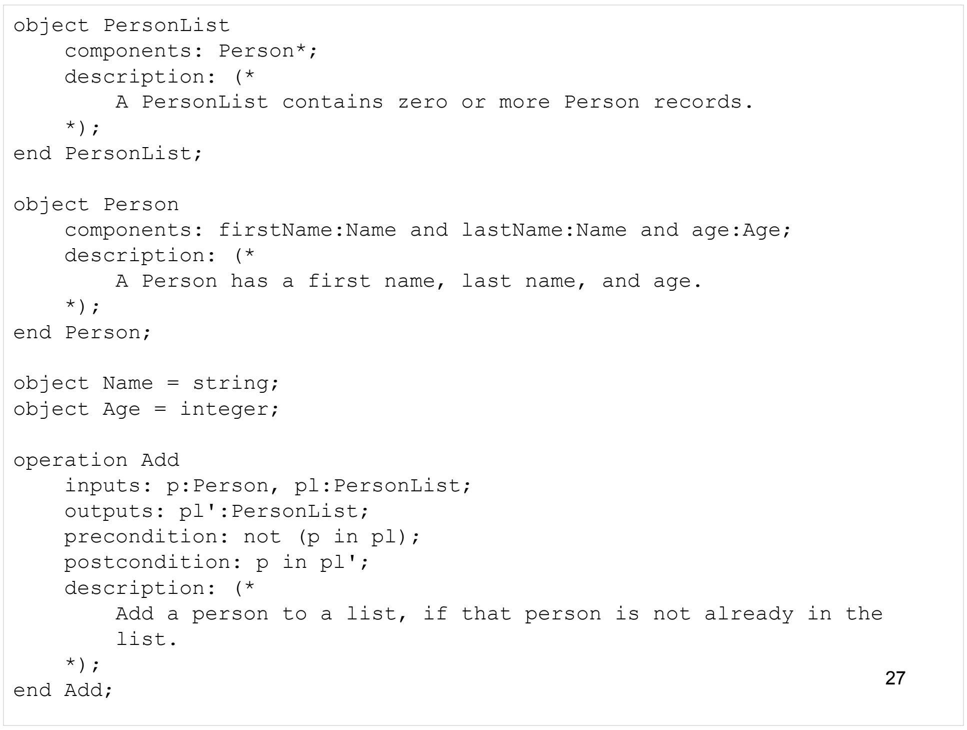
The bottom line: formal methods have been shown to improve the software development process.

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Chapter 3: Demonstration of Tool Capabilities

- Simple illustrative example
- Objects and operations



```
object PersonList
    components: Person*;
    description: (*
        A PersonList contains zero or more Person records.
    *);
end PersonList;

object Person
    components: firstName:Name and lastName:Name and age:Age;
    description: (*
        A Person has a first name, last name, and age.
    *);
end Person;

object Name = string;
object Age = integer;

operation Add
    inputs: p:Person, pl:PersonList;
    outputs: pl':PersonList;
    precondition: not (p in pl);
    postcondition: p in pl';
    description: (*
        Add a person to a list, if that person is not already in the
        list.
    *);
end Add;
```

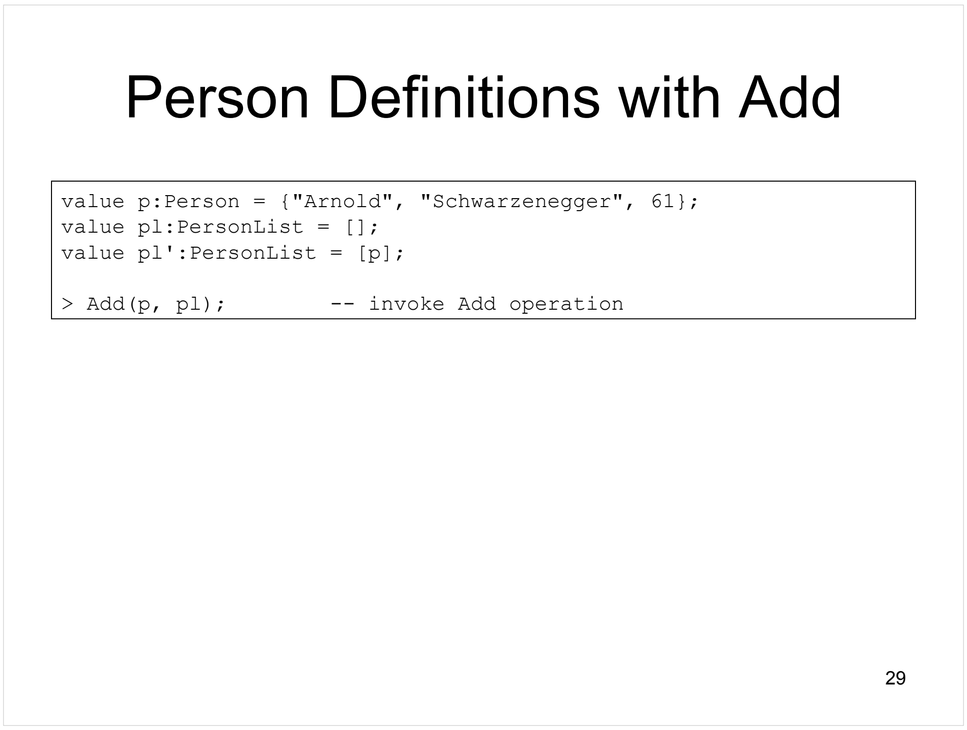
27

1. At the top we have four object definitions: PersonList, Person, Name, Age.
2. Note that the Person and PersonList objects have components.
3. Name and Age use an optional short form of object definition, useful for scalars.
4. We also have operation Add, which has inputs, outputs, precondition, & postcondition.
    - Operations not defined with an executable body. Instead, operations are expressed as Boolean predicates that must be true before and after an operation executes – i.e., preconditions and postconditions.

- Any identifier can have an apostraphe char as a suffix; purely lexical; most often used when the input and output object are the same; e.g., p1 and p1 prime.
- The in operator is built-in and tests for list membership
- The description field shows the paren star, which encloses comments

How Does One Validate That It Is Correct?

- Static correctness validated by FMSL type checker
  - Syntactic and semantic analysis
- Focus of this thesis is determining the dynamic correctness



Person Definitions with Add

```
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];

> Add(p, pl);        -- invoke Add operation
```

29

1. Value declaration defines a constant value
2. Tuple values are enclosed in curly braces; tuples are defined with anded components
3. List values are enclosed in square brackets; lists defined with * components (like PersonList)
4. Point-to-end-of-line comments are defined with two dashes.
5. Expression evaluations are preceded with the prompt or greater than character. Could be entered in a conversational interpreter, but may appear in a specification file. It’s important to note that the prompt character indicates expression vs. specification declaration.
6. Operations are invoked in a standard fashion: operation name followed by parenthesized list of actual parameters

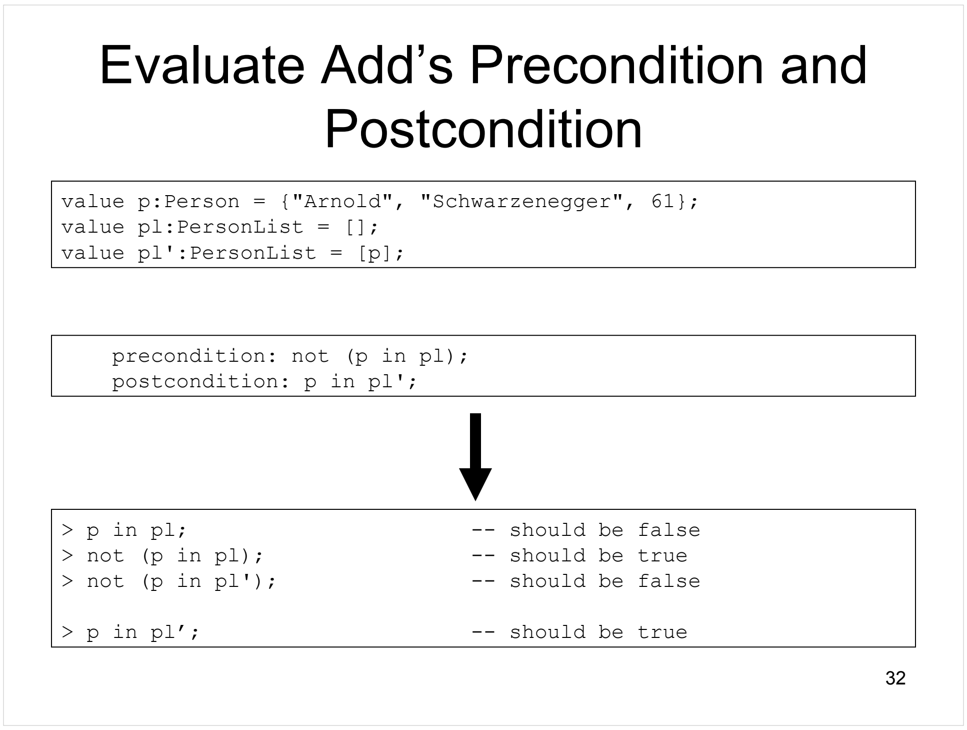
Person Definitions with Add

```
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];

> Add(p, pl);         -- invoke Add operation
```

- What value does the invocation of `Add(p, pl)` produce?
- `nil` -- the empty value for any type of object
- Same value produced for any inputs, since `Add` is defined only with a precondition and postcondition

Evaluate Add’s Precondition and Postcondition

```
operation Add
    inputs: p:Person, pl:PersonList;
    outputs: pl':PersonList;
    precondition: not (p in pl);
    postcondition: p in pl';
    description: (*
        Add a person to a list, if that person is not already in
        the list.
    *);
end Add;
```



Evaluate Add’s Precondition and Postcondition

```
value p:Person = {"Arnold", "Schwarzenegger", 61};
value pl:PersonList = [];
value pl':PersonList = [p];
```

```
    precondition: not (p in pl);
    postcondition: p in pl';
```

$\Downarrow$

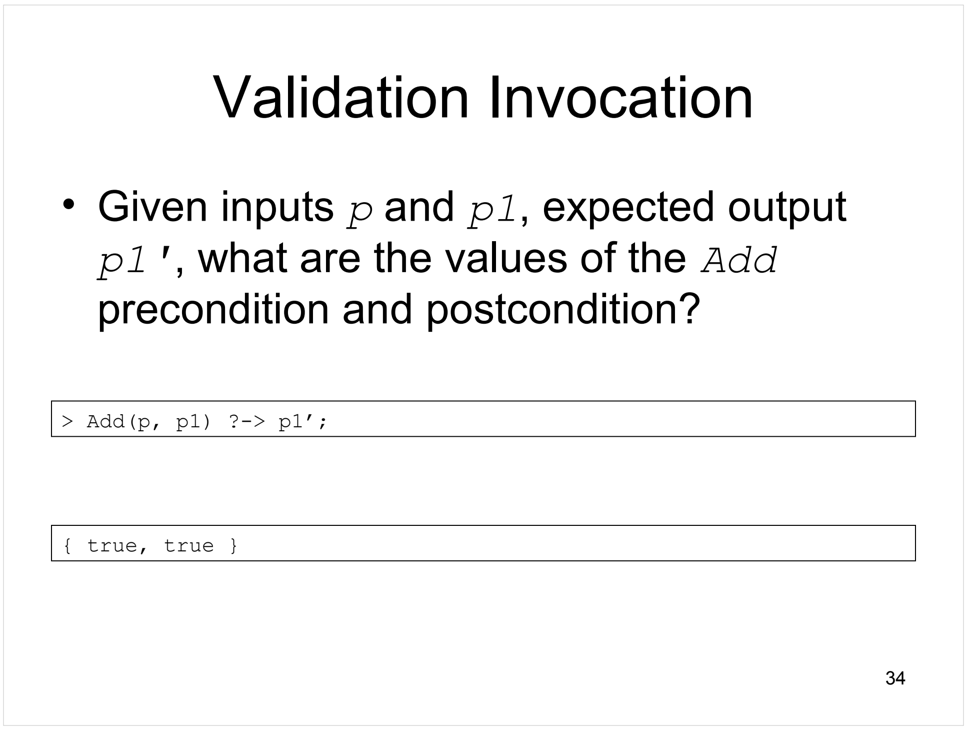
```
> p in pl;                          -- should be false
> not (p in pl);                    -- should be true
> not (p in pl');                   -- should be false

> p in pl’;                         -- should be true
```

Evaluating boolean expressions can be helpful, but it would be even handier to invoke an operation with sample input and output values directly. This kind of validation invocation can be characterized as follows:

Validation Invocation

- Given inputs *p* and *p1*, expected output *p1'*, what are the values of the *Add* precondition and postcondition?

```
> Add(p, p1) ?-> p1';
```



Validation Invocation

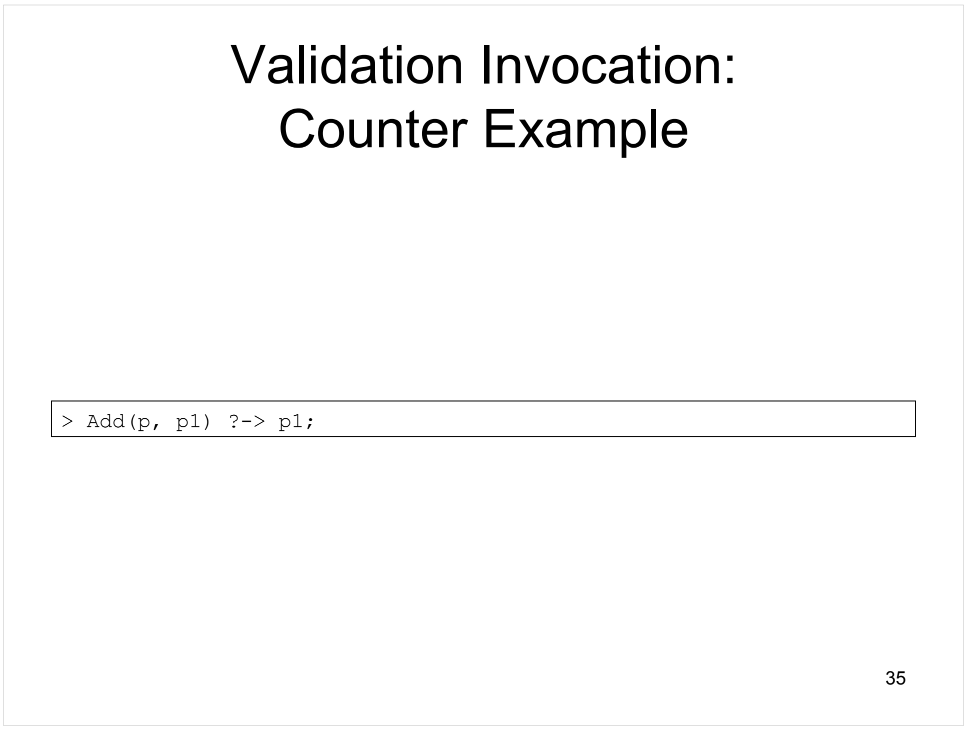
- Given inputs *p* and *p1*, expected output *p1'*, what are the values of the *Add* precondition and postcondition?

```
> Add(p, p1) ?-> p1';
```

```
{ true, true }
```

34

1. The first part of a validation invocation looks like a regular operation call
2. The question mark arrow is the validation operator
3. The output of true, true is the standard curly brace notation for a boolean two-tuple.
4. The first tuple field corresponds to a precondition evaluation result
5. The second tuple field corresponds to a postcondition evaluation result



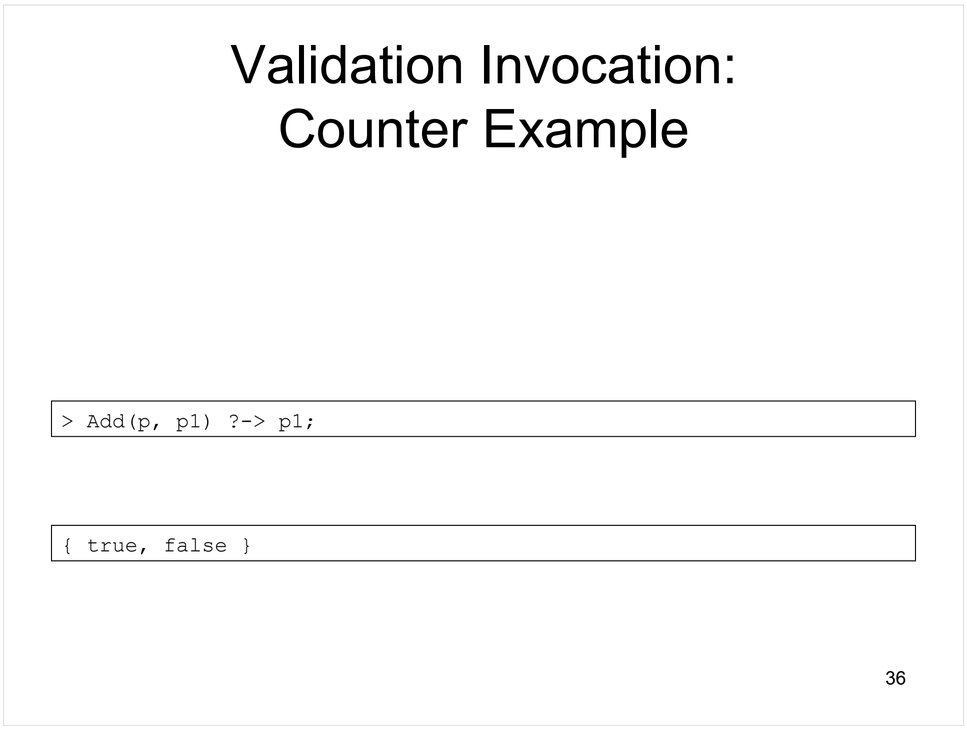
Validation Invocation: Counter Example

```
> Add(p, p1) ?-> p1;
```

35

Here we have a counter example.

Recall that p1 is a list with no elements, or a nil list. Add’s postcondition asserted that p must be in the output list. Since p is not in p1 – the list with no elements – the postcondition evaluates to false.



Validation Invocation: Counter Example

```
> Add(p, p1) ?-> p1;
```

```
{ true, false }
```

And so the two-tuple result comes back true, false since the postcondition evaluates to false for this set of inputs and outputs.

Expression Evaluation in FMSL

- Entails invoking an operator or operation and returning the calculated result
- Collection of built-in Boolean, arithmetic, tuple, and list expressions

Boolean Expression Examples

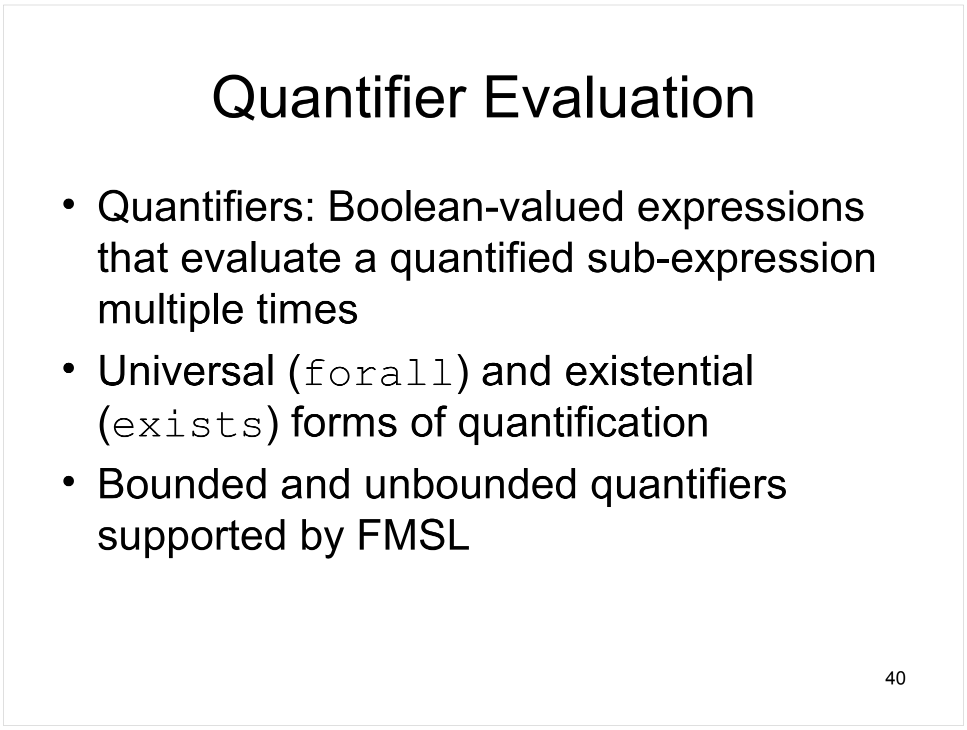
```
(*
 * Declare short value names for true and false.
 *)
val t:boolean = true;
val f:boolean = false;

(*
 * Boolean operator examples
 *)
> not t;                        -- evaluates t false
> t and f;                      -- evaluates t false
> t or f;                       -- evaluates to true
> t xor f;                      -- evaluates to true
> t => f;                       -- evaluates to false
> t <=> f;                      -- evaluates to true
```

Note the abbreviated keyword val in place of value. FMSL provides abbreviated versions of major key words.

Arithmetic Expression Example

```
(*
 * Declare and assign values to x, y
 *)
val x:real = 3.141592654;
val y:real = 2.718281828;

(*
 * Evaluate x divided by y and output the result
 *)
> x / y;
```

```
Output:

1.15573
```

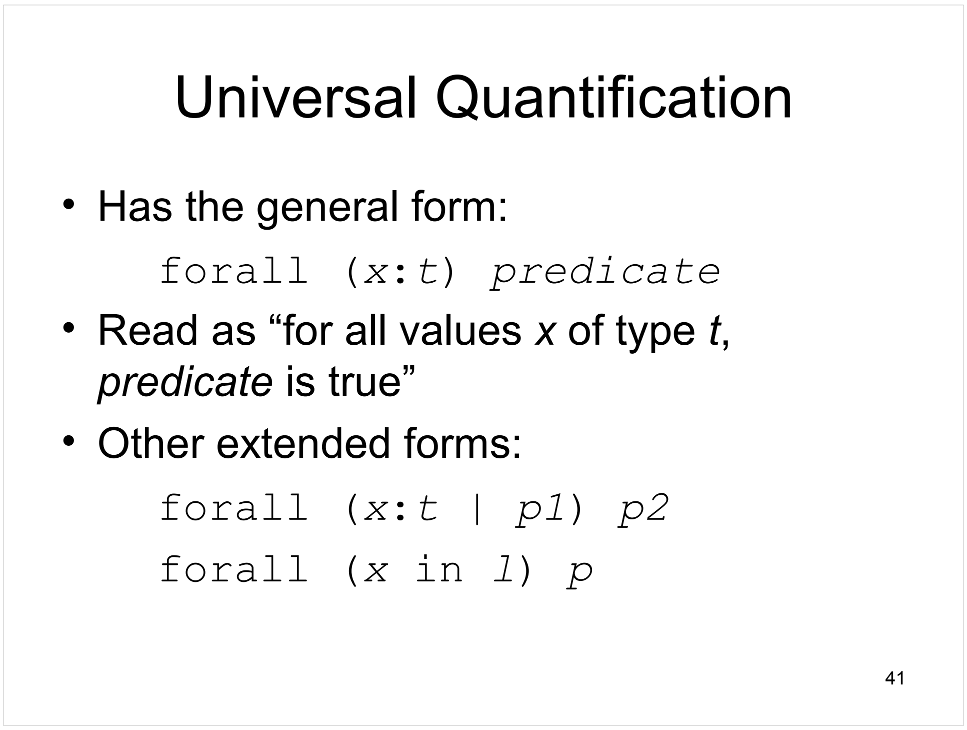


Quantifier Evaluation

- Quantifiers: Boolean-valued expressions that evaluate a quantified sub-expression multiple times
- Universal (`forall`) and existential (`exists`) forms of quantification
- Bounded and unbounded quantifiers supported by FMSL

A bounded quantifier ranges over a discrete set of values

An unbounded quantifier ranges over all of the values in a type of object. For example, for types grounded in integer, real, or string the quantifier range is unbounded.



Universal Quantification

- Has the general form:

```
forall (x:t) predicate
```

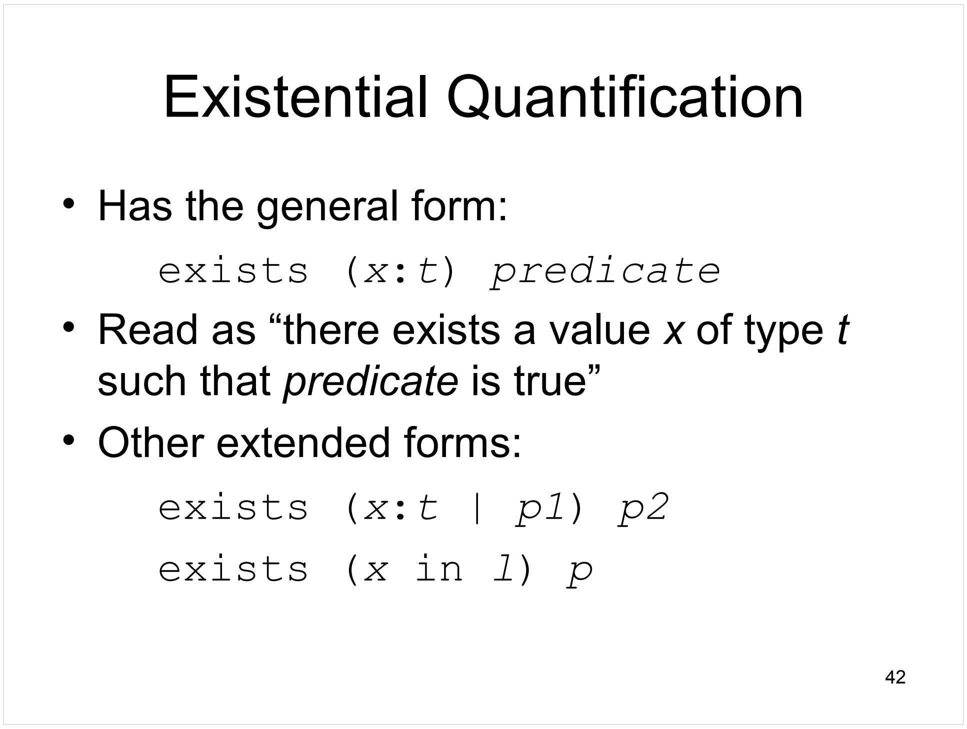
- Read as “for all values *x* of type *t*, *predicate* is true”
- Other extended forms:

```
forall (x:t | p1) p2
forall (x in l) p
```

41

For all x of type t, such that p1 is true, p2 is true

For all x in l, p is true



Existential Quantification

- Has the general form:

```
exists (x:t) predicate
```

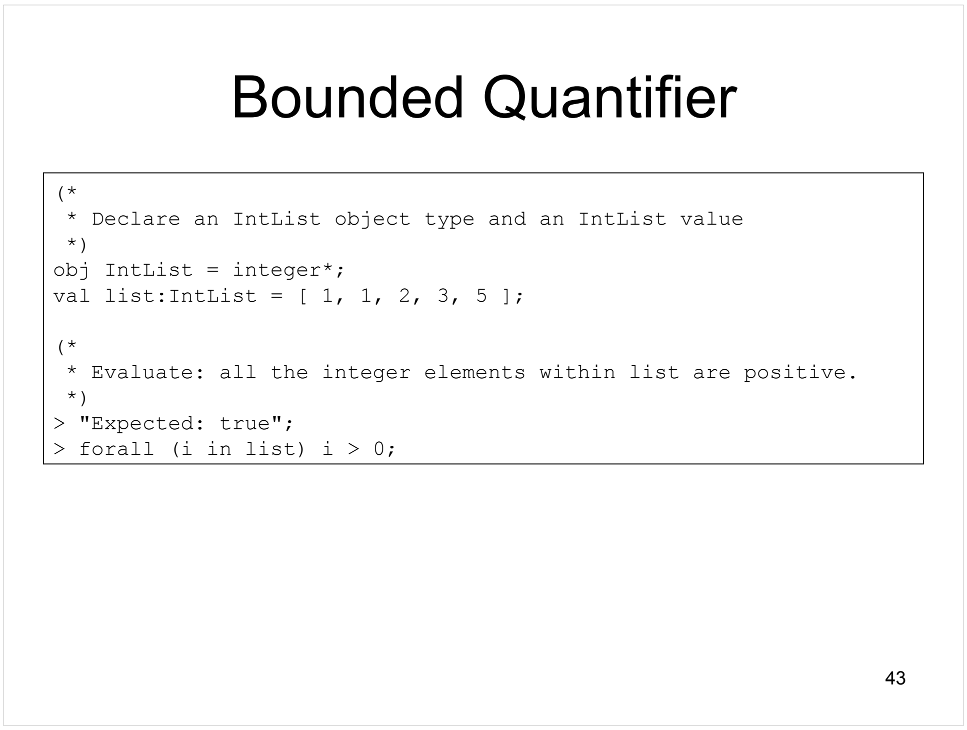
- Read as "there exists a value *x* of type *t* such that *predicate* is true"
- Other extended forms:

```
exists (x:t | p1) p2
exists (x in l) p
```

42

There exists a value x of type t, such that p1 is true and p2 is true

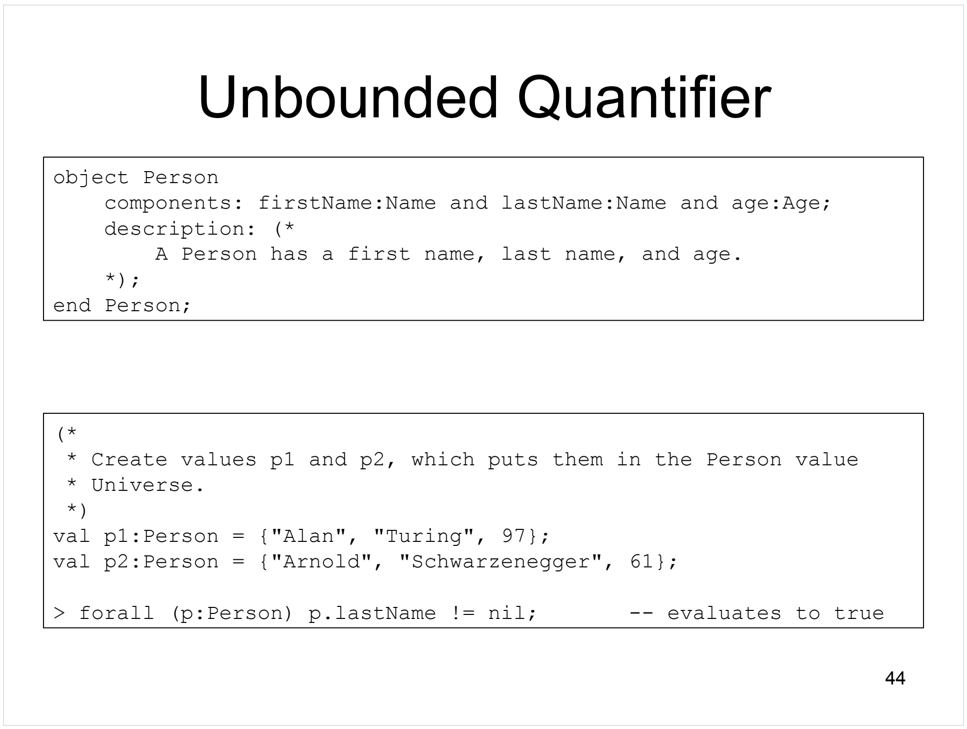
There exists a value x in l such that p is true



Bounded Quantifier

```
(*
 * Declare an IntList object type and an IntList value
 *)
obj IntList = integer*;
val list:IntList = [ 1, 1, 2, 3, 5 ];

(*
 * Evaluate: all the integer elements within list are positive.
 *)
> "Expected: true";
> forall (i in list) i > 0;
```

This example creates a list of integers, and uses a bounded universal form of quantification to evaluate whether all the integer elements are positive. Since all the integers – 1 1 2 3 5 – are positive, the quantifier expression evaluates to true.



Unbounded Quantifier

```
object Person
    components: firstName:Name and lastName:Name and age:Age;
    description: (*
        A Person has a first name, last name, and age.
    *);
end Person;
```

```
(*
 * Create values p1 and p2, which puts them in the Person value
 * Universe.
 *)
val p1:Person = {"Alan", "Turing", 97};
val p2:Person = {"Arnold", "Schwarzenegger", 61};

> forall (p:Person) p.lastName != nil;       -- evaluates to true
```

Recall the Person object definition from earlier on.

Declares two Person values

Evaluates unbounded quantifier to test that all the Person objects have non-nil last names

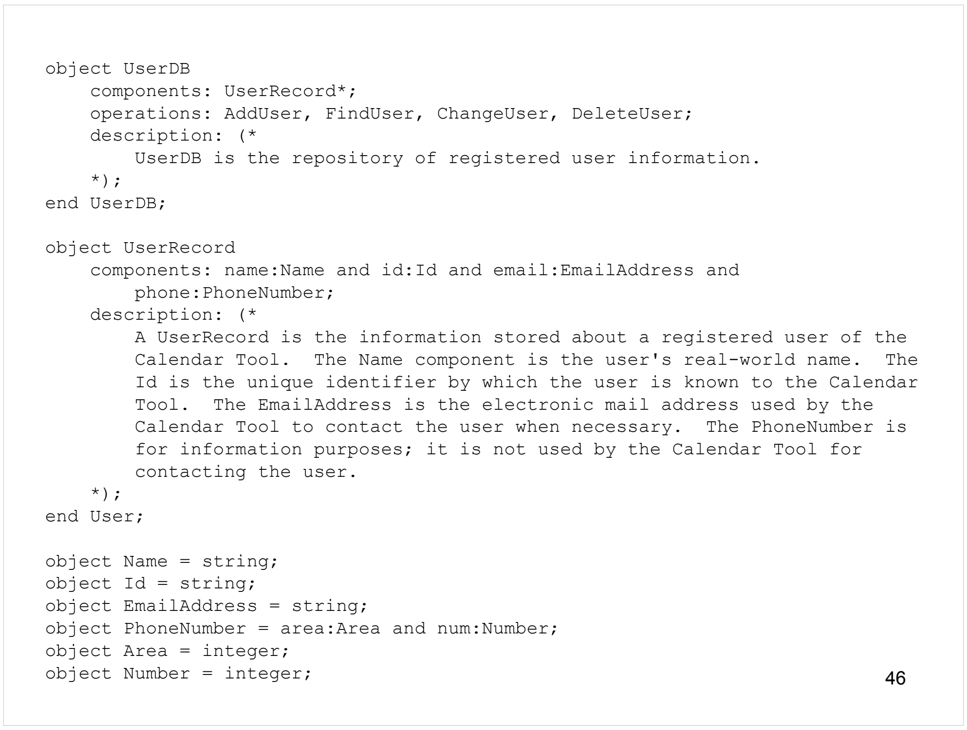
Since the two existing Person objects have non-nil last names, the result of the unbounded quantifier expression evaluation is true.

Conceptually, the universe of all values of type Person is unbounded since it consists of component types integer and string. A means must be established to execute the quantifier in bounded time. In FMSL, the universe for an unbounded quantifier consists of all values of the quantified type that have come into existence during a particular execution session.

I'll talk about more details later on in the presentation.

User Database Specification Example

- Pedagogical example for a distributed calendaring application
- Used for undergrad instruction at Cal Poly (CSC 308 – Gene Fisher)



```
object UserDB
    components: UserRecord*;
    operations: AddUser, FindUser, ChangeUser, DeleteUser;
    description: (*
        UserDB is the repository of registered user information.
    *);
end UserDB;

object UserRecord
    components: name:Name and id:Id and email:EmailAddress and
        phone:PhoneNumber;
    description: (*
        A UserRecord is the information stored about a registered user of the
        Calendar Tool.  The Name component is the user's real-world name.  The
        Id is the unique identifier by which the user is known to the Calendar
        Tool.  The EmailAddress is the electronic mail address used by the
        Calendar Tool to contact the user when necessary.  The PhoneNumber is
        for information purposes; it is not used by the Calendar Tool for
        contacting the user.
    *);
end User;

object Name = string;
object Id = string;
object EmailAddress = string;
object PhoneNumber = area:Area and num:Number;
object Area = integer;
object Number = integer;
```

These definitions describe individual components of a user record and a user record database.

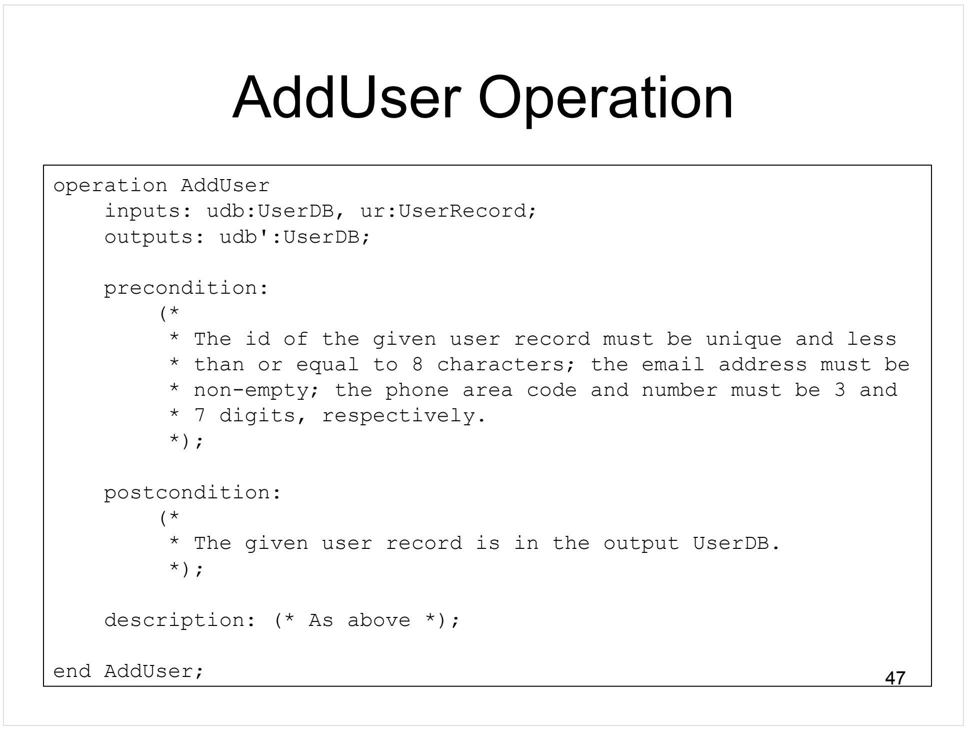
UserDB is a list of UserRecord objects

UserRecord has components name, id, email, and phone

Individual scalar components are defined below.

The following example demonstrates the utility of FMSL's operation validation capabilities, as it follows the incremental development of the specification of the AddUser operation.

First we'll start with a precondition and postcondition described in plain English.

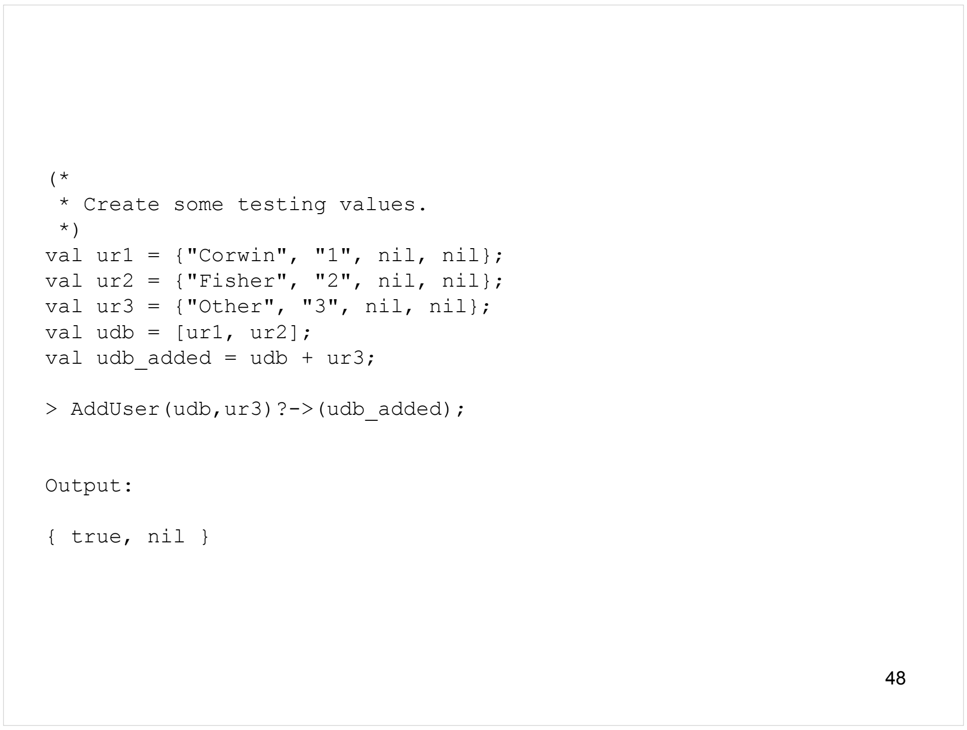


AddUser Operation

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition:
        (*
         * The id of the given user record must be unique and less
         * than or equal to 8 characters; the email address must be
         * non-empty; the phone area code and number must be 3 and
         * 7 digits, respectively.
         *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *);

    description: (* As above *);

end AddUser;
```

- READ THE PRECONDITION AND POSTCONDITION -

Although precondition and postcondition are described with plain English comments, the AddUser operation already is executable through the validation operator.



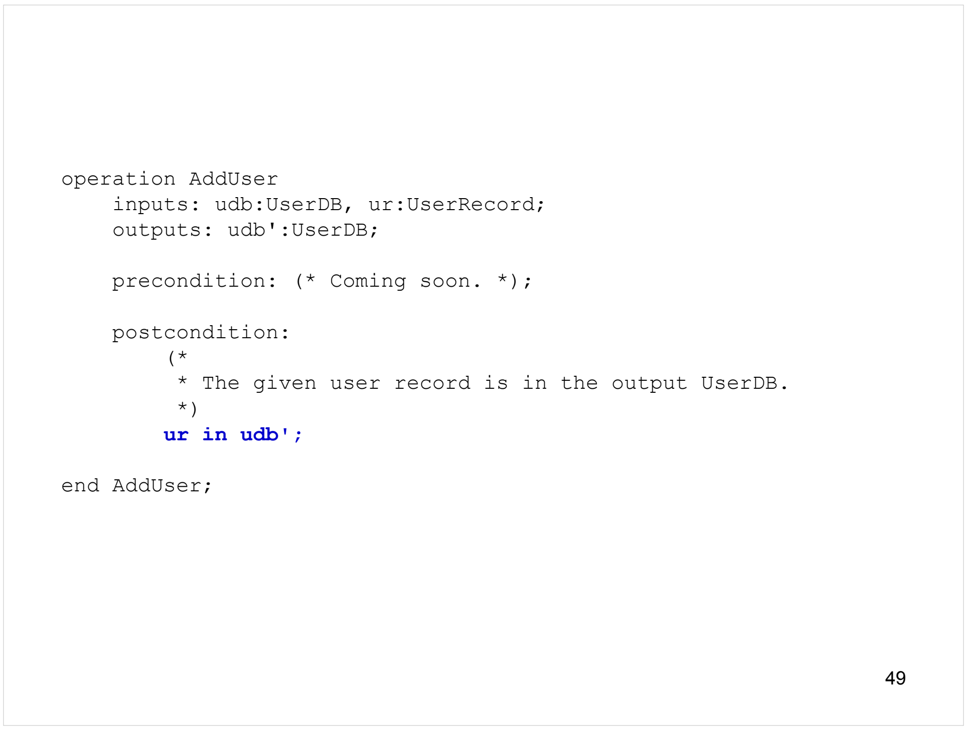
```
(*
 * Create some testing values.
 *)
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val udb = [ur1, ur2];
val udb_added = udb + ur3;

> AddUser(udb,ur3)?->(udb_added);


Output:

{ true, nil }
```

The code here creates three sample user record values, an initial user db value consisting of elements ur1 and ur2, and an expected output user db that consists of udb with ur3 concatenated to it.

Finally the code invokes the validation operator to test AddUser with these values. What would be the result of this evaluation?

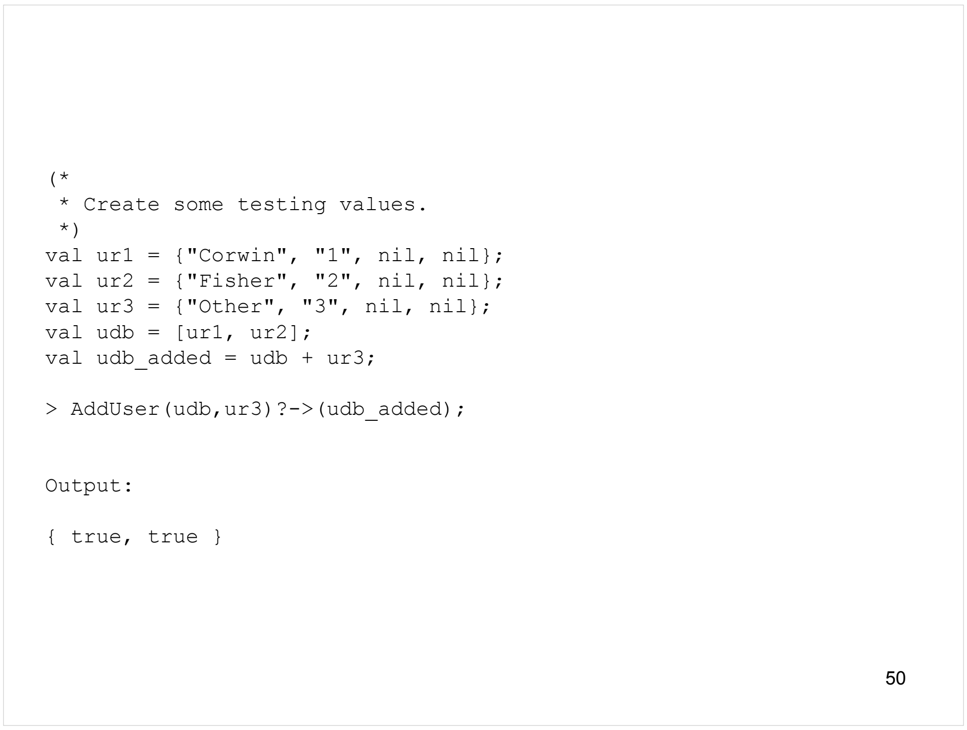
Recall that both the precondition and postcondition are described in plain English. In FMSL, by definition, if there is no precondition – that is, if there are no formally defined entry criteria – the precondition evaluation result is true. Also by definition, if there is no postcondition then the postcondition evaluation result is set nil. So, the two-tuple result you can see consists of true, nil.

While executability isn’t especially interesting in this example, more interesting examples follow.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    precondition: (* Coming soon. *);

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        ur in udb';

end AddUser;
```

The postcondition here describes the essence of an additive collection operation: that the given user record is in the output user DB

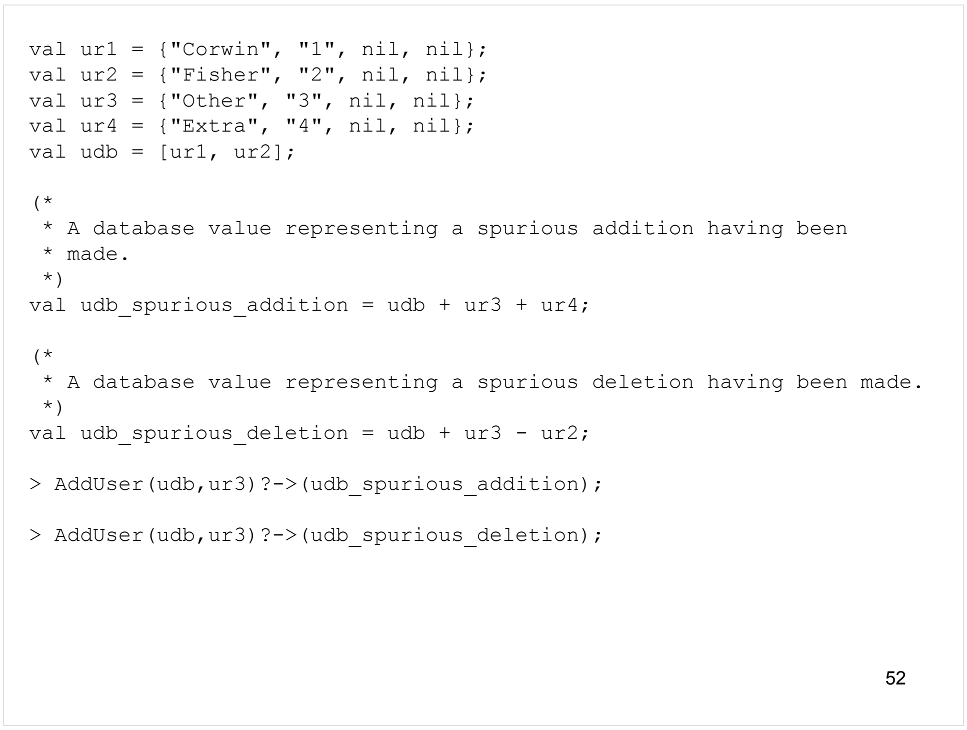


```
(*
 * Create some testing values.
 *)
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val udb = [ur1, ur2];
val udb_added = udb + ur3;

> AddUser(udb,ur3)?->(udb_added);


Output:

{ true, true }
```

Here we see the same testing values we used in the previous example. In this case the postcondition tests whether the given user record is in the output UserDB, which it is. So, the resulting two-tuple evaluation of this validation operator invocation is true, true.

Fundamental Question

- Are the preconditions and postconditions strong enough?
- In the AddUser example, the precondition is non-existent and thus it’s maximally weak
- To test postcondition strength, we can use the validation operator to run some example inputs and outputs against AddUser



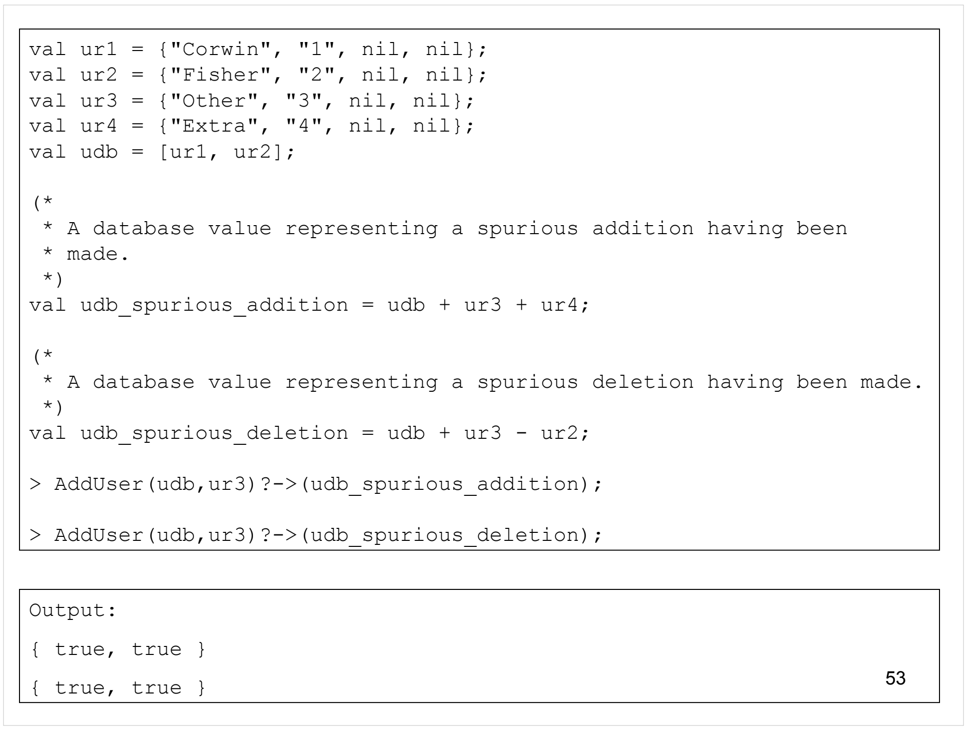
```
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val ur4 = {"Extra", "4", nil, nil};
val udb = [ur1, ur2];

(*
 * A database value representing a spurious addition having been
 * made.
 *)
val udb_spurious_addition = udb + ur3 + ur4;

(*
 * A database value representing a spurious deletion having been made.
 *)
val udb_spurious_deletion = udb + ur3 - ur2;

> AddUser(udb,ur3)?->(udb_spurious_addition);

> AddUser(udb,ur3)?->(udb_spurious_deletion);
```

52

Here we see two uses of the validation operator.

In the first we test whether the postcondition guards against spurious additions, since the output userdb contains an extra user record. In the second we test whether the postcondition guards against spurious deletions, the output userdb is missing a record that should be there.

Logically, we don't want AddUser to accept spurious additions or deletions, so in both cases we expect the validation operator evaluations to be two-tuples of true, false.

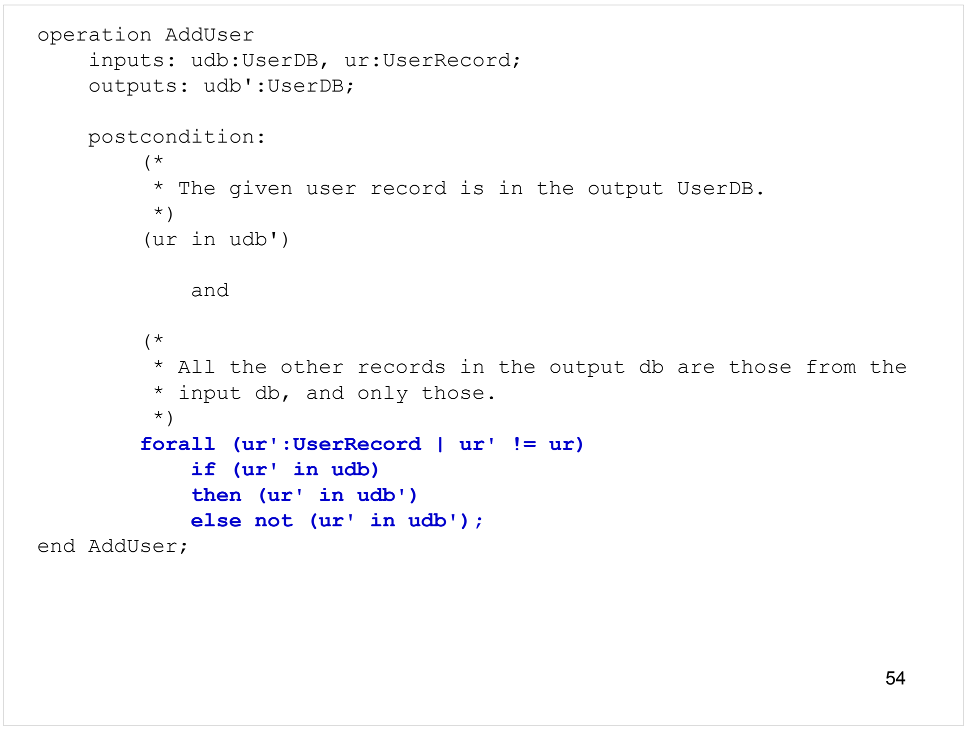


```
val ur1 = {"Corwin", "1", nil, nil};
val ur2 = {"Fisher", "2", nil, nil};
val ur3 = {"Other", "3", nil, nil};
val ur4 = {"Extra", "4", nil, nil};
val udb = [ur1, ur2];

(*
 * A database value representing a spurious addition having been
 * made.
 *)
val udb_spurious_addition = udb + ur3 + ur4;

(*
 * A database value representing a spurious deletion having been made.
 *)
val udb_spurious_deletion = udb + ur3 - ur2;

> AddUser(udb,ur3)?->(udb_spurious_addition);

> AddUser(udb,ur3)?->(udb_spurious_deletion);
```

```
Output:
{ true, true }
{ true, true }
```

Notice that the two tuples each come back as true, true, which is not what we wanted. What that tells us is that the postcondition – that the given user record is in the output userdb – is not strong enough.

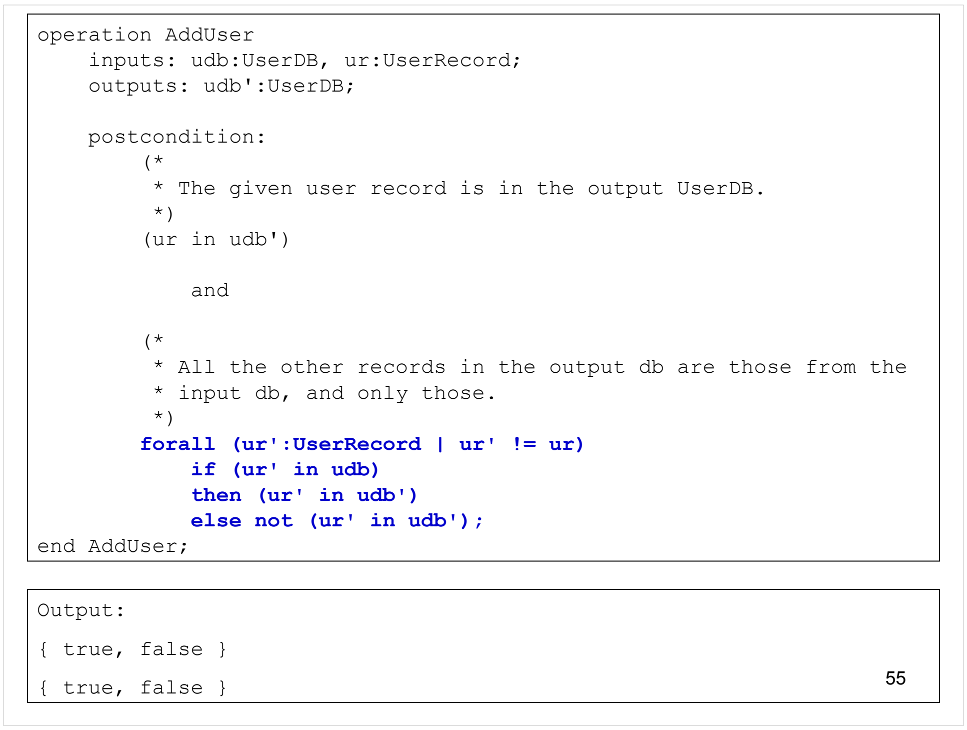
To strengthen the postcondition, we want to make sure that all the other records in the output db are those from the input db, and only those. We can describe that with the following definition of AddUser.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        (ur in udb')

            and

        (*
         * All the other records in the output db are those from the
         * input db, and only those.
         *)
        forall (ur':UserRecord | ur' != ur)
            if (ur' in udb)
            then (ur' in udb')
            else not (ur' in udb');
end AddUser;
```

Here we see the additional statements that strengthen the postcondition, [READ POSTCONDITION BLUE OUT LOUD]

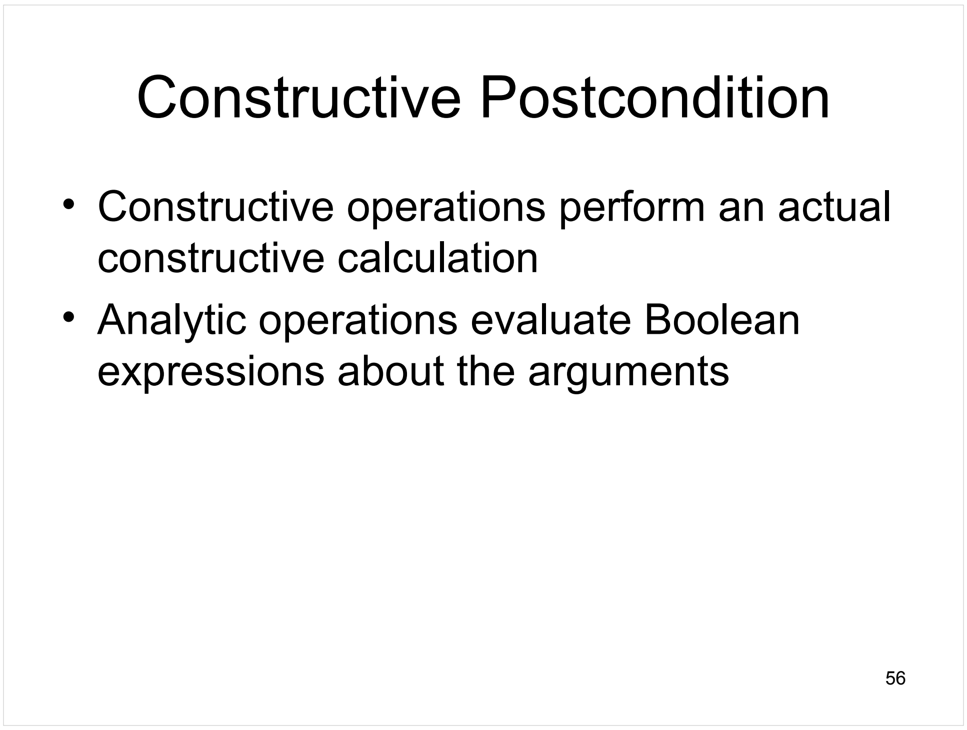
When we re-run the validation operator invocations with this updated AddUser definition, we see some good results.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        (ur in udb')

            and

        (*
         * All the other records in the output db are those from the
         * input db, and only those.
         *)
        forall (ur':UserRecord | ur' != ur)
            if (ur' in udb)
            then (ur' in udb')
            else not (ur' in udb');
end AddUser;
```

```
Output:
{ true, false }
{ true, false }
```

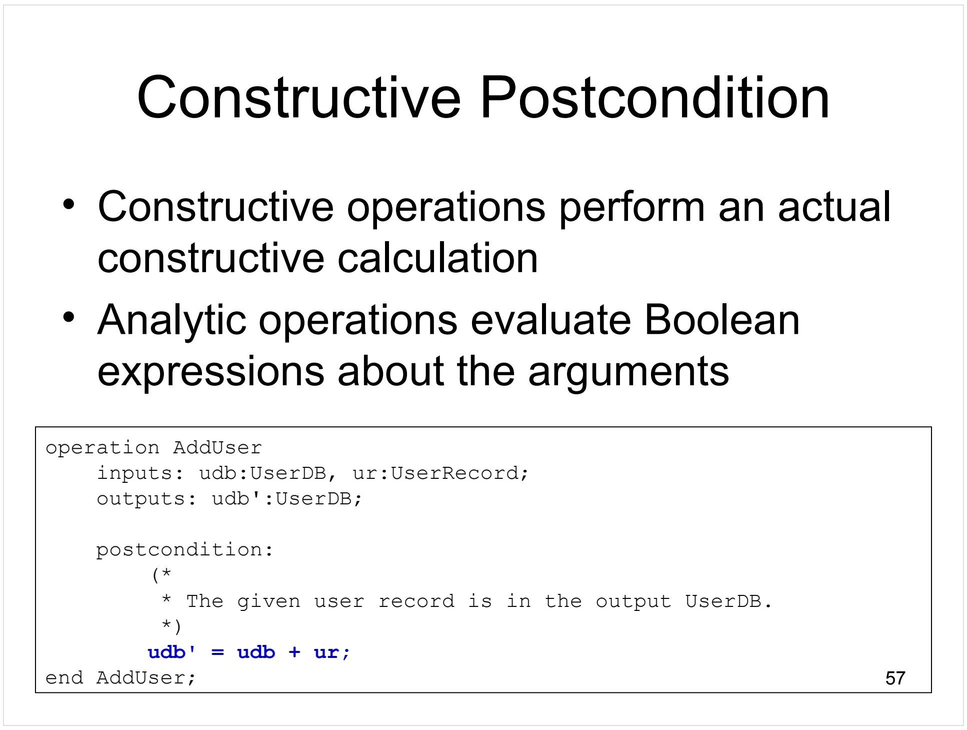
By good results I mean that the validation operator invocation rejects the spurious addition and deletion in the output userdb.



Constructive Postcondition

- Constructive operations perform an actual constructive calculation
- Analytic operations evaluate Boolean expressions about the arguments

In some cases a precondition or postcondition that utilizes constructive operations may be clearer than its corresponding analytic operation-based counterpart



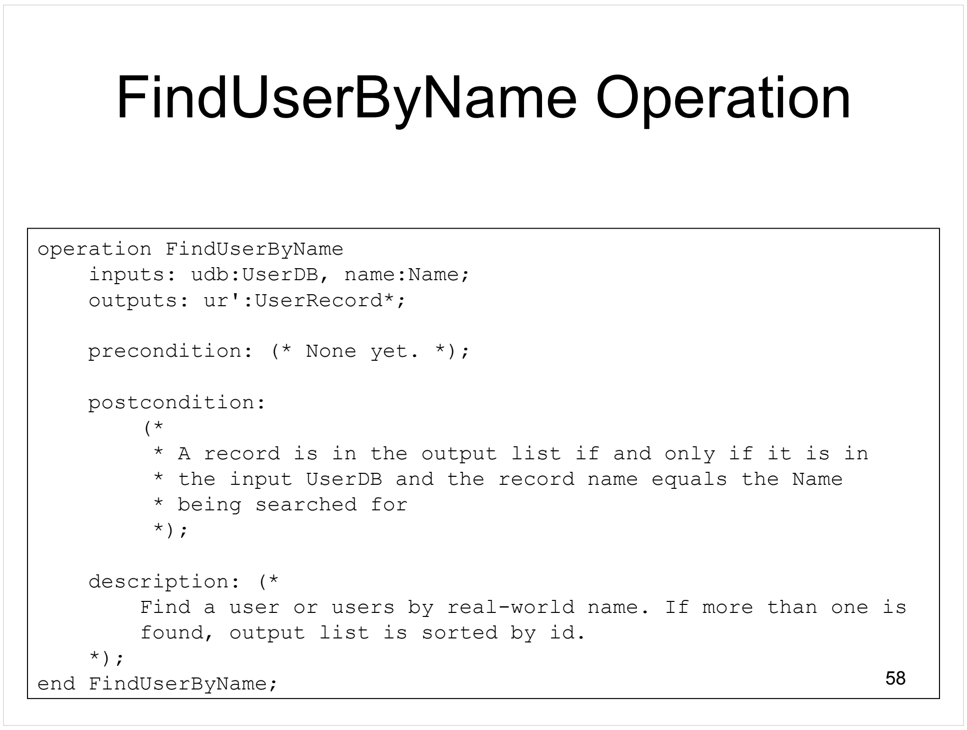
Constructive Postcondition

- Constructive operations perform an actual constructive calculation
- Analytic operations evaluate Boolean expressions about the arguments

```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;

    postcondition:
        (*
         * The given user record is in the output UserDB.
         *)
        udb' = udb + ur;
end AddUser;
```

57

The next example follows the iterative development of the specification of the FindUserByName operation.

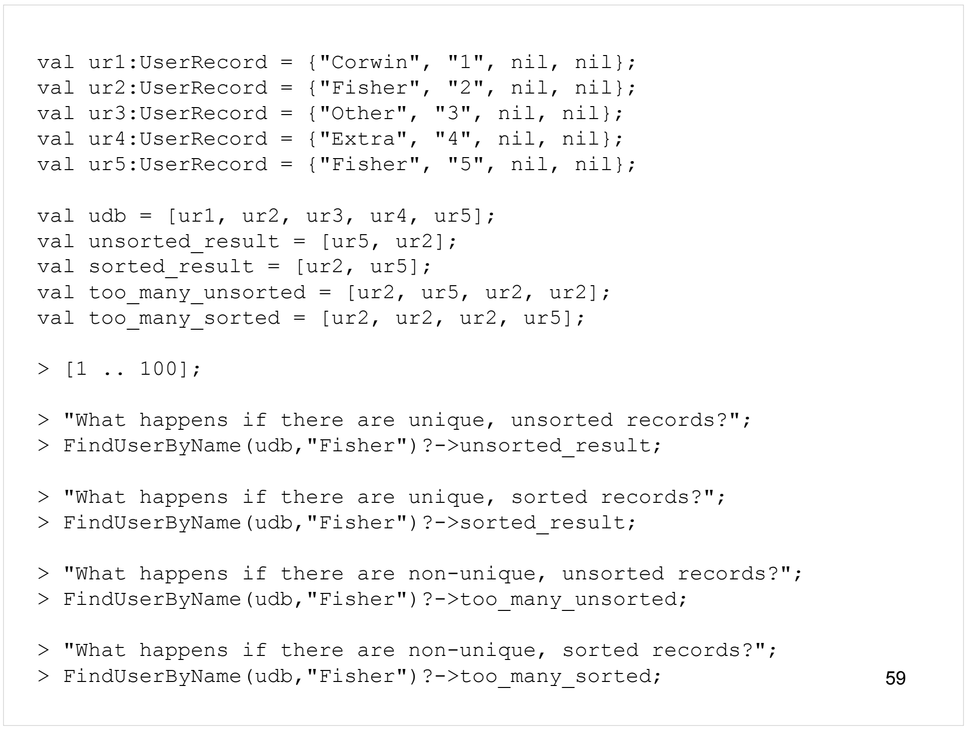


FindUserByName Operation

```
operation FindUserByName
    inputs: udb:UserDB, name:Name;
    outputs: ur':UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * A record is in the output list if and only if it is in
         * the input UserDB and the record name equals the Name
         * being searched for
         *);

    description: (*
        Find a user or users by real-world name. If more than one is
        found, output list is sorted by id.
    *);
end FindUserByName;
```

searches through the user database and returns records with names that match the given name input argument.

Just like with the AddUser example, we now have an operation that is well-defined enough to the point where it’s executable.



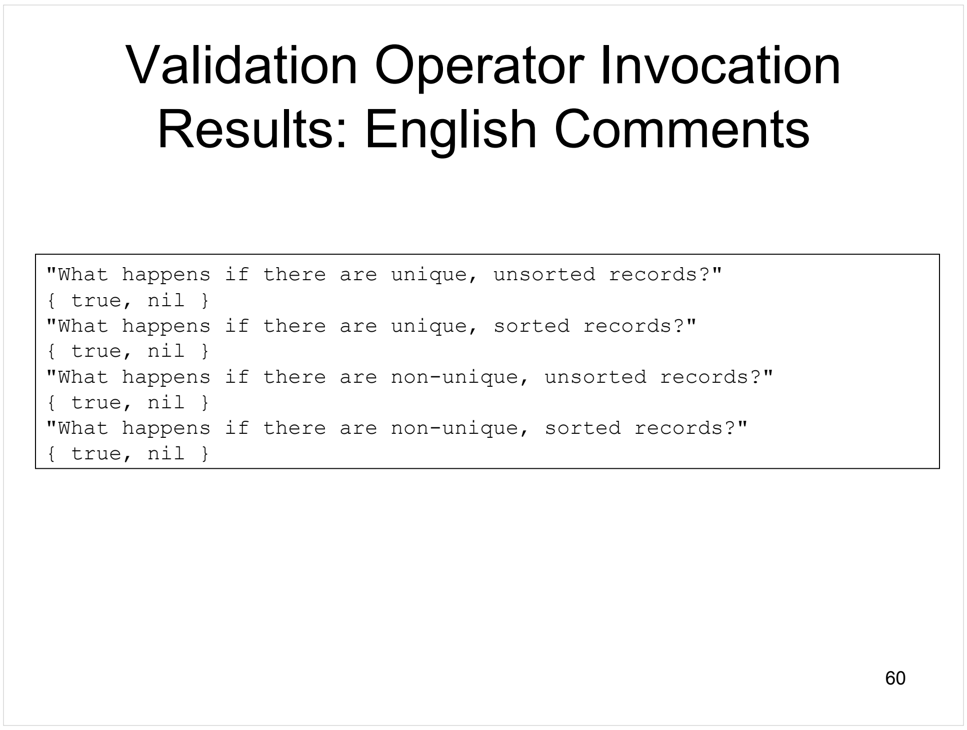
```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

Here’s a test file that creates some sample user records and other input and output values.

I’ll quickly point out that the list construction from 1 to 100 populates the integer universe, which is a topic I’ll discuss a little later.

Finally you see some validation operator invocations that search udb for “Fisher.” Note that in each of these tests the output result is different:

6. First we have an unsorted list
7. Second we have a sorted list
8. Third we have an unsorted list where multiple matches appear
9. Lastly we have a sorted list where multiple matches appear

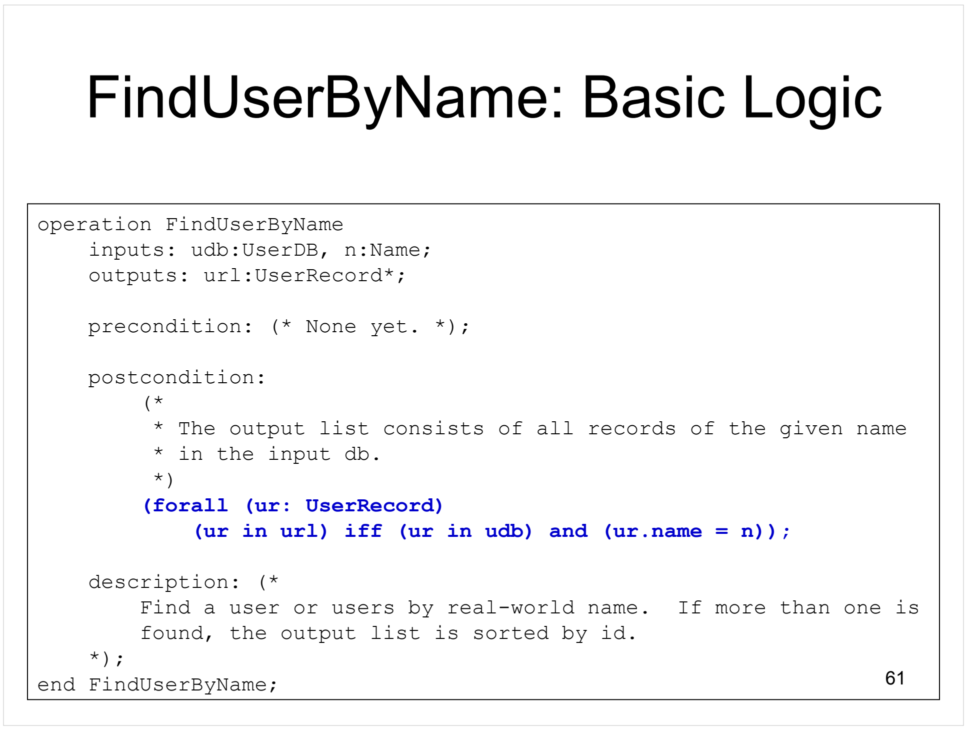


Validation Operator Invocation Results: English Comments

```
"What happens if there are unique, unsorted records?"
{ true, nil }
"What happens if there are unique, sorted records?"
{ true, nil }
"What happens if there are non-unique, unsorted records?"
{ true, nil }
"What happens if there are non-unique, sorted records?"
{ true, nil }
```

60

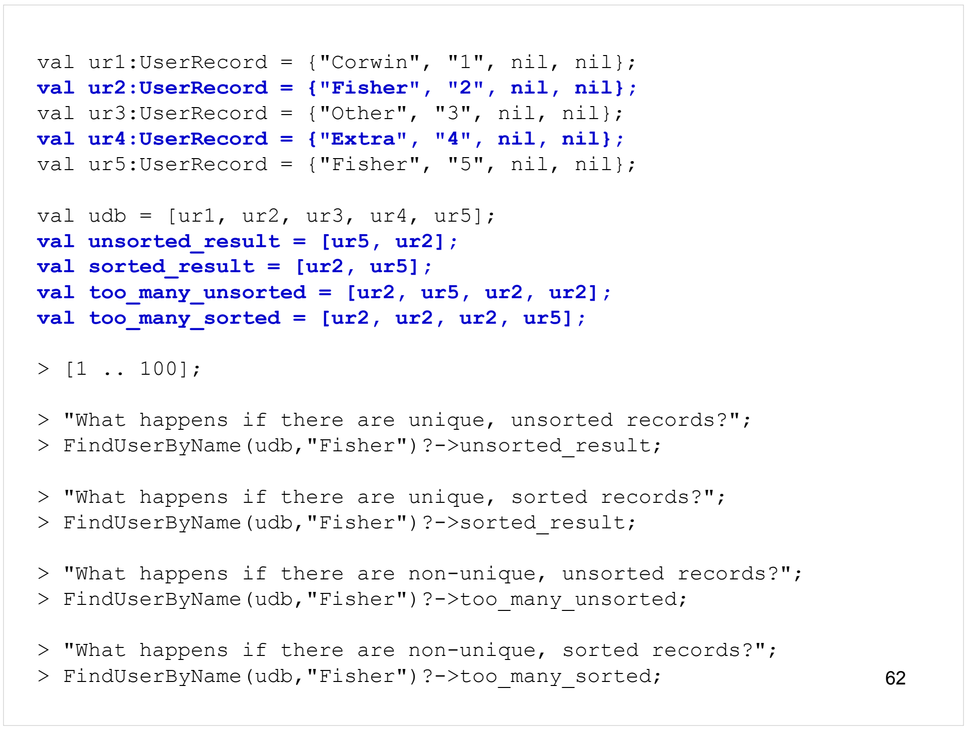
As expected, since there is no precondition defined then all the two-tuple precondition components are true. Likewise, since there is no postcondition then all the postcondition components are nil.



FindUserByName: Basic Logic

```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n));

    description: (*
        Find a user or users by real-world name.  If more than one is
        found, the output list is sorted by id.
    *);
end FindUserByName;
```

For all user records ur, ur is in the output list if and only if it’s in the input user db and the name matches the given name parameter.

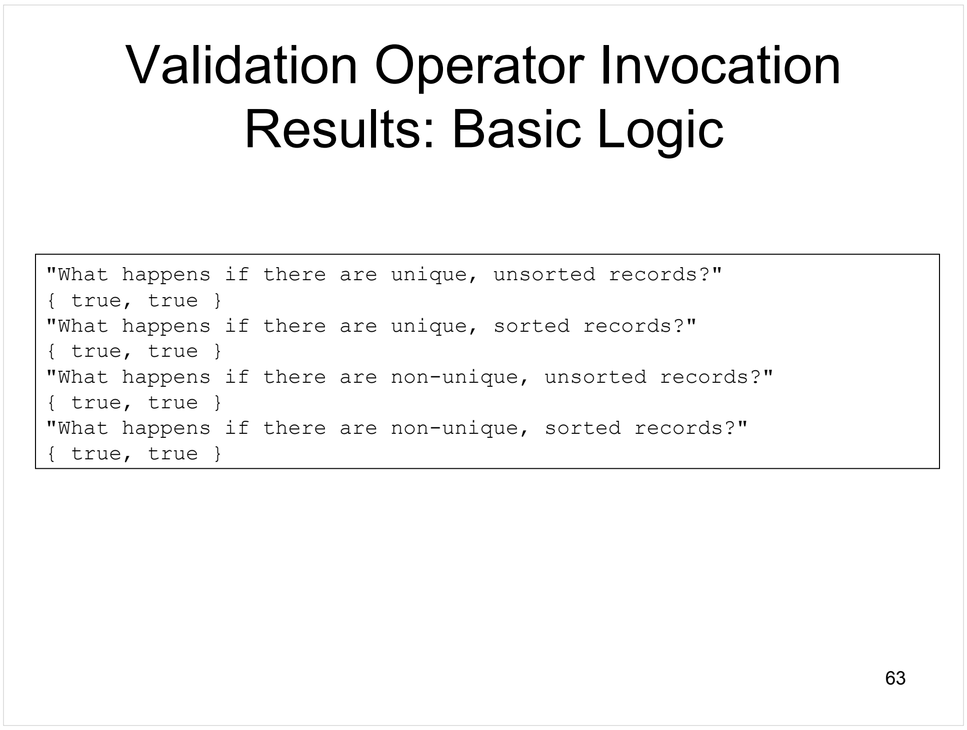


```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

62

Here’s the test file again. Notice that in all the tests we’re searching for records that have the name “Fisher”. Notice also that the test output user record lists all consist of some combination of the records with name of “Fisher,” which are ur2 and ur5.

Our postcondition should evaluate to true so long as all the records in the output list are in the input user db and have a name that matches “Fisher.” so, we expect all the postconditions to evaluate to true.

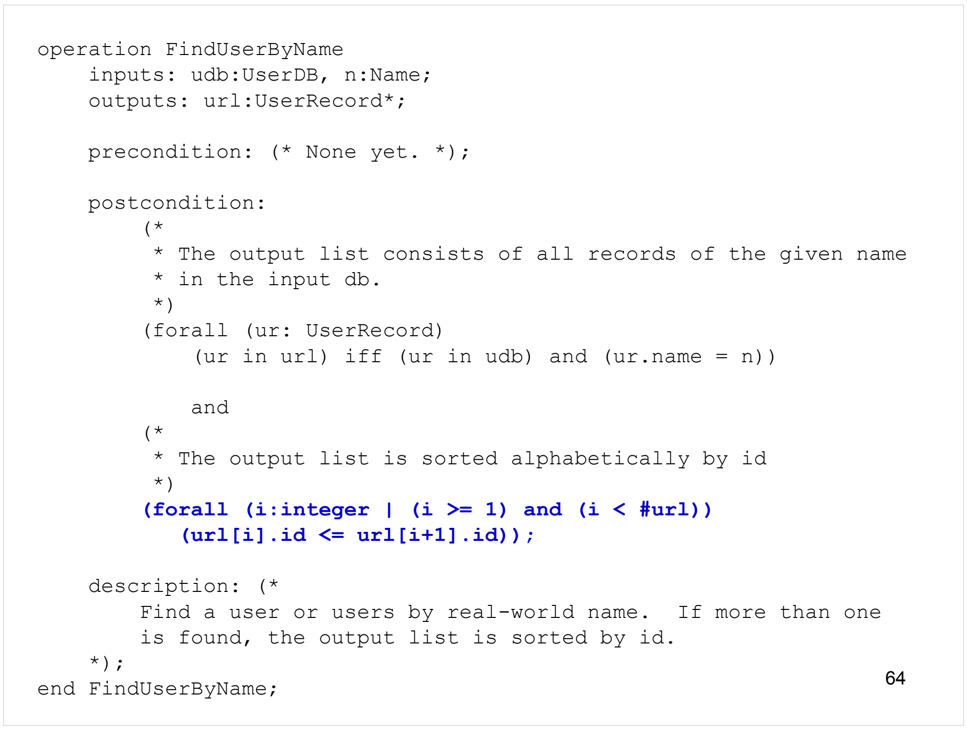


Validation Operator Invocation Results: Basic Logic

```
"What happens if there are unique, unsorted records?"
{ true, true }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, true }
"What happens if there are non-unique, sorted records?"
{ true, true }
```

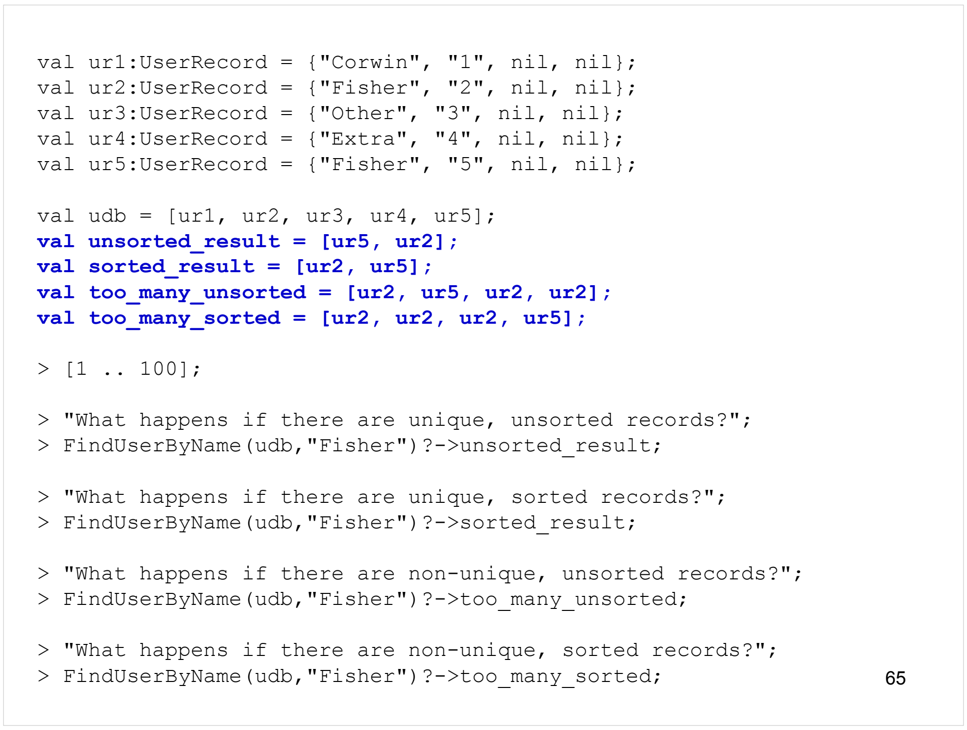
63

As expected, all the validation operator invocation two-tuples are true, true.



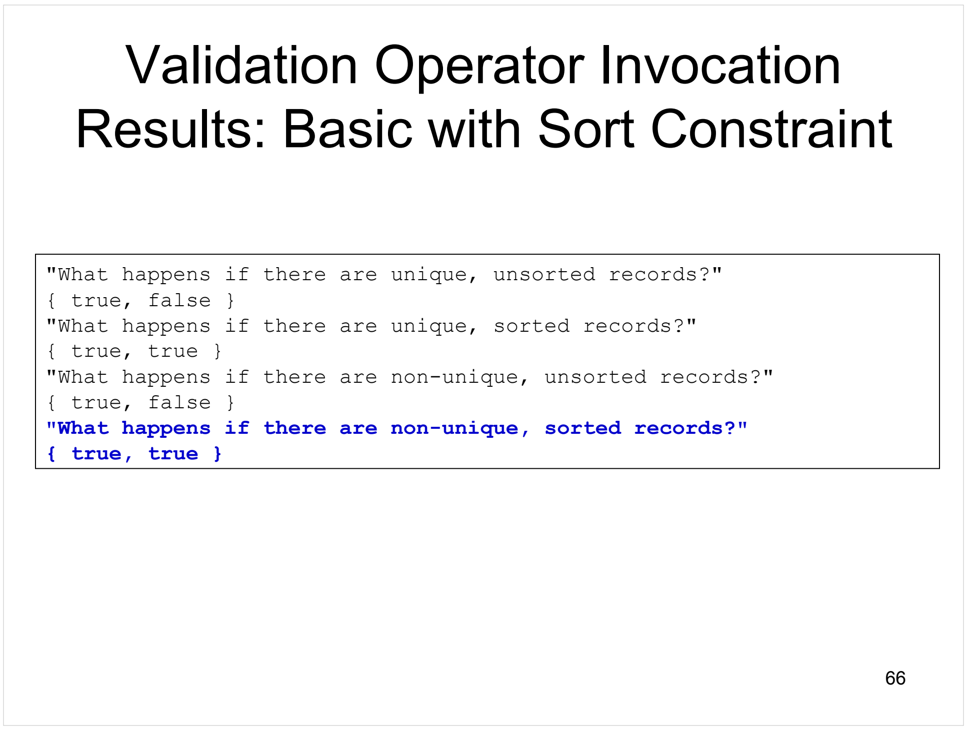
```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and
        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id <= url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one
        is found, the output list is sorted by id.
    *);
end FindUserByName;
```

This example specification of FindUserByName is a little more interesting, since it has a sort constraint in that the output user record list should be sorted by id.



```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

Now we expect the postcondition field in the resulting two-tuple to be false in the unsorted cases; otherwise, it should be true.

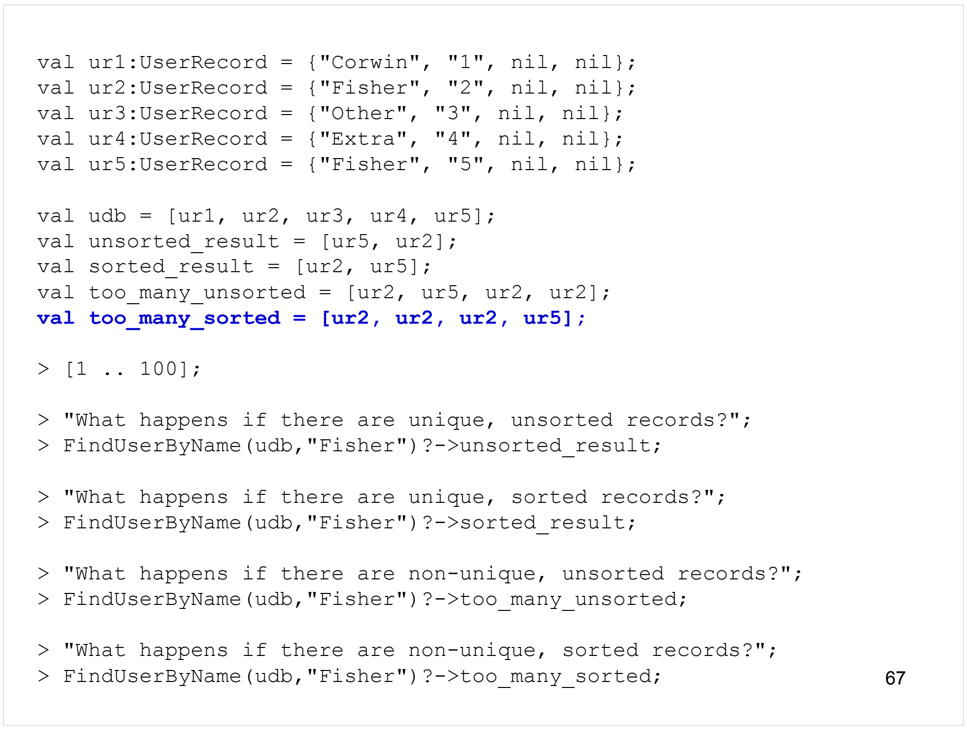


Validation Operator Invocation Results: Basic with Sort Constraint

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, true }
```

66

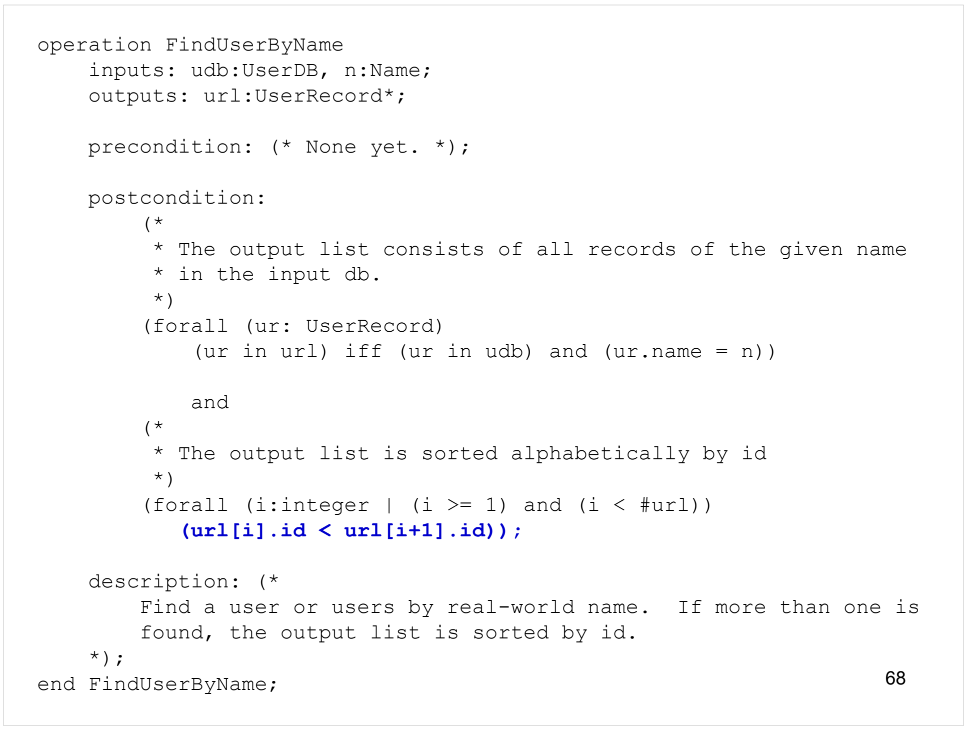
I want to highlight here the last example, where the postcondition field shows true. Note that the test user record list here has records sorted properly, except there are multiple occurrences of ur2.



```
val ur1:UserRecord = {"Corwin", "1", nil, nil};
val ur2:UserRecord = {"Fisher", "2", nil, nil};
val ur3:UserRecord = {"Other", "3", nil, nil};
val ur4:UserRecord = {"Extra", "4", nil, nil};
val ur5:UserRecord = {"Fisher", "5", nil, nil};

val udb = [ur1, ur2, ur3, ur4, ur5];
val unsorted_result = [ur5, ur2];
val sorted_result = [ur2, ur5];
val too_many_unsorted = [ur2, ur5, ur2, ur2];
val too_many_sorted = [ur2, ur2, ur2, ur5];

> [1 .. 100];

> "What happens if there are unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->unsorted_result;

> "What happens if there are unique, sorted records?";
> FindUserByName(udb,"Fisher")?->sorted_result;

> "What happens if there are non-unique, unsorted records?";
> FindUserByName(udb,"Fisher")?->too_many_unsorted;

> "What happens if there are non-unique, sorted records?";
> FindUserByName(udb,"Fisher")?->too_many_sorted;
```

67

It’s this issue of cardinality that previously had not been covered in the CSC 308 example. That is, the example specification of FindUser allowed multiple occurrences of records, which was unintended behavior. Going through examples for this thesis revealed that the postcondition was not strong enough, and it can be strengthened with a simple change (that may save Gene from some future embarrassment).

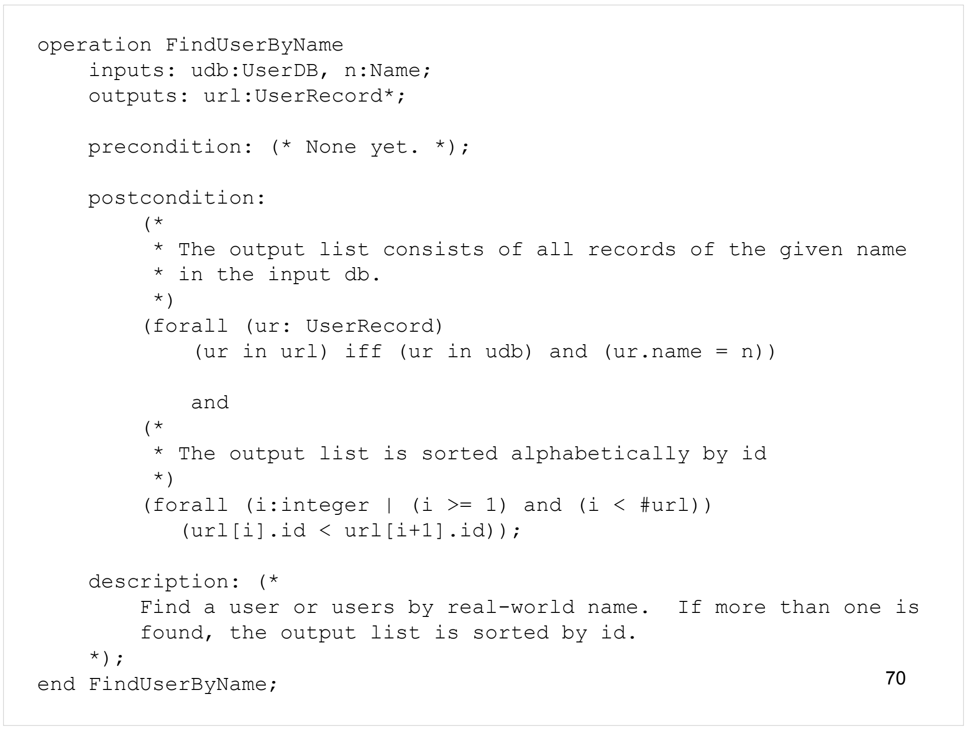


```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and
        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id < url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one is
        found, the output list is sorted by id.
    *);
end FindUserByName;
```

The simple change can be seen here where we use the less than operator, instead of less than or equal, to test sort order. This change forces uniqueness in the output user record list. When we re-run the tests against this FindUser operation with a strengthened postcondition, we should only one postcondition field be true: where the output list is sorted and the elements are unique.

Validation Operator Invocation Results: Strengthened Logic

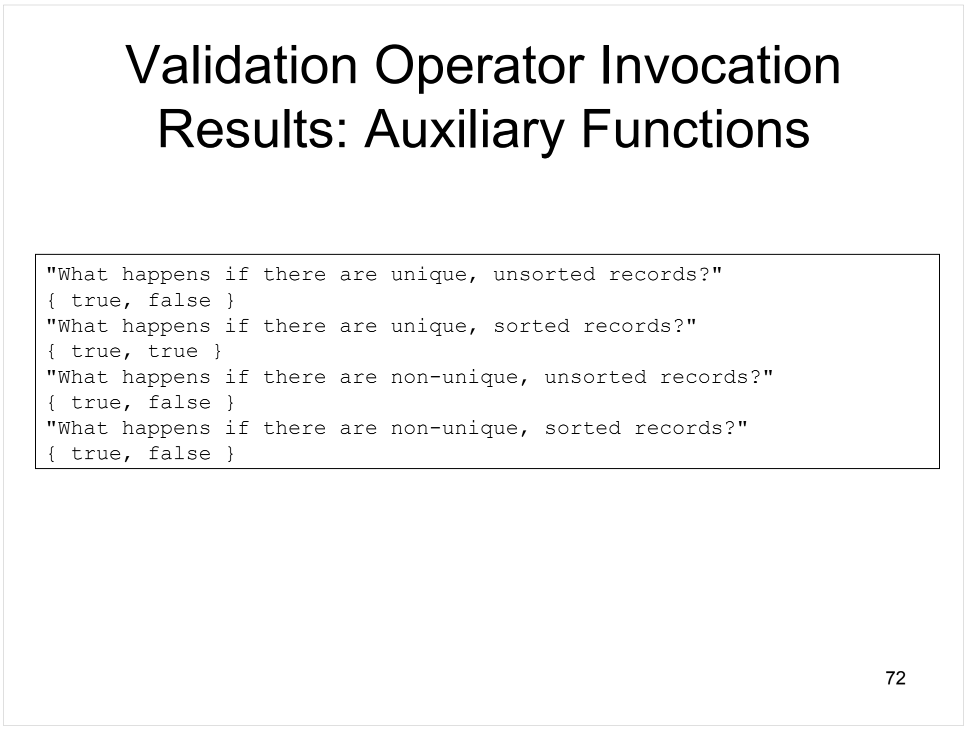
```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, false }
```



```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;

    precondition: (* None yet. *);

    postcondition:
        (*
         * The output list consists of all records of the given name
         * in the input db.
         *)
        (forall (ur: UserRecord)
            (ur in url) iff (ur in udb) and (ur.name = n))

            and
        (*
         * The output list is sorted alphabetically by id
         *)
        (forall (i:integer | (i >= 1) and (i < #url))
           (url[i].id < url[i+1].id));

    description: (*
        Find a user or users by real-world name.  If more than one is
        found, the output list is sorted by id.
    *);
end FindUserByName;
```

While we discovered this specification to be strong enough, some might argue that it’s difficult to read. To improve readability, FMSL allows the user to create auxiliary functions that can be used in preconditions and postconditions.

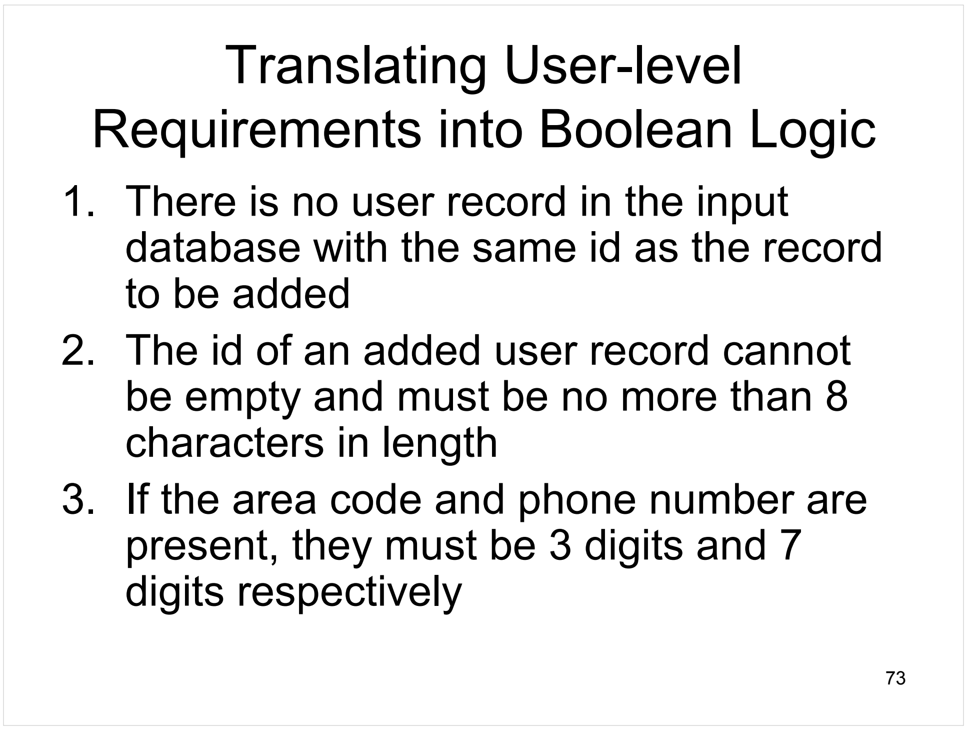
```
operation FindUserByName
    inputs: udb:UserDB, n:Name;
    outputs: url:UserRecord*;
    postcondition:
        RecordsFound(udb,n,url)
            and
        SortedById(url);
end FindUserByName;

function RecordsFound(udb:UserDB, n:Name, url:UserRecord*) =
    (*
     * The output list consists of all records of the given name in
     * the input db.
     *)
    (forall (ur' in url)
       (ur' in udb)
          and
       (ur'.name = n));

function SortedById(url:UserRecord*) =
    (*
     * The output list is sorted alphabetically by id.
     *)
        (if (#url > 1) then
            (forall (i in [1..(#url - 1)])
               url[i].id < url[i+1].id)
         else true);
```



Validation Operator Invocation Results: Auxiliary Functions

```
"What happens if there are unique, unsorted records?"
{ true, false }
"What happens if there are unique, sorted records?"
{ true, true }
"What happens if there are non-unique, unsorted records?"
{ true, false }
"What happens if there are non-unique, sorted records?"
{ true, false }
```

As we’d hoped, this perhaps easier-to-read specification was equivalent to the FindUser with strengthened postcondition.



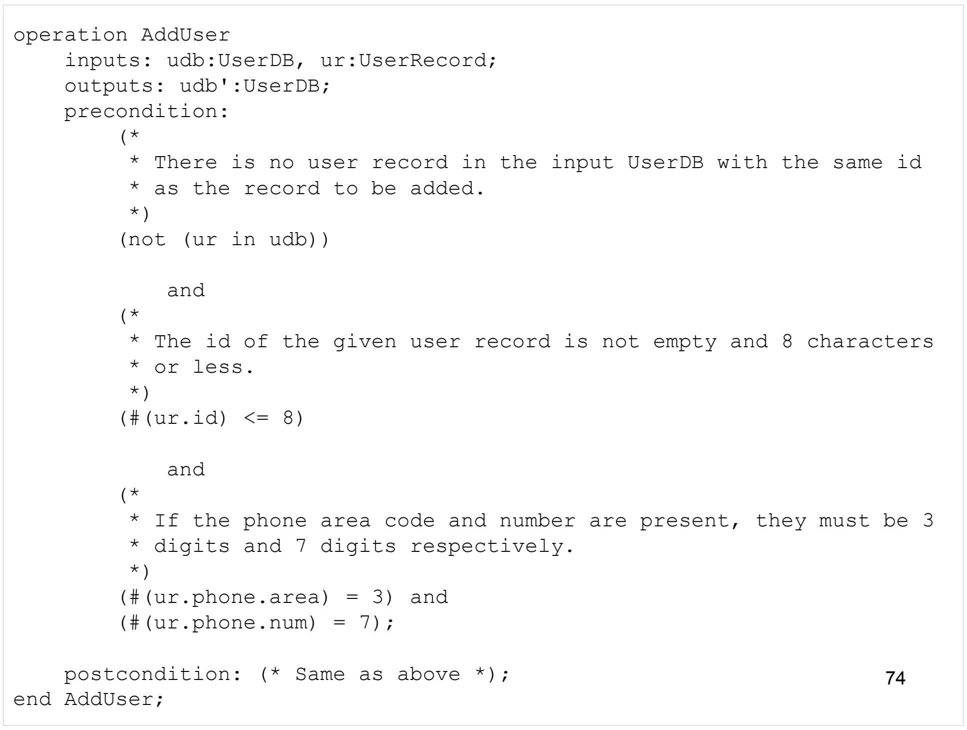
Translating User-level Requirements into Boolean Logic

1. There is no user record in the input database with the same id as the record to be added
2. The id of an added user record cannot be empty and must be no more than 8 characters in length
3. If the area code and phone number are present, they must be 3 digits and 7 digits respectively

73

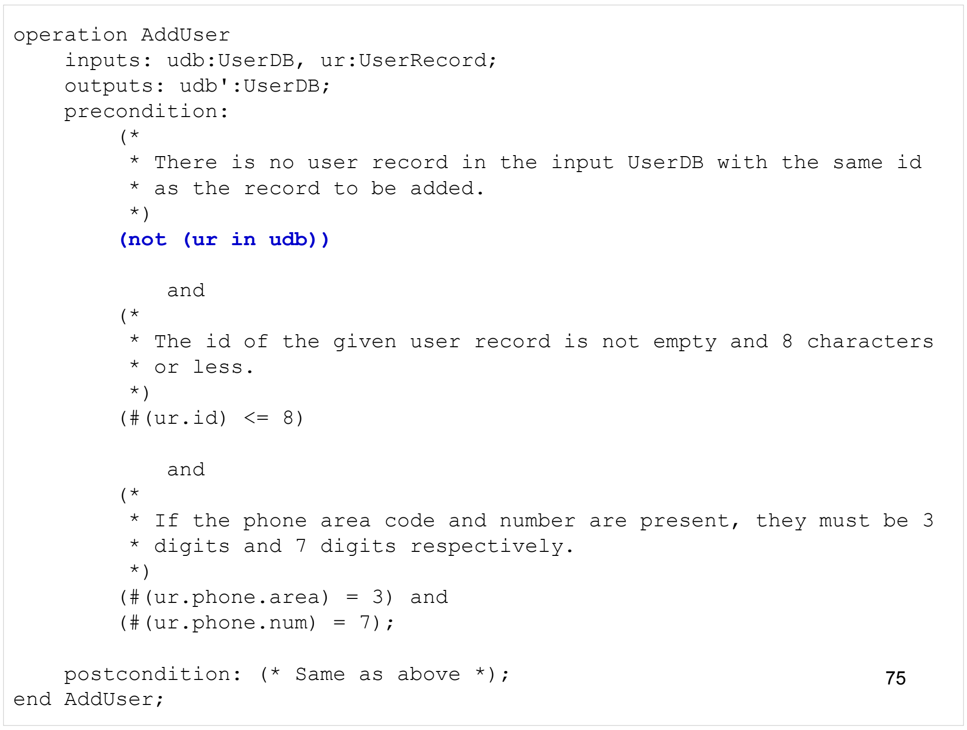
An important part of refining a specification is translating user-level requirements, stated in English prose, into Boolean logic. Exploratory expression evaluation, including validation invocations, can be useful in this translation process.

These are typical user-level requirements for an operation like adding a record to a database, i.e., the AddUser operation described in the previous section of the thesis.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (ur in udb))

            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

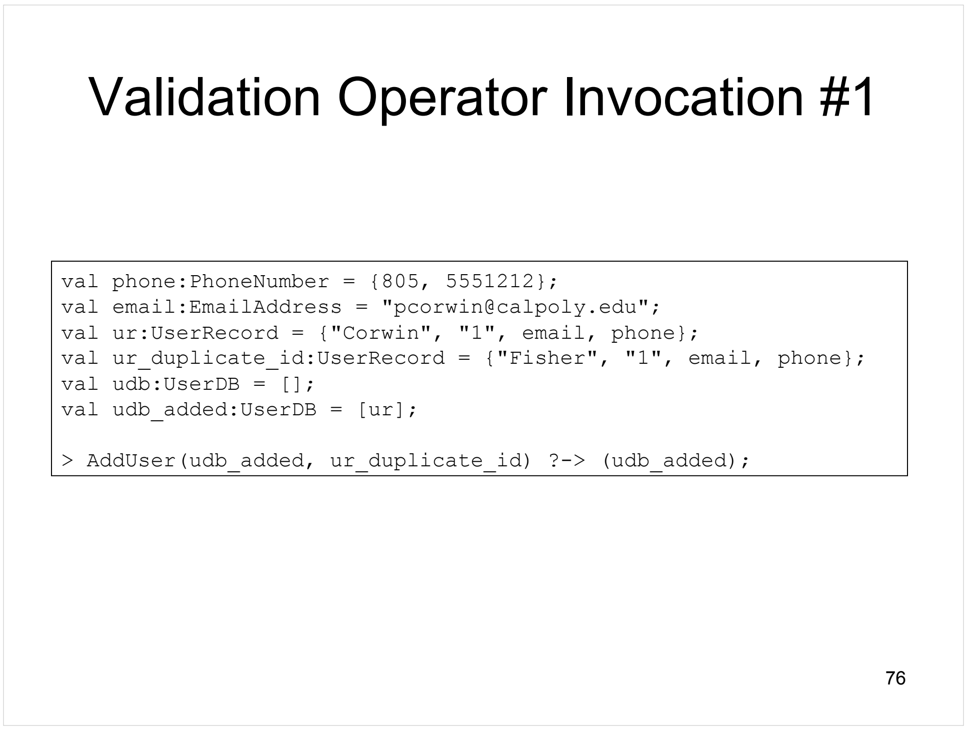
This sample characterizes the kind of logic oversights that have been observed regularly in students' initial efforts to translate user-level requirements from English prose into formal logic. The following slides highlight how FMSL's validation operator invocations can help reveal the logic errors.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (ur in udb))

            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

75

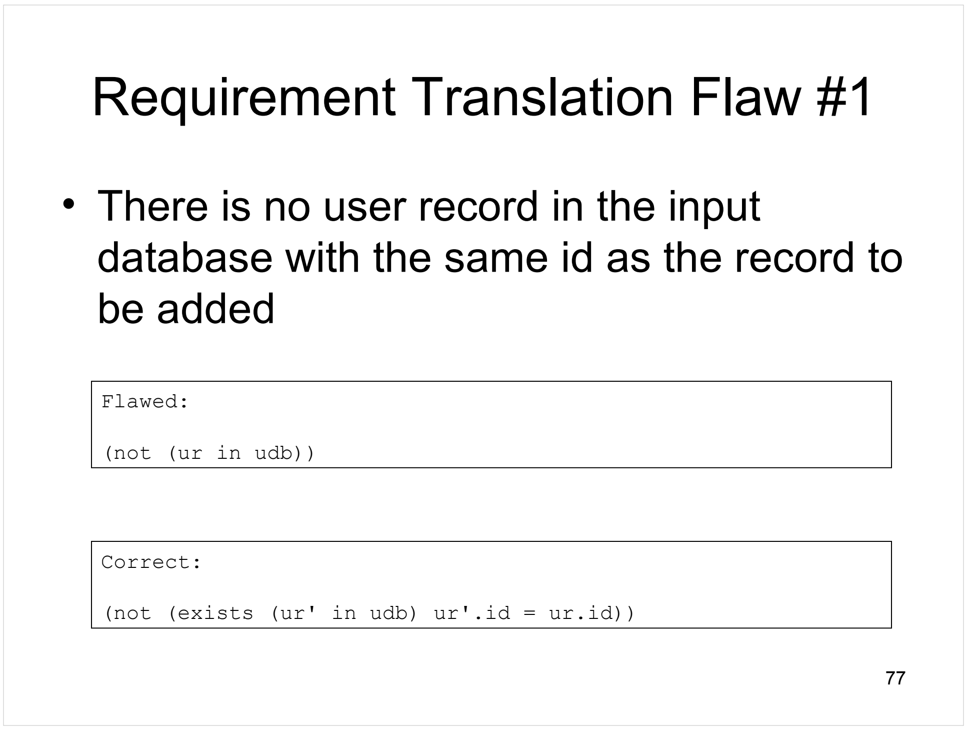
We’ll first examine the top-most piece of logic with the test file on the next slide.



Validation Operator Invocation #1

```
val phone:PhoneNumber = {805, 5551212};
val email:EmailAddress = "pcorwin@calpoly.edu";
val ur:UserRecord = {"Corwin", "1", email, phone};
val ur_duplicate_id:UserRecord = {"Fisher", "1", email, phone};
val udb:UserDB = [];
val udb_added:UserDB = [ur];

> AddUser(udb_added, ur_duplicate_id) ?-> (udb_added);
```

Here you can see the creation of two user record values that share all components in common except for the name field. The correct output of this validation is { false, nil }, since the requirement called for a unique ID and here the two user records have the same id of “1”. By running this example through we can see that the existing precondition is not strong enough.



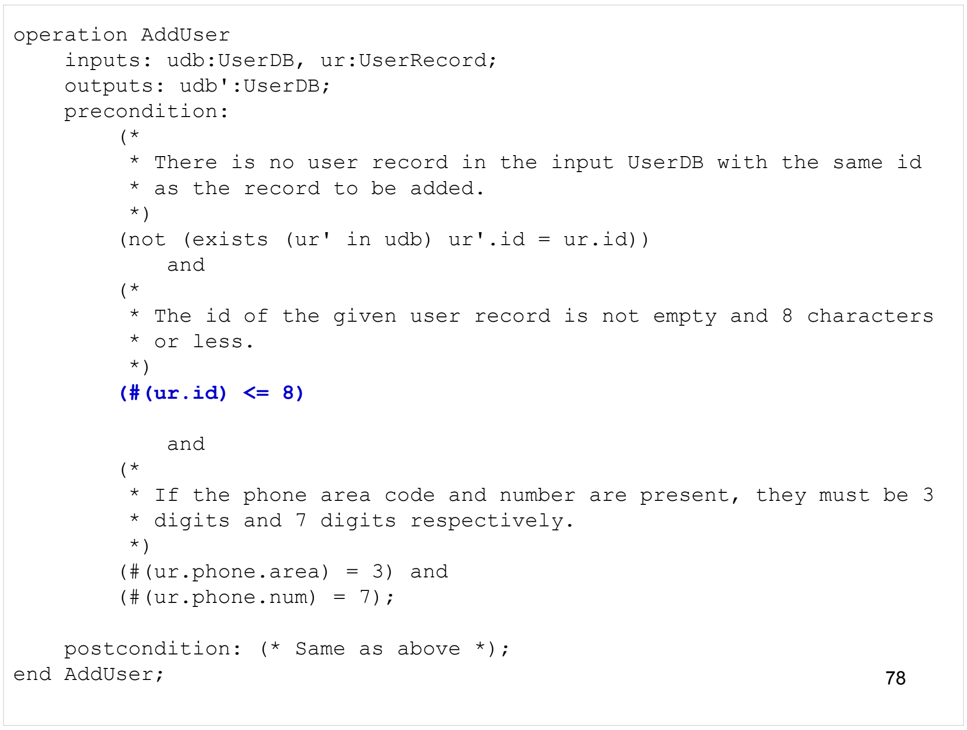
Requirement Translation Flaw #1

- There is no user record in the input database with the same id as the record to be added

```
Flawed:

(not (ur in udb))
```

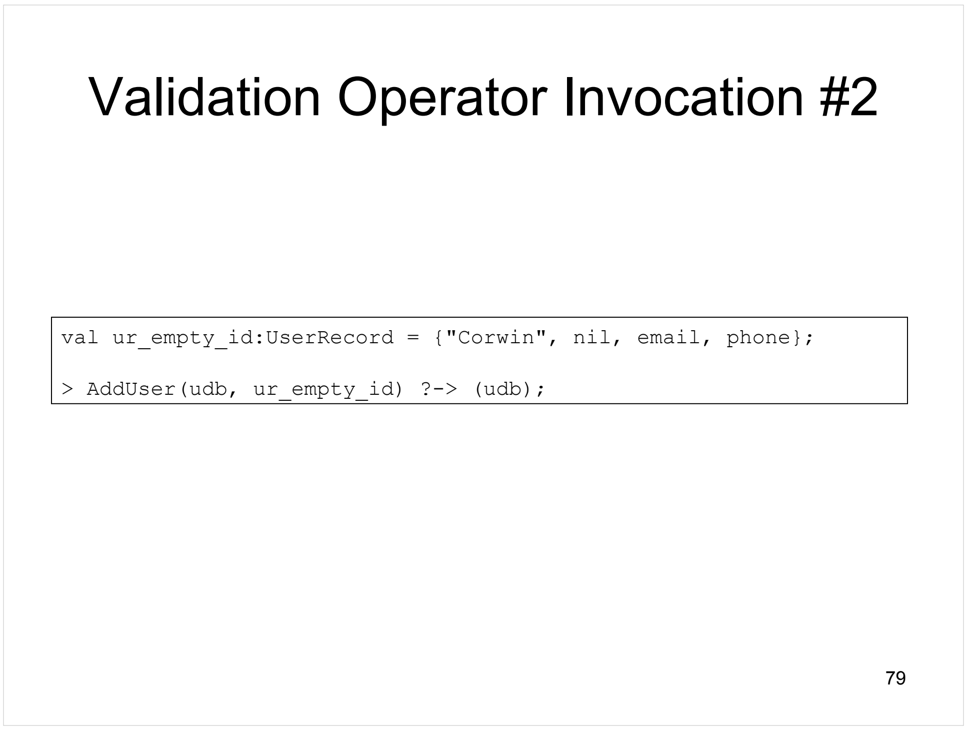
```
Correct:

(not (exists (ur' in udb) ur'.id = ur.id))
```

Here we can see the flawed logic vs. the correct logic, the latter of which properly formalizes the requirement.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))
            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

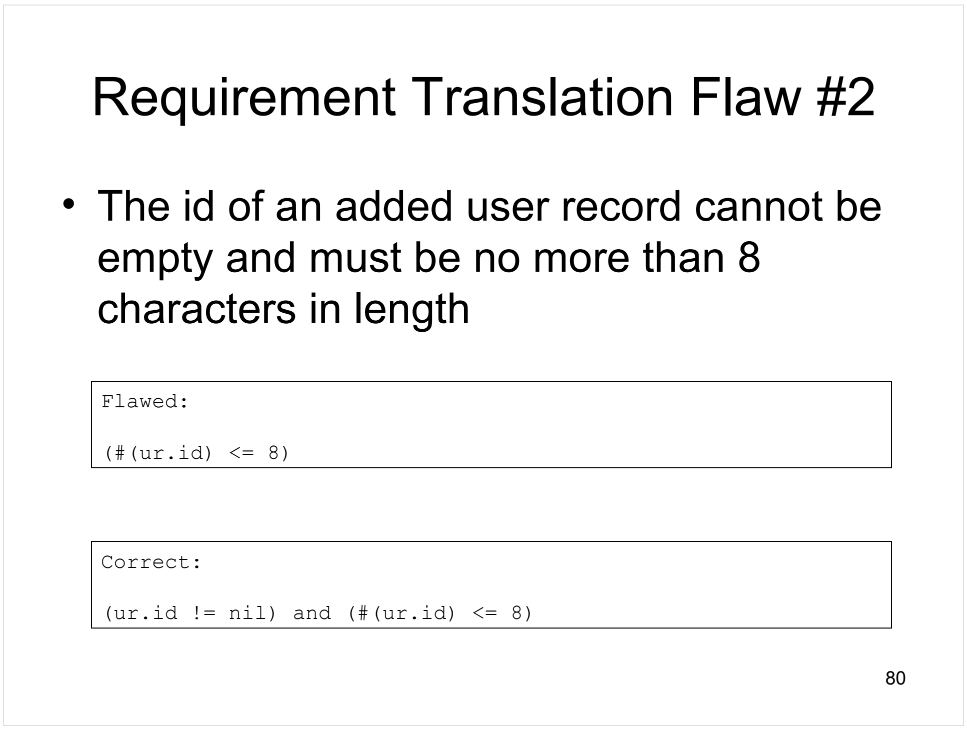
So we’ve fixed the first piece of logic; now we’ll examine the second piece of logic.



Validation Operator Invocation #2

```
val ur_empty_id:UserRecord = {"Corwin", nil, email, phone};

> AddUser(udb, ur_empty_id) ?-> (udb);
```

The result of this evaluation also should be { false, nil }, since we’re passing in an empty ID.



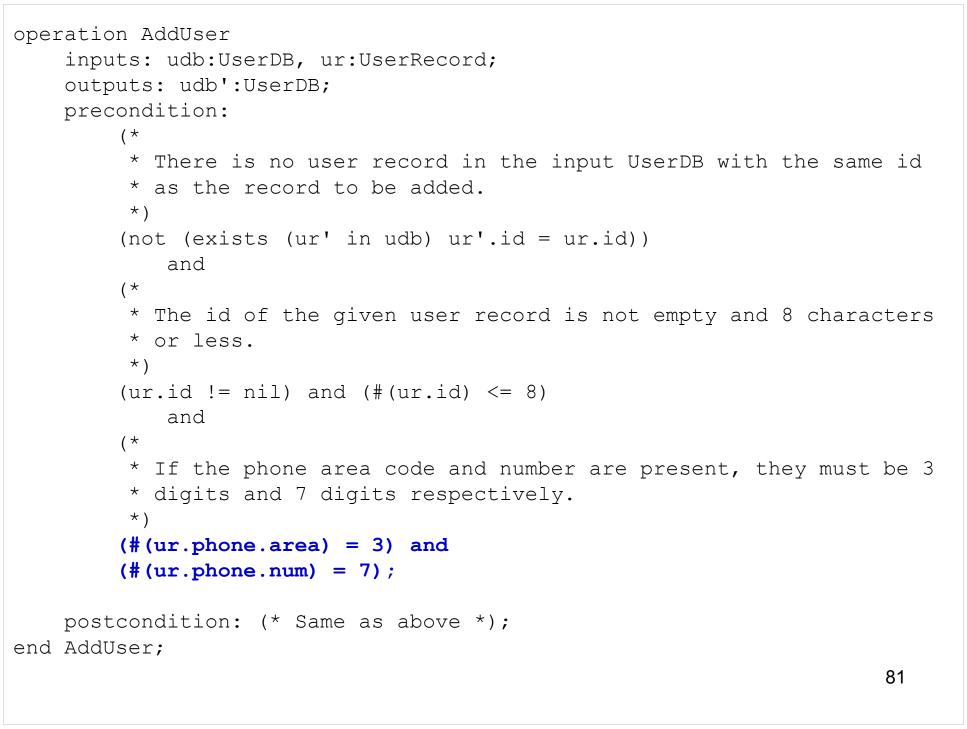
Requirement Translation Flaw #2

- The id of an added user record cannot be empty and must be no more than 8 characters in length

```
Flawed:

(#(ur.id) <= 8)
```

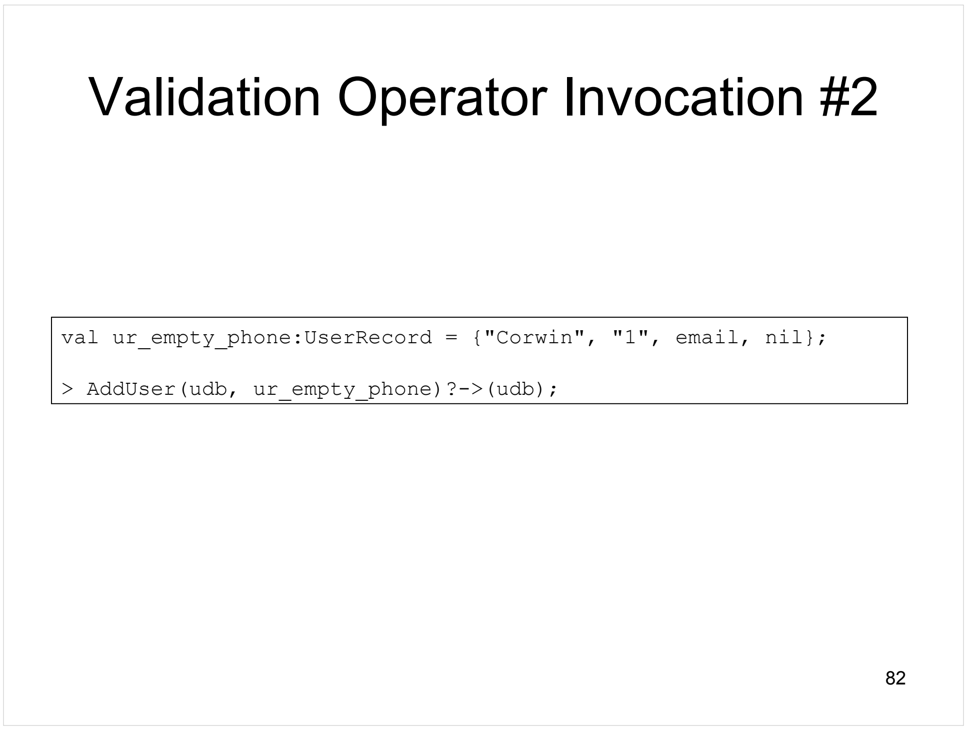
```
Correct:

(ur.id != nil) and (#(ur.id) <= 8)
```

Per FMSL's specific semantics, the length of nil evaluates to 0, which is less than or equal to 8, and so the case would pass through. The correct version of the logic explicitly states that the id field is not empty and its length is less than or equal to 8.



```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))
            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)
            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (#(ur.phone.area) = 3) and
        (#(ur.phone.num) = 7);

    postcondition: (* Same as above *);
end AddUser;
```

Now that we've corrected the first two pieces of logic, we'll examine the last.

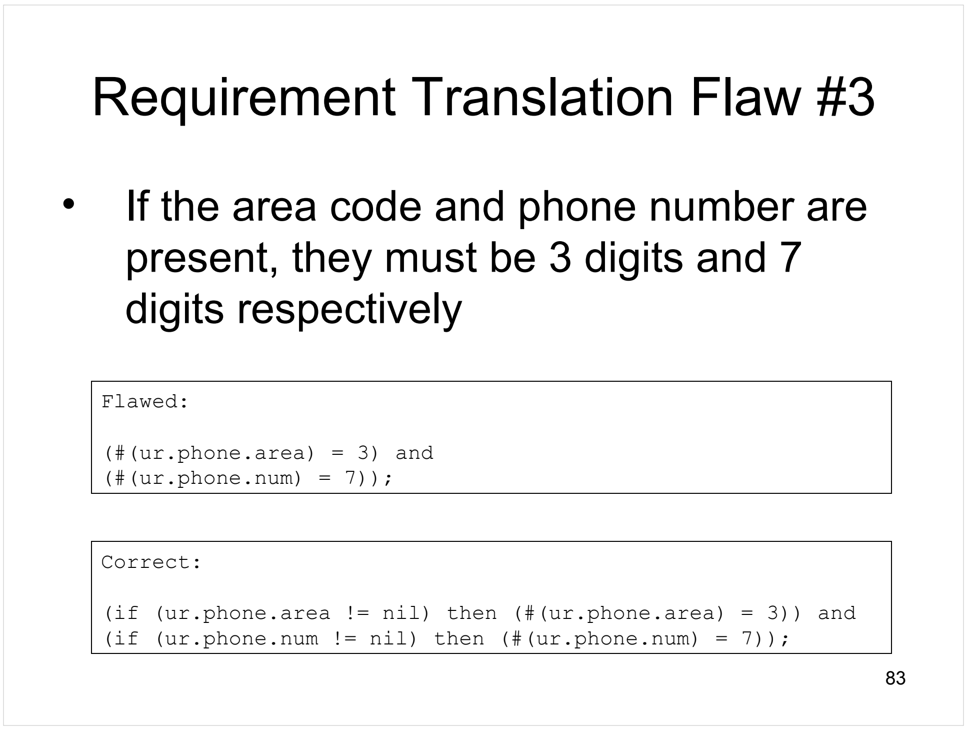


Validation Operator Invocation #2

```
val ur_empty_phone:UserRecord = {"Corwin", "1", email, nil};

> AddUser(udb, ur_empty_phone)?->(udb);
```

82

The result of this evaluation should be { true, nil }, since it’s acceptable for the phone field to be empty.



Requirement Translation Flaw #3

- If the area code and phone number are present, they must be 3 digits and 7 digits respectively

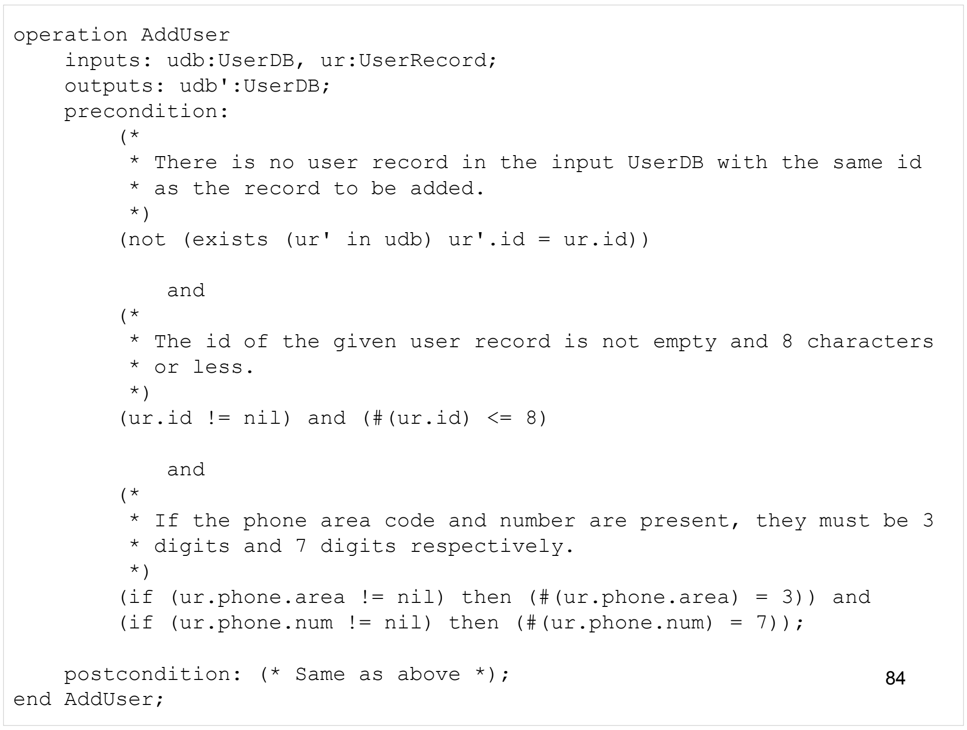
```
Flawed:

(#(ur.phone.area) = 3) and
(#(ur.phone.num) = 7));
```

```
Correct:

(if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
(if (ur.phone.num != nil) then (#(ur.phone.num) = 7));
```

83

The previous validation operator invocation incorrectly evaluates to { false, nil }, since it checks that the area and num fields have lengths equal to 3 and 7, respectively. The correct logic below incorporates a check for nil, which accommodates the first part of the requirement that states “if the area code and phone number are present…”

Now, in this corrected logic we’re making an assumption that it’s ok for only one of area code or phone number to be present. At this point we would go back to the requirement author or customer to request clarification.



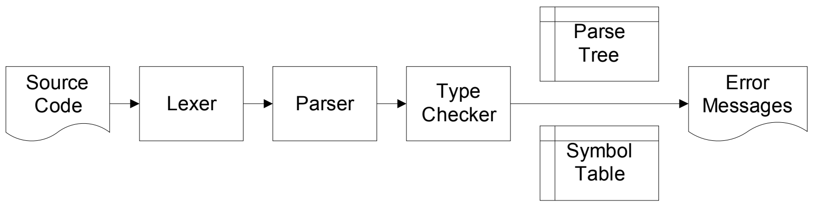
```
operation AddUser
    inputs: udb:UserDB, ur:UserRecord;
    outputs: udb':UserDB;
    precondition:
        (*
         * There is no user record in the input UserDB with the same id
         * as the record to be added.
         *)
        (not (exists (ur' in udb) ur'.id = ur.id))

            and
        (*
         * The id of the given user record is not empty and 8 characters
         * or less.
         *)
        (ur.id != nil) and (#(ur.id) <= 8)

            and
        (*
         * If the phone area code and number are present, they must be 3
         * digits and 7 digits respectively.
         *)
        (if (ur.phone.area != nil) then (#(ur.phone.area) = 3)) and
        (if (ur.phone.num != nil) then (#(ur.phone.num) = 7));

    postcondition: (* Same as above *);
end AddUser;
```

So here’s the updated description that we refined by iteratively testing validation operator invocations.

Presentation Outline

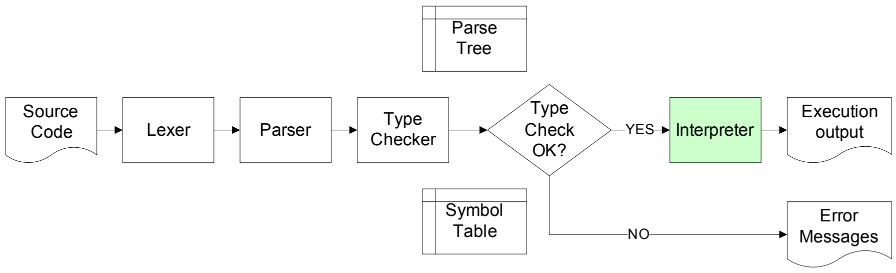
- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

FMSL Pre-Thesis



Prior to the work of this thesis, the mechanized checking of an FMSL specification was limited to static syntax and semantic analysis. As with most programming language compilers, the output of the static analysis is empty, unless errors are detected.

FMSL Post-Thesis



87

The work for this thesis has added support for evaluating expressions through a functional interpreter, which is a valid path provided there are no type check errors. The execution output from the interpreter includes expression evaluation results as well as any run-time errors, such as division by zero.

New to FMSL:
The Functional Interpreter

- Expression evaluation
- Function / operation evaluation
- Execution of preconditions and postconditions
- Quantifier evaluation
- Value universe
- Validation operator

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Basic Object Types

- boolean
- integer
- real
- string
- nil

Boolean: true/false

Integer: non-fraction numbers

Real: double-precision decimal numbers

String: sequences of characters

Nil – the empty value for any object

Basic Operators: booleans

Operator	**Description**	**Returns**
`not`	negation	`boolean`
`and`	conjunction	`boolean`
`or`	disjunction	`boolean`
`xor`	exclusive disjunction	`boolean`
`=>`	implication	`boolean`
`<=>`	two-way implication; if and only if	`boolean`
`if b1 then b2` where `b1`, `b2` are Boolean expressions	conditional	`boolean`
`if b1 then b2 else b3` where `b1`, `b2`, `b3` are Boolean expressions	conditional with else	`boolean`

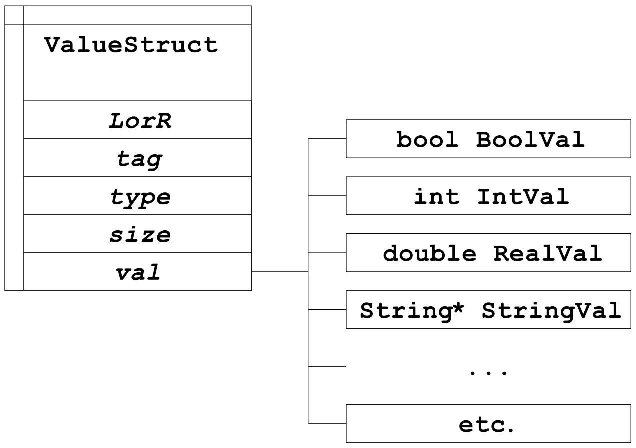
Basic Operators: integers and reals

Operator	Description	Returns
`+`	Addition	`integer` or `real`
`-`	Subtraction	`integer` or `real`
`*`	multiplication	`integer` or `real`
`/`	Division	`integer` or `real`
`mod`	Modulus	`integer`
`+ (unary)`	returns 1*the number	`integer` or `real`
`- (unary)`	returns -1*the number	`integer` or `real`
`=`	Equality	`boolean`
`!=`	Inequality	`boolean`
`>`	greater than	`boolean`
`<`	less than	`boolean`
`>=`	greater than or equal to	`boolean`
`<=`	less than or equal to	`boolean`

Basic Operators: strings

Operator	Description	Returns
`=`	equality	`boolean`
`!=`	inequality	`boolean`
`#`	string length	`integer`
`in`	membership test	`boolean`
`+`	concatenation	`string`
`[n]`	single character selection	`string`
`[m .. n]`	range / substring selection	`string`

Internal Representation of Values



ValueStruct is a C structure that stores information about an object value.

LorR indicates whether the underlying value is an L or R value (location vs. a value)

Tag indicates the general type of the value

Type is the full type structure, whose utility is most obvious in complex object types

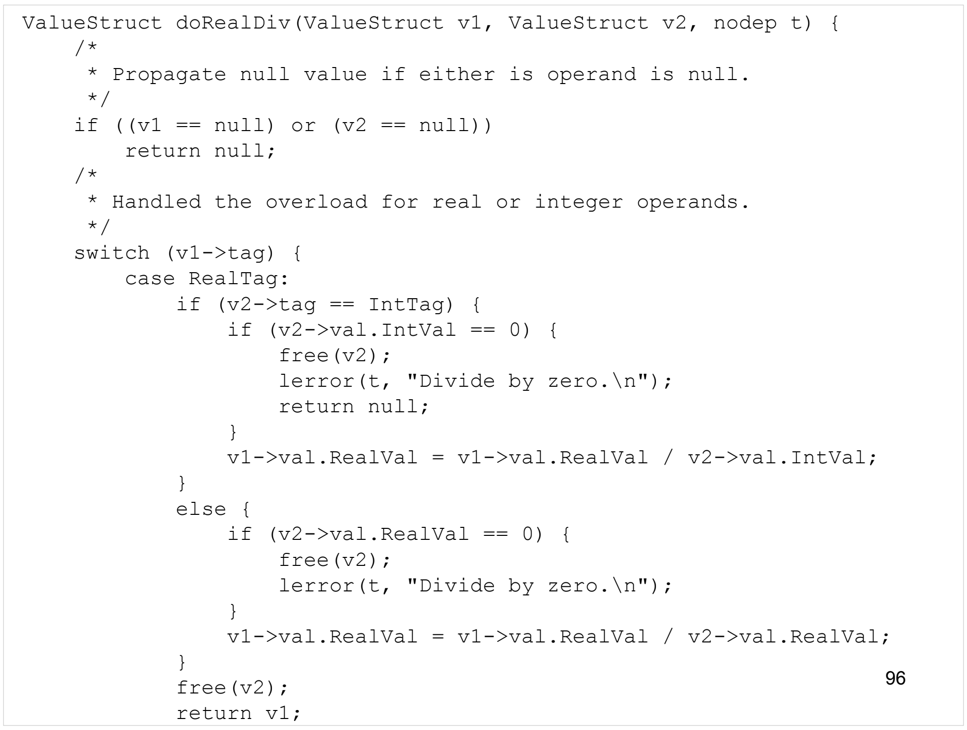
Size is the type size which can be a number of elements or bytes of storage

Val is the actual byte representation in memory, and is a union as seen here

Division Operator Example

```
(*
 * Declare and assign values to x, y
 *)
val x:real = 3.141592654;
val y:real = 2.718281828;

(*
 * Evaluate x divided by y and output the result
 *)
> x / y;
```

- Determine that the expression involves a binary operator
- Determine the operator (/)
- Call and return the result of the function that performs the division, passing in as parameters the `ValueStruct`s corresponding to the x and y operands



```
ValueStruct doRealDiv(ValueStruct v1, ValueStruct v2, nodep t) {
    /*
     * Propagate null value if either is operand is null.
     */
    if ((v1 == null) or (v2 == null))
        return null;
    /*
     * Handled the overload for real or integer operands.
     */
    switch (v1->tag) {
        case RealTag:
            if (v2->tag == IntTag) {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.IntVal;
            }
            else {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                }
                v1->val.RealVal = v1->val.RealVal / v2->val.RealVal;
            }
            free(v2);
            return v1;
```

Returns a valuestruct, has valuestruct paramaters where v1 and v2 correspond to x and y. I'll get to nodep t parameter in a sec.

Depending on the tag indicators of each v1 and v2, either of which could be real or integer, we perform the division. Here you can see the check for a run-time error, division by zero, which utilizes the t parameter to help indicate the location of the run-time error.

Note that we don't have to check for non-numeric operands because the type checker already has performed that check. i.e., in order to get this far the static type checking pass must have been successful.

```
        case IntTag:
            if (v2->tag == RealTag) {
                if (v2->val.RealVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.RealVal = v1->val.IntVal / v2->val.RealVal;
                v1->tag = RealTag;
            }
            else {
                if (v2->val.IntVal == 0) {
                    free(v2);
                    lerror(t, "Divide by zero.\n");
                    return null;
                }
                v1->val.IntVal = v1->val.IntVal / v2->val.IntVal;
            }
            free(v2);
            return v1;
    }
}
```

Complex Structures

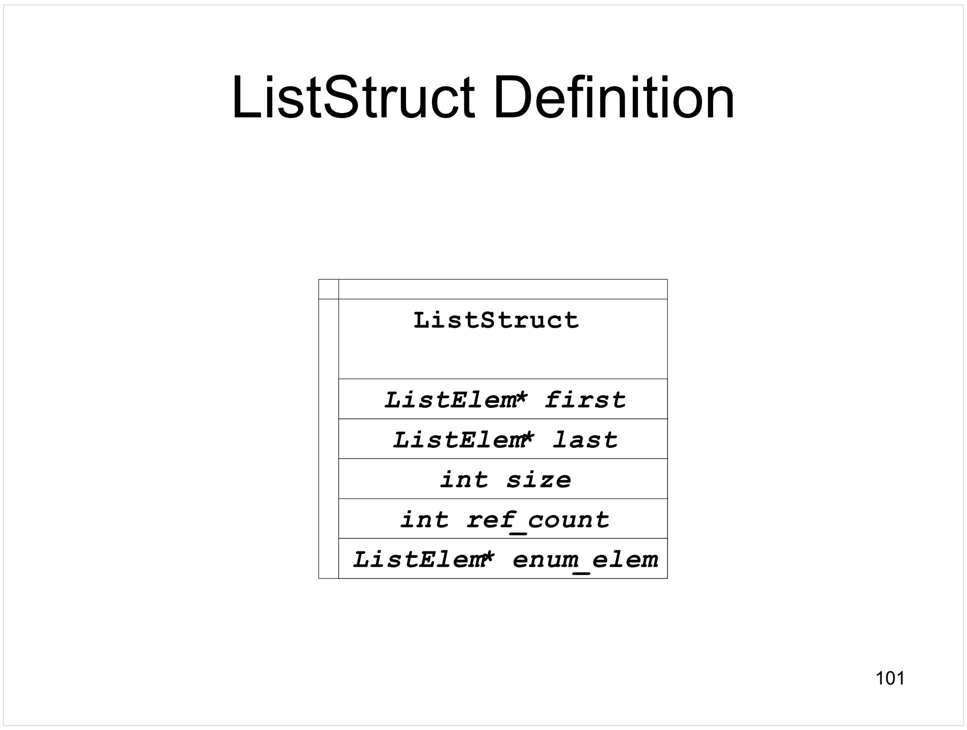
- Lists
  - Homogeneous
  - Hold zero or more object values
  - Analogous to an array with no predetermined, fixed size
- Tuples
  - Heterogeneous
  - Hold fixed number of components of specific object types
  - Analogous to a C struct

Basic Operators: lists

Operator	Description	Returns
`=`	equality	`boolean`
`!=`	inequality	`boolean`
`in`	membership	`boolean`
`#`	element count	`integer`
`+`	concatenation	`list type`
`-`	deletion from list	`list type`
`[n]`	element selection	`list type`
`[m .. n]`	range selection	`list type`

Basic Operators: tuples

Operator	Description	Returns
`=`	Equality	`boolean`
`!=`	inequality	`boolean`
`.`	field access	`any field type`



ListStruct Definition

ListStruct
ListElem first*
ListElem last*
int size
int ref_count
ListElem enum_elem*

Internal representation of a list, which can be pointed to by the val component of a ValueStruct.

Linked list

For convenience has pointers to first and last

Size

Ref_count for potential memory management clues

Enum_elem for to make for easier traversal

List Range Selection Example

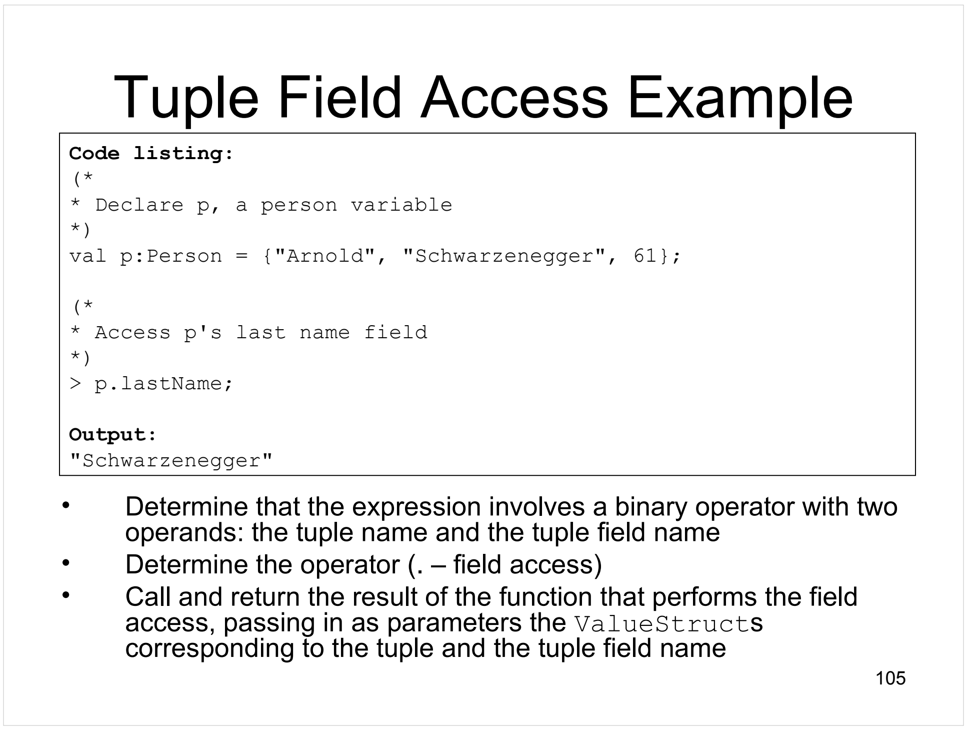
Code listing:
```
(*
* Declare the IntegerList type
*)
object IntegerList = integer*;

(*
* Declare an intlist value
*)
val intlist:IntegerList = [1,1,2,3,5,3+5];

(*
* Select the subcomponents at indexes 3, 4, and 5.
*)
> intlist[3..5];
```

Output:
```
[ 2, 3, 5 ]
```

List Range Selection Example

- Determine that the expression involves a ternary operator with three operands: list, lower bound, upper bound
- Determine the operator ([] – list range selection)
- Call and return the result of the function that performs the list range selection, passing in as parameters the `ValueStruct`s corresponding to the list, lower bound, and upper bound operands

```
ValueStruct doArraySliceRef(v1, v2, v3)
    ValueStruct v1;
    ValueStruct v2;
    ValueStruct v3;
{
    ValueStruct result;
    int i;
    /* start building the new list */
    result = MakeVal(RVAL, v1->type);
    if (v1->tag == ListTag) {
        result->val.ListVal = NewList();
        /*
         * loop through from lower .. upper and add the elements
         * to result.
         */
        for (i = v2->val.IntVal; i <= v3->val.IntVal; i++) {
            PutList(result->val.ListVal,
                    GetListNth(v1->val.ListVal, i));
        }
    }
    else if (v1->tag == StringTag) {
        result->val.StringVal =
            (String *)SubString(v1->val.StringVal,
                                v2->val.IntVal,
                                v3->val.IntVal);
    }
    return result;
}
```



Tuple Field Access Example

```
Code listing:
(*
* Declare p, a person variable
*)
val p:Person = {"Arnold", "Schwarzenegger", 61};

(*
* Access p's last name field
*)
> p.lastName;

Output:
"Schwarzenegger"
```

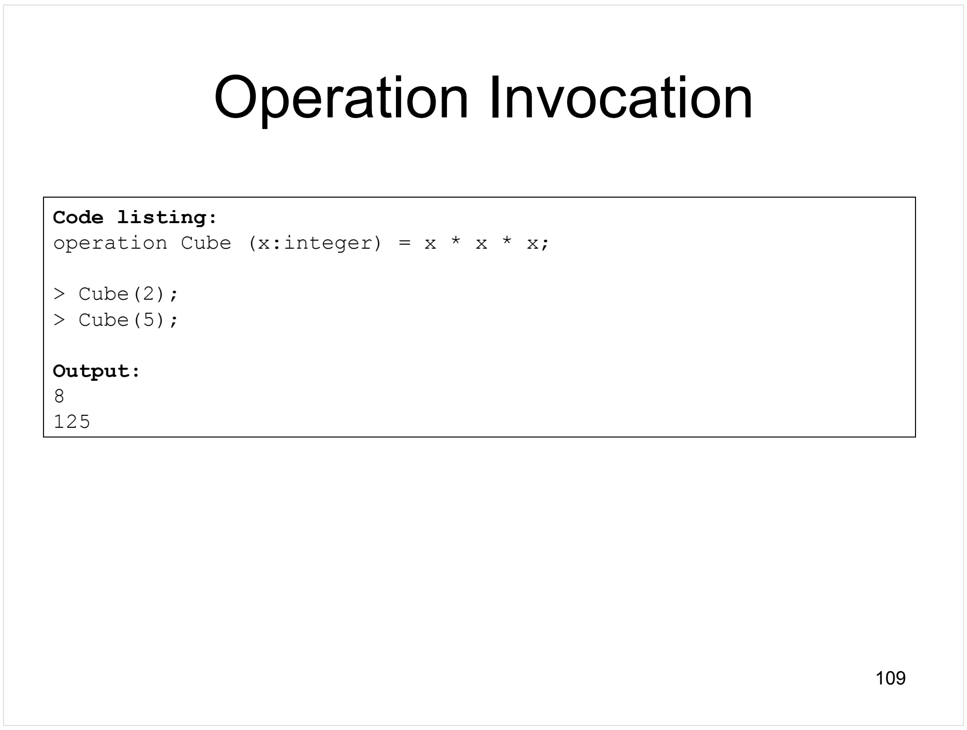
- Determine that the expression involves a binary operator with two operands: the tuple name and the tuple field name
- Determine the operator (. – field access)
- Call and return the result of the function that performs the field access, passing in as parameters the `ValueStruct`s corresponding to the tuple and the tuple field name

Internally, tuples are stored in a manner very similar to the way lists are stored.

```
ValueStruct RecordRef(desig, field)
    ValueStruct desig;  /* L-value for the left operand. */
    nodep field;        /* Ident for the right operand. */
{
    ValueStruct valueField,
        tuple,
        newDesig;
    SymtabEntry *f;
    int n;
    TypeStruct type = ResolveIdentType(desig->type, null, false),
        fieldType;

    /*
     * Deal with nil desig, i.e., just return it as is.
     */
    if (isNilValue(desig)) {
        return desig;
    }
```

```
/*
 * coming in, desig->LVal should point to the ValueStruct of the
 * struct
 */
if (field->header.name == Yident) {
    f = LookupIn(field->components.atom.val.text,
                    type->components.type.kind.record.fieldstab);
    fieldType = ResolveIdentType(f->Type, null, false);
}
else {
    f = null;
    n = field->components.atom.val.integer;
    fieldType = ResolveIdentType(
        GetNthField(type->components.type.kind.record.fields, n)->
            components.decl.kind.field.type, null, false);
}
```

```
    if (desig->LorR == LVAL) {
        tuple = (ValueStruct)*(desig->val.LVal);
    }
    else {
        tuple = desig;
    }

    /* Note: Lists are 1-indexed */
    valueField = (ValueStruct)GetListNth(tuple->val.StructVal,
        f ? f->Info.Var.Offset + 1 : n);
    /*
     * if we have valueField filled in, use its type... otherwise use the
     * fieldType
     */
    if (!valueField) {
        newDesig = MakeVal(LVAL, fieldType);
    }
    else {
        newDesig = MakeVal(LVAL, valueField->type);
    }
    newDesig->val.LVal = (ValueStruct *) malloc(sizeof(Value **));
    *(newDesig->val.LVal) = valueField;

    return newDesig;
}
```



Operation Invocation

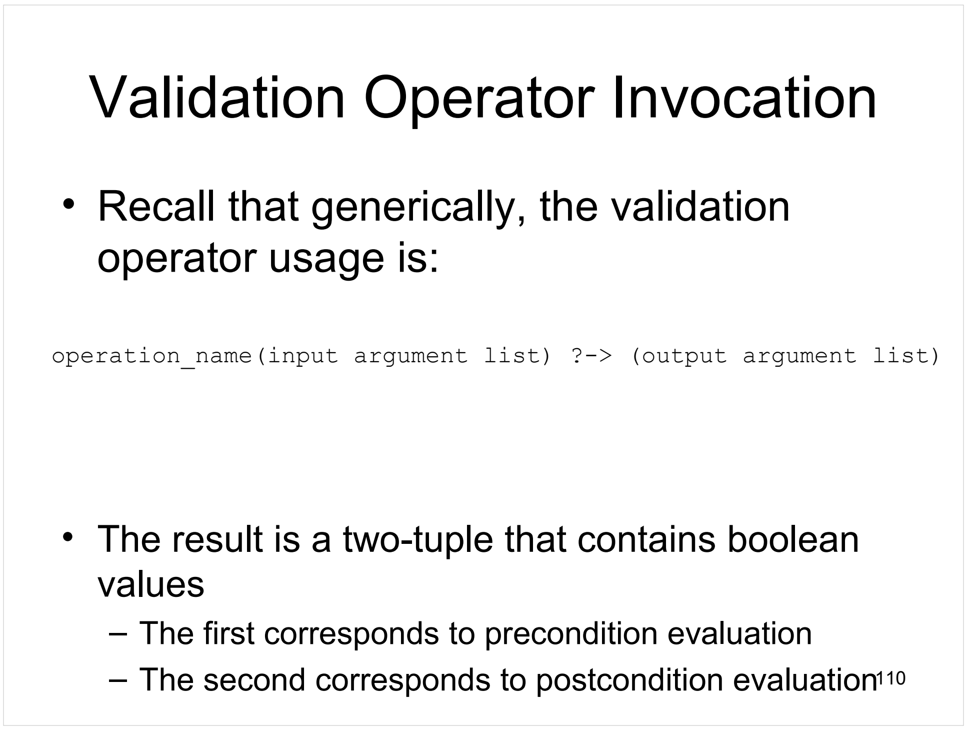
Code listing:
```
operation Cube (x:integer) = x * x * x;

> Cube(2);
> Cube(5);
```

Output:
```
8
125
```

FMSL supports the definition of computation operations. These have the standard semantics of procedural abstractions definable in almost all programming languages. Parameter passing is strictly call-by-value, and the operation result is equal to the result . When operations have no mutating set expressions, they are side-effect free.

To perform operation invocations, FMSL first pushes an activation record onto the stack. Then each input parameter is evaluated and the results are bound to the proper memory locations according to the formal parameter names. Then the local symbol table is pushed to the top of the symbol table stack and the operation body is executed. The result gets saved off before popping the activation record, and the result is returned.

See here an example cube operation, that returns the result of cubing the input integer parameter.



Validation Operator Invocation

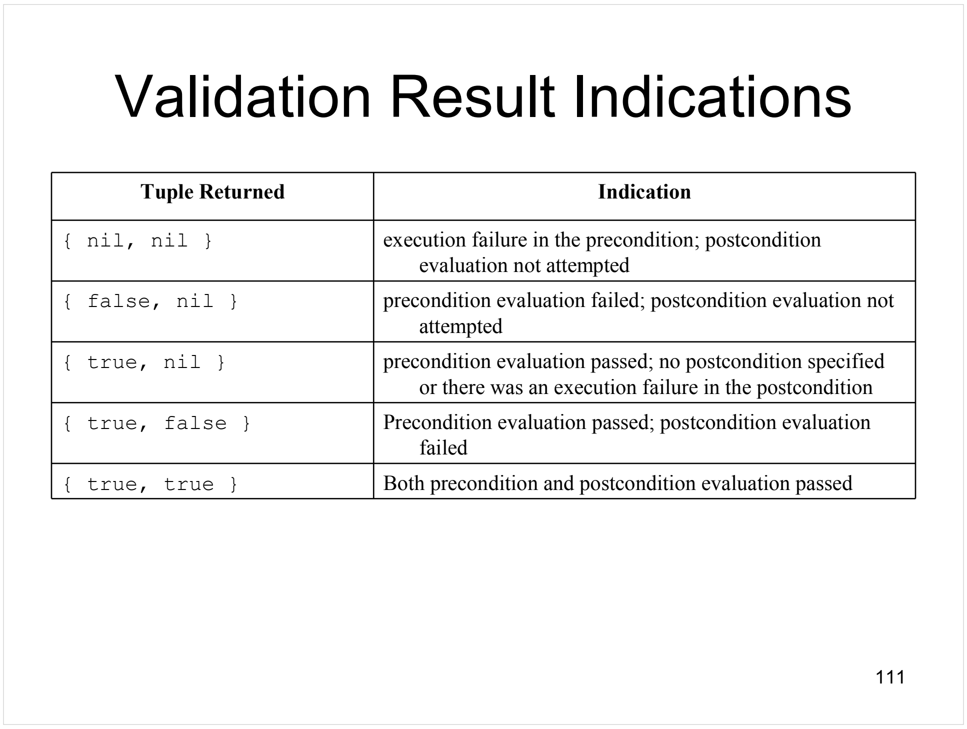
- Recall that generically, the validation operator usage is:

```
operation_name(input argument list) ?-> (output argument list)
```

- The result is a two-tuple that contains boolean values
  - The first corresponds to precondition evaluation
  - The second corresponds to postcondition evaluation

The implementation of the validation operator is similar to the implementation of an operation evaluation, in that the input and output actual parameters are put in the local symbol table and statements using those variables are executed. Whereas with operations the operation body gets executed, with the validation operator the precondition and postcondition bodies are executed.

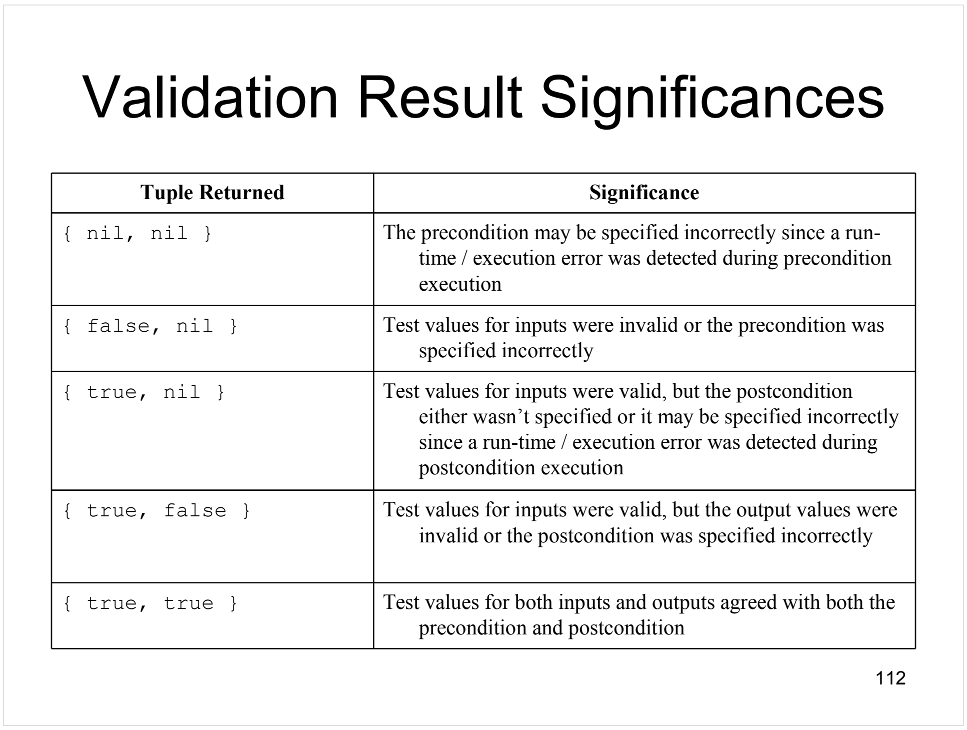
The results of those executions are… [READ BULLET 2]



Validation Result Indications

Tuple Returned	Indication
`{ nil, nil }`	execution failure in the precondition; postcondition evaluation not attempted
`{ false, nil }`	precondition evaluation failed; postcondition evaluation not attempted
`{ true, nil }`	precondition evaluation passed; no postcondition specified or there was an execution failure in the postcondition
`{ true, false }`	Precondition evaluation passed; postcondition evaluation failed
`{ true, true }`	Both precondition and postcondition evaluation passed

The execution failure referred to in the indication column results from an expression returning a nil value. Genuine failures include division by zero, index out of bounds, or access to uninitialized tuple fields.



Validation Result Significances

Tuple Returned	Significance
`{ nil, nil }`	The precondition may be specified incorrectly since a run-time / execution error was detected during precondition execution
`{ false, nil }`	Test values for inputs were invalid or the precondition was specified incorrectly
`{ true, nil }`	Test values for inputs were valid, but the postcondition either wasn't specified or it may be specified incorrectly since a run-time / execution error was detected during postcondition execution
`{ true, false }`	Test values for inputs were valid, but the output values were invalid or the postcondition was specified incorrectly
`{ true, true }`	Test values for both inputs and outputs agreed with both the precondition and postcondition

112

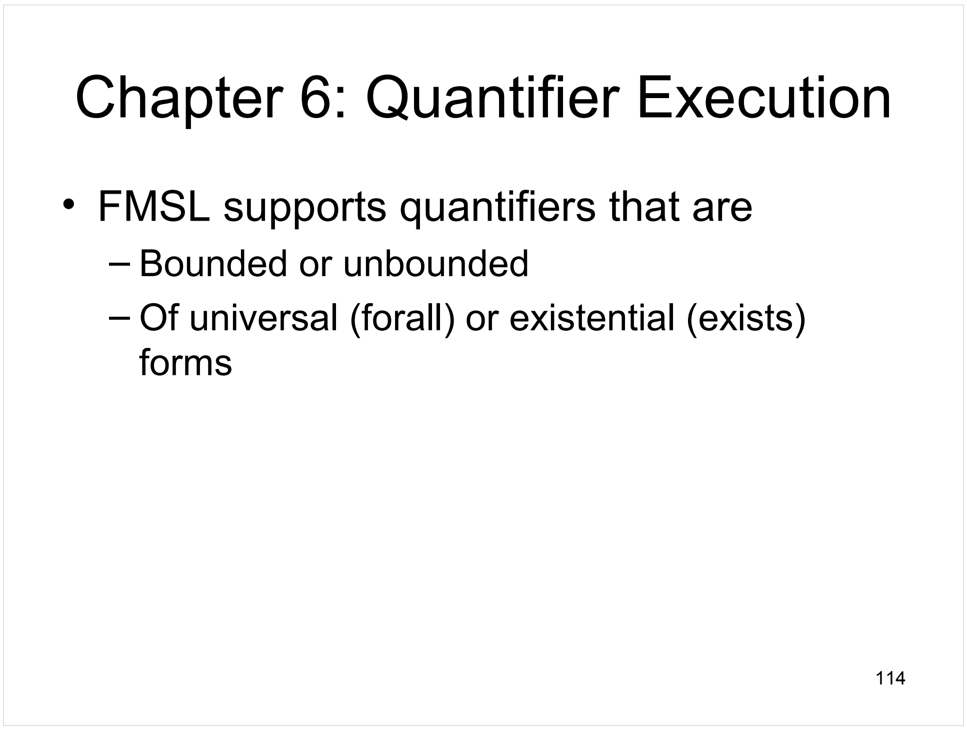
Perhaps more meaningful than the previous slide of indications are these significances, since they give us inferences about input values, output values, and the precondition and postcondition logic.

While some symbolic model checking tools initialize input fields only to values that adhere to the precondition, with FMSL's validation operator the user also can get additional, helpful assurance that there is an absence of unintended behavior. i.e., the user will want to list inputs, outputs, and expected result to get the most helpful feedback about potential specification or test data errors.

Last of ch 5 – next slide is ch 6

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions



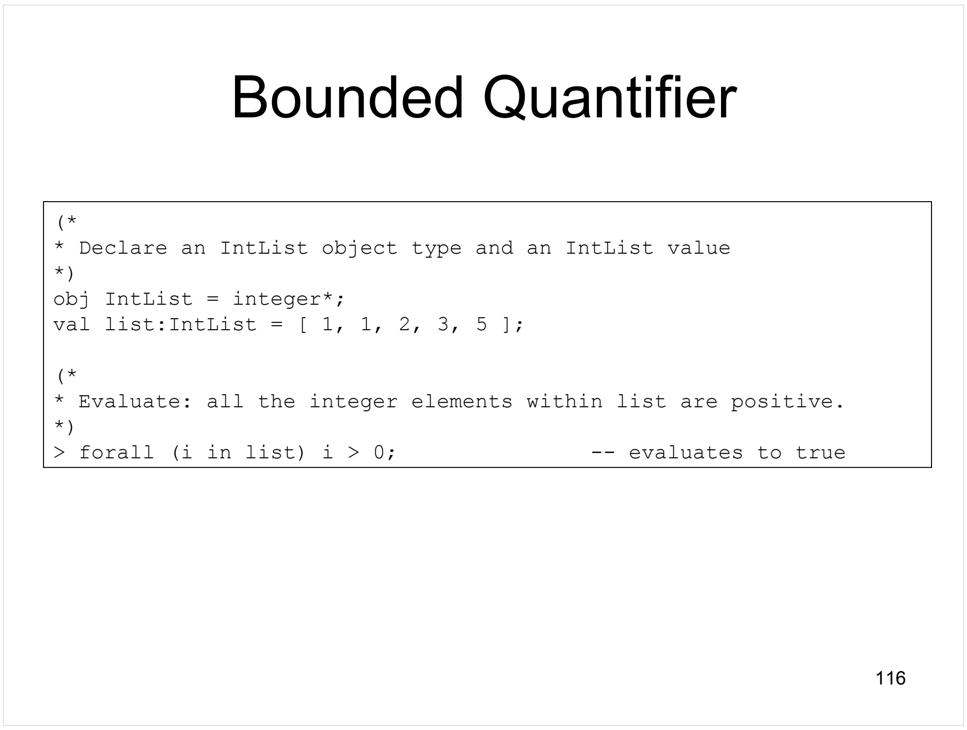
Chapter 6: Quantifier Execution

- FMSL supports quantifiers that are
  - Bounded or unbounded
  - Of universal (forall) or existential (exists) forms

Bounded quantifiers iterate over a discrete range, while unbounded quantifiers iterate over values within a universe that is unbounded or conceptually infinitely large. For example, an unbounded quantifier might iterate over the set of all integers.

Quantifier Syntax

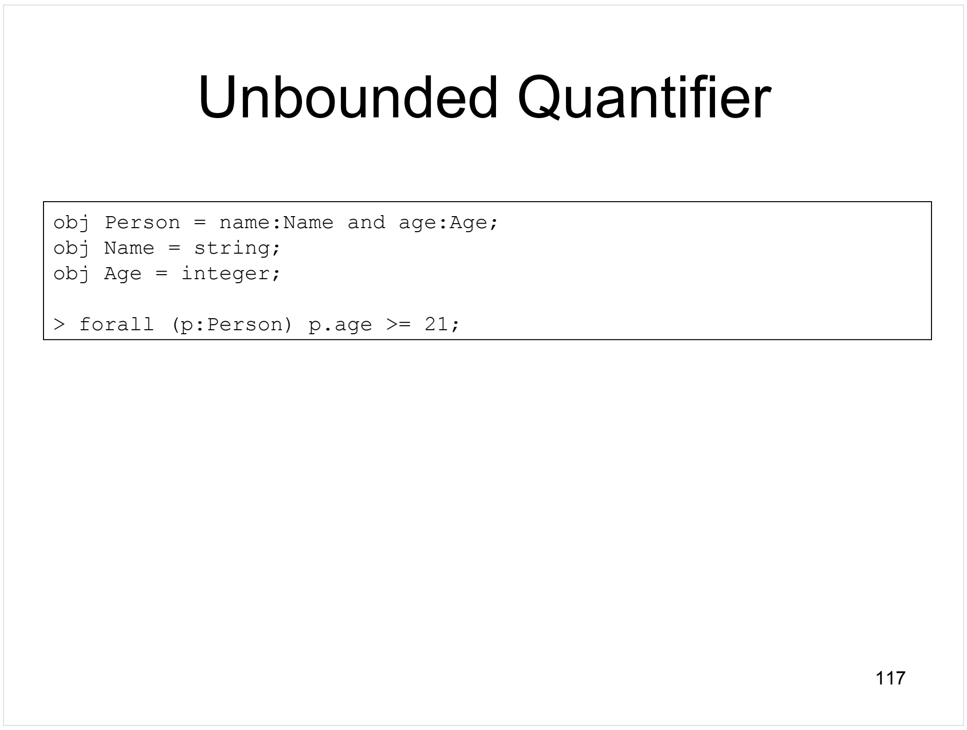
Syntax	**Quantifier Type**	**Reads Like …**
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true



Bounded Quantifier

```
(*
* Declare an IntList object type and an IntList value
*)
obj IntList = integer*;
val list:IntList = [ 1, 1, 2, 3, 5 ];

(*
* Evaluate: all the integer elements within list are positive.
*)
> forall (i in list) i > 0;              -- evaluates to true
```

Implementing this form of quantifier was relatively straightforward compared to unbounded quantifiers.



Unbounded Quantifier

```
obj Person = name:Name and age:Age;
obj Name = string;
obj Age = integer;

> forall (p:Person) p.age >= 21;
```

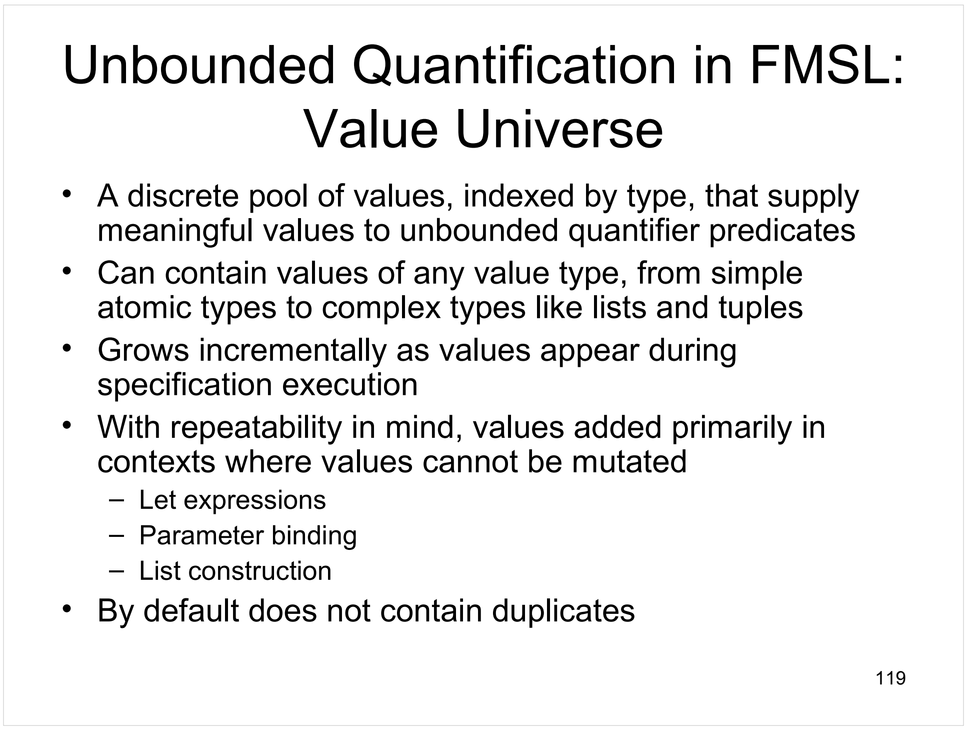
117

Here’s an example of an unbounded quantifier. Unlike the bounded case, it is not immediately clear how the interpreter should evaluate this form of quantifier that concerns this simplified Person object.

For this thesis, FMSL evaluates unbounded quantifiers by iterating through an incrementally built universe of values and evaluating the predicate for each value.

Other Methods of Unbounded Quantifier Execution

- Aslantest
  - When it cannot automatically reduce an expression to true or false, it suspends execution and prompts for user input
- Jahob
  - pickAny: picks an arbitrary value, optionally bounded by lemmas that the user can input, that is placed into the unbounded quantifier
- Executable Z
  - Treats unbounded quantification as a source of non-executability, so such statements are treated as compiler assumptions (and not executable statements)

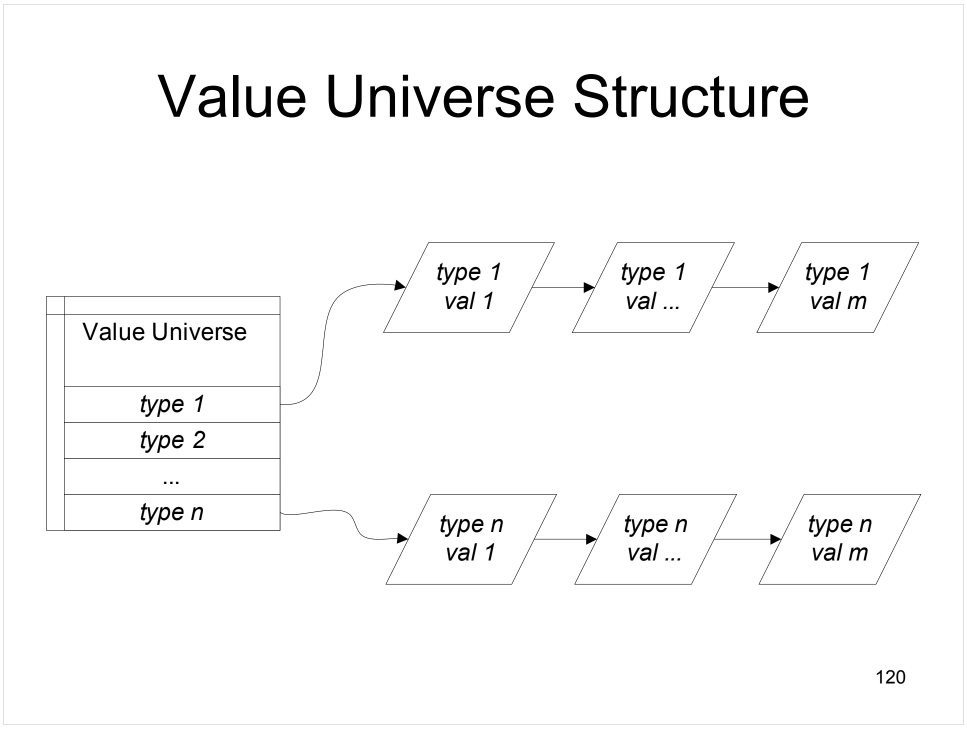


Unbounded Quantification in FMSL: Value Universe

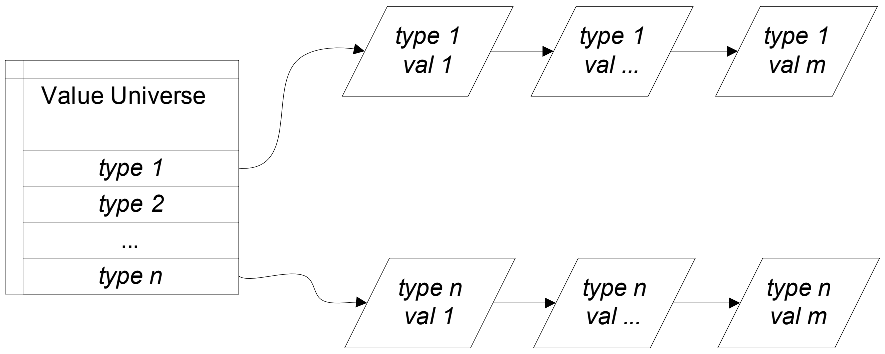
- A discrete pool of values, indexed by type, that supply meaningful values to unbounded quantifier predicates
- Can contain values of any value type, from simple atomic types to complex types like lists and tuples
- Grows incrementally as values appear during specification execution
- With repeatability in mind, values added primarily in contexts where values cannot be mutated
  - Let expressions
  - Parameter binding
  - List construction
- By default does not contain duplicates

Mutation is still possible, and so the universe values can be changed in some cases, the FMSL user should understand that performing mutations can cause undesirable side effects that ripple throughout the universe and normal execution (and non-repeatability). This is consistent with the notion that value mutations are generally considered harmful in a functional environment.

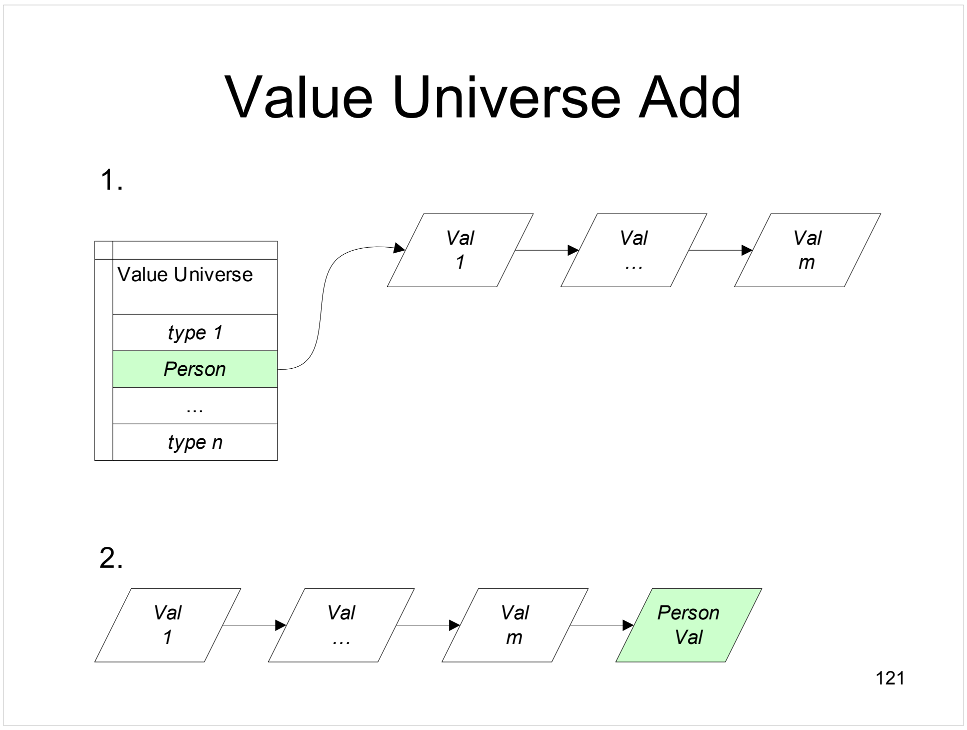
On the topic of duplicates: this adds up-front processing time when calculating whether to add a value to the universe, but it saves memory and reduces processing time during evaluation of unbounded quantifiers.



Value Universe Structure

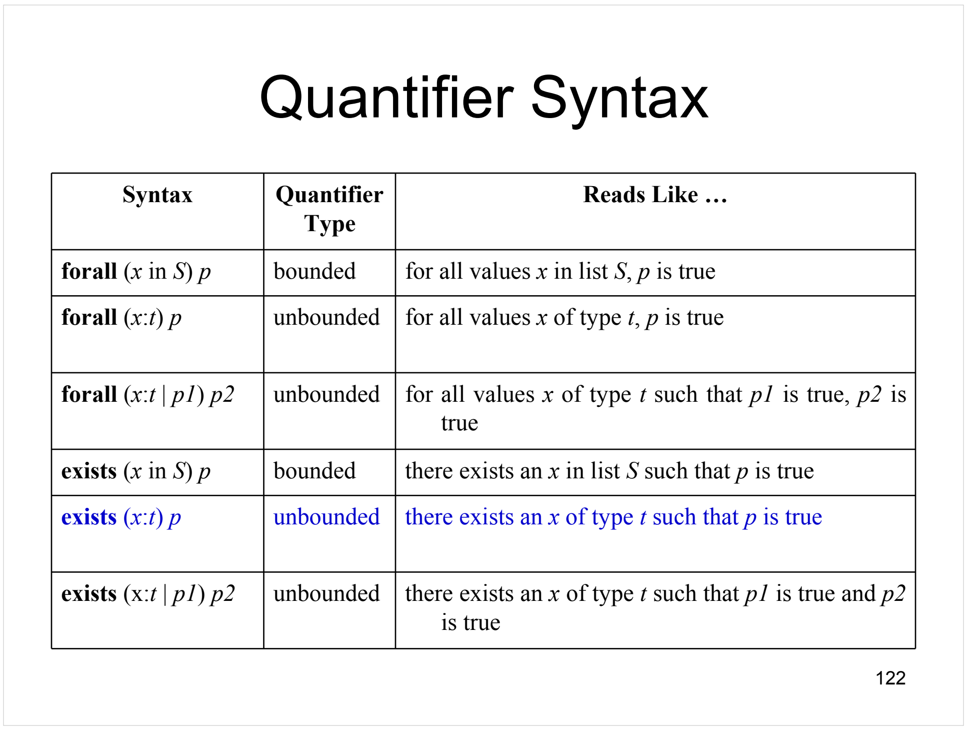


Implemented internally as a block of memory where each memory slot is a pointer to a homogeneous list of values for a particular type.



Value Universe Add

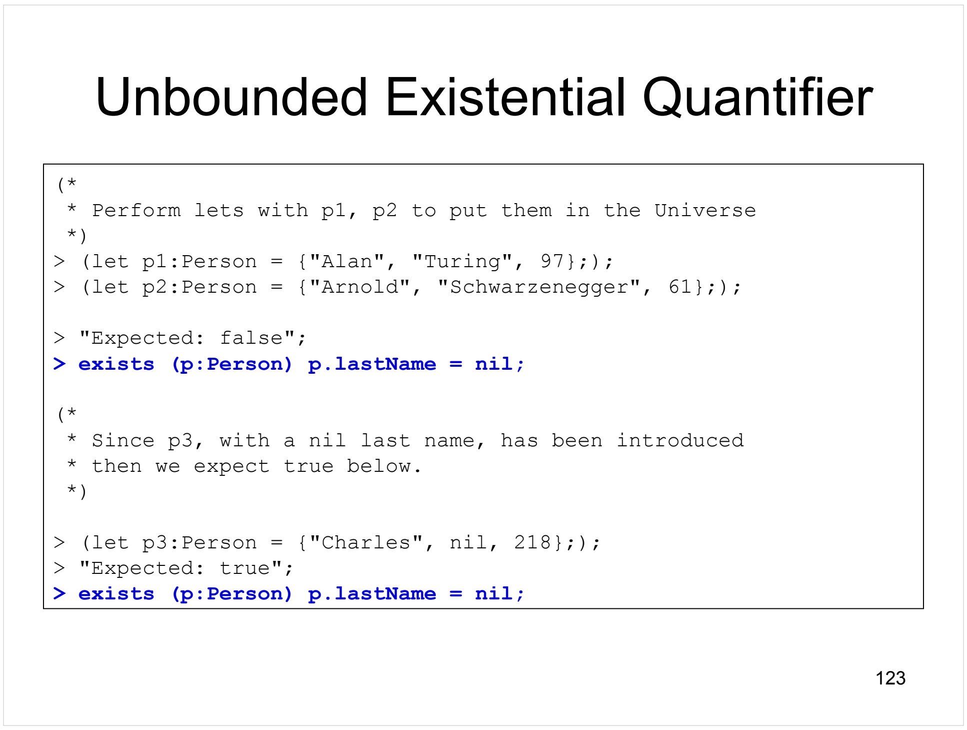
1. Look up the Person memory slot by hashing the Person type name to an index location
2. If such a slot doesn’t exist, assign one and create a value list
3. If it does exist, traverse through the Person value list to see whether the value already exists
4. If the value of interest exists, do nothing. If it does not exist, add it to the end of the value list.



Quantifier Syntax

Syntax	Quantifier Type	Reads Like …
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true

We're going to walk through an example that takes this form of unbounded existential quantifier

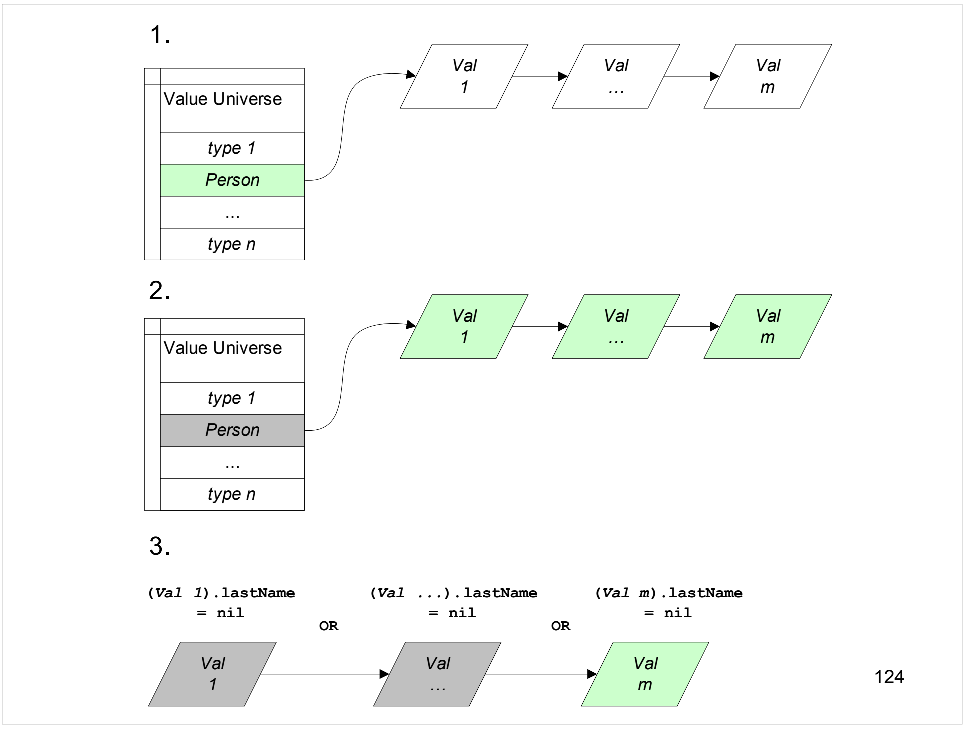


Unbounded Existential Quantifier

```
(*
 * Perform lets with p1, p2 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);

> "Expected: false";
> exists (p:Person) p.lastName = nil;

(*
 * Since p3, with a nil last name, has been introduced
 * then we expect true below.
 *)

> (let p3:Person = {"Charles", nil, 218};);
> "Expected: true";
> exists (p:Person) p.lastName = nil;
```

These existential quantifiers evaluate that there exists an object p of type person such that p’s last name field is nil.

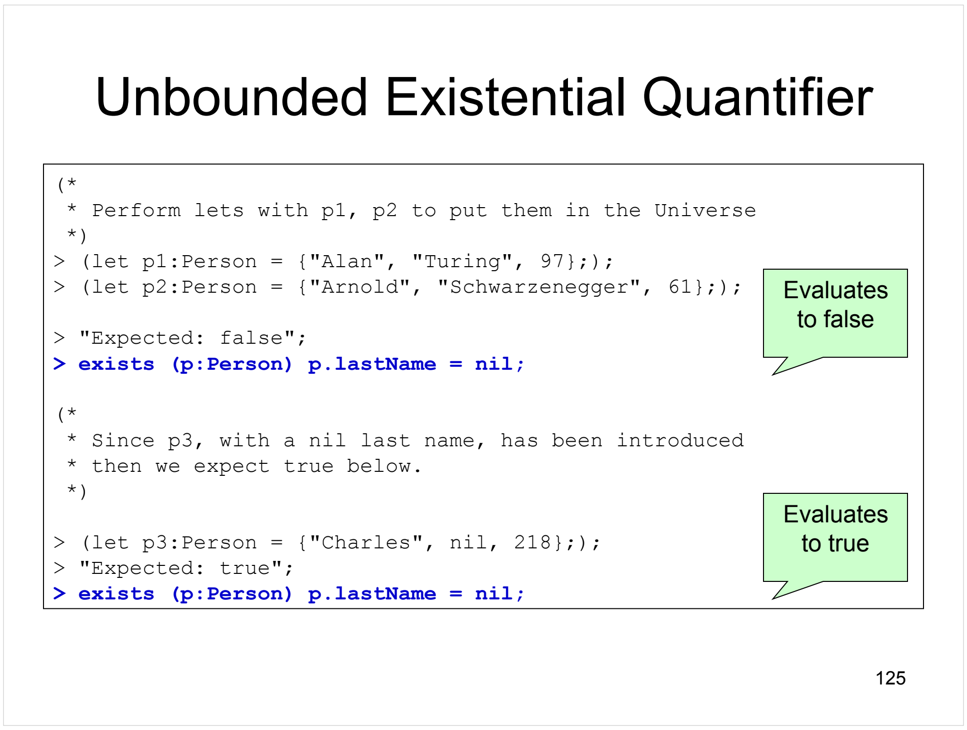
Notice that upon executing the first unbounded existential quantifier, all the person values introduced into the universe have non-nil last names. The same can’t be said upon executing the second quantifier.



Somewhat generically, here's how the unbounded quantifier, **exists (p:Person) p.lastName = nil**, gets evaluated:

3. The interpreter first identifies that p is of object type Person
4. It then Hashes the Person type name to locate the slot in the value universe where person values should be found
5. After discovering that there are Person values in the universe, the interpreter iterates through each Person value, temporarily assigning the current Person value to p in the local symbol table.
6. At each stop along the way, the interpreter evaluates the predicate, p.lastName = nil, and ORs the results together to arrive at the final evaluation result

A forall evaluation happens in a very similar fashion, except that the predicate evaluation results are ANDd together instead of ORd to arrive at the final result.



Unbounded Existential Quantifier

```
(*
 * Perform lets with p1, p2 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);

> "Expected: false";
> exists (p:Person) p.lastName = nil;

(*
 * Since p3, with a nil last name, has been introduced
 * then we expect true below.
 *)

> (let p3:Person = {"Charles", nil, 218};);
> "Expected: true";
> exists (p:Person) p.lastName = nil;
```

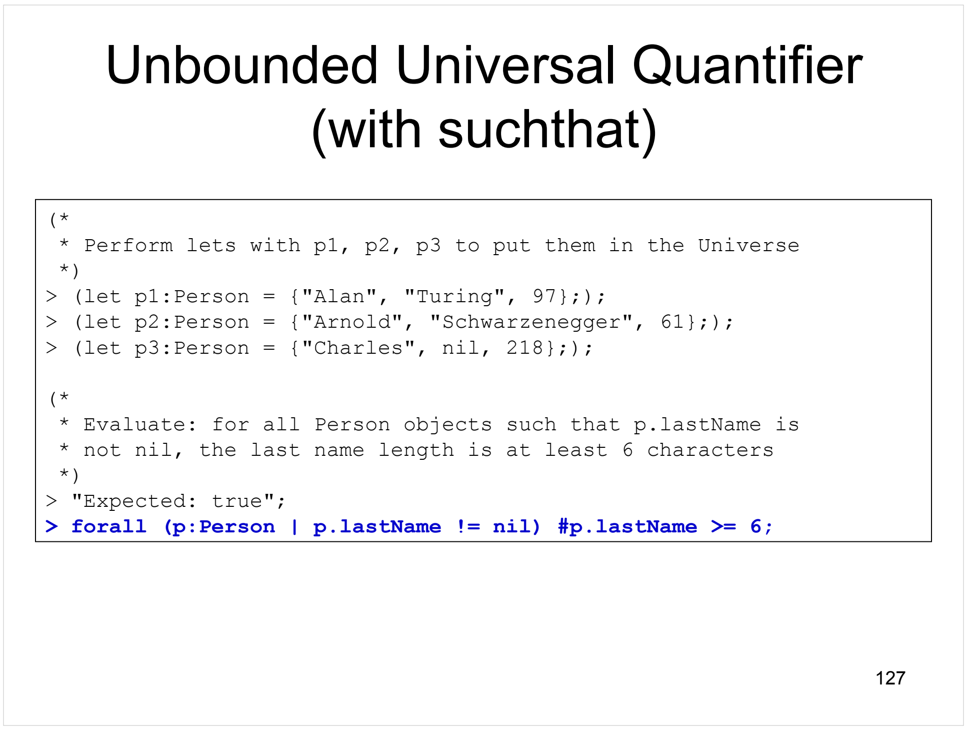
Evaluates to false

Evaluates to true

Knowing the implementation of the value universe, what we expected in each case was what we saw. First false, second true.

Quantifier Syntax

Syntax	Quantifier Type	Reads Like …
forall (*x* in *S*) *p*	bounded	for all values *x* in list *S*, *p* is true
forall (*x*:*t*) *p*	unbounded	for all values *x* of type *t*, *p* is true
forall (*x*:*t* | *p1*) *p2*	unbounded	for all values *x* of type *t* such that *p1* is true, *p2* is true
exists (*x* in *S*) *p*	bounded	there exists an *x* in list *S* such that *p* is true
exists (*x*:*t*) *p*	unbounded	there exists an *x* of type *t* such that *p* is true
exists (x:*t* | *p1*) *p2*	unbounded	there exists an *x* of type *t* such that *p1* is true and *p2* is true



Unbounded Universal Quantifier (with suchthat)

```
(*
 * Perform lets with p1, p2, p3 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);
> (let p3:Person = {"Charles", nil, 218};);

(*
 * Evaluate: for all Person objects such that p.lastName is
 * not nil, the last name length is at least 6 characters
 *)
> "Expected: true";
> forall (p:Person | p.lastName != nil) #p.lastName >= 6;
```

This unbounded universal quantifier evaluates that all Person objects that have a last name defined have a last name length of at least 6 characters. Or, in other words, forall objects p of type person, such that p's last name field is not nil, the length of the last name is at least 6 characters.

Like in the above existential quantifier example, the interpreter iterates over person values in the value universe, or p1, p2, and p3. The last names of p1 and p2 both are at least 6 characters, and although p3's last name not at least 6 characters, that's OK since we have our suchthat clause which precludes that character length evaluation.

Unbounded Universal Quantifier (with suchthat)

```
(*
 * Perform lets with p1, p2, p3 to put them in the Universe
 *)
> (let p1:Person = {"Alan", "Turing", 97};);
> (let p2:Person = {"Arnold", "Schwarzenegger", 61};);
> (let p3:Person = {"Charles", nil, 218};);

(*
 * Evaluate: for all Person objects such that p.lastName is
 * not nil, the last name length is at least 6 characters
 *)
> "Expected: true";
> forall (p:Person | p.lastName != nil) #p.lastName >= 6;
```

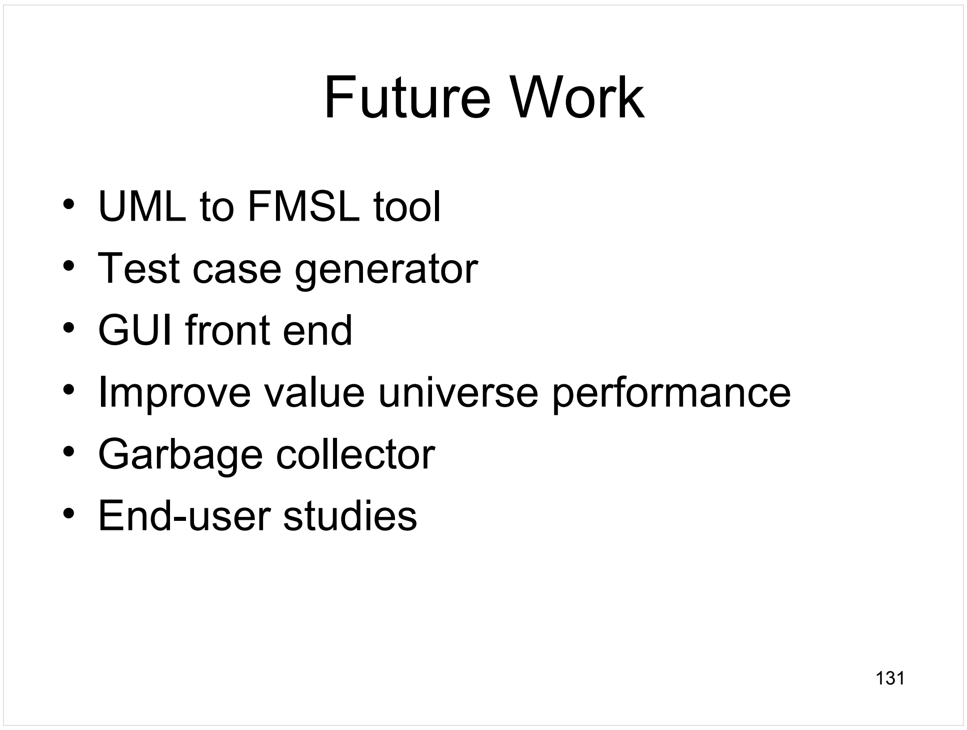
Evaluates to true

Presentation Outline

- Chapter 1: Introduction
- Chapter 2: Background and Related Work
- Chapter 3: Demonstration of Tool Capabilities
- Chapter 4: Overall System Design
- Chapter 5: The Functional Interpreter
- Chapter 6: Quantifier Execution
- Chapter 7: Conclusions

Summary of Contributions

1. The design and implementation of a functional interpreter for a formal specification language, rendering the language executable for the first time
2. The design and implementation of a novel technique to execute purely predicative specifications, using validation operator invocations
3. Demonstration of how the execution capabilities can be used to validate formal specifications
4. A thorough discussion of how the specification execution capabilities fit into the realm of light-weight and heavy-weight formal methods



Future Work

- UML to FMSL tool
- Test case generator
- GUI front end
- Improve value universe performance
- Garbage collector
- End-user studies

As UML is the standard for modeling software applications, a UML front-end for creating FMSL models could speed up the FMSL formal description process.

Currently FMSL validation operator test cases must be generated by hand, but combining automated test case generation with FMSL's specification execution capabilities could make FMSL an even more useful tool for validating specifications.

A GUI front end that helps the user to manage a specification's test suite could help speed up and streamline the test case generation and evaluation process.

As the values in the universe are maintained in a simple linked list structure, the implementation could be modified to use a structure that improves performance when checking for duplicates during value adds; one such structure could involve a hashing scheme.

As the current FMSL implementation does not manage memory very carefully, an improvement could be to utilize a third-party C-based garbage collector.

To assess the efficacy of the incremental validation tool, groups of tool users should be studied. For example, one section could use the validation tool while another would not, and student specifications could be assessed quantitatively and qualitatively to determine their accuracy, completeness, consistency, and soundness.

Questions?

- Click to add an outline

The End