Engineering Specifications and Mathematics for Verified Software

Hampton Smith
Clemson University
May 16th, 2013

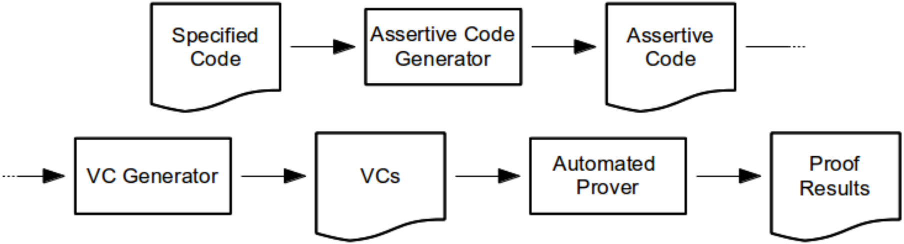
Committee
Murali Sitaraman (Chair)
Brian C. Dean
Jason O. Hallstrom
Roy P. Pargas

Layout

What is verified software?

- Mathematically prove properties of a program
  - No null dereferences
  - No buffer overflows
  - No deadlock
  - Termination
  - Full behavior
- Requires formal semantics
- Description of the desired behavior in a formal language
- Can be demonstrated by hand or mechanically

How do we verify?



Example Systems

Practical Systems

- Existing industrial languages (C, Java)
- Limited mathematical language
- Focus on verifying narrow properties
- Automatic proofs
- Accomplishments: Automatically verified linked data structures
- Example systems: Jahob, Verifast

Pure Systems

- Research or pure mathematical language
- Rich mathematical language
- Full verification (up to termination)
- Interactive proofs
- Accomplishments: Interactively verified C compiler, OS kernel
- Example systems: Coq, Issabelle

Jahob Example

```
public boolean contains(Object elem)
    /*: requires "init"
        ensures "(result = (EX i. (i, elem) : content))";
    */
    {
        int index = indexOfInt(elem);
        //: noteThat PosIndex: "0 <= index --> ((index,elem) : content)";
        //: noteThat NegIndex: "index = -1 --> ~(EX i. (i,elem) : content)";
        //: noteThat IndexLemma: "0 <= index | index = -1";
        boolean res = (0 <= index);
        /*: note ResultLemma: "res = (EX i. (i, elem) : content)"
                from PosIndex, NegIndex, IndexLemma;
        */
        return res;
    }
```

Coq Example

```
Definition divPre (args:nat*nat) : Prop := (snd args)<>0.

Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
    let (n, d):=args in let (q,r):=res in q*d+r=n /\ r<d.

Function div (p:nat*nat) {measure fst} : nat*nat :=
    match p with
    | (_,0) => (0,0)
    | (a,b) => if le_lt_dec b a
               then let (x,y):=div (a-b,b) in (1+x,y)
               else (0,a)
    end.

Theorem div_correct : forall(p:nat*nat),
    divPre p -> divRel p (div p).
```

Coq Example

```
unfold divPre, divRel.
intro p.
functional induction (div p); simpl.
intro H; elim H; reflexivity.
replace (div (a-b,b)) with
    (fst (div (a-b,b)),snd (div (a-b,b))) in IHp0.
simpl in *.
intro H; elim (IHp0 H); intros.
split.
change (b + (fst (x,y0)) * b + (snd (x,y0)) = a).
rewrite <- el.
omega.
change (snd (x,y0)<b); rewrite <- e1; assumption.
symmetry; apply surjective_pairing.
auto.
Qed.
```

Best of Both Worlds?

- Practical Systems
  - Flexible, integrated specification
  - Component support
  - Automatic proofs
- Pure Systems
  - Modular mathematics and specifications
  - Protection from certain complications (preferably still with the flexibility to use them)
  - Rich, extensible mathematical language

Problem Statement

- Architecture and implementation of a minimalist rewrite prover to explore those prover capabilities practically necessary to mechanically verify well-engineered, modular components.
- Design and implementation of an extensible, flexible supporting mathematical framework for a practical verification system that permits reuse as well as the development of a rich set of models and assertions.
- Design and implementation of a well-integrated specification framework that is explicitly designed to work with the mathematical system, supporting verifiability by allowing simple, flexible specifications and supporting scalability by encouraging verified component reuse.
- Validation of our central hypothesis via application of the minimalist prover to software constructed using the mathematical and specification framework.

Dissertation Goal

In a verification system, an extensible, flexible mathematics and specification subsystem enables better-engineered component specifications and thus more straightforward proof obligations that are easily dispatched by even minimalistic automated provers. Design, development, and experimentation with such a verification system is the goal of this dissertation.

Layout

Minimalist Prover: Motivation

- Prover is the final phase of the verification pipeline
- Sole determiner of which VCs can or can not be automatically proved
- Nearly all verification efforts focused on more efficient provers
- Our hypothesis suggests that, in many cases, *flexibility* may trump raw performance

Minimalist Prover: Contributions

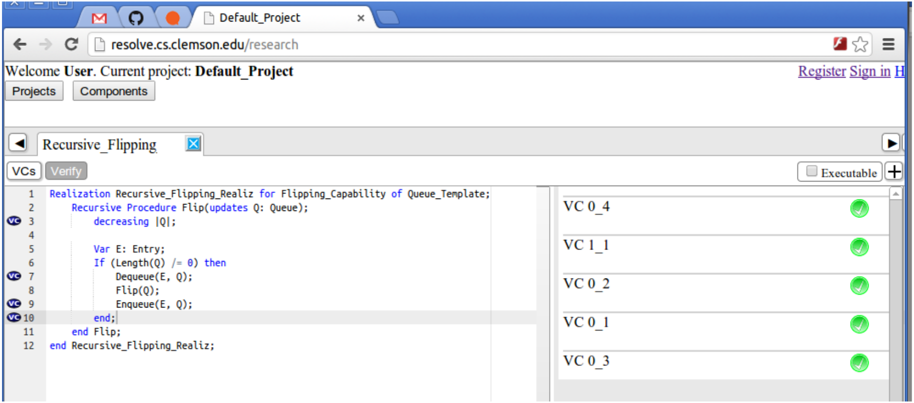
- We demonstrate how a number of components can be engineered in a rigorous style to ease the verification process
- We experiment with a suite of prover heuristics intended to expose the programmer's underlying logic
- We confirm empirically that well-designed components built on an expressive mathematical framework can be dispatched by a minimalist prover
- Among these components, we present a mechanically verified generic sorting algorithm—a first
- For full details, see Chapter 7

Minimalist Prover: Design

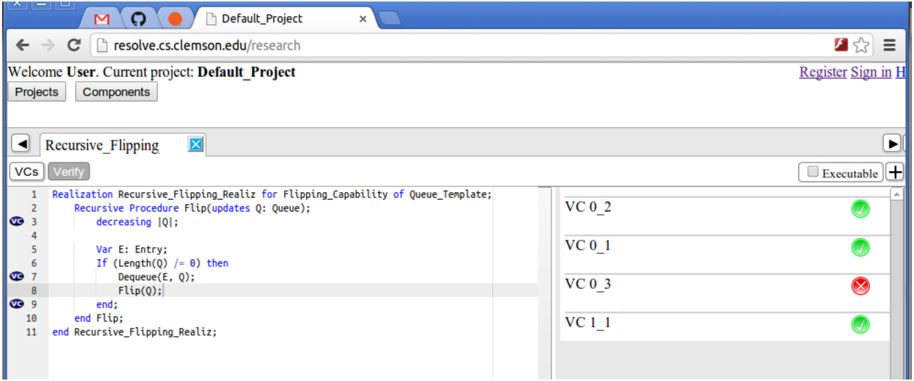
- Simple rewrite prover
- New features only when justified by VCs from real verification problems
- Flexible for experimentation
- Data collection for comparison
- Pedagogical deployments

Demo: Flipping a Queue

Demo: Flipping a Queue



Demo: Flipping a Queue



Automated Prover Algorithm

1. Expand variables
   `i = j - 1`
2. Develop antecedent
   `A and (A implies B) implies B`
3. Explore consequent
   - Tethered depth-first search
     Prevents `i < j implies i < j + 1` or `Reverse(Empty_String) = Empty_String` from being applied ad nausium
   - Terminates when proof space is exhausted or all consequents are dispatched

Automated Prover Algorithm

- Extremely straightforward
- Similar to how a human mathematician might perform a proof
- Many irrelevant antecedent developments are likely to be made
- Antecedent development happens once, up-front, so time is less of an issue, but space complexity is combinatorial
- During consequent exploration, full proof space must be searched. Combinatorial time complexity is a problem.

Automated Prover Algorithm: Heuristics

- Detect and avoid useless transformations
  `S → S o Empty_String`
- Develop only about relevant terms
  `f(a) and g(b) implies h(b)`
- Diversify givens
- Minimize as a preprocessing step
- Detect cycles
- Prioritize transformations as a preprocessing step
  1. Reduce unique symbols
  2. Reduce function applications

Experimental Evaluation: Overview

- Questions
  - Is such a minimalist prover practical?
  - Are the heuristics effective?
- Approaches
  - Series of verification benchmarks over multiple domains: integers, arrays, queues, trees
  - Collect metrics
  - How effective is the prover at dispatching VCs in a reasonable amount of time?
  - What causes VCs not to prove?
  - How does disabling each heuristic impact verification metrics?
- For full details, see Chapter 7

Experimental Evaluation: Metrics

- VCs proved
- Real time
- Operative steps
- Search steps (subset of operative steps occurring during consequent exploration)

Sorting a Queue

Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

```
Enhancement Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
                  Queue_Template;
            uses String_Theory, Total_Preordering_Theory;
            requires Is_Total_Preordering(LEQV);

      Operation Sort(updates Q : Queue);
            ensures for all i, j : Z where 0 < i < j <= |Q|,
                           LEQV(Element_At(i, Q), Element_At(j, Q)) and
                  for all e : Entity, Occurs_Ct(e, Q) = Occurs_Ct(e, #Q);

end;
```

Sorting a Queue

Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

```
Enhancement Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
                        Queue_Template;
                uses String_Theory, Total_Preordering_Theory;
                requires Is_Total_Preordering(LEQV);

        Operation Sort(updates Q : Queue);
                ensures Is_Conformal_With(LEQV, Q) and Is_Permutation(#Q, Q);

end;
```

Sorting a Queue

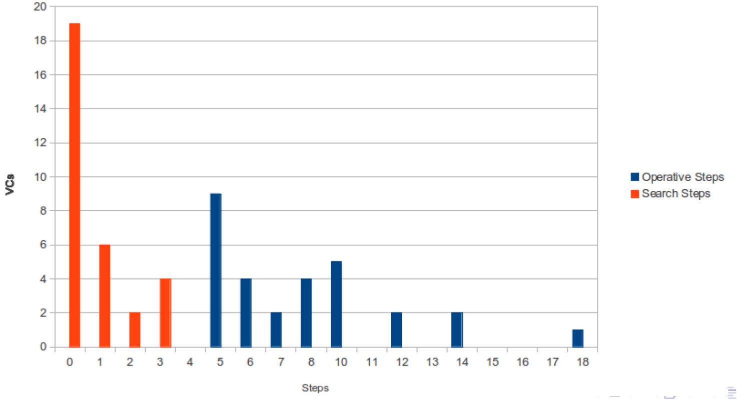
```
Operation Remove_Min(updates Q : Queue; replaces Min : Entry);
        requires |Q| /= 0;
        ensures Is_Permutation(Q o <Min>, #Q) and
                Is_Universally_Related(<Min>, Q, LEQV) and
                |Q| = |#Q| - 1;
Procedure
        Var Considered_Entry : Entry;
        Var New_Queue : Queue;

        Dequeue(Min, Q);
        While (Length(Q) > 0)
                changing Q, New_Queue, Min, Considered_Entry;
                maintaining Is_Permutation(
                                New_Queue o Q o <Min>, #Q) and
                        Is_Universally_Related(<Min>, New_Queue, LEQV);
                decreasing |Q|;
        do
                Dequeue(Considered_Entry, Q);

                if (Compare(Considered_Entry, Min)) then
                        Min :=: Considered_Entry;
                end;

                Enqueue(Considered_Entry, New_Queue);
        end;

        New_Queue :=: Q;
end;
```

Sorting a Queue

```
Procedure Sort(updates Q : Queue);
        Var Sorted_Queue : Queue;
        Var Lowest_Remaining : Entry;

        While (Length(Q) > 0)
                changing Q, Sorted_Queue, Lowest_Remaining;
                maintaining Is_Permutation(Q o Sorted_Queue, #Q) and
                        Is_Conformal_With(LEQV, Sorted_Queue) and
                        Is_Universally_Related(Sorted_Queue, Q, LEQV);;
                decreasing |Q|;
        do
                Remove_Min(Q, Lowest_Remaining);
                Enqueue(Lowest_Remaining, Sorted_Queue);
        end;

        Q :=: Sorted_Queue;
end;
```

Sorting a Queue

- VCs proved: 16/16
- Mean time: 1781 ms
- Median proof steps: 8
- Median search steps: 0



Demo: Array Realization of a Stack

Array Realization of a Stack

Specify a user-defined LIFO stack ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an array implementation of that ADT.

Array Realization of a Stack

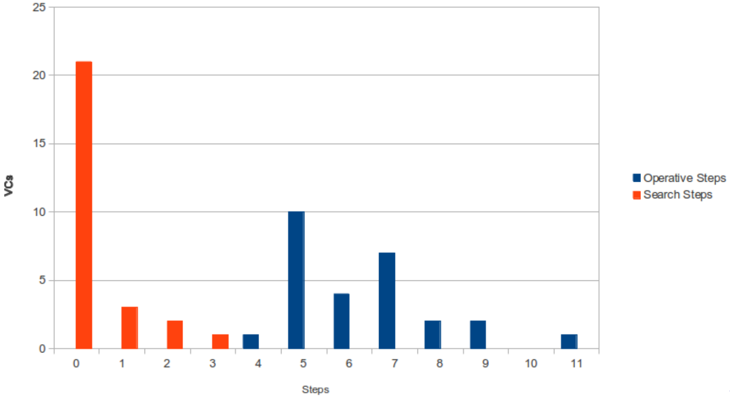
```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory, Integer_Theory;
        requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = Empty_String;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures  S = <#E> o #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires |S| /= 0;
        ensures #S = <R> o S;

    Operation Depth(restores S: Stack): Integer;
        ensures Depth = (|S|);

    Operation Rem_Capacity(restores S: Stack): Integer;
        ensures Rem_Capacity = (Max_Depth - |S|);

    Operation Clear(clears S: Stack);

end;
```

Array Realization of a Stack

```
Realization Array_Realiz for Stack_Template;
          uses Binary_Iterator_Theory;

     Type Stack is represented by Record
          Contents: Array 1..Max_Depth of Entry;
          Top: Integer;
     end;
     convention (* representation invariant *)
          0 <= S.Top <= Max_Depth;
     correspondence (* abstraction function *)
          Conc.S = Reverse(Concatenate(S.Contents, S.Top));

     Procedure Push(alters E: Entry; updates S: Stack);
          S.Top := S.Top + 1;
          E :=: S.Contents[S.Top];
     end;

     Procedure Pop(replaces R: Entry; updates S: Stack);
          R :=: S.Contents[S.Top];
          S.Top := S.Top - 1;
     end;

     Procedure Depth(preserves S: Stack): Integer;
          Depth := S.Top;
     end;

     ...
end;
```

Array Realization of a Stack

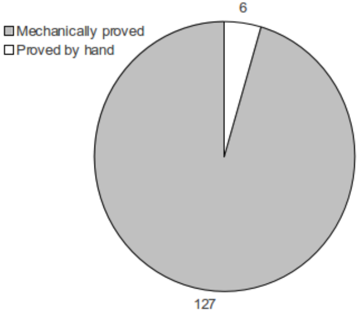
- VCs proved: 27/27
- Mean time: 1707.1 ms
- Median proof steps: 6
- Median search steps: 0



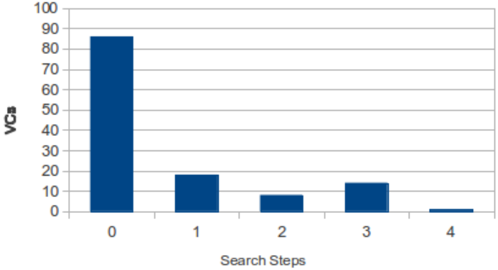
Overall Experimentation Results

- Six representative examples over integers, arrays, stacks, queues, and trees:
  - Add/Multiply Integers
  - Binary Search Array
  - Sort a Queue
  - Flip a Queue
  - Array Implementation of Stack
  - Modify and Restore a Tree
- VCs proved: 127/133
- Mean time: 2493 ms
- Median proof steps: 7
- Median search steps: 0

Overall Experimentation Results



(a) Verification results



(b) Number of Search Steps Required by Proofs

Heuristic Evaluation

	$\sum \Delta$Proved	$\overline{\Delta t/\sigma}$	$\sum \Delta t$	$\overline{\Delta\text{steps}}$	$\sum \Delta$steps	$\overline{\Delta\text{search}}$	$\sum \Delta$search
With useless transformations	-12	4.07	83438	0	0	0	0
Developing about irrelevant terms	-1	8.80	154530	0	0	0	0
Not checking for diversity of givens	-6	-5.55	-142026	-0.02	-4	-0.01	-1
No minimization	-10	2.53	36651	0.02	3	0.28	35
No cycle detection	0	0.60	9629	0.08	11	0.08	11
No prioritization of transformations	-19	2.60	17577	0.05	6	0.04	5

Layout

1. Introduction
   - Example Systems
   - Problem Statement
2. Minimalist Prover
   - Research
   - Evaluation
3. **Mathematical Flexibility**
   - Research
   - Evaluation
4. Conclusions and Future Directions

Mathematical Flexibility: Motivation

- Mathematical system is the language of specification
- It is the source of an increase in effort for verified software
- Therefore, it needs to be familiar and its results reusable
- Pure systems contain many useful features that practical systems do not take advantage of

Mathematical Flexibility: Contributions

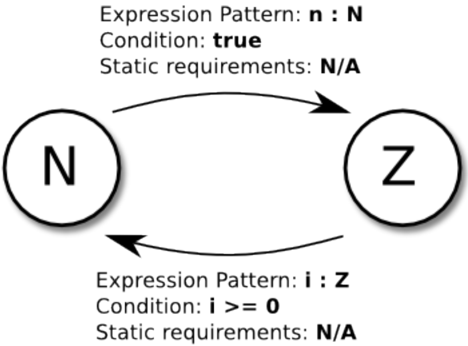
- We demonstrate how a number of features from pure systems (higher-order definitions, first-class types, etc.) can be utilized to ease the verification task in a practical system
- We provide a mathematical foundation for several preexisting RESOLVE features
- We introduce novel tools for static reasoning in the presence of dependent types
- For full details, see Chapter 5

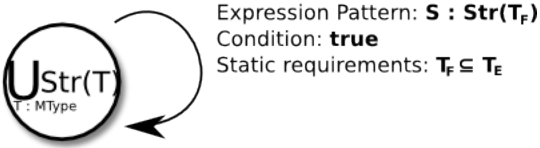
Tools for Static Reasoning

- First-class types permit undecidable type relationships
- Nonetheless, static typing is a useful tool
- *Type theorems* are a novel compromise introduced by this research

```
Definition Str : MType -> MType = ...;

Type Theorem Str_Subset_Theorem:
        For all T1 : MType,
        For all T2 : Powerset(T1),
        For all S : Str(T2),
                S : Str(T1);
```

Tools for Static Reasoning





Sorting a Queue

```
Enhancement Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
            Queue_Template;
      uses String_Theory, Total_Preordering_Theory;
      requires Is_Total_Preordering(LEQV);

      Operation Sort(updates Q : Queue);
            ensures Is_Conformal_With(LEQV, Q) and Is_Permutation(#Q, Q);

end Sorting_Capability;
```

Sorting a Queue

```
Precis String_Theory;

    --The type of all strings of heterogenous type
    Definition SStr : MType = ...;

    --A function that restricts SStr to the type of all strings of some homogenous
    --type
    Definition Str : MType -> MType = ...;

    Type Theorem All_Strs_In_SStr:
        For all T : MType,
        For all S : Str(T),
                S : SStr;

    --If R is a subset of T, then Str(R) is a subset of Str(T)
    Type Theorem Str_Subsets:
        For all T : MType,
        For all R : Powerset(T),
        For all s : Str(R),
                s : Str(T);

    Definition Empty_String : SStr = ...;

    Type Theorem Empty_String_In_All_Strs:
        For all T : MType,
                Empty_String : Str(T);

    --String length
    Definition |(s : SStr)| : N = ...;

  ...
end;
```

Sorting a Queue

```
Precis String_Theory;
    ...

        --String concatenation
        Inductive Definition (S : SStr) o (T : SStr): SStr is
                (i)  S o Empty_String = S;
                (ii) For all e : Entity, S o ext(T, e) = ext(S o T, e);

        Type Theorem Concatenation_Preserves_Generic_Type:
                For all T : MType,
                For all U, V : Str(T),
                        U o V : Str(T);

    ...
end;
```

Sorting a Queue

```
Goal:
Is_Universally_Related(<Considered_Entry'>, (New_Queue'  o <Min'>), LEQV)

Given:
(((((((((Last_Char_Num  > 0) and
(((min_int  <= 0) and
(0  < max_int)) and
((Max_Length  > 0) and
((min_int  <= Max_Length) and
(Max_Length  <= max_int))))) and
Is_Total_Preordering(LEQV)) and
(Entry.is_initial(Min) and
((|Q|  <= Max_Length) and
|Q|  /= 0))) and
Q = (<Min''>  o Q''')) and
(Is_Permutation(((New_Queue'  o Q'')  o <Min'>), Q) and
Is_Universally_Related(<Min'>, New_Queue', LEQV))) and
(|Q''|  > 0)) and
Q'' = (<Considered_Entry'>  o Q')) and
LEQV(Considered_Entry', Min'))
```

Sorting a Queue

```
For all f : (Entity * Entity) -> B,
For all E1, E2 : Entity,
For all S : String,
      Is_Total_Preordering(f) and
      f(E1, E2) and
      Is_Universally_Related(<E2>, S, f)
            implies Is_Unversally_Related(<E1>, S);
```

Error Analysis and Reporting

Array Realization of a Stack

```
Realization Array_Realiz for Stack_Template;
            uses Binary_Iterator_Theory;

        Type Stack is represented by Record
                        Contents: Array 1..Max_Depth of Entry;
                        Top: Integer;
                end;
                convention (* representation invariant *)
                        0 <= S.Top <= Max_Depth;
                correspondence (* abstraction function *)
                        Conc.S = Reverse(Concatenate(S.Contents, S.Top));

    ...
end;
```

Array Realization of a Stack

```
Theory Binary_Iterator_Theory;
            uses Integer_Theory, String_Theory;

        Definition Iterative_Apply(Step : ((Range : MType) * (V : MType)) -> Range,
                Start : Range, Value_Function : Z -> V, Value_Count : Z) : Range;

        Definition Concatenate(Value_Function : Z -> (T : MType),
                        Value_Count : Z) : Str(T) =
                Iterative_Apply(lambda (s : Str(T), t : T).(s o <t>),
                        Empty_String, Value_Function, Value_Count);

    ...
end;
```

Error Analysis and Reporting

Classroom Experiment

- Mathematical development assignment given to a graduate-level programming languages class for extra-credit
- Example demonstrating first-class types and type theorems
- No formal training
- Assignment asked increasingly difficult questions: last three required analysis and adaptation
  1. Asserts that `Without_Last_Zero(10) = 1`. This may require an extra step to establish proper symbols.
  2. Asserts that for any multiple of ten, `t`, `Next_Even(t) mod 10 = 2`. This may require some additional steps to establish proper symbols and relationships.
  3. Asserts that for all multiples of ten, `t`, and integers, `i`, `Without_Last_Zero(t * i) = i`. This may require some additional steps.

Classroom Experiment

- 7/9 students participated
- All but one student successfully completed questions 10 and 11 correctly
- Two of the seven completed 12 correctly

Layout

Conclusion

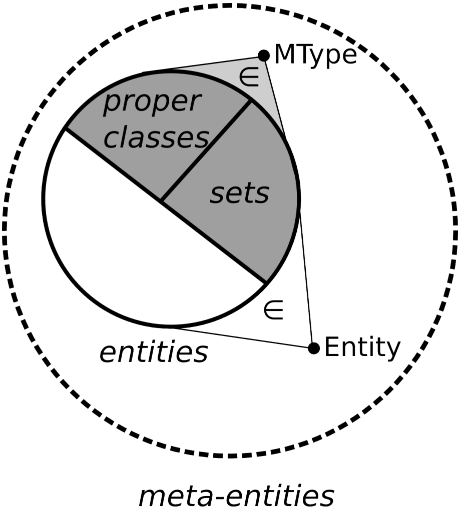
- Systems need not be limited to the features of a pure or a practical system. Our hybrid system incorporates features of practical verification systems (static checking, efficient implementation, polymorphism) with pure mathematical systems (dependent types, higher-order logic, mathematical reusability.)
- Novel mechanism for static reasoning can be used to bridge the gap between undecidable, but flexible type systems and constrained, hierarchical systems.
- It can be demonstrated empirically that using such a language, a programmer is capable of creating components about which reasoning is sufficiently easy that VCs can be dispatched by a minimalist prover.
- This includes a verified generic sorting algorithm—a first.
- A variety of useful heuristics exist to help a minimalist prover expose programmer intuition.

Future Directions

- Better transformation fitness functions
- Other prover styles
- Evaluate usability of new features
- Increase type-system intuitiveness
- Prover scalability

Questions?

Type Universe



Specification Style

```
Operation Dequeue(replaces R: Entry; updates Q: Queue);
        requires |Q| /= 0;
        ensures #Q = <R> o Q;
```

```
Operation Dequeue(replaces R: Entry; updates Q: Queue);
        requires |Q| /= 0;
        ensures R = Element_At(1, #Q) and
                Q = Substring(#Q, 2, |#Q| - 1);
```

Implicit

	Time (ms)	σ	Steps	Search
VC 0_1	1520	141	5	0
VC 0_2	3118	295	7	0
VC 0_3	2741	222	8	0
VC 0_4	2174	170	9	2
VC 1_1	366	83	10	0

Figure: Recursive `Flipping_Capability` results

Explicit

	Time (ms)	σ	Steps	Search
VC 0_1	2199	160	5	0
VC 0_2	1468	182	6	0
VC 0_3	2149	253	8	0
VC 0_4	1589	225	8	3
VC 1_1	845	139	10	0

Figure: Recursive `Flipping_Capability` results, based on an explicitly-specified queue

Comparison

- Explicit 1669 ms longer
- Saved two proof steps
- Cost one search step