A Type-Based Approach to Verified Software

Aaron Stump
Dept. of Computer Science
The University of Iowa
Iowa City, Iowa, USA
Acknowledgments

- Ting Zhang for this invitation.
- U. Iowa Computational Logic Center:
  - Faculty: AS, Cesare Tinelli.
  - Postdocs: Garrin Kimmell, Tehme Kahsai.
  - Doctoral: F. Fu, T. Liang, J. McClurg, C. Oliver, D. Oe, A. Reynolds.
  - Undergraduates: JJ Meyer.

http://clc.cs.uiowa.edu

- NSF: CAREER, Trellys grant.
About This Talk

Part 1: The Verification Renaissance.
Part 2: Type-based Verification in GURU.
Part 3: versat, a Verified Modern SAT Solver.
Part 4: Glimpse Ahead.
Verification Reborn

Language-Based Verification Will Change the World,
Computing systems are doing so much:

Why can’t we guarantee they work?
Why not just use testing?

+ Integrates well with programming.
+ No new languages, tools required.
+ Conclusive evidence for bugs.
Why not just use testing?

+ Integrates well with programming.
+ No new languages, tools required.
+ Conclusive evidence for bugs.

– Difficult to assess coverage.
– Cannot demonstrate absence of bugs.
– No guarantees for safety-critical systems.

Alternative: Formal Verification
Instead of tests, use proofs.

- Deduction and proof provide universal guarantees.
- Prove that software has specified properties.
- From this...

“seL4: formal verification of an OS kernel”, Klein et al., SOSP 2009.
Proofs and Size of Systems

- **seL4 microkernel (mobile phones):**
  - Around 9,000 lines of code.
  - 200,000 lines of computer-checked proof, written by hand.
  - Isabelle proof tool.
  - My estimate: 1 line of proof = 10 lines of code.
  - So equivalent to 2M lines of code.

- **Airbus A380:**
  - Millions of lines of code.
  - cf. Mercedes S-class: 100M lines of code.
  - Astrée can analyze 100Kloc programs.

Why the difference in scale?
Traditionally, Two Kinds of Computer Proof

1. Automated Theorem Proving (Astrée).
   - Fully automatic.
   - Shallow reasoning, but
   - Large formulas.

2. Computer-Checked Manual Proof (Isabelle)
   - Written by hand.
   - Needed for deep reasoning.
   - Use solvers to fill in easy parts.
Large formulas (50 megabytes or more).
Large formulas (50 megabytes or more).
Programs as Proofs?

- Solvers test huge formulas.
- So solvers must be very efficient.
- So solvers must be complicated.
- What if the solver is wrong?
- Who watches the watchers?
Programs as Proofs?

- Solvers test huge formulas.
- So solvers must be very efficient.
- So solvers must be complicated.
- What if the solver is wrong?
- Who watches the watchers?

*We will return to this with versat.*
Type-Based Verification in Guru

Between Heaven and Hell
If you dislike proofs:

Heaven: Fully automatic solvers
Hell: Manual proof

If you like specification:

Heaven: Expressive language, rich specifications
Hell: Impoverished language
If you dislike proofs:

<table>
<thead>
<tr>
<th>Heaven</th>
<th>Fully automatic solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hell</td>
<td>Manual proof</td>
</tr>
</tbody>
</table>

If you like specification:

<table>
<thead>
<tr>
<th>Heaven</th>
<th>Expressive language, rich specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hell</td>
<td>Impoverished language</td>
</tr>
</tbody>
</table>

Earth:

Rich specifications => manual proof.
Automatic solvers => weak specifications.
How can we combine solvers and rich specifications?
How can we combine solvers and rich specifications?

Two traditional answers:

1. Use solvers for easy parts of manual proofs (ISABELLE, COQ).
2. Pose intermediate lemmas, to prove automatically (ACL2).
Manual Proof as External Verification

Manual proof:

2 artifacts: proof and program.

Proof is external to program.
An Alternative

External verification:

program → proof
An Alternative

External verification:

program \rightarrow proof

Internal verification:

1 artifact: program with proofs inside.

Proof is internal to program.
External verification:

append : Fun(A:type)(l1 l2 : <list A>). <list A>

length_append :
  Forall(A:type)(l1 l2:<list A>).
  { (length (append l1 l2)) = (plus (length l1) (length l2)) }

Internal verification:

<vec A n> – type for lists of As of length n.

append :
  <vec A (plus n m)>

These are dependent types.
Advantage: Internal Verification

- **Annotate instead of prove.**
  - Sprinkle annotations just where needed.
  - External proofs must consider even irrelevant code.

- **Verify less.**
  - Theorem provers usually require totality.
  - Can be a major proof obligation (or even false).
  - Dependently typed PLs do not.

- **Control usage.**
  - Dependent types great for software protocols.
    - open (read|write)* close.
    - cf. FINE [Chen, Swamy, Chugh, PLDI 2010]
    - also ensuring in-bounds array access: `read a i P`.
  - No so easy to verify externally.
Tour-de-force verification is powerful, extremely costly.
Verification is much more than tour-de-force!
Internal verification of lighter properties can go mainstream.
Continuum of correctness:

| Type Safety | High Quality | Tour-de-force Verification |

Let programmer find the sweet spot.
Proofs and Programs in Guru

- Polymorphic higher-order functional programs.
  - Indexed algebraic datatypes, pattern-matching.
  - Dependent types.
  - General recursion.

- First-order proofs with induction.
  - Structural induction on datatypes.
  - Quantify over program types, not formulas.
  - Includes some non-constructive principles.
    - Case split on termination of a term.

Aaron Stump
Types for Verified Software
Iowa State, 2011
Mutable State

- How to incorporate mutable state (like arrays)?
- Simple idea: functional modeling.
  - Define inefficient functional model.
  - Swap out during compilation.
- Arrays modeled as vectors.

\[ <\text{array } A \ w> \implies <\text{vec } A \ (\text{word}_\text{to}_\text{nat} \ w)> \]

- Require proofs for array accesses.
- How to ensure soundness with destructive update?
- *Resource typing*: statically track memory, no GC.
The **Guru Compiler** ([www.guru-lang.org](http://www.guru-lang.org))

Guru source code → Parser → Type/proof-checker → Pull out λs → Resource analysis → Linearization → Compile datatypes → C target code

**CARRAWAY Layer**

No GC!
versat

A Verified Modern SAT Solver

Main developer: Duckki Oe
versat Overview

- Modern SAT solver with all the trimmings.
  - clause learning.
  - watched literals.
  - optimized conflict analysis.
  - non-chronological backtracking.

- Implemented in GURU.

- Statically verified sound.
  - If versat says unsat
  - Then input formula is contradictory.

- Efficient.
  - Uses standard efficient data structures.
  - Can handle formulas on modern scale (10k vars, 100k clauses).
  - First efficient verified solver.

- Around 8kloc code and proofs.
  - Cf. Paper by Filip Marić 2010, 25kloc ISABELLE.
Main Specification

- The `solve` function has type:
  \[ \text{Fun}(F:\text{formula})(\ldots).<\text{answer } F> \]

- `formula` is list of list-based clauses.

- `answer` records proof for `unsat` case:
  \[
  \text{Inductive } \text{answer} : \text{Fun}(F:\text{formula}).\text{type} := \\
  \quad \text{sat} : \text{Fun}(\text{spec } F:\text{formula}).<\text{answer } F> \\
  \mid \text{unsat} : \text{Fun}(\text{spec } F:\text{formula})(\text{spec } p:<\text{pf } F (\text{nil lit}>) ).<\text{answer } F>
  \]

- `pf` is a simple indexed datatype of resolution proofs.

- We have proved that a resolution proof exists.

- Not constructed at run-time.
Other Properties

**Verified:**
- Connection between array-based, list-based clauses.
- Array-accesses in bounds.
- No leaks, double deletes (resource typing).

**Not verified:**
- Completeness.
- Termination.
  - Other approaches require this.
  - Uninteresting in practice, due to NP-completeness.
## Empirical Evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>File Size</th>
<th>Answer</th>
<th>versat</th>
<th>minisat</th>
<th>tinisat</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE09-07</td>
<td>442K</td>
<td>S</td>
<td>125.26</td>
<td>8.53</td>
<td>0.89</td>
</tr>
<tr>
<td>countbitsrotate016</td>
<td>82K</td>
<td>U</td>
<td>114.20</td>
<td>34.17</td>
<td>29.61</td>
</tr>
<tr>
<td>een-tipb-sr06-par1</td>
<td>8.8M</td>
<td>U</td>
<td>7.06</td>
<td>0.74</td>
<td>0.59</td>
</tr>
<tr>
<td>een-tipb-sr06-tc6b</td>
<td>2M</td>
<td>U</td>
<td>2.71</td>
<td>0.18</td>
<td>0.13</td>
</tr>
<tr>
<td>grieu-vmpc-s05-24s</td>
<td>905K</td>
<td>S</td>
<td>756.54</td>
<td>8.56</td>
<td>20.04</td>
</tr>
<tr>
<td>grieu-vmpc-s05-25</td>
<td>0.9M</td>
<td>S</td>
<td>372.37</td>
<td>19.29</td>
<td>186.77</td>
</tr>
<tr>
<td>gss-14-s100</td>
<td>1.5M</td>
<td>S</td>
<td>673.45</td>
<td>29.02</td>
<td>6.71</td>
</tr>
<tr>
<td>gus-md5-04</td>
<td>4.0M</td>
<td>U</td>
<td>35.69</td>
<td>2.27</td>
<td>7.81</td>
</tr>
<tr>
<td>icbrt1_32</td>
<td>494K</td>
<td>U</td>
<td>30.66</td>
<td>7.41</td>
<td>30.51</td>
</tr>
<tr>
<td>manol-pipe-c10id_s</td>
<td>9.4M</td>
<td>U</td>
<td>800.27</td>
<td>1.23</td>
<td>3.01</td>
</tr>
<tr>
<td>manol-pipe-c10ni_s</td>
<td>11M</td>
<td>U</td>
<td>13.81</td>
<td>2.02</td>
<td>6.83</td>
</tr>
<tr>
<td>stric-bmc-ibm-10</td>
<td>6.1M</td>
<td>S</td>
<td>730.29</td>
<td>0.53</td>
<td>0.78</td>
</tr>
<tr>
<td>vange-col-inithx.i.1-cn-54</td>
<td>8.9M</td>
<td>S</td>
<td>48.42</td>
<td>1.10</td>
<td>1.90</td>
</tr>
</tbody>
</table>
Next Steps for **versat**

- Performance improvements.
- Prove some remaining lemmas.
  - Currently proved 136 lemmas.
  - 68 unproved.
  - About specificational functions.
- What can you do with a verified SAT solver?
- On Duckki Oe’s homepage (Projects – versat):
  - **Guru** code for **versat-0.4**.
  - Generated C code.
Glimpse Ahead

2. Equality, Quasi-Implicit Products, and Large Eliminations.
Trellys

U. Penn.  Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg
Iowa AS, Harley Eades, Frank Fu
PSU  Tim Sheard, Ki Yung Ahn, Nathan Collins

- Large NSF project, 2009-2013.
- New dependently typed PL called TRELLYS.
- Improves on GURU, related languages:
  - Much more powerful type system for programs.
  - Eliminate even more termination requirements.
  - Aiming for elegant surface language.
Conclusion

- Type-based approach to verified software.
- Guru verified-programming language.
- Case study: versat.
- First verification of efficient modern SAT solver.
- Future work: keep exploring this rich area!
- Slides online at my blog, QA9:
  
  queuea9.wordpress.com

Thank you again!