Building Verified Software with Dependent Types

Aaron Stump

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

- Sriram 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, StarExec grant.
About This Talk

- Part 1: The GURU dependently typed programming language.
- Part 2: Case study on versat, verified modern SAT solver.
- Part 3: Glimpse ahead.
GURU and Dependent Types

What is the Appeal of Dependent Types?

- Lots of tour-de-force verification happening.
  - CompCert verified C compiler (42kloc CoQ).
  - seL4 verified microkernel (200kloc ISABELLE).
  - Metatheory of Standard ML (30kloc Twelf).
  - Total correctness of a modern SAT solver (Marić, 25kloc ISABELLE).

- Dependent types are much lighter.
  - versat only 7.8K Guru, verified sound.

Why?
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)>
Advantage: Dependent Types

- 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.
Verification: Less is More

- Tour-de-force verification is powerful, extremely costly.
- Verification is much more than tour-de-force!
- Verification of lighter properties can go mainstream.
- Continuum of correctness:

<table>
<thead>
<tr>
<th>Type Safety</th>
<th>High Quality</th>
<th>Tour-de-force Verification</th>
</tr>
</thead>
</table>

- Let programmer find the sweet spot.
Anatomy of a Dependently Typed PL

- Programs vs. proofs.
- General recursion.
- Specificational data.
- Equality.
- Mutable state.
- Compilation.
- Automation.

Consider Guru’s approach.

www.guru-lang.org
Programs and Proofs

- Need notation for proofs.
  - Sometimes external theorem is most natural.
  - For example, associativity of append.
  - Also for type equivalences.

- One solution: Curry-Howard.

\[ \text{proofs} \quad = \quad \text{programs} \]
Programs and Proofs

- Need notation for proofs.
  - Sometimes external theorem is most natural.
  - For example, associativity of append.
  - Also for type equivalences.

- One solution: Curry-Howard.

* Cute, but not a good idea.
  - Not every program makes sense as a proof.
    - loop : False
  - Not every proof makes sense as a program.
    - non-constructive proofs cannot be executed.
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.
Equality and Casts

- Can change type of a term with a cast.

\[
\Gamma \vdash t : T_1 \quad \Gamma \vdash P : T_1 = T_2 \\
\frac{}{\Gamma \vdash \text{cast } t \text{ by } P : T_2}
\]

- Example:
  - Have \( l : \langle \text{vec } A \ (x+y) \rangle \)
  - Want \( \langle \text{vec } A \ (y+x) \rangle \)
  - Use:
    \[
    \text{cast } l \text{ by cong } \langle \text{vec } A \ * \rangle \ [\text{plus_comm } x \ y]
    \]

- Casts erased during compilation.
- Also for proving equations.
  - Avoids need for \textit{axiom } K, proving proofs equal.
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.

\[
<\text{array } A \ w> \implies <\text{vec } A \ (\text{word\_to\_nat } w)>
\]

- Require proofs for array accesses.
- How to ensure soundness with destructive update?
Resource Typing

- Additional analysis beyond regular type-checking.
- Tracks all memory statically: no GC!
- Limitations:
  - Dag-like immutable state: OK.
  - Unaliased mutable state: OK.
  - Aliased mutable state: No.
- Reference counting for dag-like data.
- Linear restriction for mutable data.
- Notion of pinning helps:
  - If $x : T$ and $y$ pointing into memory reachable from $x$.
  - Then $y : \langle x \rangle T$.
  - $y$ is pinning $x$.
  - Must consume $y$ before $x$. 

Aaron Stump
Verified Software with Dependent Types
CU Boulder, 2011
The **GURU Compiler** ([www.guru-lang.org](http://www.guru-lang.org))

<table>
<thead>
<tr>
<th>Guru source code</th>
<th>Parser</th>
<th>Type/proof-checker</th>
<th>Pull out $\lambda$s</th>
<th>Resource analysis</th>
<th>Linearization</th>
<th>Compile datatypes</th>
<th>C target code</th>
<th><strong>CARRAWAY Layer</strong></th>
<th>No GC!</th>
</tr>
</thead>
</table>

Aaron Stump

Verified Software with Dependent Types

CU Boulder, 2011
versat

A Verified Modern SAT Solver

Main developer: Duckki Oe

Under review for SAT 2011.
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 \texttt{versat says unsat}
  - Then input clause is contradictory.
- Efficient.
  - Uses standard efficient data structures.
  - Can handle formulas on modern scale (10k vars, 100k clauses).
  - Not competitive with state of the art yet.
Main Specification

- The `solve` function has type:
  
  \[
  \text{Fun}(\text{nv:word}) \times (\text{nv_ub:} \{ (\text{ltword nv var_upper_bound}) = \text{tt} \}) \times (\text{F:formula}). <\text{answer F}>
  \]

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

- `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.
  - Would have to show recursions terminate.
  - Also that some run-time checks never fail.
  - Would be very difficult.
Verifying Optimized Conflict Analysis

- Compute useful learned clause from contradiction.
  - Done by optimized resolution.
    - Table-based algorithm.
    - No intermediate clauses.
    - Most difficult verification in versat.
    - Around 6 invariants.

- Example theorem: efficient table-cleanup.

Define `cl_has_all_vars_implies_clear_vars_like_new`:

```plaintext
Forall (nv:word)
  (vt:<array assignment nv>)
  (c:clause)
  (u:{ (cl_valid nv c) = tt })
  (r:{ (cl_has_all_vars c vt) = tt })
  {. (clear_vars vt c) = (array_new nv UN) } := ...
```
## 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 112 lemmas.
  - 79 unproved.
  - About specificational functions.
- What can you do with a verified SAT solver?
  - One idea: compress SAT part of SMT proofs.
  - Others?
- On Duckki Oe’s homepage (Projects – versat):
  - Guru code for versat-0.4.
  - Generated C code.
Glimpse Ahead
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.
  - Much more expressive logic.
  - Aiming for elegant surface language.
Blaise

- Garrin Kimmell, JJ Meyer, Austin Laugesen.
- Resource typing for aliased mutable state.
  - Goal: no GC!
  - Approach: statically enforce a memory-usage protocol.
  - Spanning tree on every data structure.
  - Reciprocal back pointer for every alias pointer.
  - Clean up aliasing cells on deletion.

Why is GC bad?
  - Performance hit.
  - Nightmare to engineer in compiler (see HASKELL).
Conclusion

- Verified programming with dependent types.
- Guru language design.
- 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!