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.
  - Master's: E. Bavier, H. Eades, T. Jensen, A. Laugesen, CJ Palmer.
  - 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

1. *Verified Programming in Guru*, PLPV 2009.
2. *Resource Typing in Guru*, PLPV 2010.

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?

Internal vs. External Verification

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 `A`s of length `n`.

```
append :
  Fun(A:type)(spec n m:nat)(l1 : <vec A n>)(l2 : <vec A m>).
    <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

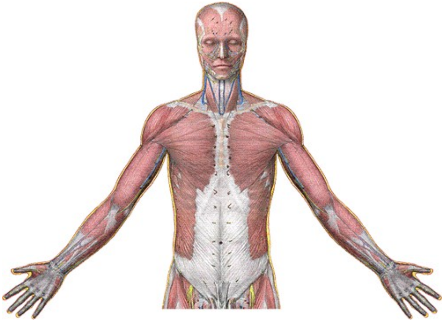
- Tour-de-force verification is powerful, extremely costly.
- Verification is much more than tour-de-force!
- Verification of lighter properties can go mainstream.
- Continuum of correctness:

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

- Let programmer find the sweet spot.

Anatomy of a Dependently Typed PL

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

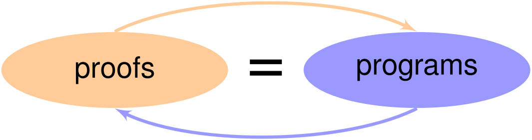


Consider GURU's approach.

`www.guru-lang.org`

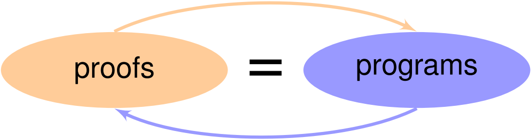
Programs and Proofs

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



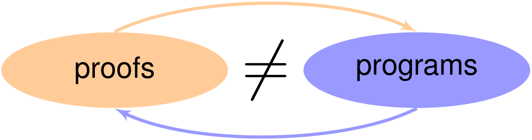
Programs and Proofs

- 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.

$$\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash P : T_1 = T_2}{\Gamma \vdash \texttt{cast}\ t\ \texttt{by}\ P\ :\ T_2}$$

- Example:
  - Have `l : <vec A (x+y)>`
  - Want `<vec A (y+x)>`
  - Use:

```
cast l by cong <vec A *> [plus_comm x y]
```

- Casts erased during compilation.
- Also for proving equations.
  - Avoids need for *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.

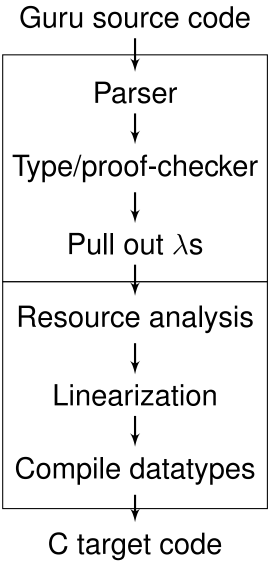
```
<array A w>  ⟹  <vec A (word_to_nat w)>
```

- 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:<x>T.
  - y is pinning x.
  - Must consume y before x.

The GURU Compiler (`www.guru-lang.org`)



CARRAWAY Layer

No GC!

versat

A Verified Modern SAT Solver

Main developer: Duckki Oe

Under review for SAT 2011.

`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 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:

```
Fun(nv:word)
   (nv_ub:{ (ltword nv var_upper_bound) = tt })
   (F:formula).
<answer F>
```

- `formula` is list of list-based clauses.
- `answer` records proof for `unsat` case:

```
Inductive answer : Fun(F:formula).type :=
  sat  : Fun(spec F:formula).<answer F>
| 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.
- <u>Not</u> 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 :
  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

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

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

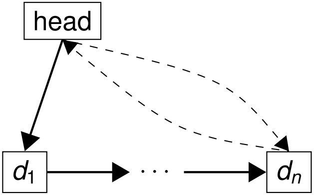
Trellys

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!