Building Verified Language Tools in Operational Type Theory

Aaron Stump

Computational Logic Center
Computer Science Department
The University of Iowa

Thanks to Morgan Deters and Todd Schiller.

Funding from NSF CAREER.

From Meta-Theory to Tools

- Mechanized meta-theory great.

- Verified language tools also great!

- The combination definitely the greatest.

Meta-theory and Tools for LF

- Paper meta-theory for LF [Harper+05],[Watkins+02].
- Machine-checked meta-theory for LF [Urban+08].
- Unverified tools for LF: TWELF, FLIT, SC, LFSC.
- Verified tool (this talk): GOLFSOCK.
  - Verify that optimized LF checker builds type-correct LF.
  - Uses a declarative presentation of LF.
  - Still efficient, but much more trustworthy.
  - Partial verification.

Incremental Checking

- Basic idea: interleave parsing and checking [Stump08].
- Combine with bidirectional type checking.
  - Synthesizing: $\Gamma \vdash t \Rightarrow T$.
  - Checking: $\Gamma \vdash t \Leftarrow T$.
- ASTs built for subterms iff they will appear in the type T. E.g.,

```
(refl x+y) => x+y == x+y
```

  - AST must be built for `x+y`.
  - But not `(refl x+y)`.
- C++ implementation: small footprint, fastest checker I know.

A Need for Correctness

- LF with Side Conditions (LFSC) proposed for SMT.
  - Satisfiability Modulo Theories.
  - SMT solvers check large formulas, produce big proofs.
  - Must check proofs efficiently.
  - LFSC provides flexible intermediate proof language.
  - Extends LF with computational side conditions.
- Problems with C++ checker:
  - Lack of memory safety => many days with `valgrind`.
  - Optimizations reduce trustworthiness.
- As features are added to checker, trust diminishes.
- Additional assurance is required.

Towards A Verified LFSC Checker

- GOLFSOCK ("GURU LFSC").
  - GURU is a verified functional programming language.
  - Supports mutable state, non-termination, I/O.
  - Verification via dependent types, induction proofs.
  - Type/proof checker, compiler to efficient C code.
  - Beating native code OCAML on small testcases.
- Status:
  - Incremental LF checking implemented.
  - Running reasonably fast: 50% slower than C++ version.
  - Specification: ASTs we build are type correct LF.
  - Expressed with dependent types, declarative LF.
  - 4300 lines code, proof; 13000 lines standard library (e.g., tries).

GURU and Operational Type Theory

- GURU implements Operational Type Theory (OpTT).
- OpTT is new type theory intended to:
  - Combine programming, theorem proving (cf. ATS, Epigram, Ynot).
  - Allow general recursion, other effects.
  - Retain sound logic.
  - Retain decidability of type checking.
  - Support external reasoning about dependently typed programs.
  - Support compilation to efficient executables.
- Critical design idea: separate different reductions.
  - Reduction for definitional equality ($\equiv$).
  - Reduction for programs.
  - Normalization (aka, cut elimination) for proofs.

Rejection of Curry-Howard

- Proofs $\neq$ Programs, Formulas $\neq$ Types.

terms : types

`fun(A:type)(x:A).x` : `Fun(A:type)(x:A).A`

proofs : formulas

`foralli(x:nat).truei` : `Forall(x:nat).True`

- Otherwise non-terminating programs $=$ unsound proofs.

Rejection of Conversion

- Definitional equality ($\equiv$) cannot include program reduction.
- Otherwise type checking undecidable.
- Adopt a very weak $\equiv$ ($\equiv_\alpha$, definitions, sugar).
- Constrast with strong conversion relations.
  - CIC: $\equiv$ includes $\equiv_\beta$, terminating recursion.
  - CCIC: $\equiv$ uses decision procedures, hypotheses.
- With conversion, lose definitional transparency.
- Typing holds modulo $\equiv$, but not other operations.
  - $\Gamma \vdash t : T \quad \texttt{=>} \quad \Gamma' \vdash t' : T'$ with $\Gamma \equiv \Gamma'$, $t \equiv t'$, $T \equiv T'$.
  - Rewriting modulo $\equiv_\beta$ only recently decidable [Stirling06].
  - In Coq, many tactics do not work modulo $\equiv$.
  - In GURU, all tactics work modulo $\equiv$.

Operational Equality

- Due to weak $\equiv$, need casts in code (and proofs):

$$\frac{\texttt{t} : \texttt{T}_1 \quad \texttt{P} : \{\texttt{T}_1 \;=\; \texttt{T}_2\}}{\texttt{cast t by P} : \texttt{T}_2}$$

- Reasoning about code with casts tedious in other systems.
- In OpTT, reason about unannotated programs.
  - Propositional equality `{ t = t' }` holds if $t \downarrow t'$.
  - No type annotations, casts, proofs in `t`, `t'`.
  - No *specificational data*.
  - Vastly simplifies external reasoning about code.
  - Annotations dropped by definitional equality.

Example: Vector Append

```
Inductive vec : Fun(A:type)(n:nat).type :=
  vecn : Fun(A:type).<vec A Z>
| vecc : Fun(A:type)(spec n:nat)(a:A)(l:<vec A n>).
              <vec A (S n)>.

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

vec_append_assoc :
  Forall(A:type)(n1 : nat)(l1 : <vec A n1>)
        (n2 n3 : nat)(l2 : <vec A n2>)(l3 : <vec A n3>).
  { (vec_append (vec_append l1 l2) l3) =
    (vec_append l1 (vec_append l2 l3)) }
```

Functional Modeling and Ownership

- Following [Swierstra+07]: awkwardness => modeling school.
- Awkward squad modeled functionally.
  - Standard input is a list of chars.
  - `getc()` is head.
  - Mutable arrays of length n are vectors of length n.
  - `read` and `write` are pure, $O(n)$ operations.
- Reason about code using functional model.
- Replace during compilation with non-functional implementation.
- Restrict usage for soundness (monads or linear types).
- GURU uses linear types.
  - Fit well with *ownership types*.
  - GURU statically tracks ownership of all data.
  - Enables reference counting for memory management.
  - Function inputs `unowned`, `owned`, `unique`, or `unique_owned`.

GOLFSOCK: Symbols

- Incrementally consume textual input, type check LF.
- LF variables (constants) implemented as 32-bit words.
  - Implementation with `nat` too slow.
  - Words are functionally modeled as `<vec bool 32>`.
  - Trusted operations: increment, equality check, create 0.
  - Reason via model, also via conversion to `nat`.
- Symbol table maps strings (lists of chars) to (var, type) pairs.
- Symbol table implemented as a trie.
- Mutable char-indexed arrays of subtries at each node.

GOLFSOCK: LF derivations

- Code “builds” specificational LF derivations.
- For $\Gamma \vdash t \Leftarrow T$ (or $\Gamma \vdash t \Rightarrow T$), we build `<deriv G t T>`.
- Context encoded as a list of (var,type) pairs.
- Must map the symbol table to context.
  - Difficult.
  - Must prove lemmas like trie membership => context membership.
  - Resulting context is not ordered.
  - Phrase typing rules for unordered contexts.
  - Ok, because vars uniquely named.

Empirical Results

benchmark	size (MB)	C++ impl	GOLFSOCK	TWELF
cnt01e	2.6	1.3	2.0	14.0
tree-exa2-10	3.1	1.7	2.5	18.6
cnt01re	4.6	2.4	3.6	218.4
toilet_02_01.2	11	5.8	8.8	1143.8
1qbf-160cl.0	20	10.0	14.1	timeout
tree-exa2-15	37	19.9	31.2	timeout
toilet_02_01.3	110	58.6	89.7	exception

Figure: Checking times in seconds for QBF benchmarks

- Good, since some optimizations not implemented.

Conclusion

- GOLFSOCK: towards verified, efficient language tools.
- OpTT makes this easier:
  - Not required *a priori* to prove termination.
  - Reason about code with annotations dropped.
  - Use dependent types for big functions (`check`, 1200 lines).
  - Supports functional modeling.
- Onward towards verified, efficient software!

www.guru-lang.org