Verified Programming in Guru

Aaron Stump[1] Morgan Deters[2] Adam Petcher[3]
Todd Schiller[3] Timothy Simpson[3]

[1]Computational Logic Center
CS, The University of Iowa

[2]LSI, Universitat Politècnica de Catalunya, Spain

[3]CSE, Washington University in St. Louis

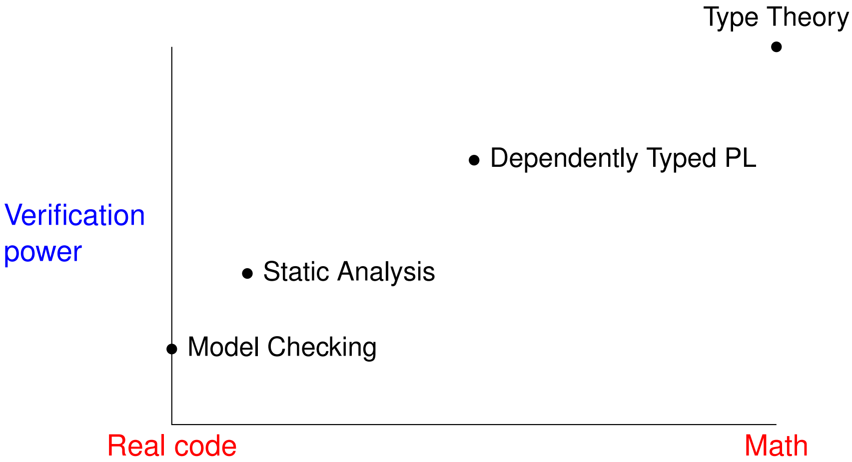
Funding from NSF CAREER.

A Vexing Continuum

Real code	Math. functions
concurrent	sequential
imperative	pure
general recursive	total

Where is your verification method?

Plotting Some Approaches



The GURU Approach

Real code	$\twoheadleftarrow$ GURU	Math. functions

General recursion
Dependently typed programs
External theorems about programs
Unaliased mutable state
No concurrency
No aliasing (for mutable state)

Basic GURU Design

- Terms : Types.
- Proofs : Formulas.
- “Full-spectrum” dependency.
  - Types can contain arbitrary terms.
  - Definitional equality very weak (no β).
  - Type checking decidable.
  - Explicit casts.
- Proofs and types can appear in terms.
  - computationally irrelevant.
  - erased by compilation, definitional equality.

- Today:
  - specificational data.
  - ownership and memory management.
  - functional modeling.

Specificational Data

- Programmer can designate argument positions `spec`.
  - for constructors, functions.
  - can use a `spec x` in a `spec` argument.
  - also in types, proofs.
  - nowhere else.
  - enforced separately from type checking.
- `spec` args erased by compilation [Brady+03], def. equality.
- Improves efficiency, simplifies proofs.

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)>
```

Compiled to C: `gvec gvec_append(gtype gA, gvec gl1, gvec gl2);`

```
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)) }
```

Memory Management in GURU

- Currently, no aliasing.
  - All data inductive.
  - Reference graph acyclic.
- Use reference counting, not GC.
- Programs use explicit `inc`, `dec`.
- Static analysis ensures no leaks, no double deletes.
- Analysis runs after type checking.
- Reduce need for `inc`/`dec` with ownership annotations.

Example: Filling a List

```
fun fill(A:type)(a:A)(n:nat):<list A>.
  match n with
    Z => (nil A)
  | S n' => (cons A a (fill A a n'))
  end.
```

- This type checks, but needs `inc/dec` to compile.
- By default, inputs `unowned` by caller.
- Function must consume each input exactly once.

Compilable Version

```
fun fill(A:type)(a:A)(n:nat):<list A>.
  match n with
    Z => dec a (nil A)
  | S n' => (cons A inc a (fill A a n'))
  end.
```

- `dec a t`: consume reference, evaluate `t`.
- `inc a`: create new reference.
- `n` is consumed by `match`.

A Different Version Using `owned`

```
fun fill(A:type)(owned a:A)(owned n:nat):<list A>.
  match n with
    Z => (nil A)
  | S n' => (cons A inc a (fill A a n'))
  end.
```

- `a`, `n` are `owned` by caller.
- Function must still `inc` for `cons` of `a`.
- No need to `dec a` in `Z` case.
- `match` does not consume `owned` `n`.
- `n'` automatically `owned` in second case.

Reference Counting Implementation

- One byte for constructor tag, three for reference count.
- When refcount = 0:
  - put item on per-constructor freelist.
  - $O(1)$ time.
- When allocating from free list:
  - `dec` subdata.
  - $O(d)$ time, where d is arity of constructor.
- Around 4x faster than `malloc`/`free`.
- For generic code:
  - pass `int` tags for types.
  - code for `inc`/`dec` indexed by tag.

Functional Modeling

- Awkward squad via functional modeling [Swierstra+07].
  - Identify interface.
  - Define pure functional model.
  - Use model for type checking, theorem proving.
  - Replace during compilation.
  - Use linear types (`unique`) to ensure equivalence.
- Examples in GURU:
  - Basic I/O.
  - 32-bit words with increment.
  - ASCII characters.
  - `char`-indexed mutable arrays.

Character-Indexed Mutable Arrays

- Model `charvec` as `<vec A 128>`.
- Interface is:

```
mk_charvec : Fun (A:type)(a:A):unique <charvec A>

cvget : Fun(A:type)(unique_owned l:<charvec A>)
           (c:char) : A

cvupdate : Fun(A:type)(c:char)(a:A)
              (unique l:<charvec A>) : unique <charvec A>.
```

- `cvget` does not consume the array.
- `cvupdate` does.

Future Work

- Goal: efficient verified FP with effects.
- So far:
  - general recursion
  - mutable structures
  - good performance via refcounting.
- Next up: aliasing.
  - idea: maintain a spanning tree of primary pointers.
  - these have type `unique <aliased A n>`.
  - `n` is number of outstanding aliases.
  - to traverse alias, shift primaries/aliases.
  - use a physical equality to prove equivalent.
  - eliminate shifting code during compilation.
- Version 1.0 is close to release:

  `guru-lang.googlecode.com`