Dependently Typed Programming with Mutable State

Aaron Stump¹  Evan Austin²

¹Computer Science
The University of Iowa

²Computer Science
The University of Kansas

U.S. National Science Foundation CAREER grant.
What Are Dependent Types?

- Indexed datatypes:
  - `<list A n>` instead of `<list A>
  - `<balanced_tree A d>` instead of `<tree A>
  - `<lam_t max_var>` instead of `lam_t`

- Dependent function types:
  ```
  (u:{(in x l) = tt}).
  <list A n>
  
  append : Fun(A:type)(n1 n2:nat)
  (l1:<list A n1>)(l2:<list A n2>).
  <list A (plus n1 n2>)
  ```

- Computing a type by recursion:
  ```
  printf : Fun(s:format_string).(printf_t s)
  
  (printf_t "%d"++s) => (int -> (printf_t s))
  (printf_t "%x"++s) => (ptr -> (printf_t s))
  (printf_t []) => unit
  ```
Why Dependent Types Matter

Incrementality

1Title of invited talk at POPL 2006 by James McKenna.
Why Dependent Types Matter

Incrementality       Intensionality

1Title of invited talk at POPL 2006 by James McKenna.
Incrementality

- Adding verification usually is a big leap.
  - new specification language (at least first-order logic); and
  - new proof language(s), or
  - unpredictable, tricky tools (“you need an expert”).

- Not a big leap with dependent types.
  - From `<list A>` to `<list A n>` is easier.
  - Add verification judiciously, “pay as you go”.

- Goal: enable gradual increase in code quality.
  - Deep verification is at one limit.
  - Lightweight verification can improve code a lot.
Intensionality (Policies versus Properties)

- Properties expressing facts about code.
- Policies restrict how code can be used.
- Stating (proving) a property from a policy may be hard.
- Example policies:
  - Files may not be accessed after they are closed.
  - Uninitialized array locations may not be read.
  - Data computed from user’s contact list cannot be auto-emailed. \(^2\)

\(^2\)See [Swamy, Chen, and Chugh 2009]
GURU at a High-Level

- Pure functional language + logical theory. \(^3\)
  - Includes indexed datatypes, dependent function types.
  - Terms : Types.
  - Proofs : Formulas.

- Inspired by Coq/CIC, but with some improvements:
  - General recursion for terms.
    - Proofs are still sound.
    - Explicit casts instead of conversion => type equivalence still decidable.
  - Annotations dropped for type equivalence.
    - Including types, specificational (“ghost”) data, and proofs.
    - Avoids problems with equality of proofs.
    - Like Implicit Calculus of Constructions (ICC).
  - Resource-tracking analysis [new!]

\(^3\)See [Stump, Deters, Petcher, Schiller, Simpson 2009]
Functional Modeling for Imperative Abstractions

- I/O, mutable arrays, cyclic structures, etc.
- Do not fit well into pure FP.
  - Define a pure functional model (e.g., `<list A n>` for arrays). 4
  - Model is faithful, but slow.
  - Use during reasoning.
  - Replace with imperative code during compilation.
  - Use *linear* (aka *unique*) types to keep in synch.

4Cf. [Swierstra and Altenkirch 2007]
Example: Word-Indexed Mutable Arrays

- **Type**: `<warray A N L>`.  
  - A is type of elements.  
  - N is length of array.  
  - L is list of initialized locations.

- `(new_array A N) : <warray A N []>`.  

- **Writing to index i**:  
  - requires proof: i < N.  
  - functional model: consume old array, produce updated one.  
  - imperative implementation: just do the assignment.  
  - array’s type changes: `<warray A N i::L>`.

- **Reading from index i**:  
  - does not consume array.  
  - requires proof: i ∈ L.
Example: FIFO Queues

- Mutable singly-linked list, with direct pointer to end.
- **Aliasing!**
- **GURU** approach: *heaplets* (part of heap).

<table>
<thead>
<tr>
<th>Type</th>
<th>Functional Model</th>
<th>Imperative Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;heaplet A I&gt;</td>
<td>list of aliased values</td>
<td>nothing</td>
</tr>
<tr>
<td>&lt;alias I&gt;</td>
<td>index into heaplet I</td>
<td>reference-counted pointer</td>
</tr>
</tbody>
</table>

- **Unverified queue:**
  - Just memory safety.
  - 138 lines total (6 lines proof).

- **Verified queue:**
  - Prove that $q_{in}$-node has no next-pointer.
  - Requires reasoning about aliases.
  - 310 lines total (178 lines proof).
Resource-Tracking and Memory Management

- Memory deallocated explicitly.
- Resource-tracking analysis ensures safety.
- Different resource types available.
  - `unowned`: for reference-counted data.
  - `unique`: for mutable data structures.
  - `<owned x>`: for pinning references.

```plaintext
x: unowned
y: <owned x>
```

Not allowed to consume `x` until `y` is consumed.

Can safely omit inc/dec for `y`.

- **Guru**: no garbage collection!
- “Garbage Collection: Java Application Servers Achilles’ Heel”

---

5[Xian, Srisa-an, Jiang 08]
Empirical Comparison

Benchmark 1: In array storing \([0, 2^{20})\), do binary search for each element.

Benchmark 2: push all words in “War and Peace” through 2 queues.

<table>
<thead>
<tr>
<th>Language</th>
<th>Avg Real Time</th>
<th>Language</th>
<th>Avg Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HASKEll</strong></td>
<td>1.18 s</td>
<td><strong>HASKEll</strong></td>
<td>1.08 s</td>
</tr>
<tr>
<td><strong>HASKEll (No GC)</strong></td>
<td>0.49 s</td>
<td><strong>HASKEll (No GC)</strong></td>
<td>0.53 s</td>
</tr>
<tr>
<td><strong>OCAML</strong></td>
<td>0.61 s</td>
<td><strong>OCAML</strong></td>
<td>0.66 s</td>
</tr>
<tr>
<td><strong>OCAML (No GC)</strong></td>
<td>0.54 s</td>
<td><strong>OCAML (No GC)</strong></td>
<td>0.37 s</td>
</tr>
<tr>
<td><strong>GURU</strong></td>
<td>0.42 s</td>
<td><strong>GURU</strong></td>
<td>0.60 s</td>
</tr>
</tbody>
</table>

Compilers: ghc 6.10.4, ocamlopt 3.11.1, gcc 4.3.3
Machine: 2.67Ghz Intel Xeon, 8 GB mem, Linux 2.6.18
Current Projects

- **versat**: verified modern SAT solver.
  - Complex code, uses mutable state.
  - Not too large.
  - Simple spec.: learned clauses derivable by resolution from input clauses.
  - With Duckki Oe, Derek Bruce.

- **GOLFSOCK**: verified LFSC proof checker.
  - LFSC = (Edinburgh) Logical Framework with Side Conditions.
  - My proposal for a meta-language for SMT proofs.
  - Fast C++ implementation (45% overhead for QF_IDL, difficulty 0-3). \(^6\)
  - With Cesare Tinelli, Clark Barrett, Tianyi Liang, Yeting Ge, Andrew Reynolds.

- Implementation in **GURU** in progress.

---

\(^6\)See [Oe, Stump, Reynolds 2009]
Current Projects

- **versat**: verified modern SAT solver.
  - Complex code, uses mutable state.
  - Not too large.
  - Simple spec.: learned clauses derivable by resolution from input clauses.
  - With Duckki Oe, Derek Bruce.

- **GOLFSOCK**: verified LFSC proof checker.
  - LFSC = (Edinburgh) Logical Framework with Side Conditions.
  - My proposal for a meta-language for SMT proofs.
  - Fast C++ implementation (45% overhead for QF_IDL, difficulty 0-3). \(^6\)
  - With Cesare Tinelli, Clark Barrett, Tianyi Liang, Yeting Ge, Andrew Reynolds.

- Implementation in **GURU** in progress.

- “Eat your own dog food!”

---

\(^6\)See [Oe, Stump, Reynolds 2009]
Current Projects

- **versat**: verified modern SAT solver.
  - Complex code, uses mutable state.
  - Not too large.
  - Simple spec.: learned clauses derivable by resolution from input clauses.
  - With Duckki Oe, Derek Bruce.

- **GOLFSOCK**: verified LFSC proof checker.
  - LFSC = (Edinburgh) Logical Framework with Side Conditions.
  - My proposal for a meta-language for SMT proofs.
  - Fast C++ implementation (45% overhead for QF_IDL, difficulty 0-3).  
  - With Cesare Tinelli, Clark Barrett, Tianyi Liang, Yeting Ge, Andrew Reynolds.

- Implementation in **GURU** in progress.

- “Eat your own dog food!”

- Let’s eat what we grow.

---

6See [Oe, Stump, Reynolds 2009]
Future Goals

- More imperative abstractions:
  - Statically reference-counted heaplets.
  - Doubly-linked lists, hashmaps, etc.

- More automation:
  - Currently: `hypjoin t t' by p1 ... pn end`\(^7\).
  - Extend to first-order formulas?
  - Goal: understandable, predictable tactics ("no expert needed").

- (For you) to learn more:
  - Version 1.0 is close to release:
    - `www.guru-lang.org`
  - "Verified Programming in Guru" book.

\(^7\)See [Petcher, Stump 2009].