Resource Typing in Guru

Aaron Stump\textsuperscript{1}  Evan Austin\textsuperscript{2}

\textsuperscript{1}Computer Science
The University of Iowa

\textsuperscript{2}Computer Science
The University of Kansas

U.S. National Science Foundation CAREER grant.
The GURU Verified-Programming Language

- Pure functional language + logical theory.
  - 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!]

Stump, Austin (Iowa, Kansas) Resources in Guru PLPV 2010
The Guru Compiler

Guru source code

Parser

Type/proof-checker

\lambda\text{-lifting}

Resource analysis

Linearization

Compile datatypes

C target code

CARRAWAY Layer
Functional Modeling for Imperative Abstractions

- I/O, mutable arrays, cyclic structures, etc.
- Do not fit well into pure FP.
- Approach: functional modeling. ¹
  - Define a pure functional model (e.g., `<list A n> for arrays`).
  - Model is faithful, but slow.
  - Use during reasoning.
  - Replace with imperative code during compilation.
  - Use *linear* types (alternatively, monads) to keep in synch.

Combining dependent and linear typing is powerful.
  - Cf. “Safe Programming with Pointers through Stateful Views” [Zhu,Xi 2005].
  - Also, “End-to-end Verification of Security Enforcement is Fine” [Swamy,Chen,Chugh 2009].

¹ Cf. “Beauty in the Beast” [Swierstra and Altenkirch 2007]
Idea: explore resource management with a framework. Framework implements concepts of resource, subresource. Different resource abstractions then defined:

- reference-counted data
- unique references
- heap abstractions
- read-only views

On top of these, build data abstractions:

- Mutable array abstractions.
- Aliased data structures (e.g., FIFO queues).
A Framework for Resources

Fundamental ideas:

1. A resource can only be used by one entity at a time.
2. A resource can be temporarily decomposed into subresources.

Resource abstraction defined by *primitives*:

- a trusted resource type,
- a functional model in Guru,
- trusted C code implementing the primitive.

Resource analysis:

- Check linearity conditions (used exactly once, affine).
- Track subresource relationships.
- Enforce *consumption annotations* on input variables:
  - ⋆ (default) – consume exactly once/affine.
  - ^ – consume but do not return.
  - ! – do not consume.
Subresources

- “Deathly Hallows” as subresource of Harry Potter boxed set.
- Cannot use boxed set until all individual volumes returned.
- Sublist \( l' \) as a subresource of \( (\text{cons } x \ l') \).
- Subresource relationship based on type \( <R \ x> \): 
  - \( x : R \) – \( x \) has resource type \( R \).
  - \( y : <R' \ x> \) – \( y \) has resource type \( R' \), and is a subresource of \( x \).
- Cannot consume \( x \) until all subresources have been consumed.
- Need \( ^\hat{\ } \) (“consume but do not return”) to consume \( y : <R' \ x> \).
Resource Abstraction: Reference-Counted Data

ResourceType unowned [...].

Define primitive inc
  : Fun(spec A:type)(! #unowned y:A).#unowned A
  := fun(A:type)(y:A).y
«END
  inline void *ginc(void *y) { [...] }  
END.

Define primitive dec
  : Fun(A:type)(^#unowned y:A).void
  := fun(A:type)(y:A).voidi
«END
  void gdec(int A, void *r) { [...] }  
END.

- Inductive (tree-like) data are reference-counted.
- (Flat types like bool are untracked.)
Resource Abstraction: Owned References

ResourceType owned affine.

Define primitive inspect
  : Fun(spec A:type)(!#unowned x:A).#<owned x> A
  := fun(A:type)(x:A).x
«END
  #define ginspect(x) x
END.

\[ x : A \]
\[ y : #<owned x> A \]

- This \( y \) is \textit{pinning} \( x \).
- Cannot consume \( x \) while \( y \) is live.
  - No inc, dec required for \( y \).
  - improved performance, still memory safe.
Mutable State and Readers/Writers

- For writing mutable state, require unique reference.
- Can implement readers/writers, using subresource idea.
  - Must check in the read-only views to get the read/write one.
  - For read/write, $x : \#\text{unique}$.
  - For read-only, $y : \#\langle\text{unique\_owned} \ x\rangle$.

- Use \texttt{unique/unique\_owned} for arrays, queues, tries, etc.
Data Abstraction: Word-Indexed Mutable Arrays

- **Type:** `<warray A N L>`.  
- **Resource types:** `unique/unique_owned`.  
  - `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`.  

---

Stump, Austin (Iowa, Kansas)  
Resources in Guru  
PLPV 2010
Data Abstraction: FIFO Queues

- Mutable singly-linked list, with direct pointer to enqueue-end.
- **Aliasing.**
- Resource abstraction: *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).
Garbage collection has led to great productivity gains...
... but can hurt performance.
No continuum in mainstream: all GC (slow) or no GC (unsafe).
Guru does not use GC.
- Resource abstractions are memory safe.
- But heaplet can leak memory for cyclic structures.
A perfect world might provide:
- GC’ed regions for productivity.
- Heavier abstractions for safety without GC.
  * E.g., compile-time reference counting.
  * Significant verification burden.
- Key: ability to choose which is more appropriate.
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>Mutable Array Test</th>
<th>Language</th>
<th>Time</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASKELL</td>
<td>1.18 s</td>
<td>581K</td>
<td></td>
</tr>
<tr>
<td>HASKELL (No GC)</td>
<td>0.49 s</td>
<td>131K</td>
<td></td>
</tr>
<tr>
<td>OCAML</td>
<td>0.61 s</td>
<td>37K</td>
<td></td>
</tr>
<tr>
<td>OCAML (No GC)</td>
<td>0.54 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GURU</td>
<td>0.42 s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue Test</th>
<th>Language</th>
<th>Time</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASKELL</td>
<td>1.08 s</td>
<td>614K</td>
<td></td>
</tr>
<tr>
<td>HASKELL (No GC)</td>
<td>0.53 s</td>
<td>132K</td>
<td></td>
</tr>
<tr>
<td>OCAML</td>
<td>0.66 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCAML (No GC)</td>
<td>0.37 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GURU</td>
<td>0.60 s</td>
<td>37K</td>
<td></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
Implementations: Data.Sequence (HASKELL), references (OCAML).
Future Directions

- Better abstractions for aliased structures.
- Realistic applications.
  - **versat**: verified modern SAT solver.
    - Complex code, uses mutable state.
    - Not too large.
    - Simple spec.: learned clauses derivable by resolution from input clauses.
- Meta-theoretic work on resources.
- To learn more:

  www.guru-lang.org

Extra Slides
Initializing Subdata in match-cases

- **Init-function** defined as part of resource abstraction.
- Suppose matching on \( x : r \), subdatum \( y : r' \).
- **Init-function** for \( r - r' \) initializes \( y \).

\[
\text{Init } \text{ginit\_unowned\_unowned}(\#\text{unowned } x)(\#\text{unowned } y).\#\text{unowned}
\]

```c
inline void *ginit_unowned_unowned(int A, void *x, void *y) {
    ginc(y);
    return y;
}
```

**END.**

\[
\text{Init } \text{ginit\_owned\_unowned}(\#\text{owned } x)(\#\text{unowned } y).\#<\text{owned } x>
\]

```c
#define ginit_owned_unowned(A,x,y) y
```

**END.**

- **Compressing chains of ownership:**

\[
\begin{align*}
t : <r \ y> & \quad y : <r' \ z> \\
\hline
\end{align*}
\]

\(
\@ t : <r \ z>
\)