Resource Typing in Guru

Aaron Stump
CS Dept.
The University of Iowa, USA
astump@acm.org

Evan Austin
EECS Dept.
The University of Kansas, USA
ecaustin@ittc.ku.edu

Abstract

This paper presents a resource typing framework for the Guru verified-programming language, in which abstractions for various kinds of program resources can be defined. Implemented examples include reference-counted data, mutable arrays, and heap-allocated mutable aliased data. The approach enables efficient, type-safe programming with mutable and aliased data structures, with explicit deallocation (not garbage collection). We evaluate performance of the approach with two verified benchmarks, one involving mutable arrays, and another involving FIFO queues.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Mechanical verification; F.4.1 [*Mathematical Logic*]: Mechanical theorem proving

General Terms Languages, Verification

Keywords Dependently Typed Programming, Resource Types, Aliasing, Language-Based Verification

1. Introduction

Dependent types are of significant current interest for practical verified programming, as a substantial number of recent works attest (e.g., (Oury and Swierstra 2008; Norell 2007; Pasalic et al. 2007; Sheard 2006; Nanevski et al. 2008; Licata and Harper 2005; Chen and Xi 2005)). Dependent types hold out the promise of incrementally extending the verification power of traditional type systems. With traditional verification methods, there is a significant shift in thinking required for programmers to apply the method. In contrast, with dependent types, programmers can incrementally enrich types with which they are well familiar. For example, it is a relatively small conceptual leap to go from a type like $\langle \texttt{list}\ A\rangle$ (for homogeneous lists) to a type like $\langle \texttt{list}\ A\ n\rangle$ (for homogeneous lists of length n). Of course, the richer the properties expressed by the types, the more verification burden we might expect. But still, with dependent types, there seems a better change of incrementally exploring the continuum of correctness, starting with current programming practice, than with traditional verification methods.

But current dependent type systems are largely confined to functional programming languages. Despite the great intellectual depth of the literature on functional programming, and enthusiastic user communities, industrial adoption of functional programming is miniscule compared to languages like C/C++ or Java. There are, no doubt, many non-techical reasons for this, such as maturity of standard libraries (it is very difficult to compete, for example, with the massive standard library for Java). But there is at least one technical issue which has always held back functional programming languages, and others based on automatic memory management: efficiency. Again, despite tremendous invention in the field, garbage collection remains costly, as several recent studies report (Xian et al. 2008; Hertz and Berger 2005).

This paper shows how to combine the expressive power of dependent types with linear types to define imperative abstractions for different kinds of resources, without the need for garbage collection. This is carried out in the context of the Guru verified-programming language, a dependently typed pure functional programming language. Resources are created and consumed explicitly. The type system rules out memory errors such as accessing deallocated or uninitialized memory by ensuring that each resource is consumed exactly once. The definition of a resource is flexible, so that resource consumption need not correspond to memory deallocation. This flexibility is achieved through a **resource typing framework**, in which different resource management schemes can be defined by a set of primitive types, along with operations on those types. The definition of a term-level primitive consists of its type, an optional pure functional term which models the primitive, and (imperative) C code which the Guru compiler will use for the definition of the primitive. The connection between this C code and the pure functional model (if supplied) must be trusted. The resource typing framework – in particular, its linear types – helps ensure that the abstraction is used only in ways where the behaviors of the imperative implementation and functional model coincide. But since the number of lines of C code required to implement, say, mutable arrays is very small (on the order of tens of lines), we contend that this is a reasonable compromise between trustedness and the burden of verification.

We begin with an overview of resource typing in GURU, focusing on several example resource types. We then consider several example imperative abstractions built using those resource types. These include abstractions for mutable arrays, and an example of an abstraction using aliasing, namely FIFO queues. We give a system of resource-typing rules, and state its soundness with respect to a resource-manipulating semantics. Finally, we give some empirical results comparing our approach to memory management with that based on garbage collection in OCAML and HASKELL on some benchmark programs.

The reader may find the sources for all the examples in this paper in the Google Code repository for GURU, at `www.guru-lang.org`. Most are in GURU's standard library, `guru-lang/lib/`. A few (noted below) are in a special directory along with the source of this paper: `guru-lang/papers/resource-typing/tests/`.

PLPV'10, January 19, 2010, Madrid, Spain.
Copyright © 2010 ACM . . . $5.00

2. Overview of GURU's Resource Typing

In this section, we define a resource-typing framework, orthogonal to GURU's datatype typing, upon which we can build various imperative abstractions. We start with the familiar idea of modeling resources using **linear types**. A resource is considered to be an entity to which exactly one other entity refers, while it exists in the program. So a resource in this sense is intended to be like a physical resource, such as a bicycle: at most one person can make use of the bicycle at a time. This is the first principle of our resource-typing framework: we must have exactly one reference to each extant resource at any point during evaluation. To this we add a second principle, inspired by examples of physical resource, like the bicycle example. For after all, more than one person can use the bicycle, if one rides on the handle bars and another on the seat. Resources often consist of **sub-resources** (like the handle bars and seat of the bicycle), which may be used independently. But while they are in use, the resource of which they are parts is not available for use as a whole. This is our second principle: resources may be divided into sub-resources, but in that case, the whole resource cannot be used by another entity until use of the sub-resources is complete.

Our framework allows us to track the sub-resource relationship statically. It also enforces the linearity discipline. The framework allows different resource types to be declared, together with primitive operations on them. Primitives are declared with an optional functional model, and then C code providing its (imperative) implementation. When such a primitive is declared, its resource type is declared axiomatically: we do not check that the functional model satisfies that resource type, but just take it as a given that the C code's behavior matches that specified by the resource type. As stated above, the behavioral equivalence of the C code and the functional model is not proven, and must be trusted. It is our contention that this is a reasonable tradeoff between verifiability and trustworthiness.

A final note on **usability** of the framework: while we are currently programming directly in the language described below, we envision this serving as a core language for a surface language with more inference than we have here (see the Conclusion).

2.1 Quick introduction to GURU

In order to move quickly to concrete examples, we survey first the syntax and informal semantics of terms and types in GURU, including resource-type annotations. We omit the syntax for proofs P and formulas F, which are separate syntactic categories in GURU. The syntax is given in Figure 1, where we write $\underline{o}$ to mean an optional occurrence of o.

Terms and types are as reported in our previous work (Stump et al. 2009), except for the ownership annotations o, with their accompanying *consumption annotations* i. For terms t, we have general recursive functions introduced with `fun`, which lists inputs and their types first, then a ":", then the output type, then a ".", and then the body of the function. We support finite failure via `abort`. Operationally, `impossible` terms behave like `abort`, but come with a proof that that point in the code can never be reached (because the assumptions in scope at that point are contradictory). The `do`-`end` construct is just a prefix notation for sequencing of terms. It is not the monadic `do` familiar from HASKELL. We have explicit casts to change the type of a term t using an equation between types proved by a proof P. We also have `let`- and `match`-terms. Right after the `match` keyword, we may optionally have !, which means that the scrutinee will not be consumed by the match. The default is that the scrutinee will be consumed. These `let`- and `match`-terms bind additional assumption variables after the `by` keyword, which provide information in the rest of those terms about the computation that must have led there. For example, in `let`-terms, the assumption variable has classifier $\{x = t\}$, following the

$$
\begin{array}{lll}
t & ::= & x \mid\mid \\
 & & c \mid\mid \\
 & & \texttt{fun}\, x(\underline{i_1}\ \underline{o_1}\, x_1 \ :\ A_1) \cdots (\underline{i_n}\ \underline{o_n}\, x_n \ :\ A_n) \\
 & & \quad :\underline{o}\, T.\, t \mid\mid \\
 & & (t\ X) \mid\mid \\
 & & \texttt{cast}\ t\ \texttt{by}\ P \mid\mid \\
 & & \texttt{abort}\ T \mid\mid \\
 & & \texttt{impossible}\ P\ T \mid\mid \\
 & & \texttt{do}\ t_1\ \ldots\ t_n\ \texttt{end} \mid\mid \\
 & & \texttt{let}\ x\ \texttt{=}\ t\ \texttt{by}\ y\ \texttt{in}\ t' \mid\mid \\
 & & \texttt{match}\,\underline{\texttt{!}}\ t\ \texttt{by}\ x\ y\ \texttt{with} \\
 & & \quad c_1\ \bar{x}_1\ \texttt{=>}\ t_1 | \cdots | c_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end} \mid\mid \\
 & & \texttt{existse\_term}\ P\ t \mid\mid \\
 & & @\, t \\
 & & \\
X & ::= & t \mid\mid T \mid\mid P \\
 & & \\
A & ::= & T \mid\mid \texttt{type} \mid\mid F \\
 & & \\
T & ::= & x \mid\mid d \mid\mid \texttt{!} \mid\mid \texttt{Fun}(\underline{i}\ \underline{o}\, x \ :\ A).\underline{o}\, T \mid\mid \langle T\ Y \rangle \\
 & & \\
Y & ::= & t \mid\mid T \\
 & & \\
o & ::= & \#\texttt{untracked} \mid\mid \texttt{spec} \mid\mid \#\texttt{r} \mid\mid \#\langle r\ x \rangle \\
 & & \\
i & ::= & \hat{}\ \mid\mid ! \mid\mid 1
\end{array}
$$

Figure 1. Terms (t) and types (T); underlined occurrences are optional

`let`-definition of x to t. This is useful informationa in the body of the `let`, since it tells us that t has terminated, and that x is provably equal to its value. The `existse_term` is a rarely used construct for elimination of a proved existential formula, inside a term. The final term construct @ t is for compressing chains of ownership, discussed further below.

A few words are warranted on the Operational Type Theory (OpTT) upon which GURU is based. Provable equality between terms in GURU has the meaning that the two terms either both diverge using GURU's call-by-value operational semantics, or join at a common value. Provable equality between types has the meaning that the types are equal modulo provable equality of any terms they contain (as indices to indexed datatypes). OpTT preserves decidability of typing and logical consistency in the presence of general recursion. Definitional (automatic) equality is very weak, and does not include computation. Casts are used to change the type of a term, thus placing evaluation under the control of the programmer, instead of the type checker. The axiom $\texttt{join}_n\ t_1\ t_2$ states that t_1 and t_2 are joinable in at most n steps (in practice, n is omitted and enforced by a global timeout). To ensure logical consistency, proofs are syntactically separated from terms, and a straightforward termination analysis is used to ensure that induction proofs are well-founded. The separation of logical and computational parts in a type-theoretic language has been independently proposed in Luo's Logic-Enriched Type Theory (Luo 2008)).

2.2 Ownership annotations

Let us now consider the ownership annotations o and consumption annotations i (of Figure 1). Every term in GURU has a resource type, as well as a datatype. The ownership annotations o are given for function inputs and outputs, in order to state that resource type, as well as to express subresource relationships. The different kinds of resource types are:

- #untracked. This is for data that are not being treated as resources, such as scalar data (booleans, machine words, etc.). Note that we do not perform closure conversion in GURU, so that we may also treat functions as untracked data (closure conversion would require tracking the memory for the closure).
- spec. This is for specificational data, that are dropped during compilation and by GURU's definitional equality. Note that the current design does not allow declaring an additional resource type (other than spec) for specificational data.
- #r. This is for data of a declared resource type r. We envision library designers crafting a small number of resource types (as we have done so far), that can then be used by GURU programmers.
- #<r x>. This is for data of a declared resource type r, which are subresources of x, where x is a variable. If we have y : #<r x>, then we say that y is pinning x. Our static analysis, defined below, will require that all pinning resources of x are consumed before x itself is consumed.

If an ownership annotation is omitted, GURU chooses a default of either #untracked, if the datatype is flat (like bool), a function type, or declared untracked by the user; or else #unowned, discussed below.

Consumption annotations i refine somewhat the basic resource-tracking of linear types. Without any consumption annotation, the meaning is that the input must be consumed exactly once by the function. An input can be consumed by passing it to another function whose type asserts that it will consume the input (including resource-managing primitive functions which actually deallocate memory). This includes passing the input to a term constructor. It can also be consumed by returning it from the function. The additional annotations i mean:

- ˆ : consume (exactly once) but do not return. This annotation means that the input is to be consumed by the function, but not returned, neither directly, nor in another data structure.
- **!**: do not consume. This annotation means that the input will not be consumed at all by the function. This is primarily used when defining primitive operations for resource types.
- **1**: consume, where returning counts as consuming. This annotation is not implemented in GURU, since leaving off the consumption annotation already has this meaning. 1 is included just for uniformity below.

3. Example Resource Types

In this section, we consider examples of resource types and their associated primitive operations. We use these resource types to help build higher-level imperative abstractions, described later in the paper.

3.1 Reference-Counted Data

The most commonly used resource type in GURU is #unowned. This is the resource type for reference-counted data, which are not pinned by some other reference. Figure 2 gives a listing from the unowned.g library file (see guru-lang/lib on the 1.0 branch in the Google Code repository, via www.guru-lang.org), which declares this resource type. This listing has four commands: a ResourceType-command to declare the unowned resource type; an Init-command to declare how subdata are to be initialized when extracting them during pattern-matching; and two Define-commands, defining primitive functions for incrementing and decrementing the reference count. Let us look at these in turn:

- the ResourceType-command. This declares the resource type, together with a primitive function consume_unowned. This function will be used by the GURU for consuming elements of that resource type, in one special situation: match-terms. By default, when we match on a piece of data, that data will be consumed. The GURU compiler will insert a call to this consume-function after extracting the subdata.

  The functional model given for consume_unowned is a trivial GURU function. The C code for consume_unowned, which follows "<<END", uses functions dec, op, and releaes, provided by the GURU runtime, to check the reference count, which are stored in all but the low 8 bits of a single word per constructor application. If the reference count is 0 (so all those bits at and above bit 8 are off), then the data is released.
- the Init-command. This specifies how to initialize subdata y of x when we pattern-match on x. The GURU compiler will insert a call to the given C function, whenever the resource type of the scrutinee of a match is the same as the first resource type listed (for x) in the Init-command, and the resource type of the subdatum is the same as the second resource type listed (for y). Here, we initialize reference-counted subdata by incrementing their reference counts. Consuming a piece of data will consume its subdata, so if the reference count of the scrutinee falls to 0, we will decrement the reference counts of its subdata when it is consumed. That is why we increment the subdata's reference counts with this Init-function, so that we still have a reference to them even after the scrutinee is consumed.
- The Define-commands. The keyword primitive signals that these are primitives. Here, it is the resource types declared for the primitives that are of most interest. For inc, the resource type is

  ```
  Fun(spec A:type)(! #unowned y:A).#unowned A
  ```

  This says that inc takes a type A, and then an element y of that type. The resource type of y is #unowned, and the consumption annotation is !, indicating that this y will not be consumed by inc. Then inc will produce a new #unowned A as output. So the type is telling us (and the resource-tracking analysis) that when we increment a reference count of some object x, we do not consume the reference to x, but gain an additional new reference to it from inc. So where there was one reference to x before, there are now two references. So we are thinking of each reference to x as a separate resource to be tracked.

  The type for dec just says that it consumes the given reference. The consumption annotation ˆ says that it does not return that reference – but this is not important in this case.

The attentive reader might be wondering about why we use spec with the type A given to inc, but not with the type A given to dec. The reason is that at runtime, compiled GURU programs pass integers corresponding to type families (that is, one integer for all different instances of the type family for homogeneous lists). This is for the benefit of the runtime function release, which actually manages memory for data constructed with the different term constructors. So even though all type annotations are dropped during formal reasoning in GURU's logic, some are preserved during compilation. Here, it turns out that inc does not need the type at runtime, while dec (which ends up calling release) does. Note that the C code for the one case has A as an argument, while in the other it does not.

3.2 Owned Data

One criticism of reference counting as a memory management technique is that memory traffic generated by incrementing and decre-

```
ResourceType unowned with
  Define primitive consume_unowned : Fun(A:type)(^ #unowned r:A).void
  := fun(A:type)(r:A).voidi
<<END
  inline void gconsume_unowned(int A, void *r) {
    if (r == 0) return;
    dec(r);
    // fprintf(stdout,"gdec(%x) = %d\n", r, op(r) >> 8);
    if (op(r) < 256)
      release(A,r,1);
  }
END.

Init ginit_unowned_unowned(#unowned x)(#unowned y).#unowned
<<END
  inline void *ginit_unowned_unowned(int A,void *x,void *y) {
    ginc(y);
    return y;
  }
END.

Define primitive inc : Fun(spec A:type)(! #unowned y:A).#unowned A := fun(A:type)(y:A).y
<<END
  inline void *ginc(void *y) {
    inc(y);
    // fprintf(stdout,"ginc(%x) = %d\n", y, op(y) >> 8);
    return y;
  }
END.

Define primitive dec : Fun(A:type)(^#unowned y:A).void := fun(A:type)(y:A).voidi
<<END
  #define gdec(A,y) gconsume_unowned(A,y)
END.
```

Figure 2. The unowned resource type

menting reference counts is costly in terms of performance. We mitigate this cost by defining a second resource type, for references which may be created and consumed without increments and decrements of the reference count. The idea is that if such a reference r is (ultimately) pinning a reference-counted reference x (that is, an unowned reference), then it is safe to compute with r, without fear that the referenced data object's memory will be reclaimed. The reference-counted reference x cannot be consumed until r is, since r is pinning x. So we can use r safely without incrementing the reference count. This idea is the reason for the ^ consumption annotation. We need to know that the pinning reference r is really gone, before it is safe to consume x. If the pinning reference has been consumed by being returned by the function, possibly inside a data structure, then this sort of consumption is not sufficient to guarantee safety: the type system will have lost track of the connection between r and x. If, on the other hand, a function is known to consume r and not return it, then this means that the reference r is dead, and no longer pins x.

We list some representative primitives for this resource type in Figure 3 (for the rest, see `guru-lang/lib/owned.g`), and explain them as follows:

- `consume_owned`. As explained, we do not need to decrement the reference count for x when we consume it.
- `inspect`. This takes a reference-counted (`unowned`) reference x, and returns an `owned` reference pinning x. This is how we obtain an `owned` reference.
- `clone_owned`. This takes an `owned` reference y and creates a new `owned` reference pinning y. Chains of ownership, where x pins y and y pins z, can be built up in this way. Occasionally it is necessary to collapse such chains, in which case we use the @ t term-construct (Figure 1). If x pins y and y pins z, then @ x pins z directly.
- `inc_owned`. This takes an `owned` reference y and creates a new `unowned` reference to the same object. This requires, of course, that the reference count be incremented.
- `ginit_owned_unowned`. This initializes an `unowned` subdatum y of an `owned` scrutinee x. In this case, the initialized y pins x. This initialization has the effect of propagating the `owned` resource type during pattern-matching, and is another point at which chains of ownership can develop.

3.3 Unique Data

A simple resource type is that for an object to which exactly one reference is allowed while the object exists, without the possibility of multiple references as above (although we will relax that slightly below). The resource type for this is `unique`, defined in Figure 4. The interesting point to note here is that when pattern-matching on a `unique` object, we are insisting (with the `must_consume_scrutinee` directive) that the scrutinee is indeed consumed. We do not provide any way of initializing `unique` subdata from a scrutinee which is not itself `unique`. This ensures that `unique` must propagate from subdata to containing data. That is necessary, of course, to prevent duplicating a `unique` resource via a duplicated reference to an object containing the resource. If a term constructor's type does not declare that it creates `unique` data, while stating that it accepts `unique` arguments, then that constructor could be applied, but never pattern-matched against (due to the deliberate omission of the appropriate `Init`-function).

```
ResourceType owned with
  Define primitive consume_owned : Fun(A:type)(^#owned x:A).void
  := fun(A:type)(x:A).voidi <<END
  inline void gconsume_owned(int A, void *x) { }
END.

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.

Define primitive clone_owned : Fun(spec A:type)(! #owned y:A).#<owned y> A
  := fun(A:type)(y:A).y <<END
  #define gclone_owned(y) y
END.

Define primitive inc_owned : Fun(spec A:type)(!#owned y:A).#unowned A
  := fun(A:type)(y:A).y <<END
  inline void *ginc_owned(void *y) {
    inc(y);
    return y;
  }
END.

Init ginit_owned_unowned(#owned x)(#unowned y).#<owned x> <<END
  #define ginit_owned_unowned(A,x,y) y
END.
```

Figure 3. The owned resource type (selected primitives)

```
ResourceType unique with
  Define primitive consume_unique : Fun(A:type)(^#unique x:A).void
    := fun(A:type)(x:A).voidi <<END
  inline void gconsume_unique(int A, void *x) {
    release(A,x,0);
  }
END.

Init must_consume_scrutinee ginit_unique_unique(#unique x)(#unique y).#unique <<END
  #define ginit_unique_unique(A,x,y) y
END.

Init must_consume_scrutinee ginit_unique_owned(#unique x)(#owned y).#owned <<END
  #define ginit_unique_owned(A,x,y) y
END.

Init must_consume_scrutinee ginit_unique_unowned(#unique x)(#unowned y).#unowned <<END
  #define ginit_unique_unowned( A, x, y) y
END.
```

Figure 4. The unique resource type

3.4 Unique-owned resource type

We introduce a final resource type unique_owned, which we use to implement readers/writers locking. We can obtain multiple unique_owned references r from a unique reference x, but those references pin x. Our higher-level imperative abstractions insist that operations which update the object o to which x refers consume x, producing as output a new reference to the (imperatively) updated o. Since they consume the unique x, our resource-tracking discipline forbids there from being any pinning references to x extant. This ensures (statically) that all read-only references to o are consumed, before the read-write reference x can be used. The Init-function for initializing an unowned subdatum y of a unique_owned scrutinee x initializes that subdatum to be of resource type `<owned x>`, thus propagating the "owned" quality. The listing is omitted for space reasons, but see lib/unique_owned.g.

4. Array Abstractions

In this section, we consider two different abstractions of mutable arrays, built using the resource types described in the previous section. These array abstractions include a small number of primitive operations, with a very small number of additional lines of C code to trust (20 for warray, 27 for qcharray).

4.1 Mutable Arrays of Reference-Counted Data

We can easily define an abstraction for mutable word-indexed arrays storing reference-counted data, where we statically ensure that array-bounds are respected. The corresponding library file is warray.g. First, we define `<warray A n>`, where A is a type and n is a word. The intention is that n is the length, as a word, of the array. We define that warray-type as a primitive abbreviation for `<vec A (to_nat wordlen n)>`. Primitive abbreviations like this one are not included in GURU's definitional equality when type

```
Define primitive warray_get
   : Fun(spec A:type)(spec n:word)
        (! #unique_owned l:<warray A n>)
        (i:word)
        (u:{(lt (to_nat i) (to_nat n)) = tt}).
        #<owned l> A :=
  fun(A:type)(spec n:word)
     (l:<warray A n>)(i:word)
     (u:{(lt (to_nat i) (to_nat n)) = tt}).
   (vec_get A (to_nat wordlen n) l (to_nat wordlen i) u)
<<END
inline void* gwarray_get(void **l, int i) { return l[i]; }
END.
```

Figure 5. Definition of warray_get primitive

checking code that may be compiled. This is to ensure that compiled code only accesses warrays using warray's primitive operations, not operations (like pattern-matching) on the vec type. In code specially marked as specificational, or in proofs, however, the abbreviation is included in definitional equality, since for reasoning purposes, we will treat warrays as vectors.

It should also be explained that in that definition, word is the type for 32-bit machine words, which are functionally modeled as bitvectors of length 32, and imperatively implemented as actual machine words. The to_nat function converts a bitvector to a unary natural number, for more convenient reasoning. Appealing to such functions inside of types does not cause those functions to be executed at run-time, or even compile-time. Since compile-time evaluation is entirely under the control of the programmer, he may find ways to prove equalities about applications of (expensive) functions like to_nat without actually performing long executions of them.

Then we define four primitive operations, to allocate, deallocate, read to, and write from a warray. Figure 5 lists the code for just one representative operation, namely the read operation. The functional model of this operation is just to use vec_get to look up the value stored at location (to_nat wordlen i). The running time for this, of course, is exponential in the size (as a word) of i. The imperative implementation just uses the standard C constant-time array access operation. Note that the input u to the primitive is required to be a proof that the index i is less than the length n of the array, when those words are converted to nats. By requiring this proof, we statically ensure array-bounds are respected whenever warray_get is used. The warray_set primitive has a similar requirement.

As far as resource typing goes, we see that warray_get requires a unique_owned warray l. So this is a read-only reference to the warray. The function returns an element of datatype A, with resource type <owned l>. We use a pinning owned resource type here instead of unowned, just to avoid incrementing the reference count for the returned data. The calling context can always do that, using inc_owned. Of course, the array l is pinned by this <owned l> reference, until the latter is consumed. So we will not be able to update the array until the pinning reference has been consumed (possibly just by exchanging it for an unowned reference).

4.2 Mutable Arrays of Unique Data

We now consider mutable arrays storing unique data. Currently, we have implemented this imperative abstraction for character-indexed arrays, qcharray. Implementing it for word-indexed arrays poses no significant challenges, but remains to future work. The basic idea here is to track statically the set of indices at which the array is currently unavailable for reading. Such indices correspond either to uninitialized locations in the array, or to locations where the data has been "checked out" for writing. The set of such indices is tracked as a string (which in GURU is a list of characters), included as an additional parameter to the qcharray type. Checking out the element at index c from a qcharray (i.e., reading from the array) requires a proof that c is not in the list of indices which are already checked out. Similarly, checking in an element at index c from a qcharray (i.e., writing to the array) requires that the list of checked-out elements contains c. We list the types of selected primitives for qcharray in Figure 4.2. To discuss just a couple of those: the type for qcharray_empty says that it returns a qcharray where all the indices are checked out (since they are all uninitialized); and qcharray_out returns a pair, as an element of the qcharray_mod_t type listed in the figure, of the element that has been checked out, and the qcharray. The type of the qcharray has been modified so that it now includes the index c for the checked-out element.

```
qcharray_empty
: Fun(A:type).#unique <qcharray A all_chars>

Inductive qcharray_mod_t
   : Fun(A:type)(c:char)(s:string).type :=
 mk_qcharray_mod :
   Fun(A:type)(#unique a:A)
      (spec c:char)(spec s:string)
      (#unique l:<qcharray A (stringc c s)>).
    #unique <qcharray_mod_t A c s>.

qcharray_out
: Fun(A:type)(#untracked c:char)(spec s : string)
      (#unique l:<qcharray A s>)
      (u : { (string_mem c s) = ff}).
   #unique <qcharray_mod_t A c s>

qcharray_in
: Fun(spec A:type)(#untracked c:char)
      (#unique a:A)(spec s1 s2:string)
      (#unique l:<qcharray A (string_app s1 (stringc c s2))>).
   #unique <qcharray A (string_app s1 s2)>
```

Figure 6. Types for selected qcharray primitives

5. Abstractions Using Aliasing

In this section, we consider how to implement imperative abstractions with aliased heap-allocated data. We first discuss the rheaplet abstraction, which provides a functional model of a portion of the heap, and then see how to use it to implement FIFO queues. We give first a simple unverified implementation of FIFO queues, and then a verified version. In the course of implementing the verified version, we uncovered a bug in the unverified version, which was not revealed by the benchmark testcase (used for the empirical evaluation below).

5.1 Modeling the aliased heap with rheaplets

To implement an aliased data structure, it is necessary to find a way to model of the non-local communication which arises when a structure is updated via one reference, and then read via another reference. We rely here on a functional model of a portion of the heap, which we call a rheaplet. The "-let" is to indicate that this is just a portion of the heap, not its entirety; and the "r-" is because we use run-time reference counting of the number of aliases to each cell, to manage memory. This approach statically guarantees that uninitialized or deleted memory will never be read or written. It should be acknowledged from the beginning, however, that this approach can leak memory if cycles are formed in the reference graph. We have several things to say about that issue, in Section 8 below.

```
rheaplet_get
: Fun(spec A:type)(spec I:rheaplet_id)(^#owned p:<alias I>)
     (!#unique_owned h:<rheaplet A I>).
   #<owned h> A

rheaplet_set
: Fun(A:type)(spec I:rheaplet_id)(^#owned p:<alias I>)
     (#unique h:<rheaplet A I>)(a:A).
   #unique <rheaplet A I>
```

Figure 7. The types of the primitives for reading and writing an `rheaplet` via an alias

The `rheaplet` abstraction is based on a type family `<rheaplet A I>`, for heaplets storing aliased data of type A. The I is a `rheaplet_id`, which we use to prevent (statically) aliases associated with one heaplet from being used to access another. The type of aliases is then `<alias I>`, for aliases into the heaplet with heaplet id I. In the functional model, a heaplet is a list of elements of type A, and an alias is a unary natural number giving a position into that list. In the imperative implementation, `rheaplets` and `rheaplet_ids` are just dummy scalar values (retained during compilation because we currently do not support specificational data of resource type other than the default, and so these cannot be declared specificational). The imperative implementation of aliases is explained next.

There are three `rheaplet` primitives. The first ("`rheaplet_in`") is for adding some data to the `rheaplet`. In the functional model, the data is added by appending it to the list which functionally models the heaplet. In the imperative implementation, adding data to a heaplet allocates a 3-word cell in memory, with one word each for a reference count for the number of outstanding aliases, the type (represented as an integer, as mentioned above) of the data, and the pointer to the data. The imperative implementation of aliases is just as references to this 3-word cell. This means that aliases can be managed using the usual primitives for reference-counted data (Section 3.1 above). The level of indirection added by the 3-word cell means that we can change the object to which the aliases refer, by changing the pointer (the last word) in the cell. This indirection imposes an additional cost in memory and running time, and so it is particularly important to evaluate it empirically, as we do below. The `rheaplet_in` function returns the updated heaplet, and the new alias.

Finally, the types for the primitives for reading and writing via an alias are given in Figure 7. They are similar in spirit to the types for reading and writing mutable arrays above. The main point to note is the use of the `rheaplet_id` I to connect an alias with its `rheaplet`. Even though we may deallocate a cell in the imperative implementation (if the number of outstanding aliases to that cell falls to zero), in the functional model we never remove an element from the list which model the heaplet. So aliases are always valid indices into the list of their associated heaplet.

5.2 Unverified FIFO Queues

Using the `rheaplet` primitives, we have implemented a statically memory-safe FIFO queue abstraction. The source code for this may be found as `queue.g` in the tests subdirectory for this paper (see the Introduction). A queue consists of a singly-linked list from the `qout`-end, where data is dequeued, to the `qin` end, where it is enqueued. Each cell of the queue is allocated in a heaplet associated with the queue. This is because the last such cell has two aliases (all the others have one), which ends up forcing all cells to be heaplet-allocated. Figure 8 lists the datatype definition for queues (omitting a simple definition for queue cells), and then the types for the operations on queues. Note that these operations are not primitives. They are built using the above abstractions, and require no additional untrusted C code. Note also that since this data structure does not form cycles in the heap, our reference-counting approach for `rheaplets` is sufficient to reclaim all unreachable memory (though this is not statically verified).

```
Define type_family_abbrev
  rheaplet_queue :=
fun(A:type)(I:rheaplet_id).
 <rheaplet <queue_cell A I> I>.

Inductive queue : Fun(A:type).type :=
 queue_datac
   : Fun(A:type)(spec I:rheaplet_id)
       (#unique h:<rheaplet_queue A I>)
       (qin qout : <alias I>).
     #unique <queue A>
| queue_datan
   : Fun(A:type)(spec I:rheaplet_id)
        (#unique h:<rheaplet_queue A I>).
      #unique <queue A>.

queue_is_empty
 : Fun(spec A:type)
      (^#unique_owned q:<queue A>).bool

queue_front
 : Fun(spec A:type)
      (^#unique_owned q:<queue A>)
      (u:{ (queue_is_empty q) = ff }). A

enqueue
 : Fun(spec A:type)
      (#unique q:<queue A>)(a:A).
    #unique <queue A>

dequeue
 : Fun(spec A:type)
      (#unique q:<queue A>)
      (u:{ (queue_is_empty q) = ff }).
    #unique <queue A>
```

Figure 8. Types for the unverified queue abstraction

We just consider a few points about the typings in the figure. Both empty (`queue_datan`) and non-empty (`queue_datac`) queues have a unique reference to a `rheaplet` of queue cells. Additionally, non-empty queues have references to `qout` and `qin`. Allocating a new queue with `queue_new` (not shown) creates a new heaplet. The `queue_front` and `dequeue` functions require a proof that the queue is non-empty. These proofs allow us to show (via `impossible`-terms, mentioned in Section 2.1 above) that several situations cannot arise. There is only one place where the queue code uses `abort`, in response to violation of an unverified invariant that the `qin`-cell has no next pointer. We will eliminate this `abort` in the next section. The total number of lines of code, with comments, is 138. Of these, only 6 are proofs, used in the `impossible`-terms just mentioned.

5.3 Verified FIFO Queues

In order to verify that the `qin`-cell of a queue has no next pointer, we must reason explicitly about the heap. This was not required for the implementation of the queue data structure as above. In particular, it is necessary to reason explicitly about the equality and disequality of aliases. This imposes a significant burden of proof, which future work must lighten in order for this method to be practical; more is said about this in Section 8 below. Nevertheless, the current approach provides an adequate foundation for more automated methods of verifying such properties.

Figure 5.3 shows the invariants needed to verify that the qin-cell of the queue has no next pointer, expressed as additional proof arguments required by the constructors for the verified queue type. The full source code for the verified queue may be found as queue2.g in the tests directory for this paper (see the Introduction). The new invariants for non-empty queues are the following:

- **inv_qin.** This expresses that qin is a valid index into the heaplet h; inv_qout is similar.
- **inv.** This expresses that all the next pointers in the queue cells stored in the heaplet h are valid indices into h. Note that the @<...> construct is GURU notation for application of a defined predicate symbol (whose definition here we omit).
- **inv_qin2.** This expresses our desired invariant, that the qin cell has no next pointer.

Some discussion of these invariants is warranted. First, the reader might well wonder why we are introducing these explicit statements that aliases being valid indices into the heaplet. After all, we maintain the relationship between aliases and heaplets statically, using the shared rheaplet_id. While we have designed our static typing to ensure that heaplets are accessed only by their own aliases, this does not imply a principle like:

```
Forall(A:type)(I:rheaplet_id)
      (a:<alias I>)(r:<rheaplet A I>).
  { (lt a (length r)) = tt }
```

Indeed, this principle is unfortunately false: aliases created later need not be valid indices into earlier versions of heaplet with heaplet id I. The problem is due at least in part to the fact that our logic is intuitionistic (GURU's design would also allow classical logic), while the state involved must be reasoned about linearly. How to recover something like this principle is not yet clear, though see the discussion below (Section 8).

The verified versions of the queue functions of Figure 8 above have the same types as given there. The difference is that now whenever the constructors for the queue type are used to (re)construct a queue, we must supply proofs of our invariant properties. These proofs are built manually, with modest aid from GURU's lone but reasonably powerful tactic, called hypjoin. This tactic automatically proves terms equal (under some reasonable conditions about termination) iff they are joinable in the operational semantics modulo ground equations specified (by giving proofs for them) by the user (Petcher and Stump 2009; Petcher 2008). Despite this tactic, the proofs are mostly manual, swelling the original 138 lines for the unverified queue to 310, not counting general list lemmas from the standard library and general heaplet lemmas, such as, for example, this one expressing that the operation of adding an element to the heaplet is successful (reading the updated heaplet h' using the new alias p returns the added element a):

```
rheaplet_in_get
: Forall(A:type)(I:rheaplet_id)(h h':<rheaplet A I>)
     (p:<alias I>)(a:A)
     (u:{(rheaplet_in h a) = (return_rheaplet_in h' p)}).
   { (rheaplet_get p h') = a }
```

As noted above, while proving these invariants for the verified queue we uncovered a bug in the unverified queue. The bug is rather insidious, because it is not uncovered by our benchmark testcase. This benchmark enqueues space-delimited strings read from standard input, and then dequeues them all, printing the last string dequeued. The bug was that the unverified code was inadvertently swapping qin and qout in the dequeue operation. On our benchmark, the effect was to dequeue the first element (at qout), and then the last element (at the new qout, which is the previous qin).

```
Inductive queue : Fun(A:type).type :=
 queue_datac
 : Fun(A:type)(spec I:rheaplet_id)
      (#unique h:<rheaplet_queue A I>)
      (qin qout : <alias I>)
      (inv_qin : {(lt qin (length h)) = tt})
      (inv_qout : {(lt qout (length h)) = tt})
      (inv : @<rheaplet_queue_inv A I h>)
      (inv_qin2 : {(queue_cell_has_next
                      (rheaplet_get qin h)) = ff}).
    #unique <queue A>
| queue_datan
  : Fun(A:type)(spec I:rheaplet_id)
       (#unique h:<rheaplet_queue A I>)
       (inv:@<rheaplet_queue_inv A I h>).
     #unique <queue A>.
```

Figure 9. The main datatype, with invariants, for the verified queue abstraction

Since there is no next pointer from that erroneous qout, it appears to dequeue as though it has removed the last element. All memory is correctly freed, thanks to our linear typing (and confirmed by valgrind). The error was detected during proof of this intensionally quite different property, because the assertion that qin has no next-pointer cannot be proved when we swap qin and qout.

6. Type System

We now give a formal definition of the resource-tracking static analysis implemented in GURU as an abstract operational semantics. We formulate soundness with respect to an operational semantics operating on certain (finite) reference graphs. The reference graphs are in the form of what we call *shallowly augmented forests*: they are forests, except that nodes may have multiple predecessors in the graph, but then at most one of those predecessors can itself have a predecessor. This is sufficient for our functional-modeling approach, since it captures the idea that data are all tree-structured, except that there may be some additional references to the nodes (corresponding to predecessors that have no predecessor).

6.1 Assumptions on the terms

We assume that terms have already passed the datatype checker, and that calls to initialization functions have been inserted as follows. (Our implementation inserts these calls during resource-type checking, where we just check simple typing with respect to the resource types like unique.) Suppose we have an Init-command beginning like this:

```
Init ginit_r_rr(#r scrut)(#rr subdat).#rrr
```

Suppose further we have a match-case with a scrutinee of resource type r, and a pattern $c\ \bar{x}$, where $y \equiv x_i$ is of resource type rr. Also, assume the scrutinee is syntactically a variable x (we perform a simple compiler transformation to ensure this, before resource tracking begins). Then the body of the case will begin with nested let-bindings, including one of the form

```
let y = ginit_r_rr(x,y) in
```

We further assign the ginit_r_rr function this resource type, which omits the datatype annotations (like <list A> or similar) and shows only the resource-type (like unowned or unique) and consumption annotations:

```
Fun(! rr scrut)(^ r subdat).rrr
```

$$\begin{array}{rcl} U(\Delta, x, \bar{x}, 1, \bar{i}) & = & U(\Delta - x, \bar{x}, \bar{i}), \text{if } x \in \mathit{dom}(\Delta) \\ U(\Delta, x, \bar{x}, !, \bar{i}) & = & U(\Delta, \bar{x}, \bar{i}) \\ U(\Delta, x, \bar{x}, \hat{}\,, \bar{i}) & = & U(\Delta - x, \bar{x}, \bar{i}), \text{if } x \in \mathit{dom}(\Delta) \\ U(\Delta, \cdot, \cdot) & = & \Delta \end{array}$$

Figure 10. Meta-theoretic function for consuming Δ resources

Also, we assume that uses of `match` without the optional ! indication (see Figure 1) have been translated to uses which do include this annotation, and where the body of each case contains, right after the prefix of `let`-bindings just discussed, a call to the appropriate `consume`-function for `r` (the scrutinee's resource type), on the scrutinee. This desugars `match`, which does consume the scrutinee, right after initialization of subdata, into `match!`, which does not.

We assume the (singly recursive) terms have already been defunctionalized into global first-order recursive equations of the forms $(f\ \bar{x}) = t$. Term constructors we denote with c, and primitives with p. Finally, we assume that proofs (including assumption variables) as well as data marked specificational have already been dropped, after datatype checking. Also, we do not treat `existse_term`- and `do`-terms (both easily handled), for space reasons.

6.2 Graph-based operational semantics

We define a graph-based evaluation relation $\Delta; t \Downarrow_k r; \Delta'$, where t is a term; r (here and below) is a variable; Δ, Δ' are functions from variables x to values of the form $(c\ x_1\ \ldots\ c_n)$, with c an arity-n term constructor; and k a natural number. If $\Delta(x) = (c\,\bar{x})$, this means variable x is instantiated with that term, where the variables $\bar{x}$ may have their own instantiations in Δ (similarly for Δ'). Furthermore, k is a natural number used as a timer; it is a technical device to allow us to distinguish diverging and finitely failing computations (for which we derive a $\perp$ judgment in the limit) from ones which fail due to resource errors (for which there is no derivation possible at all using the rules).

Figure 11 gives the rules. The first is for expiration of the timer. In this case, the form of the judgment has $\perp$ on the right hand side of the $\Downarrow$, indicating timeout with the given timer. We take these $\perp$-judgments to propagate strictly for all rules (that is, if a premise is provable with such a judgment, then the rule also concludes with $\perp$) – but we leave this machinery unformalized. The second rule is for evaluation of variables, the third for evaluation of constructor applications (we consider uses of 0-ary constructors as degenerate applications), and the fourth for evaluation of function applications. The next two rules, for applications of primitives p, consume resources by modifying the map Δ according to p's type, and return values using a functional model unconstrained by that type. In particular, this is where we might add an additional shallow reference into a tree.

Consuming resources when a primitive is applied is done using a meta-theoretic function application $U(\Delta, \bar{x}, \bar{i})$, defined in Figure 10. We write $p :: \bar{i}$ in the rule to indicate that the consumption annotations $\bar{i}$ are the ones stated in p's type for its arguments. We also inductively define a function "$-$" on Δ and an $x \in \mathit{dom}(\Delta)$ for removing subgraphs from the graph defined Δ:

$$\frac{S_0(x) = (c\,\bar{x}) \quad \forall i.(S_0 - x_i = S_i)}{S_0 - x = (\bigcap_i S_i) \setminus \{(x, (c\,\bar{x}))\}} \qquad \frac{x \notin \mathit{dom}(S)}{S - x = S}$$

Note that if an inference contains an undefined application of a meta-theoretic function, we interpret this to mean the inference is not allowed.

$$\frac{}{\Delta; t \Downarrow_0 \perp}$$

$$\frac{x \in \mathit{dom}(\Delta)}{\Delta; x \Downarrow_{k+1} x; \Delta}$$

$$\frac{\begin{array}{l}\forall i.(\Delta_i; t_i \Downarrow_k r_i; \Delta_{i+1}) \\ r \notin \mathit{dom}(\Delta_{n+1})\end{array}}{\Delta_1; (c\,t_1 \ldots t_n) \Downarrow_{k+1} r; \{(r, (c\,\bar{r}))\} \cup \Delta_{n+1}}$$

$$\frac{\begin{array}{l}\forall i.(\Delta_i; t_i \Downarrow_k r_i; \Delta_{i+1}) \\ (f\,\bar{x}) = t \\ \Delta_{n+1}; [\bar{r}/\bar{x}]t \Downarrow_k r; \Delta'\end{array}}{\Delta_1; (f\,t_1 \ldots t_n) \Downarrow_{k+1} r; \Delta'}$$

$$\frac{\begin{array}{l}\forall i.(\Delta_i; t_i \Downarrow_k r_i; \Delta_{i+1}) \\ (p\,\bar{x}) = t \\ p :: \bar{i} \\ U(\Delta_{n+1}, \bar{x}, \bar{i}); [\bar{r}/\bar{x}]t \Downarrow_k r; \Delta'\end{array}}{\Delta_1; (p\,t_1 \ldots t_n) \Downarrow_{k+1} r; \Delta'}$$

$$\frac{\begin{array}{l}\forall i.(\Delta_i; t_i \Downarrow_k r_i; \Delta_{i+1}) \\ \text{no functional model for } p \\ p :: \bar{i} \\ r \notin \mathit{dom}(\Delta)\end{array}}{\Delta_1; (p\,t_1 \ldots t_n) \Downarrow_{k+1} r; \{(r, \cdot)\} \cup U(\Delta_{n+1}, \bar{x}, \bar{i})}$$

$$\frac{\begin{array}{c}\Delta; t \Downarrow_k (c_j\ \bar{r}); \Delta'' \\ \Delta''; [\bar{r}/\bar{x}]t_j \Downarrow_k r; \Delta'\end{array}}{\Delta; \texttt{match}\ t\ \texttt{with}\ j.(c_j\ \bar{x}_j \texttt{=>} t_j) \Downarrow_{k+1} r; \Delta'}$$

$$\frac{\begin{array}{c}\Delta; t \Downarrow_k (d\ \bar{r}); \Delta'' \\ d \notin \{c_1, \ldots, c_n\}\end{array}}{\Delta; \texttt{match}\ t\ \texttt{with}\ j.(c_j\ \bar{x}_j \texttt{=>} t_j) \Downarrow_{k+1} \perp}$$

$$\frac{\begin{array}{c}\Delta; t \Downarrow_k r; \Delta' \\ \Delta'; [r/x]t' \Downarrow_k r''; \Delta''\end{array}}{\Delta; \texttt{let}\ x\ =\ t\ \texttt{in}\ t' \Downarrow_{k+1} r''; \Delta''}$$

$$\frac{}{\Delta; \texttt{abort} \Downarrow_k \perp}$$

$$\frac{\Delta; t \Downarrow_k r; \Delta'}{\Delta; @\,t \Downarrow_{k+1} r; \Delta'}$$

Figure 11. Graph-based operational semantics

For simplicity, we assume that if a primitive p lacks a functional model, then its return type is an instance of a primitively defined type. In that case, the second rule for applications of primitives just returns a new variable, bound to an uninterpreted entity "$\cdot$" in the heap. Earlier stages of type-checking rule out pattern-matching on elements of primitively defined types, but it is more convenient not to rely on that here (so we include the second `match`-rule). Note that in the `match`-rules we use a more compact meta-theoretic notation for `match`-terms. Finally, if for all k, $\Delta; t \Downarrow_k \perp$, then we write $\Delta; t \Downarrow \perp$, and say that t diverges. This relation is not recursively enumerable, but that is not problematic since it is used only in the formulation of soundness of our resource-tracking analysis.

6.3 Resource-tracking analysis

Next, we define an abstract, modular evaluation relation $\Theta; t \Downarrow x; \Theta'$, where Θ, Θ' are functions from variables x to a consumption annotation i (see Figure 1) or an expression $\langle y \rangle$, where y is a variable pinned by x. The rules are given in Figure 12, using a meta-theoretic function V similar to U above, and defined in Figure 13. The rules of Figure 12 are syntax-directed and all decrease the size of the term from conclusion to premises. Hence, in a standard way, they determine a terminating algorithm. The first two rules are for variables and constructor applications, while in the third, g is for a regular function symbol or a primitive function symbol: the two cases are handled the same way here, based on their resource type. We treat `abort`, similarly here as above, with a judgment form ending in $\bot$, which is propagated via straightforward omitted rules.

The rules are modular, in the sense that they rely on types for all the regularly defined functions they call. These types (given in the original source code) are confirmed using the following rule:

$$\frac{\begin{array}{l}(f\ \bar{x}) = t \\ \text{Assuming } f :: \bar{i}\text{, we have: } \{(x_1, i_1), \ldots, (x_n, i_n)\}; t \Downarrow r; \{(r, 1)\}\end{array}}{f :: \bar{i}}$$

The definition of the V function is the heart of the resource-tracking analysis. It insists that pinning references cannot be consumed by being returned; that is, the corresponding consumption annotation must be ˆ . It also insists that references cannot be consumed until all their pinning references are.

Some simplifying design choices have been made in this algorithm. For example, the rule for constructor applications disallows arguments marked with ˆ (meaning to be consumed but not returned). Also, branches in `match`-cases must have the same effect on Θ, and may not return pinning references. These restrictions could perhaps be relaxed. Also, we have not formalized consuming resource named in (global) definitions, although our implementation supports this.

Let us define $\Delta^*(t)$ for shallowly augmented forests Δ to be the term which is just like t except that we replace each $x \in \mathit{dom}(\Delta)$ with $\Delta^*(\Delta(x))$. The fact that Δ is a shallowly augmented forest ensures well-foundedness of this definition. If for every $x \in \mathit{dom}(\Delta)$ we have that $\Delta^*(x)$ is ground (no free variables), then we will call Δ ground. Also, let us write $\sigma :: \Theta \to \Delta$ to indicate that:

1. σ is an injective function from $\mathit{dom}(\Theta)$ to $\mathit{dom}(\Delta)$, and
2. $\forall (x, \langle y \rangle) \in \Theta$:
   (a) $y \in \mathit{dom}(\Theta)$, and
   (b) $\Delta^*(\sigma(x))$ is a subterm of $\Delta^*(\sigma(y))$ (with both defined).

THEOREM 1 (Type Soundness). *If $\Theta; t \Downarrow r; \Theta'$, then for all ground shallowly augmented forests Δ and $\sigma :: \Theta \to \Delta$, the following are both true.*

1. *Either $\Delta; \Delta^*(t) \Downarrow \bot$, or else $\Delta; \Delta^*(t) \Downarrow_k r'; \Delta'$ for some k, r', σ', and Δ' with:*
   (a) $\sigma' :: \Theta' \to \Delta'$, and
   (b) $\sigma'(r) = r'$,
   (c) $\sigma'|_{\mathit{dom}(\sigma)} \subset \sigma$.
2. *If $(x, \hat{}\ ) \in \Theta$, then for all $y \in \mathit{dom}(\Delta)$, $\Delta'^*(\sigma'(x))$ is a subterm of $\Delta'^*(y)$ only if it (already) is of $\Delta^*(z)$, for some $z \in \mathit{dom}(\Delta)$ where $\Delta^*(y)$ is a subterm of $\Delta^*(z)$.*
3. *If $(x, !) \in \Theta$, then $\sigma'(x)$ is defined.*

The first condition states that resources are consumed from Δ in a way consistent with Θ (as mediated by σ). The second states that evaluation using Δ cannot cause a reference marked with ˆ ("consume but do not return") to be embedded in a value in Δ', unless it was already so embedded in Δ. The third states that references marked not to be consumed are in fact still present in the final state. Writing out in detail the detailed proof of this theorem, which should proceed by induction on the structure of the assumed typing derivation, with sub-inductions for relating results of U and V, remains to future work. We may then easily apply this theorem to conclude that our modular analysis of recursive functions is sound.

$$\frac{x \in \mathit{dom}(\Theta)}{\Theta; x \Downarrow x; \Theta}$$

$$\frac{\begin{array}{l}\forall j.(\Theta_j; t_j \Downarrow r_j; \Theta_{j+1}) \\ c :: \bar{i} \\ \hat{}\ \notin \{\bar{i}\} \\ r \notin \mathit{dom}(\Theta)\end{array}}{\Theta_1; (c\, t_1 \ldots t_n) \Downarrow r; \{(r, 1)\} \cup V(\Theta_{n+1}, \bar{r}, \bar{i})}$$

$$\frac{\begin{array}{l}\forall j.(\Theta_j; t_j \Downarrow r_j; \Theta_{j+1}) \\ g :: \bar{i} \\ r \notin \mathit{dom}(\Theta)\end{array}}{\Theta_1; (g\ t_1 \ldots t_n) \Downarrow r; \{(r, 1)\} \cup V(\Theta_{n+1}, \bar{r}, \bar{i})}$$

$$\frac{\begin{array}{l}\forall j.(\Theta, \bar{x}_j; t_j \Downarrow r_j; \{(r_j, i)\} \cup \Theta') \\ r \notin \mathit{dom}(\Theta')\end{array}}{\Theta; \mathtt{match}\ x\ \mathtt{with}\ j.(c_j\ \bar{x}_j\mathtt{=>}t_j) \Downarrow r; \{(r, i)\} \cup \Theta'}$$

$$\frac{\begin{array}{l}\Theta; t \Downarrow r; \Theta' \\ \Theta'; [r/x]t' \Downarrow r''; \Theta''\end{array}}{\Theta; \mathtt{let}\ x\ =\ t\ \mathtt{in}\ t' \Downarrow r''; \Theta''}$$

$$\frac{}{\Theta; \mathtt{abort} \Downarrow \bot}$$

$$\frac{\begin{array}{l}\Theta; t \Downarrow r; \{(r, \langle y \rangle)\} \cup \Theta' \\ (y, \langle z \rangle) \in \Theta'\end{array}}{\Theta; @\, t \Downarrow r; \{(r, \langle z \rangle)\} \cup \Theta'}$$

Figure 12. Resource-tracking abstract operational semantics

$$\begin{array}{rcll}
V(\Theta, x, \bar{x}, 1, \bar{i}) & = & V(\Theta - x, \bar{x}, \bar{i}),\text{ if} & \begin{array}{l}1.\ x \in \mathit{dom}(\Theta) \\ 2.\ \forall y.\Theta(x) \neq \langle y \rangle \\ 3.\ \forall (y, \langle z \rangle) \in \Theta,\ z \neq x\end{array} \\
V(\Theta, x, \bar{x}, !, \bar{i}) & = & V(\Theta, \bar{x}, \bar{i}) & \\
V(\Theta, x, \bar{x}, \hat{}\ , \bar{i}) & = & V(\Theta - x, \bar{x}, \bar{i}),\text{ if } x \in \mathit{dom}(\Theta) & \\
V(\Theta, \cdot, \cdot) & = & \Theta &
\end{array}$$

Figure 13. Meta-theoretic function for consuming Θ resources

7. Empirical Results

The goal of the following tests is to compare the performance of mutable arrays and FIFO queues implemented in GURU with analogous data structures implemented in two of the leading functional programming languages currently in use, HASKELL and OCAML. Comparisons are also made to HASKELL and OCAML versions with garbage collection minimized by picking large enough values for heap sizes so that collection is not necessary (see the Makefile in the `tests` directory mentioned in the Introduction). The specifications of the test system are described in Figure 14, including compilers and optimization options.

```
Processor:   2.67 GHz Intel Xeon W3520
Memory:      8 GB
OS:          Linux version 2.6.18-164.2.1.el5
Compilers:   GHC 6.10.4 -O
             OCamlOpt 3.11.1
             Gnu gcc 4.3.3 -O4
```

Figure 14. Specifications of the test system.

```
Define warray_binary_search
  : Fun(A:type)
       (spec n:word)
       (^ #unique_owned l:<warray A n>)
       (first last:word)
       (value:A)
       (c:Fun(^ #owned a b:A). comp)
       (u:{(lt (to_nat first) (to_nat n)) = tt})
       (v:{(lt (to_nat last) (to_nat n)) = tt})
       (w:{(le (to_nat first) (to_nat last)) = tt})
       . bool
```

Figure 15. The type of the binary search algorithm for word-indexed arrays.

Mutable Array Test		
Language	Avg Real Time	Size of Binary
HASKELL	1.18 s	581K
HASKELL (No GC)	0.49 s	
OCAML	0.61 s	131K
OCAML (No GC)	0.54 s	
GURU	0.42 s	37K

Figure 16. Average real time performance of word-indexed array test.

7.1 Mutable Array Test

The first test uses a verified version of a textbook binary search algorithm using word-indexed mutable arrays. What makes the implementation in GURU unique is that it uses the `warray_get` primitive method mentioned earlier in this paper. Recall that this method requires a proof that the index of the array that is to be accessed is less than the length of the array. Therefore, for binary search to succeed, we must derive a proof that the middle point to search is less than the length of the array. This proof is derived relatively easily given three other proofs: that the first index of the search space is less than the length of the array, that the last index of the search space is less than the length of the array, and that the first index of the search space is less than or equal to the last index of the search space. This gives our binary search function the type shown in Figure 15. Also of note is the use of a generic comparator function, allowing the algorithm to work over all types of word-indexed arrays. Supplying this comparator function to the search function is similar to defining and utilizing an instance of the `Ord` type class for a data type in HASKELL.

To test the performance of binary search, an ordered array of size 2^{20} is created and then each possible index is searched for. The total time to complete both of these tasks is shown in Figure 16. The different sizes of the binary executables are somewhat striking, so we include these as well. The GURU implementation beats the implementations in HASKELL and OCAML, with garbage collectors on. The reports of others, cited in the Introduction, about the performance penalty of garbage collection are confirmed here.

Queue Test		
Language	Avg Real Time	Size of Binary
HASKELL	1.08 s	614K
HASKELL (No GC)	0.53 s	
OCAML	0.66 s	132K
OCAML (No GC)	0.37 s	
GURU	0.60 s	37K

Figure 17. Average real time performance of queue test.

7.2 FIFO Queue Test

The second test involves the methods of the FIFO queue described earlier in the paper. To begin with, two new FIFO queues are created. Then the contents of *War and Peace* are enqueued to the first queue word by word. The first queue is then dequeued one element at a time, enqueueing the item into the second queue. Once the second queue is full the contents are again dequeued and the last element is printed to the console. The results of this test are shown in Figure 17. We see similar relative performance as for the previous example. For this benchmark, GURU is easily beaten by OCAML and HASKELL, if their garbage collectors are turned off. This is as one would expect, given the much greater maturity of their implementations. Note that the verified and unverified queues have indistinguishable performance (so we show just the results for the verified queue). We conjecture that branch prediction in the CPU masks the difference between the two implementations (the verified one does not need to do a run-time check to detect the situation which is proved to be impossible).

8. Discussion

There are several points arising in this study that deserve further discussion:

The status of memory leaks. Our `rheaplet` abstraction does not guarantee absence of memory leaks, and indeed, cyclic structures can be leaked. We will attempt to persuade the reader that this is not as great a shortcoming as it might seem. First, our goal is to apply verification judiciously for improved software quality. Memory leaks are certainly software flaws, but statically ensuring that they do not exist appears costly. Indeed, dynamic methods such as garbage collection are currently so widely for just that reason: they ensure that memory is not leaked, with no additional verification cost. There is a performance cost, which we are trying to avoid. One could imagine punting and adding garbage-collected regions as additional resource types. Indeed, our `rheaplet` data structure already compromises performance in order to reduce the burden of static verification.

If static verification of the absence of memory leaks is desired, then the approach we would advocate is to use a version of heaplets (not yet implemented) where the number of outstanding aliases is statically tracked in the heaplet's type. Verifying that the number of aliases is zero when the heaplet is consumed would require significant verification effort in general, since one would need to maintain invariants expressing the aliasing structure of the heaplet, in order to be able to show statically that all aliases have been consumed. For cyclic structures, it might be necessary to disassemble the structure carefully on deallocation, in a reverse manner to its construction. This choreography will be too intricate for the verification budget of many applications.

A final note on memory leaks is that to the best of our knowledge, there are no general methods for statically verifying the absence of live leaks. And of course, the absence of leaks of unreachable memory is simply an approximation to the truly desired property, which is that memory which will no longer be used will be reclaimed. Dynamic leak-pruning methods have been considered (see, e.g., (Bond and McKinley 2009)), but a general theory for statically guaranteeing the absence of live leaks where possible appears to be lacking from the literature.

On automating proofs. Our current burden of proof for our example using aliased heap-allocated data is similar to that of the 2008 version of the Ynot system, at least in being heavier than desirable (Nanevski et al. 2008). The 2009 version of Ynot greatly decreases the burden of manual proof through clever use of automated tactics (in the COQ proof assistant) (Chlipala et al. 2009). We also agree that automation must be applied to help reduce the manual effort for proofs about the heap. But the methodology for automation we are pursuing in GURU differs greatly from that of COQ, and of at least some of the tactics reported in (Chlipala et al. 2009). The `hypjoin` tactic mentioned above is sound, but also complete (under modest conditions about termination of the functional terms involved): if two terms are joinable modulo the given ground equations, without case splitting or induction, then `hypjoin` will determine that they are. It is our hypothesis that usability of powerful tactics is greatly increased by increased predictability. Heuristic methods, or methods lacking a simple independent delineation of the class of problems they can solve , seem likely to increase fragility of proofs and difficulties for users (particularly less proficient ones). It remains to be seen which tactics fitting such a description can be devised for reasoning about the heap, or more generally. Certainly one path is to try to extend methods like `hypjoin` to the non-ground case, but this is likely to be technically challenging.

The provable connection between aliases and heaplets. The use of heaplets to model just a portion of the heap is similar in spirit to the use of separating conjunction in the separation logic approach used by the Ynot authors (Nanevski et al. 2008; Reynolds 2002). It also distinguishes such approaches from previous work of Zhu and Xi on stateful views, where a type $t@l$ is used to express that an element of type t occurs at address l, in the single implicit global heap (Zhu and Xi 2005). Above we were forced to reason in more detail than we would like about the fact that aliases associated statically (via their `rheaplet_id`) with a heaplet are actually valid indices into that heaplet (considered as a list). It might be possible to reduce this burden of proof by adding some more direct language support for heaplets. For example, we could add a feature to add proofs automatically inside terms – via new assumption variables – that valid indices remain valid under all the heaplet operations (recalling that in the functional model, we never remove a value from a heaplet). This would greatly reduce the burden of proof for the verified queue example, which is mostly concerned with proving that aliases are valid heap indices.

9. Conclusion

We have seen how a resource typing framework based on two simple principles can support a variety of low- and higher-level imperative abstractions:

- Resources may be used by at most one party at a time.
- Resources may be divided into subresources, which must be returned before the resource as a whole is usable again.

Future work includes completing the detailed proof of type soundness, using the formulation devised above. On the practical side, our focus will be on reducing the burden of annotation and proof in the ways discussed in the previous section, and also by finding ways to eliminate the need for programmers to apply resource primitives like `inc`, `dec`, and `inspect` explicitly. A simple first approach might be to devise an algorithm that will automatically insert calls to these primitives, given types for all function input variables.

Acknowledgements: the U.S. NSF for support under award CCF-0841554; and the growing group of students and researchers experimenting with GURU, especially University of Iowa students Derek Bruce, Frank Fu, John Hughes, Tianyi Liang, Duckki Oe, Andrew Reynolds, and Greg Witt.

References

M. Bond and K. McKinley. Leak Pruning. In M. Soffa and M. Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.

C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 79–90, 2009.

Matthew Hertz and Emery D. Berger. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proc. 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 313–326. ACM, 2005.

D. Licata and R. Harper. A Formulation of Dependent ML with Explicit Equality Proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University School of Computer Science, December 2005.

Z. Luo. A Type-Theoretic Framework for Formal Reasoning with Different Logical Foundations. In M. Okada and I. Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 2008.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In James Hook and Peter Thiemann, editors, *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 229–240, 2008.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

N. Oury and W. Swierstra. The Power of Pi. In P. Thiemann, editor, *Proceedings of the 2008 SIGPLAN International Conference on Functional Programming*, 2008.

E Pasalic, J. Siek, W. Taha, and S. Fogarty. Concoqtion: Indexed Types Now! In G. Ramalingam and E. Visser, editors, *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2007.

A. Petcher. Deciding Joinability Modulo Ground Equations in Operational Type Theory. Master's thesis, Washington University in Saint Louis, May 2008. Available from `http://www.cs.uiowa.edu/~astump`.

A. Petcher and A. Stump. Deciding Joinability Modulo Ground Equations in Operational Type Theory. In S. Lengrand and D. Miller, editors, *Proof Search in Type Theories (PSTT), workshop affiliated with the Conference on Automated Deduction (CADE)*, 2009.

J. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science*, 2002.

T. Sheard. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification*, 2006.

A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.

F. Xian, W. Srisa-an, and H. Jiang. Garbage collection: Java application servers' Achilles heel. *Science of Computer Programming*, 70(2-3):89 – 110, 2008.

D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97. Springer, 2005.