A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs

PAUL HE, University of Pennsylvania, USA
EDDY WESTBROOK, Galois, Inc., USA
BRENT CARMER, Galois, Inc., USA
CHRIS PHIFER, Galois, Inc., USA
VALENTIN ROBERT, Galois, Inc., USA
KARL SMELTZER, Galois, Inc., USA
ANDREI ŞTEFĂNESCU, Galois, Inc., USA
AARON TOMB, Galois, Inc., USA
ADAM WICK, Galois, Inc., USA
MATTHEW YACAVONE, Galois, Inc., USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Verifying imperative programs is hard. A key difficulty is that the specification of what an imperative program does is often intertwined with details about pointers and imperative state. Although there are a number of powerful separation logics that allow the details of imperative state to be captured and managed, these details are complicated and reasoning about them requires significant time and expertise. In this paper, we take a different approach: a memory-safe type system that, as part of type-checking, extracts functional specifications from imperative programs. This disentangles imperative state, which is handled by the type system, from functional specifications, which can be verified without reference to pointers. A key difficulty is that sometimes memory safety depends crucially on the functional specification of a program; e.g., an array index is only memory-safe if the index is in bounds. To handle this case, our specification extraction inserts dynamic checks into the specification. Verification then requires the additional proof that none of these checks fail. However, these checks are in a purely functional language, and so this proof also requires no reasoning about pointers.

CCS Concepts: • **Software and its engineering → Software verification**; • **Theory of computation → Program specifications**; **Type structures**.

Additional Key Words and Phrases: Specification extraction, type systems, pointers, memory safety

ACM Reference Format:
Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. 2021. A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 135 (October 2021), 29 pages. https://doi.org/10.1145/3485512

Authors' addresses: Paul He, University of Pennsylvania, Philadelphia, USA; Eddy Westbrook, Galois, Inc., Portland, USA; Brent Carmer, Galois, Inc., Portland, USA; Chris Phifer, Galois, Inc., Portland, USA; Valentin Robert, Galois, Inc., Portland, USA; Karl Smeltzer, Galois, Inc., Portland, USA; Andrei Ştefănescu, Galois, Inc., Portland, USA; Aaron Tomb, Galois, Inc., Portland, USA; Adam Wick, Galois, Inc., Portland, USA; Matthew Yacavone, Galois, Inc., Portland, USA; Steve Zdancewic, University of Pennsylvania, Philadelphia, USA.

2475-1421/2021/10-ART135
https://doi.org/10.1145/3485512

1 INTRODUCTION

Verifying imperative programs is hard. Although there are frameworks like Iris [Jung et al. 2018, 2015] and the Verified Software Toolchain (VST) [Appel et al. 2014; Beringer and Appel 2019] that provide powerful separation logics to verify imperative programs, they require a high degree of time and expertise to effectively use them. In fact, figuring out how best to apply these tools is an active area of research, where new approaches and techniques are still being developed to make using these tools more practical [Jung et al. 2020; Krishna et al. 2020].

There is one verification approach, however, that has seen widespread adoption by a large number of developers: the Rust type system. Rust allows programmers to prove that their imperative programs are memory safe by writing types that track memory ownership; type-checking then ensures that data with these types is used only in memory-safe ways. The rate of adoption of Rust demonstrates that this approach is both intuitive enough for programmers to use and powerful enough to write, for instance, an operating system [Redox Developers [n.d.]]. The main limitation of Rust in the context of verification, though, is that it can only prove memory safety; how can we get more?

In this paper, we show how to extend the idea of a memory-safe type system like Rust's to prove functional correctness of imperative programs. The aspirational idea of our approach is that *imperative programs that are well-typed in a memory-safe type system should have equivalent pure functional interpretations*. More precisely, we formalize a relational type system that type-checks imperative programs, and, in the process, extracts a pure functional specification of a program's behavior. Programs over imperative data structures are extracted to the corresponding pure functions over functional versions of these data structures; e.g., a linked list reversal program becomes a pure function for reversing functional lists. The resulting specification can then be used to reason about the correctness of the program. Soundness of the type system implies that the low-level implementation simulates the high-level extracted specification. This approach of relating imperative programs to functional specifications is not new, and is in fact a key part of the DeepSpec [Koh et al. 2019] and Igloo [Sprenger et al. 2020] approaches, which have both been used effectively to verify complex and non-trivial programs. The main differentiator of our approach is that the functional specifications are generated automatically by type-checking the imperative programs, rather than being crafted by hand and manually proved to simulate their imperative programs. This has the potential to greatly reduce the effort required to verify imperative programs.

A key technical issue in trying to disentangle memory safety from functional specifications is that there are cases where the memory safety of an imperative program depends crucially on its functional behaviors. A common example is a program that manipulates arrays but omits bounds checks for performance reasons. Such a program is memory safe only if we can verify that all array index computations are in bounds. Array index computations are part of the functional specification of a program, since, intuitively, arrays represent lists or sequences and indexing into them extracts the ith element of such a list. Further, array index computations can involve arbitrary computation, making array bounds checking undecidable in general and thus difficult to perform with a type-checker.

To handle this issue, we allow our functional specifications to contain errors, which represent cases where type-checking fails. Thus, for instance, an array indexing operation in an imperative program translates to a functional specification that tests whether the computed index is in bounds and raises an error if it is not. To accommodate this, soundness for our type system is defined using a notion of bisimulation *up to errors in the specification*, where execution traces in the imperative program and its functional specification precisely correspond to each other except that errors in the specification can correspond to any trace in the imperative program. If we can prove that a specification never raises an error, then this reduces to the standard notion of bisimulation, which in turn ensures that the implementation is also error-free. Additionally, bisimulation implies that all temporal properties are preserved, so, e.g., if liveness and fairness properties can be proved of the extracted specification, then equivalent liveness and fairness properties are guaranteed to hold for the original imperative program.

This approach is implemented as a tool called Heapster, which itself is an extension of SAW, the Software Analysis Workbench [Chudnov et al. 2018; Dockins et al. 2016]. SAW has existing support for ingesting low-level input languages like LLVM. Heapster then performs specification extraction on ingested LLVM to generate functional models expressed in Coq using the type-checking process described in this paper. Heapster is currently being used to build a formally verified implementation of the IPSec protocol for a Department of Defense application. The current paper formalizes a simplified version of Heapster in Coq and proves it correct, in the sense that the generated specifications do indeed model the imperative programs they are generated from. While this paper focuses on the theoretical basis of Heapster, we also describe preliminary results for using the Heapster tool.[1]

The contributions of this paper are as follows:

- We propose a new technique for automatically extracting functional specifications from imperative programs, that is structured as a novel relational type system.
- The rules of the type system are proved sound by showing that the specifications they extract are bisimilar to the input imperative programs, which in turn ensures that all temporal properties of the program are preserved by specification extraction.
- The type system is built on a novel formulation of rely-guarantee separation logic, where separateness and related notions are defined in an abstract, general way that is independent of memories and pointers.
- The approach has been implemented as a tool called Heapster, which has been applied to real-world code implementing a non-trivial data structure and exposed some bugs in this code.
- The approach is formalized and proved correct in Coq, using interaction trees [Xia et al. 2020] to define the semantics of computations.

The rest of the paper is organized as follows. Section 1.1 gives a high-level overview of our approach. Section 2 gives background on interaction trees and how we use them to describe program semantics in our approach. Section 3 presents a novel semantics of separation logic called *rely-guarantee permission sets* that is used to relate imperative programs and their functional specifications. Section 4 defines the notion of bisimulation up to errors in the specification and uses it to define a semantic type system for extracting specifications. Section 5 describes the typing rules of our system. Section 6 describes the process of using the Heapster tool. Section 7 presents the results of using Heapster on real-world code. Finally, Sections 8 and 9 give related work and conclude.

1.1 Specification Extraction via Type-Checking

Our approach to specification extraction starts with a type system for memory safety. As in Rust, our type system controls pointer aliasing by requiring that mutable pointers cannot alias any other pointers. That is, every pointer is marked as either *shared read*, meaning it can be duplicated but not written to, or *exclusive write*, meaning it allows reads and writes but is a unique pointer that

[1] Our paper artifact can be found at https://doi.org/10.5281/zenodo.5519606. The latest versions of the formalization and tool can be found at http://github.com/GaloisInc/heapster-formalization and https://github.com/GaloisInc/saw-script respectively.

does not alias any other pointer value that is currently in scope. This typing discipline on pointers ensures a strong form of separation, or pointer locality, where writes to any pointer cannot affect any other pointer values in scope. This restriction, in turn, guarantees powerful properties like race-freedom.

The key observation of our approach is that programs with this form of pointer locality essentially behave like pure functional programs over the current values of all of the pointers they manipulate. That is, an imperative program satisfying this typing discipline that has N pointer variables can be described by a pure functional program with N inputs and N outputs, representing the input and output values of these pointers.[2] Pointer locality ensures that modifications of one pointer variable will not affect the values of any of the others, and so these N inputs and outputs in the corresponding functional program can be kept as distinct values. This in turn ensures that this translation to a functional description can be done compositionally, without requiring any reasoning about potential pointer aliasing or concurrent mutation of pointer values.

In general, however, imperative programs can manipulate arbitrary data structures with an unbounded number of pointers, and so translating to a specification over a fixed number N of pointer values is insufficient. To address this, we use a *permission type system* that describes the memory shapes and associated pointer access permissions associated with all values in scope. These permission types can describe arbitrary recursive data structures with an unbounded number of pointers. Our approach then translates the input and output permission types of a program fragment to pure functional types by erasing the pointer types, that is, by replacing pointer types with the unit type. For programs with N pointer variables, this translation corresponds to N input and output values, as described above. Well-typed program fragments are then translated in a compositional manner to functional specifications whose input and output types are given by translating their input and output permission types.

For example, consider the following C definition of linked lists of buffers for storing 64-bit values:

```
typedef struct bufs { struct bufs *next; int64_t len; int64_t data[]; } bufs;
```

With this definition, the pointer type `bufs*` represents a list of buffers, where each buffer is represented as an array of 64-bit values. A pointer of this type is either `NULL`, representing the empty list, or a valid pointer to a `bufs` structure, which contains a recursive (`NULL` or valid) pointer to the tail of the list, the length of the buffer, and the buffer contents.

To describe linked list pointers in Heapster, a user could define the Heapster type

$$\begin{aligned}\mathsf{Bufs} := \mu X.\mathsf{eq}(\mathsf{VNum}\ 0) \vee (\exists len\!:\!\mathsf{BV}\ 64.\mathsf{ptr}((\mathsf{W},0) \mapsto X) * \mathsf{ptr}((\mathsf{W},8) \mapsto \mathsf{eq}(\mathsf{VNum}\ len))\\ * \mathsf{arr}((\mathsf{W},16,len) \mapsto \exists x\!:\!\mathsf{BV}\ 64.\mathsf{eq}(\mathsf{VNum}\ x)))\end{aligned}$$

Heapster types, also called *permission types* or just *permissions*, represent knowledge about the shape of a C value along with permissions about what actions can be performed with the value. The Bufs type is, at the top level, a recursive type where the variable X recursively refers to Bufs itself in its definition. The body of this recursive type is a disjunctive type, stating that a C value is either of the left-hand or right-hand type. The left-hand type is an equality type eq(VNum 0), stating that a value is known to be equal to the C numeric value 0, i.e., a null pointer. The VNum constructor in our semantic model builds imperative values that are numeric. The right-hand type is a separating conjunction of two pointer types and an array pointer type inside an existential permission that quantifies over some 64-bit bitvector value *len*. This permission type states that offset 0 points to a value which is itself a pointer satisfying Bufs, offset 8 points to a numeric value that is currently equal to *len*, and offsets starting at 16 point to an array of *len* 64-bit array cells.

[2] Technically speaking, the values of read-only pointers cannot change, and so do not necessarily need to be returned as outputs; this is one of many optimizations that can be applied as part of the specification extraction process.

Each array cell satisfies the permission $\exists x\!:\!\mathsf{BV}\ 64.\mathsf{eq}(\mathsf{VNum}\ x)$, which intuitively states that it is a numeric value. Because numeric values are so common, we use the abbreviation int64 to refer to this permission.

More generally, Heapster pointer types have the form $\mathsf{ptr}((rw, o) \mapsto P)$, where rw is either R (read) or W (write), o is a numeric pointer offset, and P is a permission type. This permission allows reading or writing, depending on rw, the value at offset o of a pointer, and further states that the current value at that location has permission type P, which is called the *content permission* of the pointer permission. Additionally, write permissions are *exclusive*, just like mutable pointer types in Rust, meaning that the Heapster type system ensures that the pointer value of a pointer with a write permission never aliases the pointer value of any other pointer permission in scope. Thus the Heapster type system is a form of separation logic, where the separating conjunction implies that the write pointer permissions on one side do not alias any pointer permissions on the other. Array pointer types have the form $\mathsf{arr}((rw, o, l) \mapsto P)$, indicating an array that can either be read or written, depending on rw, starting at offset o and having length l, which gives the number of array cells (not the number of bytes). The current value of each array cell has permissions P. For cell size sz, this array pointer type is equivalent to the conjunction of l different pointer permissions at offsets o, $o + sz$, $o + 2 * sz$, etc. The Heapster tool allows different cell sizes, but the Coq formalization uses a simplified version where all array cells have the size of one imperative value.

In order to type-check iterative loops that traverse a recursive data structure like a linked list, it is often necessary to use invariants stating that one variable is reachable from another by following zero or more pointer fields. For linked lists in particular, this form of reachability is called a *list segment* [Reynolds 2002]. In Heapster, we capture this notion with *reachability permissions*, which are recursively-defined permissions with a free variable r that specifies the value that is reached by this permission. For example, we define the list reachability permission

$$\begin{aligned}\mathsf{BufsR}(r) := \mu X.\mathsf{eq}(r) \vee (\exists len\!:\!\mathsf{BV}\ 64.\mathsf{ptr}((\mathsf{W}, 0) \mapsto X) * \mathsf{ptr}((\mathsf{W}, 8) \mapsto \mathsf{eq}(\mathsf{VNum}\ len))\\ * \mathsf{arr}((\mathsf{W}, 16, len) \mapsto \exists x\!:\!\mathsf{BV}\ 64.\mathsf{eq}(\mathsf{VNum}\ x)))\end{aligned}$$

Permission $x\!:\!\mathsf{BufsR}(r)$ intuitively states that x either equals r or points to a bufs structure whose next field recursively satisfies $\mathsf{BufsR}(r)$. The Bufs permission can be recovered from this definition by using $\mathsf{BufsR}(\mathsf{VNum}\ 0)$, which intuitively says that NULL is reachable from the value with this permission. Reachability permissions satisfy *reflexivity*, meaning that any variable reaches itself, as well as *transitivity*, meaning that if x reaches y and y reaches z then x reaches z. These properties are used as type-checking rules.

The first step of specification extraction in Heapster is type-checking, where the user specifies input and output permissions types for an imperative function and Heapster then type-checks it with respect to these permission types. The user can also supply type-checking hints, which are useful for specifying loop invariants. Heapster operates on LLVM code, as LLVM is low-level enough that Heapster does not have to handle all the complexities of C semantics but high-level enough to abstract away the details of individual architectures. The accompanying Coq formalization uses a more abstract, simplified assembly language. Rather than giving the type-checking rules in detail here, we illustrate the Heapster type-checking process using examples. We use examples written in C, to make them more readable than LLVM, but write them in a low-level style that is closer to LLVM than standard C conventions. This includes using single static assignment (SSA) form and using explicit allocation and deallocation of local variables, which are explicitly written as pointers.

Figure 1 shows how to type-check the function clearbufs, which zeroes out the first element of every buffer in a buffer list. The gray boxes in the figure show the inferred Heapster types at each point in the program. The idea of this function is that buffers are null-terminated, and so

```
void clearbufs (bufs *lst) {
  lst:Bufs
  const bufs **lp = malloc(8);
  lst:Bufs, lp:ptr((W, 0) ↦ true) ∗ block(8)
  lst:Bufs, lp:lclptr(true)
  *lp = lst;
  lst:Bufs, lp:lclptr(eq(lst))
  while (true) {
    lst:BufsR(z), z:Bufs, lp:lclptr(eq(z))†
    const bufs *l = *lp;
    lst:BufsR(l), l:Bufs, lp:lclptr(eq(l))
    ..., l:eq(VNum 0)  OR  ..., l:∃len:....
    if (l == NULL) {
      lst:BufsR(l), l:eq(VNum 0), lp:lclptr(eq(l))
      free(lp);
      lst:BufsR(l), l:eq(VNum 0)
      return;
      lst:Bufs
    } else {
      ..., len:BV 64 ∗ l:... ∗ arr((W, 16, len) ↦ int64)
      ... ∗ l:... ∗ ptr((W, 16) ↦ int64)
                  ∗ arr((W, 24, len − 1) ↦ int64)
      l→ data[0] = 0;
      ... ∗ l:... ∗ ptr((W, 16) ↦ eq(VNum 0))
                  ∗ arr((W, 24, len − 1) ↦ int64)
      ..., len:BV 64 ∗ l:... ∗ arr((W, 16, len) ↦ int64)
      const bufs *nxt = l→ next;
      ..., l:ptr((W, 0) ↦ eq(nxt)) ∗ ..., nxt:Bufs
      *lp = nxt;
      lst:BufsR(l), l:ptr((W, 0) ↦ eq(nxt)) ∗ ...,
                          nxt:Bufs, lp:lclptr(eq(nxt))
      lst:BufsR(nxt), nxt:Bufs, lp:lclptr(eq(nxt))
}}}
```

Fig. 1. Type-Checking the clearbufs Function Against the User-Supplied Type lst:Bufs ⊸ lst:Bufs (The gray types are inferred by the tool, except the † loop invariant)

this operation clears every buffer, marking it as empty. The function is type-checked against the user-supplied type lst:Bufs ⊸ lst : Bufs, stating that the input value lst satisfies the Bufs permission on input and that it retains this permission on output. The input list lst is first assumed to have type Bufs. The first line allocates a local variable lp. Mutable local variables in C are compiled to pointers to stack-allocated objects, so this approach of allocating lp as a pointer to a linked list pointer is similar to how a mutable local variable of type bufs* is compiled.[3] Allocation is performed by calling malloc with the size of 8 bytes, which is the size of a 64-bit pointer.[4] The result is a pointer value for lp that points to some uninitialized value. The fact that it is uninitialized is represented using the vacuous true permission. The newly-allocated value lp also satisfies the permission block(8), which says that lp points to the beginning of an allocated block whose size is 8 bytes. This information is needed for deallocation by the free function later. For the sake of brevity, we introduce the abbreviation $\mathsf{lclptr}(P) = \mathsf{ptr}((\mathsf{W}, 0) \mapsto P) * \mathsf{block}(8)$ for a local variable pointer whose current value satisfies P. The input value lst is then assigned to (the contents of) the local variable lp. This replaces permission ptr((W, 0) ↦ true) on lp with ptr((W, 0) ↦ eq(lst)). That is, the resulting permission after this assignment is lp:lclptr(eq(lst)).

After this, the function enters the while loop, which has invariant lst:BufsR(z), z: Bufs, lp : lclptr(eq(z)). Loop invariants with reachability permissions currently must be input by the user, as they are not inferred by the tool. This invariant states that the current value of lp is some value z that satisfies Bufs and that is reachable by

[3] In order to simplify the presentation, we model stack allocation here with malloc and free; the Heapster tool models it more precisely, but we omit these details here.

[4] In fact malloc requires a permission to allocate, discussed in Section 5.5, which intuitively represents permission to modify the free list structure maintained by the allocator. This permission is implicitly added everywhere by the Heapster tool, so the user need not reason about it explicitly and we omit it from Figure 1.

a list segment starting at `lst`. On entry to the loop, z is equal to `lst`. The reachability permission z : BufsR(`lst`) part of the loop invariant is proved by the reflexivity rule for BufsR, while the remainder of the loop invariant is already available in the current permission typing context.

In the `while` loop, the value of z is first read out to the variable `l`. In order to type-check the following null test, the recursive Bufs permission must be unfolded to a disjunctive permission, which must then be eliminated. Disjunctive elimination means that the remaining code is type-checked twice, once with the left-hand and once with the right-hand disjunct. Thus, Figure 1 displays the two disjunctive possibilities for the permission on `l`. In the left-hand case of the disjunctive elimination, we have that `l` equals `NULL`, so the null test succeeds and type-checking proceeds into the `if` branch of the conditional statement. In this case, `lp` is deallocated by passing it to `free`. Type-checking `free` requires both a block(N) permission, so that we know how many bytes are going to be freed, along with exclusive write permissions to those N bytes. These permissions are then consumed by the call. We are then left with a reachability permission stating that `lst` reaches `l`. After the `return`, this reachability permission is combined with the fact that `l` equals `NULL` to get out the required Bufs permission on `l` for output.

In the right-hand case, `l` is a non-null pointer to a `bufs` struct. This means the null test fails, and type-checking proceeds into the `else` branch. The first statement of this branch zeroes out the first element of the `data` array. In order to type-check this assignment, an existential elimination step is first performed on the Bufs permission on `l`. This binds a bitvector variable *len* and leaves pointer permissions on `l` for the `next` and `len` fields along with the array pointer permission arr((W, 16, *len*) ↦ int64) for `data`. The resulting permission is displayed in Figure 1 immediately after the `else`. The array permission is then split into a single pointer permission for offset 16, which is the 0th cell of the `data` array, along with an array pointer permission for the remaining cells of `data`. The pointer permission can then be used to type-check the assignment to `l→ data[0]`.

In general, however, splitting array permissions in this way *is not memory safe*, because nothing in our current typing context ensures that the index where the splitting occurs is inside the array bounds. Specifically, in this case, the length of the array could be zero. The standard approach to proving memory safety in separation logic is to encode the required predicates on numerical values into the preconditions of the function. In this case, the precondition would require that all arrays in the input buffer list have non-zero length. Although the required precondition seems straightforward in this case, the preconditions for a program can become arbitrarily complex, making it difficult to automate the generation and discharging of these preconditions. The user must then become involved, and must learn to use a combined logic of both separation and also properties on numeric values. Instead, the Heapster approach allows array bounds checks to be *deferred* until after separation type-checking, by encoding the bounds checks into the extracted functional specification. That is, the extracted functional specification for this array splitting step contains the specification "if index in bounds then ... else error". This allows the verification of numeric properties to be *decoupled* from proving separation properties, which in turn leads to a much simpler logic of separation that can be automated as a type system. Although the unary version of the Heapster type system, which ignores the generated specifications, does not guarantee memory safety by itself, it does satisfy a slightly more subtle property: it guarantees memory safety *up to errors in the generated specification*. That is, if we type-check a program and also verify that its specification is error free, we guarantee memory safety. We have found, however, that even the type-checking step can help in detecting memory safety bugs, as discussed in Section 7.

After `l→ data[0]` is assigned 0, the corresponding pointer permission has contents permission eq(`VNum 0`). This is generalized to int64 with an existential introduction step. The resulting pointer permission is then re-combined with the array pointer permission for the rest of the `data` array to get back the full array pointer permission for `data`. Unlike the splitting step, recombining array permissions is always memory safe. The `nxt` variable is then assigned the value of `l`→ `next`, giving the Bufs permission to `nxt` and replacing the corresponding pointer permission on `l` with $\mathtt{ptr}((\mathsf{W},0) \mapsto \mathsf{eq}(\mathtt{nxt}))$. This is a form of *borrowing*, where the data structure referred to by `l` contains a field whose permissions are currently being held by another variable in scope, in this case `nxt`. This borrowing is needed in order to perform the next iteration of the `while` loop on this value. The `lp` variable is then updated to equal `nxt`. After this, we have that `lst` reaches `l`, which in turn reaches `nxt` in one step. Transitivity of BufsR then yields that `lst` reaches `nxt`, after consuming the permissions on `l`. This intuitively "gives back" the permissions on `l` to the reachability permission on `lst`, which were borrowed in the previous iteration. The invariant needed to jump back to the head of the `while` loop is then re-established, and the control flow jumps back to the head of the loop.

Heapster permission types not only describe the memory shape and associated permissions associated with an imperative value, but are also translated to a purely functional type whose elements are used to represent C values with that type. This translation is performed in a syntax-directed, compositional way. Recursive permissions are translated to recursive types, disjunctive permissions are translated to sum types, and separating conjunctions are converted to product types. Existential permissions are converted to dependent pairs, i.e., sigma types. Equality permissions are converted to the unit type Unit, because, intuitively, the identity of a C value with equality type is already known, so the functional specification need not keep any information about it. Pointer permissions $\mathtt{ptr}((rw,o) \mapsto P)$ are erased, meaning they are translated to the result of translating the contents type P. Array pointer permissions $\mathsf{arr}((rw,o,l) \mapsto P)$ are translated to sized vectors $\mathsf{Vect}\ T\ l$, where T is the translation of P. Using these rules, Bufs is translated to the recursive type

$$\mathsf{BufsTrans} := \mu X.\mathsf{Unit} + (\Sigma len{:}\mathsf{BV}\ 64.\ X \times \mathsf{Unit} \times \mathsf{Vect}\ (\Sigma x{:}\mathsf{BV}\ 64.\ \mathsf{Unit})\ len)$$

Note that this type is isomorphic to the type of lists of vectors of 64-bit bitvectors of some length len. Similarly, the list reachability permission $\mathsf{BufsR}(r)$ is also translated to the BufsTrans type, since it differs only in the target of the equality permission it uses in the left-hand disjunct.

The core idea of Heapster specification extraction is that each typing rule used to type-check an imperative program can be translated to a purely functional program fragment, whose input and output types are the result of translating the input and output Heapster permissions of the typing rule. The introduction and elimination rules for each permission construct are translated to the introduction and elimination combinators for the corresponding functional type: the fold and unfold type-checking rules on recursive permissions are translated to functions for folding and unfolding recursive types; uses of disjunctive introduction become the constructors of the sum type, while disjunctive elimination becomes sum elimination, i.e., a pattern-match; and the existential introduction and elimination rules are translated to construction and projection of dependent pairs. Because pointer permissions are erased by the translation, the translation of the pointer introduction and elimination rules, along with those of the load, store, `malloc`, and `free` typing rules, are straightforward. Control structures are then translated to the same control structures in the extracted specification.

We defer detailed discussion of this translation to later sections, and instead describe it at a high level by continuing our `clearbufs` example from above. The specification extracted from this example is given in Figure 2, as a monadic function clearbufs_spec on the translation BufsTrans of the input and output permission `l`:Bufs. The `while` loop is translated to the iteration operation

$$\mathsf{iter} : (I \to \mathsf{CompM}\ (I + R)) \to I \to \mathsf{CompM}\ R$$

that takes a monadic function f from inputs of type I to outputs that are either return values of type R or input values of type I for a successive call to f, along with an initial input value

$$\begin{array}{lcl}
\text{clearbufs\_spec} & : & \text{BufsTrans} \to \text{CompM BufsTrans} \\
\text{clearbufs\_spec} & = & \lambda lst.\,\text{iter}\ (\lambda x : \text{BufsTrans} \times \text{BufsTrans} \times \text{Unit}.\ \text{case unfold}\ (x.2)\ \text{of} \\
 & & \qquad\qquad \text{inl unit} \Rightarrow \text{return}\ (\text{inr}\ (\text{append}\ x.1\ (\text{fold}\ (\text{inl unit})))) \\
 & & \qquad\qquad \text{inr}\ \{len, (nxt, \text{unit}, d)\} \Rightarrow \\
 & & \qquad\qquad\quad \text{trySplit}\ d\ 0\ (\lambda hd.\,\lambda tl. \\
 & & \qquad\qquad\qquad \text{return}\ (\text{inl}\ (\text{append}\ x.1\ [\langle\{0, \text{unit}\}\rangle +\!\!+\, tl], nxt, \text{unit})))) \\
 & & \qquad\quad ([\,], lst, \text{unit})
\end{array}$$

Fig. 2. Specification Extraction for the clearbufs Function

of type I, and iteratively calls f on the successive inputs until it returns an output of type R, if ever. The input type used for iter is the product type BufsTrans × BufsTrans × Unit corresponding to the translation of the invariant permission lst : BufsR(z), z : Bufs, lp : lclptr(eq(z)) used to type-check the while loop in clearbufs. The initial value passed to iter results from translating the portion of clearbufs before the while loop begins. This yields the triple $([\,], l, \text{unit})$ of the empty list, l, and the unit object: the first element is the translation of the reflexivity rule used to prove lst:BufsR(lst), the second corresponds to the input Bufs permission on lst cast to apply to the existentially-quantified variable z, and the third is the trivial unit object. Note that the translation of malloc is trivial, because it returns pointer permissions whose content permissions are all the trivial true permission, the translation of which is just the unit type.

The body of the iterated function used with iter unfolds and then pattern-matches on the second projection of the input, corresponding to the recursive permission unfolding step followed by the disjunctive elimination step on the value l read from *lp. The left-hand case of this pattern-match corresponds to the null test succeeding. This case returns an inr tagged value, indicating a final return value of the iter, which corresponds to the return statement. The value returned is the list append of $x.1$ and the term fold (inl unit), which corresponds to the BufsR transitivity rule applied to lst:BufsR(l), whose translation is $x.1$, and the permission l:eq(VNum 0), coerced to a Bufs permission via disjunctive introduction followed by recursive permission folding.

The right-hand case of the pattern-match corresponds to the null test failing, in which case l points to a struct that is represented by a dependent pair of a length len and a triple of: a BufsTrans value nxt for the next field; a unit object for the equality permission on the len field; and a vector $data$ for the data field. The typing rule that splits the array permission is translated to the function

$$\text{trySplit} : \text{Vect}\ A\ len \to \forall i : \text{BV}\ 64.(\text{Vect}\ A\ i \to \text{Vect}\ A\ (len - i) \to \text{CompM}\ R) \to \text{CompM}\ R$$

that takes in a vector v of length l and an index i and tries to split the vector at index i. If $i \leq len$, the split succeeds, and the head and tail are passed to the supplied function. Otherwise, an error is thrown. In the case of clearbufs_spec, the d vector is split at offset 0. If the split succeeds, the tail of d, passed as the local variable tl, is appended, using notation $+\!\!+$, to $\langle\{0, \text{unit}\}\rangle$, which is notation for the one-element vector of the dependent pair of 0 and the unit value. This is the translation of the array recombining rule. The resulting vector is then put into a one-element list, which is appended to the BufsTrans list in $x.1$. This step corresponds to the BufsR transitivity rule at the end of the while loop that combines the reachability permission from lst to l with that from l to nxt. Finally, this list is combined with nxt and the unit value to make a triple, corresponding to the proof of the loop invariant at the bottom of the while loop. This triple is returned as the next input to iter using the inl constructor.

As this example shows, the Heapster approach can extract pure functional specifications from nontrivial imperative programs. The remainder of the paper puts this process on a sound formal foundation by proving a bisimulation between the specifications and the imperative programs from which they are extracted.

2 INTERACTION TREES FOR PROGRAM SEMANTICS

To represent the semantics of both our imperative programs and their functional specifications, we use a dependently-typed lambda-calculus with sums, products, dependent pairs, and strictly positive inductive types. Interaction trees [Xia et al. 2020] are also added to this language, as described below. We use a very shallow denotational semantics of this language, mapping, e.g., lambda-abstractions to meta-theoretic functions, products to meta-theoretic pairs, etc. Because this semantics is so shallow, and the terms of our language are very close to the terms in the meta-theory defining their semantics, we sometimes blur the distinction between object and meta-language when it is convenient. In fact, in the accompanying Coq development, we simply use a sub-language of Coq, the meta-theory, as our object language.

Among other things, this denotational semantics justifies the use of set-theoretic notation below. More specifically, when it does not cause confusion, we often use a type T as the set of ground values of type T, writing $e \in T$ to say that e is a ground expression of type T and use $P \subseteq T$ to indicate that P is a predicate over these ground values.

To represent computations in this language, we use a monad based on *interaction trees* [Xia et al. 2020], a coinductive framework for defining potentially infinite computations using a form of free monads [Swierstra 2008]. Given a function E on types and a type R, the set itree $E\ R$ of *interaction trees with events from E and return type R* is defined as the greatest fixed-point of the constructors

$$\mathsf{return} : R \to \mathsf{itree}\ E\ R \qquad\qquad \tau : \mathsf{itree}\ E\ R \to \mathsf{itree}\ E\ R$$
$$\mathsf{vis} : \forall X. E\ X \to (X \to \mathsf{itree}\ E\ R) \to \mathsf{itree}\ E\ R$$

Each interaction tree, or itree, is a computation that either returns a value of type R or takes a "computation step" followed by a continuation that gives the rest of the computation. A step can either be an *internal* step, denoted with the τ constructor, or an *external* step, also called a *visible* step, denoted with the vis constructor. Internal steps have no effect, while external steps mean that some *event* of type $E\ X$ occurs. Events can interact with the environment as well as expect a response. The type argument X denotes the type of this response, which can then be consumed by the remaining computation. This is denoted by putting the remaining computation inside a function that takes in an X. Because itrees are coinductive, they can contain potentially infinitely many steps. For example, the infinite itree $\mathsf{spin} = \tau\ (\tau\ (\ldots))$ denotes the computation that runs forever and does nothing. We use t, possibly with subscripts, to denote itrees.

Itrees form a monad, with return as a unit and a bind operation >>=, which is written infix, as $t \mathbin{>\!\!>\!\!=} f$. This intuitively represents the sequencing of any events in t followed by those in f, where f consumes the return value(s) of t. We use notation $x \leftarrow t_1; t_2$ for $t_1 \mathbin{>\!\!>\!\!=} \lambda x.\, t_2$. Itrees also admit an *iteration operation*

$$\mathsf{iter} : (I \to \mathsf{itree}\ E\ (I + R)) \to I \to \mathsf{itree}\ E\ R$$

which takes a function f from inputs of type I to computations that return either a return value of type R or another input of type I to be iteratively passed to f again. Each call to f inserts an internal τ step. Thus, for instance, the infinite loop spin can be written as $\mathsf{iter}\ (\lambda x.\, \mathsf{return}\ (\mathsf{inl}\ x))\ \mathsf{unit}$, which takes the unit value unit as input, passes it to the lambda-abstraction, get the unit value returned by the lambda-abstraction, passes it to the lambda-abstraction again, etc.

In this paper, we use events for state, binary choice, and exceptions. We omit the details of defining the corresponding event type, and simply use CompM $S\ R$ for the type of itrees containing these events, where S is the state type. This yields operations

$$\mathsf{modify} : (S \to S) \to \mathsf{CompM}\ S\ S \qquad\qquad \mathsf{throw} : \mathsf{CompM}\ S\ R$$
$$\mathsf{or} : \mathsf{CompM}\ S\ R \to \mathsf{CompM}\ S\ R \to \mathsf{CompM}\ S\ R$$

Intuitively, $\mathsf{modify}\ f$ applies f to the current value of the state, returning its value before f was applied. This allows the state to be both read and written in a single operation. The operation $\mathsf{or}\ t_1\ t_2$ non-deterministically chooses either t_1 or t_2. Finally, throw denotes the erroneous computation.

To give a semantics to these computations, we define the *steps-to* relation $(s_1, t_1) \longrightarrow (s_2, t_2)$ on pairs of a state of type S and an itree of type $\mathsf{CompM}\ S\ R$ by the following three cases:

$$\begin{array}{rcl} (s, \tau\ t) & \longrightarrow & (s, t) \\ (s, \mathsf{modify}\ f \mathbin{>\!\!>\!\!=} g) & \longrightarrow & (f\ s, g\ s) \\ (s, \mathsf{or}\ t_1\ t_2) & \longrightarrow & (s, t_i) \end{array}$$

The functional specifications that are extracted from imperative programs are modeled as computations over the unit type as the state type, that is, as computations of type $\mathsf{CompM}\ \mathsf{Unit}\ R$ for some R. For imperative programs, the state type is the type Mem of memories in a simplified, 64-bit version of the CompCert memory model [Leroy and Blazy 2008]. In this memory model, a *memory* is a finite map from natural numbers to blocks of allocated memory, where a *block* is a finite sequence of imperative values. Imperative values are elements of the inductive type Val defined with constructors $\mathsf{VNum} : \mathsf{BV}\ 64 \rightarrow \mathsf{Val}$ and $\mathsf{VPtr} : \mathsf{Nat} \rightarrow \mathsf{BV}\ 64 \rightarrow \mathsf{Val}$, where VNum builds *numeric* 64-bit values and VPtr builds a *pointer* from a block number plus an offset into that block. For the sake of simplicity, the accompanying Coq formalization uses natural numbers in place of bitvectors in Val.

Translating imperative programs into an itree semantics has been covered in other work [Xia et al. 2020], so we only give a high-level description here. This translation takes as input a control flow graph (CFG) representation of the program in single static assignment (SSA) form. It starts by defining input types $I_1, \ldots, I_N$ for each of the N basic blocks of the program, where I_i is the product $\mathsf{Val} \times \ldots \times \mathsf{Val}$ of values, with one value for each input register of block i. Each block is then translated to a monadic function $\mathsf{blockF}_i : I_i \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ ((I_1 + \ldots + I_N) + \mathsf{Val})$ that either returns a final result of type Val to be returned from the overall function, or an input of type I_j, indicating a jump or conditional branch to block j. This is done by translating each instruction to a monadic function. Because the program is in SSA form, each instruction uses a fresh output register, and there is no need to model mutation of registers. Arithmetic and comparison operations on numeric values are modeled using the function $\mathsf{getNum} : \mathsf{Val} \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ (\mathsf{BV}\ 64)$ to coerce imperative values to numeric values, if possible, and otherwise throw an error. Pointer reads, pointer writes, allocation, and deallocation are performed by monadic operations that query and update the state, also using throw for undefined or erroneous computations. Again, we omit the details, only mentioning here that these operations have the following types:

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{load} & : & \mathsf{Val} \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ \mathsf{Val} & \mathsf{malloc} & : & \mathsf{Val} \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ \mathsf{Val} \\ \mathsf{store} & : & \mathsf{Val} \rightarrow \mathsf{Val} \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ \mathsf{Unit} & \mathsf{free} & : & \mathsf{Val} \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ \mathsf{Unit} \end{array}$$

The individual blockF_i functions that are generated from this translation are then combined using sum elimination (i.e., pattern-matching expressions) into a function f of type $(I_1 + \ldots + I_N) \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ (I_1 + \ldots + I_N) + \mathsf{Val}$. Assuming that block 1 is the initial entry of the function, the translation uses f to build the monadic function

$$\lambda x.\, \mathsf{iter}\ f\ (\mathsf{inl}\ x) : I_1 \rightarrow \mathsf{CompM}\ \mathsf{Mem}\ \mathsf{Val}$$

for the overall semantics of the function.

For the sake of presentation and simplicity, we focus on iteration rather than general recursion. Although itrees do admit a fixed-point operation for defining recursion, this operation changes the event type E to include additional recursive call events, which adds an extra layer of complexity to the definition of typing. We do not anticipate that this will introduce any fundamental difficulties beyond this added layer of complexity. Note that the Heapster tool does handle recursion.

3 RELY-GUARANTEE PERMISSIONS

In this section we will formalize the concepts that underlie the types that we will use to extract functional specifications from imperative programs. These types are built from a novel concept called *rely-guarantee permissions*, which define a form of separation logic of what state updates a permission allows (the "guarantee") and what state updates it can tolerate happening in parallel (the "rely"). Our approach is very general, in that it abstracts out the details of memories and pointers. Instead, rely-guarantee permissions can be defined for *any* arbitrary state type; separateness and separating conjunction are also defined in a general, semantic way. This makes it different from past approaches that incorporate rely-guarantee into separation logic [Dodds et al. 2009; Feng 2009], as these approaches define permissions and separateness in terms of memory locations. Our logic also makes the separating conjunction operation total, where the conjunction of two permissions is a form of inconsistent permission; this avoids needing to prove separateness side conditions when using the logic.

Definition 3.1 (Rely-guarantee permission). Given state type S, we define a *(rely-guarantee) permission over* S as a tuple $\pi = (R, G, P)$ where

(1) R and G are preorders on S
(2) P is a predicate on S
(3) P respects R, i.e. if $x\,R\,y$ and $P(x)$ then $P(y)$

The *rely* preorder R specifies the state changes that other processes or actions independent of this permission are allowed to perform while this permission is being held. The *guarantee* preorder G specifies what state changes this permission allows the holder to perform. Finally, P is a *precondition* that describes the assumptions this permission makes about the state. The fact that the precondition respects the rely means that no other code is allowed to violate the precondition while its owner expects it to hold, though, as we shall point out below, a permission is allowed to violate its own preconditon. We write perm_S for the set of permissions over S. We use the notations R_π, G_π, and P_π to denote the respective components of a permission π.

Pointer permissions are defined as follows. Given $s \in \mathsf{Mem}$, $p \in \mathsf{Val}$, and $v \in \mathsf{Unit} + \mathsf{Val}$, let $\mathsf{read}\ s\ p : \mathsf{Unit} + \mathsf{Val}$ be the operation that returns the Val at the address referenced by p, if p is a valid pointer, or $\mathsf{inl\ unit}$ otherwise, and let $\mathsf{write}\ s\ p\ v : \mathsf{Unit} + \mathsf{Mem}$ be the operation that returns the new memory resulting from setting the value pointed to by p to v, or returns $\mathsf{inl\ unit}$ if p is not a valid pointer in s. Note that write can deallocate p by writing $\mathsf{inl\ unit}$. We then define the *pointer read* and *write permissions* on values $p, v \in \mathsf{Val}$ as follows:

$$\begin{aligned}
\mathsf{ptrp}_\mathsf{R}(p,v) &= (\{\, (s_1,s_2) \mid\ \mathsf{read}\ s_1\ p = \mathsf{read}\ s_2\ p \,\},\ \ (=),\ \ \{\, s \mid\ \mathsf{read}\ s\ p = \mathsf{inr}\ v \,\})\\
\mathsf{ptrp}_\mathsf{W}(p,v) &= (\{\, (s_1,s_2) \mid \mathsf{read}\, s_1\, p = \mathsf{read}\, s_2\, p \,\},\ \ \{\, (s_1,s_2) \mid \exists v'.\mathsf{write}\, s_1\, p\, v' = \mathsf{inr}\ s_2 \,\},\\
&\qquad \{\, s \mid \mathsf{read}\, s\, p = \mathsf{inr}\, v \,\})
\end{aligned}$$

These are the read and write equivalents of the $p \mapsto v$ proposition of standard separation logic stating that p is a pointer that currently points to value v. The rely of each permission allows anything in the state to be modified other than the value pointed to by p, while the precondition requires p to be a valid pointer which currently points to value v. The guarantee for ptrp_R does not allow any updates to be performed, while the guarantee for ptrp_W allows the value pointed to by p to change. Note that this guarantee allows ptrp_W to violate its own precondition, by writing a new value v' to p; in this case, the typing rules presented in Section 5 update it to a new permission $\mathsf{ptrp}_\mathsf{W}(p, v')$. Also note that write permissions allow a pointer to become deallocated, and thus are more like exclusive permissions in other systems.

Arbitrary predicates $P \subseteq S$ can be lifted to the *predicate permission* $\mathsf{pred}(P) = (R, =, P)$, where R is the relation $\{\, (s_1, s_2) \mid\ P(s_1)$ iff $P(s_2) \,\}$. Predicate permissions can tolerate any updates in the state that preserve P or its negation, but do not themselves permit any, and require P to hold. As an example, the $\mathsf{block}(N)$ permission type used in Section 1.1 can be defined using the predicate permission $\mathsf{pred}(\mathsf{blockp}(p, N))$, where blockp is defined as

$$\mathsf{blockp}(p, N) = \{\, s \mid \exists n.p = \mathsf{VPtr}\ n\ 0 \land \mathsf{blocksize}(s, n) = N \,\}$$

and where $\mathsf{blocksize}(s, n)$ gives the size of logical block number n in state s.

Definition 3.2 (Permission ordering). Given permissions $\pi_1 = (R_1, G_1, P_1)$ and $\pi_2 = (R_2, G_2, P_2)$, we define the *permission ordering* $\pi_1 \leq \pi_2$ to hold if $R_1 \supseteq R_2$, $G_1 \subseteq G_2$, and $P_1 \supseteq P_2$.

Intuitively, a "bigger" permission allows more state updates than a smaller one, meaning its guarantee is bigger. It also has stronger restrictions on state updates that it tolerates, as well as a stronger precondition, captured by having a smaller rely and precondition. Note, for example, that $\mathsf{ptrp}_{\mathsf{W}}(x, v) \geq \mathsf{ptrp}_{\mathsf{R}}(x, v)$.

Definition 3.3 (Separate permissions). Given permissions $\pi_1 = (R_1, G_1, P_1)$ and $\pi_2 = (R_2, G_2, P_2)$, we say they are *separate*, written $\pi_1 \perp \pi_2$, iff $G_1 \subseteq R_2$ and $G_2 \subseteq R_1$.

Two permissions are separate if the updates of one are tolerated by the other. For example, we have that $\mathsf{ptrp}_{\mathsf{R}}(x, v)$ is separate from itself, while $\mathsf{ptrp}_{\mathsf{W}}(x, v)$ is not separate from itself or from $\mathsf{ptrp}_{\mathsf{R}}(x, v)$. Only permissions that are separate can coexist with each other, to ensure that program fragments do not step on each other's toes. For instance, each pointer can either have one write permission or multiple read permissions at any time, but not both.

We then use separateness to define separating conjunction:

Definition 3.4 (Separating conjunction). Given permissions $\pi_1 = (R_1, G_1, P_1)$ and $\pi_2 = (R_2, G_2, P_2)$, we define the *separating conjunction* $\pi_1 * \pi_2$ of π_1 and π_2 by:

$$\pi_1 * \pi_2 = (R_1 \cap R_2, (G_1 \cup G_2)^*, P)$$

where $P = P_1 \cap P_2$ if $\pi_1 \perp \pi_2$ and $\emptyset$ otherwise, and $(-)^*$ denotes reflexive-transitive closure.

Informally, $\pi_1 * \pi_2$ permits the use of both π_1 and π_2. The rely only tolerates updates allowed by both π_1 and π_2, while the guarantee allows any number of updates to be performed using either π_1 or π_2. The precondition requires the preconditions of both π_1 and π_2 to hold, while also requiring them to be separate. If they are not separate, then $\pi_1 * \pi_2$ has a false precondition, i.e., it is inconsistent. This makes $*$ a total function on permissions, so we do not require tedious separateness side conditions to be proved when using our logic, and this also allows us to more easily lift separating conjunction to permission sets, defined below.

A key issue with permissions as we have defined them so far is how to represent disjunction. Even though permissions do form a complete lattice, if we try to define, for example, the notion "x is either a valid write pointer or it is `NULL`" as the meet of a $\mathsf{ptrp}_{\mathsf{W}}$ permission and a predicate stating that x equals `NULL`, the guarantee is the intersection of the guarantees of the two permissions, meaning that the resulting permission does not permit updating the value pointed to by x in any case. Instead, we represent disjunctions as sets of permissions, which intuitively represent the set of disjunctive possibilities. To simplify some of the technical details, these sets are required to be upwards closed.

Definition 3.5 (Permission sets). A *permission set over S* is an upwards-closed set of permissions, that is, a subset $\Pi \subseteq \mathsf{perm}_S$ such that $\pi_1 \in \Pi$ and $\pi_1 \leq \pi_2$ implies $\pi_2 \in \Pi$. We write Perms_S for the permission sets over S. The *permission set ordering* $\Pi_1 \leq \Pi_2$ holds iff $\Pi_1 \supseteq \Pi_2$. We say that Π_1 *entails* Π_2, written $\Pi_1 \models \Pi_2$, iff $\Pi_2 \leq \Pi_1$.

The entailment $\Pi_1 \models \Pi_2$ says that Π_1 is "bigger" than Π_2, which intuitively means that holding permissions Π_1 entitles one to perform any action permitted by permissions Π_2. Note that this is dual to the usual ordering on propositions, where the "smaller" proposition entails the bigger one.

The permission sets form a complete lattice, where we write True and False for the least and greatest elements, respectively. Note that False is intuitively the inconsistent or contradictory permission set that entails all others, because it allows everything, as it is the greatest permission, while True is the vacuously true permission set that is entailed by all others, as it allows nothing. Again, note that this is dual from the standard order on propositions. We also define the meet $\sqcap$ that takes the greatest lower bound, i.e., the union, of a set P of permission sets. This intuitively forms the disjunction of all elements of P.

Finally, we lift the definition of $*$ to permission sets:

Definition 3.6 (Separating conjunction for permission sets). For permission sets Π_1 and Π_2, we define the *separating conjunction*

$$\Pi_1 * \Pi_2 = \{\, \pi \mid \exists \pi_1 \in \Pi_1, \exists \pi_2 \in \Pi_2, \pi_1 * \pi_2 \leq \pi \,\}$$

We can then prove the usual rules for separating conjunction (with respect to permission entailment) using these definitions: associativity, commutativity, monotonicity, and the fact that True is an identity for $*$. In the rest of the paper, we will use these rules implicitly when manipulating permission sets.

4 BISIMULATION AND PERMISSION TYPING

We now define the typing relation that relates imperative programs to their functional specifications. The core concept is *stuttering bisimulation up to errors on the right*, which relate pairs of computations and states, parameterized by input and output permissions. At a high level, this relation says the states satisfy the precondition of the input permission, and that the computations both respect the guarantee of the input permission and both behave similarly at their respective states. Furthermore, the output permission acts as a postcondition, giving the permissions that the program has once the computations have finished executing. The permissions are over pairs of states, which intuitively represents the notion that $\approx$ is operating over both computations at the same time. Throughout this section and the next, we assume state types S_i for the imperative implementation and S_s for its specification.

The following captures how permissions are allowed to change during execution:

Definition 4.1. Permission π_2 *preserves separability of* π_1, written $\pi_1 \rightsquigarrow \pi_2$, iff $\pi_1 \perp \pi$ implies $\pi_2 \perp \pi$ for all π.

The following lemma about $\rightsquigarrow$ show that it is quite similar to $\leq$, except that $\rightsquigarrow$ does not impose any ordering on the preconditions of permissions.

Lemma 4.2. *$\pi_1 \rightsquigarrow \pi_2$ iff $R_{\pi_1} \subseteq R_{\pi_2}$ and $G_{\pi_2} \subseteq G_{\pi_1}$.*

The use of $\rightsquigarrow$ encapsulates the idea that the preconditions, which capture what is currently true of the state, can change as the program executes. With this definition, we can define this notion of stuttering bisimulation formally:

Definition 4.3 (Stuttering bisimulation up to errors on the right). Let $s_i \in S_i$, $s_s \in S_s$, $t_i \in \mathsf{CompM}\ S_i\ R_i$, and $t_s \in \mathsf{CompM}\ S_s\ R_s$. Given a permission $\pi \in \mathsf{perm}_{S_i \times S_s}$ (the "input permission") and a function $F : (R_i \times R_s) \to \mathsf{Perms}_{S_i \times S_s}$ (the "output permission set", describing the permissions held once the computations terminate), we define *stuttering bisimulation up to errors on the right* as the biggest relation $\approx_{\pi,F}$ such that:

(1) $(\mathsf{return}\ r_1, s_i) \approx_{\pi,F} (\mathsf{return}\ r_2, s_s)$ for all $r_1 \in R_i$ and $r_2 \in R_s$ where $\pi \in F(r_1, r_2)$
(2) $(t_i, s_i) \approx_{\pi,F} (\mathsf{throw}, s_s)$
(3) $(t_i, s_i) \approx_{\pi,F} (t_s, s_s)$ where for any t_i', s_i' such that $(t_i, s_i) \longrightarrow (t_i', s_i')$, $(s_i, s_s) G_\pi (s_i', s_s)$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i', s_i') \approx_{\pi',F} (t_s, s_s)$.
(4) $(t_i, s_i) \approx_{\pi,F} (t_s, s_s)$ where for any t_s', s_s' such that $(t_s, s_s) \longrightarrow (t_s', s_s')$, $(s_i, s_s) G_\pi (s_i, s_s')$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i, s_i) \approx_{\pi',F} (t_s', s_s')$.
(5) if t_i and t_s start with the same type of event, then $(t_i, s_i) \approx_{\pi,F} (t_s, s_s)$ when
  - for any t_i', s_i' such that $(t_i, s_i) \longrightarrow (t_i', s_i')$, there exists t_s', s_s' such that $(t_s, s_s) \longrightarrow (t_s', s_s')$, where $(s_i, s_s) G_\pi (s_i', s_s')$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i', s_i') \approx_{\pi',F} (t_s', s_s')$.
  - for any t_s', s_s' such that $(t_s, s_s) \longrightarrow (t_s', s_s')$, there exists t_i', s_i' such that $(t_i, s_i) \longrightarrow (t_i', s_i')$, where $(s_i, s_s) G_\pi (s_i', s_s')$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i', s_i') \approx_{\pi',F} (t_s', s_s')$.

Furthermore, cases 3 and 4 cannot be applied in an infinite chain, to ensure diverging computations are not related to every program.[5]

Case 1 states that returns are related when the input permission is in the output permission set for the values that are returned, while case 2 states that an error on the right is related to any computation on the left. Cases 3 and 4 allow one of side of the bisimulation to take a step, as long as this step is allowed by the input permission and the resulting computation and state is still related to the other side by a new permission. The condition preventing infinite chains of these two cases ensures that an infinite path on one side always corresponds to an infinite path on the other; otherwise, for instance, an infinite loop on one side would be related to any program on the other. In our Coq formalization, this condition is enforced by a nested inductive-coinductive definition. Finally, case 5 allows both sides to take a step in lock-step with the same conditions on permissions as the previous cases for the left and right sides.

Using this definition, if we prove the specification has no errors, then neither does the implementation:

Theorem 4.4 (Error-Freedom). *If* $(t_i, s_i) \approx_{\pi,F} (t_s, s_s)$ *and* $(t_s, s_s) \not\rightarrow^* \mathsf{throw}$ *then* $(t_i, s_i) \not\rightarrow^* \mathsf{throw}$.

It is well-known that stuttering bisimulation preserves all temporal properties that do not use the next-time operator [Browne et al. 1988; de Nicola and Vaandrager 1990]. This ensures that our specifications satisfy, *e.g.*, the same safety, liveness, and fairness properties as our imperative programs, assuming the specifications do not have errors. One complication is that the input permission π and its precondition can change during execution of our programs and so cannot be used to reason about any but the initial states. However, if we pick a permission π' that is separate from π, its precondition is guaranteed to be invariant during execution, and so can be used to relate the intermediate states of the imperative program and its specification. Intuitively, π' is a permission that is being held by some hypothetical observer that is trying to prove temporal properties. Separateness from π means that updates that can be performed by the computations are tolerated by this observer in building proofs.

In more detail, consider the following fragment of CTL [Clarke and Emerson 1981; Clarke et al. 1999], where $\mathrm{St}(P)$ denotes a *state predicate* for predicate $P \subseteq S$ over values in state type S:

$$\phi ::= \mathrm{St}(P) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \mathsf{EF}\ \phi \mid \mathsf{EG}\ \phi \mid \mathsf{AF}\ \phi \mid \mathsf{AG}\ \phi$$

The *satisfaction relation* $t, s \models \phi$ between t of type $\mathrm{CompM}\ S\ R$, s of type S, and CTL formula ϕ over S is defined in the standard way, where, e.g., $t, s \models \mathrm{St}(P)$ iff $P(s)$ holds, $t, s \models \mathsf{EF}\ \phi$ holds iff there exists some t' and s' with $(t, s) \longrightarrow^* (t', s')$ such that $t', s' \models \phi$, etc.

[5] In the Coq development, this is implemented using a mixed inductive-coinductive definition.

Definition 4.5 (π-similarity). Let $\pi \in \mathsf{perm}_{S_i \times S_s}$, $P_i \subseteq S_i$, and $P_s \subseteq S_s$. P_i and P_s are *π-similar* iff, for all $s_i \in S_i$ and $s_s \in S_s$ such that $P_\pi(s_i, s_s)$, we have $P_i(s_i) \iff P_s(s_s)$. CTL formulas ϕ_1 and ϕ_2 are *π-similar* iff ϕ_2 can be obtained from ϕ_1 by replacing every subformula $\mathsf{St}(P_i)$ with some $\mathsf{St}(P_s)$ for P_i π-similar to P_s.

THEOREM 4.6 (SOUNDNESS). *If $(t_i, s_i) \approx_{\pi_1,F} (t_s, s_s)$, $(t_s, s_s) \not\mapsto^* \mathsf{throw}$, ϕ_1 and ϕ_2 are π_2-similar, and $P_{\pi_1 * \pi_2}(s_i, s_s)$, then $t_i, s_i \models \phi_1 \iff t_s, s_s \models \phi_2$. Note that $P_{\pi_1 * \pi_2}(s_i, s_s)$ implies $\pi_1 \perp \pi_2$.*

The typing relation generalizes over valid input states of both programs:

Definition 4.7 (Typing). Let $\Pi \in \mathsf{Perms}_{S_i \times S_s}$, $F : (R_i \times R_s) \rightarrow \mathsf{Perms}_{S_i \times S_s}$, $t_i \in \mathsf{CompM}\ S_i\ R_i$, and $t_s \in \mathsf{CompM}\ S_s\ R_s$. We say imperative program t_i is *well-typed* with input permissions Π, output permissions F, and functional specification t_s, written $\Pi \vdash t_i \rightsquigarrow t_s : F$, iff $\forall \pi \in \Pi, \forall s_i, s_s$ such that $P_\pi(s_i, s_s)$, $(t_i, s_i) \approx_{\pi,F} (t_s, s_s)$.

5 TYPING RULES FOR SPEC EXTRACTION

We now use the above definitions to define typing rules for extracting functional specifications from imperative programs. Since we present the type system semantically, several things differ from a standard syntactic presentation. Rather than quantifying over all valid substitutions of free variables, we mix this step into the presentation by using universal quantification to represent free variables. Type soundness is proven semantically [Milner 1978], where theorem 4.4 in conjunction with the definition of typing acts as our adequacy theorem, and each of the semantic typing rules in this section is proven as a lemma.

Definition 5.1 (Permission types). An *(A_i, A_s)-permission type* is a function from an implementation value of type A_i and a specification value of type A_s to a permission set in $\mathsf{Perms}_{S_i \times S_s}$.

We write $\mathsf{PType}(A_i, A_s)$ for the set of (A_i, A_s)-permission types. We also write $x_i : T \triangleright x_s$ for the application $T(x_i, x_s)$ of T to $x_i \in A_i$ and $x_s \in A_s$. Intuitively, $x_i : T \triangleright x_s$ represents the permission to represent imperative value x_i with specification value x_s. If, for example, x_s always represents x_i in all states, then $x_i : T \triangleright x_s$ is the vacuously true permission set True, while if it never does this returns the inconsistent permission set False. The permission $x_i : T \triangleright x_s$ can also come with the ability to modify some portion of the state; e.g., as we shall see in Section 5.4, pointer permissions allow the value that is pointed in the implementation state to to be modified.

The remainder of this section gives the Heapster typing rules, which include rules both for permission set entailment $\Pi_1 \models \Pi_2$ and for typing. The two relations are linked by the following rule, which (when read backwards) allows the input permission type to be weakened and the output permission type to be strengthened:

$$\frac{\Pi_1 \models \Pi_2 \quad \Pi_2 \vdash t_i \rightsquigarrow t_s : T_2 \quad \forall x_i, x_s, x_i : T_2 \triangleright x_s \models x_i : T_1 \triangleright x_s}{\Pi_1 \vdash t_i \rightsquigarrow t_s : T_1}\ \textsc{Weak}$$

5.1 Permission Type Connectives

In this section, we introduce some basic connectives for permission types. These are given in Figure 3, assuming $T \in \mathsf{PType}(A_i, A_s)$, $U \in \mathsf{PType}(B_i, B_s)$, and $V \in \mathsf{PType}(A_i, B_s)$ are permission types, noting that T and V have the same imperative type, and that $F : A \rightarrow \mathsf{PType}(A_i, A_s)$ is a function and Π a permission set. The first of these is the product permission type $T \otimes U$, that relates imperative value x_i to specification value x_s by relating their first projections with T and their second projections with U. The second is the sum permission type $T \oplus U$, that relates left-hand imperative values $\mathsf{inl}\ x_i$ to left-hand specification values $\mathsf{inl}\ x_s$ using T and relates right-hand imperative values $\mathsf{inr}\ x_i$ to right-hand specification values $\mathsf{inr}\ x_s$ using U. The third permission type is the separating conjunction permission type $T \star V$, that relates imperative value x_i to a pair of specification values by relating x_i to the first one with T and to the second one with V. The permission sets resulting from these two relations are conjoined together. The fourth permission type is the permission set conjunction type $T \oslash \Pi$, that relates imperative values to specification values using T but also conjoins permission set Π to the resulting permission set.

$$\begin{aligned}
x_i : (T \otimes U) \triangleright x_s &= (x_i.1 : T \triangleright x_s.1) * (x_i.2 : U \triangleright x_s.2) \\
\mathsf{inl}\ x_i : (T \oplus U) \triangleright \mathsf{inl}\ x_s &= x_i : T \triangleright x_s \\
\mathsf{inr}\ x_i : (T \oplus U) \triangleright \mathsf{inr}\ x_s &= x_i : U \triangleright x_s \\
x_i : (T \star V) \triangleright x_s &= (x_i : T \triangleright x_s.1) * (x_i : U \triangleright x_s.2) \\
x_i : (T \oslash \Pi) \triangleright x_s &= x_i : T \triangleright x_s * \Pi \\
x_i : \mathsf{True} \triangleright \mathsf{unit} &= \mathsf{True} \\
x_i : (T \vee V) \triangleright \mathsf{inl}\ x_s &= x_i : T \triangleright x_s \\
x_i : (T \vee V) \triangleright \mathsf{inr}\ x_s &= x_i : U \triangleright x_s \\
x_i : (\exists z : A.F\ z) \triangleright \{z, x_s\} &= x_i : F\ z \triangleright x_s
\end{aligned}$$

Fig. 3. Connectives for Permission Types

The next permission type is the vacuous True permission type, that relates any imperative value to the unit object and returns the True permission set. Note that using True for both the permission set and permission type is a slight abuse of notation, but it is always clear from context which is intended. The following permission type is the disjunctive permission type $T \vee V$, that either relates an imperative value x_i to a left-hand specification value $\mathsf{inl}\ x_s$ by relating x_i to x_s using T or relates x_i to a right-hand specification value $\mathsf{inr}\ x_s$ by relating x_i to x_s using V. Finally, $\exists z : A.F\ z$ is the existential permission type, that relates an imperative value x_i to a dependent pair $\{z, x_s\}$ by relating x_i to the second projection x_s of the specification value using the permission type $F\ z$ resulting from applying the body of the existential to the first projection z of the specification value.

The rules for these connectives are given in Figure 4. The ProdI and ProdE rules introduce and eliminate product types $T \otimes U$ by introducing and eliminating products on both sides. The SumI1 and SumI2 rules introduce sum types $T \oplus U$ by applying the same sum constructor on both sides, while the SumE rule performs sum elimination on both sides using case expressions. Note that the elimination rule is a typing rule and not an entailment rule, because it must insert case expressions around the entire imperative program and specification. The StarI and StarE rules introduce and eliminate separating conjunction types $T \star U$ by folding and unfolding the definition of that type, while PermsI and PermsE introduce and eliminate the permission set conjunction type $T \oslash \Pi$ by folding and unfolding that definition. The central use of the latter type $T \oslash \Pi$ is in the Frame rule, which conjoins the same permission set Π_2 onto the input and output permissions of any typing derivation. Following is the TrueI rule, which vacuously introduces the True type by relating any x_i to unit. Next are the disjunctive introduction rules OrI1 and OrI2 that introduce a disjunctive type $T \vee U$ from type T or U, respectively, by applying the appropriate constructor to the specification value, along with the disjunctive elimination rule OrE, that eliminates a disjunctive type $T \vee U$ by inserting a sum elimination into the specification value. Similarly to the SumE rule, OrE is a typing and not an entailment rule. Finally, rules ExI and ExE introduce and eliminate the existential type $\exists z : A.F\ z$ by constructing and eliminating, respectively, a dependent pair in the specification.

5.2 Equality Permissions

The equality permission type $\mathsf{eq}(y)$ is defined as follows:

$$x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} = \begin{cases} \mathsf{True} & \text{if } x_i = y \\ \mathsf{False} & \text{otherwise} \end{cases}$$

When $x_i = y$, permission type $\mathsf{eq}(y)$ is the same as permission type True, and is otherwise inconsistent, i.e., x_i is not related to anything. Figure 5 contains rules EqRefl, EqSym, EqTrans, and EqCtx for reflexivity, symmetry, transitivity, and contextual closure of equality permissions. It

$$\frac{}{x_i : T_1 \triangleright x_s * y_i : T_2 \triangleright y_s \models (x_i, y_i) : T_1 \otimes T_2 \triangleright (x_s, y_s)}\ \textsc{ProdI}$$

$$\frac{}{x_i : T_1 \otimes T_2 \triangleright x_s \models x_i.1 : T_1 \triangleright x_s.1 * x_i.2 : T_2 \triangleright x_s.2}\ \textsc{ProdE}$$

$$\frac{}{x_i : T_1 \triangleright x_s \models \mathsf{inl}\ x_i : T_1 \oplus T_2 \triangleright \mathsf{inl}\ x_s}\ \textsc{SumI1} \qquad \frac{}{x_i : T_2 \triangleright x_s \models \mathsf{inr}\ x_i : T_1 \oplus T_2 \triangleright \mathsf{inr}\ x_s}\ \textsc{SumI2}$$

$$\frac{\forall y_i, y_s, \Pi * y_i : T_1 \triangleright y_s \vdash t_{i1} \rightsquigarrow t_{s1} : U \quad \forall z_i, z_s, \Pi * z_i : T_2 \triangleright z_s \vdash t_{i2} \rightsquigarrow t_{s2} : U}{\Pi * x_i : T_1 \oplus T_2 \triangleright x_s \vdash \mathsf{case}\ x_i\ \mathsf{of}\ (\lambda y_i.\, t_{i1})\ (\lambda z_i.\, t_{i2}) \rightsquigarrow \mathsf{case}\ x_s\ \mathsf{of}\ (\lambda y_s.\, t_{s1})\ (\lambda z_s.\, t_{s2}) : U}\ \textsc{SumE}$$

$$\frac{}{x_i : T \triangleright x_s * x_i : U \triangleright y_s \models x_i : T \star U \triangleright (x_s, y_s)}\ \textsc{StarI} \qquad \frac{}{x_i : T \star U \triangleright x_s \models x_i : T \triangleright x_s.1 * x_i : U \triangleright x_s.2}\ \textsc{StarE}$$

$$\frac{}{(x_i : T \triangleright x_s) * \Pi \models x_i : T \oslash \Pi \triangleright x_s}\ \textsc{PermsI} \qquad \frac{}{x_i : T \oslash \Pi \triangleright x_s \models (x_i : T \triangleright x_s) * \Pi}\ \textsc{PermsE}$$

$$\frac{\Pi_1 \vdash t_i \rightsquigarrow t_s : T}{\Pi_1 * \Pi_2 \vdash t_i \rightsquigarrow t_s : T \oslash \Pi_2}\ \textsc{Frame} \qquad \frac{}{\Pi \models \Pi * x_i : \mathsf{True} \triangleright \mathsf{unit}}\ \textsc{TrueI}$$

$$\frac{}{x_i : T \triangleright x_s \models x_i : T \vee U \triangleright \mathsf{inl}\ x_s}\ \textsc{OrI1} \qquad \frac{}{x_i : U \triangleright x_s \models x_i : T \vee U \triangleright \mathsf{inr}\ x_s}\ \textsc{OrI2}$$

$$\frac{\forall z_1, \Pi * x_i : T_1 \triangleright z_1 \vdash t_i \rightsquigarrow t_{s1} : U \quad \forall z_1, \Pi * x_i : T_2 \triangleright z_2 \vdash t_i \rightsquigarrow t_{s2} : U}{\Pi * x_i : T_1 \vee T_2 \triangleright x_s \vdash t_i \rightsquigarrow \mathsf{case}\ x_s\ \mathsf{of}\ (\lambda z_1.\, t_{s1})\ (\lambda z_2.\, t_{s2}) : U}\ \textsc{OrE}$$

$$\frac{}{x_i : F\ y_s \triangleright x_s \models x_i : \exists z{:}A.F\ z \triangleright \{y_s, x_s\}}\ \textsc{ExI} \qquad \frac{}{x_i : \exists z{:}A.F\ z \triangleright x_s \models x_i : F\ x_s.1 \triangleright x_s.2}\ \textsc{ExE}$$

Fig. 4. Typing Rules for Permission Connectives

$$\frac{}{\Pi \models \Pi * x_i : \mathsf{eq}(x_i) \triangleright \mathsf{unit}}\ \textsc{EqRefl} \qquad \frac{}{x_i : \mathsf{eq}(y_i) \triangleright \mathsf{unit} * y_i : \mathsf{eq}(z_i) \triangleright \mathsf{unit} \models x_i : \mathsf{eq}(z_i) \triangleright \mathsf{unit}}\ \textsc{EqTrans}$$

$$\frac{}{x_i : \mathsf{eq}(y_i) \triangleright \mathsf{unit} \models y_i : \mathsf{eq}(x_i) \triangleright \mathsf{unit}}\ \textsc{EqSym} \qquad \frac{}{x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} \models x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} * x_i : \mathsf{eq}(y) \triangleright \mathsf{unit}}\ \textsc{EqDup}$$

$$\frac{}{x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} \models f\ x_i : \mathsf{eq}(f\ y) \triangleright \mathsf{unit}}\ \textsc{EqCtx} \qquad \frac{}{x_i : \mathsf{eq}(y_i) \triangleright \mathsf{unit} * y_i : T \triangleright y_s \models x_i : T \triangleright y_s}\ \textsc{Cast}$$

Fig. 5. Typing Rules Involving Equality

also contains rule EqDup, which allow equality permissions to be duplicated. Finally, it contains the Cast rule, that allows a permission type for imperative value y_i to be transferred to x_i when we know x_i equals y_i through an $\mathsf{eq}(y_i)$ permission type on x_i.

5.3 Instructions

Figure 6 shows how to type-check the combinators used to model most of the assembly-language instructions in the program. We defer the operations related to pointers and memory to Section 5.4. The rule in Figure 6, Ret, extracts a specification that returns x_s from one that returns x_i when these are related by the input permission. Next is the Bind rule, which extracts a specification for $t_i >>= f_i$ by extracting specifications for t_i and f_i, using the output permission type of the former as the input permission type for the latter, and binds the results together. To type the getNum operation, which is used to model numeric operations like arithmetic and comparison instructions, the GetNum rule requires that its argument be a numeric Val. This is specified with an equality permission saying that the argument equals VNum y_i for some y_i. The output permission then says that the return value equals y_i. The specification of getNum returns the trivial unit value, as equality permissions have no computational content in specifications.

Following are the If and Iter rules, that extract conditionals and iter loops from conditionals and iter loops. For the former, the rule requires the condition x_i in the imperative program to satisfy an equality permission relating it to some Boolean value y_s, which is used as the condition in the

$$\frac{}{x_i : T \triangleright x_s \vdash \mathsf{return}\ x_i \rightsquigarrow \mathsf{return}\ x_s : T}\ \textsc{Ret} \qquad \frac{\Pi \vdash t_i \rightsquigarrow t_s : T \quad \forall x_i, x_s, x_i : T \triangleright x_s \vdash f_i\ x_i \rightsquigarrow f_s\ x_s : U}{\Pi \vdash t_i \mathbin{>\!\!>\!=} f_i \rightsquigarrow t_s \mathbin{>\!\!>\!=} f_s : U}\ \textsc{Bind}$$

$$\frac{}{\begin{array}{l} x_i : \mathsf{eq}(\mathsf{VNum}\ y_i) \triangleright \mathsf{unit} \\ \quad \vdash \mathsf{getNum}\ x_i \rightsquigarrow \mathsf{return\ unit} : \mathsf{eq}(y_i) \end{array}}\ \textsc{GetNum} \qquad \frac{\forall y_i, y_s, y_i : T \triangleright y_s \vdash f_i\ y_i \rightsquigarrow f_s\ y_s : T \oplus U}{x_i : T \triangleright x_s \vdash \mathsf{iter}\ f_i\ x_i \rightsquigarrow \mathsf{iter}\ f_s\ x_s : U}\ \textsc{Iter}$$

$$\frac{\Pi \vdash t_{i1} \rightsquigarrow t_{s1} : U \quad \Pi \vdash t_{i2} \rightsquigarrow t_{s2} : U}{\Pi * x_i : \mathsf{eq}(y_s) \triangleright \mathsf{unit} \vdash \mathsf{if}\ x_i\ \mathsf{then}\ t_{i1}\ \mathsf{else}\ t_{i2} \rightsquigarrow \mathsf{if}\ y_s\ \mathsf{then}\ t_{s1}\ \mathsf{else}\ t_{s2} : U}\ \textsc{If} \qquad \frac{}{P \vdash t_i \rightsquigarrow \mathsf{throw} : U}\ \textsc{Err}$$

Fig. 6. Typing Rules for Instructions

specification. The "then" and "else" branches for the specification are then obtained by extracting specifications from those of the imperative program. For the latter rule, the function argument f_i of iter must extract to a function f_s in the specification, with input permission type T relating their inputs and output permission type $T \oplus U$ relating their outputs. The second arguments of iter in the implementation and specification must then be related via T. Finally, the Err rule can extract an error specification from any imperative program if we cannot type-check it some other way.

5.4 Pointer Permissions

Pointer permissions are a key part of how Heapster erases pointers from imperative programs to turn them into purely functional specifications. At a high level, they represent pointers with the same specification value as the imperative value being pointed to. Thus, the pointers themselves become transparent in terms of how they are represented.

In more detail, let valAdd : Val → BV 64 → Val be the operation that adds a numeric offset to either the numeric value or pointer offset of an imperative value. Further, let rw be a *read-write modality*, which is one of the two objects R and W, and let $o \in \mathsf{BV}\ 64$ and $T \in \mathsf{PType}(\mathsf{Val}, A_s)$ for some type A_s. We then define the pointer permission type $\mathsf{ptr}((rw, o) \mapsto T)$ as follows, recalling the definitions of the individual pointer permissions ptrp_R and ptrp_W from Section 3:

$$x_i : (\mathsf{ptr}((rw, o) \mapsto T)) \triangleright x_s = \{\, \pi \mid \exists v. \pi \geq \mathsf{ptrp}_{rw}(\mathsf{valAdd}\ x_i\ o, v) * v : T \triangleright x_s \,\}$$

We call T the *content type* of the pointer permission type $\mathsf{ptr}((rw, o) \mapsto T)$.

The rules for pointer permissions and the instructions related to pointers are shown in figure 7. The PtrI and PtrE rules are similar to the rules ExI and ExE for existential types, where PtrI introduces a pointer permission with content type T from one of the form $\mathsf{ptr}((rw, o) \mapsto \mathsf{eq}(y_i))$ and PtrE eliminates a pointer permission to get out a Val y_i and a permission type $\mathsf{ptr}((rw, o) \mapsto \mathsf{eq}(y_i))$. Not only does $\mathsf{ptr}((rw, o) \mapsto \mathsf{eq}(y_i))$ state that x_i currently points to y_i, it also represents a form of permission borrowing, similar in spirit to the borrowing of the Rust type system. It says that y_i holds the permissions that were originally part of those held by x_i, and can be given back by the PtrI rule. The ReadDup rule allows a read pointer permission to be duplicated, though its contents must be an equality permission so that it can be duplicated as well. Rule PtrOff changes a pointer permission on x_i to a pointer permission on the addition valAdd x_i o_2 of an offset to x_i by subtracting the offset o_2 from that of the pointer permission. The Load rule requires x_i to point to a known value y_i, which can be achieved by first applying PtrE to a (read or write) pointer permission. The permission for the returned value is then $\mathsf{eq}(y_i)$. The Store rule requires a write pointer permission, whose content permission is changed to an equality to the value that is written. Because pointers are transparent objects that do not appear in specifications, the specifications returned by both the Load and Store rules just return the unit value; their main effect is on the pointer permissions in their inputs and outputs. The IsNull1 and IsNull2 rules then type the

$$\frac{}{x_i : \mathsf{ptr}((rw,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} * y_i : T \triangleright y_s \models x_i : \mathsf{ptr}((rw,o) \mapsto T) \triangleright y_s}\ \textsc{PtrI}$$

$$\frac{\forall y_i, \Pi * x_i : \mathsf{ptr}((rw,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} * y_i : T \triangleright x_s \vdash t_i \rightsquigarrow t_s : U}{\Pi * x_i : \mathsf{ptr}((rw,o) \mapsto T) \triangleright x_s \vdash t_i \rightsquigarrow t_s : U}\ \textsc{PtrE}$$

$$\frac{}{x_i : \mathsf{ptr}((R,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} \models \begin{array}{l} x_i : \mathsf{ptr}((R,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} * \\ x_i : \mathsf{ptr}((R,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} \end{array}}\ \textsc{ReadDup}$$

$$\frac{}{x_i : \mathsf{ptr}((rw,o_1) \mapsto T) \triangleright x_s \models \mathsf{valAdd}\ x_i\ o_2 : \mathsf{ptr}((rw,o_1 - o_2) \mapsto T) \triangleright x_s}\ \textsc{PtrOff}$$

$$\frac{}{x_i : \mathsf{ptr}((rw,0) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} \vdash \mathsf{load}\ x_i \rightsquigarrow \mathsf{return\ unit} : \mathsf{eq}(y_i) \oslash x_i : \mathsf{ptr}((rw,0) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit}}\ \textsc{Load}$$

$$\frac{}{x_i : \mathsf{ptr}((W,0) \mapsto T)) \triangleright x_s \vdash \mathsf{store}\ x_i\ y_i \rightsquigarrow \mathsf{return\ unit} : \mathsf{True} \oslash x_i : \mathsf{ptr}((W,0) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit}}\ \textsc{Store}$$

$$\frac{}{x_i : \mathsf{ptr}((rw,o) \mapsto T) \triangleright x_s \vdash \mathsf{isNull}\ x_i \rightsquigarrow \mathsf{return\ unit} : \mathsf{eq}(\mathsf{false}) \oslash x_i : \mathsf{ptr}((rw,o) \mapsto T) \triangleright x_s}\ \textsc{IsNull1}$$

$$\frac{}{x_i : \mathsf{eq}(0) \triangleright x_s \vdash \mathsf{isNull}\ x_i \rightsquigarrow \mathsf{return\ unit} : \mathsf{eq}(\mathsf{true})}\ \textsc{IsNull2}$$

Fig. 7. Typing Rules for Pointer Permissions

isNull operator in two cases, one where the argument is a pointer, in which case the output type says it returns true, and one where the argument equals 0, where output type says it returns false.

5.5 Array Permissions

Array pointer permissions are defined as repeated pointer permissions, one for each valid offset into the array. Let rw be a read-write modality, let $o, l \in \mathsf{BV}\ 64$, and let $T \in \mathsf{PType}(\mathsf{Val}, A_s)$. The *array pointer permission type* $\mathsf{arr}((rw,o,l) \mapsto T)$ is the $(\mathsf{Val}, \mathsf{Vect}\ A_s\ l)$-permission type such that

$$x_i : \mathsf{arr}((rw,o,l) \mapsto T) \triangleright x_s = \mathop{\ast}_{0 \le i < l} x_i : \mathsf{ptr}((rw, o + 8 * i) \mapsto T) \triangleright x_s[i]$$

That is, $\mathsf{arr}((rw,o,l) \mapsto T)$ is defined as the permissions for l pointers to the 8-byte words starting at offset o. The specification value x_s is a vector of l values, where the ith element acts as the specification value for the ith pointer permission. The Heapster tool in fact extends array permissions and their rules in a number of ways, such as to support arrays of different cell sizes, but these extensions are beyond the scope of this paper.

Also relevant to array permissions are `malloc` and `free`, which in general (de)allocate arrays of arbitrary size. The *allocation permission set* alloc is defined as the upwards closure of the set of all permissions $\mathsf{alloc}(n)$, defined as

$$\begin{array}{l} \mathsf{alloc}(n) = (\{\, (s_1,s_2) \mid \forall m \ge n.\mathsf{blkeq}(s_1,s_2,m) \,\},\ \ \{\, (s_1,s_2) \mid \forall m < n.\mathsf{blkeq}(s_1,s_2,m) \,\} \\ \qquad\qquad\quad \{\, s \mid \mathsf{numblocks}(s) = n \,\}) \end{array}$$

where $\mathsf{blkeq}(s_1,s_2,m)$ requires the mth memory block to have the same size and values in both s_1 and s_2 and where $\mathsf{numblocks}(s)$ is the number of memory blocks that have been allocated so far. This permission says that n blocks have been allocated so far, and gives exclusive permission to all blocks that have not yet been allocated. The *block permission type* $\mathsf{block}(N)$ is defined as the $(\mathsf{Val}, \mathsf{Unit})$-permission type such that

$$x_i : (\mathsf{block}(N)) \triangleright \mathsf{unit} = \mathsf{pred}(\mathsf{blockp}(p, N))$$

where $\mathsf{blockp}(p, N)$ is defined in Section 3 as the predicate stating that p points to the beginning of a block of size N.

$$\frac{\forall x_{s1}.x_{s2}.\Pi * x_i : \mathsf{arr}((rw, o, l') \mapsto T) \triangleright x_{s1} * x_i : \mathsf{arr}((rw, o + l', l - l') \mapsto T) \triangleright x_{s2} \vdash t_i \rightsquigarrow f_s(x_{s1}, x_{s2}) : U}{\Pi * x_i : \mathsf{arr}((rw, o, l) \mapsto T) \triangleright x_s \vdash t_i \rightsquigarrow \mathsf{trySplit}\ x_s\ l'\ f_s : U}\ \textsc{ArrSplit}$$

$$\frac{}{x_i : \mathsf{arr}((rw,o,l') \mapsto T) \triangleright x_{s1} * x_i : \mathsf{arr}((rw,o + l',l) \mapsto T) \triangleright x_{s2} \models x_i : \mathsf{arr}((rw,o,l + l') \mapsto T) \triangleright x_{s1} +\!\!+ x_{s2}}\ \textsc{ArrCombine}$$

$$\frac{}{x_i : \mathsf{arr}((rw, o, 1) \mapsto T) \triangleright x_s \models x_i : \mathsf{ptr}((rw, o) \mapsto T) \triangleright x_s[0]}\ \textsc{ArrPtr}$$

$$\frac{}{x_i : \mathsf{ptr}((rw, o) \mapsto T) \triangleright x_s \models x_i : \mathsf{arr}((rw, o, 1) \mapsto T) \triangleright \langle x_s \rangle}\ \textsc{PtrArr}$$

$$\frac{}{x_i : \mathsf{eq}(\mathsf{VNum}(8 \times l)) \triangleright x_s * \mathsf{alloc} \vdash \mathsf{malloc}\ x_i \rightsquigarrow \mathsf{return}\ \langle \mathsf{unit}, \dots \rangle : \mathsf{arr}((\mathsf{W},0,l) \mapsto \mathsf{True}) \star \mathsf{block}(8 \times l) \oslash \mathsf{alloc}}\ \textsc{Malloc}$$

$$\frac{}{x_i : \mathsf{arr}((\mathsf{W}, 0, l) \mapsto \mathsf{True}) \star \mathsf{block}(8 * l) \triangleright x_s \vdash \mathsf{free}\ x_i \rightsquigarrow \mathsf{return\ unit} : \mathsf{True}}\ \textsc{Free}$$

Fig. 8. Typing Rules for Array Permissions

The typing rules for arrays are shown in Figure 8. The first rule, ArrSplit, splits an array permission $\mathsf{arr}((rw, o, l) \mapsto T)$ in two at offset l', leaving an array permission of length l' and one of length $l - l'$ starting at offset $o + l'$. The specification uses the function

$$\mathsf{trySplit} : \mathsf{Vect}\ A\ l \to \Pi l' : \mathsf{BV}\ 64.\ (\mathsf{Vect}\ A\ l' \to \mathsf{Vect}\ A\ (l - l') \to \mathsf{CompM}\ S\ R) \to \mathsf{CompM}\ S\ R$$

where $\mathsf{trySplit}\ v\ l'\ f$ tries to split a vector v of length l into one of length l' and one of length $l - l'$, passing the results to f on success and throwing an error on failure. Note that, other than the Err rule, this is the only rule that introduces errors into the specification. The ArrCombine rule performs the converse, combining two consecutive array permissions into one. Its specification appends the two corresponding specification vectors with the vector append operation $+\!\!+$. The ArrPtr rule converts an array permission with length 1 into the equivalent pointer permission, using a specification that extracts the single element of the corresponding vector. The PtrArr does the opposite, building an array permission of length 1 from a pointer permission. To type-check an array indexing operation at index i, ArrSplit can be applied twice to split an array permission into the portions of the array before i, after i, and containing only i. ArrPtr can then be applied to convert this latter portion into a pointer permission to be used with load or store.

5.6 Recursive and Reachability Permissions

Recursive permission types allow Heapster to represent recursive data structures like the `bufs` structure from Section 1.1. Reachability permissions are a specific form of recursive permission that intuitively states that an implementation value y_i is reachable via 0 or more pointer steps from the value x_i holding that permission, that satisfy reflexivity and transitivity rules. The corresponding specification is a list of the specifications of all of the steps along the path from x_i to y_i.

Let G be a type-level function with fixed-point X, equipped with functions $\mathsf{fold} : G\ X \to X$ and $\mathsf{unfold} : X \to G\ X$ that form an isomorphism, and let $F : \mathsf{PType}(A, X) \to \mathsf{PType}(A, G\ X)$ be a function on permission types that is monotonic with respect to the permission ordering $\leq$. Further, define function $F' : \mathsf{PType}(A, X) \to \mathsf{PType}(A, X)$ such that $x_i : (F'\ T) \triangleright x_s$ equals the permission set $x_i : (F\ T) \triangleright \mathsf{unfold}\ x_s$. We then define the *least fixed-point permission* μF as the least fixed-point of F' with respect to $\leq$. We sometimes write $\mu X.T$ with X free in T for the least fixed-point permission $\mu(\lambda X.\ T)$. A *reachability permission* is a $(\mathsf{Val}, \mathsf{List}\ A_s)$-permission type $(\mu X.\mathsf{eq}(y) \vee (T \star \mathsf{ptr}((rw, o) \mapsto X)))$ for $T \in \mathsf{PType}(\mathsf{Val}, A_s)$ that does not contain y or X free.

Figure 9 shows the typing rules related to recursive permissions. The Fold maps from permission type $F\ (\mu F)$ by applying the fold function in the specification, while the Unfold does the reverse by applying unfold. The ReflR rule proves reflexivity of reachability permissions, intuitively stating that x_i is always reachable from itself. The resulting specification is the empty list, because x_i

$$\frac{}{x_i : F\ (\mu F) \triangleright x_s \models x_i : \mu F \triangleright \text{fold } x_s}\ \textsc{Fold} \qquad \frac{}{x_i : \mu F \triangleright x_s \models x_i : F\ (\mu F) \triangleright \text{unfold } x_s}\ \textsc{Unfold}$$

$$\frac{}{\text{True} \models x_i : (\mu X.\text{eq}(x_i) \lor T \star \text{ptr}((rw, o) \mapsto X)) \triangleright [\,]}\ \textsc{ReflR}$$

$$\frac{}{\begin{array}{r} x_i : (\mu X.\text{eq}(y_i) \lor T \star \text{ptr}((rw, o) \mapsto X)) \triangleright x_s * y_i : (\mu X.\text{eq}(z_i) \lor T \star \text{ptr}((rw, o) \mapsto X)) \triangleright y_s \\ \models x_i : (\mu X.\text{eq}(z_i) \lor T \star \text{ptr}((rw, o) \mapsto X)) \triangleright \text{append } x_s\ y_s \end{array}}\ \textsc{TransR}$$

Fig. 9. Typing Rules for Recursive and Reachability Permissions

reaches itself in 0 steps. The TransR rule proves transitivity, stating that if x_i reaches y_i and y_i reaches z_i then x_i reaches z_i. The specification appends the two lists.

6 THE HEAPSTER TOOL

The Heapster tool implements the type-checking and specification extraction process described in the previous sections. This section describes usage of the tool from a user's perspective.

The user starts with their imperative code as usual, written in any language that compiles to LLVM bitcode. Heapster can be used to verify each function in the input program. The user must write Heapster types—giving the permissions that are held before and after the function call—for each function by hand and pass this to the tool along with the code. Given these annotations, the Heapster tool uses a straightforward algorithm, not described here for space reasons, to apply the type-checking rules presented in this paper.

Some functions, such as those using loops, cannot be type-checked automatically, and require the user to provide type-checking hints to Heapster. These hints are intermediate Heapster types that hold at certain points in the function, acting as loop invariants.

Specifically, most iterative loops over recursive data structures require hints from the user to manually generalize the permission describing the data structure to a reachability permission. We anticipate that it is possible to do this generalization automatically using a widening algorithm, but we leave this to future work. Recursive loops, which the tool supports even though the formalization does not, do not require hints, however. This is because the permission types on the function itself, which are supplied by the user, also serve as the invariant for the recursive loop performed by the function.

Once type-checking succeeds, Heapster outputs a functional specification in Coq. The user can then prove arbitrary properties about this functional program, such as functional correctness or absence of errors. In particular, the absence of errors in the functional program guarantees memory safety for the imperative program.

For each of examples we present below, we first type-check the code using Heapster types. Since the Err rule can type-check any imperative program, we take successful type-checking to mean that the program can be type-checked without use of the Err rule. In some cases, such as when the error computation only appears in unreachable branches of an if statement, we loosen this definition and consider that successful as well. Note that this is an informal measure; it is not necessary to disallow uses of Err, but its use in the type-checking process usually indicates that something has gone wrong.

We use an approach based on Dijkstra monads [Maillard et al. 2019; Silver and Zdancewic 2021] to prove two results about each extracted specification. First, that error is unreachable by proving that the specification refines a returning (and therefore non-error) computation, and second, that the specification refines a hand-written, higher-level specification of the program. We have developed an automated tactic for performing these refinement proofs, and have had some success in using it. The main limitation in using this tactic is performance: its implementation is based on Coq

```
void xor_swap(uint64_t *x, uint64_t *y) {
  *x = *x ^ *y;
  *y = *x ^ *y;
  *x = *x ^ *y;
}
```

```
int64_t is_elem (int64_t x, list64_t *l) {
  if (l == NULL)
    return 0;
  else if (l→ data == x)
    return 1;
  else
    return is_elem (x,  l→ next);
}
```

Fig. 10. C Examples to be Type-checked with Heapster

typeclasses, which can become slow for large terms. We are currently reimplementing this tactic to overcome these performance limitations by using computational reflection [Boutin 1997; Malecha et al. 2014].

As a first example consider the swap function xor_swap implemented using an exclusive or operation in Figure 10. We type-check this function using the following Heapster command, which says that the arguments are pointers to 64-bit bitvectors both before and after the function call (The "-o" is used to separate the input and output permissions of the function):

```
heapster_typecheck_fun env "xor_swap" "(x:bv 64, y:bv 64).
arg0: ptr((W,0) |-> eq(llvmword(x))), arg1: ptr((W,0) |-> eq(llvmword(y))) -o
arg0: ptr((W,0) |-> exists z:bv 64.eq(llvmword(z))),
arg1: ptr((W,0) |-> exists z:bv 64.eq(llvmword(z))), ret:true";
```

Heapster extracts the following functional specification when it finishes type-checking:

```
fixM
 (fun _ :  bitvector 64 → bitvector 64 → {_ : bitvector 64 & unit} * ({ _ :  bitvector 64 & unit} * unit) ⇒
  fun e0 e1 :  bitvector 64 ⇒
   letRecM tt
     (returnM (existT (fun _ :  bitvector 64 ⇒ unit)
                (SAWCorePrelude.bvXor 64 (SAWCorePrelude.bvXor 64 e0 e1)
                  (SAWCorePrelude.bvXor 64 (SAWCorePrelude.bvXor 64 e0 e1) e1))  tt,
              (existT (fun _ :  bitvector 64 ⇒ unit)
                (SAWCorePrelude.bvXor 64 (SAWCorePrelude.bvXor 64 e0 e1) e1) tt,  tt)))
```

Note that unlike the idealized specification shown in figure 2, this specification is far more verbose, and uses different operations such as fixM for general recursion instead of iter. Future work includes updating the Heapster tool to use less general operations such as iter when general recursion is not required, as well as expanding the formalization to include recursion instead of just iteration.

Finally, we use the automated Coq tactic to prove that this specification refines the error-free program and that it refines the following hand-written specification:

```
Definition xor_swap_spec x1 x2 :
  CompM ({ _ :  bitvector 64 & unit} * ({ _ :  bitvector 64 & unit} * unit))  :=
  returnM (existT _ x2 tt,  (( existT _ x1 tt),   tt)).
```

The proof of the second refinement requires some facts about exclusive or which had to be proved manually.

For a second, more complex example, consider the following function which searches a linked list for a value: We type-check the function using the following Heapster command:

```
heapster_typecheck_fun env "is_elem" "(x:bv 64).
```

```
arg0:eq(llvmword(x)), arg1:List<(exists y:(bv 64).eq(llvmword(y))),always,R> -o
arg0:true, arg1:true, ret:exists z:(bv 64).eq(llvmword(z))";
```

This Heapster type says that the arguments are a 64-bit bitvector and a linked list of 64-bit bitvectors, and that the return value is also a 64-bit bitvector. Note that here we drop the output types for the arguments because the input types are non-exclusive and can be copied as needed by the caller to preserve them on output.

The list permission type used earlier must first be defined by the user using the following command:

```
heapster_define_recursive_perm env "List"
"X:perm(llvmptr 64), l:lifetime, rw:rwmodality" "llvmptr 64"
["eq(llvmword(0))","[l]ptr((rw,0) |-> X) * [l]ptr((rw,8) |-> List<X,l,rw>)"]
"List_def" "foldList" "unfoldList";
```

This command defines a list as a recursive permission type which is either NULL or is a pointer permission to offset 0 with some content type, and another pointer permission to offset 8 which has the same recursive list type as its content type. The type is parameterized by the content type, a lifetime (which is currently unused), and a read-write modality. The last line of the command connects the Heapster type to a definition of lists in Coq.

The extracted specification is:

```
fixM (fun is_elem : bitvector 64 → List {_ : bitvector 64 & unit} → {_ : bitvector 64 & unit} ⇒
   (fun (e0 : bitvector 64) (p0 : List {_ : bitvector 64 & unit}) ⇒
    letRecM tt
      either unit ({_ : bitvector 64 & unit} ∗ (List_def {_ : bitvector 64 & unit} ∗ unit))
         (CompM {_ : bitvector 64 & unit})
         (fun _ : unit ⇒ returnM (existT (fun _ : bitvector 64 ⇒ unit) (intToBv 64 0) tt))
         (fun x_right0 : {_ : bitvector 64 & unit} ∗ (List_def {_ : bitvector 64 & unit} ∗ unit) ⇒
          if not (bvEq 1 (if bvEq 64 (projT1 (fst x_right0)) e0 then intToBv 1 (−1) else intToBv 1 0)
                 (intToBv 1 0))
          then returnM (existT (fun _ : bitvector 64 ⇒ unit) (intToBv 64 1) tt)
          else is_elem e0 (fst (snd x_right0)) >>=
               (fun call_ret_val : {_ : bitvector 64 & unit} ⇒
               returnM (existT (fun _ : bitvector 64 ⇒ unit) (projT1 call_ret_val) tt)))
         (unfoldList {_ : bitvector 64 & unit} p0)))
```

The automated tactic successfully proves that this specification refines the error-free specification and that it refines the following hand-written specification:

```
Definition is_elem_fun (x:bitvector 64) :
  list {_:bitvector 64 & unit} → CompM {_:bitvector 64 & unit} :=
  list_rect (fun _ ⇒ CompM {_:bitvector 64 & unit})
            (returnM (existT _ (intToBv 64 0) tt))
            (fun y l' rec ⇒
               if bvEq 64 (projT1 y) x then returnM (existT _ (intToBv 64 1) tt) else rec).
```

Users can prove arbitrary properties of the extracted specifications beyond these two approaches. For instance, we have also verified the correctness property that the specification returns 1 when the specified bitvector is in the list and 0 when it is not. However, this proof requires some amount of manual effort on top of the automation to prove some of the subgoals.

As shown in these examples, the amount of input the user has to provide to obtain the functional specification is relatively small. The Heapster types are straightforward and just include type information, not behavioral information, and providing loop invariants is par for the course.

Compared to other systems, such as VST [Appel et al. 2014; Beringer and Appel 2019] and Iris [Jung et al. 2018, 2015], where one can do general verification of programs, our approach greatly simplifies writing specifications for imperative code. In those systems, the specifications must be written by hand whereas in Heapster, they are automatically extracted using the typing information. Note that the specifications generated by Heapster seem to have comparable size and complexity to those used in systems like VST and Iris, though it is hard to form a direct "apples-to-apples" comparison thanks to differences in the logics and other formalisms.

7 RESULTS

We are using the Heapster tool to verify an implementation of the Encapsulating Security Payload (ESP) protocol of IPSec. In this section, we report on our experience verifying the functions for manipulating a data structure that is central to this implementation. The data structure is called an mbox, short for "memory box". It is a version of the mbuf structure [Jeker 2008] used by the OpenBSD network stack, and represents a network packet as a linked list of chunks, or "boxes". These linked lists are also called mbox *chains*. This representation allows additional headers and footers to be added to packets without copying.

The mbox data structure is defined as follows:

```
struct mbox { size_t start; size_t len; struct mbox *next; uint8_t data[128]; };
```

Each mbox structure represents a chunk of memory. It contains a fixed-size array data of 128 bytes, along with fields start and len that indicate what portion of the data array makes up the current chunk. Thus, for instance, bytes can be removed from the beginning or end of a chunk just by modifying start or len. The next field points to the next chunk in the current packet.

Figure 11 lists the functions on mbox structures, along with a description of each function, the number of iterative loops in each function, whether the function has successfully been type-checked with Heapster to date, and, if so, the number of user-supplied hints needed to type-check the function. The "TC" column indicates functions that have been type-checked successfully by the criteria specified in section 6. The "V" column indicates functions whose extracted specifications have been verified to be error-free and to refine a higher-level hand-written specification, as described in section 6. As shown in the figure, each function with an iterative loop requires a hint to supply its loop invariant.

One function that could not be type-checked is mbox_increment, because this function depends on the endianness of the architecture, which is not yet supported by Heapster. The remaining functions that cannot yet be type-checked with Heapster are mbox_new, mbox_free, and mbox_all_freed, which are part of the custom allocator and deallocator for mbox. Just as the standard malloc and free require special-purpose typing rules, so too do custom allocators and deallocators, at least at the current level of development of the Heapster tool. To support functions that require their own custom rules, Heapster allows the user to posit function types and translations as type-checking axioms. Type-checking axioms are also useful for handling opaque, "black-box" operations like system or library calls whose semantics are not available to the Heapster tool. For these mbox functions, only the rand and memcpy functions had to be axiomatized. In the future we plan to generate Coq proof goals for type-checking axioms, to allow users to verify that their axioms are correct, though this functionality has not yet been implemented. Figure 11 indicates which functions rely on type-checking axioms.

Figure 11 shows that we have had success verifying four of the mbox functions using our automated Coq tactic. The main barrier to verifying more of the functions is performance of the tactic, which we are working to address.

Function Name	**Description**	**#Loops**	**#Hints**	**TC**	**V**
`mbox_new`	Allocate an `mbox`	1	N/A		
`mbox_free`	Deallocate an `mbox`	0	N/A		
`mbox_free_chain`	Deallocate an `mbox` chain	1	1	✓†	✓
`mbox_from_buffer`	Allocate an `mbox` chain from a buffer	1	1	✓†	
`mbox_to_buffer`	Copy the contents of an `mbox` chain to a buffer	1	1	✓†	
`mbox_len`	Compute the length in bytes of an `mbox` chain	1	1	✓	
`mbox_concat`	Concatenate an `mbox` chain after a single `mbox`	0	0	✓	✓
`mbox_concat_chains`	Concatenate two `mbox` chains	1	1	✓	
`mbox_split_at`	Split an `mbox` chain into two chains	0^*	0	✓†	
`mbox_copy`	Copy a single `mbox`	0	0	✓†	✓
`mbox_copy_chain`	Copy an `mbox` chain	0^*	0	✓	
`mbox_detach`	Detach the first `mbox` from a chain	0	0	✓	✓
`mbox_detach_from_end`	Detach the first N bytes of from an `mbox` chain	0	0	✓	
`mbox_increment`	Increment the first 128 bits of an `mbox` as a bitvector	0	N/A		
`mbox_randomize`	Randomize the contents of an `mbox`	1	1	✓†	
`mbox_eq`	Test if the contents of two `mbox` chains are equal	1	1	✓	
`mbox_drop`	Remove bytes from the start of an `mbox`	0^*	0	✓	
`mbox_all_freed`	Test that all `mbox` structures have been freed	1	N/A		

* the function is recursive

† the function calls a non-type-checked function

Fig. 11. The Results of Heapster Specification Extraction on `mbox` Functions

Even without verification of the extracted specifications, though, just the process of type-checking helped us to uncover a number of bugs related to memory safety. The first of these was a bug in `mbox_split_at`, where the splitting was being done incorrectly when the index for splitting landed exactly on the boundary between two `mbox` chunks. The second was a bug in `mbox_to_buffer` where array size was being checked incorrectly. Finally, there were a number of missing `NULL` checks.

8 RELATED WORK

Closely related to Heapster is the Electrolysis system [Ullrich 2016], which extracts functional specifications in the Lean theorem prover from imperative programs written in Rust. Immutable references are erased to pure functional values, just as in Heapster, while mutable references are translated to lenses that describe them as a substructure of some local variable in scope. This approach does have some limitations: it cannot represent the arbitrary dynamic allocation of `malloc`, and it is difficult for it to handle more complex patterns like mutable references nested in larger structures or modified in loops. However, the authors have shown it to be effective enough to translate 45% of the Rust `core` library to Lean.

RustHorn [Matsushita et al. 2020] uses the ownership information of Rust types to translate Rust code to constrained Horn clauses. Heapster supports a more general class of source programs at the cost of more user input, but uses the same idea of using ownership information to extract simpler representations of source programs.

As a separation logic, Heapster is closest to Viper [Müller et al. 2016], including its recent Prusti front-end for Rust [Astrauskas et al. 2019]. Viper is a system for verification of imperative programs using separation logic. It supports a high degree of automation in this verification, just like Heapster, and it also allows users to write functional specifications for their imperative programs. A key difference, however, is that users of Viper have to encode these specifications manually as pre- and post-conditions of functions, whereas Heapster extracts an entire functional specification automatically from just the memory-safety types of a function. Viper also uses the implicit dynamic frames formulation of separation logic [Smans et al. 2012], which contains assertions like "l is a valid pointer", similar to our $\mathrm{ptr}((rw, o) \mapsto T)$ permissions, instead of the standard separation logic assertions $l \mapsto x$ that location l currently points to x. The implicit dynamic frames logic is in fact known to be equivalent to the existential and disjunctive fragment of standard separation logic [Parkinson and Summers 2011].

There are also a number of other tools for verifying imperative programs in Coq using separation logic, including Iris [Jung et al. 2018, 2015] and the Verified Software Toolchain [Appel et al. 2014; Beringer and Appel 2019]. These tools use powerful higher-order concurrent separation logics that can support reasoning over complex concurrent data structures. For instance, Iris was recently used to verify a distributed database [Gondelman et al. 2021]. With this power comes a high degree of complexity, however, and these tools often require a high degree of time and expertise to effectively use them.

There have been a number of concurrent separation logics that incorporate rely-guarantee reasoning [Dodds et al. 2009; Feng 2009; Vafeiadis and Parkinson 2007]. There are even type systems that incorporate rely-guarantee [Gordon et al. 2013]. There are also a number of meta-theoretic frameworks for defining the semantics of separation logics that can support rely-guarantee reasoning [Bizjak and Birkedal 2017; Calcagno et al. 2007; Dinsdale-Young et al. 2013]. A key differentiator of our approach is the focus on a relational semantics of separation logic that captures the notion that a component of one program is represented by another in a way that comes with permissions to modify state. A small technical difference is that these approaches are all based on partial commutative monoids, where $\mathtt{p} * \mathtt{q}$ is undefined when $\mathtt{p}$ and $\mathtt{q}$ are not separate, whereas the separating conjunction $\mathtt{T} \star \mathtt{U}$ of two permission types is defined and asserts that the current state is one in which $\mathtt{T}$ and $\mathtt{U}$ are separate.

9 CONCLUSION

In this paper, we have presented a type system that can extract functional specifications from imperative programs. To do this, we have developed a novel, relational semantics of a form of separation logic called *permission types* that can represent impure values in imperative programs with pure values in specifications. The extracted specifications are stuttering bisimilar up to errors to the original imperative programs, meaning that we can prove temporal properties like safety, liveness, and fairness of the latter by proving those temporal properties, plus error freedom, of the former. Even though we have not explored it much here, our framework supports nondeterminism, which suggests it can handle concurrent programs. We plan to explore this as future work. Additionally, we plan to explore applying the Heapster approach to Rust programs, which we anticipate will be a sweet spot for Heapster: because the Rust type system already captures memory safety and permissions, we anticipate that it should require few if any user annotations to extract functional specifications from Rust programs.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1521539 and the ONR under Grant No. N00014-17-1-2930. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the ONR.

REFERENCES

Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. In *Proceedings of the 34th Annual ACM Conference on Object-Oriented Programming Systems,* *Languages and Applications (OOPSLA).*

Lennart Beringer and Andrew W. Appel. 2019. Abstraction and Subsumption in Modular Verification of C Programs. In *Proceedings of the 23rd International Symposium on Formal Methods (FM).*

Aleš Bizjak and Lars Birkedal. 2017. On Models of Higher-Order Separation Logic. In *Proceedings of the 33rd Conference on the Mathematical Foundations of Programming Semantics.*

Samuel Boutin. 1997. Using reflection to build efficient and certified decision procedures. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS).*

Michael C Browne, Edmund Clarke, and Orna Grumberg. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science* 59 (1988). Issue 1–2.

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *Proceedings of the Twenty-Second Annual IEEE Symposium on Logic in Computer Science (LICS).*

Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. 2018. Continuous Formal Verification of Amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV).*

Edmund M. Clarke and E. Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs.*

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking.* MIT Press.

Rocco de Nicola and Frits Vaandrager. 1990. Three Logics for Branching Bisimulation. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS).*

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).*

Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Proceedings of the 8th International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE).*

Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *18th European Symposium on Programming (ESOP).*

Xinyu Feng. 2009. Local Rely-Guarantee Reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).*

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation Logic. In *Proceedings of the 48th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-Guarantee References for Refinement Types over Aliased Mutable Data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

Claudio Jeker. 2008. OpenBSD Network Stack Internals. In *Proceedings of AsiaBSDCon.*

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. In *Proceedings of the 47th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL).*

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Joseph Turon, Lars Birkedal, and Derek R Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).*

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019).* Association for Computing Machinery, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).*

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41, 1 (2008).

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martinez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP).*

Gregory Malecha, Adam Chlipala, and Thomas Braibant. 2014. Compositional Computational Reflection. In *Proceedings of the 5th International Conference on Interactive Theorem Proving (ITP).*

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-based verification for Rust programs. In *European Symposium on Programming*. Springer, Cham, 484–514.

Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences* 17, 3 (1978), 348–375.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

Matthew J. Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *Proceedings of the 20th European Symposium on Programming (ESOP)*.

Redox Developers. [n.d.]. The Redox Operating System. https://doc.redox-os.org/book/. Accessed: Nov 13, 2020.

John C Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*.

Lucas Silver and Steve Zdancewic. 2021. Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees. In *48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems* 34, 1 (2012).

Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification. In *Proceedings of the 2020 ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

Wouter Swierstra. 2008. Datatypes à la Carte. *Journal of Functional Programming* 18, 4 (2008).

Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification*. Master's thesis. Karlsruhe Institute of Technology.

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR)*.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.