Seam: Provably Safe Local Edits on Graphs

MANOLIS PAPADAKIS, Stanford University, USA
GILBERT LOUIS BERNSTEIN, Stanford University, USA
RAHUL SHARMA, Microsoft Research, India
ALEX AIKEN, Stanford University, USA
PAT HANRAHAN, Stanford University, USA

Algorithms that create and mutate graph data structures are challenging to implement correctly. However, verifying even basic properties of low-level implementations, such as referential integrity and memory safety, remains non-trivial. Furthermore, any extension to such a data structure multiplies the complexity of its implementation, while compounding the challenges in reasoning about correctness. We take a language design approach to this problem. We propose Seam, a language for expressing local edits to graph-like data structures, based on a relational data model, and such that data integrity can be verified automatically. We present a verification method that leverages an SMT solver, and prove it sound and precise (complete modulo termination of the SMT solver). We evaluate the verification capabilities of Seam empirically, and demonstrate its applicability to a variety of examples, most notably a new class of verification tasks derived from geometric remeshing operations used in scientific simulation and computer graphics. We describe our prototype implementation of a Seam compiler that generates low-level code, which can then be integrated into larger applications. We evaluate our compiler on a sample application, and demonstrate competitive execution time, compared to hand-written implementations.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Formal software verification**;

Additional Key Words and Phrases: Domain-specific languages, Verification, Graph data structures

ACM Reference Format:
Manolis Papadakis, Gilbert Louis Bernstein, Rahul Sharma, Alex Aiken, and Pat Hanrahan. 2017. Seam: Provably Safe Local Edits on Graphs. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 78 (October 2017), 29 pages.
https://doi.org/10.1145/3133902

1 INTRODUCTION

Local edits on graphs are operations that mutate graph structure by deleting, creating, and rewiring nodes and edges in local neighborhoods. Such operations touch only a portion of the graph, within a statically bounded number of hops away from their arguments. For instance, contracting an edge, splitting an edge, and merging two vertices are all local edits of graphs. In applications, these basic kinds of operations manifest in richer, more structured settings. For example, two locations in a geospatial service are discovered to be duplicates, and must be merged; elements of a geometric mesh are subdivided to increase local resolution; a user on a social network blocks another user's access.

2475-1421/2017/10-ART78
https://doi.org/10.1145/3133902

Reasoning about the correctness of local edit operations is non-trivial, and this makes it easy to introduce subtle bugs when implementing them. Such operations can freely delete, create, or update objects and the connections between them, and thus low-level implementations have the potential to violate basic memory correctness (e.g., no dangling pointers) or application-specific invariants over the stored data (e.g., a policy that a blocked user is not allowed to tag their blocker in photographs). We refer to these properties collectively as *data integrity*.

Each successive extension to the overall data structure (i.e., the addition of more classes of objects, edit operations, and invariants) becomes increasingly harder to reason about, and thus implement correctly. New invariants must be checked against all existing operations. A new operation often requires new access patterns (e.g., finding all the triangles connected to a given vertex, or finding all the photos taken by a blocked user that tag the blocking user), and introducing additional data structures to support those access patterns multiplies code complexity, as these new data structures must be maintained in all operations.

This paper presents a method for verifying that local edit operations preserve data integrity. Rather than attempt to directly verify the complex code that results from writing such programs in general-purpose languages, we design and implement a domain-specific language, called Seam, to simplify the development and verification of graph data structures and local edit operations on them. Seam provides sound (no false negatives) and precise (no false positives) verification of local graph edits, by utilizing an SMT solver: If all the proof obligations that Seam generates are discharged by the solver, then the operation in question is guaranteed to preserve data integrity in all executions. If a proof obligation is disproven, then Seam provides a concrete execution input, defined entirely over the data structures of the original program, that demonstrates the erroneous behavior. Because the underlying logic that Seam uses is undecidable, non-termination is also a possible outcome of attempting to verify a graph operation, though our verifier terminates on all of our examples.

Key to our verification approach is the choice to disallow general recursion, and instead restrict Seam's looping construct to be a map over a finite (but unbounded) set. This looping construct is sufficient for expressing local graph edit operations, and allows us to precisely identify each point in an operation's execution by the valuation of all visible loop variables. In turn, this enables the verifier to precisely and succinctly name all values, especially all newly-created values, to which a reference in the code can evaluate, using quantification over the sets being looped over.

Another crucial design choice is Seam's transactional execution semantics: Memory operations are collected in a log, and applied atomically after execution finishes. Thus, our verifier needs only reason about the initial and final states of the data structure being updated, not intermediate states. Additionally, we build our abstract representation of an operation's effects using multisets, rather than sets, which allows us to reason about duplicate values appearing at different points in the execution (e.g., the same value appearing twice in delete statements). Finally, we allow programmers to define the contents of auxiliary data structures declaratively, in terms of the base data, and the Seam compiler produces correct-by-construction code to maintain them, thus reducing the amount of code to verify.

Besides verification, our goals in designing Seam included programmer productivity and performance of generated code competitive with native compiled code (so that code generated by Seam can be incorporated into performance-critical applications). Perhaps surprisingly, in Seam these goals are reinforced, rather than hindered, by the need to support verification. Implementing local graph edits by hand requires solving complicated memory and index management problems. Programmers struggle to manage this complexity, leading to simplified and inefficient implementation strategies. In Seam, an operation's memory safety is statically verified, so the programmer

does not need to perform any runtime checks, and auxiliary data structures are managed automatically, by compiler-generated code. This reduction in the programmer's responsibilities also translates to more concise code. While preliminary, our performance results are very encouraging: Seam-generated sequential code runs faster than hand-written sequential C++ code for at least one application, compared to two existing code bases, precisely because those code bases adopt inefficient implementations to reduce conceptual complexity. Furthermore, the corresponding Seam code is an order of magnitude shorter than the handwritten implementations.

This paper makes the following contributions:

- We propose a new declarative language, Seam, that allows programmers to express local graph edits more concisely, compared to general-purpose language such as C++ or Java. Our compiler translates Seam programs into low-level code that can be integrated into native applications.
- We present a sound and precise method for statically verifying that Seam operations maintain data integrity in all executions (thus obviating the need for dynamic checks). Given an operation, our verifier produces proof obligations that, if proven valid by an SMT solver, guarantee that the operation is memory-safe, maintains referential integrity, and preserves all user-specified invariants. If a proof obligation is disproven, the counter-example provided by the SMT solver is translated into an operation input that is guaranteed to demonstrate erroneous behavior.
- We demonstrate our method's applicability to a new class of data structure verification tasks that arise from geometric remeshing operations used in scientific simulation and computer graphics.
- We present an empirical evaluation demonstrating the effectiveness of our verification method in practice. Although verification of Seam programs is generally undecidable, all of our examples are verified in under a minute, and their majority in under a second.
- We demonstrate end-to-end integration of Seam-generated code into an example application, and give preliminary runtime measurements showing that the Seam compiler can generate more efficient code than existing handwritten C++ code in at least one case.

2 EXAMPLES

Seam is a domain-specific language for describing graph data structures and writing local graph edits. Data structures are described using a formalism analogous to a relational database schema. Graph modification operations are written in a custom language, designed to support features commonly used in local graph edits, while remaining amenable to formal verification. Programmers can use these operations as modification primitives, composing them into larger applications. Seam itself lacks any sequential execution constructs to compose the individual edit operations. Thus, a Seam program needs to be paired with a host language that performs the initial build of the data structures and invokes the operations.

In this section we give an overview of Seam's design using two examples: a simple social network and a triangle mesh. More examples of Seam code can be found in Appendix D[1]. We discuss Seam's limitations in Section 10.

2.1 A Simple Social Network

Consider a simple social network with user accounts that follow one another. We can model the `schema` for such a data structure in Seam, as shown in Figure 1a. The keyword `table` indicates an unordered set of elements, and `field` defines per-element data stored in a table. For instance, every

[1]The Appendix is available on the ACM digital library, as a supplementary document to this paper.

```
schema SocialNet
  table Account
  table Follow
  field Follow.src:Account
  field Follow.dst:Account
  invariant no_follow_self( f:Follow )
    assert(f.src != f.dst)
  end
end

operation SocialNet.Remove( a:Account )
  delete a
  for f in Follow where f.src == a do
    delete f
  end
  for f in Follow where f.dst == a do
    delete f
  end
end
```

(a) Original schema and operations

```
-- Incorrect version, will not compile
operation SocialNet.Merge( a1:Account, a2:Account )
  delete a2
  for f in Follow where f.src == a2 do
    f.src = a1 end
  for f in Follow where f.dst == a2 do
    f.dst = a1 end
end

-- Compiler error message
Invariant 'no_follow_self' violated, on input:
  Accounts = { a1, a2 }
  Follows  = { f1 }
  f1.src = a1, f1.dst = a2

-- Corrected version
operation SocialNet.Merge( a1:Account, a2:Account )
  delete a2
  for f in Follow where f.src == a2 do
    if f.dst == a1 then delete f else f.src = a1 end
  end
  for f in Follow where f.dst == a2 do
    if f.src == a1 then delete f else f.dst = a1 end
  end
end
```

(b) Adding an account merge operation

Fig. 1. A simple social network data structure

`Follow` models a kind of directed edge between accounts, by including a `src` and a `dst` `Account`. Each entry in a table is unique (even if it happens to have the same values as another entry for all fields), and addressable by a unique identifier, akin to unique row identifiers in relational databases.

Seam programmers can specify `invariant`s by writing a function that `assert`s the properties defining the invariant. Invariants are thus expressed using familiar control flow constructs. By restricting invariants to have a single argument, we help ensure that they are local and can be efficiently checked. Here, `no_follow_self` in Figure 1a is a simple invariant that disallows self-loops.

Once a schema has been defined, a Seam programmer can write `operation`s on that schema. For instance, the operation `SocialNet.Remove` of our example schema removes a user account from the network.

In Seam, `table`s represent unordered sets. Consequently, our operation definitions must be insensitive to the order in which the table elements are looped over. Furthermore, to keep operations local, we only allow looping over elements connected to a variable already in scope. These two decisions lead to a constrained loop construct that provides access to all elements connected with some input element (`a` in the above operation), through a specific field (`src` and `dst`, for the first and the second loop respectively).

Operations are not just order-insensitive, they are transactional. All the operation's effects on memory (`delete`s, `new`s, and field writes) are applied atomically, after the body of the operation is done executing. As a result, even though Seam operations are written as pseudo-imperative code, their semantics are declarative. This behavior further frees the programmer from the responsibility of ensuring that effects are correctly ordered. For instance, in operation `SocialNet.Remove` (Figure 1a), `Account a` is deleted before being used in the two loops.

Seam statically checks that all references to the deleted `Account a` have either been removed or re-assigned at compile time. If we were using a database, we could have handled this particular case by declaring an `ON DELETE CASCADE` policy [Ullman et al. 2002] between `Accounts` and `Follows`. However, such policies have the drawback of defining behavior globally, rather than per-operation.

Suppose we realize that our network contains duplicate accounts, so we decide to implement an operation to merge accounts (Figure 1b). This operation deletes one of the two accounts, but rather than delete the connected follows, re-routes them to the retained account. As this example shows, different applications, and even different operations within the same application, may require different approaches to maintaining data integrity.

If we are in a rush, we might quickly code the account merge operation as shown at the top of Figure 1b, without realizing that it can violate our invariant `no_follow_self`. By using the model-finding capabilities of SMT solvers, Seam is able to provide programmers with concise counter-example inputs. Here, the compiler responds with a minimal network (containing two accounts) that satisfies the invariant, but will no longer do so after `SocialNet.Merge` executes. If we modify the loops to check for follows between the merged accounts, and delete those follows, then we get an operation (bottom of Figure 1b) that successfully compiles.

As we add more types of elements, operations or invariants to our data structure, Seam automatically checks for problems arising from their interactions.

2.2 Triangle Mesh: Edge-Based Remeshing

Geometric data structures, used to represent 2D and 3D shapes, are a rich class of complex graph data structures. For instance, a triangle mesh (a common input format for rendering engines, such as OpenGL) consists of triangles and vertices, where each triangle has three distinct vertices. Figure 2a defines a possible implementation of a triangle mesh data structure in Seam.

In implementations of triangle mesh data structures, the edges between vertices are often defined as another table of explicitly managed elements. However, this is redundant; once the triangles are specified, all the edges between their vertices are implicitly defined. To help ease and automate the programmer's work in such situations, we leverage the concept of a *view* from databases. A `view` is a set of typed tuples, along with a function (`viewdef`) that computes its contents based on the contents of the base tables. Such view definition functions take a single argument and use the same control constructs as operations, but use `emit` statements in place of `delete`, `new` and field-write statements.

One of the more complex triangle mesh-based techniques, called *adaptive remeshing*, is the continuous adaptation of a mesh, to adjust its resolution and fidelity over the course of a larger computation [Brochu and Bridson 2009; Da et al. 2014; Narain et al. 2013, 2012; Pfaff et al. 2014; Wojtan et al. 2009, 2010]. This technique is critical in the simulation of certain phenomena, such as paper folding and paper tearing. A common approach to remeshing is to define two local edit operations, edge-collapse and edge-split (Figure 3), to reduce and increase the resolution respectively. The edge-split operation creates a new *split vertex*, and then replaces every triangle connected to the edge with the two triangles resulting from splitting that edge. The edge-collapse operation collapses one vertex into another by deleting it (similar to account merging), then re-routing any connected triangles to the remaining vertex. We implement these operations in Figure 2c.

This example illustrates how the view mechanism eliminates a combinatorial increase in the amount of code, as an application is extended. The views `TV` and `TE` in this example are each used in only one of the two operations. The Seam compiler automatically generates code to propagate updates to the views, and adds it to both operations. If we had instead represented these views as base tables (equivalently to what programmers have to do when they develop similar applications manually), we would have had to write update code for the combination of each view and each

```
schema TriMesh
  table Tri  -- triangles
  table Vert -- vertices
  field Tri.v0:Vert
  field Tri.v1:Vert
  field Tri.v2:Vert
  invariant non_degenerate( t:Tri )
    assert(t.v0 != t.v1 and t.v1 != t.v2
                        and t.v0 != t.v2)
  end
  view Edge : { hd:Vert, tl:Vert }
  -- def: hd & tl appear on same triangle
  view TE : { t:Tri, hd:Vert, tl:Vert }
  -- def: {hd,tl} is an edge of t
  view TV : { t:Tri, v:Vert }
  -- def: v is a vertex of t
  viewdef( t:Tri )
    emit { t, t.v0 } into TV
    emit { t, t.v1 } into TV
    emit { t, t.v2 } into TV
    emit { t.v0, t.v1 } into Edge
    emit { t, t.v0, t.v1 } into TE
    emit { t.v1, t.v2 } into Edge
    emit { t, t.v1, t.v2 } into TE
    ...
  end
end
```

(a) Schema

```
invariant unique_tri_key( t:Tri )
  assert(forall t2:Tri
         where t2.v0 == t.v0,
               t2.v1 == t.v1,
               t2.v2 == t.v2
         : t == t2)
end
```

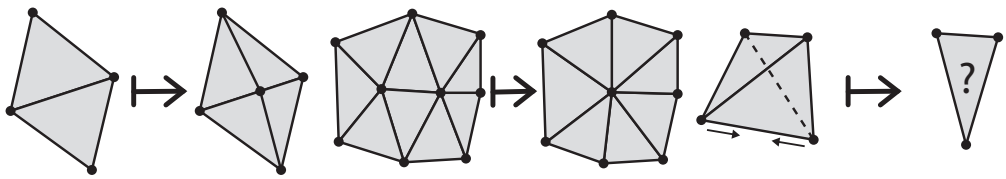
(b) Additional invariant: no duplicate triangles

```
operation TriMesh.EdgeSplit( e:Edge )
  let vh = e.hd -- head of e
  let vt = e.tl -- tail of e
  let sv = new Vert -- vertex splitting the edge
  for t_e in TE
  where t_e.hd == vh, t_e.tl == vt do
    let t = t_e.t
    -- the following line is a pattern-match
    let vh, vt, vopp = t.v0, t.v1, t.v2
    delete t
    new t_h:Tri { v0 = vh, v1 = sv, v2 = vopp }
    new t_t:Tri { v0 = sv, v1 = vt, v2 = vopp }
  end
end

operation TriMesh.EdgeCollapse( e:Edge )
  let vh = e.hd
  let vt = e.tl
  -- strategy: delete vt, then
  -- redirect touching triangles to vh
  delete vt
  for t_v in TV where t_v.v == vt do
    let t = t_v.t
    if t.v0 == vh or t.v1 == vh or t.v2 == vh then
      delete t
    else
      if     t.v0 == vt then t.v0 = vh
      elseif t.v1 == vt then t.v1 = vh
      elseif t.v2 == vt then t.v2 = vh end
    end
  end
end
```

(c) Operations

Fig. 2. A triangle mesh data structure



(a) `TriMesh.EdgeSplit` (b) `TriMesh.EdgeCollapse` (c) Tetrahedral Collapse Edge Case

Fig. 3. Edge-based remeshing operations

operation, i.e. we would have had to maintain a quadratically rather than linearly growing amount of code, as new operations and views are added.

This example also illustrates how Seam can automatically identify problematic edge cases, that the programmer could easily miss. Suppose we decide we need to enforce an additional invariant, that no duplicate triangles are allowed (i.e., triangles with the same set of vertices). Seam lets us concisely express this requirement (Figure 2b). Having done so, the `EdgeSplit` operation still compiles, but `EdgeCollapse` does not. In fact, this problem is a well-known edge case, which Seam identifies automatically: a triangle mesh on four vertices, forming a tetrahedron (Figure 3c). When one of the edges is collapsed (indicated in the figure by the two arrows, which show the two endpoints of the edge being merged into one), two triangles are deleted, and the remaining two triangles coincide on the same three remaining vertices. In some papers [Da et al. 2014] this issue is addressed by deleting one of the two triangles, effectively merging them. In other papers [Brochu and Bridson 2009] this case is handled by deleting both triangles. Which rule is appropriate depends on what the mesh is modeling. Seam directs the programmer's attention to such edge cases, rather than trying to automatically resolve these fundamentally application-specific decisions.

Finally, this example illustrates how Seam helps support the safe evolution of data structure code over time, as application requirements change. Consider the case of the ArcSim code base, which was used in the publication of three successive research papers [Narain et al. 2013, 2012; Pfaff et al. 2014], on cloth, paper folding, and paper tearing simulation respectively—all relying on triangle mesh edge-based remeshing. As the code base evolved, the schema changed. Initially, the triangle mesh (implemented as a C++ data structure) resembled our `TriMesh` schema, without views. However, by the third paper, a new requirement had been added. The triangle mesh now had to track two "versions" of the paper sheet being simulated: its torn topology, as well as its topology as if it had not been torn. Both topologies were modeled using triangles and vertices, with vertices having a one-to-one correspondence between topologies, except for one case: A single vertex on the untorn topology, if located directly on a tear, would correspond to two (or more) vertices on the torn topology, tracking its position on the two (or more) sides of the tear. We could easily model this extension in Seam, by declaring a new `UntornVertex` table, and adding an `untorn_vertex` field to the `Vertex` table. Then, instead of having to manually reason about the effects of this modeling change on all existing operations, we could rely on the compiler to direct our attention to the minimal set of edge-cases that need to be addressed.

3 TECHNICAL OVERVIEW

The main technical result of this paper is a proof of correctness: Provided the operations in a Seam program typecheck and pass the SMT-based checker, then they are memory-safe and preserve all user-specified invariants (Theorem 5.1). Conversely, if an operation fails some check, then there exists some input for which the operation violates memory safety or invalidates some invariant, and we can construct such an input from the counter-example provided by the SMT solver (Theorem 5.2). In the following sections, we give the major definitions and lemmas required for this proof; the complete proof can be found in Appendix C.

The first step towards expressing our theorems is to formally define a core Seam language (Section 4), such that full Seam programs can be reduced to it (Section 6). We develop a syntax and type system for this core language, and define its execution semantics over a memory model that maps each schema component onto an abstract machine memory. To capture the transactional nature of operation execution, we separate Seam's operational semantics in two stages: an *execution* stage, which produces a *log* of effects to be applied, and a log *application* stage. This separation allows us to formulate the property that data integrity is maintained as a *validity* property of the log. Our definition of a valid log includes a non-obvious *reorderability* component, which is useful

for guaranteeing that the operation is consistently defined, regardless of table iteration or log playback order.

In Section 5 we describe how to construct an abstract operation representation that succinctly describes the contents of all logs that may be produced in any execution, independent of a particular input model or argument assignment. We use this representation to construct a formula which encodes the property that the operation maintains data integrity in all executions. We delegate the task of proving this formula to an SMT solver.

4 LANGUAGE & SEMANTICS

In the following, we use the notation $\langle \ldots \rangle$ to create tuple values. We denote the set of k-length tuples, whose i-th element is drawn from set X_i, as $X_1 \times \ldots \times X_k$. We use $|t|$ for the length of tuple t, and $\pi_i(t)$ to project out its i-th element. We denote the set of sequences containing elements from X as $seq(X)$. We use $+\!\!+$ to concatenate sequences, and $[\ldots]$ to form sequence literals. We denote the length of a sequence s as $|s|$ and its i-th element, where $i \in \{1, \ldots, |s|\}$, as $s[i]$. We use the shorthand $x \in s$ to mean $\exists i \in \{1, \ldots, |s|\}.\ s[i] = x$. Note that we use 1-based numbering for both tuples and sequences.

4.1 Syntax

We present our formal discussion on a simplified version of Seam, called the *core language*. We discuss how to reduce full Seam programs to this language in Section 6. A program in this language consists of a single schema, possibly several invariants, and a single operation. The constructs of the core language are defined in Figure 4.

Any reference appearing in a Seam operation evaluates at runtime to either an entry that existed before the operation executed, or to an entry allocated during the current execution. We separate these two categories of references syntactically: *var* and *expr* can only refer to pre-existing entries, while a *nid* (new-id) can only refer to a newly-allocated entry. The programmer cannot read fields off of a *nid*, only use it directly in assignments. This behavior follows from our two-stage execution model: Allocation of new entries does not actually occur until after execution has completed, therefore it does not make sense to read their fields.

In the core language, references to newly allocated entries always appear annotated with a sequence of variables, forming a *parameterized nid* (*pnid*). Let $p = n[v_1, \ldots, v_k]$ be such a *pnid*, corresponding to *nid* n, and let s be the `new` statement that introduces it. Then $v_1, \ldots, v_k$ are the variables in the `for`-loops under which s is nested, ordered from outer-most to inner-most loop. All instances of the same *pnid* carry the same annotation. User-written operations do not include these annotations; our compiler adds this information automatically. Figure 2 in Appendix B outlines how to check that such annotations are consistent with the code. These annotations do not add any information to the program, but they help to simplify our formal presentation.

Seam's control constructs are standard, with the exception of the `for` loop, which has a restricted range: At runtime, a loop `for` v `in` R `do` s will execute once for each entry on table R, in an arbitrary order. This restriction is crucial in enabling our verification approach, because it allows any point in the execution to be uniquely identified as a combination of the specific statement being executed, and the values of all loop variables that are in-scope at that statement. This is true in particular for `new` statements, which implies that we can precisely name the values allocated during an execution, by the unique *nid* used at their allocation site, and the values of loop variables visible at their point of allocation. This fact motivates the annotation of *nid*s with loop variables, as described previously: The values of those loop variables are exactly the information we need to uniquely identify a newly-allocated value, wherever it might appear in the code.

$$
\begin{array}{rll}
R \in tab & \text{(table name)} & \\
f \in fld & \text{(field name)} & \\
v \in var & \text{(ref. to existing entry)} & \\
n \in nid & \text{(ref. to new entry)} & \\
p \in pnid & ::= nid\ \texttt{[}\ var^*\ \texttt{]} & \\
e \in expr & ::= var \mid expr\ .\ fld & \\
r \in ref & ::= expr \mid pnid & \\
c \in cond & ::= \texttt{true} & \\
& \mid expr == expr & \\
& \mid \texttt{not}\ cond & \\
& \mid cond\ \texttt{and}\ cond & \\
& \mid \texttt{forall}\ var : tab\ .\ cond & \\
s \in stmt & ::= \texttt{new}\ pnid : tab\ \texttt{in}\ stmt & \\
& \mid \texttt{delete}\ expr & \\
& \mid ref\ .\ fld = ref & \\
& \mid \texttt{if}\ cond\ \texttt{then}\ stmt & \\
& \mid \texttt{for}\ var\ \texttt{in}\ tab\ \texttt{do}\ stmt & \\
& \mid stmt\ ;\ stmt & \\
tabdef & ::= \texttt{table}\ tab & \\
flddef & ::= \texttt{field}\ tab\ .\ fld\ :\ tab & \\
inv & ::= \texttt{inv}\ cond & \\
o \in oper & ::= \texttt{oper(}\ [var : tab]^*\ \texttt{)}\ stmt & \\
prog & ::= tabdef^*\ flddef^*\ inv^*\ oper &
\end{array}
$$

Fig. 4. Core Seam language syntax

4.2 Typing

We assume no table or field is defined more than once, or used before its definition. For every field f declared as `field` $R_1.f : R_2$ we define $dom(f) = R_1$ and $rng(f) = R_2$. For every table R we define $flds(R)$ to be those fields f with $dom(f) = R$. We assume there are no name clashes between variables and *nid*s introduced at different places in the code.

We require the program's invariants and operation to adhere to a simple type system, presented in Figure 5. The set of types is exactly the set of tables declared in the program's schema. We also define the special types *Bool*, for well-typed conditions and invariants, and *Void*, for well-typed statements and operations.

In the Appendix (Lemma C.1) we prove that an operation's typing derivation, if one exists, is unique. This implies that, for every reference r in a well-typed operation o, there must exist a single node in o's unique type derivation that maps r to some table R. We then define $type(r) = R$. Additionally, if $type(p) = R$ for some *pnid* $p = n[v_1, \ldots, v_k]$, we also define $type(n) = R$.

Note that, as part of typechecking `new` statements, we also verify that all fields on newly-allocated entries are initialized exactly once. This is easy to enforce syntactically by disallowing initializations nested under `for`-loops or `if`-conditions.

4.3 Memory Model

We use an *abstract store* to represent the memory of the machine on which a Seam operation executes. A store is a mapping from abstract memory addresses, called *locations*, to memory cells. Let $l \in loc$ denote the set of all locations. We assume the store is of unbounded size (we can always obtain a fresh, unused location), and each memory cell has enough space to store the full contents of any table entry.

A Seam program can only access a (finite) subset of the store, the locations allocated for its tables. This subset of the store is captured by the *model* M of the program's schema, which is a pair $\langle M^{tab}, M^{fld} \rangle$ of mappings, associating each schema component to its concrete representation in the store: For every table R, $M^{tab}[\![R]\!] \subseteq loc$ is the (finite) set of memory locations that store entries of R. For every field f, $M^{fld}[\![f]\!] : loc \to loc$ is a function that retrieves the value of field f stored at a memory location. $M^{fld}[\![f]\!](l)$ is undefined if no value for field f has been previously stored at

$$\Gamma \in tenv = (var \cup pnid) \to tab$$

$$\boxed{inits[\![pnid, fld]\!] : stmt \rightharpoonup \mathbb{N}}$$

$$inits[\![p, f]\!](\texttt{new}\ p' : R\ \texttt{in}\ s) = inits[\![p, f]\!](s)$$

$$inits[\![p, f]\!](\texttt{delete}\ e) = 0$$

$$inits[\![p, f]\!](r_1\ .\ f' = r_2) =$$

$$\quad 1 \text{ if } r_1 = p \text{ and } f = f', \text{ otherwise } 0$$

$$inits[\![p, f]\!](\texttt{if}\ c\ \texttt{then}\ s) =$$

$$\quad 0 \text{ if } inits[\![p, f]\!](s) = 0, \text{ otherwise undefined}$$

$$inits[\![p, f]\!](\texttt{for}\ v\ \texttt{in}\ R\ \texttt{do}\ s) =$$

$$\quad 0 \text{ if } inits[\![p, f]\!](s) = 0, \text{ otherwise undefined}$$

$$inits[\![p, f]\!](s_1\ ;\ s_2) = inits[\![p, f]\!](s_1) + inits[\![p, f]\!](s_2)$$

$$\boxed{tenv \vdash ref : tab}$$

$$\frac{\Gamma(v) = R}{\Gamma \vdash v : R} \qquad \frac{\Gamma \vdash e : dom(f)}{\Gamma \vdash e.f : rng(f)} \qquad \frac{\Gamma(p) = R}{\Gamma \vdash p : R}$$

$$\boxed{tenv \vdash cond : Bool}$$

$$\Gamma \vdash \texttt{true} : Bool \qquad \frac{\Gamma \vdash e_1 : R \qquad \Gamma \vdash e_2 : R}{\Gamma \vdash e_1 == e_2 : Bool} \qquad \frac{\Gamma \vdash c : Bool}{\Gamma \vdash \texttt{not}\ c : Bool}$$

$$\frac{\Gamma \vdash c_1 : Bool \qquad \Gamma \vdash c_2 : Bool}{\Gamma \vdash c_1\ \texttt{and}\ c_2 : Bool} \qquad \frac{\Gamma[v \mapsto R] \vdash c : Bool}{\Gamma \vdash \texttt{forall}\ v : R\ .\ c : Bool}$$

$$\boxed{tenv \vdash stmt : Void}$$

$$\frac{\bigwedge_{f \in flds(R)} inits[\![p, f]\!](s) = 1 \qquad \Gamma[p \mapsto R] \vdash s : Void}{\Gamma \vdash \texttt{new}\ p : R\ \texttt{in}\ s : Void} \qquad \frac{\Gamma \vdash c : Bool \qquad \Gamma \vdash s : Void}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ s : Void}$$

$$\frac{\Gamma \vdash r_1 : dom(f) \qquad \Gamma \vdash r_2 : rng(f)}{\Gamma \vdash r_1.f = r_2 : Void} \qquad \frac{\Gamma \vdash e : R}{\Gamma \vdash \texttt{delete}\ e : Void}$$

$$\frac{\Gamma[v \mapsto R] \vdash s : Void}{\Gamma \vdash \texttt{for}\ v\ \texttt{in}\ R\ \texttt{do}\ s : Void} \qquad \frac{\Gamma \vdash s_1 : Void \qquad \Gamma \vdash s_2 : Void}{\Gamma \vdash s_1 ; s_2 : Void}$$

$$\boxed{\vdash inv : Bool} \qquad \boxed{\vdash oper : Void}$$

$$\frac{\varnothing \vdash c : Bool}{\vdash \texttt{inv}\ c : Bool} \qquad \frac{\{\langle v_1, R_1\rangle, \ldots, \langle v_n, R_n\rangle\} \vdash s : Void}{\vdash \texttt{oper}(v_1 : R_1, \ldots, v_n : R_n)\ s : Void}$$

Fig. 5. Typing rules

location l. An operation can update its model by emitting one of three possible memory operations, which we represent as parametric functions in $model \to model$:

- Insertion: $\texttt{INS}[\![tab, loc, pnid, env]\!] : model \to model$
  $\texttt{INS}[\![R_1, l_1, p, \eta]\!](M)$, defined only if $\bigwedge_{R \in tab} l_1 \notin M^{tab}[\![R]\!]$, returns a new model N where:
  – $N^{tab}[\![R]\!] = M^{tab}[\![R]\!] \cup \{l_1\}$ if $R = R_1$, otherwise $M^{tab}[\![R]\!]$
  – $N^{fld} = M^{fld}$ (fields of newly allocated entries are uninitialized)
  The last two parameters, p and η, are only used as annotations, and have no execution semantics.
- Deletion: $\texttt{DEL}[\![tab, loc]\!] : model \to model$
  $\texttt{DEL}[\![R_1, l_1]\!](M)$, defined only if $l_1 \in M^{tab}[\![R_1]\!]$, returns a new model N where:
  – $N^{tab}[\![R]\!] = M^{tab}[\![R]\!] \setminus \{l_1\}$ if $R = R_1$, otherwise $M^{tab}[\![R]\!]$
  – $N^{fld}[\![f]\!] = M^{fld}[\![f]\!]\big|_{N^{tab}[\![dom(f)]\!]}$ (fields of deleted entries become undefined)
- Update: $\texttt{UPD}[\![fld, loc, loc]\!] : model \to model$
  $\texttt{UPD}[\![f_1, l_1, l_2]\!](M)$, defined only if $l_1 \in M^{tab}[\![dom(f_1)]\!]$ and $l_2 \in M^{tab}[\![rng(f_1)]\!]$, returns a new model N where:
  – $N^{tab}[\![R]\!] = M^{tab}[\![R]\!]$
  – $N^{fld}[\![f]\!](l) = l_2$ if $f = f_1$ and $l = l_1$, otherwise $M^{fld}[\![f]\!](l)$

4.4 Operational Semantics

A Seam operation executes over an input model M and an environment η which maps the operation's arguments to locations on the appropriate tables. The execution proceeds according to the big-step semantics in Figure 6.

Executing an operation does not update the store immediately. Instead, the memory operations emitted by the execution are collected in a *log* $L = \langle L^i, L^d, L^u \rangle$, which is a triple of $seq(model \to model)$, containing all the emitted insertions, deletions and updates respectively. We lift the concatenation operator $+\!\!+$ to logs in the natural way: $L_1 +\!\!+_{log} L_2 = \langle L_1^i +\!\!+ L_2^i, L_1^d +\!\!+ L_2^d, L_1^u +\!\!+ L_2^u \rangle$. We say that L_1 is a permutation of L_2 iff each of L_1^i, L_1^d and L_1^u is a permutation of L_2^i, L_2^d and L_2^u respectively.

After execution finishes, the resulting log L is applied on the input model M, to form the output model $M' = apply[\![L]\!](M)$, where $apply[\![log]\!] : model \to model$ is defined as $apply[\![\langle L^i, L^d, L^u \rangle]\!] = L^u_{|L^u|} \circ \ldots \circ L^u_1 \circ L^d_{|L^d|} \circ \ldots \circ L^d_1 \circ L^i_{|L^i|} \circ \ldots \circ L^i_1$ (memory operations are applied in the order: insertions, deletions, updates).

Note that application of a log can fail, in case one of its constituent memory operations is undefined. The failure cases of memory operations were chosen to mirror error scenarios on a typical memory system. In particular, we have defined deletions to fail in cases that would trigger a double-free error on a real computer.

4.5 Safety Properties

We want to provide a language expressive enough that programmers can implement complex local edit operations. This flexibility, however, allows programmers to write operations that may (under certain inputs) execute successfully, but modify the store in such a way that data integrity is violated. Our solution is to formally define data integrity as a property of the program's model, called *validity*. We can then reduce the problem of verifying that an operation always maintains data integrity to the problem of verifying that the operation will never transform a valid model into an invalid one.

Additionally, we have chosen to leave certain details of operation execution undefined, to provide more flexibility to the implementation, and enable further optimizations in the future. In particular, the order in which table entries are visited during the execution of a `for`-loop is left undefined,

$$\eta \in env = (var \cup pnid) \to loc$$

$$\boxed{E[\![ref]\!] : model \times env \to loc}$$

$$E[\![v]\!](M,\eta) = \eta(v)$$

$$E[\![p]\!](M,\eta) = \eta(p)$$

$$E[\![e.f]\!](M,\eta) = M^{fld}[\![f]\!]\Big(E[\![e]\!](M,\eta)\Big)$$

$$\boxed{C[\![cond]\!] : model \times env \to Bool}$$

$$C[\![\texttt{true}]\!](M,\eta) = \text{TRUE}$$

$$C[\![\texttt{not } c]\!](M,\eta) = \neg\, C[\![c]\!](M,\eta)$$

$$C[\![e_1 == e_2]\!](M,\eta) = \Big(E[\![e_1]\!](M,\eta) = E[\![e_2]\!](M,\eta)\Big)$$

$$C[\![c_1 \texttt{ and } c_2]\!](M,\eta) = C[\![c_1]\!](M,\eta) \wedge C[\![c_2]\!](M,\eta)$$

$$C[\![\texttt{forall } v : R \,.\, c]\!](M,\eta) = \bigwedge_{l \in M^{tab}[\![R]\!]} C[\![c]\!](M, \eta[v \mapsto l])$$

$$\boxed{model, env \vdash oper \Downarrow log}$$

$$\frac{\eta = \{\langle v_1, l_1\rangle, \ldots, \langle v_n, l_n\rangle\} \quad \bigwedge_{i\in\{1,\ldots,n\}} l_i \in M^{tab}[\![R_i]\!] \quad M,\eta \vdash s \Downarrow L}{M,\eta \vdash \texttt{oper}(v_1 : R_1, \ldots, v_n : R_n)\ s \Downarrow L}$$

$$\boxed{model, env \vdash stmt \Downarrow log}$$

$$\frac{l \text{ is a fresh location} \quad M,\eta[p \mapsto l] \vdash s \Downarrow L \quad L' = \Big\langle \big[\texttt{INS}[\![R,l,p,\eta]\!]\big] +\!\!+ L^i, L^d, L^u \Big\rangle}{M,\eta \vdash \texttt{new } p : R \texttt{ in } s \Downarrow L'}$$

$$\frac{E[\![e]\!](M,\eta) = l \quad l \in M^{tab}[\![R]\!]}{M,\eta \vdash \texttt{delete } e \Downarrow \Big\langle [\,], \big[\texttt{DEL}[\![R,l]\!]\big], [\,]\Big\rangle}$$

$$\frac{E[\![r_1]\!](M,\eta) = l_1 \quad E[\![r_2]\!](M,\eta) = l_2}{M,\eta \vdash r_1.f = r_2 \Downarrow \Big\langle [\,], [\,], \big[\texttt{UPD}[\![f,l_1,l_2]\!]\big]\Big\rangle}$$

$$\frac{\neg\, C[\![c]\!](M,\eta)}{M,\eta \vdash \texttt{if } c \texttt{ then } s \Downarrow \Big\langle [\,], [\,], [\,]\Big\rangle}$$

$$\frac{C[\![c]\!](M,\eta) \quad M,\eta \vdash s \Downarrow L}{M,\eta \vdash \texttt{if } c \texttt{ then } s \Downarrow L}$$

$$\frac{M^{tab}[\![R]\!] = \{l_1, \ldots, l_n\} \quad \bigwedge_{i\in\{1,\ldots,n\}} M,\eta[v \mapsto l_i] \vdash s \Downarrow L_i \quad L' = L_1 +\!\!+_{log} \ldots +\!\!+_{log} L_n}{M,\eta \vdash \texttt{for } v \texttt{ in } R \texttt{ do } s \Downarrow L'}$$

$$\frac{M,\eta \vdash s_1 \Downarrow L_1 \quad M,\eta \vdash s_2 \Downarrow L_2}{M,\eta \vdash s_1 ; s_2 \Downarrow L_1 +\!\!+_{log} L_2}$$

Fig. 6. Operational semantics

which means that executing a single operation on a particular input can have multiple possible results. We make this non-determinism explicit in our semantics, so we can detect cases where a program's behavior is dependent on some loop's iteration order.

Definition 4.1 (Model validity)**.** A model M is *valid* iff all the following hold:

- No location is mapped to multiple tables:
  $\bigwedge_{R \neq R'} M^{tab}[\![R]\!] \cap M^{tab}[\![R']\!] = \varnothing$.
- Every field is defined on exactly those locations allocated to its table:
  $\bigwedge_{f \in fld} domain(M^{fld}[\![f]\!]) = M^{tab}[\![dom(f)]\!]$.
- Every field valuation falls within the field's range:
  $\bigwedge_{f \in fld} codomain(M^{fld}[\![f]\!]) \subseteq M^{tab}[\![rng(f)]\!]$.
- For each invariant defined as `inv` c, $C[\![c]\!](M, \varnothing)$ holds.

Note that the first three components of model validity follow directly from the mathematical definition of an abstract memory model, given in Section 4.3: Unless these three properties are satisfied, it is impossible to map the model meaningfully onto a real machine memory. Seam operations, however, are able to modify their model such that it violates one or more of these properties.

Definition 4.2 (Well-formed input)**.** A *well-formed input* for an operation defined as `oper`$(v_1 : R_1, \ldots, v_n : R_n)\ s$ is any pair of model M and environment η_0 that satisfy the following properties: M is a valid model, and η_0 maps exactly the arguments $v_1, \ldots, v_n$ of the operation, and is type-correct: $\bigwedge_{i \in \{1,\ldots,n\}} \eta_0(v_i) \in M^{tab}[\![R_i]\!]$.

Lemma 4.3 (Preservation)**.** *Let r be a reference appearing in operation o, with type$(r) = R$. Let $\langle M, \eta_0 \rangle$ be a well-formed input for o, and $E[\![r]\!](M, \eta)$ an evaluation of r occurring during execution of o on this input. Then, that evaluation always succeeds, producing some location l, where:*

- *If r is an expression, then $l \in M^{tab}[\![R]\!]$.*
- *If r is a pnid p and the execution terminates producing some log L, then $\exists \eta'$. $\texttt{INS}[\![R, l, p, \eta']\!] \in L^i$.*

The last lemma implies that, during execution of a well-typed operation on a well-formed input, any (arbitrarily nested) chain of type-correct field dereferences can be successfully followed, and the result will be a pre-existing location, on the appropriate table. Additionally, any reference to a newly-allocated entry evaluates to a location (allocated during execution of a `new` statement) that is scheduled for insertion.

Lemma 4.4 (Execution termination)**.** *If $\langle M, \eta_0 \rangle$ is a well-formed input for operation o, then execution of o on that input always completes, producing some log L: $M, \eta_0 \vdash o \Downarrow L$.*

Lemma 4.5 (Basic log properties)**.** *Assume that operation o executes on some well-formed input, producing a log L. Then, the following properties hold for L:*

- *Inserted locations are not present on any existing table.*
- *Each inserted location is distinct from all the others.*
- *Deleted locations exist on the appropriate table.*
- *Inserted locations are not deleted.*
- *Updates are type-correct:* $\texttt{UPD}[\![f, l_1, l_2]\!] \in L^u \Rightarrow l_1 \in M^{tab}[\![dom(f)]\!] \vee \exists p, \eta.\ \texttt{INS}[\![dom(f), l_1, p, \eta]\!] \in L^i$ *(and similarly for l_2 and $rng(f)$).*

Assume we are given an operation o and a well-formed input $\langle M, \eta_0 \rangle$ for o. Execution of o will always complete, producing some log L. Whether the execution was successful or not will be determined when we try to apply L to the input model M. We can thus reason about the properties of the execution by inspecting just the output log.

Definition 4.6 (Execution correctness)**.** The execution of some operation o on a well-formed input $\langle M, \eta_0 \rangle$, producing an output log L, is *correct* iff the output log L has all the following properties:

- *Applicable*: $apply[\![L]\!](M)$ is defined.
- *Validity-preserving*: $apply[\![L]\!](M)$ is a valid model.
- *Reorderable*: For any permutation L' of L, $apply[\![L']\!](M)$ is defined, and identical to $apply[\![L]\!](M)$.

Definition 4.7 (Operation safety)**.** An operation o is *safe* if it executes correctly on all well-formed inputs.

The last lemma implies that a safe operation *always* terminates when executed on a well-formed input, and the generated log can *always* be successfully applied on that input, is reorderable, and its application generates a valid output model.

Of the three properties required for a correct execution, the first two are self-explanatory: A correct execution must, at the very least, transform a valid model into another valid model. The reorderability property is useful for guaranteeing that an operation has the same behavior, regardless of the order in which the entries of a table are visited during the execution of a `for`-loop. The other source of non-determinism in the operational semantics is the arbitrary choice of a fresh

location, at each execution of a new statement. These fresh locations, however, are distinct from each other and from all pre-existing locations, therefore we can ignore this non-deterministic behavior, by considering the equivalence class of logs up to renaming of newly-allocated locations.

Lemma 4.8 (Execution non-determinism)**.** *Assume two possible executions of operation o on a well-formed input $\langle M, \eta_0 \rangle$, producing logs L_1 and L_2. Then L_2 is a permutation of L_1, up to renaming of inserted locations.*

Our final lemma connects the log properties required for a correct execution with a different set of properties, which can be more easily generalized to symbolic inputs, for use by our verification method.

Lemma 4.9 (Execution correctness—alternate)**.** *Assume an operation o, a well-formed input $\langle M, \eta_0 \rangle$ for o, and an execution $M, \eta_0 \vdash o \Downarrow L$. If L has all of the following properties, then the execution was correct:*

- NO-DOUBLE-FREE:
  $\bigwedge_{R \in tab} \nexists l, i \neq j.\ \left(L^d[i] = \texttt{DEL}[\![R, l]\!]\right) \wedge \left(L^d[j] = \texttt{DEL}[\![R, l]\!]\right)$
  i.e. the operation did not emit multiple deletions for the same entry
- NO-DOUBLE-ASSIGN:
  $\bigwedge_{f \in fld} \nexists l, i \neq j.\ \left(\exists l'.\ L^u[i] = \texttt{UPD}[\![f, l, l']\!]\right) \wedge \left(\exists l''.\ L^u[j] = \texttt{UPD}[\![f, l, l'']\!]\right)$
  i.e. the operation did not update any entry more than once on the same field
- NO-WRITE-ON-FREED:
  $\bigwedge_{f \in fld} \nexists l.\ \left(\texttt{DEL}[\![dom(f), l]\!] \in L^d\right) \wedge \left(\exists l'.\ \texttt{UPD}[\![f, l, l']\!] \in L^u\right)$
  i.e. the operation did not update any field of a deleted entry
- NO-SET-TO-FREED:
  $\bigwedge_{f \in fld} \nexists l.\ \left(\texttt{DEL}[\![rng(f), l]\!] \in L^d\right) \wedge \left(\exists l'.\ \texttt{UPD}[\![f, l', l]\!] \in L^u\right)$
  i.e. the operation did not set any field to point to a deleted entry
- NO-DANGLING-REFS:
  $\bigwedge_{f \in fld} \forall l.\ \left(\texttt{DEL}[\![rng(f), M^{fld}[\![f]\!](l)]\!] \in L^d\right) \Rightarrow \left(\texttt{DEL}[\![dom(f), l]\!] \in L^d\right) \vee \left(\exists l'.\ \texttt{UPD}[\![f, l, l']\!] \in L^u\right)$
  i.e. the operation updated or deleted all pointers to deleted entries
- INVS-MAINTAINED:
  $apply[\![L]\!](M)$ satisfies all invariants

Note that our safety properties only apply to each execution of an individual operation. Seam operations are meant to be invoked by a host application, that also initializes the data structures. We assume that the host program checks data integrity at initialization time, runs Seam operations atomically and sequentially, and cannot directly modify the data structures. Thus, by a simple inductive argument, we conclude that, if all invoked operations are safe, then the full program cannot violate data integrity.

Also note that our definition of safety only covers memory safety and maintenance of user-defined invariants. It is possible to write a safe operation that has logical bugs, or one that will never run because the invariants of its schema are impossible to satisfy.

5 VERIFICATION

This section presents our main result, a sound and precise method for checking whether a Seam operation is safe.

5.1 Abstraction of Program Values

We associate every reference r in the body of an operation with a *symbolic bag*, an abstract representation implicitly parameterized on the input model and argument assignment, that succinctly

$$select^k = \overbrace{ref \times \ldots \times ref}^{k}$$
$$g \in from = var \to tab$$
$$c \in where = cond$$
$$b \in bag^k ::= \varnothing \mid bag^k \uplus bag^k \mid \{select^k \mid from\ ;\ where\}$$
$$abs \ni A = \langle A^i, A^d, A^u \rangle$$
$$A^i \in nid \to bag^1$$
$$A^d \in tab \to bag^1$$
$$A^u \in fld \to bag^2$$

(a) Representation of effects

$$\boxed{\vdash oper \Uparrow abs}$$
$$\frac{\varnothing, \texttt{true} \vdash s \Uparrow A}{\vdash \texttt{oper}(v_1 : R_1, \ldots, v_n : R_n)\ s \Uparrow A}$$

(b) Symbolic execution of operations

$$\boxed{from, where \vdash stmt \Uparrow abs}$$
$$\frac{g, c \vdash s \Uparrow A \qquad b = \{p \mid g\ ;\ c\} \qquad p = n[\ldots]}{g, c \vdash \texttt{new}\ p : R\ \texttt{in}\ s \Uparrow \langle A^i[n \mapsto b], A^d, A^u \rangle}$$
$$\frac{A^i(n) = A^d(R \neq type(e)) = A^u(f) = \varnothing \qquad A^d(type(e)) = \{e \mid g\ ;\ c\}}{g, c \vdash \texttt{delete}\ e \Uparrow A}$$
$$\frac{A^i(n) = A^d(R) = A^u(f' \neq f) = \varnothing \qquad A^u(f) = \{\langle r_1, r_2 \rangle \mid g\ ;\ c\}}{g, c \vdash r_1.f = r_2 \Uparrow A}$$
$$\frac{g, c\ \texttt{and}\ c' \vdash s \Uparrow A}{g, c \vdash \texttt{if}\ c'\ \texttt{then}\ s \Uparrow A}$$
$$\frac{g[v \mapsto R], c \vdash s \Uparrow A}{g, c \vdash \texttt{for}\ v\ \texttt{in}\ R\ \texttt{do}\ s \Uparrow A}$$
$$\frac{g, c \vdash s_1 \Uparrow A_1 \qquad g, c \vdash s_2 \Uparrow A_2}{g, c \vdash s_1 ; s_2 \Uparrow A_1 \uplus_{abs} A_2}$$

(c) Symbolic execution of statements

Fig. 7. Representing Seam operations symbolically

describes all values that r can possibly evaluate to during any execution. We combine multiple bags, representing different references in the code, and tagged with the operation associated with each reference (e.g., deleted, or used as the key for an update), to form a complete abstraction of the operation. We also use symbolic bags to track pairs of references, so we can reason about keys and values of updates together.

Figure 7a lists the three possible forms of a symbolic bag: $\varnothing$ represents the empty bag, the $\uplus$ operator represents multiset sum (non-deduplicating union, used to combine information from different statements), and $\{r \mid g\ ;\ c\}$ is a *comprehension* (an analogue to set comprehension syntax, used to describe the effects of a specific statement). On a comprehension $\{r \mid g\ ;\ c\}$, any variable v appearing in r or c must be either an argument to the modeled operation, or introduced in g. We lift tuple projection to bags of tuples in the natural way: $\pi_i^{bag} : bag^k \to bag^1$.

Let $b = \{r \mid g\ ;\ c\}$ be a comprehension, representing the values to which some reference r can evaluate, and let $g = \{\langle v_1, R_1 \rangle, \ldots, \langle v_k, R_k \rangle\}$. We also write b as $\{r \mid v_1 : R_1, \ldots, v_k : R_k\ ;\ c\}$. The collection of values that b represents can be intuitively understood as the result of the SQL query `SELECT` r `FROM` $R_1\ v_1, \ldots, R_k\ v_k$ `WHERE` c: For every combination of values $l_1 \in R_1, \ldots, l_k \in R_k$ that satisfy $c[l_1/v_1, \ldots, l_k/v_k]$, b contains one instance of $r[l_1/v_1, \ldots, l_k/v_k]$ (b may contain duplicates of the same value). The *range* g on the bag represents the set of `for`-loops under which the modeled reference is nested, and the *filter* c the aggregate condition of all the containing `if`-statements. The reference r itself forms the *projection*, defining the syntactic form of the modeled reference, which is enough to fully define the value to which that reference would evaluate at runtime, given a valuation of the operation arguments, and loop variables in g. This is possible to do in Seam, because our looping constructs are essentially maps over finite sets, rather than general while-loops.

Unlike sets, symbolic bags include multiplicity information, which allows us to model cases where a reference evaluates to the same value on two separate occasions during the same execution, e.g. on two different iterations of a containing loop, and also cases where the same value occurs twice in the same execution, at the execution of two different statements. Like sets, symbolic bags do not explicitly include a notion of ordering. However, simply checking that certain bags do not contain duplicates is enough to verify that the modeled operation can never exhibit different behavior under different execution orders.

Note that our abstraction scheme actually involves no loss of precision. In a symbolic bag comprehension, values are tracked symbolically and the entire path predicate is included as the *filter* component. Rather, symbolic bags abstract over the two sources of non-determinism in our operational semantics: the iteration order of loops, and the choice of fresh locations made at the execution of `new` statements. We can ignore these aspects of execution because they are irrelevant for our verification method[2] and, if we verify that an operation is safe, we also guarantee that it executes correctly regardless of loop execution order and fresh location choices (Definition 4.7).

5.2 Abstraction of Operation Effects

Our goal in this section is to define an abstract representation for Seam operations, such that a single abstraction can succinctly represent the effects of the operation over all possible inputs. We will build this representation as a collection of symbolic bags, each modeling a different aspect of the operation's behavior.

We call our abstract representation of an operation an *abstract log*, $A \in abs$. We define an abstract log to represent all the concrete logs that can possibly be emitted by executing the modeled operation on any well-formed input. Specifically, each $A \in abs$ is comprised of the following parts:

- $A^i : nid \rightarrow bag^1$: a comprehension $\{p \mid g \; ; \; c\}$ for every *nid* n, recording the loops under which n's introduction statement s is nested (as a range g), and the aggregate condition up to s (as a filter c)
- $A^d : tab \rightarrow bag^1$: a bag for every table R, representing the entries deleted from that table
- $A^u : fld \rightarrow bag^2$: a bag for every field f, representing all the updates performed on it, as key-value pairs

We lift the $\uplus$ operator to abstract logs: If $A = A_1 \uplus_{abs} A_2$ then $A^i(n) = A_1^i(n) \uplus A_2^i(n)$ (similarly for A^d and A^u).

Figures 7b and 7c describe how to extract an abstract log from the code of an operation. As mentioned previously, a symbolic bag modeling a specific reference in the code needs to encode the entered loops, the aggregate path predicate up to that point, and the syntactic form of the reference. Therefore, our symbolic execution environment needs to include a set g of the entered loops, and a condition c that encodes the running path predicate.

5.3 Translation to Logic

Our goal in this section is to derive a formula which, if proven, verifies the safety of the operation under consideration. We will rely on an SMT solver to actually prove this formula. For our application, the fragment of first order logic with no predicate symbols except equality, as outlined in Figure 8b, is sufficient.

As part of this logical modeling, we need to define our domain of discourse. We need to define this domain such that the set of valid instances correspond exactly to the set of valid models of the operation's schema, and translating between them is straightforward. The domain should

[2]The specific locations returned by the execution of `new` statements are irrelevant, but the verification method does depend on the fact that different executions of such statements return distinct fresh locations.

therefore parallel the components of the operation's schema. Specifically, the domain will contain the following sorts and functions:

- For every table R, the following uninterpreted sorts: $\widetilde{R}$, which represents its initial contents, $\widetilde{R}^{\delta}$, which represents the entries inserted into it, and $\widetilde{R}^{out}$, which represents its final contents.
- For every field f with $dom(f) = R_1$ and $rng(f) = R_2$, the following uninterpreted functions: $\widetilde{f} : \widetilde{R}_1 \to \widetilde{R}_2$, which represents its initial contents, and $\widetilde{f}^{out} : \widetilde{R}_1^{out} \to \widetilde{R}_2^{out} \cup \{\texttt{NIL}\}$, which represents its final contents.
- For every *nid* n introduced in a statement `new` $n[v_1, \ldots, v_k]$: R `in` s, an uninterpreted function $\widetilde{n}$ with signature $\widetilde{n} : \widetilde{R}_1 \times \ldots \times \widetilde{R}_k \to \widetilde{R}^{\delta} \cup \{\texttt{NIL}\}$ (where $R_i = type(v_i)$), that names the entry which is allocated on each execution of the `new` statement.

All sorts are disjoint, except for the following pairs of sorts, for every R: $\widetilde{R}$ and $\widetilde{R}^{out}$, $\widetilde{R}^{\delta}$ and $\widetilde{R}^{out}$. The extra element NIL that we include in the codomain of $\widetilde{f}^{out}$ and $\widetilde{n}$ functions does not correspond to any value that occurs during execution. It is only included as a convenience, to simplify the development of the proofs.

5.4 Checking Operation Safety

Let $o =$ `oper`$(v_1 : R_1, \ldots, v_k : R_k)\ s$ be the operation we want to verify, and $\vdash o \Uparrow A$. Proving that this operation is safe reduces to proving that the formula in Figure 8c is valid. This formula uses a number of predicates on symbolic bags, which are defined on Figure 8a (we only give the definitions for bags in bag^1; these can be easily extended to bag^k). It assumes the following axioms:

SORT-DISJ: Only appropriate sorts may share elements:

Sorts representing the initial contents of different tables are disjoint: $\bigwedge_{R_1 \neq R_2} \widetilde{R}_1 \cap \widetilde{R}_2 = \varnothing$

Sorts representing the newly-created entries on different tables are disjoint: $\bigwedge_{R_1 \neq R_2} \widetilde{R}_1^{\delta} \cap \widetilde{R}_2^{\delta} = \varnothing$

The previous two categories of sorts are pairwise disjoint: $\bigwedge_{R_1, R_2 \in tab} \widetilde{R}_1^{\delta} \cap \widetilde{R}_2 = \varnothing$

INV-SAT: Any valid domain instance must satisfy the operation's invariants: $\bigwedge_{\texttt{inv}\ c} \Phi(c)$

NFUN-DEF: The $\widetilde{n}$ functions properly model the execution semantics of `new` statements: Assuming $A^i(n) = \{r \mid v_1 : R_1, \ldots, v_k : R_k\ ;\ c\}$, $R = type(n)$ and $A^i(n') = \{r' \mid v'_1 : R'_1, \ldots, v'_m : R'_m\ ;\ c'\}$:

Every execution of a `new` statement produces a new entry (represented by some element in $\widetilde{R}^{\delta}$):

$\bigwedge_{n \in nid}\ \forall \widetilde{v}_1 \in \widetilde{R}_1, \ldots, \widetilde{v}_k \in \widetilde{R}_k.\ \Phi(c) \Rightarrow \widetilde{n}(\widetilde{v}_1, \ldots, \widetilde{v}_k) \in \widetilde{R}^{\delta}$

Each element of a $\widetilde{R}^{\delta}$ sort corresponds to some execution of a `new` statement:

$\forall x \in \widetilde{R}^{\delta}.\ \bigvee_{n \in nid}\ \exists \widetilde{v}_1 \in \widetilde{R}_1, \ldots, \widetilde{v}_k \in \widetilde{R}_k.\ \Phi(c) \wedge x = \widetilde{n}(\widetilde{v}_1, \ldots, \widetilde{v}_k)$

Different executions of a `new` statement (for different iterations of its containing loops) allocate distinct entries:

$\bigwedge_n \forall x_1, y_1 \in \widetilde{R}_1, \ldots, x_k, y_k \in \widetilde{R}_k.\ \Phi(c)[x_i/\widetilde{v}_i] \wedge \Phi(c)[y_i/\widetilde{v}_i] \wedge \widetilde{n}(x_1, \ldots, x_k) = \widetilde{n}(y_1, \ldots, y_k) \Rightarrow \bigwedge_i x_i = y_i$

Entries allocated at different `new` statements are always distinct:

$\bigwedge_{n' \neq n} \forall \widetilde{v}_1 \in \widetilde{R}_1, \ldots, \widetilde{v}_k \in \widetilde{R}_k, \widetilde{v}'_1 \in \widetilde{R}'_1, \ldots, \widetilde{v}'_m \in \widetilde{R}'_m.\ \Phi(c) \wedge \Phi(c') \Rightarrow \widetilde{n}(\widetilde{v}_1, \ldots, \widetilde{v}_k) \neq \widetilde{n}'(\widetilde{v}'_1, \ldots, \widetilde{v}'_m)$

ROUT-DEF: For every $R \in tab$, the contents of $\widetilde{R}^{out}$ are consistent with the insertions and deletions occurring in the operation: $\bigwedge_{R \in tab} \widetilde{R}^{out} = \left\{x \in \widetilde{R} \;\middle|\; \neg\ in\Big(x, A^d(R)\Big)\right\} \cup \widetilde{R}^{\delta}$

FOUT-DEF: For every $f \in fld$ with $dom(f) = R_1$ and $rng(f) = R_2$, the definition of $\widetilde{f}^{out}$ is consistent with the updates occurring in the operation:

Updates are reflected on the final contents of the field:

$\forall x \in \widetilde{R}_1^{out}, y \in \widetilde{R}_2^{out}.\ in\Big(\langle x, y\rangle, A^u(f)\Big) \wedge \Big(\nexists z \in \widetilde{R}_2^{out}.\ y \neq z \wedge in\Big(\langle x, z\rangle, A^u(f)\Big)\Big) \Rightarrow \widetilde{f}^{out}(x) = y$

Non-deleted, non-updated entries retain their previous valuations on f:

$\forall x \in \widetilde{R}_1.\ \neg\ in\Big(x, A^d(R_1)\Big) \wedge \neg\ in\Big(x, \pi_1^{bag}(A^u(f))\Big) \wedge \neg\ in\Big(\widetilde{f}(x), A^d(R_2)\Big) \Rightarrow \widetilde{f}^{out}(x) = \widetilde{f}(x)$

$\boxed{in : term \times bag^1 \to form}$

$in(t, \varnothing) = \text{FALSE}$

$in(t, b_1 \uplus b_2) = in(t, b_1) \vee in(t, b_2)$

$in\Big(t, \{r \mid v_1 : R_1, \ldots, v_k : R_k \;;\; c\}\Big) =$

$\quad \exists \widetilde{v}_1 \in \widetilde{R}_1, \ldots, \widetilde{v}_k \in \widetilde{R}_k.\ \Big(\Phi(c) \wedge t = T(r)\Big)$

$\boxed{nodups : bag^1 \to form}$

$nodups(\varnothing) = \text{TRUE}$

$nodups(b_1 \uplus b_2) =$

$\quad nodups(b_1) \wedge nodups(b_2) \wedge$

$\quad \nexists x.\ in(x, b_1) \wedge in(x, b_2)$

$nodups\Big(\{r \mid v_1 : R_1, \ldots, v_k : R_k \;;\; c\}\Big) =$

$\quad \forall x_1, y_1 \in \widetilde{R}_1, \ldots, x_k, y_k \in \widetilde{R}_k.$

$\quad\quad (x_1 \neq y_1 \vee \ldots \vee x_k \neq y_k) \wedge$

$\quad\quad \Phi(c)[x_i/\widetilde{v}_i] \wedge \Phi(c)[y_i/\widetilde{v}_i]$

$\quad\quad\quad \Rightarrow T(r)[x_i/\widetilde{v}_i] \neq T(r)[y_i/\widetilde{v}_i]$

$\boxed{T : ref \to term}$

$T(v) = \widetilde{v}$

$T(e.f) = \widetilde{f}\Big(T(e)\Big)$

$T(n[v_1, \ldots, v_k]) = \widetilde{n}(\widetilde{v}_1, \ldots, \widetilde{v}_k)$

$\boxed{\Phi : cond \to form}$

$\Phi(\texttt{true}) = \text{TRUE}$

$\Phi(e_1 == e_2) = \Big(T(e_1) = T(e_2)\Big)$

$\Phi(\texttt{not}\ c) = \neg\, \Phi(c)$

$\Phi(c_1\ \texttt{and}\ c_2) = \Phi(c_1) \wedge \Phi(c_2)$

$\Phi(\texttt{forall}\ v : R\ .\ c) = \forall \widetilde{v} \in \widetilde{R}.\ \Phi(c)$

(a) Predicates on symbolic bags

$x, y, \widetilde{v} \in lvar \quad$ (1-to-1 mapping from v to $\widetilde{v}$)

$\widetilde{R}, \widetilde{R}^{\delta}, \widetilde{R}^{out} \in sort$

$\widetilde{n}, \widetilde{f}, \widetilde{f}^{out} \in fun$

$t \in term ::= lvar \mid fun\ (\ term^*\ )$

$\phi \in form ::= \text{TRUE} \mid \text{FALSE} \mid \neg\ form$

$\quad\quad \mid term = term \mid term \neq term$

$\quad\quad \mid form \wedge form \mid form \vee form$

$\quad\quad \mid \forall\ lvar \in sort.\ form$

$\quad\quad \mid \exists\ lvar \in sort.\ form$

(b) Logic fragment

$\forall \widetilde{v}_1 \in \widetilde{R}_1, \ldots, \widetilde{v}_k \in \widetilde{R}_k.$

[axioms] $\Rightarrow$

NO-DOUBLE-FREE :

$$\bigwedge_{R \in tab} nodups\Big(A^d(R)\Big)$$

NO-DOUBLE-ASSIGN :

$$\bigwedge_{f \in fld} nodups\Big(\pi_1^{bag}(A^u(f))\Big)$$

NO-WRITE-ON-FREED :

$$\bigwedge_{f \in fld} \nexists x \in \widetilde{R}.\ in\Big(x, A^d(R)\Big) \wedge in\Big(x, \pi_1^{bag}(A^u(f))\Big) \quad \text{where } R = dom(f)$$

NO-SET-TO-FREED :

$$\bigwedge_{f \in fld} \nexists x \in \widetilde{R}.\ in\Big(x, A^d(R)\Big) \wedge in\Big(x, \pi_2^{bag}(A^u(f))\Big) \quad \text{where } R = rng(f)$$

NO-DANGLING-REFS :

$$\bigwedge_{f \in fld} \forall x \in \widetilde{R}_1.\ in\Big(\widetilde{f}(x), A^d(R_2)\Big) \Rightarrow in\Big(x, A^d(R_1)\Big) \vee in\Big(x, \pi_1^{bag}(A^u(f))\Big) \quad \text{where } R_1 = dom(f) \text{ and } R_2 = rng(f)$$

INVS-MAINTAINED :

$$\bigwedge_{\texttt{inv}\ c} \Phi(c)\Big[\widetilde{f}^{out}/\widetilde{f}\Big]\Big[\widetilde{R}^{out}/\widetilde{R}\Big]$$

(c) Formula asserting $\vdash \texttt{oper}(v_1 : R_1, \ldots, v_k : R_k)\ s \Uparrow A$ is safe

Fig. 8. Verification of Seam operations

5.5 Proof of Correctness

Theorem 5.1 (Soundness of translation to logic)**.** *Let o be an operation with $\vdash o \Uparrow A$, and ϕ the formula defined in Figure 8c based on A. If ϕ holds, then o is safe.*

Proof. (Sketch) Assume that o is not safe. Then, there must exist some well-formed input $\langle M, \eta \rangle$ to o, and a possible execution $M, \eta \vdash o \Downarrow L$ on that input that is incorrect. Therefore, L must violate (at least) one of the properties outlined in Lemma 4.9. We can construct an instance D of ϕ's domain based on the valid model M (for the initial-state sorts and functions: each $M^{tab}[\![R]\!]$ is used as $\widetilde{R}$, each $M^{fld}[\![f]\!]$ as $\widetilde{f}$) and the execution producing L (for the rest of the sorts and functions, which model the modifications occurring during the operation). The environment η can be used to instantiate each of the universally quantified variables $\widetilde{v}_i$ at the head of ϕ, to $\eta(v_i)$. The resulting formula will be false: the domain instance D that we constructed satisfies all the axioms for the given instantiation of the quantified variables, but falsifies (at least) one of the conjuncts (the one corresponding to the correctness property that L violates). □

Theorem 5.2 (Completeness of translation to logic)**.** *Let o be an operation with $\vdash o \Uparrow A$, and ϕ the formula defined in Figure 8c based on A. If ϕ does not hold, then o is not safe.*

Proof. (Sketch) If ϕ does not hold, then there must exist some instance D of its domain of discourse and some instantiation of its quantified variables that satisfies all axioms on the precedent, but falsifies (at least) one of the conjuncts on the antecedent. Based on these, we can construct a well-formed input $\langle M, \eta \rangle$ to o (each $M^{tab}[\![R]\!]$ mirrors $\widetilde{R}$, each $M^{fld}[\![f]\!]$ mirrors $\widetilde{f}$, and η mirrors the instantiation of quantified variables), such that $M, \eta \vdash o \Downarrow L$ and L violates one of the properties from Lemma 4.9 (the one corresponding to ϕ's conjunct that is violated). □

5.6 Translation to SMT Query

The final step in checking an operation's safety is to actually verify the formula ϕ we have constructed. To do that, we use an SMT solver capable of reasoning in the theory of uninterpreted functions.

The logic fragment we have used in this section is not fully compatible with SMTLIB, the logic dialect accepted by modern SMT solvers. In particular, SMTLIB assumes that uninterpreted sorts are disjoint, which means we cannot directly express $\widetilde{R}^{\delta}$, $\widetilde{R}^{out}$ sorts (because we have assumed they can overlap with the corresponding $\widetilde{R}$ input-state sort), or $\widetilde{f}^{out}$, $\widetilde{n}$ functions (which have one of the problematic sorts as their domain or codomain). Thus, we have to drop these sorts and functions from the domain of discourse, and translate ϕ such that all quantifiers range over an input-state sort $\widetilde{R}$, and all functions occurring in terms are input-state functions $\widetilde{f}$. We perform this translation by applying the following four rewrites. Each rewrite is applied until fixpoint, before proceeding to the next one.

(1) For an equality or disequality involving more than one final-state function $\widetilde{f}^{out}$, decompose it into sub-formulas, each including exactly one call to a final-state function. For example, $z = \widetilde{g}^{out}\left(\widetilde{f}^{out}(x)\right)$ is decomposed into $\exists y \in \widetilde{R}^{out}.\ y = \widetilde{f}^{out}(x) \wedge z = \widetilde{g}^{out}(y)$, where $R = rng(f)$.

(2) For a $\forall x \in \widetilde{R}^{out}$ or $\exists x \in \widetilde{R}^{out}$ quantifier, use the appropriate ROUT-DEF axiom to decompose it into cases. For example, assume n is the only *nid* of type R in the operation, and $A^i(n) = \{r \mid v_1 : R_1, v_2 : R_2\ ;\ c\}$. Then $\forall x \in \widetilde{R}^{out}.\ \phi'$ becomes $\left(\forall x \in \widetilde{R}.\ \neg\ in(x, A^d(R)) \Rightarrow \phi'\right) \wedge \left(\forall \widetilde{v}_1 \in \widetilde{R}_1, \widetilde{v}_2 \in \widetilde{R}_2.\ \Phi(c) \Rightarrow \phi'[\widetilde{n}(\widetilde{v}_1, \widetilde{v}_2)/x]\right)$.

(3) For an equality or disequality involving a final-state function $\widetilde{f^{out}}$, use the appropriate FOUT-DEF axioms to decompose it into cases. For example, $y = \widetilde{f^{out}}(x)$ becomes $in\Big(\langle x, y\rangle, A^u(f)\Big) \vee \neg in\Big(x, A^d(R)\Big) \wedge \neg in\Big(x, \pi_1^{bag}(A^u(f))\Big) \wedge y = \widetilde{f}(x)$.
(4) For an equality or disequality involving a *nid* function $\widetilde{n}$, use the appropriate NFUN-DEF axioms to decompose it into cases. For example, $\widetilde{n}(x_1, \ldots, x_k) = \widetilde{n}(y_1, \ldots, y_k)$ becomes $x_1 = y_1 \wedge \ldots \wedge x_k = y_k$, while $\widetilde{n}(x_1, \ldots, x_k) = \widetilde{n'}(y_1, \ldots, y_m)$ and $\widetilde{n}(x_1, \ldots, x_k) = \widetilde{f}(x)$ are both trivially false.

Each time the first rewrite is applied, it reduces the number of nested $\widetilde{f^{out}}$ calls in ϕ by one. Each time the second rewrite is applied, it reduces the nesting depth of final-state quantifiers in ϕ by one and does not introduce any nested $\widetilde{f^{out}}$ calls. Each application of the third rewrite reduces the number of final-state functions appearing in ϕ by one and does not introduce any nested $\widetilde{f^{out}}$ calls or final-state quantifiers. On a formula with no final-state functions, each application of the fourth rewrite will remove at least one instance of a $\widetilde{n}$ function, while not introducing any final-state quantifiers or functions. Combining all these facts, we conclude that the full rewrite process is guaranteed to terminate.

For each rewrite, the original and final forms of the affected sub-formula can be shown to be logically equivalent, by applying the axioms in both directions. For the first three rewrites, which only affect the INVS-MAINTAINED part of ϕ, we additionally perform some simplifications that assume all the other properties, i.e. we implicitly convert ϕ from $Rest \wedge InvMaint$ to $Rest \wedge (Rest \Rightarrow InvMaint)$. For example, if NO-DANGLING-REFS holds, then for $dom(f) = R_1$ and $rng(f) = R_2$ we have that $\neg\ in\Big(x, A^d(R_1)\Big) \wedge \neg\ in\Big(x, \pi_1^{bag}(A^u(f))\Big)$ implies $\neg\ in\Big(\widetilde{f}(x), A^d(R_2)\Big)$, so we do not need to include the last constraint in the third rewrite. The last two rewrites also exploit the fact that, by the time they are applied to ϕ, all variables range over input-state sorts. Finally, the fourth rewrite exploits the fact that, whenever a $\widetilde{n}(\widetilde{v_1}, \ldots, \widetilde{v_k})$ term appears in ϕ, it is guarded by (at least) the constraint that the `new` statement which emits $\widetilde{n}(\widetilde{v_1}, \ldots, \widetilde{v_k})$ is reached.

After this process is completed, all axioms except INV-SAT have become irrelevant and can be dropped (because they reference a dropped domain, or, in the case of SORT-DISJ, are trivially satisfied). We also note that, after dropping the $\widetilde{n}$ functions from the domain of discourse, all uninterpreted functions we use are now monadic.

5.7 Undecidability of Verification

Our verification problem is undecidable in the general case, because the fragment L of first order logic that we use (Figure 8b) subsumes the fragment L' that includes a single sort, universal quantification, equality and two function symbols. The fragment L' is undecidable ([Börger et al. 2001], Theorem 4.0.1). All of the above features are necessary for modeling Seam programs: A Seam schema can include arbitrarily complex invariants, involving both universal and existential quantification, with no restriction on alternation. The verification tasks produced for operations on such a schema will contain formulas with correspondingly alternating quantifiers, as both hypotheses and goals. Function symbols are necessary for encoding the non-nullable fields on table entries. Equality is a necessary operator for testing connectivity in user code.

6 LANGUAGE EXTENSIONS

The full Seam language includes a number of constructs not present in the core language. Most of these can be reduced to the core language in a straightforward way. We present the most interesting cases here, and describe the rest in Appendix A.

6.1 Pre-Image Looping

In the full Seam language we disallow scoping `for`-loops and `forall` conditions over an entire table R. Instead, the user is required to constrain the iteration to the *pre-images* of bound variables, like so: `for x in R where x.`f_1 `==` e_1, ..., `x.`f_n `==` e_n `do` s. We can reduce such constrained looping constructs to the core language, by converting the `where` clause into an `if`-statement (or implication, in the case of `forall`).

We do not need to carry this restriction over to the core language, because that language is only used for the translation to logic, and the translation does not differentiate between a condition appearing in a `where` clause and one appearing in a nested `if`-statement. The reason for restricting programmers to pre-image loops is to limit the memory access patterns available to them, to only image operations (i.e., following a pointer stored in a field) and pre-image operations (i.e., accessing the entries that point to some in-scope value), thus guaranteeing that any operation they write is local. The locality of operations is an important part of ensuring that current SMT solver heuristics work well for our domain (see Section 8.2).

Another reason for this restriction is that it simplifies the task of generating efficient code. By restricting loops to pre-images, we require programmers to be explicit about their access patterns, and define whatever acceleration structures they need as views. This is not the norm with query languages such as SQL, the more traditional formalism that programmers use to interact with a relational data model. However, by requiring accesses to be explicit, we do not need to employ any form of query planning. Additionally, we can guarantee that both primitive access patterns we support are executed with minimal overhead: An image operation is simply a pointer dereference, and a pre-image loop can be executed efficiently by maintaining an inverted index on the field it uses (see Section 7). Therefore, unlike query languages, Seam can guarantee predictable performance, allowing users to reason about the performance characteristics of their code.

6.2 View Reduction

In the full language, view definitions are written using a syntax similar to operations, except only one argument is allowed, and `new`, `delete` and field-write statements are replaced by the single `emit {`$e_1, \ldots, e_k$`} in` *view* statement. For the purposes of verification, we simply inline the definition of each view V, whenever it is referenced in the code. This transformation proceeds by first converting a view definition to a symbolic bag, and then converting that bag into the code that gets inlined in place of a view reference.

To derive the symbolic bag b that describes a view V, we adapt the symbolic execution algorithm from Figure 7 to collect all the tuples of values that appear in `emit ... in` V statements. Then, we further rewrite b, treating it as a set (we can do this for bags that describe views, since the contents of view tables get deduplicated), to create a single comprehension, with only variables in the projection component. We do this by introducing output variables, e.g.:

$$\left\{x.f \;\middle|\; x{:}X, y_1{:}Y \;;\; y_1.g = x\right\} \uplus \left\{y_2.g \;\middle|\; y_2{:}Y, z{:}Z \;;\; z.h = y_2\right\} =$$
$$\left\{w \;\middle|\; w{:}X, x{:}X, y_1{:}Y, y_2{:}Y, z{:}Z \;;\; (w = x.f \wedge y_1.g = x) \vee (w = y_2.g \wedge z.h = y_2)\right\}$$

In the full language, a view can appear as the range of an iteration construct (i.e., a `for`-loop or `forall` condition), or as an argument (to an operation or invariant). We decompose an iteration over the contents of a view into multiple nested loops, ranging over all possible combinations of projected values, guarded by a condition that checks whether each such combination was indeed emitted into the view. For instance, suppose we have a view A described by the symbolic bag $\{\langle \mathtt{x}, \mathtt{y}\rangle \mid \mathtt{x} : \mathtt{X}, \mathtt{y} : \mathtt{Y}, \mathtt{z} : \mathtt{Z} \;;\; c\}$, and a loop `for a in A do` s. Then, the inlined version of this loop would be: `for` $\mathtt{a}_x$ `in X do for` $\mathtt{a}_y$ `in Y do if exists` $\mathtt{a}_z$ `: Z .` $c[\mathtt{a}_x/\mathtt{x}, \mathtt{a}_y/\mathtt{y}, \mathtt{a}_z/\mathtt{z}]$ `then` s', where s'

is the same as s, except `a.x` is replaced by $\texttt{a}_x$, `a.y` is replaced by $\texttt{a}_y$, and `a == e` is decomposed into $\texttt{a}_x$ `==` e`.x` `and` $\texttt{a}_y$ `==` e`.y`. An argument of view type is similarly decomposed into multiple arguments, one for each projected value, and with the guard condition wrapping the entire operation/invariant body.

7 IMPLEMENTATION

Our prototype compiler is written in Lua [Ierusalimschy et al. 1996] and uses Terra [DeVito et al. 2013] as a code-generation interface to LLVM [Lattner and Adve 2004]. Formulas produced by our verifier are checked using the CVC4 [Deters et al. 2014] SMT solver.

Most of our prototype implementation is determined by the layout of a Seam data structure. We represent table entries with 32-bit unsigned integers (keys), that index into multiple dynamic arrays (akin to a struct-of-arrays representation): (i) one array for every field, (ii) an extra array of reference counts for tracking external references, (iii) inverted indices, that enable efficient execution of pre-image loops, and (iv) view materializations indexed by the table. To give an example of an inverted index, consider the `SocialNet` example, and the loop that retrieves all `f in Follow where f.src == a`. To support this access, we store an inverted index of `Follow.src` on the `Account` table, which stores for every `Account a` a dynamic array of all `Follow` keys `f` satisfying `f.src == a`. This index is incrementally maintained whenever entries in `Follow` are inserted, updated, or deleted.

Seam maintains views similar to the way it maintains inverted indices. Tuples of a view V are grouped by whichever field they are accessed by in pre-image loops, and each tuple is annotated with the number of `emit` statement executions that generated it. We automatically derive incremental view maintenance code using a variation of Koch's algebraic "discrete derivative" approach [Koch 2010], and insert it where appropriate in the operation's code. For the examples in this paper, this method is excessive, since all our views simply consist of a list of `emit` statements, for which incremental update functions are trivial to derive.

To support the hosting of Seam code in larger applications, our compiler, given a schema and set of operations, generates a library of routines corresponding to the written operations, routines for initializing and navigating (but not directly modifying) the data structures, and routines for checking whether references to table entries held by the host program are stale. The initialization routines, besides bulk-loading the initial table data into Seam's internal representation, validate this data against the invariants, and build the view tables. This library is compatible with the C ABI, and thus can be used by any language that includes a C FFI.

One complication arises from the fact that the host program can retain references to Seam objects after those objects have been deleted. To handle this issue, the emitted API does not return raw identifiers to the host code, but instead wraps them into opaque reference objects. The host code can use appropriate routines to test whether the element pointed to by such a reference still exists.

8 EXPERIMENTS

8.1 Verification Timings

We implemented the verification method outlined in Section 5, and ran it to verify a collection of operations on the following schemas (each schema is followed by a description of its user-defined invariants):

- UGraph: A simple undirected graph
  - Self-loops are not allowed.
  - Any two edges cannot have the same set of vertices, even if those vertices appear in a different order on the two edges.

- DiGraph: A simple directed graph
  - Self-loops are not allowed.
  - Duplicate edges are not allowed.
  - For every edge, the symmetric edge also exists.
- SocialNetAsym: The social network example from Figure 1a
  - Self-`Follows` are not allowed.
- SocialNetSymm: A variant of SocialNetAsym, where the `Follow` relation is bidirectional
  - Self-`Follows` are not allowed.
  - For every `Follow`, the symmetric `Follow` also exists.
- SocialNetUnq: A variant of SocialNetAsym, where no duplicate `Follows` are allowed
  - Self-`Follows` are not allowed.
  - Duplicate `Follows` are not allowed.
- HalfEdge: An edge-based mesh [Sack and Urrutia 2000], where edges are connected directly with each other, through `next`, `prev` and `opp` pointers
  - The `next` pointer is the inverse of the `prev` pointer.
  - For every edge, the symmetric edge also exists.
  - Moving around a face, all edges have the same `face` pointer: `e.next.face == e.face`
  - Moving around a node, all edges have the same `head` pointer: `e.next.opp.head == e.head`
- BasicTriMesh: The triangle mesh example from Figure 2a
  - The three vertices of each triangle must be distinct.
- UnqTriMesh: A variant of BasicTriMesh
  - The three vertices of each triangle must be distinct.
  - Any two triangles cannot have the same set of vertices, even if those vertices appear in a different order on the two triangles.
- UnqTetMesh: A mesh composed of tetrahedra, each represented as a tuple of four vertices
  - The four vertices of each tetrahedron must be distinct.
  - Any two tetrahedra cannot have the same set of vertices, even if those vertices appear in a different order on the two tetrahedra.

The functionality of most operations should be apparent from their name. The operations on TriMesh and TetMesh implement local transformations commonly used in mesh improvement algorithms [Brochu and Bridson 2009; Klingner and Shewchuk 2008]. The full code for these examples can be found in Appendix D.

In Table 1 we report on the wall-clock time taken by the SMT solver to prove our queries. The time taken by our toolset (to process the code and produce the SMT proof obligations) is around 3s for the UnqTetMesh schema, and between 0.15s and 0.3s for each of the other schemas. We performed our measurements on a laptop with a 2GHz Intel i7-4750HQ processor, running Linux Mint 18. We used a nightly build of CVC4 from 2017-02-27, optimized for Linux x86_64 machines, running under finite-model-finding mode [Reynolds et al. 2013].

The main point to note about these measurements is that most of the solver's time is spent proving invariant maintenance, rather than basic memory safety. This is to be expected, as invariant maintenance is the only component of the proof task that requires reasoning about the final state of the data structure, after all effects have been applied, rather than only reasoning about components of the operation's effects in isolation.

8.2 Verification Discussion

The proof obligations we emit make heavy use of quantification over uninterpreted sorts, and thus the efficiency of our verification method is heavily dependent on good quantifier instantiation

Schema	Operation	Mem. Safety	Inv. Maint.
UGraph	deleteVertex	0.005s	0.008s
DiGraph	deleteVertex	0.015s	0.032s
SocialNetAsym	addFollow	0.006s	0.004s
	mergeAccounts	0.012s	0.006s
	removeAccount	0.005s	0.004s
SocialNetSymm	addFollow	0.006s	0.008s
	mergeAccounts	0.011s	0.027s
	removeAccount	0.006s	0.010s
SocialNetUnq	addFollow	0.005s	0.006s
	mergeAccounts	0.027s	0.040s
	removeAccount	0.006s	0.006s
HalfEdge	collapseEdge	0.049s	0.079s
	splitEdge	0.009s	0.065s
BasicTriMesh	collapseEdge	0.014s	0.019s
	flipEdge	0.011s	0.061s
	splitEdge	0.010s	0.045s
UnqTriMesh	collapseEdge	0.074s	0.208s
	flipEdge	0.016s	0.212s
	splitEdge	0.011s	0.292s
UnqTetMesh	addSteinerPoint	0.011s	2.722s
	splitEdge	0.014s	1.830s
	removeEdge	0.027s	26.154s
	flip23	0.053s	32.467s
	flip32	0.102s	13.196s

Table 1. Time taken by CVC4 to verify various operations, broken down into time spent verifying invariant maintenance and basic memory safety (all other properties together)

support. In particular, the finite-model-finding heuristic is a good fit for the kinds of queries that result from typical local edit operations: Such operations are centered on a specific region of the data structure, and have limited range (this is guaranteed by Seam's restricted looping construct). Therefore, it is highly likely that a small input suffices to demonstrate a potential error in the operation. By construction, finite-model-finding efficiently explores all valid instances of its domain of discourse, in order of increasing cardinality. Therefore, if a counter-example exists that disproves one of our queries, it is likely small, and finite-model-finding is likely to find it quickly.

Note, however, that the search strategy employed by finite-model-finding is heuristic-driven, and thus can provide no performance guarantees. This behavior is hinted at by the solver timings in Table 1: As schemas become more complex (i.e., the number of tables, fields and invariants increases), the dimensionality of the solver's search space increases (i.e., the solver needs to fill in more sorts and functions on each candidate instance), and the solver runtime increases accordingly.

In practice, we have found that finite-model-finding works well for our domain, and has been very useful at uncovering problematic corner cases. While writing our verification examples, we have often introduced bugs, which we were able to quickly identify using the counter-examples produced by the SMT solver.

8.3 End-to-End Application Execution Measurements

We implemented a version of the *Enright Test* [Brochu and Bridson 2009] in Seam, using Terra as the host language. This test uses a geometric flow to severely distort and then un-distort a sphere over the course of 300 computed timesteps of simulation. Remeshing is performed on every timestep. Additionally, we modified the Enright Test implementation in Brochu's ElTopo codebase [Brochu and Bridson 2009], and implemented a version of the test using the remeshing code from the TopTop codebase [Bernstein and Wojtan 2013]. Both ElTopo and TopTop are written in C++. All three versions were written or modified to remove all remeshing/improvement operations outside of edge-split and edge-collapse, which was the common denominator supported by all three. The scheduling of these operations differs between the three systems: ElTopo priority sorts an array and makes multiple passes until no more operations need to be applied; TopTop uses a priority queue; our Seam code simply makes a single pass over the edges that exist at the outset of a remeshing pass. To reduce the effect of this variance, we restricted all 3 systems to make only one pass over the edges, in arbitrary order.

All code was run on a 2015 Macbook with an Intel Core M-5Y71 processor clocked up to 1.3 GHz and more than enough RAM. Total wall-clock times for the three variations were 4.5 sec (Seam), 10.8 sec (TopTop), 1145 sec (ElTopo). Peak memory usage for the three variations was 4.9 MB (Seam), 3.7 MB (TopTop), and 6.8 MB (ElTopo). Profiling reveals that TopTop spends around 45% of its time generating a `TopoCache` data structure at the start of every remeshing pass. In terms of Seam, this is analogous to regenerating the views and indices. This data structure is then "committed" and discarded at the end of a remeshing pass. Profiling ElTopo reveals that it spends around 90% of its time in a `defrag` pass over the data at the end of every remeshing pass. Because all references to an element must be updated whenever an element is moved, this re-arrangement pass can become quite expensive. By contrast, Seam uses free-lists and strictly incremental maintenance of data structures—the data structure is fully valid after the completion of any individual remeshing operation.

The entire Seam application, including the test harness, is about 800 lines of code, 135 of which are Seam code. TopTop's data structures and application code take about 3000 lines, after slicing out unused code and support libraries. ElTopo—which is less obvious how to slice—is a library of 25,000 lines of code. We can also look specifically at the code required to write edge split and collapse in the various systems. Seam takes 22 and 33 lines respectively. TopTop takes 212 and 300 respectively. ElTopo takes 225 and 344 respectively.

What should we observe from these comparisons? First, and most importantly, existing code is not heavily performance tuned. A Seam back-end that we know contains significant performance flaws can achieve better performance than hand-written C++ code. Second, in the absence of more automated memory and indexing management, programmers are forced to make trade-offs between performance and comprehensibility in this kind of complex remeshing code. Third, this memory management and index maintenance code leaks into higher-level algorithms rather than remaining encapsulated; as a result, it becomes disproportionately difficult to change memory management or indexing strategies in pre-existing code. Fourth, and finally, Seam allows programmers to express local edit operations more concisely than general-purpose languages, by providing domain-specific primitives, and automatically generating the code that handles memory management.

9 RELATED WORK

Seam is the first language that supports sound and precise verification of local graph edits, including being able to verify memory safety and other invariants of realistic, difficult examples.

Seam uses relations to specify graphs declaratively and generates low-level imperative code. The most closely related work to Seam is "Data Representation Synthesis" [Hawkins et al. 2011], which uses relations to specify containers declaratively, and exports C++ implementations. This system is limited to a single container, and is thus less expressive than Seam, which can express multiple containers with connections between them. Much of the machinery of Seam is aimed at ensuring that pointers between containers are not left dangling, a problem that is absent in the single container setting. However, unlike "Concurrent Data Representation Synthesis" [Hawkins et al. 2012], which can specify a multi-threaded container, Seam has been designed for the sequential setting.

Domain-specific languages have been used to generate formally verified implementations of Puppet updates [Shambaugh et al. 2016], network configurations [Anderson et al. 2014], data partitioning [Treichler et al. 2016], etc. Seam is a domain-specific language for implementing local edits on graphs. Currently, developers implement edits in general-purpose languages such as C++. There are a huge number of tools that verify low-level implementations, and we cannot cover them all here. These include tools based on shape analysis [Sagiv et al. 1999], separation logic [Calcagno et al. 2009; Piskac et al. 2013], symbolic execution [Itzhaky et al. 2013; Lahiri and Qadeer 2008], interactive theorem provers [Bhargavan et al. 2017; Lamport 2002; Wilcox et al. 2015], etc. These tools fall broadly into two categories: those that perform bounded checking (e.g., Alloy [Jackson 2002], Sketch [Solar-Lezama et al. 2005], etc.) and are unsound in the presence of loops (with no fixed bound on the number of iterations), and those that are sound but require inductive invariants, which are inferred heuristically or provided manually. Sacrificing the expressiveness provided by general-purpose languages (e.g., no transitive closure [Strecker 2011]) allows Seam to verify `for` loops, with no a priori bound on the number of iterations, soundly, without the need for inductive invariants.

Seam checks both memory safety and user-provided invariants on schemas by querying SMT solvers, which provide counter-examples when the implementations are incorrect. The counter-examples are not available when verifying database transactions using refinement types [Baltopoulos et al. 2011] or weakest preconditions [Benedikt et al. 1996]. If we restrict our attention to memory safety, then languages such as CCured [Necula et al. 2002], Rust [Matsakis and II 2014], or Cyclone [Hicks et al. 2003] can help developers write memory safe code by a combination of static analysis and runtime checks. However, implementing and maintaining local edits in such languages would still require significant effort. Instead of asking programmers to write low-level code and then attempt to verify/monitor it, the Seam compiler verifies the high-level Seam programs and exports low-level code.

We note that the low-level code generated by the Seam compiler is, in fact, unverified. Although the Seam programs are verified, the Seam compiler is not a certified compiler like CompCert [Leroy 2009]. Verifying that the translation performed by the Seam compiler is correct is a translation validation problem [Pnueli et al. 1998] that is not addressed here. Furthermore, our soundness guarantees rely on the soundness of CVC4. Instead of using a general purpose SMT solver, we could have attempted to construct a specialized decision procedure, such as in [Klarlund and Schwartzbach 1993; McPeak and Necula 2005; Walukiewicz 2002]. Our reason behind using CVC4 is pragmatic; it performs well in this domain.

To the best of our knowledge, this is the first paper that performs formal verification of implementations in computational geometry. Prior work provides much weaker guarantees [Czumaj et al. 2000; Mehlhorn et al. 1996].

10 CONCLUSION & FUTURE WORK

Seam explores one way to verify and synthesize programs that manipulate data structures, by focusing on local graph edit operations. In exploring this idea, we avoided primitives that were not necessary for expressing local connectivity properties, e.g. nullable references (i.e., option types), enumeration types, tagged union types, and non-connectivity predicates, including any form of ordering operator (which means that Seam currently cannot express all properties of certain data structures, such as search trees). These features are not fundamentally incompatible with Seam's design, and we believe that Seam can be extended with some or all of them, though ensuring that verification remains efficient will be a challenge.

Conversely, the lack of general recursion is a fundamental limitation of Seam's design, which restricts the programs that can be written in the language. For example, it is impossible to express non-local updates purely in Seam. However, some variants of non-local updates (e.g., worklist-based iterative mesh improvement algorithms) can be expressed by utilizing the host language to freely traverse the data structure (in read-only mode) and invoke Seam operations repeatedly, each execution centered where appropriate. Note that this scheme recovers some expressivity without sacrificing Seam's data integrity guarantees.

Another consequence of Seam's restricted recursion is the lack of a transitive closure operator, which is necessary for expressing certain data structure invariants. In some cases, however, properties that require transitive closure can be decomposed into equivalent (or slightly stronger) local properties [Lev-Ami et al. 2005] that are expressible in Seam, potentially after the introduction of auxiliary "ghost fields" [McPeak and Necula 2005]. For example, consider the fundamental invariant of a cyclic singly-linked list: "Starting from any node n, if we follow *next* pointers, we will eventually reach n." This property cannot be directly encoded in Seam, however we can instead introduce an additional *prev* pointer on every node, and express the logically equivalent invariant "For every node n, $n.next.prev = n$."

Our implementation leaves optimization largely unaddressed. In particular, we believe that data representation synthesis ideas [Hawkins et al. 2011] could be effectively ported to this richer multi-table setting.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments and suggestions. We would also like to thank Clark Barrett and Andrew Reynolds for their help in using CVC4 effectively. This work was funded in part by grant NSF CCF-1160904. This material is based upon work supported by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1.

REFERENCES

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetkAT: Semantic Foundations for Networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 113–126.

Ioannis G. Baltopoulos, Johannes Borgström, and Andrew D. Gordon. 2011. Maintaining Database Integrity with Refinement Types. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 484–509. http://dl.acm.org/citation.cfm?id=2032497.2032530

Michael Benedikt, Timothy Griffin, and Leonid Libkin. 1996. Verifiable Properties of Database Transactions. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 117–127.

Gilbert Louis Bernstein and Chris Wojtan. 2013. Putting Holes in Holey Geometry: Topology Change for Arbitrary Surfaces. *ACM Trans. Graph.* 32, 4, Article 34 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2462027

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoué. 2017. Verified

Low-Level Programming Embedded in F*. *CoRR* abs/1703.00053 (2017).

Egon Börger, Erich Grädel, and Yuri Gurevich. 2001. *The Classical Decision Problem*. Springer Science & Business Media.

Tyson Brochu and Robert Bridson. 2009. Robust Topological Operations for Dynamic Explicit Surfaces. *SIAM J. Sci. Comput.* 31, 4 (June 2009), 2472–2493. https://doi.org/10.1137/080737617

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 289–300.

Artur Czumaj, Christian Sohler, and Martin Ziegler. 2000. Property Testing in Computational Geometry. In *European Symposium on Algorithms*. Springer, 155–166.

Fang Da, Christopher Batty, and Eitan Grinspun. 2014. Multimaterial Mesh-based Surface Tracking. *ACM Trans. Graph.* 33, 4, Article 112 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601146

Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. 2014. A Tour of CVC4: How It Works, and How to Use It. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD '14)*. FMCAD Inc, Austin, TX, Article 4, 1 pages. http://dl.acm.org/citation.cfm?id=2682923.2682928

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-stage Language for High-performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 105–116. https://doi.org/10.1145/2491956.2462166

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. Data Representation Synthesis. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1993498.1993504

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2012. Concurrent Data Representation Synthesis. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 417–428.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2003. *Safe and Flexible Memory Management in Cyclone*. Technical Report.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua—An Extensible Extension Language. *Software: Practice and Experience* 26, 6 (1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning about Reachability in Linked Data Structures. In *International Conference on Computer Aided Verification*. Springer, 756–772.

Daniel Jackson. 2002. Alloy: A New Technology for Software Modelling. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. 20.

Nils Klarlund and Michael I Schwartzbach. 1993. Graph Types. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 196–205.

Bryan Matthew Klingner and Jonathan Richard Shewchuk. 2008. Aggressive Tetrahedral Mesh Improvement. In *Proceedings of the 16th International Meshing Roundtable*. Springer, 3–23.

Christoph Koch. 2010. Incremental Query Evaluation in a Ring of Databases. In *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '10)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/1807085.1807100

Shuvendu Lahiri and Shaz Qadeer. 2008. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. *ACM SIGPLAN Notices* 43, 1 (2008), 171–182.

Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Tal Lev-Ami, Neil Immerman, Tom Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. 2005. Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In *International Conference on Automated Deduction*. Springer, 99–115.

Nicholas D. Matsakis and Felix S. Klock II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*. 103–104.

Scott McPeak and George C Necula. 2005. Data Structure Specifications via Local Equality Axioms. In *International Conference on Computer Aided Verification*. Springer, 476–490.

Kurt Mehlhorn, Stefan Näher, Thomas Schilz, Stefan Schirra, Michael Seel, Raimund Seidel, and Christian Uhrig. 1996. Checking Geometric Programs or Verification of Geometric Structures. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*. ACM, 159–165.

Rahul Narain, Tobias Pfaff, and James F. O'Brien. 2013. Folding and Crumpling Adaptive Sheets. *ACM Trans. Graph.* 32, 4, Article 51 (July 2013), 8 pages. https://doi.org/10.1145/2461912.2462010

Rahul Narain, Armin Samii, and James F. O'Brien. 2012. Adaptive Anisotropic Remeshing for Cloth Simulation. *ACM Trans. Graph.* 31, 6, Article 152 (Nov. 2012), 10 pages. https://doi.org/10.1145/2366145.2366171

George C. Necula, Scott McPeak, and Westley Weimer. 2002. Ccured: Type-Safe Retrofitting of Legacy Code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. 128–139.

Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien. 2014. Adaptive Tearing and Cracking of Thin Sheets. *ACM Trans. Graph.* 33, 4, Article 110 (July 2014), 9 pages. https://doi.org/10.1145/2601097.2601132

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *International Conference on Computer Aided Verification*. Springer, 773–789.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. 151–166.

Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. 2013. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *Proceedings of the 24th International Conference on Automated Deduction (CADE '13) (Lecture Notes in Computer Science)*, Maria Paola Bonacina (Ed.), Vol. 7898. Springer Berlin Heidelberg, 377–391. http://www.cs.nyu.edu/~barrett/pubs/RTG+13.pdf Lake Placid, NY.

J.-R. Sack and J. Urrutia (Eds.). 2000. *Handbook of Computational Geometry*. North-Holland Publishing Co., Amsterdam, The Netherlands.

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-Valued Logic. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 105–118.

Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 416–430.

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 281–294.

Martin Strecker. 2011. Locality in Reasoning about Graph Transformations. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*. Springer, 169–181.

Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent Partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 344–358.

Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer D. Widom. 2002. *Database Systems: The Complete Book*. Prentice Hall, Chapter 7.

Igor Walukiewicz. 2002. Monadic Second-Order Logic on Tree-Like Structures. *Theoretical Computer Science* 275, 1 (2002), 311–346.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368.

Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. 2009. Deforming Meshes that Split and Merge. *ACM Trans. Graph.* 28, 3, Article 76 (July 2009), 10 pages. https://doi.org/10.1145/1531326.1531382

Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. 2010. Physics-Inspired Topology Changes for Thin Fluid Features. *ACM Trans. Graph.* 29, 4, Article 50 (July 2010), 8 pages. https://doi.org/10.1145/1778765.1778787