Synthesizing Iterators from Abstraction Functions

Derek Rayside, Vajihollah Montaghami,
Francesca Leung, Albert Yuen, and
Kevin Xu
University of Waterloo
drayside@uwaterloo.ca

Daniel Jackson
MIT CSAIL
dnj@csail.mit.edu

ABSTRACT

A technique for synthesizing iterators from declarative abstraction functions written in a relational logic specification language is described. The logic includes a transitive closure operator that makes it convenient for expressing reachability queries on linked data structures. Some optimizations, including tuple elimination, iterator flattening, and traversal state reduction, are used to improve performance of the generated iterators.

A case study demonstrates that most of the iterators in the widely used JDK Collections classes can be replaced with code synthesized from declarative abstraction functions. These synthesized iterators perform competitively with the hand-written originals.

In a user study the synthesized iterators always passed more test cases than the hand-written ones, were almost always as efficient, usually took less programmer effort, and were the qualitative preference of all participants who provided free-form comments.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented programming; D.2.3 [**Software Engineering**]: Coding tools and techniques

General Terms

Languages

Keywords

iterator, Java, Alloy, abstraction functions, synthesis

1. INTRODUCTION

In many modern specification languages (*e.g.*, Larch/Modula-3 [18], JML [2, 23], Jahob [21, 50], JFSL [6, 49]) the abstract state of an ADT is described by a set of specification fields. Each specification field has an associated *abstraction function* that computes its abstract value from the concrete representation [12]. If these abstraction functions are written in a declarative logic then they are usually used for static verification (*e.g.*, Larch/Modula-3 [18], Jahob [21, 50] and JFSL [6, 49]). If these abstraction functions are written in executable imperative code then they are usually used for runtime verification (*e.g.*, JML [3]). Abstraction functions are also used for data structure repair [5] and in refinement [32].

In previous work [38] we showed that abstraction functions can be used to define object equality, and that executable abstraction functions can be used to generate executable implementations of object equality and hashing methods. These automatically generated implementations pass more object-contract compliance test cases than the implementations hand-written by expert programmers [38]. Object-contract compliance is notoriously difficult (*e.g.*, [1, 34, 35, 37, 46]).

There are many good reasons for writing abstraction functions. Yet programmers in practice seem not to write them quite as often as a researcher might hope. We conjecture that the main reason for this is that the software engineering benefits of specifying abstraction functions feel, for many programmers, somewhat removed from the task of producing running code. Therefore, we hope to tip the balance by synthesizing imperative code directly from declarative abstraction functions. Our goal is to justify specifying abstraction functions in terms of the task of producing working code. Once the programmer has abstraction functions they can then reap all of the other benefits mentioned above.

For a typical ADT implementation a one or two line declarative abstraction function can replace dozens of lines of imperative code. A number of researchers have commented on the difficulty of writing iterators by hand in Java-like languages [15, 20, 29, 30, 33].

Our user study demonstrates that, with relatively little training, it takes less effort to write declarative abstraction functions than the imperative iterators that we synthesize from them. Our user study and a case study demonstrate that the generated code is competitive with hand-written code in terms of both speed and space. The case study also demonstrates that our approach is applicable to a wide range of important cases.

Our work is related to a wide variety of other works, including executable specifications, heap query languages, query-based debugging, and heap traversal frameworks. In summary, our work is distinguished from the prior research in that our queries (abstraction functions) are written in a relational logic that includes both join and transitive closure operators and is usable for static verification. Our work therefore has greater synergy with other software engineering tasks that make use of specifications. We discuss related work in more detail below.

2. JFORGE SPECIFICATION LANGUAGE

We work with the JForge Specification Language (JFSL) for Java (*i.e.*, imperative object-oriented) programs based on the Alloy [14] relational logic. JFSL was developed to support static verification of functional properties of single-threaded programs [6, 49].

GPCE'12, September 26–27, 2012, Dresden, Germany.

This section gives a brief overview of the pertinent features of JFSL. The expressions and formulæ of JFSL are essentially Alloy. JFSL adds on top of this special annotations to connect these expressions and formulæ to code. Yessenov discusses [49] the syntax [49, §3] and semantics [49, §4] of JFSL in detail. The JFSL annotations include specification field and abstraction function definition (@SpecField), class invariants (@Invariant), method pre-conditions (@Requires and @Returns), method post-conditions (@Ensures), and method frame-conditions (@Modifies). We are primarily interested in the @SpecField annotation, which is where the programmer declares abstraction functions. The general form of the @SpecField annotation is: @SpecField(" f : r from g | α "), where f is the name of the specification field being defined, r is an expression that describes the multiplicity and type of f's range, g is f's data group [24], and α is a formula describing f's abstraction function. We do not make use of the data group g and so it is excluded from the listings and from further discussion.

The abstraction function α is a JFSL formula. It is most commonly of the form this.$f = e$, where e is some Alloy/JFSL expression. However, the abstraction function α does not need to be in this assignment-like equational form. Any valid JFSL formula will do. Another common form for α is $p(e_1, e_2, ..., e_n, \text{this}.f)$, where p is some logical predicate and e_i are JFSL expressions. We provide a library of such predicates, mostly for working with sequences, for the programmer's convenience.

In our case study we replace (almost) all manually written iterators in the JDK Collections v1.4 with iterators synthesized from declarative abstraction functions written in JFSL. The most interesting examples from this case study are TreeMap and HashMap, which are listed in Figures 1 and 2, respectively. We will use the TreeMap.entries abstraction function as our ongoing example.

In JFSL fields are modelled as binary relations. Some common JFSL/Alloy operators used in the abstraction functions for TreeMap (Figure 1) include relational join (.), reflexive transitive closure (*), union (+), and set difference (−).

```
@SpecField({
  "public entries : set Map.Entry |
                    this.entries = this.root.∗(left+right) − null",
  "public keys : set Object | this.keys = this.entries.key",
  "public values : set Object | this.values = this.entries.value" })
public class TreeMap extends AbstractMap
  implements SortedMap, Cloneable, Serializable
```

Figure 1: Spec. fields and abstraction funs for java.util.TreeMap

The specification fields and abstraction functions for HashMap are listed in Figure 2. The abstraction function for the entries specification field is the most interesting of the three. It makes use to two JFSL keywords: elems and **int**. The latter represents the set of all integers. The former is a ternary relation that maps array objects to their contents. The content of an array is modelled as a binary relation from ints to objects.

```
@SpecField({
  "public entries : set Map.Entry |
            this.entries = (int.(this.table.elems)).∗next − null",
  "public keys : set Object | this.keys = this.entries.key − null",
  "public values : set Object | this.values = this.entries.value" })
public class HashMap extends AbstractMap
  implements Map, Cloneable, Serializable
```

Figure 2: Spec. fields and abstraction funs for java.util.HashMap

3. EXECUTING NAVIGABLE EXPRS.

The abstract value of each specification field is a set or a sequence of concrete objects. We synthesize Java iterators to yield these sets or sequences. Our technique is a syntax-directed translation in which each node in the abstraction function AST is compiled into an iterator that consumes the output of the iterators corresponding to its child nodes.

We apply three optimizations to improve the runtime characteristics of the code synthesized from navigable expressions. First, we define navigable expressions so that no intermediate tuples need be constructed during their evaluation (*tuple elimination*). Our generated code is structured as a composition of streams represented as Java iterators. Our second optimization is to *flatten* iterators when the number of elements that they will return can be determined at compile time. Iterator flattening involves unrolling the iterator and inlining it into its caller. Our final optimization requires an analysis of both the abstraction function and the program to determine if the portion of the heap to be navigated by the abstraction function is tree-like or cyclic. If the portion of the heap to be traversed is always tree-like then the execution of the abstraction function does not need to remember previously visited objects.

The remainder of this section is organized as follows: an example of a synthesized iterator; definition of navigable expressions; definition of valuation functions for synthesis; descriptions of tuple elimination and traversal state reduction optimizations.

3.1 Example Generated Code

Figure 3 lists the code generated for the reflexive-closure-join (.*) node in our example abstraction function (this.root.*(left+right)−null) from java.util.TreeMap. This code has been reformatted for presentation, excess braces removed, *etc.*

Our synthesizer is written in Meta-AspectJ [13].

```
public class GeneratedIterator1 implements Iterator {
  private final Deque stack = new ArrayDeque(10);
  private TreeMap.Entry o = null;

  public GeneratedIterator1(final TreeMap.Entry obj) { this.o = obj; }

  public boolean hasNext() {
    if (o != null) return true;
    if (!stack.isEmpty()) return true;
    return false;
  }
  public Object next() {
    if (!hasNext()) throw new NoSuchElementException();
    while (o != null) {
      stack.push(o);
      o = o.left; // iterator inlined here
    }
    assert (!stack.isEmpty());
    TreeMap.Entry result = (TreeMap.Entry) (stack.pop());
    o = result.right; // iterator inlined here
    return result;
  }
```

Figure 3: Example of generated code

The iterators corresponding to the union operator (+) and field accesses (left and right) have been inlined into this code. The value expected by the constructor is the result of evaluating this.root.

Since the programmer has subtracted null at the top level there is no need for this code to return null. Our synthesizer takes advantage of this observation by removing flags and conditionals that would otherwise be present.

(a) Standard definitions

$$\begin{aligned}
&\mathbf{U} :: \text{UnaryExpr} \rightarrow \text{Heap} \rightarrow \text{Set}\\
&\quad \mathbf{U}\llbracket \mathsf{this} \rrbracket(h) &&= h(\mathsf{this})\\
&\quad \mathbf{U}\llbracket \mathsf{null} \rrbracket(h) &&= \{null\}\\
&\quad \mathbf{U}\llbracket u_1 + u_2 \rrbracket(h) &&= \mathbf{U}\llbracket u_1 \rrbracket(h) \ \cup\ \mathbf{U}\llbracket u_2 \rrbracket(h)\\
&\quad \mathbf{U}\llbracket u_1 - u_2 \rrbracket(h) &&= \mathbf{U}\llbracket u_1 \rrbracket(h) \ -\ \mathbf{U}\llbracket u_2 \rrbracket(h)\\
&\quad \mathbf{U}\llbracket u.b \rrbracket(h) &&= \mathbf{B}\llbracket b \rrbracket(h,\ \mathbf{U}\llbracket u \rrbracket(h))\\
&\quad \mathbf{U}\llbracket u.\hat{}\,b \rrbracket(h) &&= \mathbf{U}\llbracket u \rrbracket(h).\hat{}\,\mathbf{B}\llbracket b \rrbracket(h)\\
&\quad \mathbf{U}\llbracket u.{*}b \rrbracket(h) &&= \mathbf{U}\llbracket u \rrbracket(h) \ \cup\ \mathbf{U}\llbracket u.\hat{}\,b \rrbracket(h)\\
&\quad \mathbf{U}\llbracket \mathsf{int}.q \rrbracket(h) &&= \{o \mid \exists\, i \mid \langle i, o\rangle \in \mathbf{Q}\llbracket q \rrbracket\}\\
&\mathbf{B} :: \text{BinaryExpr} \rightarrow \text{Heap} \rightarrow \text{Set} \rightarrow \text{Set}\\
&\quad \mathbf{B}\llbracket b_1 + b_2 \rrbracket(h, s) &&= \mathbf{B}\llbracket b_1 \rrbracket(h, s) \ \cup\ \mathbf{B}\llbracket b_2 \rrbracket(h, s)\\
&\quad \mathbf{B}\llbracket b_1 - b_2 \rrbracket(h, s) &&= \mathbf{B}\llbracket b_1 \rrbracket(h, s) \ -\ \mathbf{B}\llbracket b_2 \rrbracket(h, s)\\
&\quad \mathbf{B}\llbracket f \rrbracket(h, s) &&= \{\langle o\rangle \mid \exists\, t \in s \mid \langle t, o\rangle \in h(f)\ \}\\
&\mathbf{Q} :: \text{SeqExpr} \rightarrow \text{Heap} \rightarrow \text{Sequence}\\
&\quad \mathbf{Q}\llbracket a \rrbracket &&= h(a)\\
&\quad \mathbf{Q}\llbracket q.f \rrbracket &&= \{\langle i, o\rangle \mid \mathbf{B}\llbracket f \rrbracket(h, i.\mathbf{Q}\llbracket q \rrbracket) = o\ \}
\end{aligned}$$

(b) Reduced-state definitions

$$\begin{aligned}
&\mathbf{U} :: \text{UnaryExpr} \rightarrow \text{Heap} \rightarrow \text{Sequence}\\
&\quad \mathbf{U}\llbracket \mathsf{this} \rrbracket(h) &&= \{\langle 0, h(\mathsf{this})\rangle\}\\
&\quad \mathbf{U}\llbracket \mathsf{null} \rrbracket(h) &&= \{\langle 0, null\rangle\}\\
&\quad \mathbf{U}\llbracket u_1 + u_2 \rrbracket(h) &&= \mathbf{U}\llbracket u_1 \rrbracket(h) \ \oplus\ \mathbf{U}\llbracket u_2 \rrbracket(h)\\
&\quad \mathbf{U}\llbracket u_1 - u_2 \rrbracket(h) &&= \mathbf{U}\llbracket u_1 \rrbracket\\
&\quad \mathbf{U}\llbracket u.b \rrbracket(h) &&= \mathbf{B}\llbracket b \rrbracket(h, \mathbf{U}\llbracket u \rrbracket(h))\\
&\quad \mathbf{U}\llbracket u.\hat{}\,b \rrbracket(h) &&= traverse(b, h, \mathbf{U}\llbracket u \rrbracket(h))\\
&\quad \mathbf{U}\llbracket u.{*}b \rrbracket(h) &&= \mathbf{U}\llbracket u \rrbracket(h) \ \oplus\ \mathbf{U}\llbracket u.\hat{}\,b \rrbracket(h)\\
&\quad \mathbf{U}\llbracket \mathsf{int}.q \rrbracket(h) &&= \mathbf{Q}\llbracket q \rrbracket(h)\\
&\mathbf{B} :: \text{BinaryExpr} \rightarrow \text{Heap} \rightarrow \text{Sequence} \rightarrow \text{Sequence}\\
&\quad \mathbf{B}\llbracket b_1 + b_2 \rrbracket(h, l) &&= \mathbf{B}\llbracket b_1 \rrbracket(h, l) \ \oplus\ \mathbf{B}\llbracket b_2 \rrbracket(h, l)\\
&\quad \mathbf{B}\llbracket b_1 - b_2 \rrbracket(h, l) &&= \mathbf{B}\llbracket b_1 \rrbracket(h, l)\\
&\quad \mathbf{B}\llbracket f \rrbracket(h, l) &&= \{\langle i, o\rangle \mid \exists\, \langle i, t\rangle \in l \mid \langle t, o\rangle \in h(f)\ \}\\
&\mathbf{Q} :: \text{SeqExpr} \rightarrow \text{Heap} \rightarrow \text{Sequence}\\
&\quad \mathbf{Q}\llbracket a \rrbracket &&= h(a)\\
&\quad \mathbf{Q}\llbracket q.f \rrbracket &&= \{\langle i, o\rangle \mid \mathbf{B}\llbracket f \rrbracket(h, i.\mathbf{Q}\llbracket q \rrbracket) = o\ \}
\end{aligned}$$

Operators and functions:

relational join: **.** transitive closure: ^ cardinality: # concatenation: $l_1 \oplus l_2 \equiv\ l_1 \cup \{\ \langle i, o\rangle \mid \langle i - \#l_1, o\rangle \in l_2\}$

$traverse(b, h, l) \equiv$ let $l' = B\llbracket b \rrbracket(h, l)$ in if $l' = \emptyset$ then $\emptyset$ else $l' \oplus traverse(b, h, l')$

Variable naming conventions:

u	unary expression	f	field expression	q	sequence expression	s	set of objects
b	binary expression	a	array expression	l	sequence of objects	h	heap

Figure 4: Valuation functions

$$\begin{aligned}
&\mathbf{I}\llbracket u_1 + u_2 \rrbracket &&= \forall\, h \mid (\mathbf{U}\llbracket u_1 \rrbracket(h)\ \cap\ \mathbf{U}\llbracket u_2 \rrbracket(h)) = \emptyset\\
&\mathbf{I}\llbracket u_1 - u_2 \rrbracket &&= \forall\, h \mid (\mathbf{U}\llbracket u_1 \rrbracket(h)\ \cap\ \mathbf{U}\llbracket u_2 \rrbracket(h)) = \emptyset\\
&\mathbf{I}\llbracket \mathsf{int}.q \rrbracket &&= \forall\, h \mid \nexists\, i, j \mid i \neq j \wedge i.\mathbf{Q}\llbracket q \rrbracket(h) = j.\mathbf{Q}\llbracket q \rrbracket(h)\\
&\mathbf{I}\llbracket b_1 + b_2 \rrbracket &&= \forall\, h \mid (\mathbf{B}\llbracket b_1 \rrbracket(h)\ \cap\ \mathbf{B}\llbracket b_2 \rrbracket(h)) = \emptyset\\
&\mathbf{I}\llbracket b_1 - b_2 \rrbracket &&= \forall\, h \mid (\mathbf{B}\llbracket b_1 \rrbracket(h)\ \cap\ \mathbf{B}\llbracket b_2 \rrbracket(h)) = \emptyset\\
&\mathbf{I}\llbracket u.{*}b \rrbracket &&= \mathbf{I}\llbracket u.\hat{}\,b \rrbracket\\
&\mathbf{I}\llbracket u.b \rrbracket &&= \forall\, h \mid \nexists\, o \mid \exists\, p, q \mid p \neq q\ \wedge\ (p \cup q) \in \mathbf{U}\llbracket u \rrbracket(h)\ \wedge\ (\langle p, o\rangle \cup \langle q, o\rangle) \in \mathbf{B}\llbracket b \rrbracket(h)\\
&\mathbf{I}\llbracket u.\hat{}\,b \rrbracket &&= \forall\, h \mid (\nexists\, o \mid \exists\, p, q \mid p \neq q \wedge (p \cup q) \in (\mathbf{U}\llbracket u \rrbracket(h) \cup \mathbf{U}\llbracket u \rrbracket(h).\hat{}\,\mathbf{B}\llbracket b \rrbracket(h))\ \wedge\ (\langle p, o\rangle \cup \langle q, o\rangle) \in \hat{}\,\mathbf{B}\llbracket b \rrbracket(h)) \wedge\\
& &&\qquad (\nexists\, o \mid \langle o, o\rangle \in \hat{}\,\mathbf{B}\llbracket b \rrbracket(h) \wedge \exists\, x \mid x \in \mathbf{U}\llbracket u \rrbracket(h) \wedge \langle x, o\rangle \in \hat{}\,\mathbf{B}\llbracket b \rrbracket(h)) \wedge\\
& &&\qquad (\nexists\, p, q \mid p \neq q \wedge (p \cup q) \in \mathbf{U}\llbracket u \rrbracket(h) \wedge \langle p, q\rangle \in \hat{}\,\mathbf{B}\llbracket b \rrbracket(h))
\end{aligned}$$

Figure 5: Optimization condition generation (quantification is over heaps that can be constructed by the program)

3.2 Navigable Expressions

Navigable expressions are a subset of the JFSL expression grammar in that start from the receiver object (this), traverse fields only in the forwards direction, and result in either a set or a sequence.

Figure 6 presents the grammar of navigable expressions. Productions are classified according to the arity of their output (unary or binary) because the computation is different. The computed result of a navigable expression is always a set or a sequence, and never a binary relation. This restriction is important for enabling the tuple elimination optimization.

The productions for transitive closure are coupled with a join: .^ (join-transitive-closure) and .* (join-reflexive-transitive-closure). This coupling enables the tuple elimination optimization and ensures that the result of a navigable expression is a set or a sequence.

3.3 Valuation Functions

Figure 4 gives the valuation functions for our translation. Figure 4 a gives the standard definitions, and Figure 4 b gives the definitions with reduced state (explained below).

The heap is modelled as a function from field names to binary relations that map objects to the field's value in that object. For simplicity of the formalism names are treated as global here. Sequences are modelled as binary relations mapping consecutive integers to objects, starting at 0.

```
NavigableExpr  ::=  UnaryExpr
                |   SeqExpr
UnaryExpr      ::=  this
                |   null
                |   UnaryExpr + UnaryExpr
                |   UnaryExpr − UnaryExpr
                |   UnaryExpr.BinaryExpr
                |   UnaryExpr.^BinaryExpr
                |   UnaryExpr.*BinaryExpr
                |   int.SeqExpr
BinaryExpr     ::=  FieldExpr
                |   BinaryExpr + BinaryExpr
                |   BinaryExpr − BinaryExpr
SeqExpr        ::=  ArrayExpr
                |   SeqExpr . FieldExpr
```

Figure 6: Grammar of navigable expressions

3.4 Tuple Elimination

Our first optimization is *tuple elimination*, a form of deforestation [47]. Consider our ongoing example: this.root.*(left+right)−null. The tuples for the relations root, left, and right exist as references in the heap. However, the binary relation denoted by the union (+) operator does not already exist in the heap. The idea of deforestation [47] is to avoid explicitly constructing such intermediate values.

We have defined navigable expressions so that tuple elimination for these intermediate binary nodes is always possible. The key insight is that the only unary AST nodes with binary children involve joins and the only binary AST leaves are fields (the tuples for which already exist in the heap as references). Therefore we can push the computation of the joins down into the leaf nodes.

Tuple elimination is reflected in the valuation functions of Figure 4 a in two ways. First, in the type of function **B**, which is BinaryExpr → Heap → Set → Set, rather than BinaryExpr → Heap → BinaryRelation, suggesting that we might not need to explicitly construct the tuples of the binary relation, but instead provide the left column of the relation as an input and have only the right column as the output. Second, in the function for join (**U**⟦u.b⟧(h)), where the value of the unary expression *u* is used as an input in the evaluation of the binary expression *b*. With tuple elimination, the result of a binary union (the + in our ongoing example) is the unary union of the output of its children (left and right).

3.5 Traversal State Reduction

The heap of an object-oriented program can, and often does, involve both directed and undirected cyclic references. Consequently, the traversal of the heap to accumulate the results of a navigable expression may need to remember which objects have been visited.

However, in some cases we know statically that the traversal will never encounter the same object twice, as in our java.util.TreeMap example. If the code of java.util.TreeMap never constructs cyclic heaps then the traversal implied by our ongoing abstraction function will never visit the same object twice. In other words, if the code of TreeMap respects some invariants then the abstraction function traversal will never visit the same object twice.

Figure 5 shows how to derive the desired invariants, which we call *optimization conditions*, from the abstraction function. These optimization conditions may be discharged in a variety of ways. We generate the optimization conditions in JFSL/Alloy to support static verification and also as imperative Java to support runtime verification.

There are three matters of interest: (1) how this state-reduction changes the computations described above; (2) how the optimization conditions are generated from the abstraction function; and (3) establishing that if the optimization conditions hold then the state-reduced valuations may be substituted for the normal valuations.

Reduced-State Computations.

Figure 4 b lists alternative definitions for the valuation functions of Figure 4 a. There are a number of differences:

- set union has been replaced by sequence concatenation;
- set difference is simply the minuend;
- the computation of expressions of the form int.*q* assumes that all of the objects in the sequence are unique;
- we introduce the helper function *traverse*, which can be a simple tree traversal in any order (*e.g.*, in-order, depth-first, breadth-first, *etc.*).

It is straightforward to implement these reduced-state computations in a lazy manner.

Optimization Condition Generation.

The optimization conditions generated for each production in the navigable expression grammar are listed in Figure 5. These optimization conditions are quantified over heaps that can be constructed by the program. In other words, these optimization conditions are heap invariants.

Many of the optimization conditions are fairly simple. For example, for a union production the optimization condition says that the intersection must be empty.

The optimization condition for join is slightly more complicated. It says that there are no two distinct source objects *p* and *q* that lead us to the same target object *x*.

The optimization condition generation patterns for join-closure and join-reflexive-closure are the same, and are complicated. Again, the intuition is to ensure that the navigation is tree-like. There are three conditions: (1) there are no two distinct source objects that lead to the same target object; (2) there are no reachable cycles; (3) the navigation does not enter the tree at two hierarchically related positions. The first two conditions establish that the relation on the right-hand side is tree-like. The third condition checks that the left-hand side enters that tree appropriately. We learned about this third condition through the soundness analysis discussed next.

Soundness of Optimization Condition Generation.

How do we know that the optimization conditions described by Figure 5 are sufficient to ensure that the computations given in Figure 4b will produce the same results as the ones given in Figure 4a? We used a number of Alloy models:

- the syntax of navigable expressions;
- the semantics of navigable expressions;
- the optimization condition generation;
- a *forested* model of the reduced-state computations, in which intermediate tuples are present;
- a *deforested* model of the reduced-state computations, in which the intermediate tuples have been eliminated.

The analysis searches for a field abstraction function AST and a program heap (constrained by the optimization conditions for the given field abstraction function) for which the reduced-state computation is inconsistent with the semantics. The model contains an Alloy rendition of the navigable expression grammar, and the analysis uses that to construct possible abstraction functions. No counterexample is found, giving support to the claim that that reduced-state traversals produce semantically correct results when the optimization conditions hold. The specific checks that we performed with the Alloy Analyzer were:

- the forested computation model is consistent with the semantics for every node in the abstraction function AST;
- the deforested computation model is consistent with the semantics for every unary node in the abstraction function AST, which includes the root node;
- the forested and deforested models always produce the same ultimate results for every abstraction function and program heap.

The Alloy Analyzer performs a bounded analysis. We checked that optimization condition generation is sound for all abstraction functions with up to eight nodes in their AST, which is large enough for the examples in our case study.

4. CASE STUDY: JDK COLLECTIONS V1.4

We evaluate the expressiveness and performance of our approach on a subset of the standard Java libraries, the JDK Collections v1.4. This data-structure library has a high density of interesting abstraction functions. Figure 7 lists the five abstract types in the JDK Collections v1.4 that we are interested in, counts their subtypes, and names their specification fields.

Type	Subtypes			Specification Fields
	Abstract	Concrete	Total	
List	2	14	16	values
Set	2	22	24	elts
Map	2	13	15	entries, keys, values
Map.Entry	0	9	9	key, value
Iterator	5	29	34	n/a
Total	11	87	98	

Figure 7: Count of interesting classes in JDK Collections v1.4

We were able to automatically synthesize 25 out of the 29 concrete iterators in the JDK Collections v1.4 from abstraction functions written in JFSL. The remaining cases that our approach did not cover are:

- Two map implementations do not have instantiated Entry objects. Our synthesizer produces iterators that return objects that already exist: our iterators do not instantiate new objects. Affected specification fields are: IdentityHashMap.entries and Collections.UnmodifiableMap.UnmodifiableEntrySet.elts.

- Collections.UnmodifiableCollection.iterator() is a wrapper defined in terms of the code of the wrapped object rather than in terms of a specification field.

- TreeMap.SubMap.entries: The TreeMap class requires that its keys be drawn from a totally ordered domain, such as strings or integers. Consequently, it supports an operation to create a 'submap' comprising only entries whose keys are within some lower and upper bound in the key domain. Making the determination that a key falls within the selected range requires calling a Comparator object, and that is beyond the abilities of our current synthesizer.

- TreeMap.SubMap.keySet.iterator(): Normally this inherits the implementation from AbstractMap.KeySet. Our synthesized code for AbstractMap.KeySet works for every other subclass of AbstractMap, but because of the issue with SubMap discussed in the previous bullet point the synthesized code does not work for SubMap.

Regression Test Results. We tested our synthesized iterators with the JCK and with a test suite generated by the Randoop [35] tool. Our synthesized iterators do not yet support the remove method and also do not yet throw ConcurrentModificationException.

All of the 20,661 Randoop test cases passed except for those that failed because the remove method is not supported.

Of the 1261 JCK test cases we ran, 1212 passed; 19 failed because the remove method is not supported; 27 failed because our iterators do not throw ConcurrentModificationException; and 3 failed for other reasons. JCK test cases BitSet2011 and Vector2037 failed due to hard-coded hash code and toString values, respectively.

JCK test case IdentityHashMap2027 failed for a more interesting reason. The representation of IdentityHashMap does not explicitly distinguish between absent values and null values, and so our abstraction function did not properly capture the programmer's intent in this case. Interestingly, the representation of IdentityHashMap does explicitly distinguish between null and absent keys by using a special token. We think the best resolution here would be to change the code to use the special token approach for values as well.

Optimization Condition Discharge. In order to synthesize code with the traversal state reduction optimization we need to know that the optimization conditions hold: *i.e.*, we need to know that the traversals are tree-like. We discharged the optimization conditions dynamically by synthesizing code that checks the optimization conditions and then injecting that code at public method boundaries. We ran these instrumented versions of the JDK collections with both the JCK and the Randoop tests. All of the optimization conditions for all specification fields passed except for the specification field Map.values. A map may have two distinct keys that map to the same value: *e.g.*, $\langle a, x\rangle$ and $\langle b, x\rangle$. Abstraction functions for Map.values therefore cannot use the state reduction optimization and must dynamically track visited objects.

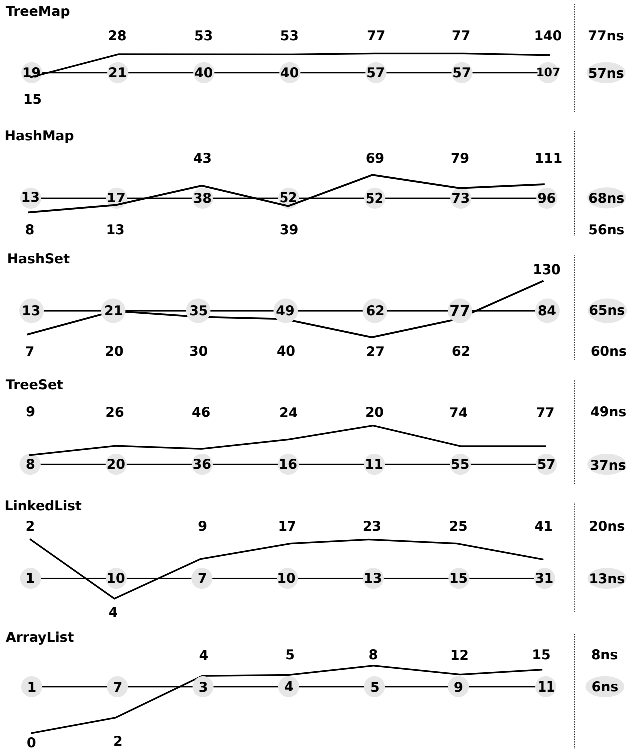

Figure 8: Performance comparison of synthesized iterators (varying lines) and original hand-written iterators (straight lines) for common JDK collections classes. Numbers are times in milliseconds. Measurements taken for collections of various sizes: 100k, 250k, 500k, 750k, 1M, 1.5M, 2M. Summary column at right shows median time per element in nanoseconds.

4.1 Performance

Figure 8 presents performance measurements for our synthesized iterators on the six most common JDK collections classes. Measurements were taken on a 3.4GHz quad-core i7 with 16GB of physical RAM, running JDK 1.6 with a 1GB heap. No other users were logged in, and no other computationally intensive processes were running. Times in the figure were produced in the following way: after a warmup run, seven raw measurements were taken; then the best and worst were discarded, and the displayed number is the arithmetic mean of the middle five. A different selection of random integers was used for each of the seven raw measurements.

The horizontal lines in Figure 8 visualize the time taken by the hand-written iterators that ship with the JDK. These times are the baseline for comparison. The variables lines in Figure 8 visualize the time taken by our synthesized iterators. If the variable line is below the straight line then our code is faster; if the variable line is above the straight line then our code is slower. The numbers in Figure 8 are times in milliseconds. The data points in Figure 8 are for different sizes of collections: 100k, 250k, 500k, 750k, 1M, 2M random integers.

The summary column on the right hand side of Figure 8 gives the median time per element in nanoseconds. In other words, a characterization of how long it takes to call the next method. The original hand-written iterators have a lower per-element times in four out of the six cases. Our synthesized code has lower per-element times for HashMap and HashSet.

We draw two main conclusions from this performance analysis: the synthesized code scales to large inputs; and the synthesized code runs competitively with the hand-written code. Sometimes the synthesized code is faster, sometimes the hand-written code is faster. In either case the difference is not a large factor. For a performance critical iterator it probably makes sense to specify the abstraction function first and use the synthesized iterator as a benchmark both for performance and for correctness. In less performance critical contexts the potential advantages of implementing by hand may not be cost-effective.

5. USER STUDY

We asked a group of programmers to write both JFSL/Alloy abstraction functions and a Java iterator for a b-tree representing a set of integers. Our system synthesized iterators from their abstraction functions. Let J_p name the iterator written by hand in Java by participant p. Let A_p name the abstraction function written in JFSL/Alloy by participant p. Let S_p name the iterator synthesized by our system from A_p. In the subsequent text we will often drop the p subscript when speaking of a participant in general.

We used tests to assess the correctness of both the J and S iterators. We gave the participants a set of tests, U, and a test harness to run them. We kept another set of tests, V, in reserve to run after the users had submitted their work (J and A).

We summarize the results (Figures 10, 11, and 9), then explain the experimental methodology, and then detail the results.

The main objective result of the study is that despite years of experience programming in Java (Figure 9) and some previous experience implementing iterators (Figure 9), no participant was able to write an iterator J that passed all test cases (Figure 10). By contrast, with less than one day of training (Figure 9), every participant wrote an abstraction function A that passed all tests (namely, the iterator S, synthesized from A, passed all tests).

The second objective result of the study (Figure 11) is that only one user wrote an iterator J that was more efficient than our synthesized iterator S; two users wrote less efficient code by hand J than we synthesized; the remaining participants achieved the same efficiency as the synthesized code.

The third objective result of the study is that the time it takes to write J or A varies by individual: some are faster with one and some are faster with the other (Figure 9). The average of our participants came out about the same.

The qualitative results of the study are (1) most participants preferred to write the abstraction function A rather than the iterator J, once they got over the learning curve; and (2) participants found the usability of our research prototype to be lacking.

	Tests Failed					
	Hand-written Java J			**Synthesized Iterator** S		
User	Basic	Empty	Orders	Basic	Empty	Orders
P28		×			*all tests passed*	
P31			2			
P22	×	×	2–7			
P01		×	2			
P00	×	×	2–7			
P29			2,6,7			
P24	×		2–5			
P30		×				

Figure 10: Every hand-written iterator fails some test. Every synthesized iterator passes all tests. The *basic* tests are the ones we provided to paricipants. The *empty* test is an empty b-tree. The *orders* column provides results when varying the fan-out of the b-tree.

	Space Efficiency	
User	**Hand-written Java** J	**Synthesized Iterator** S
P28	$O(n)$	$O(\log n)$
P31	$O(n)$	$O(\log n)$
P22	$O(\log n)$	$O(\log n)$
P01	$O(\log n)$	$O(\log n)$
P00	$O(\log n)$	$O(\log n)$
P29	$O(1)$	$O(\log n)$
P24	$O(\log n)$	$O(\log n)$
P30	$O(\log n)$	$O(\log n)$

Figure 11: Only one programmer wrote code that was more space-efficient than the synthesized code. Two programmers wrote code that was less space-efficient than the synthesized code.

5.1 Methodology

The participants were all senior undergraduate software engineering students in their 4A (penultimate) term. These students have each had five industrial internships (20 months total) in addition to their schooling, as part of Waterloo's co-operative education program. The participants reported internships at companies such as Amazon, Google, RIM, Sybase, *etc.* Participant 22, for example, has had three internships at Google (12 months total).

All of the participants have studied binary trees and b-trees in previous courses. None of the participants had previous knowledge of abstraction functions. Experience with Java ranged between two and five years, with an average of four years (Figure 9). All participants had previously used iterators; only some participants had previously written an iterator (indicated with '+I' in Figure 9). Participant 29 could not remember if she had ever written an iterator (indicated with '?I' in Figure 9).

All of the participants were taught Alloy in a course they were enrolled in. This training occurred about a month before our study. The course included two hours of lecture on Alloy and a group assignment on spanning trees that was significantly harder than our user study task and was expected to have taken about four hours.

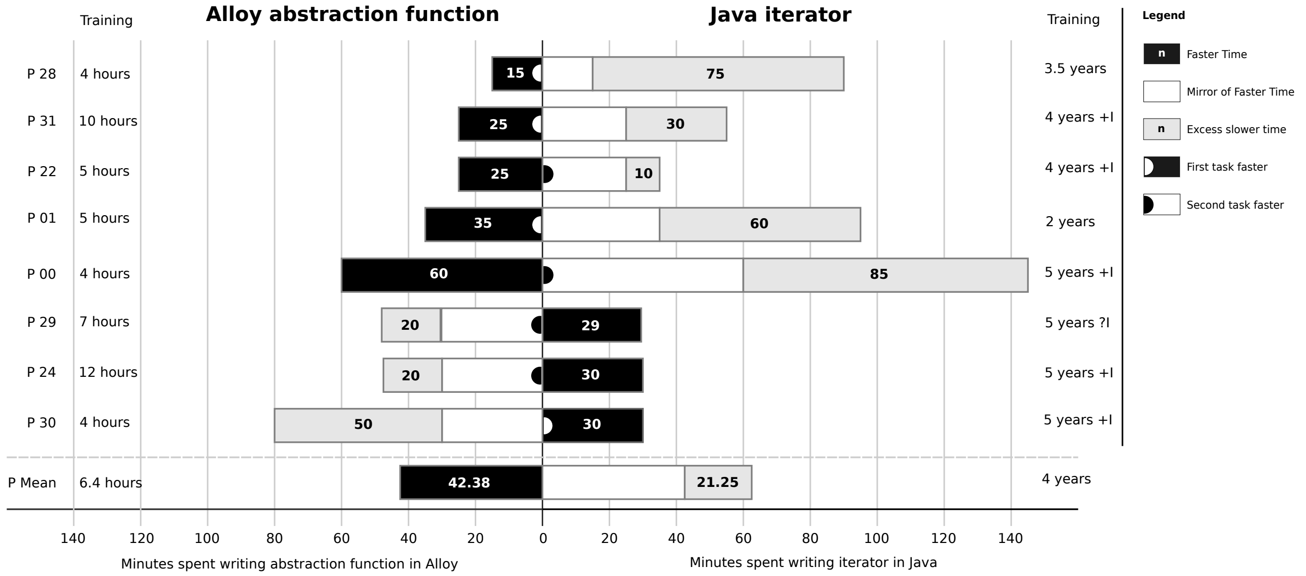

Figure 9: Time on task with each approach. Some participants were faster implementing Java iterator *J*; others were faster specifying JFSL/Alloy abstraction function *A*. On average, the times are not significantly different: there is substantial variation between individuals that is not explained by the technology used, nor previous experience with that technology, nor the order in which the tasks were completed.

The mean of 6.4 hours (Figure 9) of previous Alloy experience is consistent with this expectation.

Participants were provided with the following materials: a half-page introduction to specification fields and abstraction functions; example iterator implementations and abstraction functions for SingletonIntSet and ArrayIntSet classes; an Alloy quick-reference sheet; a book on Alloy [14]; and a data-structures textbook [4].

Participants were also provided with the b-tree source code, pictures of b-tree instances, traversal pseudo-code, and an inefficient iterator implementation. This inefficient iterator traverses the b-tree and adds each element (integer) to a HashSet and then returns that HashSet's iterator. The b-tree source code was adapted from the Java LDAP project (`http://sf.net/projects/javaldap`).

Participants were provided with a test harness and some basic test cases (U). The harness ran these tests (U) on both J and S. One test populated the b-tree with the integers from 1 to 10. The other test populated the b-tree with a random set of ten integers.

The participants worked on their own laptops inside a virtual machine image that we had prepared with the Eclipse IDE and all of the necessary software.

Participants were provided with specification field declarations for the b-tree. Their task was to write the associated abstraction functions for those specification fields. For class BTNode participants were expected to write the following expression (italics):

children : **set** BTNode | this.children = *int.(this.btnArray.elems)*

For class BTreeIntSet participants were expected to write the following two expressions (given in italics):

nodes : **set** BTNode | this.nodes = *this.root.*children*

values : **set** Integer | this.values = *int.(this.nodes.kArray.elems)*

Before attempting the b-tree exercise participants were required to complete an analogous exercise with a binary tree to get them familiar with writing abstraction functions, implementing iterators, and the tool environment. We adapted the binary tree implementation used for this warmup exercise from Kodkod [44].

Participants were allowed to choose whether they preferred to specify the abstraction functions A first or implement the iterator J first. These preferences are indicated by the semi-circles on the timing bars in Figure 9. Five of the participants specified abstraction functions A first and three implemented the iterator first J. Half of the participants were faster at the task they performed first and half were faster and the task they performed second.

Participants were compensated for their time with bonus marks in a course. The course had 74 students enrolled, divided into 20 project groups of 3-4 students each. Five of these groups elected to conduct user studies for their project. Students in the course could choose to spend their time participating in as many of these studies as they wanted to (except for any study that they were conducting). All participants in all studies were compensated at the same rate. In total, 49 students in the course participated in the studies: 22 participated in exactly one study; 15 in two studies; 7 in three studies; 3 in four studies; and 2 in all five studies. Participants were compensated for their time in a study even if they left the study partway through (as 9 of the 17 participants in our study did).

5.2 Results

We discuss four results: correctness (Figure 10), efficiency (Figure 11), time on task (Figure 9), and qualitative feedback.

5.2.1 Correctness

Figure 10 details the test cases that were passed and failed by hand-written iterators J and synthesized iterators S: the latter passed all tests; whereas every hand-written iterator failed some test.

The test cases are partitioned into three groups in Figure 10: *basic*, *empty*, and *orders*. The basic tests are the ones we provided to the participants: one test adds ten sequential integers (0–9) to the b-tree, while the other test adds ten random integers. Three participants submitted code J that failed these basic test cases.

We ran the empty and order tests after the participants had completed their tasks. The empty test checks that an iterator on an empty tree has no elements. Five out of eight of the hand-written iterators J failed this test. The order tests add ten sequential or ten random integers to the b-tree, but first set the 'order' of the tree to a different value. 'Order' is sometimes used to refer to the number of keys or number of children of a b-tree node: different authors use the term differently. In the Java LDAP b-tree implementation, a node may have up to $2k$ children and $2k - 1$ keys, where k is the order of the tree. The basic tests we provided to the participants set the tree order at 3. Our orders tests exercised orders 2 through 7. Six out of eight hand-written iterators J failed at some order. The set of orders that an implementation failed on seems to vary, indicating that there might be a number of corner cases to consider and that each programmer only considered a subset of these cases.

No participant wrote extra test cases. We did not ask them to do so. Doing so would have increased their times.

5.2.2 Code Efficiency

Our synthesized iterators S for the b-tree use $O(\log n)$ space to store their position in the tree. As shown in Figure 9, this is the same space used by most of the hand-written iterators J. Two participants, 28 and 31, used more space: $O(n)$. Only one participant, 29, used less space: $O(1)$.

The space used by the participants' iterators J varied substantially. Most participants wrote iterators J that used $O(\log n)$ space: *i.e.*, their iterators J used a stack to remember their position in the tree, which is what our synthesized iterators S do for this b-tree implementation.

Only Participant 29 wrote an iterator J_{29} that used less space. She had the insight that the parent pointer could be used to traverse up the tree, and so the stack is not necessary: a constant amount of space is all that is required. This insight is beyond the current capabilities of our synthesizer.

Participants 28 and 31 wrote iterators J that used linear space (*i.e.*, more than our synthesized code). Participant 28's J_{28} eagerly traversed the entire tree and stored it in a list, in spite of the fact that we had already provided the participants with an implementation along these lines and told them to write something more efficient. Participant 31's J_{31} encoded the state of the traversal in the nodes themselves by adding fields to the nodes to indicate whether they had been visited already. This design consumes extra space even when an iteration is not in progress and it prevents multiple iterators from operating simultaneously.

5.2.3 Task Times

Figure 9 visualizes the time taken by each participant for each task (A and J). Five of the eight participants were faster at writing the abstraction function A, and three were faster at writing the Java iterator J. The time taken to specify A ranged from 15 to 80 minutes, with a mean and median of 42 minutes. The time taken to implement J ranged from 29 minutes to 145 minutes, with a mean of 63 minutes and a median of 45 minutes. The mean, median, minimum, and maximum times for specifying A are all less than the corresponding times for implementing J. A histogram analysis (not shown) concludes that the task times are normally distributed.

The semi-circles in Figure 9 indicate which task the participant performed first. There is no correlation between which task the participant performed first and which task they performed fastest.

The main conclusion that we draw from this task time data is that there is substantial variation amongst programmers: task time is not strongly correlated with any of the other measurements we have made. This inconclusive task timing result is in contrast to the categorical results we observed in terms of the correctness and efficiency of the code J written by the participants and the synthesized code S derived from the programmer's abstraction function A. In other words, the choice of technology does not have strong predictive value for how long it will take a programmer to produce a first draft, but it does strongly predict whether that first draft will be correct and efficient.

5.2.4 Qualitative Feedback

Participants were asked to complete a questionnaire that included a free-form question at the end: "What are your general comments about this user study?" All eight participants who completed the study also completed the questionnaire. Seven of the participants who quit the study before completing the b-tree exercise submitted a questionnaire. We report on the qualitative impressions of all fifteen of these participants, since they all had some qualitative experience with the ideas and tools.

Eight of these fifteen made a comment indicating that they were not as familiar with JFSL/Alloy and how it mapped to Java as they would like. Verbally, but not in writing, some participants also confessed to us that they had not properly read the background material that we provided, and that if they had done so that would have saved them some time specifying the abstraction function A.

Six participants complained about the prototype nature of our synthesizer: its poor error messages and lack of debugging support. Four participants complained about the environment we provided: the virtual machine, build process, IDE, *etc.*

Three participants complained that the study took too long, including one participant who completed the study.

Participant 01 was concerned that "there is a little too much magic going on", but also said that the abstraction functions A were easier to write than the iterators J. Participant 27 also expressed concerns about not knowing what is going on behind the scenes, but also said: "The problem with coding in Java is that there is a lot of things that can go wrong. Even for a simple binary tree iterator, it will take at least a few tens of lines. Error prone."

All seven participants who commented on the relative ease of specifying abstraction function A versus implementing iterator J (including participants who were faster at writing the Java code) said that the Alloy was easier once they understood how it worked. Participant 22 captured the general sentiment: "I do find the Alloy easier, once I see it and understand it. It's easy to write inefficient Java code; harder to write it efficiently."

6. RELATED WORK

Synthesis from relational specifications. The RelC system [9] takes as input a relational specification, an optional decomposition specification, and a library of base data structures; it produces as output an implementation of the relational specification in terms of the base data structures. Our system is complementary to RelC: our system takes as input a base data structure and a relational abstraction function, and produces as output an iterator over that base data structure. The RelC system produces code that calls this iterator. The relational specification language accepted by RelC is richer than the language accepted by our system, but our system produces code that navigates individual pointers whereas RelC produces code that manipulates the API's of the base data structures.

Jennisys [25] synthesizes programs from specifications (including specification fields and abstraction functions) using program verification technology. Our technique uses this kind of heavyweight technology only for optimization: in the absence of verification tools our technique will still synthesize executable code.

Specification execution without synthesis. Some recent work uses Kodkod/SAT to execute partial specification on concrete inputs [31, 39, 41]. This approach does not synthesize source code.

This approach can be very competitive with hand-written code for algorithmically complex problems, but usually does not scale to large inputs. We synthesize source code that scales to large inputs for algorithmically simple problems. Our approach and the SAT-based execution approach are therefore complementary.

Executing algebraic specifications. There is a long history of executing algebraic specifications by term re-writing (*e.g.*, [7, 10, 11, 16, 17, 43]). This is related to our work in that it has to do with executing specifications, but the kind of specification language used is different and the synthesis technique is different.

Using specifications to improve program performance. Our static cycle detection optimization uses specifications to improve program performance. This kind of approach appears uncommon.

Vandevoorde and Guttag [45] proposed a system where the programmer would write two implementations of a given method: a general purpose one and a specialized one that performed faster if extra pre-conditions held. The programmer would also write specifications for these two implementations, giving the extra pre-conditions for the specialized implementation. The system would then analyze each call site to determine if the faster implementation could be used at that site. In this approach the programmer writes two implementations and two specifications, and the system does not generate any code. In our approach the programmer writes only one specification, and the system analyses the specification and the program to determine if it can generate more efficient code.

Dynamic detection of violated invariants. Demsky et al. [5] develop their own specification language for expressing data structure invariants. Their system monitors the invariants at runtime, and if it detects a violation it executes a repair action. Monitoring the invariants requires executing the abstraction functions. While this language has sets and binary relations, it does not have any of the usual set operations such as union or intersection; it also does not have transitive closure. In their language, abstraction functions are expressed as a set of rules, where each rule has a quantifier, a guard, and a consequence. The rules are evaluated dynamically until a fixed-point is reached. One gets the effect of union, intersection, closure, *etc.*, through these fixed-point semantics.

They perform three optimizations in their dynamic evaluation of the abstraction functions: determining when the fixed-point can be computed in a single pass, and two deforestations (which they call 'relation elimination' and 'set elimination'). Their relation elimination deforestation is analogous to our tuple elimination. Their set elimination deforestation has a similar result as our static cycle detection, but achieves this end in a different way. Their fixed-point elimination optimization is obviously particular to their specification language. We note that the linked data structure abstraction functions that we discuss in this paper would not be candidates for this fixed-point elimination optimization.

Heap traversal frameworks. The most important difference between our work and heap traversal frameworks such as Demeter [28], DiSTiL [42], and Scrap Your Boilerplate [22] is that with our system the programmer uses a language designed for verification, rather than with a language designed for data structure traversal. Our approach, therefore, has synergy with other software engineering tasks that make use of specifications.

Heap query languages & query-based debugging. Our work is related to heap query languages such as JQL [48], FQL [36], and DeAL [40]; and to query-based debugging [26, 27]. An important difference with our work is that most of these query languages do not include a transitive closure operator. Much of the power, simplicity (for the programmer), and complexity (for synthesis) of our approach comes from Alloy's transitive closure operators.

DeAL [40] is an exception, as it includes reachability. DeAL's reachability facility computes reachability through any path in the heap, whereas our approach follows only paths described by the programmer in the abstraction function. There are some properties, such as object ownership, that are better handled by DeAL's approach. Most abstraction functions, however, require the programmer to specify the paths more precisely.

On the implementation side, the evaluation of DeAL predicates piggy-backs on the garbage collector. This has the very nice property of making the evaluation of the predicates almost free at runtime. It also means that DeAL gets to use markbits for free (since the GC needs to use markbits anyways). Therefore, our traversal state reduction optimization is not relevant for DeAL. On the other hand, DeAL predicates are evaluated against the entire heap, whereas the iterators we synthesize typically look at only a small subset of the heap.

Database query optimization. There is a vast literature on query optimization in the database community. Two important issues in this area are finding the appropriate tuples in a relation and deciding on the direction in which to evaluate joins. These are not concerns for us. We have defined navigable expressions so that we always perform joins in the direction of the field references in the heap, and we are only performing joins on objects that we have a handle on (we never have to go searching through the heap looking for a particular object).

7. CONCLUSION

Writing iterators in Java-like languages is hard [15, 20, 29, 30, 33]. We have developed a technique for synthesizing iterators from declarative abstraction functions written in Alloy, a first-order relational logic with transitive closure.

Three optimizations were used to make the synthesized code competitive with hand-written code: tuple elimination, iterator flattening, and traversal state reduction. Traversal state reduction involves generating putative invariants (*optimization conditions*) from the abstraction function. We generate these optimization conditions as logical formulæ and as executable code.

A case study of the widely used JDK Collections library demonstrated that our approach is applicable to a wide range of practically important cases and that the performance of the synthesized code is competitive with hand-written code.

In a user study writing the declarative abstraction function and having the iterator synthesized always produced code with fewer errors; almost always produced equivalently efficient code (some people write more efficient code, but more people write less efficient code); usually took less programmer effort; and was the qualitative preference of all participants who provided feedback.

With a few hours of logic training and our synthesizer programmers can be more productive writing specifications than they are writing iterators by hand. Specifying a declarative abstraction function is a more efficient way to write an iterator.

Acknowledgments

We thank Kuat Yessenov and Greg Dennis for their assistance with JFSL. We thank Jonathan Edwards, Aleksandar Milicevic, James Noble, Martin Rinard, Steve Ward and the anonymous referees for helpful comments on earlier drafts. We thank Jacques Carette, Krzysztof Czarnecki, Brian Demsky, Bernd Fischer, Alec Heller, Felix Klock, Joseph P. Near, Viera Proulx, Yannis Smaragdakis, and Derek Thurn for constructive discussions. Gary Leavens, Rustan Leino, and Carroll Morgan were generous with their time in helping us to track down the origin of specification fields [18].

References

[1] J. Bloch. *Effective Java*. Addison-Wesley, Reading, Mass., 2001.

[2] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[3] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, Apr. 2003.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2nd edition, 2001.

[5] B. Demsky, C. Cadar, D. Roy, and M. C. Rinard. Efficient specification-assisted error localization. In *WODA*, 2004.

[6] G. Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, MIT, 2009.

[7] A. Gesar, H. Hussmann, and A. Muck. A compiler for a class of conditional term rewriting systems. In Kaplan and Jouannaud [19].

[8] *ECOOP*, volume 1628 of *LNCS*, 1999. Springer-Verlag.

[9] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, June 2011.

[10] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP*. July 2003. ISBN 3-540-40531-3.

[11] T. Heuillard. Compiling conditional rewriting systems. In Kaplan and Jouannaud [19].

[12] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, Dec. 1972.

[13] S. S. Huan, D. Zook, and Y. Smaragdakis. Domain-specific languages and program generation with Meta-AspectJ. *TOSEM*, 18(2), Oct. 2008.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, revised edition, Jan. 2012.

[15] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: proof rules and implementation. In *Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[16] L. Jadoul, L. Duponcheel, and W. Van Puymbroeck. An algebraic data type specification language and its rapid prototyping environment. In *ICSE*, May 1989.

[17] P. Jalote. Synthesizing implementations of abstract data types from axiomatic specifications. *Software—Practice & Experience*, 17(11), Nov. 1987.

[18] K. D. Jones. A Semantics for a Larch/Modula-3 Interface Language. In *Workshop on Larch*. July 1992.

[19] *Proceedings of the 1st International Workshop on Term Rewriting Systems*, volume 308 of *LNCS*, 1988. Springer-Verlag.

[20] T. Kühne. Internal iteration externalized. In Guerraoui [8].

[21] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, MIT, 2007.

[22] R. Laemmel and S. P. Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI*, 2003.

[23] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Apr. 2003. URL `http://www.jmlspecs.org`.

[24] K. R. M. Leino. Data groups: specifying the modification of extended state. In *OOPSLA*, Oct. 1998.

[25] K. R. M. Leino and A. Milicevic. Program extrapolation with Jennisys. Technical Report KRML219, Microsoft Research, 2012. URL `http://research.microsoft.com/pubs/158573/krml219.pdf`.

[26] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA*, Oct. 1997.

[27] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In Guerraoui [8], pages 135–160.

[28] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS, 1996.

[29] J. Liu and A. C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *PADL*, 2003.

[30] J. Liu, A. Kimball, and A. C. Myers. Interruptible iterators. In *POPL*, Jan. 2006.

[31] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, 2011.

[32] C. Morgan. *Programming from Specifications*. Prentice-Hall, Inc., 2nd edition, 1998. First edition 1990.

[33] S. Murer, S. Omohundro, D. Stoutamire, and C. Szypersky. Iteration abstraction in Sather. *TOPLAS*, 18(1), 1996.

[34] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, Nov. 2008.

[35] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[36] A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004.

[37] V. K. Proulx and W. Jossey. Unit test support for Java via reflection and annotations. In *Principles and Practice of Programming in Java*, 2009.

[38] D. Rayside, Z. Benjamin, R. Singh, J. P. Near, A. Milicevic, and D. Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In *ICSE*, 2009.

[39] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Proceedings of Onward'09*, Oct. 2009.

[40] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *OOPSLA*, Oct. 2010.

[41] H. Samimi, E. D. Aung, and T. Millstein. Falling back on executable specifications. In *ECOOP*. June 2010.

[42] Y. Smaragdakis and D. Batory. DiSTiL: a Transformation Library for Data Structures. In *DSL*, 1997.

[43] M. K. Srivas. *Automatic synthesis of implementations for abstract data types from algebraic specifications*. PhD thesis, MIT, 1982.

[44] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*. Mar. 2007.

[45] M. T. Vandevoorde and J. V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *FSE*, Dec. 1994.

[46] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In *ECOOP*. July 2007.

[47] P. Wadler. Deforestation: transforming programs to eliminate trees. *TCS*, 73:231–248, 1990.

[48] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for Java. In *ECOOP*. July 2006.

[49] K. Yessenov. A light-weight specification language for bounded program verification. Master's thesis, MIT, May 2009.

[50] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, June 2008.