Program Extrapolation with Jennisys

K. Rustan M. Leino[0] and Aleksandar Milicevic[1]

[0] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[1] Massachusetts Institute of Technology (MIT), Cambridge, MA, USA
aleks@csail.mit.edu

Manuscript KRML 219, 7 February 2012.

Abstract. The desired behavior of a program can be described using an abstract model. Compiling such a model into executable code requires advanced compilation techniques known as synthesis. This paper presents a language, called Jennisys, where programming is done by introducing an abstract model, defining a concrete data representation for the model, and then being aided by automatic synthesis to produce executable code. The paper also presents a synthesis technique for the language. The technique is built on an automatic program verifier that, via an underlying SMT solver, is capable of providing concrete models to failed verifications. The technique proceeds by obtaining sample input/output values from concrete models and then extrapolating programs from the sample points. The synthesis aims to produce code with assignments, branching structure, and possibly recursive calls. A prototype of the language and synthesis technique has been implemented.

0 Introduction

One important approach to ensuring program correctness is to raise the level of abstraction provided by programming languages. If a language lends itself to clean descriptions of solutions in the problem domain, then a programmer may be more likely to get programs correct. Two desiderata in this approach, which may seem to be at odds with each other, are (D0) allowing higher-level descriptions of programs in a general-purpose programming language and (D1) allowing efficient run-time representations of programs [19]. In this paper, we present a language framework that combines these two. The language is called Jennisys, and because it allows program designs to be recorded ahead of their concrete implementations, the language slogan is “This is where programs begin”.

Most programming languages provide some delineation between the public specification of a procedure, type, or module and the private implementation thereof. In some cases, a public specification consists of just a type signature; in other cases, it may include a behavioral contract [4, 15]. Jennisys takes the delineation a step further, dividing every program component (or class, if you will) into three parts, which allows a separation between data structure definitions and code.

The first part of a Jennisys component is the *public interface* (cf. Fig. 0, the **`interface`** declaration).

```
interface IntSet {
  var elems: set[int]

  constructor Dupleton(x, y: int)
    requires x ≠ y
    elems := {x y}

  constructor Singleton(x: int)
    elems := {x}

  method Find(x: int)
      returns (ret: bool)
    ret := x ∈ elems
}
```

```
datamodel IntSet {
  var data: int
  var left, right: IntSet

  frame left * right

  invariant
    elems = {data} +
            (left ≠ null ? left.elems : {}) +
            (right ≠ null ? right.elems : {})
    left ≠ null  ⟹
        ∀ e • e ∈ left.elems ⟹ e < data
    right ≠ null ⟹
        ∀ e • e ∈ right.elems ⟹ data < e
}
```

Fig. 0. A Jennisys public *interface* `IntSet` that defines an integer set, and a concrete *data structure*, namely a binary tree. Model variable `elems` is used to describe the behavior of the operations, but is itself not compiled into executable code.

It defines an abstract model of the component, given in terms of variables whose types are often mathematical structures, like sets and sequences. The public interface also defines the component's operations and the behavioral effects of these, typically given in terms of simple code snippets that act on the model variables. The model variables and the code acting on these describe the component, but are not compiled as part of the run-time manifestation of the program.

The second part describes the data structure used to represent the component at run time (cf. Fig. 0, the **`datamodel`** declaration). More specifically, it declares concrete variables (object fields, if you will) that are part of each instance of the component, gives an account of which other component instances are part of the representation (this is called the *frame*), and specifies an *invariant* that both constrains the concrete variables and frame and couples these with the model variables in the public interface.

The third part of a Jennisys component is responsible for the executable code that will implement the component operations (e.g. **`code`** `IntSet{}`). The vision is for the language to provide the programmer with a variety of ways to produce the code, including automatic code synthesis (which is our focus in this paper), code-generation hints, program sketches [25], and, as a last resort, old-fashioned manual coding.

Jennisys is general purpose, addressing (D0). Its public interfaces let programmers write clean descriptions whose correctness can more easily be ascertained by human scrutiny. The variety of ways to obtain code aims to speed up code production and maintainance. The data-structure description addresses (D1) by letting programmers use their insights into defining good data structures.

Jennisys is still a prototype. In this paper, we focus on the automatic code synthesis. In particular, we contribute a technique that from abstract variables, abstract code, concrete variables, and a coupling invariant (in other words, from the **`interface`** and **`datamodel`** of a component) synthesizes loop-less structured programs, where each "if" branch contains assignments to modifiable fields (one assignment per field) and possibly some method calls. The synthesis technique is most readily applicable to constructors, but its class of applications extends beyond that; for example, we show we can synthesize code for some recursive methods for traversing or computing something about complex data structures.

In a nutshell, our technique uses a program verifier to obtain sample input/output values that satisfy the given specifications. The sample values are then extrapolated into code for all input values. Frequently, the synthesized code will contain necessary branching structure, will allocate new instances of other components, and will call methods on those components in order to change their state.

1 Examples

In this section, we give examples that illustrate the use of Jennisys and the code it synthesizes.

Figure 0 showsa Jennisys component that we will use as a running example, `IntSet`.[0] Abstractly, an `IntSet` is an integer set, which we define by model variable `elems`. The constructors create a set of size 1 or 2, respectively, and method `Find` returns whether or not a given integer is part of the set. An operation can define a *precondition* (keyword **`requires`**), which obligates callers to invoke the operation only when the condition is met. The effect of an operation is given by assignments or (as the second example will show) relations (constraints) on the pre- and post-states.

A concrete data structure for `IntSet` is described using the **`datamodel`** declaration.

The variable declarations (e.g. **`var`** `data`: **`int`**) introduce the familiar fields of a binary-tree node. Unlike the model variables in the public interface, these concrete variables will be present in the run-time representation of `IntSet` components.

The **`frame`** declaration says that the memory locations used to represent an `IntSet` include not just the `IntSet` object itself, but also those those memory locations that are used to represent the `IntSet` components `left` and `right`. The star, inspired by the notation of separation logic [18], says that the sets of memory locations used by `left` and `right` are disjoint. The frame declaration is necessary for the verification of candidate synthesized programs and tells the synthesis engine which parts of the underlying state a method may mutate.

The **`invariant`** declaration defines a relation between the model variables and the concrete variables, as in a *coupling invariant* (aka an *abstraction invariant* or *retrieve relation*, see, e.g., [0, 1, 11]). It also constrains the values of the concrete representation, as in a *class invariant* [15].

From Fig. 0, Jennisys automatically synthesizes code for the three operations. We give an excerpt of that code in Fig. 1. The target language is Dafny [13], which for us has the advantage that we can use the Dafny verifier to double check the correctness of the synthesized code. We also use Dafny during the synthesis itself. Dafny compiles

[0] In this paper, we sometimes stray from the concrete syntax of our Jennisys prototype in order to make programs easier to read. Most notably, we replace the ASCII syntax of some operators by common mathematical notation. The actual Jennisys programs are available online in the Jennisys distribution.

to the .NET virtual machine, so there is in principle no reason why Jennisys could not compile to any Java-like language.

Some interesting things to note about the synthesized code are the **`if`** statements in `Dupleton` and `Find`. Also, note the dynamic allocation of objects on line 50, the call of another constructor on line 51 (to respect the abstraction boundary of the non-**`this`** object `gensym85`), the use of ghost variables on lines 2, 46, . . . (these are needed only during verification), and the recursive calls to `Find` on lines 83, 84, 88, and 92.

Note, although the operations in the public `IntSet` interface can only construct sets with cardinality 1 or 2 (because our Jennisys prototype is currently not up to the task of synthesizing code for a `Union` method), this data representation allows arbitrarily large sets. Indeed, the synthesized code for `Find` will work for any concrete `IntSet` .

2 Dynamic Synthesis

This section focuses on the algorithm for program synthesis behind Jennisys. We call the algorithm *dynamic synthesis*, because it combines ideas from both concrete and symbolic execution, in a way that is similar to what concolic testing [22, 28] does. In contrast to concolic testing, however, declarative specifications are being executed, rather than traditional imperative code.

We first describe how Dafny [13], a program verifier for functional correctness, can be used to execute first-order declarative specifications of Jennisys. Dafny is implemented on top of Boogie [3], an intermediate verification language, which uses an SMT solver, namely Z3 [5], to automatically discharge verification conditions.

Executing a specification gives only a single valid input/output (or pre-state/post-state) pair, that is, a concrete instance of the program heap for which the specification holds. In order to synthesize a program that is correct for all possible cases (i.e. all possible pre-states) this is clearly not enough. To this end, we next present an algorithm for systematic state exploration and program extrapolation from concrete instances. Since the problem of synthesis is undecidable, the algorithm does not always succeed, but when it does, the synthesized program is provably correct (Dafny can verify it against the original specification).

2.0 Concrete Specification Execution with Dafny

By "executing a method specification" we mean "finding *some* pre- and post-states that satisfy that specification". This is different from some previous work where executing specifications is part of the runtime system (e.g. [16,20]); in those systems, the pre-state is always explicitly known (it is the state of the running program before the specification is executed), so the goal there is to find a valid post-state for *a given* pre-state.

Dafny, even though designed for program verification, can be facilitated for this task. The basic idea is to tell Dafny to assume (using the **`assume`** keyword in Dafny) both the pre-condition and the post-condition, and then ask it to derive false from there (see Fig. 2 for an example). If Dafny succeeds, the pre- and post-conditions are mutually inconsistent, so any attempt to synthesize code for such a specification must be futile. Otherwise, a counterexample where all the assumed constraints hold is returned, so concrete values for a pre- and post-state can be directly extracted from it.

```
class IntSet {
  ghost var Repr: set<object>, elems: set<int>;
  var data: int, left: IntSet, right: IntSet;

  function Valid(): bool
```

(omitted, is defined to return **true** when the invariants of all reachable objects hold)

```
  method Dupleton(x: int, y: int)
    modifies this;
    requires x ≠ y;
    ensures Valid() ∧ fresh(Repr - {this});
    ensures elems = {x, y};
  {
    if (x < y) {
      var gensym85 := new IntSet;
      gensym85.Singleton(y);
      this.data := x;
      this.elems := {x, y};
      this.left := null;
      this.right := gensym85;
      this.Repr := {this} + this.right.Repr;
      assert gensym85.Valid();
    } else {
```

(the other case is symmetric)

```
  } }

  method Find(x: int) returns (ret: bool)
    requires Valid();
    ensures Valid() ∧ fresh(Repr - old(Repr));
    ensures ret = (x ∈ elems);
    decreases Repr;
  {
    if (this.left ≠ null ∧ this.right ≠ null) {
      var x_27 := this.left.Find(x);
      var x_28 := this.right.Find(x);
      ret := (x = this.data ∨ x_27) ∨ x_28;
    } else {
      if (this.left ≠ null) {
        var x_29 := this.left.Find(x);
        ret := x = this.data ∨ x_29;
      } else {
        if (this.right ≠ null) {
          var x_30 := this.right.Find(x);
          ret := x = this.data ∨ x_30;
        } else {
          ret := x = this.data;
  } } } }

}
```

Fig. 1. Excerpts of the Dafny code that Jennisys synthesizes for the `IntSet` example. For brevity, some lines have been combined in the figure. Dafny's ghost variables are not present during the run-time execution of Dafny programs, but are needed to verify the correctness of the synthesized program.

```
class IntSet {
  ghost var elems: set<int>;  var data: int;  var left, right: SetNode;
  method Dupleton() modifies this; {
    var x, y: int;
    assume a ≠ b ∧ elems = {a, b} ∧ Invariant();
    assert false;}}
```

Fig. 2. Translation of `IntSet.Dupleton` into Dafny for specification execution. Since Jennisys and Dafny share the same language (modulo the exact syntax), this translation is straightforward.

2.1 Symbolic and Concrete Execution Combined

Assigning concrete values (constants) obtained by executing a specification to output variables is unlikely to result in a correct program. The goal is, therefore, to try and generalize from a concrete instance and find symbolic assignments instead. Furthermore, even though more general than constants, such symbolic assignments might be correct only for certain program scenarios represented by the concrete instance used. When that is the case, a logical condition (guard) must be inferred to characterize these particular scenarios. If it can be verified that the symbolic assignments are correct given the inferred guard, one branch of the target program is successfully synthesized. To discover the rest of the program, a new program specification is created by adding a negation of the inferred guard as a fresh pre-condition, and the synthesis process is then recursively repeated for the new program specification.

To solve the problem of finding a guard and symbolic assignments for output variables, the specification is first **partially evaluated** with respect to the previously obtained concrete instance. This process yields a specification that is simpler and more specific to the current instance. This simplified specification is then **symbolically executed** to arrive at a set of symbolic expressions that can be used, depending on the type, as potential guards or variable assignments.

3 Synthesis Algorithm in Depth

3.0 Partial Specification Evaluation

Let us assume throughout this section that executing the specification of the `Dupleton` method yielded the instance shown in Fig. 3(a).

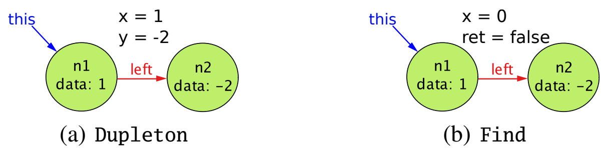

Fig. 3. Concrete instances obtained for methods `Dupleton` and `Find`.

The original specification of a method can be unfolded for a given particular instance by means of unrolling the quantifiers that assert the invariant for all objects on the heap. Unfolding the specification of `Dupleton` for the instance from Fig. 3(a) gives:

```
x ≠ y ∧ n1.elems = {x y}

n1.elems = {n1.data} + (n1.left ≠ null ? n1.left.elems : {}) +
                       (n1.right ≠ null ? n1.right.elems : {})
n1.left ≠ null  ⟹ ∀ e • e in n1.left.elems ⟹ e < n1.data
n1.right ≠ null ⟹ ∀ e • e in n1.right.elems ⟹ n1.data < e

n2.elems = {n2.data} + (n2.left ≠ null ? n2.left.elems : {}) +
                       (n2.right ≠ null ? n2.right.elems : {})
n2.left ≠ null  ⟹ ∀ e • e in n2.left.elems ⟹ e < n2.data
n2.right ≠ null ⟹ ∀ e • e in n2.right.elems ⟹ n2.data < e
```

This expression as a whole must evaluate to true, because the instance was generated so that the specification holds for it. However, the insight is that some subexpressions of the specification need not be relevant for the particular instance at hand (e.g. a consequent of an implication whose antecedent is false). For example, in our concrete instance, `n1.right`, `n2.left`, and `n2.right` are all **`null`**, so with that in mind, the previous constraint can easily be simplified to arrive at what we will refer to as a *heap expression*:

```
x ≠ y ∧ n1.elems = {x y} ∧ n2.elems = {n2.data}
n1.elems = {n1.data} + n1.left.elems
∀ e • e in n1.left.elems ⟹ e < n1.data
```

We call this notion of simplification *partial evaluation* and define it formally in Fig. 4[1]. The basic idea is to drop all disjunction terms that when fully evaluated (using the *eval*[2] function) give false, since they are likely to be irrelevant for the current instance.

The *apply* function is used to reconstruct symbolic expressions along the way. Its significance is that it additionally performs some well-known simplifications. Beside short-circuiting boolean expressions, it implements several rules specifically designed for the task of synthesis. The most interesting example would be the simplification rules for operations over sequences, as depicted in Fig. 5; in particular, they enable decomposition of specifications involving sequences into smaller bits which are often simpler to synthesize code from.

3.1 Symbolic Specification Execution

After obtaining a heap expression for the current instance (by computing $\mathcal{E}(e)$ for the unfolded version of the original specification e), a *database* of premises is created. The initial set of premises includes the conjuncts of the heap expression (first column in the listing below), as well as $v = eval(v)$ mappings for all variables (last two columns). In our running example, the initial database contains the following premises:

[1] For brevity, the instance parameter is not explicitly used in the definition in Fig. 4; it is instead assumed to be the “current” instance.

[2] The *eval* function, given a concrete instance evaluates an expression to a constant. This is a well-known notion of evaluation, so we do not give a formal definition here.

$\mathcal{E} : Expr \to Expr$		
		simple rewriting
$\mathcal{E}(Const)$	$\equiv$	$Const$
$\mathcal{E}(Var)$	$\equiv$	Var
$\mathcal{E}(\lvert e \rvert)$	$\equiv$	$apply(\lvert\rvert,\ \mathcal{E}(e))$
$\mathcal{E}([e_0, e_1, \ldots, e_{n-1}])$	$\equiv$	**List.map** $\mathcal{E}\ [e_0, e_1, \ldots, e_{n-1}]$
$\mathcal{E}(\{e_0, e_1, \ldots, e_{n-1}\})$	$\equiv$	**Set.map** $\mathcal{E}\ \{e_0, e_1, \ldots, e_{n-1}\}$
$\mathcal{E}(lst[idx])$	$\equiv$	$apply([],\ \mathcal{E}(lst),\ \mathcal{E}(idx))$
$\mathcal{E}(e_1\ \rho\ e_2)$	$\equiv$	$apply(\rho,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
		$\rho\ -$ relational operator: $=,\ \neq,\ <,\ \leq,\ >,\ \geq,\ \in,\ \notin$
$\mathcal{E}(e_1\ \alpha\ e_2)$	$\equiv$	$apply(\alpha,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
		$\alpha\ -$ arithmetic operator: $+,\ -,\ *,\ /,\ \%$
$\mathcal{E}(\forall v \bullet e)$	$\equiv$	$\forall v \bullet e$
		simplification of logic expressions
$\mathcal{E}(c\ ?\ t : e)$	$\equiv$	**if** $eval(c)$ **then** $\mathcal{E}(t)$ **else** $\mathcal{E}(e)$
$\mathcal{E}(e_1 \wedge e_2)$	$\equiv$	$apply(\wedge,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
$\mathcal{E}(e_1 \vee e_2)$	$\equiv$	**match** $eval(e_1),\ eval(e_2)$ **with**
		$\mid$ true, true $\to apply(\vee,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
		$\mid$ true, false $\to \mathcal{E}(e_1)$
		$\mid$ false, true $\to \mathcal{E}(e_2)$
		$\mid$ false, false $\to False$
$\mathcal{E}(e_1 \implies e_2)$	$\equiv$	$\mathcal{E}(\neg e_1 \vee e_2)$
$\mathcal{E}(e_1 \iff e_2)$	$\equiv$	$\mathcal{E}((e_1 \wedge e_2) \vee (\neg e_1 \wedge \neg e_2))$
$\mathcal{E}(\neg e)$	$\equiv$	$apply(\neg,\ \mathcal{E}(e))$

helper functions:

$eval\ : Expr \to Const$	– evaluates an expression to a constant wrt the current instance
$apply : Op \to Expr$ list $\to Expr$	– applies a given operator to given operands

Fig. 4. Partial Expression Evaluation Function: partially evaluates a given expression with respect to a concrete instance, making it simpler and more specific to that instance.

$apply : Op \to Expr$ list $\to Expr$		
		Simplifications for the || operator
$apply(\lvert\rvert,\ l_1 + l_2)$	$\equiv$	$apply(\lvert\rvert,\ l_1) + apply(\lvert\rvert,\ l_2)$
$apply(\lvert\rvert,\ [e_0,\ \ldots,\ e_{n-1}])$	$\equiv$	n
		Simplifications for the [] operator
$apply([],\ [e_0,\ \ldots,\ e_i,\ \ldots,\ e_{n-1}],\ i)$	$\equiv$	e_i
$apply([],\ [e_0,\ \ldots,\ e_{k-1}] + l,\ i)$	$\equiv$	**if** $i < k$ **then** e_i **else** $apply([],\ l,\ i - k)$

Fig. 5. Simplifications of the sequence length and sequence select expressions performed by the *apply* function.

		inference rules for $\in$	
$x \in []$	$\vdash$	$False$	(1)
$x \in [e]$	$\vdash$	$x = e$	(2)
$x \in [e_0, e_1, \ldots, e_{n-1}]$	$\vdash$	$x \in [e_0] + [e_1, \ldots, e_{n-1}]$	(3)
$x \in \{\}$	$\vdash$	$False$	(4)
$x \in \{e\}$	$\vdash$	$x = e$	(5)
$x \in \{e_0, e_1, \ldots, e_{n-1}\}$	$\vdash$	$x \in \{e_0\} + \{e_1, \ldots, e_{n-1}\}$	(6)
$x \in e_1 + e_2$			
when $eval(x \in e_1) \wedge eval(x \notin e_2)$	$\vdash$	$x \in e_1$	(7)
when $eval(x \notin e_1) \wedge eval(x \in e_2)$	$\vdash$	$x \in e_2$	(8)
else	$\vdash$	$x \in e_1 \vee x \in e_2$	(9)
		inference rules for $\forall$	
$\forall x \in [e_0,\ \ldots,\ e_{n-1}] \bullet p$	$\vdash$	$p[x \rightsquigarrow e_0] \wedge \ldots \wedge p[x \rightsquigarrow e_{n-1}]$	(10)
$\forall x \in \{e_0,\ \ldots,\ e_{n-1}\} \bullet p$	$\vdash$	$p[x \rightsquigarrow e_0] \wedge \ldots \wedge p[x \rightsquigarrow e_{n-1}]$	(11)
$\forall x \in e_1 + e_2 \bullet p$	$\vdash$	$(\forall x \in e_1 \bullet p) \wedge (\forall x \in e_2 \bullet p)$	(12)

Fig. 6. Inference rules for symbolic execution.

```
x ≠ y                                       x = 1            y = -2
n1.elems = {x y}                            n1.data = 1      n2.data = -2
n1.elems = {n1.data} + n1.left.elems        n1.left = n2     n2.left = null
∀ e ● e in n1.left.elems ⟹ e < n1.data      n1.right = null  n2.right = null
n2.elems = {n2.data}                        this = n1
```

Using a set of *inference rules* (defined in Fig. 6), new premises are derived from existing ones and they are added to the database. This process is repeated until either a fixpoint or a predefined maximum number of iterations is reached.

The main purpose of the inference rules from Fig. 6 is to decompose and simplify expressions over the built-in data structures. For example, from a specification like $x \in e_1 + e_2$, and a concrete instance in which x is not in the sequence e_2, $x \in e_1$ can be safely derived. These rules derive expressions specific to the current instance, and thus help infer appropriate guards and symbolic assignments. Some rules are independent of the concrete instance, e.g. $x \in [e_0,\ e_1,\ \cdots,\ e_{n-1}] \vdash x \in [e_0] + [e_1,\ \cdots,\ e_{n-1}]$; their purpose is mainly to enable rules of the previous kind to get instantiated more often.

3.2 Choosing Correct Assignments for Output Variables

At the end of the previous step, the database might (and typically does) contain multiple assignments for each variable. Jennisys automatically rules some of them out, and uses a heuristic to rank the remaining ones. For instance, Jennisys prefers those that contain symbolic, rather than constant values.

From the initial database for the `Dupleton` example, just by applying transitivity of equality, the following *candidate solution* is quickly discovered (other assignments exists in the database, but they all contain constant values):

```
n1.elems := {x y}; n1.data := x;  n1.left := n2;   n1.right := null;
```

```
n2.elems := {y};   n2.data := y;  n2.left := null; n2.right := null;
```

As expected, this solution does not verify against the original specification of the `Dupleton` method. Knowing the properties of binary search trees, we can easily conclude that the solution we just discovered is valid if it is known that `y < x` holds. In the next subsection we show how Jennisys automatically infers this condition (guard).

3.3 Inference of Guards

The main insight for successful guard inference is that an appropriate guard is likely to be a logical property of the current instance. Therefore, a guard is likely to consist of one or more premises from the database.

Going back to the example, the `y < x` condition was indeed derived during the fixpoint algorithm. Just by applying transitivity of equality, the premise containing a universal quantifier can be rewritten as $\forall$ `e` **in** `{n2.data}` • `e < n1.data`. By applying rule 11 next, `n2.data < n1.data` is derived, which immediately leads to `y < x`.

Jennisys selects a candidate guard by going through the database and looking for boolean-typed expressions that involve only unmodifiable variables and constants (expressions without constants are again ranked higher). When multiple such expressions exist, a conjunction of all of them is used first. If a candidate solution verifies under the assumption of a selected guard, the guard is minimized by iteratively trying to remove one clause at a time. For example, during the synthesis of the `Dupleton` method, $x \neq y \wedge y < x$ was selected as a guard first, and was next minimized to `y < x`.

3.4 Top-level Algorithm

The top-level synthesis function, `synth`, is given in Fig. 7. At the very beginning, it invokes `synth_branch` to find a solution for the current instance only (exactly by following the procedure described so far). If no verified solution is found (line 4), Jennisys gives up. If both guard and a solution were found (line 6), the guard is negated and appended to the list of pre-conditions to ensure that all subsequent concrete instances obtained by executing the method's specification fall outside this of branch. The whole process is then repeated to find a solution for the *else* branch. Finally, if a solution was found for which a guard was not needed (line 5), a solution for the entire program is discovered (not just the current instance!). That is true because a solution is just proven unconditionally correct for the portion of the program space not covered by the previously synthesized branches. This solution represents the last *else* branch of the *if-then-elif-...-else* structure that our approach synthesizes.

3.5 Termination

An important question is whether the algorithm is guaranteed always to terminate. The only way for the `synth` function not to terminate is if it is possible to forever keep generating new concrete instances and each time finding a guarded solution. The way we generate new instances guarantees that at each step the remainder of the search space is getting smaller, because every new instance is guaranteed to be outside of

```
let Solution = FlatSol | IfThenElse(Guard, FlatSol, Solution)
let rec synth (m: Method): Solution =
  let guardOpt,flatSolOpt = synth_branch m
  match flatSolOpt, guardOpt with
  | None, _                    -> None
  | Some(flatSol), None        -> Some(flatSol)
  | Some(flatSol), Some(guard) -> match synth (AddPrecondition m (not guard)) with
                                    | None -> None
                                    | Some(solElse) -> Some(IfThenElse(guard, flatSol, solElse))
```

Fig. 7. Top level algorithm in pseudo F#

the previously discovered classes of programs (characterized by previously discovered guards). However, it can happen that at each step the inferred guard is over-constrained (e.g. it does not allow any instance other than the current one). In that case, if the search space is unbounded (that is, there are infinitely many different instances for the program under analysis), the algorithm potentially never terminates. To mitigate this, Jennisys always prefers solutions/guards with no constants so that at every step the remainder of the search space is shrunk as much as possible. In practice, this means that the algorithm is likely to either terminate with a solution or fail quickly.

4 Synthesizing Recursive Methods

The synthesis algorithm described so far was designed to support constructors in the form of a single (but of arbitrary length) *if-then-elseif-...-else* structure, where the only allowed statements are assignments to output variables. In this section, we show how we extended the algorithm to allow method calls (including recursion) in the assignment statements. Allowing recursion somewhat makes up for the lack of looping constructs.

Two modifications to the synthesis algorithm are needed: (0) after building the initial set of premises, **parameterized** expressions corresponding to method specifications are added to the database; and (1) the inference engine is modified so that it allows matching with **unification**. To illustrate this, consider the `IntSet.Find` method. After its specification is executed for the first time, let us assume that the instance from Fig. 3(b) is discovered. The initial set of premises looks almost the same as before (since the instance is almost the same), with a difference of the first line (the pre-condition from the previous example) being replaced with the post-condition of the `Find` method (`ret = x ∈ this.elems`) and a parameterized specification for `Find` (`$this.Find($x) = $x ∈ $this.elems`). The derivation then goes as follows:

$$\begin{array}{ll}
\texttt{ret = x} \in \textbf{this}\texttt{.elems = (\{}\textbf{this}\texttt{.data\} + }\textbf{this}\texttt{.left.elems)} & \\
\quad \rightarrow \quad \texttt{ret = x} \in \texttt{\{}\textbf{this}\texttt{.data\}} \vee \texttt{x} \in \textbf{this}\texttt{.left.elems} & \text{(by rule 9)} \\
\quad \rightarrow \quad \texttt{ret = (x = }\textbf{this}\texttt{.data)} \vee \texttt{x} \in \textbf{this}\texttt{.left.elems} & \text{(by rule 5)} \\
\quad \rightarrow \quad \texttt{ret = (x = }\textbf{this}\texttt{.data)} \vee \textbf{this}\texttt{.left.Find(x)} & \left(\begin{array}{c} \text{matching with unif.} \\ \texttt{\$this} \rightsquigarrow \textbf{this}\texttt{.left} \\ \texttt{\$x} \rightsquigarrow \texttt{x} \end{array}\right)
\end{array}$$

The remainder of the process stays the same. The guard for this instance (**`this`**`.left` $\neq$ **`null`**) is easily inferred (note that it is okay now to use **`this`**`.left` in the guard, because **`this`** is unmodifiable in this case) and the process continues the same way to synthesize the rest of the program. The final program is shown in Fig. 1.

	IntSet.Singleton	IntSet.Dupleton	IntSet.Find	List.Singleton	List.Dupleton	List.Elems	List.Get	List.Find	List.Size	DList.Singleton	DList.Dupleton	DList.Elems	DList.Get	DList.Find	DList.Size	Math.Min2	Math.Min3	Math.Min4	Math.Abs
#branches	1	2	4	1	1	2	3	2	2	1	1	2	3	2	2	2	3	4	2
time (s)	3.8	14.8	51.2	3.4	5.3	9.2	14.7	12.0	9.2	3.6	16.8	9.6	15.3	12.0	9.6	6.1	13.9	26.0	7.4

Table 0. Synthesis times and number of branches of the synthesized code for several examples.

5 Benchmarks

Table 0 shows synthesis times of the three methods from the `IntSet` module (Fig. 0), two constructors and four recursive methods for both singly- (`List`) and doubly-linked (`DList`) lists, and four simple math operations (minimum of two, three, and four integers, and absolute value of an integer). All experiments were done on an Intel® Core™2 Duo CPU @ 2.40GHz computer, with 4GB of RAM. Jennisys programs for the singly-linked list and some of the math operations are given in Fig. 8; the synthesized code for the singly-linked list methods is given in Fig. 9. It is important to note the declarativeness of the specifications and that no additional input is required from the user. This idealistic approach is currently the main reason why Jennisys can synthesize only a limited array of programs, namely constructors and certain read-only recursive methods. We do envision, however, that this technique can be useful for synthesizing certain domain-specific applications (e.g. configuration problems) as well as a broader class of recursive methods (e.g. list insertion); we have yet to explore these possibilities.

```
interface List[T] {
  var list: seq[T]
  invariant |list| > 0

  constructor Singleton(t: T)
    ensures list = [t]
  constructor Dupleton(p: T, q: T)
    ensures list = [p q]
  method List() returns (ret: seq[T])
    ensures ret = list
  method Size() returns (ret: int)
    ensures ret = |list|
  method Get(idx: int) returns (ret: T)
    requires 0 ≤ idx ∧ idx < |list|
    ensures ret = list[idx]
  method Find(n: T) returns (ret: bool)
    ensures ret = (n ∈ list)
}

interface Math {
  method Min3(a, b, c: int) returns (ret: int)
    ensures ret in {a b c}
    ensures ∀ x • x in {a b c} ⟹ ret ≤ x
  method Abs(a: int) returns (ret: int)
    ensures ret in {a (-a)} ∧ ret ≥ 0
}

datamodel List[T] {
  var data: T
  var next: List[T]

  frame next

  invariant
    next = null  ⟹ list = [data]
    next ≠ null ⟹ list = [data] + next.list
}
```

Fig. 8. Implementation of a singly-linked list and some math operations in Jennisys.

6 Related Work

Automatic synthesis of programs from specifications, or *automatic programming*, has been a dream for more than four decades. The idea that software engineering would be a better place if programmers could spend their time editing specifications, rather than trying to maintain optimized programs, is argued convincingly in a paper that tried to predict the future [2].

```
method Get(idx: int) returns (ret: T)
  requires Valid();
  requires 0 ≤ idx;
  requires idx < |list|;
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = list[idx];
  decreases Repr;
{
  if (this.next = null) {
    ret := this.data;
  } else {
    if (idx = 0) {
      ret := this.data;
    } else {
      var x_6 := this.next.Get(idx - 1);
      ret := x_6;
    }
  }
}
```

```
method Find(n: T) returns (ret: bool)
  requires Valid();
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = (n in list);
  decreases Repr;
{
  if (this.next = null) {
    ret := n = this.data;
  } else {
    var x_5 := this.next.Find(n);
    ret := n = this.data ∨ x_5;
  }
}
```

```
method Elems() returns (ret: seq<T>)
  requires Valid();
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = list;
  decreases Repr;
{
  if (this.next = null) {
    ret := [this.data];
  } else {
    var x_7 := this.next.Elems();
    ret := [this.data] + x_7;
  }
}
```

```
method Size() returns (ret: int)
  requires Valid();
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = |list|;
  decreases Repr;
{
  if (this.next = null) {
    ret := 1;
  } else {
    var x_8 := this.next.Size();
    ret := 1 + x_8;
  }
}
```

Fig. 9. Dafny code that Jennisys synthesizes for the `List` benchmark examples from Table 0.

Pioneering efforts in the synthesis area around 1970 used theorem provers to verify the existence of an output for every input and then synthesized executable programs from the ingredients of these proofs [6, 14]. More encompassing development systems with synthesis included the 1970s PSI program synthesis system [7] and the 1980s Programmer's Apprentice project [19]. These ambitious systems tried to aid programmers by engaging in a dialog about the program to be developed, offering advice, keeping track of details, and synthesizing code. The systems made use of a significant knowledge base of the domains and template scenarios (so-called *clichés*) of the programs to be developed, and PSI also included a major natural-language component. In comparison, the abstract models one can define in Jennisys look much more like programs.

Developed in the late 1980s, the comprehensive KIDS system provided a number of tools to support algorithm design and program transformations [24]. Besides major design decisions like semantically instantiating algorithm templates, the operations perfomed with KIDS are correctness-preserving transformations—*refinement steps*—that can take an algorithm description into an efficient program.

The Jennisys language is in many ways similar to one step of a refinement process, where Jennisys offers synthesis as one way of obtaining the refined program. Monahan has suggested defining components in three parts (`spec`/`abstr`/`impl`) [17], which is also what Jennisys does. Jennisys also shares in the vision of the language SETL [21], which sought to provide ways to first describe programs cleanly and then provide them with efficient data representations.

The construction of programs from examples is a powerful idea that has been explored from the 1970s. For example, THESYS synthesis system generated LISP programs [27] and QBE generated SQL queries [29]. Queries by Example became a competitive feature of the Paradox relational database system in the 1980s and 1990s, and techniques with similar goals are being explored in the context of spreadsheets today [9]. Jennisys also extrapolates programs from examples, but the examples are not supplied by users but are instead sample points from specifications supplied by users. An advantage of having specifications is that one can then verify the synthesized program, as opposed to just knowing it is correct for the examples provided.

Interest in program synthesis seems to be on the rise again, possibly in part due to the success of SMT solvers in program verification and other applications. Kuncak et al. are exploring features like generalized assignments in a mainstream programming language, backed up by automatic synthesis procedures [12]. The PINS [26] system takes a program and a template and synthesizes an inversion of the given program.

The technique of program sketching lets programmers supply some ingredients of a program (i.e. a program sketch) while a tool worries about the details to find a correct way to combine the ingredients [25]. The notion of correctness is taken from another correct (but presumably inefficient) implementation of the same program, that has to be supplied by the user. Storyboard programming [23] improves on that idea by letting the users draw a series of input/output examples instead of providing an alternative correct implementation (it still requires a sketch, though). Similarly, instead of a sketch, the Brahma tool [8, 10] takes a library of components to be used as building blocks and either an explicit set of input/output pairs or a specification describing the relation between inputs and outputs, and synthesizes a loop-free program (currently focused on bit-vector manipulations) from the given components. In comparison, Jennisys does not require any input from the user other than a specification in the form of pre- and post-conditions, it targets object-oriented programs with dynamic allocation, but is unable to synthesize as wide a class of programs as the storyboard programming.

7 Conclusion

In this paper, we have contributed a language design that promotes writing down an abstract model of each component, gives control over the data structure used to implement a component, and opens the door for synthesis techniques to fill in the code. The paper also contributes a synthesis technique for the language, which operates in the context of an infinite state space, dynamic object allocation, and object references. Finally, the paper contributes a prototype implementation of the language and synthesis technique.

Still, much work lies ahead. We are interested to see how far our idealistic approach to synthesis can be pushed (e.g. by improving the inference engine to support mutating methods) and explore different domains of programs that can be automatically synthesized purely from specifications. In the end, we also want to give programmers ways, like in program sketching [25], to provide hints about how to generate code.

References

0. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
1. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
2. R. Balzer, T. E. Cheatham, Jr., and C. Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, 16(11):39–45, 1983.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
4. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *J. STTT*, 7(3):212–232, 2005.
5. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
6. C. Green. Application of theorem proving to problem solving. In *IJCAI 1969*, pages 219–240. William Kaufmann, 1969.
7. C. Green. The design of the PSI program synthesis system. In *ICSE*, pages 4–18. IEEE Computer Society, 1976.
8. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM.
9. W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI 2011*, pages 317–328. ACM, 2011.
10. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.
11. C. B. Jones. *Systematic Software Development using VDM*. Series in Computer Science. Prentice-Hall International, second edition, 1990.
12. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI 2010*, pages 316–329. ACM, 2010.
13. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
14. Z. Manna and R. J. Waldinger. Towards automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
15. B. Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.
16. A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.
17. R. Monahan. *Data Refinement in Object-Oriented Verification*. PhD thesis, Dublin City University, 2010.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.
19. C. Rich and R. C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, 1988.
20. H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.
21. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer, 1986.
22. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

23. R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *ESEC/FSE 2011*, pages 289–299, New York, NY, USA, 2011. ACM.
24. D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.
25. A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS 2006*, pages 404–415. ACM, 2006.
26. S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI 2011*, pages 492–503. ACM, 2011.
27. P. D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.
28. N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, pages 281–292, 2005.
29. M. M. Zloof. Query by example. In *AFIPS National Computer Conference 1975*, pages 431–438. AFIPS Press, 1975.