Deductive Program Repair

Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak*

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. We present an approach to program repair and its application to programs with recursive functions over unbounded data types. Our approach formulates program repair in the framework of deductive synthesis that uses existing program structure as a hint to guide synthesis. We introduce a new specification construct for symbolic tests. We rely on such user-specified tests as well as automatically generated ones to localize the fault and speed up synthesis. Our implementation is able to eliminate errors within seconds from a variety of functional programs, including symbolic computation code and implementations of functional data structures. The resulting programs are formally verified by the Leon system.

1 Introduction

This paper explores the problem of automatically repairing programs written as a set of mutually recursive functions in a purely functional subset of Scala. We consider a function to be subject to repair if it does not satisfy its specification, expressed in the form of pre- and postcondition. The task of repair consists of automatically generating an alternative implementation that meets the specification. The repair problem has been studied in the past for reactive and pushdown systems [8, 10, 11, 19, 20, 26]. We view repair as generalizing, for example, the *choose* construct of complete functional synthesis [15], sketching [21,22], and program templates [23], because the exact location and nature of expressions to be synthesized is left to the algorithm. Repair is thus related to localization of error causes [12,14,27]. To speed up our repair approach, we do use *coarse-grained* error localization based on derived test inputs. However, a more precise nature of the fault is in fact the *outcome* of our tool, because the repair identifies a particular change that makes the program correct. Using tests alone as a criterion for correctness is appealing for performance reasons [7, 17, 18], but this can lead to erroneous repairs. We therefore leverage prior work [13] on verifying and synthesizing recursive functional programs with *unbounded* data-types (trees, lists, integers) to provide strong correctness guarantees, while at the same time optimizing our technique to use automatically derived tests. By phrasing the problem of repair as one of synthesis and introducing tailored deduction rules

* This work is supported in part by the European Research Council (ERC) Project *Implicit Programming* and Swiss National Science Foundation Grant *Constraint Solving Infrastructure for Program Analysis*.

that use the original implementation as guide, we allow the repair-oriented synthesis procedure to automatically find correct fixes, in the worst case resorting to re-synthesizing the desired function from scratch. To make the repair approach practical, we found it beneficial to extend the power and generality of the synthesis engine itself, as well as to introduce explicit support for symbolic tests in the specification language and the repair algorithm.

Contributions. The overall contribution of this paper is a new repair algorithm and its implementation inside a deductive synthesis framework for recursive functional programs. The specific new techniques we contribute are the following.

- **Exploration of similar expressions**. We present an algorithm for expression repair based on a grammar for generating expressions *similar* to a given expression (according to an error model we propose). We use such grammars within our new generic symbolic term exploration routine, which leverages test inputs as well as an SMT solver, and efficiently explores the space of expressions that contain recursive calls whose evaluation depends on the expression being synthesized.
- **Fault localization**. To narrow down repair to a program fragment, we localize the error by doing dynamic analysis using test inputs generated automatically from specifications. We combine two automatic sources of inputs: enumeration techniques and SMT-based techniques. We collect traces leading to erroneous executions and compute common prefixes of branching decisions. We show that this localization is in practice sufficiently precise to repair sizeable functions efficiently.
- **Symbolic examples**. We propose an intuitive way of specifying possibly symbolic input-output examples using pattern matching of Scala. This allows the user to partially specify a function without necessarily having to provide full inputs and outputs. Additionally, it enables the developer to easily describe properties of generic (polymorphic) functions. We present an algorithm for deriving new examples from existing ones, which improves the usefulness of example sets for fault localization and repair.
  In our experience, the combination of formal specification and symbolic examples gives the user significant flexibility when specifying functions, and increases success rates when discovering and repairing program faults.
- **Integration into a deductive synthesis and verification framework**. Our repair system is part of a deductive verification system, so it can automatically produce new inputs from specification, prove correctness of code for all inputs ranging over an unbounded domain, and synthesize program fragments using deductive synthesis rules that include common recursion schemas.

The repair approach offers significant improvements compared with synthesis from scratch. Synthesis alone scales poorly when the expression to synthesize is large. Fault localization focuses synthesis on the smaller, invalid portions of the program and thus results in big performance gains. The source code of our tool and additional details are available from `http://leon.epfl.ch` as well as `http://lara.epfl.ch/w/leon-repair`.

```
abstract class Expr
case class Plus(lhs: Expr, rhs: Expr)
  extends Expr
... // 9 more subclasses

abstract class SExpr
case class SPlus(lhs: SExpr,
  rhs: SExpr) extends SExpr
... // 5 more subclasses

abstract class Type
case object IntType extends Type
case object BoolType extends Type

def typeOf(e: Expr): Option[Type] =
  ...

def semI(t: Expr): Int = {
require(typeOf(t)==Some(IntType))
  ...
}
def semB(t : Expr) : Boolean = {
require(typeOf(t)==Some(BoolType))
  ...
}
def simSem(e : SExpr) : Int = ...
```

```
def desugar(e: Expr) : SExpr = {
e match {
 case Plus (lhs, rhs) ⇒
   SPlus(desugar(lhs), desugar(rhs))
 case Minus(lhs, rhs) ⇒
   SPlus(desugar(lhs), Neg(desugar(rhs)))
 case And(lhs, rhs) ⇒
   SIte(desugar(lhs), desugar(rhs), SLiteral(0))
 case Or(lhs, rhs) ⇒
   SIte(desugar(lhs), SLiteral(1), desugar(rhs))
 case Not(e) ⇒
   SIte(desugar(e), SLiteral(0), SLiteral(1))
 case Ite(cond, thn, els) ⇒
   SIte(desugar(cond), desugar(els), desugar(thn))
 case IntLiteral(v) ⇒
   SLiteral(v)
 case BoolLiteral(b) ⇒
   SLiteral(if (b) 1 else 0)}
 ...
} ensuring { res ⇒ typeOf(e) match {
  case Some(IntType) ⇒
    simSem(res) == semI(e)
  case Some(BoolType) ⇒
    simSem(res) == if (semB(e)) 1 else 0
  case None() ⇒ true }
}
```

Fig. 1. The syntax tree translation in function desugar has a strong **ensuring** clause, requiring semantic equivalence of transformed and the original tree, as defined by several recursive evaluation functions. desugar contains an error. Our system finds it, repairs the function, and proves the resulting program correct.

Example. Consider the following functionality inspired by a part of a compiler. We wish to transform (desugar) an abstract syntax-tree of a typed expression language into a simpler untyped language, simplifying some of the constructs and changing the representation of some of the types, while preserving the semantics of the transformed expression. In Figure 1, the original syntax trees are represented by the class Expr and its subclasses, whereas the resulting untyped language trees are given by SExpr. A syntax tree of Expr either evaluates to an integer, to a boolean, or to no value if it is not well typed. We capture this by defining a type-checking function typeOf, along with two separate semantic functions, semI and semB. SExpr, on the other hand, always evaluates to an integer, as defined by the simSem function. For brevity, most subclass definitions are omitted.

The desugar function translates a syntax tree of Expr into one of SExpr. We expect the function to ensure that the transformation preserves the semantics of the tree: originally integer-valued trees evaluate to the same value, boolean-valued trees now evaluate to 0 and 1, representing **false** and **true**, respectively,

and mistyped trees are left unconstrained. This is expressed in the postcondition of desugar.

The implementation in Figure 1 contains a bug: the thn and els branches of the Ite case have been accidentally switched. Using tests automatically generated using generic enumeration of small values, as well as from a verification attempt of desugar, our tool is able to find a coarse-grained location of the bug, as the body of the relevant case of the match statement. During repair, one of the rules performs a semantic exploration of expressions similar to the invalid one. It discovers that using the expression SIte(desugar(cond), desugar(thn), desugar(els)) instead of the invalid one makes the discovered tests pass. The system can then formally verify that the repaired program meets the specification for all inputs. If we try to introduce similar bugs in the correct desugar function, or to replace the entire body of a case with a dummy value, the system successfully recovers the intended case of the transformation. In some cases our system can repair multiple simultaneous errors; the mechanism behind that is explained in Section 2.2. Note that the developer communicates with our system only by writing code and specifications, both of which are functions in an existing functional programming language. This illustrates the potential of repair as a scalable and developer-friendly deployment of synthesis in software development.

2 Deductive Guided Repair

We next describe our deductive repair framework. The framework currently works under several assumptions, which we consider reasonable given the state of the art in repair of infinite-state programs. We consider the specifications of functions as correct; the code is assumed wrong if it cannot be proven correct with respect to this specification for all of the infinitely many inputs. If the specification includes input-output tests, it follows that the repaired function must have the same behavior on these tests. We do not guarantee that the output of the function is the same as the original one on tests not covered by the specification, though the repair algorithm tends to preserve some of the existing behaviors due to the local nature of repair. It is the responsibility of the developer to sufficiently specify the function being repaired. Although under-specified benchmarks may produce unexpected expressions as repair solutions, we found that even partial specifications often yield the desired repairs. A particularly effective specification style in our experience is to give a partial specification that depends on all components of the structure (for example, describes property of the set of stored elements), and then additionally provide a finite number of symbolic input-output tests. We assume that only one function of the program is invalid; the implementation of all other functions is considered valid as far as the repair of interest is concerned. Finally, we assume that all functions of the program, even the invalid one, terminate.

Stages of the Repair Algorithm. The function being repaired passes through the following stages, which we describe in the rest of the paper:

- **Test generation and verification**. We combine enumeration- and SMT-based techniques to either verify the validity of the function, or, if it is not valid, discover counterexamples (examples of misbehaviors).
- **Fault localization.** Our localization algorithm then selects the smallest expression executed in all failing tests, modulo recursion.
- **Synthesis of similar expressions.** This erroneous expression is replaced by a "program hole". The now-incomplete function is sent to synthesis, with the previous expression used as a synthesis hint. (Neither the notion of holes nor the notion of synthesis hints has been introduced in prior work on deductive synthesis [13].)
- **Verification of the solution.** Lastly, the system attempts to prove the validity of the discovered solution. Our results in Section 5, Figure 4 indicate in which cases the synthesized function passed the verification.

Repair Framework. Our starting point is the deductive synthesis framework first introduced in [13]. We show how this framework can be applied to program repair by introducing dedicated rules, as well as special predicates. We reuse the notation for synthesis tasks $[\![\bar{a}\ \langle \Pi \rhd \phi \rangle\ \bar{x}]\!]$: $\bar{a}$ denotes the set of *input variables*, $\bar{x}$ denotes the set of *output variables*, ϕ is the *synthesis predicate*, and Π is the *path condition* to the synthesis problem. The framework relies on deduction rules that take such an input synthesis problem and either (1) solve it immediately by returning the tuple $\langle P \mid T \rangle$ where P corresponds to the precondition under which the term T is a solution, or (2) decompose it into sub-problems, and define a way to compute the overall solution from sub-solutions. We illustrate these rules as well as their notation with a rule for splitting a problem containing a top-level or:

$$\frac{[\![\bar{a}\ \langle \Pi \rhd \phi_1 \rangle\ \bar{x}]\!] \vdash \langle P_1 \mid T_1 \rangle \qquad [\![\bar{a}\ \langle \Pi \rhd \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_2 \mid T_2 \rangle}{[\![\bar{a}\ \langle \Pi \rhd \phi_1 \vee \phi_2 \rangle\ \bar{x}]\!] \vdash \langle P_1 \vee P_2 \mid \mathsf{if}(P_1)\ \{T_1\}\ \mathsf{else}\ \{T_2\} \rangle}$$

This rule should be interpreted as follows: from an input synthesis problem $[\![\bar{a}\ \langle \Pi \rhd \phi_1 \vee \phi_2 \rangle\ \bar{x}]\!]$, the rule decomposes it in two subproblems: $[\![\bar{a}\ \langle \Pi \rhd \phi_1 \rangle\ \bar{x}]\!]$ and $[\![\bar{a}\ \langle \Pi \rhd \phi_2 \rangle\ \bar{x}]\!]$. Given corresponding solutions $\langle P_1 \mid T_1 \rangle$ and $\langle P_2 \mid T_2 \rangle$, the rule solves the input problem with $\langle P_1 \vee P_2 \mid \mathsf{if}(P_1)\ \{T_1\}\ \mathsf{else}\ \{T_2\} \rangle$.

To track the original (incorrect) implementation along instantiations of our deductive synthesis rules, we introduce a *guiding predicate* into the path condition of the synthesis problem. We refer to this guiding predicate as $\odot[\mathsf{expr}]$, where expr represents the original expression. This predicate does not have any logical meaning in the path-condition (it is equivalent to **true**), but it provides syntactic information that can be used by repair-dedicated rules. These rules are covered in detail in Sections 2.1, 2.2 and 3.

2.1 Fault Localization

A contribution of our system is the ability to focus the repair problem to a small sub-part of the function's body that is responsible for its erroneous behavior. The underlying hypothesis is that most of the original implementation is correct. This

technique allows us to reuse as much of the original implementation as possible and minimizes the size of the expression given to subsequent more expensive techniques. Focusing also has the profitable side-effect of making repair more predictable, even in the presence of weak specifications: repaired implementation tends to produce programs that preserve some of the existing branches, and thus have the same behavior on the executions that use only these preserved branches. We rely on the list of examples that fail the function specification to lead us to the source of the problem: if all failing examples only use one branch of some branching expression in the program, then we assume that the error is contained in that branch. We define $\mathcal{F}$ as the set of all inputs of collected failing tests (see Section 4). We describe focusing using the following rules.

If-Focus. Given the input problem $[\![\bar{a}\ \langle \odot[\mathsf{if}(c)\ \{t\}\ \mathsf{else}\ \{e\}] \rhd \phi\rangle\ \ \bar{x}]\!]$ we first check if there is an alternative condition expression such that all failing tests succeed:

If-Focus-Condition:

$$\frac{\exists C.\forall \bar{i} \in \mathcal{F}.\ \phi[\bar{x} \mapsto \mathsf{if}(C(\bar{a}))\ \{t\}\ \mathsf{else}\ \{e\}, \bar{a} \mapsto \bar{i}] \quad [\![\bar{a}\ \langle \odot[c] \wedge \Pi \rhd \phi[\bar{x} \mapsto \mathsf{if}(x')\ \{t\}\ \mathsf{else}\ \{e\}]\rangle\ \ x']\!] \vdash \langle P \mid T\rangle}{[\![\bar{a}\ \langle \odot[\mathsf{if}(c)\ \{t\}\ \mathsf{else}\ \{e\}] \wedge \Pi \rhd \phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{if}(T)\ \{t\}\ \mathsf{else}\ \{e\}\rangle}$$

Instead of solving this higher-order hypothesis, we execute the function and non-deterministically consider both branches of the **if** (and do so within recursive invocations as well). If a valid execution exists for each failing test, the formula is considered satisfiable enabling us to focus on the condition. Otherwise, we check whether c evaluates to either **true** or **false** for all failing inputs, allowing us to focus on the corresponding branch:

If-Focus-Then:

$$\frac{[\![\bar{a}\ \langle \odot[t] \wedge c \wedge \Pi \rhd \phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid T\rangle \qquad \forall \bar{i} \in \mathcal{F}.c[\bar{a} \mapsto \bar{i}]}{[\![\bar{a}\ \langle \odot[\mathsf{if}(c)\ \{t\}\ \mathsf{else}\ \{e\}] \wedge \Pi \rhd \phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{if}(c)\ \{T\}\ \mathsf{else}\ \{e\}\rangle}$$

If-Focus-Else:

$$\frac{[\![\bar{a}\ \langle \odot[e] \wedge \neg c \wedge \Pi \rhd \phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid T\rangle \qquad \forall \bar{i} \in \mathcal{F}.\neg c[\bar{a} \mapsto \bar{i}]}{[\![\bar{a}\ \langle \odot[\mathsf{if}(c)\ \{t\}\ \mathsf{else}\ \{e\}] \wedge \Pi \rhd \phi\rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{if}(c)\ \{t\}\ \mathsf{else}\ \{T\}\rangle}$$

We use analogous rules to repair **match** expressions, which are ubiquitous in our programs. Here, if all failing tests lead to one particular branch of the **match**, we focus on that particular branch.

The above rules use tests to locally approximate the validity of branches. They are sound only if $\mathcal{F}$ is sufficiently large. Our system therefore performs an end-to-end verification for the complete solution, ensuring the overall soundness.

2.2 Guided Decompositions

In case focusing rules fail to identify a single branch of an **if**- or **match**-expression as responsible, we might still benefit from reusing most of the expression. In the case of **if**, reuse is limited to the **if**-condition, but for a **match**-expression, this may extend to multiple valid cases. To this end, we introduce rules analogous to focus, that do decompositions based on the guide.

If-Split:
$$\frac{[\![\bar{a}\ \ \langle \odot[t] \wedge c \wedge \Pi \rhd \phi \rangle\ \ \bar{x}]\!] \vdash \langle P_1 \mid T_1 \rangle \qquad [\![\bar{a}\ \ \langle \odot[e] \wedge \neg c \wedge \Pi \rhd \phi \rangle\ \ \bar{x}]\!] \vdash \langle P_2 \mid T_2 \rangle}{[\![\bar{a}\ \ \langle \odot[\mathsf{if}(c)\ \{t\}\ \mathsf{else}\ \{e\}] \wedge \Pi \rhd \phi \rangle\ \ \bar{x}]\!] \vdash \langle (c \wedge P_1) \vee (\neg c \wedge P_2) \mid \mathsf{if}(c)\ \{T_1\}\ \mathsf{else}\ \{T_2\} \rangle}$$

To reuse the valid branches of an **if** or a **match**-expression on which focus failed, we introduce a rule that solves the problem if the guiding expression satisfies the specification.

$$\textsc{Guided-Verify:}\ \frac{\Pi \models \phi[\bar{x} \mapsto \mathsf{term}]}{[\![\bar{a}\ \ \langle \odot[term] \wedge \Pi \rhd \phi \rangle\ \ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \mathsf{term} \rangle}$$

2.3 Generating Recursive Calls

Our purely functional language often requires us to synthesize recursive implementations. Consequently, the synthesizer must be able to generate calls to the function currently getting synthesized. However, we must take special care to avoid introducing calls resulting in a non-terminating implementation. (Such an erroneous implementation would be conceived as valid if it trivially satisfies the specification due to inductive hypothesis over a non-well-founded relation.)

Our technique consists of recording the arguments a at the entry point of the function, f, and keeping track of these arguments through the decompositions. We represent this information with a syntactic predicate $\Downarrow[\mathsf{f(a)}]$, similar to the guiding predicate from the previous sections. We then heuristically assume that reducing the arguments a will not introduce non-terminating calls.

We illustrate this mechanism by considering the desugar function shown in Figure 1. We start by injecting the entry call information as

$$[\![e\ \ \langle \Downarrow[\mathsf{desugar(e)}] \wedge \ldots \rhd \phi \rangle\ \ x]\!]$$

This synthesis problem will then be decomposed by the various deduction rules available in the framework. An interesting case to consider is a decomposition by pattern-matching on e which specializes the problem to known variants of Expr. The specialized problem for the Plus variant will look as follows:

$$[\![e1\,, e2\ \ \langle \Downarrow[\mathsf{desugar(Plus(e1,\ e2))}] \wedge \ldots \rhd \phi \rangle\ \ x]\!]$$

As a result, we assume that the calls desugar(e1) and desugar(e2) are likely to terminate, so they are considered as candidate expressions when symbolically exploring terms, as explained in Section 3.

This relatively simple technique allows us to introduce recursive calls while filtering trivially non-terminating calls. In the case where it still introduces infinite recursion, we can discard the solution using a more expensive termination checker, though we found that this is seldom needed in practice.

2.4 Synthesis within Repair

The repair-specific rules described earlier aim at solving repair problems according to the error model. Thanks to integration into the Leon synthesis framework, general synthesis rules also apply, which enables the repair of more intricate errors. This achieves an appealing combination between fast repairs for predictable errors and expressive, albeit slower, repairs for more complicated errors.

3 Counterexample-Guided Similar-Term Exploration

After following the overall structure of the original problem, it is often the case that the remaining erroneous branches can be fixed by applying small changes to their implementations. For instance, an expression calling a function might be wrong only in one of its arguments or have two of its arguments swapped. We exploit this assumption by considering different variations to the original expression. Due to the lack of a large code base in pure Scala subset that Leon handles, we cannot use statistically informed techniques such as [9], so we define an error model following our intuition and experience from previous work.

We use the notation $G(\mathsf{expr})$ to denote the space of variations of expr and define it in the form of a grammar as

$$G(\mathsf{expr}) ::= G_{swap}(\mathsf{expr}) \mid G_{arg}(\mathsf{expr}) \mid G_{*^2}(\mathsf{expr})$$

with the following forms of variations.

Swapping arguments. We consider here all the variants of swapping two arguments that are compatible type-wise. For instance, for an operation with three operands of the same type:

$$G_{swap}(\mathsf{op(a,b,c)}) ::= \mathsf{op(b,a,c)} \mid \mathsf{op(a,c,b)} \mid \mathsf{op(c,b,a)}$$

Generalizing one argument. This variation corresponds to making a mistake in only one argument of the operation we generalize:

$$G_{arg}(\mathsf{op(a,b,c)}) ::= \mathsf{op}(G(\mathsf{a}),\mathsf{b},\mathsf{c}) \mid \mathsf{op}(\mathsf{a},G(\mathsf{b}),\mathsf{c}) \mid \mathsf{op}(\mathsf{a},\mathsf{b},G(\mathsf{c}))$$

Bounded arbitrary expression. We consider a grammar of interesting expressions of the given type and of limited depth. This grammar considers all operations in scope as well as all input variables. It also considers safe recursive calls discovered in Section 2.3. Finally, it includes the guiding expression as a terminal, which corresponds to possibly wrapping the source expression in an operation. For example, given a predicate $\Downarrow[\mathsf{listSum(Cons(h,t))}]$ and a mod function $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ in scope, an integer operation $\mathsf{op(a,b,c)}$ is generalized as:

$$\begin{aligned}
G_{*^2}(\mathsf{op(a,b,c)}) ::= \;& G_{Int2} \mid G_{Int1} \mid G_{Int0} \\
G_{Int2} ::= \;& G_{Int1} + G_{Int1} \\
\mid\; & G_{Int1} - G_{Int1} \\
\mid\; & \mathsf{mod}(G_{Int1}, G_{Int1}) \\
\mid\; & \mathsf{listSum(t)}
\end{aligned}
\qquad
\begin{aligned}
G_{Int1} ::= \;& G_{Int0} + G_{Int0} \\
\mid\; & G_{Int0} - G_{Int0} \\
\mid\; & \mathsf{mod}(G_{Int0}, G_{Int0}) \\
\mid\; & \mathsf{listSum(t)} \\
G_{Int0} ::= \;& 0 \mid 1 \mid \mathsf{h} \mid \mathsf{op(a,b,c)}
\end{aligned}$$

Our grammars cover a range of variations corresponding to common errors. During synthesis, the system generates a specific grammar for each invocation of this repair rule, and explores symbolically the space of all expressions in the grammar. We rely on a CEGIS-loop bootstrapped with our test inputs to explore these expressions. This can be abstractly represented by the following rule:

$$\text{CEGIS-Gen: } \frac{\exists \mathsf{T} \in \mathcal{L}(G(\mathsf{term}))\ \forall \bar{a}.\Pi \implies \phi[\bar{x} \mapsto \mathsf{T}]}{[\![\bar{a}\ \ \langle \odot[term] \wedge \Pi \rhd \phi \rangle\ \ \bar{x}]\!] \vdash \langle \mathsf{true} \mid \mathsf{T} \rangle}$$

Even though this rule is inherently incomplete, it is able to fix common errors efficiently. Our deductive approach allows us to introduce such tailored rules without loss of generality: errors that go beyond this model may be repaired using more general, albeit slower synthesis rules.

Precise handling of recursive calls in CEGIS. Our system uses a symbolic approach to avoid enumerating expressions explicitly [13]. When considering recursive calls among possible expressions within CEGIS, the interpretation of such calls needs to refer back to this same expression. Our previous approach [13] treats recursive invocations of the function under synthesis as satisfying only the postcondition, leading to spurious counter-examples. Our new solution first constructs a parametrized program explicitly representing the search space: given a grammar G at a certain unfolding level, we construct a function cTree($\bar{a}$, B) in which we describe non-terminals as values with each production guarded by a distinct entry of the B array, as in the following repair a case of the size function.

```
def cTree[T](h: T, t: List[T],
             B: Array[Boolean]) = {
  val c1 = if (B(0)) 0
           else if (B(1)) 1
           else if (B(2)) size(t, B)
           else _
  val c2 = if (B(3)) 0
           else if (B(4)) 1
           else if (B(5)) size(t, B)
           else _
  val c3 = if (B(6)) c1 + c2
           else if (B(7)) c1 - c2
           else _
  c3 }
```

```
def size[T](l: List[T],
            B: Array[Boolean]): Int = {
  l match {
    case Cons(h, t) ⇒ cTree(h, t, B)
    case Nil() ⇒ 0
  }
}

def nonEmpty(l: List[T],
             B: Array[Boolean]) = {
  size(l, B) > 0
}
```

In this new program, the function under repair is defined using the partial solution corresponding to the current deduction tree, in which we call cTree at the point of the CEGIS invocation. Other unsolved branches of the deduction tree become synthesis holes. We augment transitive callers with this additional B argument, passing it accordingly. This ensures that a specific valuation of B corresponds exactly to a program where the point of CEGIS invocation is replaced by the corresponding expression. We rely on tests collected in Section 4 to test individual valuations of B, removing failing expression from the search space. Fi-

nally, we perform CEGIS using symbolic term exploration with the SMT solver to find candidate expressions [13].

4 Generating and Using Tests for Repair

Tests play an essential role in our framework, allowing us to gather information about the valid and invalid parts of the function. In this section we elaborate on how we select, generate, and filter examples of inputs and possibly outputs. Several components of our system then make use of these examples. We distinguish two kinds of tests: input tests and input-output tests. Namely, input tests provide valid inputs for the function according to its precondition, while input-output tests also specify the exact output corresponding to each input.

Extraction and Generation of Tests. Our system relies on three main sources for tests that are used to make the repair process more efficient.

1) User-provided symbolic input-output tests. It is often interesting for the user to specify how a function behaves by listing a few examples providing inputs and corresponding outputs. However, having to provide full inputs and outputs can be tedious and impractical. To make specifying families of tests convenient, we define a **passes** construct to express input-output examples, relying on pattern matching in our language to symbolically describe sets of inputs and their corresponding outputs. This gives us an expressive way of specifying classes of input-output examples. Not only may the pattern match more than one input, but the corresponding outputs are given by an expression which may depend on the pattern's variables. Wildcard patterns are particularly useful when the function does not depend on all aspects of its inputs. For instance, a function computing the size of a generic list does not inspect the values of individual list elements. Similarly, the sum of a list of integers could be specified concisely for all lists of sizes up to 2. Both examples are illustrated by Figure 2.

```
def size[T](list: List[T]): Int = {
  list match {
    case Nil() ⇒ 0
    case Cons(h, t) ⇒ 1 + size(t)
  }
} ensuring { res ⇒
  (res ≥ 0) && 
  (list, res) passes {
    case Cons(_, Cons(_, Nil())) ⇒ 2
    case Cons(_, Nil()) ⇒ 1
    case Nil() ⇒ 0 } }
```

```
def sum(list: List[Int]): Int = {
  list match {
    case Nil() ⇒ 0
    case Cons(h, t) ⇒ h + sum(t)
  }
} ensuring { res ⇒
  (list, res) passes {
    case Cons(a, Cons(b, Nil())) ⇒ a + b
    case Cons(a, Nil()) ⇒ a
    case Nil() ⇒ 0 } }
```

Fig. 2. Partial specifications using the passes construct, allowing to match more than one inputs and providing the expected output as an expression.

Having partially symbolic input-output examples strikes a good balance between literal examples and full-functional specifications. From the symbolic tests, we generate concrete input-output examples by instantiating each pattern several

times using enumeration techniques, and executing the output expression to yield an output value. For instance, from case Cons(a, Cons(b, Nil())) ⇒ a + b we will generate the following tests resulting from replacing a, b with all combinations of values from a finite set, including, for example, test with input Cons(1, Cons(2, Nil())) and output 3. We generate up to 5 distinct tests per pattern, when possible. These symbolic specifications are the only forms of tests provided by the developer; any other tests that our system uses are derived automatically.

2) Generated Input Tests. We rely on the same enumeration technique to generate inputs satisfying the precondition of the function. Using a generate and test approach, we gather up to 400 valid input tests in the first 1000 enumerated.

3) Solver-generated Tests. Lastly, we rely on the underlying solvers for recursive functions of Leon [25] to generate counter-examples. Given that the function is invalid and that it terminates, the solver (which is complete for counter-examples) is guaranteed to eventually provide us with at least one failing test.

Classifying and Minimizing Traces. We partition the set of collected tests into passing and failing sets. A test is considered as failing if it violates a precondition, a postcondition, or emits one of various other kinds of runtime errors when the function to repair is executed on it. In the presence of recursive functions, a given test may fail within one of its recursive invocations. It is interesting in such scenarios to consider the arguments of this specific sub-invocation: they are typically smaller than the original and are better representatives of the failure. To clarify this, consider the example in Figure 3 (based on the program in Figure 1):

```
def desugar(e : Expr) : SExpr = e match { ...
  case And(lhs, rhs) ⇒ // correct
    SIte(desugar(lhs), desugar(rhs), SLiteral(0))
  case Ite(cond, thn, els) ⇒ // correct
    SIte(desugar(cond), desugar(thn), desugar(els))
  case BooleanLiteral(b) ⇒ // buggy
    if (b) SLiteral(0) else SLiteral(1)
}
```

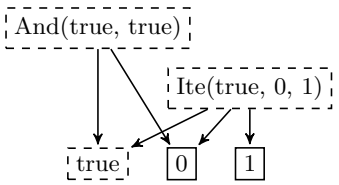

Fig. 3. Code and invocation graph for desugar. Solid borderlines stand for passing tests, dashed ones for failing ones. Type constructors for literals have been omitted.

Assume the tests collected are And(BooleanLiteral(**true**), BooleanLiteral(**true**)), Ite(BooleanLiteral(**true**), IntLiteral(0), IntLiteral(1)) and BooleanLiteral(**true**). When executed with these tests, the function produces the graph of eval invocations shown on the right of Figure 3. A trivial classification tactic would label all three tests as faulty, even though it is obvious that all errors can be explained by the bug in BooleanLiteral, due to the dependencies between tests. More generally, *a failing test should also be blamed for the failure of all other tests that invoke it transitively.* Our framework deploys this smarter classification. Thus, in our example, it would only label BooleanLiteral(**true**) as a failing example, which would lead to correct localization of the problem on the faulty branch. Note that this process

will discover new failing tests not present in the original test set, if they occur as recursive sub-invocations.

Our experience with incorporating tests into the Leon system indicate that they are proving time and again to be extremely important for the tool's efficiency, even though our system is in its spirit based on verification as opposed to testing alone. In addition to allowing us to detect errors sooner and filter out wrong synthesis candidates, tests also allow us to quickly find the approximate error location.

5 Evaluation

We evaluate our implementation on a set of benchmarks in which we manually injected errors (Figure 4). The programs mainly focus on data structure implementations and syntax tree operations. Each benchmark is comprised of algebraic data-type definitions and recursive functions that manipulate them, specified using strong yet still partial preconditions and postconditions. We manually introduced errors of different types in each copy of the benchmarks. We ran our tool unassisted until completion to obtain a repair, providing it only with the name of the file and the name of the function to repair (typically the choice of the function could also have been localized automatically by running the verification on the entire file). The experiments were run on an Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz with 16GB RAM, with 2GB given to the Java Virtual Machine. While the deductive reasoning supports parallelism in principle, our implementation is currently single-threaded.

For each benchmark of Figure 4 we provide: (1) the name of the benchmark and the broken operation; (2) a short classification of the kind of error introduced. The error kinds include: a small variation of the original program, a completely faulty **match**-case, a missing **match**-case, a missing necessary **if**-split, a missing function call, and finally, two separate variations in the same function. We describe the relevant sizes (counted in abstract syntax tree nodes) of: (3) the overall benchmark, (4) the erroneous function, (5) the localized error, and (6) the repaired expression. The full size of the program is relevant because our repair algorithm may introduce calls to any function defined in the benchmark, and also because the verification of a function depends on other functions in the file (recall Figure 1). We also include the time, in seconds, our tool took to: (7) collect and classify tests and (8) repair the broken expression. Finally, we report (9) if the system could formally (and automatically) prove the validity of the repaired implementation. Our examples are challenging to verify, let alone repair. They contain both functional and test-based specifications to capture the intended behavior. Many rely on unfolding procedure of [24, 25] to handle contracts that contain other auxiliary recursive functions. The fast exponentiation algorithm of Numerical.power relies on non-linear reasoning of the Z3 SMT solver [4].

An immediate observation is that fault localization is often able to focus the repair to a small subset of the body. Combined with the symbolic term explo-

Operation	Error	Size				Time (sec)		Proof
		Prg	Fun	Err	Fix	Test	Repair	Success
`Compiler.desugar1`	full case	1335	81	3	5	1.2	2.2	✓
`Compiler.desugar2`	full case	1330	79	2	8	1.0	10.2	✓
`Compiler.desugar3`	variation	1324	83	7	7	0.9	1.6	✓
`Compiler.desugar4`	variation	1324	83	7	7	1.4	1.7	✓
`Compiler.desugar5`	2 variations	1458	83	83	83	1.4	14.0	✓
`Compiler.simplify1`	variation	1458	30	4	4	0.8	1.7	✓
`Compiler.simplify2`	variation	1464	30	2	2	0.8	1.7	✓
`Heap.merge1`	if cond	1084	36	3	3	1.9	3.0	✓
`Heap.merge2`	variation	1084	36	1	1	1.1	1.4	✓
`Heap.merge3`	if cond	1084	36	3	3	1.9	3.1	✓
`Heap.merge4`	variation	1084	36	6	6	1.2	2.4	✓
`Heap.merge5`	if cond	1086	38	5	7	1.2	3.0	✓
`Heap.merge6`	2 variations	1084	36	36	36	1.5	12.7	
`Heap.insert`	variation	1086	8	8	6	5.2	1.4	✓
`Heap.makeNode`	variation	1086	16	7	5	2.2	1.3	✓
`List.pad`	variation	1157	34	8	6	1.0	1.4	✓
`List.++`	variation	1153	9	3	5	2.5	1.1	✓
`List.:+`	full case	1161	11	1	3	1.8	1.2	✓
`List.replace`	full case	1172	14	3	13	1.8	11.2	✓
`List.count`	variation	1185	16	3	5	0.9	1.5	✓
`List.find1`	variation	1175	21	2	4	3.0	3.8	
`List.find2`	variation	1177	23	4	6	3.0	3.7	
`List.find3`	if cond	1178	24	18	17	4.8	5.9	
`List.size`	variation	1157	10	4	4	1.7	1.2	✓
`List.sum`	variation	1175	10	4	4	1.3	1.2	✓
`List.delete`	missing call	1162	16	1	3	1.5	1.1	✓
`List.drop`	2 variations	1166	21	21	27	1.5	16.6	✓
`PropLogic.nnf1`	missing call	915	51	1	3	0.7	1.3	✓
`PropLogic.nnf2`	missing case	911	47	1	13	0.9	3.4	✓
`PropLogic.nnf3`	variation	916	51	2	4	0.9	1.2	✓
`PropLogic.nnf4`	variation	920	52	5	5	0.8	1.3	✓
`PropLogic.nnf5`	full case	916	48	1	5	0.9	1.7	✓
`Numerical.power`	variation	133	23	5	7	0.3	1.2	✓
`Numerical.moddiv`	variation	186	30	3	3	0.3	1.1	✓
`MergeSort.split`	full case	221	28	3	7	2.0	3.3	✓
`MergeSort.merge1`	variation	951	32	5	5	1.7	1.3	✓
`MergeSort.merge2`	variation	951	32	3	3	1.8	1.9	✓
`MergeSort.merge3`	variation	949	30	3	5	1.5	1.5	✓
`MergeSort.merge4`	2 variations	951	32	32	32	1.8	21.1	

Fig. 4. Automatically repaired functions using our system. We provide for each operation: a small description of the kind of error introduced, the overall program size, the size of the invalid function, the size of the erroneous expression we locate and the size of the repaired version. We then provide the times our tool took to: gather and classify tests, and repair the erroneous expression. Finally, we mention if the resulting expression verifies. The source of all benchmarks can be found on `http://lara.epfl.ch/w/leon-repair` (see also `http://leon.epfl.ch`)

ration, this translates to a fast repair if the error fell within the error model. Among the hardest benchmarks are the ones labeled as having "2 variations". For example, Compiler.desugar5 is similar to one in Figure 1 but contains two errors. In those cases, localization returns the entire **match** as the invalid expression. Our guided repair uses the existing **match** as the guide and successfully resynthesizes code that repairs both erroneous branches. Another challenging example is Heap.merge3, for which the more elaborate If-Focus-Condition rule of Section 2.1 kicks in to resynthesize the condition of the **if** expression.

The repairs listed in evaluation are not only valid according to their specification, but were also manually validated by us to match the intended behavior. A failing proof thus does not indicate a wrong repair, but rather that our system was not able to automatically derive a proof of its correctness, often due to insufficient inductive invariants. We identify three scenarios under which repair itself may not succeed: if the assumptions mentioned in Section 2 are violated, when the necessary repair is either too big or outside of the scope of general synthesis, or if test collection does not yield sufficiently many interesting failing tests to locate the error.

6 Further Related Work

Much of the prior work focused on imperative programming, without native support for algebraic data types, making it typically infeasible to even automatically verify data structure properties of the kind that our benchmarks contain. Syntax-guided synthesis format [1, 2] does not support algebraic data types, or specific notion of repair (it could be used to specify some of the sub-problems that our system generates, such those of Section 3).

GenProg [7] and SemFix [17] accept as input a C program along with user-provided sets of passing and failing test cases, but no formal specification. Our technique for fault localization is not applicable to a sequential program with side-effects, and these tools employ statistical fault localization techniques, based on program executions. GenProg applies no code synthesis, but tries to repair the program by iteratively deleting, swapping, or duplicating program statements, according to a genetic algorithm. SemFix, on the other hand, uses synthesis, but does not take into account the faulty expression while synthesizing. AutoFix-E/E2 [18] operates on Eiffel programs equipped with formal contracts. Formal contracts are used to automatically generate a set of passing and failing test cases, but not to verify candidate solutions. AutoFix-E uses an elaborate mechanism for fault localization, which combines syntactic, control flow and statistical dynamic analysis. It follows a synthesis approach with repair schemas, which reuse the faulty statement (e.g. as a branch of a conditional). Samanta et al. [20] propose abstracting a C program with a boolean constraint, repairing this constraint so that all assertions in the program are satisfied by repeatedly applying to it update schemas according to a cost model, then concretize the boolean constraint back to a repaired C program. Their approach needs developer intervention to define the cost model for each program, as well as at the

concretization step. Logozzo et al. [16] present a repair suggestion framework based on static analysis provided by the CodeContracts static checker [5]; the properties checked are typically simpler than those in our case. In [6], Gopinath et al. repair data structure operations by picking an input which exposes a suspicious statement, then using a SAT-solver to discover a corresponding concrete output that satisfies the specification. This concrete output is then abstracted to various possible expressions to yield candidate repairs, which are filtered with bounded verification. In their approach, Chandra et al. [3] consider an expression as a candidate for repair if substituting it with some concrete value fixes a failing test.

Repair has also been studied in the context of reactive and pushdown systems with otherwise finite control [8, 10, 11, 19, 20, 26]. In [26], the authors generate repairs that preserve explicitly subsets of traces of the original program, in a way strengthening the specification automatically. We deal with the case of functions from inputs to outputs equipped with contracts. In case of a weak contract we provide only heuristic guarantees that the existing behaviors are preserved, arising from the tendency of our algorithm to reuse existing parts of the program.

7 Conclusions

We have presented an approach to program repair of mutually recursive functional programs, building on top of a deductive synthesis framework. The starting point gives it the ability to verify functions, find counterexamples, and synthesize small fragments of code. When doing repair, it has proven fruitful to first localize the error and then perform synthesis on a small fragment. Tests proved very useful in performing such localization, as well as for generally speeding up synthesis and repair. In addition to deriving tests by enumeration and verification, we have introduced a specification construct that uses pattern matching to describe symbolic tests, from which we efficiently derive concrete tests without invoking full-fledged verification. In case of tests for recursive functions, we perform dependency analysis and introduce new ones to better localize the cause of the error. While localization of errors within conditional control flow can be done by analyzing test runs, the challenge remains to localize change inside large expressions with nested function calls. We have introduced the notion of *guided synthesis* that uses the previous version of the code as a guide when searching for a small change to an existing large expression. The use of a guide is very flexible, and also allows us to repair multiple errors in some cases.

Our experiments with benchmarks of thousands of syntax tree nodes in size, including tree transformations and data structure operations confirm that repair is more tractable than synthesis for functional programs. The existing (incorrect) expression provides a hint on useful code fragments from which to build a correct solution. Compared to unguided synthesis, the common case of repair remains more predictable and scalable. At the same time, the developer need not learn a notation for specifying holes or templates. We thus believe that repair is a practical way to deploy synthesis in software development.

References

1. R. Alur, R. Bodik, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. To Appear in Marktoberdrof NATO proceedings, 2014. `http://sygus.seas.upenn.edu/files/sygus_extended.pdf`, retrieved 2015-02-06.
2. R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.
3. S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In R. N. Taylor, H. C. Gall, and N. Medvidovic, editors, *ICSE*, pages 121–130. ACM, 2011.
4. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
5. M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*, pages 10–30. Springer, 2011.
6. D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *LNCS*, pages 173–188. Springer, 2011.
7. C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TSE*, 38(1):54–72, 2012.
8. A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to C. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 358–371. Springer, 2006.
9. T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.
10. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 226–238. Springer, 2005.
11. B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *JCSS*, 78(2):441–460, 2012.
12. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 437–446. ACM, 2011.
13. E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *OOPSLA*, pages 407–426. ACM, 2013.
14. R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In P. Bjesse and A. Slobodová, editors, *FMCAD*, pages 91–100. FMCAD Inc., 2011.
15. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional synthesis for linear arithmetic and sets. *STTT*, 15(5-6):455–474, 2013.
16. F. Logozzo and T. Ball. Modular and verified automatic program repair. In G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 133–146. ACM, 2012.
17. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE*, pages 772–781. IEEE / ACM, 2013.
18. Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based automated program fixing. *ArXiv e-prints*, 2011. `arXiv:1102.1059`.

19. R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In A. Cimatti and R. B. Jones, editors, *FMCAD*, pages 1–10. IEEE, 2008.
20. R. Samanta, O. Olivo, and E. A. Emerson. Cost-aware automatic program repair. In M. Müller-Olm and H. Seidl, editors, *SAS*, volume 8723 of *LNCS*, pages 268–284. Springer, 2014.
21. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
22. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
23. S. Srivastava, S. Gulwani, and J. S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.
24. P. Suter. *Programming with Specifications*. PhD thesis, EPFL, December 2012.
25. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.
26. C. von Essen and B. Jobstmann. Program repair without regret. In *CAV*, pages 896–911, 2013.
27. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *TSE*, 28(2):183–200, 2002.