Programming with Angelic Nondeterminism

Shaon Barman[†] Rastislav Bodík[†] Satish Chandra[⋆] Joel Galenson[†]
Doug Kimelman[⋆] Casey Rodarmor[†] Nicholas Tung[†]

[†]University of California, Berkeley [⋆]IBM T.J. Watson Research Center

Abstract

Angelic nondeterminism can play an important role in program development. It simplifies specifications, for example in deriving programs with a refinement calculus; it is the formal basis of regular expressions; and Floyd relied on it to concisely express backtracking algorithms such as *N-queens*.

We show that angelic nondeterminism is also useful during the development of *deterministic* programs. The semantics of our angelic operator are the same as Floyd's but we use it as a substitute for yet-to-be-written deterministic code; the final program is fully deterministic. The angelic operator divines a value that makes the program meet its specification, if possible. Because the operator is executable, it allows the programmer to test incomplete programs: if a program has no safe execution, it is already incorrect; if a program does have a safe execution, the execution may reveal an implementation strategy to the programmer.

We introduce refinement-based angelic programming, describe our embedding of angelic operators into Scala, report on our implementation with bounded model checking, and describe our experience with two case studies. In one of the studies, we use angelic operators to modularize the Deutsch-Schorr-Waite (DSW) algorithm. The modularization is performed with the notion of a *parasitic stack*, whose incomplete specification was instantiated for DSW with angelic nondeterminism.

Categories and Subject Descriptors D.2.1 [*Requirements/Specifications*]: Methodologies; D.2.2 [*Tools and Techniques*]: Top-down programming; D.2.4 [*Software/Program Verification*]: Validation

General Terms Design, Languages, Verification

Keywords Angelic non-determinism, constraints, bounded model-checking, traces, refinement

1. Introduction

Model checking and testing leverage compute power to validate *complete* programs. However, much less work has addressed the issue of how to assist with the *construction* of a program. This paper proposes using the clairvoyance of angelic nondeterminism in the solving of programming problems. The programmer encodes his partial understanding of the programming problem in an angelic program, relying on the (executable) nondeterministic choose operator to produce values that the programmer is as yet unable to compute. The system answers whether the angelic program has a safe execution for a given input; if so, the system outputs the program's traces. The process of programming with angelic nondeterminism amounts to (1) testing hypotheses about plausible solutions by formulating angelic programs and, as the understanding develops, (2) gradually refining the angelic program until all angelic nondeterminism is removed.

An angelic program is a program with a choose operator. This nondeterministic operator was first proposed by Floyd [7], who intended it as programming abstraction for hiding implementation details of backtracking search, with the goal of allowing abstract specifications of algorithms such as *N-queens*. The choose operator divines a value that makes the program terminate without violating an assertion, if possible. We say that an angelic program is *correct* if such a safe execution exists for all inputs. The operator is nondeterministic because it can evaluate to *any* suitable value; it is angelic because it collaborates with the program against the environment (i.e., the input). The input sequence can be seen as demonic nondeterminism, finding a value that might make the program fail. The choose operator is clairvoyant in that it looks ahead into the execution and returns a value that allows further choose operators to return values that lead to a successful execution, if possible. Angelic nondeterminism can be a deductive device (e.g., [4]) or it can be executable [7], with implementations relying on backtracking or constraint solving (Section 7).

Our choose operator is executable but, unlike Floyd, we propose to use it only during program development. It is used to express partially-developed programs; final programs are choose-free. The choose operator stands for code that the programmer has not yet implemented, either because he does not yet know how to implement it or because he does not even know what values it should produce. The choose operator produces values based on a correctness condition; the final program will compute these values with deterministic code fragments.

Executable angelic nondeterminism aids programming in several ways. First, the programmer can test hypotheses about his implementation strategies. He does so by formulating an angelic program, using the choose operators as substitutes for as yet unimplemented code fragments. An example of a hypothesis that the programmer may test is whether there exists a way to satisfy a postcondition in at most n iterations of a loop; the loop will typically have angelic statements in the body. If no safe angelic execution exists, the hypothesis is rejected. The operator represents the most general way of computing a subtask,

POPL'10, January 17–23, 2010, Madrid, Spain.

but the programmer can declaratively (with assertions) constrain it in ways that reflect his hypothesis. For example, the operator may be prevented from visiting a graph node multiple times. The envisioned benefit to the programmer is that the implementation strategy can be safely aborted before the programmer understood its infeasibility on his own.

Second, if the angelic program is correct, the system outputs its safe traces. The traces serve as demonstrations of how to perform the desired task, for example: how to rotate nodes to establish the red-black property. Given these demonstrations, the programmer is left with the hopefully simpler task of generalizing the traces into an algorithm that works for all inputs. These demonstrations represent vast amounts of combinatorial reasoning that the programmer may otherwise have to carry out on his own. The envisioned benefit of angelic demonstrations is a revelation of an insight into how to carry out a computation under given time or space constraints, potentially leading to faster discovery of the algorithm.

An angelic program may, of course, generate many alternative traces, and most of the traces may be the result of choose operators "abusing" their clairvoyance to complete the computation in an input-dependent fashion that does not correspond to an easily encodable deterministic algorithm. The programmer can exclude such irregular traces with assertions that limit nondeterministic choices. For example, the operator may be required to satisfy a data structure consistency invariant or to evaluate only to values that have already been stored in memory.

Third, nondeterminism allows refinement-based programming. The programmer can develop the program gradually in several ways. As mentioned above, by adding assertions, he can prune the set of safe traces, working towards a trace that corresponds to a desired algorithm. The programmer can also focus on one subproblem at a time, ignoring those subproblems expressed with choose. In general, it is not possible to refine angelic operators independently of each other, but as we show in Section 5.2, sometimes we can add sufficient constraints in the program for them to be independent. The programmer can also implement the choose operator with a subprogram that is itself nondeterministic, applying the methodology recursively until a deterministic program is obtained.

Finally, the clairvoyance of the choose operator helps the programmer avoid complex global reasoning. In Section 8, we rely on this property to modularize the Deutsch-Schorr-Waite (DSW) algorithm [15]. We refactor the algorithm to hide its backtracking logic in a *parasitic stack*, a data structure that stores its values in memory locations borrowed from the host data structure, i.e., the graph being traversed. The hard decisions of which memory locations the parasitic stack is allowed to borrow and how to restore their values are left to choose operators. The programmer has to solve only the more local problem of how to implement the stack with these borrowed locations. In fact, without the choose, we were unable to carry out this new modularization of DSW.

This paper makes these contributions:

- We propose a program development methodology based on refinement of angelic programs. In contrast to previous work in program refinement, our methodology permits the execution of incomplete programs, which helps a programmer in several ways, from rejecting infeasible implementation strategies to making reasoning more local.
- We add the choose construct to the Scala programming language [13]. We describe our implementation of angelic nondeterminism. We also present experimental results on the efficiency of two different implementation strategies, one based on backtracking and one on SAT solving.
- We present case studies of two classic problems: the Dutch Flag problem [6, 10] and the Deutsch-Schorr-Waite graph marking algorithm [15]. For the second problem, angelic programming allowed us to develop a new way of modularizing the algorithm that we believe is novel. We also include a case study on recursive list manipulation.

Section 2 gives an overview of the programming methodology using the case study of the Dutch Flag problem. Sections 3–5 formalize the methodology and give refinement transformations. Section 6 compares our (algorithmic) methodology with the (deductive) refinement of Morgan. Section 7 describes our implementation and Section 8 reports on the case study of the DSW algorithm. Related work is described in Section 9.

2. Overview

This section gives an overview of programming with angelic nondeterminism on the Dutch Flag problem [6]. The section does not give a beautified tutorial example; instead, we present rather faithfully how a programmer used the programming methodology. We highlight three aspects of angelic programming:

- How an angelic program can express a hypothesis about the programmer's implementation plan, which will be rejected if it is guaranteed not to lead to a correct implementation.
- How the programmer improves his understanding of the problem by observing angelic executions.
- How angelic programs are refined by constraining the angels and by deterministically implementing them.

Dijkstra presents the Dutch Flag problem in [6]: given an array of n pebbles, each of which is either red, white, or blue, the algorithm must sort them in-place, in order of the colors in the Dutch Flag: first red, then white, then blue. The algorithm must examine the color of each pebble at most once and can only move the pebbles by swapping. A crucial constraint is that only a constant amount of storage can be used to remember the color of pebbles that have been examined.

In traditional algorithm development, a programmer must implement the entire algorithm first before he can test it. Perhaps he can proceed bottom up, testing small subroutines first. With angelic nondeterminism, one can proceed top down, starting with a very nondeterministic program that implements coarse steps of the entire algorithm. In the case of the Dutch Flag problem, the programmer starts with this initial angelic program, which tests the hypothesis that the problem can indeed be solved with a sequence of swap operations.

Program P_0

```
while ( choose ) {
  swap(choose(n), choose(n))
}
assert isCorrect
```

The choose operator nondeterministically chooses a Boolean value, and the expression choose(n) nondeterministically chooses a value from the range $[0, n)$.

This angelic program is correct (i.e. the angels can find a safe execution that validates the hypothesis).[1] In thinking about how

[1] Our tester validates the hypothesis on only a small set of inputs, so the best we can actually ascertain is that the hypothesis has not been rejected

to refine the program, the programmer prints information about the choices being made by the angels.

Program P_0'

```
angelicprint(input)
while ( choose ) {
  i = choose(n)
  j = choose(n)
  swap(i,j)
  angelicprint(i,j)
}
assert isCorrect
```

The call angelicprint prints its arguments, but only on successful executions. Henceforth, assume that all of our programs are instrumented to print inputs and the indices of swapped pebbles on safe executions into a log. The following is the output of one of the safe executions of this program:

```
Input: bwrwrwb
(0,2) (1,4) (2,5).
```

The following are swap sequences from two of the thousands of other safe executions, one line per safe execution:

```
(4,0) (4,5) (2,1).
(0,1) (1,5) (4,1) (2,0).
```

In examining the executions, the programmer focuses on which pebbles were swapped. Not surprisingly, he discovers that the highly nondeterministic program P_0 swaps pebbles seemingly arbitrarily.

Now the programmer hypothesizes the first plausible implementation strategy. Namely, he posits that it is possible to traverse the array of pebbles left-to-right while swapping the current pebble with a suitable counterpart.

Program P_1 – refines Program P_0

```
i = 0
while ( choose ) {
  swap(i, choose(n))
  i += 1
}
assert isCorrect
```

In P_1, a value that was previously provided by an angelic operator (the first argument to swap in P_0) is now computed by deterministic code. We say that choose(n) was *implemented* by this deterministic code. Technically, P_1 is a refinement of P_0 in that it removes some of the nondeterministic choices present in P_0. In other words, P_1 generates a subset of executions of P_0.

Program P_1 is correct, too. Below, we show the log from one of the safe executions.

```
Input: bwrwrwb
(0,6) (1,0) (2,0) (3,1) (4,1) (5,3) (6,5).
```

P_1 still generates too many traces (the angels have too much freedom) but the programmer notices a pattern in some of the successful traces: towards the end of each run, the angels are swapping red pebbles to the left end of the array and blue pebbles to the right end. The programmer hypothesizes that a more deterministic program would be explicit about gathering red pebbles at the left end of the array and blue pebbles at the right from the very beginning.

This observation leads to an implementation strategy, encoded in the next program, where the programmer implements two zones of uniform color: red at the left end of the array and blue at the right. The programmer further hypothesizes that a zone of white pebbles in the middle of the array will simply "fall out" as a consequence. (The idea of three zones turns out to be exactly the insight that lead Dijkstra to his algorithm [6].)

Program P_2

```
i = 0
R = 0 // Pebbles to the left of R are red
B = n−1 // Pebbles to the right of B are blue
while ( choose ) {
  c = pebbles(i) // examine color
  if (c == red) { j = R; R += 1 }
  else if (c == blue) { j = B; B −= 1 }
  else /* (c == white) */ { j = i }
  swap(i, j)
  i += 1
}
assert isCorrect
```

In the next step, captured in program P_2, the angelic operator that provided the second argument to swap in P_1 has been replaced by deterministic code. Program P_2 is, however, not correct, as it has no successful traces. P_2 is therefore not a proper refinement of P_1. Below is a log from a prefix of a failed trace:

```
Input: bwrwrwb
(0,6) (1,1) (2,0) (3,3) (4,1).
```

Examining the log, the programmer can see a problem: after the swap (2,0), a blue pebble is brought into position 2, and the algorithm then proceeds to examine the pebble in position 3, leaving the blue pebble in position 2. This pebble is never revisited, and hence is never swapped to the right end of the array where it belongs.

Generalizing from this observation, the programmer decides that in some cases a pebble brought into position i may need to be handled before moving on to position $i+1$. He decides to handle this by not advancing i in some cases. Since the programmer is unsure of exactly when to advance i, he uses an angelic choose operator to make the decision.

The programmer returns to P_1, which encoded an infeasible implementation strategy because it allowed at most one swap per position of the array. The programmer creates P_{1a}—an alternative version of P_1 that is a less restrictive refinement of P_0.

Program P_{1a} – refines Program P_0

```
i = 0
while ( choose ) {
  j = choose(n)
  swap(i, j)
  if ( choose ) i += 1
}
assert isCorrect
```

P_{1a} is also correct.

The programmer now again applies the refinement that introduces zones of uniform color.

on those inputs. Even if correctness is proven on all inputs, there is of course no guarantee that a deterministic algorithm exists that can replace the nondeterministic operators.

Program P_3 – refines Program P_{1a}

```
i = 0
R = 0 // Pebbles to the left of this are red
B = n−1 // Pebbles to the right of this are blue
while ( choose ) {
  c = pebbles(i) // examine color
  if (c == red) { j = R; R += 1 }
  else if (c == blue) { j = B; B −= 1 }
  else /* (c == white) */ { j = i }
  swap(i, j)
  if ( choose ) i += 1
}
assert isCorrect
```

One log for this correct program, including the values of i as well as swaps, is:

```
Input: bwrwrwb
i=0:(0,6) i=0:(0,5) i=1:(1,1) i=2:(2,0) i=3:(3,3) i=4:(4,1).
```

The next step is to implement the nondeterministic operator that guards the advancement of i. Logs reveal that i is not advanced when the current pebble (i.e., the i^{th} pebble before the swap) is blue. On reflection, this is reasonable because when the current pebble is blue, it is swapped with a pebble from farther right in the array—a pebble whose color has not yet been examined and is not yet known. In contrast, when the current pebble is red, it is swapped with a pebble of a known color.

The programmer now implements the angelic operator with deterministic code, creating refinement P_4.

Program P_4 – refines Program P_3

```
i = 0
R = 0 // Pebbles to the left of this are red
B = n−1 // Pebbles to the right of this are blue
while ( choose ) {
  c = pebbles(i) // examine color
  if (c == red) { j = R; R += 1 }
  else if (c == blue) { j = B; B −= 1 }
  else /* (c == white) */ { j = i }
  swap(i, j)
  if (c != blue) i += 1 // changed from choose in P3
}
assert isCorrect
```

The final refinement must implement the non-deterministic loop bound. The programmer might rashly replace the bound with `i < n`. In this case, the programmer would find that the program is incorrect, because i would overrun the blue zone. Logs would reveal that the correct bound is `i <= B`, as shown in P_5, which is the final deterministic program.

Program P_5 – the final program; refines Program P_4

```
i = 0
R = 0 // Pebbles to the left of this are red
B = n−1 // Pebbles to the right of this are blue
while ( i <= B ) { // changed from choose in P4
  c = pebbles(i) // examine color
  if (c == red) { j = R; R += 1 }
  else if (c == blue) { j = B; B −= 1 }
  else /* (c == white) */ { j = i }
  swap(i, j)
  if (c != blue) i += 1
}
assert isCorrect
```

The development of this solution to the Dutch Flag problem showed how the programmer can pose hypotheses about his implementation plans by writing and testing angelic programs and how he can develop the final deterministic program by gradually creating refinements that involve implementing angelic constructs with deterministic code.

3. An Angelic Programming Language

In this section we describe the syntax and semantics of a small programming language for writing angelic programs. Note that our actual implementation is written as an extension to Scala.

3.1 Core language

We consider the following core language:

$$
\begin{array}{lcll}
stmt & ::= & v \;=\; expr & (Assign) \\
 & & \mathsf{assert}\ b & (Assert) \\
 & & stmt_1\ ;\ stmt_2\ ;\ \ldots\ ;\ stmt_n & (Sequence) \\
 & & \mathsf{if}\ (b)\ \{\ stmt\ \}\ \mathsf{else}\ \{\ stmt\ \} & (Conditional) \\
 & & \mathsf{while}\ (b)\ \{\ stmt\ \} & (Loop) \\
expr & ::= & \mathsf{choose} & (Angelic) \\
 & & \ldots &
\end{array}
$$

Here v is a variable and b is a boolean-valued expression. (The full language supports arrays and heap operations as well, but their handling is routine and is ignored in this section.) A program fails on assert $false$.

The only source of nondeterminism in the language is *Angelic*, which guesses a value in the domain of integers, addresses or booleans as appropriate in the context. The values of choose expressions are chosen so that, if at all possible, the execution terminates without failing any assert statement encountered in the execution.

3.2 Semantics

Following previous work [3], we give semantics of this language using the wp predicate transformer [6]. Given a statement S and a postcondition R, $wp(S, R)$ is the *w*eakest *p*recondition such that when S executes in a state satisfying that precondition, the execution results in a state satisfying R. wp for statements in the core language are straightforward:

$$
\begin{array}{rcl}
wp(v \;=\; e, R) & = & R[e/v] \\
wp(v \;=\; \mathsf{choose}, R) & = & \exists v.R \\
wp(\mathsf{assert}\ b, R) & = & b \wedge R \\
 & \ldots &
\end{array}
$$

The wp for assignment substitutes free occurrences of v in the postcondition with the expression e. The wp for choose binds free occurrences of v in R by existentially quantifying it: this expresses the notion that if possible, an angel will supply a value such that R is true. The assert statement conjoins its condition b to the postcondition, so that the program fails if b is false. The semantics of the control-flow statements are routine and are omitted.[2]

3.3 Program correctness

A program is assumed to be parameterized by an input vector $\vec{I}$, which binds the initial values of variables. A program is assumed to check for expected postconditions using assert statements. It is not necessary, though, that assert statements are placed only at the end of a program.

A *correct* program is one for which $\forall \vec{I}.wp(P(\vec{I}), true)$ is satisfiable. (We assume that all variables are defined before being

[2] We assume that the programmer guarantees the existence of ranking functions such that loops terminate independently of choose expressions.

read, so that the formula above does not contain any free variables.) The formula says that on any input there is a way in which the angel can choose values for choose expressions such that no assertion is violated in an execution of the program.

4. Traces and Program Refinement

A *trace* is a sequence of values produced by choose expressions during a *safe*—i.e. not assert-failing and terminating—execution of a program. Given an input and a recorded trace, the execution of an angelic program can be reconstructed faithfully. Because the program's postconditions may only care about the final state of variables, a particular input might result in multiple safe traces.

Define $safetraces(P(\vec{I}))$ to be the set of safe traces for a program P on input $\vec{I}$. P is *correct* if

$$\forall \vec{I}.(safetraces(P(\vec{I})) \neq \{\})$$

This correctness condition corresponds to the *wp*-based condition from the previous subsection: each trace corresponds to the values which when assigned to existentially quantified variables in the *wp*-based condition would satisfy the condition.

The purpose of refinement is to decrease dependence on the angel while carrying out effectively the same computation steps as those computed by the angels. In particular, executions that a programmer ruled out by eliminating choices available to an angel (e.g., by adding assertions) should not reappear in successor programs. If programs P and Q contain identical set of choose statements, then for Q to *refine* P, it must be the case that:

$$\forall \vec{I}.safetraces(Q(\vec{I})) \subseteq safetraces(P(\vec{I}))$$

In general Q will not contain an identical set of choose expressions as P, because a programmer typically eliminates some choose expressions from P, implementing them in Q using deterministic code or possibly using additional choose expressions. We define a set of program transformations that a programmer is allowed to use for refinements, and for which we give a suitably adjusted trace containment property.

4.1 Program transformations

We define three program transformations that, in combination, can be used to determinize an angelic program. Two of the transformations (T1 and T3) are correctness-preserving, i.e., they preserve safe traces of an angelic program, while the correctness of T2 must be validated with a checker (see Section 4.2).

T1. State inflation Consider a pair of angelic programs P and Q. We say that Q is obtained from P by state inflation if Q adds program variables and statements (including choose) in such a way that Q does not alter the traces of P: there is no change in data or control dependence of any of the statements of P. Let $proj_P$ represent a projection of safe traces of Q to the choose operators in P (all of which are also in Q). Then,

$$\forall \vec{I}.proj_P(safetraces(Q(\vec{I}))) = safetraces(P(\vec{I}))$$

The role of this preparatory step is to introduce meta-data (a.k.a. "book keeping") required to implement a choose expression; an initial version of the program may not have included all the meta-data it eventually needs.

T2.* assert *introduction If Q is obtained from P by adding an assert statement, where the (boolean) variable being asserted is one that the program already defines, we have the property that:

$$\forall \vec{I}.safetraces(Q(\vec{I})) \subseteq safetraces(P(\vec{I}))$$

This is true because the additional assert can only convert a trace that was previously safe to unsafe.

T3.* choose *determinization If P contains choose$v$ followed immediately by assert $(v = w)$ for some w, then we can replace this pair of statements by assign v w to obtain program Q. If traces of P are projected to choose expressions that are common to P and Q (using $proj_Q$) then:

$$\forall \vec{I}.safetraces(Q(\vec{I})) = proj_Q(safetraces(P(\vec{I})))$$

This is true because the angel for choose v in P could have produced whatever value Q generates in variable w.

Example 1. Suppose we are developing a program to reverse a linked list, for which we hypothesize the following angelic program works. The correctness postconditions (assertions) are omitted for clarity.

```
while (choose) {
  x = choose;
  y = choose;
  x.next = y;
}
```

Next, we might realize that we need a cursor variable cur. We use T1 to introduce it.

```
cur = in
while (choose) {
  x = choose;
  y = choose;
  cur = cur.next;
  x.next = y;
}
```

Now we add an assert using T2 to constrain one of the angelic choices to use this cur variable.

```
cur = in
while (choose) {
  x = choose;
  assert x == cur;
  y = choose;
  cur = cur.next;
  x.next = y;
}
```

We can then use T3 to determinize and remove this choose expression.

```
cur = in
while (choose) {
  x = cur
  y = choose;
  cur = cur.next;
  x.next = y;
}
```

4.2 Trace-based Refinement

Formally, we say P is *refined* by Q, denoted $P \preceq Q$, if

1. Q is derived from P using zero or more of the above transformations, and,
2. Q is correct, i.e. $\forall \vec{I}.(safetraces(Q(\vec{I})) \neq \{\})$.

The refinement relation $\preceq$ is reflexive and transitive.

Note that assert-introduction may cause the second condition to be violated. In this work, our intention is to use bounded-model-checking to establish the second condition.

The goal of refinement is to reduce the nondeterminism in a program. While state inflation of P into Q can introduce into Q new choose expressions, the non-determinism in Q with respect to P cannot increase with refinement because we do not reduce constraints on the choose expressions already present in P. This is because constraints on nondeterminism are relaxed only when statements are removed (in T3) and we remove statements only when their effect is already captured by an assertion.

Program development can be seen as creating a series of programs, $P_0, \ldots, P_n$, such that $P_i \preceq P_{i+1}, 0 \leq i < n$, where P_0 is the first angelic program a programmer creates, and P_n is free of any choose statements, i.e. a suitable deterministic program that takes the intended computation steps.

Existence of such a sequence does not mean that a programmer will make only correct refinements. In practice, a programmer will occasionally need to revert to a previous version and try some other transformation.

Adhering to the trace refinement methodology brings valuable software engineering benefits to the practical programmer; we explain these in the next section.

4.3 Program Restructuring

Sometimes a programmer might wish to refine a partial program in a way that is not expressible in terms of transformations T1–T3. Suppose s is a statement in P. In program Q, we wish to replace s with a statement s'. We assume that s is choose-free but s' is permitted to introduce new choose expressions.[3] Program Q will be a refinement of P if Q is correct and s' does not afford the choose operators in Q more freedom than they had in P. The second condition corresponds to ensuring that Q does not allow new traces:

$$\forall \vec{I}.proj_P(safetraces(Q(\vec{I}))) \subseteq safetraces(P(\vec{I}))$$

To ensure this property, we enforce, by construction, two properties:

1. The input values of s and s' are the same in all safe executions. Assume that s' has the same free variables as s. This property ensures that any choose expressions executed prior to s are constrained in Q at least as much as they were in P (the constraints are the union of constraints imposed by s and s', respectively).
2. The output values of s and s' are the same in all safe executions. Assume that s and s' update identical "output" variables. This ensures that s and s' end up in the same state, ensuring that yet to be executed choose expressions in Q will be constrained at least as much as they were in P (the constraints are the union of constraints of s and s', respectively).

Note that s' is free to contain any implementation; we do not impose any constraints on the values computed at intermediate points in s'.

The next two transformations are intended to allow safe restructuring.

T4. Coupling The goal of the construction is to assert equality on input values to s and s', as well as on their output values. This is done by recording these values in P and asserting their equality in Q. We call this transformation *coupling* of programs P and Q. Assume we can establish communication channels between P and Q. Each channel supports a $snd(v)$ operation to send a value and a $rcvandassert(v)$ operation that receives a value and asserts it to be equal to the value given as its argument. Just before executing s, P communicates all its "input" values (to s) to Q using a series of snd operations. Q receives them using $rcvandassert$ operations just before entering s', supplying *its* input values in the same sequence. Likewise, just after executing s, P communicates its "output" values over to Q, which compares them against its own output values.

T5. Uncoupling When the refinement process arrives at a choose-free program Q, the channels attached to Q are disconnected and the ancestor programs on the other end of the channels are discarded. Because Q is choose-free, the constraints imposed by the channels are not needed as they are already implicit in Q.

Henceforth, we consider this construction as part of our refinement toolset.

5. Programming with Angels

We assume that the programmer starts with a safe angelic program. Note that he may start with a relaxed correctness condition, strengthening it during the refinement process.

5.1 Methodology

As mentioned before, a programmer applies a succession of trace refinement steps until a program does not contain any choose expressions. Tool support helps in two ways:

1. *Testing hypotheses.* When the programmer carries out a transformation (T2 or T4) that adds constraints on the nondeterministic choices, he checks whether the resulting program maintains safety. If the program is not safe, the implementation strategy encoded in the angelic program is infeasible. In our implementation, this validation is performed with a bounded model checker.
2. *Trace demonstration.* The tool outputs the safe traces, which the programmer can examine to learn about the steps taken by the angel. In our case studies, these traces provided important clues as to what further refinement steps to take.

We remind the reader that execution of the angelic program is carried out on a small suite of test cases, which means that we may fail to ascertain that an angelic program (or the final program) is unsafe. As a result, it is possible that the programmer makes incorrect refinement decisions. In this sense, we are providing no more guarantees than the classical (test-driven) program development on which we wish to improve. However, as corroborated by work on using bounded model checking for software validation [8], a small test suite often reveals many software errors.

Example 2. This example illustrates that the programmer can discover that his implementation strategy is infeasible without necessarily having to obtain an incorrect angelic program. Consider Example 1, where we partially refined an angelic program for linked list reversal. If our initial refinement step in Example 1 had been slightly different, we would have produced following angelic program:

[3] Note that occurrences of choose in the statement s can be removed by lifting all instances of choose into a special argument of s, a list angels, whose elements are read by s.

```
cur = in
while (choose) {
  x = cur
  y = choose;
  x.next = y;
  cur = cur.next;
}
```

This program differs from the program produced in Example 1 in that the last two statements inside the loop have been switched. It is easy to write a program like this one (we did), considering that programmers are used to incrementing the loop induction variable at the very end of the loop.

This angelic program encodes an infeasible strategy. Specifically, the program contains a bug in that it overwrites cur.next before the old value of cur.next is read and used to obtain the next node. As a result, the angelic program is prevented from walking forward down the list. The angelic program is not incorrect, though, because some safe traces remain. For instance, in the first loop iteration, the program can "travel" to the last node and then walk the list backwards. The programmer can notice that no safe traces correspond to a desired algorithm by observing that all traces require at least $n + 1$ steps for a list with n elements. If desired, this observation can be confirmed by limiting the angelic program to n loop iterations, at which point no safe traces remain. The programmer then reverts to an earlier version of the program and corrects the implementation of how the induction variable cur is advanced through the list.

5.2 Removing angelic dependences

Informally, we say a choose expression c_1 is *independent* from a choose expression c_2 if c_1 can make its choices without considering what choice has been (or will be) made by c_2 in a given execution. Angelic programs with only independent choose expressions may lead to simpler algorithms because there is no need to deterministically implement the "communication" that happens between dependent choose expressions. We leave the full characterization of angelic dependence for future work. Here, we give only a sufficient condition for independence and discuss one of its benefits.

A sufficient condition for independence of all choose expressions in a program is that the program has only one safe trace per input. If a choose expression has the choice of only one value at any time it is evaluated, it need not consider the choices made by other angels.

When angelic choices are independent we get the benefit that choose expressions can be refined independently of each other. That is, the implementation need not consider how the other choices are determinized, as the next example illustrates.

Example 3. In Example 2, by examining traces returned by the oracle, the programmer recognized that it might be useful to enforce that the loop that reverses the elements in a linked list executes n times for a list with n elements. He might enforce this by writing the following program (with the bug from Example 2 fixed).

```
cur = in
while (n) (choose) {
  x = cur
  y = choose;
  cur = cur.next;
  x.next = y;
}
```

Here, n is the number of elements in the list and while (n) is syntactic sugar for a loop that performs no more than n iterations.

This program has exactly one trace (to see this, note that each of the n nodes in the list must have its next field overwritten, and by design the loop iterates at most n times, so to generate a safe trace the oracle must set each node's next field to the correct final value). With this, we are guaranteed to have removed the angelic dependence mentioned above. The programmer can thus consider implementing the two remaining angelic choices separately.

The loop condition is relatively easy: we know that we must ensure that cur is not null, and by examining the traces returned by the oracle we can see that this is indeed the correct condition. We can thus use T2 and T3 to generate the following program.

```
cur = in
while (n) (cur != null) {
  x = cur;
  y = choose;
  cur = cur.next;
  x.next = y;
}
```

For the remaining angelic choice, by examining the safe trace we notice that y is always chosen as the element before cur in the original list. With this insight, we can recognize that we must use T1 to introduce a new variable and have it walk down the list (as well as remove the now-unnecessary bounded loop), T2 to assert that y is chosen to be equal to it, and then T3 to generate the following fully-deterministic correct program.

```
cur = in
prev = null
while (cur != null) {
  x = cur;
  y = prev;
  cur = cur.next;
  x.next = y;
  prev = x;
}
```

5.3 Example of Program Restructuring

The next example illustrates transformations T4 and T5. Recall that, in contrast to T1–T3, the transformation T4 does not implement an instance of choose, but instead substitutes some deterministic code in program P with another (potentially angelic) code. The constraints on nondeterminism present in P are imposed on Q using communication channels between the two programs. T5 is a cleanup transformation.

Example 4. We partially develop an implementation of the ZipReverse problem assigned by Olivier Danvy at a summer school. Given two lists x and y, the problem is to compute $zip(x, reverse(y))$. For example, given x = [1, 2, 3, 4] and y = [a, b, c, d], the program outputs [(1,d), (2,c), (3,b), (4,a)]. The requirements are: neither list can be traversed more than once; the lists cannot be copied; the length of the lists is not known a priori; and lists must be manipulated with the low-level operations car, cdr and cons.

The first angelic program tests a hypothesis that the result can be computed with a sequence of cons operations. When x is a list, choose(x) selects an element from the list.

```
r = nil
while (choose) {
    r = cons((choose(x), choose(y)), r)
}
```

The hypothesis tested by the program is admittedly trivial but the program is nonetheless a useful start for refinement. This is because it has only one safe trace for each input. Considering that a list can only be created with cons, this trace shows the steps that final implementation will have to take.

The programmer then attempts to refine this program by implementing choose(x) and choose(y) with code that traverses the two lists by maintaining pointers that advance down the lists. The lists are traversed according to the restrictions of the problem statement but nondeterminism allows all legal traversals. This attempt at refinement, not shown, has no safe trace, so the programmer concludes that a recursive procedure needs to be used in place of the iterative one. This recursive angelic program, shown below, can be refined into a deterministic program; thanks to recursion, the list y can be traversed in the opposite direction, which was not possible in the iterative version.

```
r = nil
descent()

def descent() {
    if (choose) return
    descent()
    r = cons((choose(x), choose(y)), r)
}
```

How is this program a refinement of the first program? The programmer can make it so by invoking transformation T4. The program that couples the two programs can be written as follows:

```
r = nil                          ||   r = nil
ch1 = choose                          descent()
snd(ch1)
while (ch1) {                         def descent() {
  ch2 = choose(x)                       ch1 = choose
  snd(ch2)                              rcvandassert(ch1)
  ch3 = choose(y)                       if (ch1) return
  snd(ch3)                              descent()
  r = cons((ch2, ch3), r)               ch2 = choose(x)
  ch1 = choose                          rcvandassert(ch2)
  snd(ch1)                              ch3 = choose(y)
}                                       rcvandassert(ch3)
ch4 = pack(r)                           r = cons((ch2, ch3), r)
snd(ch4)                              }
                                      ch4 = pack(r)
                                      rcvandassert(ch4)
```

The program on the left is the first program, with inserted channel send operations; on the right is the second program, with inserted channel receives. The output of the statement that we have replaced with T4 is the list r, which is packed with pack(r) before it is sent across a channel. (The output of the second program is considered to be the output of the entire program.)

Superficially, we violated the conditions of T4: the program on the the left contains choose statements while T4 requires that the replaced statement be choose-free. However, conceptually we could have collected angelic values in advance in an array angels and read from it.

Once the program on the right is developed into deterministic code (not shown for lack of space), we can use T5 to disconnect the program on the left and remove the channels.

6. Comparison to Deductive Refinement

We briefly review Morgan's [11] refinement methodology, and then contrast it with ours. We note at the outset that Morgan's methodology is geared towards developing a proof of a program as it is being developed, whereas our intention is to help the programmer develop a program that is correct within the confines of bounded model checking.

In Morgan's methodology, one develops programs using refinement of specification statements. A specification statement is of the form $\vec{v} : [pre, post]$, and it can be used any place an ordinary statement may be used. The meaning of this statement is that, if executed in a state in which pre holds, it modifies state variables in $\vec{v}$ such that $post$ holds. The specification statement is a *demonic* non-deterministic statement: *all* values that satisfy $post$ must suffice. The wp-semantics of a specification statement are given as:

$$wp(\vec{v}[pre, post], R) = pre \wedge (\forall \vec{v}.post \implies R)$$

Refinement of a specification statement is either a deterministic statement or another specification statement that could be used in place of the original one. For P to be correctly refined by Q, denoted $P \leq Q$,

$$\forall R, wp(P, R) \implies wp(Q, R)$$

where wp is the weakest precondition operator and R is a postcondition.

Semantically, our definition of trace-based refinement ($\preceq$) is a special case of the refinement defined above ($\leq$).

In Morgan's methodology, a programmer makes a sequence of refinement steps until the program is completely free of nondeterministic statements. Rather than prove correctness of refinement steps from the principle above, the programmer can use a set of proven refinement *laws*, some of which we show here:

- It is always legal to strengthen the postcondition or weaken the precondition.
- One of the laws for introduction of a sequential composition requires a suitable intermediate condition (*mid* below): $\vec{v} : [pre, post] \longrightarrow \vec{v} : [pre, mid]; \vec{v} : [mid, post]$. The *mid* should be suitable in that further refinement of the two new specification statements is feasible.
- The law for introduction of a loop requires a specification to be brought in of the form: $[pre \wedge I, post \wedge I]$, where I is a suitable loop invariant. (We omit the actual transformation; see [11].)

An attractive property of Morgan's refinement is that occurrences of specification statements in a program can be refined *independently* of each other (Theorem 2, pg 8, in Morgan and Vickers [12]). This makes the approach compositional.

However, from the perspective of a practical programmer, this is also the main problem in using Morgan's methodology: individual specification statements must have sufficiently powerful postconditions and correct invariants for the programmer to make progress. If the programmer does not figure out the correct loop invariant, the methodology might get stuck later during development, and even the correct invariant must often be modified in later steps. This is not surprising: a programmer is developing a rigorous proof simultaneously with the program.

Angelic specification statements have also been suggested in the literature (e.g. Celiku and Wright [4], Ward and Hayes [20]). Ward and Hayes used angelic refinement to prove correctness of backtracking algorithms. Let us denote an angelic specification

statement as $\vec{v} : \{pre, post\}$, meaning that the $\vec{v}$ must be assigned in such a way that $post$ holds, and moreover, the values are chosen cooperatively to make the rest of the program run correctly. Formally,

$$wp(\vec{v}\{pre, post\}, R) = pre \wedge (\exists \vec{v}.post \wedge R)$$

At first glance, this seems to free the programmer from the necessity of providing suitable *post*, because the angel would always select a value that suits the rest of the computation (in addition to satisfying whatever postcondition is provided manifestly in the specification statement.). However, *local* refinement of angelic specifications, i.e. a refinement that can be applied oblivious to the rest of the program, is only allowed to *increase* choices available to the angel. This does not help the programmer to get to a deterministic program. (As stated above, their purpose was to prove correctness of backtracking algorithms.)

Celiku and Wright propose strengthening the postconditions of angelic specifications in such a way that they essentially become demonic specifications, which can then be refined using Morgan-like methodology. The process of converting angelic to demonic specification requires manual, non-local, whole program reasoning. More formally, the following conversion can be carried out for a strong enough ϕ:

$$\vec{v}\{pre, post\}; assert\ \phi \longrightarrow \vec{v}[pre, post \wedge\ \phi]$$

The problem for the programmer is to determine the ϕ that would capture the "expectation" of the rest of the program.

Our methodology is intended as a practical tool for program development, not as a proof procedure. In our methodology, we do not have compositionality: by default, angelic choose expressions cannot be refined obliviously to each other. (Removing angelic correlation, as we discussed in Section 5.2, is similar to converting angelic to demonic nondeterminism.)

7. Implementation

We have embedded the angelic choice construct into the Scala programming language [13]. The programmer passes to the angelic choice operator a list of values from which a parallel backtracking solver selects one that leads to a safe trace. The solver computes all safe traces, which can then be browsed with a user interface. The angelic choice operator ranges over arbitrary primitives or references to heap-allocated variables and can be used on arbitrary programs. We have used our implementation to develop the examples in this paper as well as several others. The rest of this section discusses our backtracking solver, as well as a more scalable SAT solver. We conclude by comparing their scalability.

7.1 Parallel Backtracking Solver

Our backtracking solver performs a depth-first traversal over the space of traces, searching for safe traces. The solver executes a program and whenever it encounters a choice operator, it pushes a new entry corresponding to the dynamic instance of this operator to a stack. It then tries the first value for the entry on the top of the stack by executing the program further. On an assertion failure, the solver backtracks to the execution point associated with the top of the stack and tries the next value. If there are no values left for this entry, it is popped from the stack and the execution backtracks to the execution point corresponding to the previous entry on the stack, continuing the process. A safe trace is found when the execution terminates without failing an assertion.

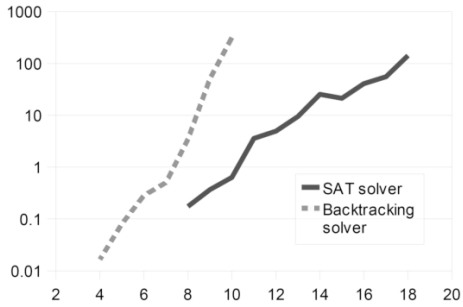

Figure 1. Time taken (in seconds) by the SAT solver and the backtracking solver when inserting a number of nodes into an initially empty binary tree.

Our backtracking solver explores the trace space in parallel. Each task receives from the dispatcher a prefix of a trace, including the corresponding stack that captures the state of the angelic operators executed on that prefix. The stack captures which values have already been tried for these operators, and it is thus an efficient way of representing the part of the space that has already been explored. Each parallel task is responsible for exploring all suffixes of the prefix. Except for communicating with the dispatcher, the tasks are independent, so the parallel computation is very efficient. Our solver is efficient in its memory use so most of its time is spent executing the program.

Backtracking is not a good fit for problems that contain assertions mostly at the end of the execution because the search takes a long time to encounter a failure. SAT solvers, on the other hand, propagate constraints and can generate new constraints (conflict clauses) as they explore the search space. The backtracking solver's scalability on these problems can be improved by adding assertions with intermediate invariants, which can drastically prune the search space.

7.2 SAT-based Angelic Solver

Our second solver is based on symbolically unrolling the program for given inputs and representing the bounded execution as a SAT formula. The satisfying assignment then gives the values of the angelic operators on these inputs. We use the program translator developed as part of the SKETCH synthesizer project [18], which achieves scalability by representing integers in sparse unary encoding, thus optimizing the SAT formula for the small values that one is likely to see during program development based on unit testing. This solver is not yet connected to our Scala-based frontend but angelic programs for that solver can be written in the SKETCH language by using simple macros.

7.3 Solver Efficiency Study

All experiments presented in this paper were performed on the backtracking solver, whose scalability was sufficient (its response was usually interactive). It is interesting to evaluate whether the SAT-based angelic solver allows us to scale to harder problems. While individual problems vary widely, some experimental performance numbers may be helpful to compare the scalability of angelic solver techniques. In Figure 1, the time of the backtracking and SAT solvers is shown on an angelic program that inserts nodes into a binary search tree. The angel chooses any node in the tree and then chooses whether to insert the new node below the left child or the right child. Assuming that we only insert into nodes already in the tree, this is a search space of $2^{n-1}n!$, where n is the number of nodes

to insert. There is only one correct solution in this space. The backtracking solver is usable until the search space is about 10^9, while the SAT-based angelic solver scales to $8 * 10^{20}$, or more than ten orders of magnitude higher. This gives us hope that angelic refinement based on executable agents may be usable on a range of realistic programs.

8. Case Study: Modularizing DSW

This section describes a case study where angelic programming aided in modularizing the Deutsch-Schorr-Waite (DSW) graph marking algorithm [15]. This algorithm has long been considered one of the most challenging pointer algorithms to reason about [2]. We modularize DSW with a new generic abstraction, called the *parasitic stack*, which is parameterized for DSW with angelic operators. Our case study shows that (*i*) modularization of DSW simplifies reasoning about the algorithm because the parasitic stack separates concerns intertwined in DSW; and (*ii*) choose operators allow the algorithm designer to sidestep global reasoning. We conjecture that modularization by angelically parameterizing an abstraction may be applicable more widely to other complex algorithms.

The DSW algorithm solves the problem of marking nodes reachable from the root of a directed graph. The crucial requirement is to do so with only constant-size additional storage. (One exception is that a node can store an index into its list of children.) This requirement is motivated by garbage collection where object tracing is invoked when the system cannot offer more than constant-size memory. If linear-size additional memory were available, one could perform a DFS traversal of the graph, using a stack for backtracking. The DSW trick is to guide the DFS traversal by temporarily "rewiring" child pointers in nodes being visited. The price for the space efficiency is intertwining of the two parts of the DFS traversal:

- *Iterative graph traversal.* Because recursive DFS traversal is ruled out, DSW relies on the more involved iterative traversal.
- *Backtracking structure.* Because an explicit stack is ruled out, DSW encodes a backtracking structure in the child pointers of graph nodes.

Ideally, these aspects should be separated, for example by hiding the backtracking structure under an abstraction. However, as shown in Figure 2, this separation seems difficult. We do not ask the reader to understand the algorithm in Figure 2; we merely want to point out the essence of what complicates modularization. In particular, note that the memory location current.children[current.idx] serves two roles: on the right-hand side of the first parallel assignment, the location stores an edge of the graph; on the left-hand side, it serves as storage for the backtracking structure. Because the role of the location changes in the middle of an assignment, it is not obvious how to hide one role under a procedural abstraction.

The original goal of our case study was not to modularize DSW; we started by asking simply whether angelic programming could aid in discovering and implementing the classical (flat) DSW algorithm. We explained the trick in DSW to a few students and then gave them access to the Scala language extended with choose operators. The insight was explained by telling them to "use the child fields in the graph nodes to encode the backtracking stack needed in DFS traversal." As students explored algorithmic ideas, it became clear that it was difficult to design DSW even with angelic nondeterminism. The reason was that the programmer found it hard to explain how the angels manipulated the individual pointer fields. It became apparent that it was necessary to raise the level of the programming abstraction so that one could observe the angelic decisions at a more meaningful semantic level. Unfortunately, as we pointed out, DSW does not seem to permit such modularization.

```
def DSW(g) {
  val vroot = new Node(g.root)
  var up, current = vroot, g.root

  while (current != vroot) {
    if (!current.visited) current.visited = true
    if (current has unvisited children) {
      current.idx = index of first unvisited child
      // the child slot changes roles in this assignment
      up, current, current.children[current.idx] =
          current, current.children[current.idx], up
    } else {
      // the child slot restores its role
      up, current, up.children[up.idx] =
          up.children[up.idx], up, current
    }
  }
}
```

Figure 2. The classical (flat) DSW algorithm.

```
def DSW(g) {
  val vroot = new Node(g.root)
  var current = g.root
  ParasiticStack.push(vroot, List(vroot,g.root))

  while (current != vroot) {
    if (!current.visited) current.visited = true
    if (current has unvisited children) {
      current.idx = index of first unvisited child
      val child = current.children[current.idx]
      ParasiticStack.push(current, List(current, child))
      current = child
    } else {
      current = ParasiticStack.pop(List(current))
    }
  }
}
```

Figure 3. The parasitic (modular) DSW algorithm.

It occurred to us that it might be possible to express DSW as using a special stack-like data structure. Like a regular stack, this data structure would support the push and pop operations with the usual semantics. Unlike a regular stack, this data structure would be implemented by borrowing memory locations from its host — in our case, the graph being traversed. The hope was that the metaphor of borrowing a location would allow us to express the transition between roles in a systematic fashion. We termed this data structure a *parasitic stack* because it borrows locations from its hosts and eventually restores them (parasites do not destroy their host). The challenge was how to express DSW with a parasitic stack, if that was at all possible.

The parasitic stack has the interface shown below. As usual, the pop operation returns the value x stored in the corresponding push operation. The nodes argument to push passes into the stack the environment of the parasitic stack's client. Through this argument, the traversal code "loans" the nodes referenced by in-scope variables to the stack. The environment is also passed into pop, where the values may be useful for restoring the values of

```
ParasiticStack {
    e = new Location // constant−size storage (one location suffices to support DSW)

    push(x,nodes) { // 'nodes' is the set of nodes offered "on loan" by the host data structure
        n = choose(nodes) // angelically select which node to borrow from the host ...
        c = choose(n.children.length) // ... and which child slot in that node to use as the location
        n.idx2 = c // remember the index of the borrowed child slot
        v = n.children[n.idx2] // read the value in the borrowed location; it may be needed for restoring the borrowed slot in pop()
        // angelically select values to store in the two locations available to us ('e' and the borrowed location)
        e, n.children[n.idx2] = angelicallySemiPermute(x, n, v, e)
    }
    pop(nodes) {
        n = choose(nodes, e) // the borrowed location better be in either 'e' or in 'nodes'
        v = n.children[n.idx2] // 'v' is the value we stored in the borrowed location
        // select what value to return and update the two locations we work with (the borrowed child slot and 'e')
        r, n.children[n.idx2], e = angelicallySemiPermute(n, v, e, *nodes) // '*nodes' unpacks arguments from list 'nodes'
        return r
    }
}
```

Figure 4. An angelic implementation of the parasitic stack, $ParasiticStack_0$.

borrowed memory locations when the stack returns them to the host.

```
// parasitic stack can borrow a field in some node from 'nodes'
push(x:Node, nodes:List[Node])
// values in 'nodes' may be useful in returning storage to host
pop(nodes:List[Node]) : Node
```

With the parasitic stack in hand, DSW can be expressed as shown in Figure 3; it can be derived almost mechanically from a recursive DFS traversal procedure. If one ignores the additional arguments to push and pop, this *parasitic DSW* appears to use a regular stack that has private storage. The parasitic DSW is thus modular in that it abstracts away the details of how the parasitic stack is implemented.

Note that at this point of our user study, we did not yet know whether DSW was expressible on top of such a stack interface. The challenge behind answering this question was to determine

- which location the parasitic stack can borrow from the host—it must be a location the host does not need until the location is returned to the host by the parasitic stack;
- how to restore the value in this location when returning it to the host; and
- how to use this location to implement the push/pop interface.

These three questions are formulated as nondeterministic choices in the implementation of the parasitic stack. We were not able to answer these questions without angelic help.

The reader might wonder if we simply substituted one hard problem, namely implementing DSW using low-level pointer manipulations, with another equally hard one, namely implementing the parasitic stack. What we achieved is that an angelic formulation of the three questions (for parasitic stack) is relatively straightforward. Furthermore, we found that the angelic answers to these questions can be interpreted by the programmer more easily than when the angels implement the low-level DSW pointer manipulations because the questions raise the level of abstraction.

The three questions are encoded in the angelic implementation of the parasitic stack, shown in Figure 4. As we discuss this code, it will be obvious that the three questions, answered by angels, capture the global reasoning necessary to make the parasitic stack operate correctly. As a result, the programmer is freed to constrain himself to local reasoning, whose goal is to describe how a *generic* parasitic stack might operate. Genericity is achieved with angelic nondeterminism. The next two paragraphs describe the angelic parasitic stack. We then use refinement to parameterize this angelic parasitic stack for DSW.

The parasitic stack in Figure 4 keeps only a single memory location (e). The push method first angelically selects which memory location to borrow from the host. This is done by selecting a suitable node n and a child slot c in that node. The borrowed location is n.children[c]. The stack (deterministically) stores the selected child slot index in the node itself, as that is allowed by the constraints of the DSW problem. Next, push reads the value in the borrowed location since it will need to be restored later and so may need to be saved. Finally, there is a hard decision. The stack has four values that it may need to remember: the pushed value x, the reference to the borrowed location n, the value in that location v, and the value in the extra location e. However, there are only two locations available to the stack: the borrowed location n and the extra location e. Clearly, only two of the four values can be stored. Perhaps the value of e is not needed, but the remaining three values are essential.

A little reasoning reveals that the parasitic stack is plausible only if the value that push must throw away is available at the time of pop from the variables of the enclosing traversal code. Therefore, we decided to make the environment of the client available to pop. The pop method first guesses which location was borrowed in the corresponding push. This location is either n or is in nodes; no other alternatives exist. Next, pop reads the value from the borrowed location. Finally, pop angelically decides (i) which value to return, (ii) how to update the extra locations, and (iii) how to restore the borrowed location. As in the case of push, it must select from among four values.

The benefits of clairvoyance should now be clear: while the human found it hard to make the global decisions necessary to instantiate the parasitic stack for DSW, these decisions were rather straightforward to formulate as nondeterministic choices.

Next, we instantiated the parasitic stack for DSW by refining it into a deterministic program. To arrive at the first refinement, we observed how the angels permuted the values. We identified a trace in which the angels performed the same permutation throughout the entire execution, except in the first call to

push. This first call was an outlier because our parasitic DSW algorithm (Figure 3) originally invoked the first push incorrectly, with the reversed environment, as follows:

ParasiticStack.push(current, List(g.root,vroot)).

After we modified the parasitic DSW, we were able to implement the permutations with deterministic code, shown below. We noticed that the value v of the borrowed location was not saved by the angel and that it was later obtained from nodes[0] in pop. This answered the question of how to restore the value in the borrowed location.

ParasiticStack$_1$ – **refines** *ParasiticStack*$_0$

```
ParasiticStack {
    e = new Location

    push(x,nodes) {
        n = choose(nodes)
        c = choose(n.children.length)
        n.idx2 = c
        // rhs was 'angelicallySemiPermute(x, v, e, n)'
        e, n.children[n.idx2] = x, e
    }
    pop(nodes) {
        n = e // rhs was 'choose(nodes, e)'
        v = n.children[n.idx2]
        // rhs was 'angelicallySemiPermute(n, v, e, *nodes)'
        r, n.children[n.idx2], e = e, nodes[0], v
        return r
    }
}
```

To arrive at the second refinement, we observed how the angels selected the borrowed node (the value n in push). We tested if the choice was consistent across all instances of push (it was), and then we implemented the angelic choice.

ParasiticStack$_2$ – **refines** *ParasiticStack*$_1$

```
ParasiticStack {
    e = new Location

    push(x,nodes) {
        n = nodes[0] // rhs was 'choose(nodes)'
        c = choose(n.children.length)
        n.idx2 = c
        e, n.children[n.idx2] = x, e
    }
    pop(nodes) { ... unchanged ... }
}
```

To perform the last refinement step, we examined how the remaining angel selected the child slot to borrow from the selected node n. We learned that it selected a slot whose value equaled nodes[1], the second variable passed to push from the traversal code. This implied that c equaled the value of n.idx maintained in the traversal code. Therefore, maintaining a separate field n.idx2 was not necessary. We turned this observation into the deterministic code shown below. This completed the construction of parasitic DSW.

ParasiticStack$_3$ – **refines** *ParasiticStack*$_2$

```
ParasiticStack {
    e = new Location

    push(x,nodes) {
        n = nodes[0]
        // invariant: n.children[c] == nodes[1],
        // hence c == n.idx == n.idx2
        // was 'c = choose(n.children.length); n.idx2 = c'
        e, n.children[n.idx] = x, e
    }
    pop(nodes) {
        n = e
        v = n.children[n.idx] // was n.idx2
        r, n.children[n.idx], e = e, nodes[0], v
        return r
    }
}
```

Angelic nondeterminism has simplified implementation of the DSW algorithm. The parasitic DSW still takes some effort to understand but the comprehension task has been broken down into three smaller questions: how the stack is implemented, which location the stack is borrowing, and how its value is restored. It would be interesting to consider whether this modularization of DSW leads to a simpler deductive proof of correctness.

9. Related Work

Floyd was one of the first to propose using angelic nondeterminism as a programming construct [7]. The concept appears also in formal languages, e.g., in nondeterministic automata [14]. Angelic non-determinism also allowed abstracting specifications by non-deterministically coordinating concurrent components [16]. In the rest of this section, we focus on work related to systematic program development.

J-R. Abrial has presented a very successful methodology for construction of event-based systems [1] and showed that the same methodology works for pointer manipulating programs [2]. Abrial's methodology starts with a declarative specification of the desired program and gives a set of laws by which specifications can be translated to lower-level specifications until they reach the level of basic program statements orchestrated by the usual control-flow constructs. At each step, his methodology asks a programmer to discharge a number of proof obligations, some of which can be automated. However, the methodology is intended primarily for programmers who are well-versed in the use of automated theorem provers.

His derivation of DSW does not modularize the constituent concepts of the *backtracking structure*, the *graph*, and the *traversal order* [2]. He defines the backtracking structure as closely tied with the traversal order: the structure is a list of nodes currently being visited. This characterization is then refined by implementing the list as a rewiring of the graph. Therefore, the content of the structure remains intimately linked with the graph structure.

Angelic nondeterminism has been used in refinement-based program development methodology proposed by Back, Wright, Morgan, and others. Back and von Wright [3] simplified problem decomposition in program refinement [11], using angelic nondeterminism to satisfy intermediate conditions that the programmer would be able to spell out only later in the development process.

In contrast to deductive angelic refinement, our angelic programs are executable, as in Floyd [7]. This allows a test of the correctness of angelic programs. If the test fails, we have conclusively proved that the angelic program is incorrect and cannot

be successfully refined. A passing test is not conclusive (we rely on bounded model checking [5]), but the executable angel shows us a demonstration of how to execute the program. Safe angelic traces thus serve as demonstrations of what steps the angel takes to execute the program, which has the benefit of increasing program understanding.

Celiku and Wright [4] show how to refine angelic nondeterministic statements to demonic ones, with the goal of being able to refine them independently. In order to achieve this goal, they prove, manually, sufficient postconditions that an angelic statement must satisfy. In general, however, angelic correctness is established with a proof that is obtained by effectively completing the refinement process all the way to the deterministic program. While such a deductive process provides a proof of the final program, it does not seem to enhance programmer productivity. Our approach changes the problem: we side-step the intermediate stage of obtaining an equivalent demonic nondeterministic program. Instead, we aim to refine to a deterministic program directly, which we achieve by making the choose operator executable.

There has been a long tradition of specification-based programming at Oxford University, and Morgan's work represents that school of thought. The Z programming methodology [19] is closely related to that of Morgan. A complete description of related work in this tradition is well outside the scope of this paper, but [11] is a good reference.

The work presented in this paper can be viewed as an evolution of the SKETCH project [18, 17]. In SKETCH, a programmer leaves syntactic "holes" in a program, which can later be filled automatically by a family of expressions. SKETCH has been shown to work very well in several application domains, in particular bit-manipulating programs, in which it is easy to give a specification of an unoptimized program but difficult to develop a correct optimized program. One of the limitations of the SKETCH work is that it requires a substantial amount of work on the part of a programmer to write the incomplete program, since holes are substitutes for only a very limited family of expressions. This not only is work for the programmer, but it also creates the possibility of making human mistakes that can cause a SKETCH synthesizer to not be able to fill in the holes. The present work addresses both criticisms.

The work presented in this paper was in part inspired by the work of Lau et al., who synthesize editing macro programs from user demonstrations of executions (i.e., traces) of the desired macros [9]. We realized that such demonstrations are useful not only for synthesis but also for gaining understanding about an algorithm that is under construction. In fact, if an oracle gave the programmers a trace of the program that they are developing, they might find it easier to debug the program as well as develop it in the first place. Program development could then be viewed as generalizing the algorithm from demonstrations. Our next observation was that such oracular traces could be created by angelic programs, which led to the work presented in this paper.

10. Conclusion

We have demonstrated that an executable implementation of angelic nondeterminism may help in program development. First, the programmer can evaluate hypotheses about his implementation strategy by testing whether his incomplete program can be executed angelically. If angelic operators cannot complete the program with values that lead to a successful execution, neither will be the programmer, and so the program follows an infeasible implementation strategy. Second, if an incomplete program can be completed angelically, the successful angelic executions may reveal steps that the incomplete algorithm should follow. The traces computed by the angelic operators may thus lead the programmer to an a-ha moment. Third, angelic operators support refinement-based programming, where the programmer develops the program gradually, by replacing angelic operators with progressively more deterministic implementations.

In contrast to angelic nondeterminism that is purely a proof device, the ability to execute, test and observe angelic programs allows the programmer to refine programs and develop abstractions, such as the parasitic stack, that seem difficult for humans without the computational power of the oracle.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge Un. Press, Aug. 1996.

[2] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME*, pages 51–74, 2003.

[3] R.-J. Back and J. von Wright. Contracts, games, and refinement. *Inf. Comput*, 156(1-2):25–45, 2000.

[4] O. Celiku and J. von Wright. Implementing angelic nondeterminism. In *APSEC*, pages 176–185. IEEE Computer Society, 2003.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[7] R. W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, oct 1967.

[8] M. F. Frias, C. G. López Pombo, G. A. Baum, N. M. Aguirre, and T. S. E. Maibaum. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Methodol.*, 14(4):478–526, 2005.

[9] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, pages 36–43, 2003.

[10] C. L. McMaster. An analysis of algorithms for the dutch national flag problem. *Commun. ACM*, 21(10):842–846, 1978.

[11] C. Morgan. *Programming from Specifications*. 1998.

[12] C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, London, 1992.

[13] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[14] M. O. Rabin and D. S. Scott. Finite automata and their decision problems. *IBM J. Res. and Develop*, 3:114–125, 1959.

[15] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, Aug. 1967.

[16] G. Smith and J. Derrick. Abstract specification in object-Z and CSP. In C. George and H. Miao, editors, *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 108–119. Springer, Nov. 2002.

[17] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 167–178, New York, NY, USA, 2007. ACM.

[18] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Not.*, 41(11):404–415, 2006.

[19] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge Un. Press, New York, NY, USA, 1988.

[20] N. Ward and I. J. Hayes. Applications of angelic nondeterminism. In P. A. Bailes, editor, *Proc. 6th Australian Software Engineering Conference (ASWEC91)*, pages 391–404. Australian Computer Society, JUL 1991.