Applications of Angelic Nondeterminism

Nigel Ward Ian Hayes

Key Center for Software Technology
Department of Computer Science
The University of Queensland

Abstract

The refinement calculus [BaW89, MoR87, Mor87] is based on the addition of *specification constructs* to Dijkstra's guarded command language [Dij76]. This paper explores the semantics and uses of a specification construct which chooses values for its variables *angelically* rather than in the usual demonic fashion. Given a number of alternatives angelic choice always makes the "correct" choice if one exists.

1 Introduction

Problems in areas such as parsing, game playing and combinatorial searching are often solved using backtracking algorithms. In [Flo67] Floyd introduces "nondeterministic" language constructs which allow these algorithms to be expressed without reference to the implementation details required for backtracking. These constructs are

- **choose**(S) — A function which arbitrarily returns an element of the set S. During execution a call to this function is called a *choice point.*
- **fail** — Signals an unsuccessful computation.
- **succeed** — Signals a successful computation.

A call to **choose**(S) nondeterministically "guesses" an element of S which makes the program terminate successfully. For example, suppose we wish to place N queens on an N by N chess board such that no queen can take any of the others. If we model a solution as a set of ordered pairs ($row, column$) where each ordered pair represents the position of one queen on the chess board then an algorithm can be expressed in Floyd's language as follows:

$row := 1;$
do $row < N \longrightarrow$
  $col :=$ **choose**$(1..N)$; $soln := soln \cup \{(row, col)\}$; $row := row + 1$
od;
if $no\_capture(soln) \longrightarrow$ **succeed**
$[]$ $\neg$ $no\_capture(soln) \longrightarrow$ **fail**
fi ,

where *no_capture*(*soln*) is true exactly when no two queens in *soln* can capture each other.

This algorithm places the queens one at a time. Each placement is made so that it is possible to place the rest of the queens on the board. That is, each placement is made so that the **succeed** statement rather than the **fail** statement is executed in the final **if** statement.

Operationally, Floyd's language can be explained as follows. If a program encounters a **fail** statement then it backtracks to the last choice point with untried alternatives, say **choose**(T), chooses an element of T which has not already been tried and continues from this point. If there are no more choice points with untried alternatives then the program aborts. Execution of a **succeed** statement causes the program to terminate. Support for this type of nondeterministic programming has been added to procedural languages such as Pascal [Lin79] and C [LiS90] and is fundamental to logic programming languages such as Prolog.

This paper gives an axiomatic semantics for a "nondeterministic" programming language similar to Floyd's by using *angelic nondeterminism*. Given a number of alternatives angelic nondeterminism always makes the "correct" choice if one exists. We examine the semantics of our language and rules for transforming specifications into programs written in it within the context of the refinement calculus [BaW89, MoR87, Mor87]. The calculus is based on the addition of specification constructs to Dijkstra's guarded command language [Dij76]. The semantics of these constructs are captured formally by considering statements to be monotonic predicate transformers from postconditions to weakest preconditions. The notion of refinement between specifications and implementations is also formalised in terms of weakest preconditions.

Section 2 presents an overview of the demonic and angelic specification constructs which extend Dijkstra's language. Section 3 forms the major part of the paper, giving an example of some uses of angelic nondeterminism in a refinement of a general list problem. The next two sections examine modifications to the program developed in Section 3. Language constructs similar to Floyd's are added to the program in Section 4 while Section 5 shows how the program can be made more efficient. The results of Sections 3, 4 and 5 are then applied to a specific list problem — the N-queens problem — in Section 6. It is assumed that the reader is familiar with one of the flavours of the refinement calculus referenced above, although emphasis is given to Morgan's style of calculus.

2 Specification Constructs

In this section we give an overview of the demonic and angelic specification constructs which extend Dijkstra's language. We capture the semantics of a construct by equating the construct with its weakest precondition to achieve a postcondition R, writing $S(R) = P$ for $wp(S, R) = P$.

The simplest specification construct is the *assumption*, written $\{pre\}$. It asserts that the predicate *pre* is true. If it is, then the construct does nothing, otherwise it aborts. Its definition in terms of weakest preconditions is

Definition 1 (Assumption) For any predicate R

$$\{pre\}(R) \mathrel{\widehat{=}} pre \wedge R \qquad \Box$$

A *demonic specification*, written $w : [post]$, nondeterministically chooses values for the variables w so that *post* is established. Its weakest precondition is

Definition 2 (Demonic Specification) For any predicate R

$$w : [post](R) \mathrel{\widehat{=}} (\forall w \bullet post \Rightarrow R) \qquad \Box$$

Demonic specifications choose values for their variables nondeterministically. That is, they do not determine unique values for their variables. As an example, consider the problem of taking a sequence of records, *in*, and sorting them by key value into a sequence, *out*. If the records are of type *REC*:

REC

$$key : \mathbb{N}$$

this problem can be specified as

$$out : [is\_perm(in, out) \wedge is\_ordered(out)] ,$$

where $is\_perm(in, out)$ is a predicate which is true precisely when *in* and *out* are permutations of each other and $is\_ordered(out)$ is true if and only if the sequence of records *out* is ordered by the $\leqslant$ relation on keys. As it stands this specification is deterministic — for any given input it always gives the same output, even if the input contains records with duplicate keys. In this case these records will appear consecutively in the output and their ordering with respect to each other is not determined. However, since we have no way of telling them apart the output sequence always appears the same. If we give records an extra field so that we can tell them apart, for example a data field:

REC

$$key : \mathbb{N}$$
$$data : Data$$

then the specification is nondeterministic. Records in the input with the same keys but different data fields still appear consecutively in the output in any order, but since we can tell which order they appear in, the specification does not determine a unique value for *out*. That is, the specification is nondeterministic.

We now move on to a definition for an *angelic specification*, written $w : \overline{[post]}$. Like its demonic counterpart this nondeterministically chooses values for w so that *post* holds. Its weakest precondition definition is, however, somewhat different.

Definition 3 (Angelic Specification) For any predicate R

$$w : \overline{[post]}(R) \mathrel{\widehat{=}} (\exists w \bullet post \wedge R) \qquad \Box$$

To explain the difference between demonic and angelic specifications we examine the following specification:

$$out : \overline{[is\_perm(in, out) \wedge is\_ordered(out)]} .$$

This is similar to the demonic specification for the sorting problem and, as before, if the records only contain a key field then the specification is deterministic. In this case the specification is equivalent to the demonic specification.

If records also contain a data field then the sequence *out* is assigned a value non-deterministically as follows: the angelic specification "looks ahead" to see how its choice for the value of *out* affects the execution of the rest of the program. It then assigns *out* a value so that the program terminates with a "correct" answer. For example, the specification

$$out : \overline{[is\_perm(in, out)]} \ \{is\_ordered(out)\}$$

makes an angelic choice for *out* so that it is a permutation of *in* and then asserts that *out* is sorted. When the angelic specification makes its choice for *out* as a permutation of *in* it "looks ahead" and sees that for the program to terminate *out* must be sorted also. That is, it chooses *out* so that both $is\_perm(in, out)$ and $is\_ordered(out)$ hold. Thus it is equivalent to the previous angelic specification:

$$out : \overline{[is\_perm(in, out) \wedge is\_ordered(out)]} .$$

To explain what is meant by "correct" answer above we use the interpretation of the weakest precondition of a program, *Prog*, with respect to a predicate R given in [Dij76]. That is, if *Prog* is executed in a state satisfying $Prog(R)$ then it is *guaranteed* to terminate in a state satisfying R.

Examining the definition of a demonic specification we see that $(\forall w \bullet post \Rightarrow R)$ characterises states such that for any assignment to w, if *post* holds then R also holds. That is, these are the states from which if we achieve *post* we are guaranteed to achieve R.

Interpreting the definition for angelic specifications, $(\exists w \bullet post \wedge R)$ characterises states in which there exists an assignment to w such that both *post* and R hold. This means that there must at least exist an assignment to w such that *post* holds. That is, these are the states from which it is *possible* for the corresponding demonic specification to achieve R. But if $w : \overline{[post]}$ is executed in such a state, then the interpretation of weakest preconditions given above tells us that we are guaranteed to achieve R. This is why $w : \overline{[post]}$ is said to be angelic: it is *guaranteed* to achieve R if executed in any state from which it is only *possible* for the corresponding demonic specification to achieve R. Angelic choice always makes the "correct" choice so that we are guaranteed to achieve the desired R (whenever this is possible).

Operationally, angelic nondeterminism can be explained in terms of parallelism. We can think of the angelic specification as running a separate process for each possible choice. Each of these processes continues to run the rest of the program based upon this choice. Any of the processes which find a solution based upon their choice may be selected as an acceptable execution of the program.

In certain cases angelic nondeterminism can also be explained in terms of backtracking. When the angelic specification is "executed" one of the alternatives is chosen. The rest of the program tries to find a solution based upon this choice. If it can then the program terminates successfully, otherwise it backtracks to the choice point and another alternative is selected. The similarities between this operational interpretation and the operational interpretation of Floyd's language are obvious. The differences will be discussed in Section 7.

3 Angelic Refinement

In this section we illustrate some applications of angelic specifications by using them in the formal refinement of a general list problem. Each step of the refinement appeals to a *refinement law.* These laws show how one piece of code can be replaced by another, while still guaranteeing correctness. They can be proved using the weakest precondition formalism as follows: program P is refined by program Q, written $P \sqsubseteq Q$ if $P(R) \Rightarrow Q(R)$ (for more details see [MoR87]).

3.1 Specification

We wish to assign a value to a sequence, a : seq T, such that a predicate $P\,a$ holds and the length of a is N, for $N \geqslant 0$. We assume that T is finite and that P holds for the empty sequence. We also place the restriction on P that if $P\,a$ holds then, for all prefixes a' of a, $P\,a'$ must also hold. That is,

$$P\,a \Rightarrow (\forall\, a' : \text{seq}\ T \bullet a' \subseteq a \Rightarrow P\,a') \,. \tag{1}$$

Assuming that this problem does have a solution our initial (demonic) specification is just

$$\{\exists\, a \bullet P\,a \wedge \#a = N\}\ a : [P\,a \wedge \#a = N] \,.$$

3.2 Refinement

During the development we introduce refinement laws pertaining to angelic nondeterminism as they are needed. For the sake of brevity, when more common refinement laws are used we simply give a reference to similar laws in Morgan's book [Mor90]. The first new law allows us to introduce an angelic specification.

Law 1 (Introduce Angelic Specification)

$$\{\exists\, w \bullet post\}\ w : [post] = w : \overline{[post]};\ \ \{post\}\ w : [post] \qquad \square$$

Since our initial specification is $\{\exists\, a \bullet P\,a \wedge \#a = N\}\ a : [P\,a \wedge \#a = N]$, this law is directly applicable and gives

$$a : \overline{[P\,a \wedge \#a = N]};\ \ \{P\,a \wedge \#a = N\}\ a : [P\,a \wedge \#a = N] \,.$$

Our development starts with a as the empty sequence and extends a one element at a time (making use of property (1)) until it is of length N. Concentrating on the

angelic specification, we initialise a as follows.

$$
\begin{array}{l}
a : \overline{[P\,a \wedge \#a = N]} \\
\sqsubseteq \\
a := \langle\rangle; \\
\{a = \langle\rangle \wedge P\,a\}\ a : \overline{[P\,a \wedge \#a = N]}
\end{array}
\qquad \text{(i)}
$$

This refinement is achieved via a sequential composition law similar to Law 4.2 in [Mor90] and relies on our assumption that $P\langle\rangle$ holds.

Focusing on line (i) above we introduce a logical constant, A, to represent the initial value of a (Law 6.2 in [Mor90]).

$$[\![\ \mathbf{con}\ A : \mathrm{seq}\ T \bullet \{a = A = \langle\rangle \wedge P\,a\}\ a : \overline{[P\,a \wedge \#a = N]}\ ]\!]$$

Before continuing we introduce a law which enables us to refine angelic specifications. An angelic specification is refined if its angelic nondeterminism is increased [BaW89]. That is, it is refined if any of the sets of values it can choose for its variables is increased.

Law 2 (Weaken Angelic Postcondition)

$$\{pre\}\ w : \overline{[post]} \sqsubseteq \{pre\}\ w : \overline{[post']}$$

if $pre \Rrightarrow (\forall w \bullet post \Rightarrow post')$ □

Although this law can be used to weaken an angelic specification it is more commonly used to refine an angelic specification to an *equivalent* angelic specification. It is interesting to note that the corresponding refinement law for demonic specifications decreases rather than increases the demonic nondeterminism. This is a consequence of a *duality* between demonic and angelic nondeterminism which is investigated in [BaW90].

Using Law 2 we can refine the text within the block to

$$\{a = A = \langle\rangle \wedge P\,a\}\ a : \overline{\left[\begin{array}{c} A \subseteq a \\ P\,a \wedge \#a = N \end{array}\right]}.$$

Next we weaken the assumption to $\{P\,a \wedge a = A\}$ (Law 1.2 in [Mor90]) so the specification is more general and collect the program fragments developed since the introduction of the constant block.

$$\left[\!\left[\ \mathbf{con}\ A : \mathrm{seq}\ T \bullet \{P\,a \wedge a = A\}\ a : \overline{\left[\begin{array}{c} A \subseteq a \\ P\,a \wedge \#a = N \end{array}\right]}\ \right]\!\right]$$

Informally this specification says "assuming $P\,a$ holds, angelically choose values for elements $(\#a+1)$ to N so that $P\,a$ holds." This suggests a recursive solution which chooses a value for one element at a time. Let *Prog* be the above program, then using a recursive block introduction rule similar to Law 14.1 in [Mor90], we have

$$\begin{array}{lll} Prog & \sqsubseteq & \mathbf{re}\,R \mathrel{\widehat{=}} \{V < \#a \leqslant N\} \bullet \\ & & \quad \{V = \#a\}\ Prog \end{array}$$

Here $\#a$ is the variant of the recursion which is bounded above by N and V is a logical constant representing its initial value. The program to be refined is $\{V = \#a\}\ Prog$. During the development we can replace any program fragments of the form $\{V < \#a \leqslant N\}Prog$ with a recursive call to R. Note that to call R we are forced to push the variant toward its bound.

To continue we move $\{V = \#a\}$ inside the constant block and concentrate on the text within that block, namely

$$\left\{ \begin{array}{c} V = \#a \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} A \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]}.$$

We proceed by realising that if $\#a = N$ we have finished. Thus, using Law 5.1 in [Mor90] we introduce an **if** statement.

$$\begin{array}{l} \textbf{if}\ \#a = N \longrightarrow \\ \quad \left\{ \begin{array}{c} V = \#a = N \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} A \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]} \qquad \text{(ii)} \\ [\!]\ \#a < N \longrightarrow \\ \quad \left\{ \begin{array}{c} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} A \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]} \qquad \text{(iii)} \\ \textbf{fi} \end{array}$$

The first alternative can be refined to $\{a = A\}a : \overline{[a = A]}$ by applying Law 2 and then weakening the assumption. This can then be refined to **skip** with the aid of the following law.

Law 3 (Skip Introduction)

$$\{w = W\}\ w : \overline{[w = W]} \sqsubseteq \textbf{skip} \qquad \square$$

We make progress on the **if** statement's second alternative by angelically choosing a value for the next element of the sequence, $a(\#A + 1)$ and then calling R again. We store the value of $a(\#A + 1)$ in a new variable, x. The following law allows us to introduce this variable and angelically initialise it.

Law 4 (Angelic Local Block)

$$\{pre\}\ w : \overline{[post]} \sqsubseteq [\![\ \textbf{var}\ x : T \bullet \{pre\}\ x : \overline{[x \in T]};\ \ w : \overline{[post']}\ ]\!]$$

if x is a fresh name and $pre \Rrightarrow (post \Rightarrow (\exists x : T \bullet post'))$ $\square$

Using this law we refine (iii) to

$$\left[\!\!\left[\textbf{var}\ x : T \bullet \left\{ \begin{array}{c} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} x : \overline{[x \in T]};\ \ a : \overline{\left[\begin{array}{c} A \frown \langle x \rangle \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]} \right]\!\!\right] \qquad \text{(iv)}$$

To append x to a we move the assumption through the choice for x.

Law 5 (Independent Assumption)

$$\{pre\}\ w : \overline{[post]} = w : \overline{[post]}\ \{pre\}$$

provided w does not occur free in pre. □

Since the assumption makes no reference to x we can use Law 5 to move it through the choice for x.

$$\text{(iv)} \sqsubseteq x : \overline{[x \in T]};\ \left\{ \begin{array}{c} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} A \frown \langle x \rangle \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]}$$

Next we use

Law 6 (Angelic Sequential Composition)

$$\{pre\}\ w : \overline{[Q]} \sqsubseteq \{pre\}\ w : \overline{[P]};\ \ w : \overline{[Q]}$$

if $pre \Rrightarrow ((\exists\, w \bullet Q) \Rightarrow (\exists\, w \bullet P))$ □

to refine the assumption and the angelic choice for a to

$$\left\{ \begin{array}{c} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} a = A \frown \langle x \rangle \\ P\,a \wedge V < \#a \leqslant N \end{array} \right]};$$
$$a : \overline{\left[\begin{array}{c} A \frown \langle x \rangle \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]}. \qquad \text{(v)}$$

The side condition for this refinement follows from assumption (1) about P.

The above program fragment appends x to a and then angelically chooses values for the rest of a. We want to make this choice for the rest of a a recursive call to R. To do this we need an assumption preceding it. The following law facilitates this.

Law 7 (Trailing Assumption)

$$w : \overline{[post \wedge post']} = w : \overline{[post]}\ \{post'\} \qquad \square$$

We apply a specialisation of this law where $post = post'$ to line (v) above. The result is

$$\left\{ \begin{array}{c} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} a = A \frown \langle x \rangle \\ P\,a \wedge V < \#a \leqslant N \end{array} \right]};$$
$$\left\{ \begin{array}{c} a = A \frown \langle x \rangle \\ P\,a \wedge V < \#a \leqslant N \end{array} \right\} a : \overline{\left[\begin{array}{c} A \frown \langle x \rangle \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]}.$$

The last line is now similar R. We refine it by introducing another constant block,

$$\left[\mathbf{con}\ A' : \mathrm{seq}\ T \bullet \left\{ \begin{array}{c} a = A' = A \frown \langle x \rangle \\ P\,a \wedge V < \#a \leqslant N \end{array} \right\} a : \overline{\left[\begin{array}{c} A \frown \langle x \rangle \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]} \right].$$

We remove all references to A and x from this block by applying Law 2 and then weakening the assumption. The result is

$$\left[\!\left[\mathbf{con}\ A' : \mathrm{seq}\ T \bullet \left\{ \begin{array}{c} V < \#a \leqslant N \\ P\,a \wedge \#a = A' \end{array} \right\} a : \overline{\left[\begin{array}{c} A' \subseteq a \\ P\,a \wedge \#a = N \end{array} \right]} \right]\!\right] .$$

If we call this text R' then the program developed since the introduction of the recursive block is

$$\begin{array}{l} [\![\ \mathbf{con}\ A : \mathrm{seq}\ T \bullet \\ \quad \mathbf{if}\ \#a = N \longrightarrow \mathbf{skip} \\ \quad [\!]\ \#a < N \longrightarrow \\ \qquad [\![\ \mathbf{var}\ x : T \bullet \\ \qquad\quad x : \overline{[x \in T]};\ \left\{ \begin{array}{c} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} a : \overline{\left[\begin{array}{c} V < \#a \leqslant N \\ a = A \frown \langle x \rangle \wedge P\,a \end{array} \right]};\ R' \\ \qquad]\!] \\ \quad \mathbf{fi} \\ ]\!] \ . \end{array}$$

This is easily refined to

$$\begin{array}{l} \mathbf{if}\ \#a = N \longrightarrow \mathbf{skip} \\ [\!]\ \#a < N \longrightarrow \\ \qquad [\![\ \mathbf{con}\ A : \mathrm{seq}\ T \bullet \\ \qquad\quad [\![\ \mathbf{var}\ x : T \bullet x : \overline{[x \in T]};\ \{a = A\}\ a : \overline{[a = A \frown \langle x \rangle]}\ ]\!] \qquad \text{(vi)} \\ \qquad]\!] ; \\ \qquad R' \\ \mathbf{fi} \ . \end{array}$$

Since R' is now outside the constant block for A we can rename A' to A in R', making it equal to R.

The choice for a on line (vi) can be refined to an assignment statement using the following definition.

Definition 4 (Simple Angelic Specification)

$$w := E \ = \ [\![\ \mathbf{con}\ W \bullet \{w = W\}\ w : \overline{[w = E_0]}\ ]\!]$$

where E_0 is $E[w \backslash W]$. □

Using this $\{a = A\}\ a : \overline{[a = A \frown \langle x \rangle]}$ on line (vi) becomes $a := a \frown \langle x \rangle$.

3.3 Program

Collecting all of the program fragments and making the recursive block a procedure gives

$$
\begin{array}{l}
\textbf{proc}\, R \mathrel{\widehat{=}} \\
\quad \textbf{if}\, \#a = N \longrightarrow \textbf{skip} \\
\quad [\!] \, \#a < N \longrightarrow \\
\qquad [\![\, \textbf{var}\, x : T \bullet x : \overline{[x \in T]};\ \ a := a \frown \langle x \rangle \,]\!] \,;\ R \\
\quad \textbf{fi} \bullet \\
a := \langle\rangle;\ R;\ \{P\, a \wedge \#a = N\}\ a : [P\, a \wedge \#a = N]
\end{array}
$$

Since procedure R is tail recursive it can be refined to a **do** loop (Law B.4 in [Mor90]). The result is

$$
\begin{array}{l}
a := \langle\rangle; \\
\textbf{do}\, \#a < N \longrightarrow \\
\qquad [\![\, \textbf{var}\, x : T \bullet x : \overline{[x \in T]};\ \ a := a \frown \langle x \rangle \,]\!] \\
\textbf{od}; \\
\{P\, a \wedge \#a = N\}\ a : [P\, a \wedge \#a = N]\ .
\end{array}
$$

4 New Constructs

In this section we introduce two new constructs into our language. These are similar to the language constructs used to implement backtracking nondeterminism in [Flo67, Lin79, LiS90]. They allow us to further refine the program presented at the end of the last section.

The first construct, $w : \textbf{guess}(S)$, angelically chooses values for w from the set S, provided that S is finite.

Definition 5 (Guess) If T is the type of w, then

$$w : \textbf{guess}(S) = \{S \in \mathbb{F}\, T\}\ w : \overline{[w \in S]} \qquad \square$$

It can be introduced via the following law.

Law 8 (Introduce Guess) Let $S = \{w : T \mid post\}$, where T is the type of w, then

$$w : \overline{[post]} \sqsubseteq w : \textbf{guess}(S)$$

if $S \in \mathbb{F}\, T$. $\square$

The second construct, $w : \textbf{succeed}(cond)$, restricts the choices that any angelic specifications preceding it can make for w. All of the choices preceding it must choose w so that *cond* holds at this point.

Definition 6 (Succeed)

$$\{cond\}\ w : [cond] = w : \mathbf{succeed}(cond) \qquad \square$$

Using these abbreviations and Law 8 we can refine our program to

$$\begin{array}{l} a := \langle\rangle; \\ \mathbf{do}\ \#a < N \longrightarrow \\ \quad [\![\ \mathbf{var}\ x : T \bullet x : \mathbf{guess}(T);\ \ a := a \frown \langle x \rangle\ ]\!] \\ \mathbf{od}; \\ a : \mathbf{succeed}(P\,a \wedge \#a = N)\ , \end{array}$$

since we originally assumed T was finite.

5 Pruning

Currently our choice for the next element of the sequence (i.e. x) uses no information from the context in which the choice is made; it just chooses a value so that $x \in T$. If we can make a more educated guess for x then the resulting program will be more efficient. Restricting the choices for x prunes the search tree which the program traverses.

Toward the end of the refinement in Section 3 the angelic choice for x was in the following context.

$$[\![\ \mathbf{con}\ A : \mathbf{seq}\ T \bullet \left[\ \mathbf{var}\ x : T \bullet \left\{ \begin{array}{l} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} x : \overline{[x \in T]};\ \ a : \overline{\left[\begin{array}{l} V < \#a \leqslant N \\ a = A \frown \langle x \rangle \wedge P\,a \end{array} \right]} \right] \ ]\!]$$

We concentrate on refining the text within the inner block with aim of restricting the choice for x. This requires use of the following law.

Law 9 (Implicit Angelic Precondition)

$$w : \overline{[post]} = \{\exists\, w \bullet post\}\ w : \overline{[post]} \qquad \square$$

If we apply this to the angelic choice for a we have.

$$\begin{array}{l} \left\{ \begin{array}{l} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} x : \overline{[x \in T]}; \\ \left\{ \exists\, a \bullet \begin{array}{c} V < \#a \leqslant N \\ a = A \frown \langle x \rangle \wedge P\,a \end{array} \right\} a : \overline{\left[\begin{array}{l} V < \#a \leqslant N \\ a = A \frown \langle x \rangle \wedge P\,a \end{array} \right]} \end{array}$$

The assumption on the second line can be weakened to $P(A \frown \langle x \rangle)$ and absorbed into the choice for x using Law 7. The result is

$$\left\{ \begin{array}{l} V = \#a < N \\ P\,a \wedge a = A \end{array} \right\} x : \overline{\left[\begin{array}{c} P(A \frown \langle x \rangle) \\ x \in T \end{array} \right]};\ \ a : \overline{\left[\begin{array}{l} V < \#a \leqslant N \\ a = A \frown \langle x \rangle \wedge P\,a \end{array} \right]}.$$

The angelic choice for x can then be refined using Law 8 to

$$x : \mathbf{guess}(\{y : T \mid P(A \frown \langle y \rangle)\}) .$$

This is exactly what we wanted. The choice for x has now been restricted. Previously any choice for x (such that $x \in T$) was satisfactory, now x must be chosen so that its addition to a does not violate P.

6 N-queens Problem

In this section we apply the results of sections 3, 4 and 5 to a specific list problem. We wish to place N queens on an N by N chess board such that no queen can take any of the others. We assume that N is greater than 3.

6.1 Specification

A solution to the problem can be modelled as a set of ordered pairs $(row, column)$ where each ordered pair represents the position of one queen. Since no two queens can take each other, no two queens can be on the same row. Thus the set of ordered pairs is actually a function from rows to columns:

$$q : 1..N \nrightarrow 1..N .$$

Assuming that our problem does have a solution, the initial specification is just

$$\{\exists\, q \bullet nc(q) \wedge \#q = N\}\ q : [nc(q) \wedge \#q = N] ,$$

where $nc(q)$ is true if and only if, in the placement q, no two queens can capture each other.

6.2 Refinement

For the results from the previous three sections to be applicable to this problem we must data refine q so that it is a sequence rather than a partial function. This data refinement is trivial and leaves us with the declaration

$$q : \mathrm{seq}(1..N)$$

and the specification

$$\{\exists\, q \bullet nc(q) \wedge \#q = N\}\ q : \overline{[nc(q) \wedge \#q = N]}$$

We must also prove that $nc(q)$ satisfies condition (1). That is,

$$nc(q) \Rightarrow (\forall\, q : \mathrm{seq}(1..N) \bullet q' \subseteq q \Rightarrow nc(q')) .$$

Suppose this were not true. Then, for any placement of queens in which no two queens can capture each other, there could exist a subset of queens in which a pair of queens do capture each other — obviously a contradiction.

Having massaged the N-queens problem into the correct form we can now apply the results of Sections 3, 4 and 5 to it. The specification,

$$\{\exists\, q \bullet nc(q) \wedge \#q = N\}\ q : \overline{[nc(q) \wedge \#q = N]}\ ,$$

can be refined to the nondeterministic program

$$\begin{array}{l} q := \langle\rangle; \\ \textbf{do}\ \#q < N \longrightarrow \\ \quad [\![\ \textbf{var}\ x : (1..N) \bullet x : \textbf{guess}(\{y : 1..N \mid nc(q \frown \langle y \rangle)\});\ q := q \frown \langle x \rangle\ ]\!] \\ \textbf{od}; \\ q : \textbf{succeed}(nc(q) \wedge \#q = N)\ . \end{array}$$

In [Dij72] Dijkstra's development of the N-queens problem includes the introduction of variables to keep track of which columns, upward diagonals and downward diagonals are in check from any of the queens currently placed. These variables could be introduced into our solution at this stage via a data refinement. Using these variables we could then further restrict the choice for x using the techniques of Section 5.

7 Conclusions

We have investigated using angelic nondeterminism in the systematic refinement of specification to code containing "nondeterministic" constructs. We successfully applied our techniques to the refinement of a general list problem and then to a specific instance of this: the N-queens problem. During this development "nondeterministic" language constructs were introduced and general refinement laws involving angelic nondeterminism were developed. Ways to make the resulting program more efficient were also examined.

Although not presented here, another refinement of the N-queens problem which did not use angelic nondeterminism was undertaken. This alternative refinement was more complex. The development was longer and the nature of the solution obtained was obscured by the conventional recursive implementation of backtracking.

The nondeterministic language we have presented is similar to Floyd's original language but not identical. Our language is defined in terms of angelic nondeterminism which can achieve more than Floyd's "backtracking" nondeterminism. The ability of an angelic specification to "look ahead" and choose execution paths which give the "correct" answer means that it can avoid divergence. Floyd's backtracking nondeterminism cannot do this. If it chooses an alternative which diverges then no more alternatives are investigated and the whole program diverges. It is interesting to note that an implementation based on the parallel interpretation of angelic nondeterminism does not suffer from this problem: even if some of the parallel processes diverge at least one is guaranteed to terminate.

In summary, angelic nondeterminism is useful for introducing nondeterministic constructs into procedural programs using the refinement paradigm. Related work by Joost Kok has used angelic nondeterminism to characterise refinement to logic programs [Kok90].

Both Morgan [MoR87] and Nelson [Nel89] mention a relationship between (demonic) miracles and backtracking algorithms. Further research could involve an investigation of this relationship and a comparison with the nondeterminism developed here.

Acknowledgements

Acknowledgement is due to Ken Robinson who first introduced us to the notion of conjugate weakest preconditions in 1986 and to Ralph Back's work on angelic nondeterminism. The authors wish to thank Carroll Morgan and Ralph Back for their thoughts on an earlier draft of this paper.

References

[BaW89] R. J. R. Back & J. von Wright, "Refinement Calculus, Part I: Sequential Nondeterministic Programs," in *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science #430, 1989, 42–66.

[BaW90] R. J. R. Back & J. von Wright, "Duality in Specification Languages: A Lattice-theoretical Approach," *Acta Informatica* 27 (1990), 583–625.

[Dij72] E. W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, O. -J. Dahl, E. W. Dijkstra & C. A. R. Hoare, eds., Academic Press, London, 1972.

[Dij76] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Flo67] R. W. Floyd, "Nondeterministic Algorithms," *Journal of the ACM* 14 (1967), 636–644.

[Kok90] J. N. Kok, "On Logic Programming and the Refinement Calculus: Semantics Based Program Transformations," Department of Computer Science, Utrecht University, Technical Report RUU-CS-90-39, 1990.

[Lin79] G. Lindstrom, "Backtracking in a Generalised Control Setting," *ACM Transactions on Programming Languages and Systems* 1 (1979), 8–26.

[LiS90] Yaowei Liu & John Staples, "btC: a Backtracking Procedural Language," working paper, Department of Computer Science, University of Queensland, 1990.

[Mor90] Carroll Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[MoR87] Carroll Morgan & Ken Robinson, "Specification Statements and Refinement," *IBM Journal of Research and Development* 31 (1987).

[Mor87] J. M. Morris, "A Theoretical Basis for Stepwise Refinement and the Programming Calculus," *Science of Computer Programming* 9 (1987), 287–306.

[Nel89] Greg Nelson, "A Generalization of Dijkstra's Calculus," *ACM Transactions on Programming Languages and Systems* 11 (1989).