EPiC Series in Computer Science
Volume 35, 2015, Pages 106–118
LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations

Compiling Hilbert's ε operator

K. Rustan M. Leino

Microsoft Research
leino@microsoft.com

Abstract

Hilbert's ε operator is a binder that picks an arbitrary element from a nonempty set. The operator is typically used in logics and proof engines. This paper contributes a discussion of considerations in supporting this operator in a programming language. More specifically, the paper presents the design choices made around supporting this operator in the verification-aware language Dafny.

1 Introduction

When describing an algorithm or giving a proof, it is common exclaim "pick an arbitrary x such that $P(x)$ holds", where $P(x)$ denotes some predicate on x. Such an operation can provide a level of abstraction, because what follows might not depend on the exact value of x, as long as $P(x)$ holds of it. In other cases, the operation may describe one step that is going to be repeated for every x satisfying $P(x)$. In formal mathematics, this operation is known as Hilbert's ε operator [6, 7], and its name stems from the fact that Hilbert used a notation like $\varepsilon x \bullet P(x)$ to denote an *expression* that evaluates to some value satisfying the predicate P.[0]

In this paper, I present the manifestation of Hilbert's ε operator in the verification-aware programming language Dafny [10]. Including this operator in a programming language poses challenges for referential transparency and for compilation. The considerations I bring up are applicable to other languages as well, even if other languages end up making different design decisions.

2 Dafny

In this section, I present some characteristics of Dafny that are relevant to this paper. More documentation of the language is found from the Dafny open-source repository dafny.codeplex.com.[1]

The Dafny language includes imperative constructs (assignments, loops, classes, etc.) and functional constructs (higher-order functions, inductive and coinductive datatypes, mathematical sets, etc.). The language also includes facilities for writing specifications and proofs. Dafny comes equipped with a verifier, which checks that the program meets its specifications and that program constructs are used correctly.

[0] In Hilbert's work, $\varepsilon x \bullet P(x)$ was a total expression that could be used even when no x satisfies $P(x)$ (but guaranteeing to return a value satisfying P only if one exists, of course). In this paper, I will instead forbid such uses of ε.

[1] Dafny can also be used directly in a web browser at rise4fun.com/dafny.

As is common among programming languages, Dafny has both *statements* and *expressions*. The statements can mutate the program state and can be *nondeterministic*. Expressions are *deterministic* (in other words, they are *functional* in their variables) and, of course, do not mutate the state.

Less common among languages is that Dafny distinguishes between *compiled* entities and *ghost* entities. A ghost entity—e.g., a ghost variable, ghost declaration, ghost expression, or ghost statement—is used only for *reasoning* about the program and is erased by the compiler. For instance, a ghost variable occupies no space at run time and an `assert` statement (which is always ghost) is not evaluated at run time.

Dafny can be used not only for writing specifications and executable code, but also for writing lemmas and proofs. These two "program" and "proof" fragments of the language have a high degree of overlap and can be mixed. For example, to help the verifier reason about a program, it is common to call lemmas at certain points in the program. In fact, in Dafny, lemmas are simply ghost methods and proofs are ghost statements.

3 Nondeterminism

Dafny contains several variations on the ε operator. In this section, I discuss the least problematic variations, which occur in statements.

3.1 Assign Such That

In addition to the ordinary assignment statement, `x := E;`, Dafny supports an *assign such that* statement: for any variable `x` and boolean expression `R`, the semantics of

```
x :| R;
```

is to set `x` to some value such that `R` will evaluate to `true`. (Note that, in difference to the ordinary assignment statement where occurrences of `x` in the right-hand side refer to the value of `x` *before* the update, occurrences of `x` in the right-hand side of the assign-such-that statement refer to the value `x` will have *after* the update.) A proof obligation for the assign-such-that statement is that some value for `x` satisfying `R` exists.[2] As with the ordinary assignment statement, it is possible to combine the declaration of the left-hand-side variable with its initial assignment, which is done by writing `var` in front of the statement.

For example, the following statement may be used in an implementation of the QuickSort algorithm:

```
var pivot :| lo ≤ pivot < hi;
```

because the correctness of the algorithm does not depend on which such `pivot` index is chosen. (The value chosen does matter for performance. Dafny includes features that let a programmer tighten up, in a *refinement module*, how the choice of `pivot` is made. [8, 12])

The assign-such-that statement is *nondeterministic* in that the language makes no guarantee about which value satisfying `R` will get picked. In fact, the value assigned to `x` may be different each time the statement is executed, even if the program state is otherwise the same.

Figure 0 shows an example use of the assign-such-that statement in a proof. The example models the semantics of a simple (and, frankly, rather uninteresting) language consisting of `Inc` commands and sequential composition. The state is represented by an integer and the `Inc` command increases that integer by 1. Predicate `BigStep` defines the big-step semantics of this language as the least fixpoint of

[2] Moreover, the expression `R` must be well defined for any `x` of its type.

```
datatype Cmd = Inc | Seq(Cmd, Cmd)
type State = int

inductive predicate BigStep(c: Cmd, s: State, t: State)
{
  match c
  case Inc ⇒
    t == s + 1
  case Seq(c0, c1) ⇒
    ∃s' • BigStep(c0, s, s') ∧ BigStep(c1, s', t)
}

inductive lemma Monotonicity(c: Cmd, s: State, t: State)
  requires BigStep(c, s, t)
  ensures s ≤ t
{
  match c
  case Inc ⇒
    // proof is trivial
  case Seq(c0, c1) ⇒
    // introduce an s' satisfying the two conjuncts
    var s' :| BigStep(c0, s, s') ∧ BigStep(c1, s', t);
    // invoke the induction hypothesis on the two sub-commands
    Monotonicity(c0, s, s');
    Monotonicity(c1, s', t);
}
```

Figure 0. An example where the proof of lemma `Monotonicity` uses an assign-such-that statement to, essentially, Skolemize the existentially quantified variable `s'` in the definition of `BigStep`.

the following two inference rules:

$$\frac{t = s+1}{BigStep(Inc,s,t)} \qquad \frac{BigStep(c0,s,s') \qquad BigStep(c1,s',t)}{BigStep(Seq(c0,c1),s,t))}$$

Lemma `Monotonicity` says that from `BigStep(c, s, t)` it follows that `s ≤ t`. In the case where `c` has the form `Seq(c0, c1)`, the definition of `BigStep` tells of the existence of a middle state `s'` in the derivation of `BigStep(c, s, t)`. The proof of the lemma wants to get its hands on such a middle state, so it uses an assign-such-that statement to, essentially, Skolemize the existential quantification in the definition of `BigStep`. The proof names the middle state `s'` and uses it to invoke the induction hypothesis once for each sub-command of the sequential composition.[3]

[3] Figure 0 illustrates the typical shape of many proofs. However, for `Monotonicity` and other simple cases, this is not the simplest proof: because Dafny calls the induction hypothesis automatically, the body of `Monotonicity` could have been left empty. [11, 13]

3.2 Verification

As I mentioned above, an assign-such-that statement `x :| R(x);` gives rise to a proof obligation that such an `x` exists. Formally, this proof obligation is

$$\exists x \bullet R(x)$$

That sounds straightforward enough, but Dafny builds on the first-order satisfiability-modulo-theories (SMT) based Boogie/Z3 tool chain [0, 3]. Ironically, SMT solvers are notoriously bad at proving such existential quantifiers *when they are simple*, or more precisely, when the existential quantifier uses only interpreted symbols. For example, the innocent-looking proof obligation for the assignment to `pivot` above is

$$\exists\, pivot \bullet lo \leq pivot \wedge pivot < hi$$

which follows from `lo < hi`. To handle these cases more smoothly, Dafny performs a simple syntactic scan of the formula `R(x)`, looking for some candidate witnesses `w0`, `w1`, ... à la [14]. Dafny then elaborates the proof obligation into:

$$R(w0) \ \vee \ R(w1) \ \vee \ \cdots \ \vee \ \exists x \bullet R(x)$$

which in practice works more often.

3.3 Compilation

As I have shown, assign-such-that statements can occur in both ghost contexts and compiled contexts. When they are to be compiled, there is fertile ground for advanced synthesis techniques to kick in and produce good code. Alas, this is not done by the current Dafny compiler. But even without such advanced compilation techniques, doing the compilation correctly still requires some care.

The main idea behind the compilation is to emit a loop that will enumerate the values in the "domain" of `x` and to test, for each such value, whether or not `R` holds. Once a value satisfying `R` has been found, the loop exits. Since the verifier has ascertained that a value for `x` satisfying `R` does exist, the loop will eventually terminate, provided the enumeration is *exhaustive*, that is, that for each value `w` in the domain of `x`, the enumeration will eventually reach `w`.[4]

The "domain" is some subset of the type of `x` that is both large enough not to exclude all values of `x` for which `R` holds and easy enough to enumerate exhaustively. Types with only a finite number of values (for example, `bool` and user-defined datatypes with only nullary constructors) can be used as their own domain. The integers can also serve as their own domain, since they can be enumerated as follows:

```
0, 1, -1, 2, -2, 3, -3, 4, -4, ..., 7, ..., -13, ...
```

Dafny will use this enumeration for integers, unless it finds a smaller domain.

Dafny does a simple syntactic analysis on the top-level conjuncts of `R` (or, rather, of `R` after some simple distributions of negations), looking for bounds on `x`. For example, if an integer inequality can in simple steps be rewritten into `x` $\leq$ `E` or `E` $\leq$ `x` where `E` is an expression that does not depend on `x`, then this inequality is used to reduce the size of the domain. Also, if a conjunct says `x in S` where `S` denotes a finite set, finite multiset, or sequence and `S` does not depend on `x`, then the elements of `S` are used as the domain.

In some cases, the type of `x` and the simple syntactic analysis do not reveal any good domain. In those cases, Dafny gives up and generates a compilation error. This can happen when the type of `x` is not enumerable (like for `real` and for coinductive datatypes) or possibly not enumerable (if the type is a

[4] LEAN calls this property *listable* [4].

type parameter). Finally, in some cases, Dafny gives up because the enumeration would have too high of a cost at run time (for example, if `x` is of a reference type and the syntactic analysis does not bear any fruit, then its domain would be all currently allocated objects of that type) or because the compiler implementation itself would require more effort than seems justifiable for the additional programs that could compile (for example, the inductive datatype `Cmd` in Figure 0, or a variation of it where `Inc` takes an integer argument).

Dafny allows the left-hand side of `:|` to be a *list* of variables. Here, it is important that the enumerations are interleaved properly. For example, consider

```
x, y :| 0 ≤ x ∧ 0 ≤ y ∧ Both1(x, y);
```

where `Both1` is a predicate that holds only if both its arguments are `1`. It would be incorrect to compile this assignment into two nested loops like

```
foreach (var xx in UpwardsOf(0)) {
  foreach (var yy in UpwardsOf(0)) {
    if (0 ≤ xx ∧ 0 ≤ yy ∧ Both1(xx, yy)) {
      x := xx;
      y := yy;
      break break;
    }
  }
}
```

because this would have the effect of never trying more than one value for `x`. A more clever interleaving of the loops could work. As a simple (but, again, without bragging rights of being efficient) solution, Dafny generates code that works in successively larger rounds, where a round puts a limit on the number of iterations that any of the loops may perform. For the example above, Dafny generates:

```
var maxIterations := 5;
while (true) {
  var budget_x := maxIterations;
  foreach (var xx in UpwardsOf(0)) {
    if (budget_x == 0) { break; }
    budget_x := budget_x - 1;
    var budget_y := maxIterations;
    foreach (var yy in UpwardsOf(0)) {
      if (budget_y == 0) { break; }
      budget_y := budget_y - 1;
      if (0 ≤ xx ∧ 0 ≤ yy ∧ Both1(xx, yy)) {
        x := xx;
        y := yy;
        break break break;
      }
    }
  }
  maxIterations := 2 * maxIterations;
}
```

As a small optimization, a loop is not held to a budget if it has been determined that the domain it is enumerating over is finite.

As a final note, it seems that an "obvious" idea is to improve compilation dramatically by simply

using the witness value that the verifier had used in the existence proof. This would be a great idea to follow up on, but it wouldn't always be easy. First, the witness may not have been determined concretely by the SMT solver. For example, it could be that the proof of the existence follows from a lemma that the program invokes before the assign-such-that statement. The verifier would then need to determine the need to remember the witness that in some way was used inside the proof of the lemma. Second, the witness may depend on the program state. That is, the SMT solver may use different witnesses along different branches of the proof. The compiled code would then need to have enough information to set up the same branching structure at run time as the SMT solver used in the proof. Third, the SMT solver is based on classical (not constructive) logic, so the existence proof may potentially have made use of the Law of the Excluded Middle. If so, there may not be a particular witness to point to.

3.4 Summary

In this section, I discussed the least problematic of Dafny's variations of the ε operator, the assign-such-that statement. It is least problematic, because the statement is allowed to be nondeterministic and is free to choose a different value each time it is executed. In this sense, it isn't really Hilbert's ε operator, because Hilbert's ε operator is axiomatized to be deterministic. We'll look at that next.

4 Determinism

In this section, I consider Dafny's variations of the ε operator that occur in expression contexts, where it has to be deterministic.

4.1 Let Such That

In addition to the ordinary let expression, which in Dafny is written `var x := E; Body`, Dafny supports a *let such that* expression: for any variable `x`, boolean expression `R`, and any expression `Body`, the semantics of

```
var x :| R; Body
```

is to bind `x` to some value such that `R` will evaluate to `true`, and then to evaluate `Body`. As for the assign-such-that statement, a proof obligation for the let-such-that expression is that some value for `x` satisfying `R` exists.

For example, the following function returns an index into the given sequence where the element `x` occurs:

```
function Find⟨T⟩(s: seq⟨T⟩, x: T): nat
  requires x in s
{
  var i :| 0 ≤ i < |s| ∧ s[i] == x;
  i
}
```

The precondition of this function, declared with the keyword `requires`, implies the existence of a value that satisfies the constraint of the let-such-that expression.

Since let-such-that is an expression, the Dafny language says it is deterministic. Determinacy of expressions is important, because it means that for any well-defined expression `E`,

$$\mathtt{E} == \mathtt{E} \tag{0}$$

is always `true`, just like in mathematics. Consequently, if `Find(s, x)` is well defined, then the equality `Find(s, x) == Find(s, x)` holds, which implies that the let-such-that expression in the body of `Find` must return the same value both times.

By what I have just said, we would also have

```
(var x :| R; Body) == (var x :| R; Body)
```

But this is a slippery slope. By alpha conversion, wouldn't we then also expect the following equality to hold:

```
(var x :| R; Body) == (var x' :| R'; Body')
```

where `x'` is fresh and `R'` and `Body'` are like `R` and `Body` but with `x` replaced by `x'`? And to obtain referential transparency, whereby replacing a subexpression by another that always returns the same value, wouldn't we expect equalities like the following also to hold:

```
(var x :| R; Body) == (var x :| (R); Body)
(var x :| x in S; Body) == (var x :| x in S + S; Body)
(var x :| 14 ≤ x < 20 ∧ IsPrime(x); x) == (var x :| x == 17 ∨ x == 19; x)
```

where `S` denotes some nonempty set, `+` is set union, and `IsPrime` is some function that returns whether or not its argument is a prime number? The verifier and compiler both need to know which expressions have to be encoded in the same way, so going down the slippery slope of saying that these equalities must hold would necessitate comparing the quadratically many pairs of let-such-that expressions in the program. Not only would each such comparison require theorem proving in general, but this analysis could not be completed until the whole program is available. This seems untenable.

For this reason, Dafny treats *each textual occurrence* of a let-such-that expression as if it were its own flavor of the ε operator. Stated differently, we can imagine an unbounded number of "colors" of ε operators, where the ε operator of one color is allowed to make its choices independently of the ε operator of other colors. Dafny then says that each textual occurrence of a let-such-that expression in a program gets its own color.

Consequently, `(var x :| R; Body) == (var x :| R; Body)` is not guaranteed in Dafny. More generally, the equality in (0) holds only if `E` does not directly mention any let-such-that expressions. However, when a let-such-that expression is placed inside a named function, like in `Find` above, then the expression `Find(x, s)` does not directly mention a let-such-that, so then the language does guarantee the equality `Find(x, s) == Find(x, s)`.

4.2 Verification

Following the ideas of anonymous lambda lifting,[5] the Dafny verifier proceeds as follows for each textual occurrence of an expression `var x :| R; Body`. It collects the non-`x` free variables of `R`, say `vars`, and introduces a fresh function symbol, say `f`, parameterized by `vars`.[6] To give `f` meaning, the verifier lays down the axiom

$$\forall vars \bullet (\exists x \bullet R) \implies R'$$

where `R'` denotes `R` in which occurrences of `x` have been replaced by `f(vars)`. For example, for the let-such-that expression `var x :| 0 ≤ x < N; 2*x`, the verifier introduces a function `f` and the axiom

$$\forall N \bullet (\exists x \bullet 0 \le x < N) \implies 0 \le f(N) < N \tag{1}$$

[5] which here might be called "epsilon lifting"

[6] If the function depends on fields in the heap, the verifier will include among `vars` the variable that is encoding the heap.

Actually, since the existential is checked (during well-formedness checking) as a precondition of the let-such-that expression, Dafny introduces a *certificate* that is assumed after the check, and this certificate is used in the antecedent of the axiom. (This is also what Dafny does for user-defined functions.) More precisely, the certificate is another fresh function symbol, `CanCall_f`, parameterized like `f`. Using this certificate, axiom (1) is written:

$$\forall N \bullet \mathit{CanCall\_f}(N) \implies 0 \leq f(N) < N$$

Once function `f` (and function `CanCall_f`) and the axiom above have been introduced, the verifier treats the let-such-that expression like it does the ordinary let expression `var x := f(vars); Body`.

A consequence of this encoding is that the verifier treats a let-such-that expression as being a function of its free variables. For example,

```
var z :| z < x + y; z
```

is a function of `x` and `y`, not of `x + y`, which would give rise a stronger determinacy guarantee.[7] To obtain the latter, one would have to say

```
var s := x + y; var z :| z < s; z
```

4.3 Compilation

For let-such-that expressions in ghost contexts, there is nothing more to say. But a let-such-that expression in a compiled context must be translated into target code that is functional in its arguments. I will illustrate the difficulty of this task through an example.

Consider a compiled function (indicated by the keyword combination `function method`) that picks an arbitrary element from a nonempty set:

```
function method Pick⟨T⟩(S: set⟨T⟩): T
  requires S ≠ {}
{
  var x :| x in S; x
}
```

Suppose `A` and `B` are sets whose union is known to be nonempty. Then, since set union is commutative, the following program is free of assertion failures and free of division-by-zero errors, as indeed the Dafny verifier can prove:

```
var x := Pick(A + B);
var y := Pick(B + A);
assert A + B == B + A;  // this assertion gets used as a lemma from which x = y follows
if x ≠ y {
  var z := 4 / 0;  // error if we ever get here (but the verifier knows we don't, phew)
}
```

Soundness requires agreement between verification and compilation. The compiled code must therefore ensure that the two calls to `Pick` return the value value. Here are some attempts to achieve that:

- *At run time, let `Pick(S)` select the smallest element of `S`.*
  The trouble with this attempt is that the element type of `S` may not have a readily defined notion of "smallest". Plus, it would only work for the specific example `Pick`, anyway.

[7] I thank Jasmin Blanchette for pointing this out to me.

- *At run time, let* `Pick(S)` *select the "first" element in the run-time representation of the given set* `S` *(for example, suppose sets are stored as linked lists).*
  The trouble with this attempt is that the first element of `A + B` may be different than the first element of `B + A`, even though the two expressions denote the same set. Plus, it would only work for the specific example `Pick`, anyway.
- *Before computing the let-such-that expression, canonicalize the run-time representation of each variable involved. That way, the iterative search for the value to return from the let-such-that expression would always proceed in the same way.*
  The trouble with this attempt is that not all types may have a readily defined canonical representation. For example, even a canonical ordering on something as innocuous as `set`⟨`Cmd`⟩ is complex and would take a long time to compute at run time.
- *For each let-such-that expression, maintain at run time an ordered list of values that have been returned. To evaluate the let-such-that expression, first try each value in the list, in order, returning the first value that satisfies the constraint. If the list contains no value that satisfies the constraint, then compute a new value (for example, using the loops described in Section 2.2), append that value to the list, and return it.*[8]
  The trouble with this approach is that list continues to grow as the program runs, which means the program will run more and more slowly.
- *Analyze the program to figure out when it would be okay to clear the list in the previously mentioned attempt.*
  This seems tricky and would require some research to figure out.

From these initial attempts at a solution, our situation does not look good. Luckily, we can instead consider changing the problem.

4.4 Russell's ι operator

Instead of doing enough bookkeeping at run time to make sure a let-such-that expression always chooses consistently among its possible values, we can restrict the language to make sure there is only one possible value to choose from. Dafny takes this approach and adds as a proof obligation, for any compiled let-such-that expression, that the value returned is unique. This means that any run-time technique for finding *some* value finds *the one and only* value.

For a let-such-that expression `var x :| R; Body`, the uniqueness proof obligation is:

$$\forall x, x' \bullet R \wedge R' \implies Body = Body'$$

where `x'` is fresh and `R'` and `Body'` are `R` and `Body` in which `x` has been replaced by `x'`. Note that the right-hand side of the implication in this proof obligation is not `x == x'`, but instead the weaker condition `Body == Body'`.

Technically, with this restriction, the compiled let-such-that statement is no longer Hilbert's ε operator, but instead Russel's *definite description* operator [15], commonly denoted as the binder ι.

Under the definite-description (that is, uniqueness) restriction on compiled let-such-that expressions, function `Pick` above is not admitted. For it to be allowed, the constraint must be strengthened to denote a unique value. In the case of the polymorphic `Pick`, strengthening the constraint is hard to do. We can write a pick function for integers, for example by always picking the smallest integer. As we strengthen the constraint, the existence condition becomes harder to verify. The following definition deals with that problem by calling a lemma just before the let-such-that expression:

[8] Manfred Broy suggested this approach to me.

```
function method PickInt(S: set⟨int⟩): int
  requires S ≠ {}
{
  ThereIsASmallest(S);
  var x :| x in S ∧ forall y • y in S ⟹ x ≤ y; x
}

lemma ThereIsASmallest(S: set⟨int⟩)
  requires S ≠ {}
  ensures ∃x • x in S ∧ forall y • y in S ⟹ x ≤ y
{
  var x :| x in S;
  if S ≠ {x} {
    ThereIsASmallest(S - {x});
  }
}
```

Note that the lemma makes use of a (nondeterministic) assign-such-that statement.

It seems unfortunate that the definition of `PickInt` runs in time $\mathcal{O}(|S|^2)$, not to mention the fact that its definition is more complicated, whereas if nondeterminism were okay, then the statement `x :| x in S;` is both simpler and runs in time $\mathcal{O}(1)$.

5 Examples

I end with two more examples.

5.1 Sum

Using the `PickInt` function above, we can define a function that computes the sum over a finite set of integers:

```
function Sum(S: set<int>): int
{
  if S == {} then 0 else
    var x := PickInt(S);
    x + Sum(S - {x})
}
```

Furthermore, we can state a lemma about the effect on `Sum` of adding an element to a set. The proof uses the fact that `PickInt` picks the smallest integer in the set.

```
lemma SumDistribution(S: set<int>, g: int)
  requires g !in S
  ensures Sum(S + {g}) == Sum(S) + g
{
  var G := S + {g};
  var x := PickInt(G);
  if x ≠ g {
    SumDistribution(S - {x}, g);
    assert G - {x} == S - {x} + {g};
```

```
  }
}
```

5.2 Axiom of Choice

It is possible to define in Dafny a function that selects one element from each of an infinite family of nonempty, possibly infinite sets. In other words, Dafny support the Axiom of Choice.

```
function F⟨T⟩(S: iset⟨T⟩): T
  requires S ≠ iset{}
{
  var x :| x in S; x
}
```

It is tempting to change this ghost function to be a compiled function. However, doing so will generate an error message that Dafny is unable to find an exhaustive enumeration for the let-such-that expression.

6 Related Work

The ε operator is at the heart of the logic of the proof assistant Isabelle/HOL, and there are techniques that attempt to turn formal Isabelle/HOL specifications into executable code. Lochbihler and Bulwahn also reach the conclusion that compilation of ε (and ι) requires uniqueness and their execution strategy is the same as I have described for let-such-that expressions in Dafny. From their experience, they caution against use of the ε operator when compilation is a goal, because refinement from an underspecified function is difficult. As an alternative, they suggest using a relational description, which is amenable to refinement. This parallels the use and refinement of the nondeterministic assign-such-that statement in Dafny.

The standard library of the proof assistant Coq [1] contains variations of the ε operator. These are defined by axioms and generally do not directly lend themselves to compilation. The `ConstructiveEpsilon` library provides an executable form of ε (and ι) in the special case where the constraint is a decidable predicate over the natural numbers. The compilation is based on linear search and thus guarantees determinacy. This special case could also be added to Dafny, providing an alternative to the uniqueness restriction.

Giese and Ahrendt consider variations of the ε operator and the conditions under which such terms are to be considered equivalent [5].

The nondeterministic variation of the ε operator has been studied in the context of logics by Blass and Gurevich [2] and by Wirth [17].

The ASM language also supports Hilbert's ε operator in the form of a "choose" binder [2]. The primary application of ASM laid in run-time testing and there was no static verifier for the language.

Eiffel's AutoProof verifier [16] supports a method `any_item` in its `MML_SET` class, which represents mathematical sets. The verifier treats the result of `S.any_item` as a function of the mathematical set that the object `S` represents. If this were to be evaluated at run time, then one would face the issues I have outlined in Section 3.2. However, the `MML` library is intended for specification purposes only.

Kuncak et al. have considered synthesis where an SMT solver is used at run time [9]. Such an approach can in many cases easily generate better run-time behavior than the compilation scheme I have mentioned here.

7 Conclusion

In this paper, I have presented the design of different variations of the ε operator in the programming language Dafny. The two statement variations are not really Hilbert's ε operator because they are nondeterministic. The compiled let-such-that variation is Russell's ι operator. Only the ghost let-such-that expression is Hilbert's ε operator.

Whatever these variations may be called, the Dafny design shows one possibility of including the choice operator in a language with a semantics that covers both verification and compilation.

In the future, it would be nice to find ways to overcome the difficulty with the compiled let-such-that expressions, in particular to make a compiled function like `PickInt` less complicated to write and more efficient to compile.

7.0.1 Acknowledgments

I'm grateful to the participants of IFIP WG 2.3 meeting 56 in Istanbul, March 2015, for their many insightful comments and feedback. Wolfgang Ahrendt, Jasmin Blanchette, Jean-Christophe Filliâtre, Daniel Grahl, and Yuri Gurevich helped me with related work.

References

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, Sept. 2006.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[3] A. Blass and Y. Gurevich. The logic of choice. *Journal of Symbolic Logic*, 65(3):1264–1310, Sept. 2000.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, Mar.–Apr. 2008.

[5] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction — CADE-25 — 25th International Conference on Automated Deduction*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, Aug. 2015.

[6] M. Giese and W. Ahrendt. Hilbert's ε-terms in automated theorem proving. In N. V. Murray:, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '99*, volume 1617 of *Lecture Notes in Computer Science*, pages 171–185. Springer, June 1999.

[7] D. Hilbert. Die logischen Grundlagen der Mathematik. *Mathematische Annalen*, 88:151–165, 1923.

[8] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 2. Springer Verlag, 1939.

[9] J. Koenig and K. R. M. Leino. Programming language features for refinement. Manuscript KRML 248, Microsoft Research, 2015.

[10] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Comfusy: A tool for complete functional synthesis. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010*, volume 6174 of *Lecture Notes in Computer Science*, pages 430–433. Springer, July 2010.

[11] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, Apr. 2010.

[12] K. R. M. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, Jan. 2012.

[13] K. R. M. Leino. Staged program development. SPLASH 2012 keynote, Oct. 2012. InfoQ video, http://www.infoq.com/presentations/Staged-Program-Development.

[14] K. R. M. Leino. Well-founded functions and extreme predicates in Dafny: A tutorial. In B. Konev, S. Schulz, and L. Simon, editors, *11th International Workshop on the Implementation of Logics*, Nov. 2015.

[15] K. R. M. Leino and R. Middelkoop. Proving consistency of pure methods and model fields. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009*, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, Mar. 2009.

[16] B. Russell. On denoting. *Mind*, 14:479–493, 1905.

[17] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-active functional verification of object-oriented programs. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems — 21st International Conference, TACAS 2015*, volume 9035 of *LNCS*, pages 566–580. Springer, Apr.

[18] C.-P. Wirth. A new indefinite semantics for Hilbert's epsilon. In U. Egly and C. G. Fermüller, editors, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002*, volume 2381 of *Lecture Notes in Computer Science*, pages 298–314. Springer, July–Aug. 2002.