検証指向言語 Dafny で「ちょっと書いて試す」を可能にするループ実装支援に関する考察

吉田 和誠　川端 英之

形式検証向けプログラミング言語 Dafny では手続型プログラムの検証ができるが，ループを実装するにはループ不変条件の明記が必須で，「少し書いて動かしながら整える」というインクリメンタルな開発を好むプログラマにとって敷居が高い．これに対し本研究では，Dafny プログラムにおいて，あたかもループを実行して試すようなやり取りでガードやループボディの妥当性を確かめながら実装を進められるユーザ支援環境の実現方式を検討している．本方式では，ユーザがループ不変条件を与えられなくてもループの実行結果に対して望む条件を徐々に書きながら使えるように，ループを再帰関数に変換して検証する．簡単なループの場合には CHC への変換と Spacer による解析によりループ不変条件を自動導出する．本発表では，VS Code プラグインとして実装したプロトタイプツール TAMESU (Tail-recursive Assertion and Method Engine for Solving Unverified-loops) を紹介しながら，ユーザ支援環境に求められる事項について議論する．

1　はじめに

多くのメインストリーム言語では，反復処理の記述に `for` や `while` 等のループを用いることは慣用的である．ループ構造は再帰呼び出しで表現することもでき，純粋関数型言語ではループ構文が用意されていない場合もあるし，末尾再帰最適化が適用できる場合には再帰呼び出しとループは同じものとみなせるが，ループ構文を用いた反復処理の表現は処理コストを見積りを視覚的に把握し易く，多くのプログラマにとってループ構文は動作を実感し易く書き易いとみなされている．

ループ構文の構文的な分かり易さは，残念ながら，反復処理の実装の正しさを保証するのに十分であるとは言えない．ループの開始条件や終了条件の不備による境界値バグ（off-by-one エラー）や不適切なガードによる無限ループ，反復の初期条件の設定ミスなどが生じ易い．これらは単純ミスと片付けられがちだが，無限ループの存在による Microsoft Zune 30GB の一斉フリーズバグ (2008)，OpenSSH における off-by-one エラー (2002) など，少なくない実害が生じたインシデントの原因である．

ループ記述に起因するバグを防ぐには，ループの反復がどのような状態から始まってどのような状態で終わるのか，またその全体を通して成立し続ける不変条件は何かという，ともするとあやふやになりがちな事項を明確に把握する必要がある．理想的には，これらを形式的に記述し，検証することが望まれる．しかし，これは一般のプログラマにとって容易とは言い難い．例えば，図 1 に示すのは，総和が最大となる配列セグメントを探すアルゴリズムをプログラミング言語 Dafny を用いてループで実装した例 [5] であるが，ループ本体のシンプルな命令列に対し，多くのループ不変条件の明記が必要になっている．こうした記述による形式検証は，Amazon や Microsoft などで暗号系や分散システムの設計・実装において使用されているが，検証にかかるコストは少なくなく，あらゆるプログラムの記述において形式検証が用いられている状況には至っていない．

我々は，ホーア論理に基づく検証機能を持つプログ

A Study on Loop Implementation Support in the Verification-Oriented Language Dafny Enabling “Quick Writing and Testing”

Asato Yoshida, Hideyuki Kawabata, 広島市立大学, Hiroshima City University.

```
// find the index range [k..m) that gives
// the largest sum of any index range in a
method MaxSegSum(a: array<int>)
  returns (k: int, m: int)
  ensures 0 <= k <= m <= a.Length
  ensures forall p,q ::
    0 <= p <= q <= a.Length
      ==> Sum(a, p, q) <= Sum(a, k, m)
{
  k, m := 0, 0;
  var s, n, c, t := 0, 0, 0, 0;
  while n < a.Length
    invariant 0 <= k <= m <= n <= a.Length
              && s == Sum(a, k, m)
    invariant forall p,q ::
                0 <= p <= q <= n
                  ==> Sum(a, p, q) <= s
    invariant 0 <= c <= n
              && t == Sum(a, c, n)
    invariant forall b ::
                0 <= b <= n
                  ==> Sum(a, b, n) <= t
  {
    t, n := t + a[n], n + 1;
    if t < 0 {
      c, t := n, 0;
    } else if s < t {
      k, m, s := c, n, t;
    }
  }
}

// sum of the elements
// in the index range [m..n)
function Sum(a: array<int>,
             m: int, n: int): int
  requires 0 <= m <= n <= a.Length
  reads a
{
  if m == n then 0
  else Sum(a, m, n - 1) + a[n - 1]
}
```

図 1　総和が最大となる配列セグメントを探すループ記述を持つ Dafny プログラム．不変条件の記述量はループボディのそれよりも多い（[5] より）

ラミング言語 Dafny において，多くのプログラマが慣れ親しんできたはずのループ構文を，慣れ親しむ過程で行ったような「ちょっと書いてみて試す」のに近い方法で記述できるようにする方法を検討している．各部の正しさをインクリメンタルに確認しながら拡張する方法で実装を進められれば，多くのプログラマにとって，形式検証付きのループ実装のハードルを下げることにつながると期待できる．

プログラミング言語 Dafny は VS Code プラグインによる検証補助機能が充実しており，プログラマがあたかも処理系と対話しながら実装と検証を進められる環境を活用することができる．我々は，VS Code プラグインに次のような拡張機能を組み込んだプロトタイプ TAMESU (**T**ail-recursive **A**ssertion and **M**ethod **E**ngine for **S**olving **U**nverified-loops) を開発中である：

1. メソッド中の書きかけの（不変条件を明記していない）ループを，同等の機能を実現する再帰関数に変換する機能
2. 1. で得た関数を用いて，ループ実行後に成立することが期待されるアサーションを検証する機能
3. 1. の変換対象のループを CHC（制約付きホーン節）に変換し，Spacer（SMT ソルバ Z3 に含まれるループ不変条件生成器）を用いてループ不変条件を生成する機能
4. 上記を総合してユーザプログラムを書き換える機能

本稿では，開発中のループ記述支援ツールについて述べ，その有効性について議論する．

2　準備：Dafny におけるループの検証

2.1　プログラミング言語 Dafny

Dafny はプログラム検証のための機能を備えた言語である．Dafny は手続き的なプログラミング (imperative programming) と関数プログラミング (functional programming) の両方のスタイルをサポートする機能を持っており，C#，Java，Go，Python 等のコードを生成できる実践的プログラミング言語である．同時に，プログラムの仕様記述のための構文や，プログラム検証や証明記述のための構文を備えている．

図 1（1 節）は，ループを用いてアルゴリズムを手続き的に実装したメソッド（`method`）と，「配列セグメントの長さ」を数学的に定義する再帰関数（`function`）を含んだ例である．図 1 中，`requires` 節および `ensures` 節は，関数やメソッドの呼び出しにかかる事前条件および事後条件をそれぞれ明記す

```
method SumByLooping(n : nat) returns (s: nat)
    ensures s == n * (n + 1) / 2
{
    s := 0;
    var i := 0;
    while i <= n
        invariant 0 <= i <= n + 1
        invariant s + i == i * (i + 1) / 2
    {
        s := s + i;
        i := i + 1;
    }
    assert s == n * (n + 1) / 2;
}
```

図 2 $\sum_{i=0}^{n} i$ を計算するループ

るもので，ループ中の `invariant` 節はループ不変条件を記述している．制御構文は条件分岐（`if` 文や `if` 式）やループ構文（`while`）など一般的なものを備えている．メソッドが複数の値を返すことができたり，複数の変数への並列代入が記述できるなどの特徴がある．

Dafny は，VS Code 等の IDE との間で LSP を介した充実した連携機能を提供しており，プログラマの記述する内容に従って即座に検証結果が確認できる．

2.2 Dafny によるループの検証

Dafny では，ループで計算される内容の検証のために，ループ不変条件の明記が必要である．例えば図 2 では，`invariant` 節を用いて次のループ不変条件を明記している．

$$0 \leq i \leq n+1 \ \wedge \ s + i = \frac{i \cdot (i+1)}{2}$$

これとループのガードの否定 $\neg(i \leq n)$ との論理積をとったものを Dafny ではループの仕様と呼ぶ．

ループ不変条件は，プログラムにおけるループに入る直前の地点と，ループボディの先頭および末尾，およびループから抜け出た地点で成立することがループの検証のための証明義務となる．それらを Dafny の検証器が自動的に判断できない場合はプログラマが間を埋める記述（補題の導入や式変形など）を加える必要がある．

Dafny では，簡素なループであっても，不変条件に基づく検証がなされない限りはループ中で行われる処理についての検証は行えない．例えば図 3 のループ直後の `assert` 節の記述は検証をパスしない．図 4 のように，ガードや不変条件で言及されていなければ，ループを抜けた後の値を用いた検証はできない．図 4 のループは不変条件を単純な式で表現できるので大きな困難なく記述可能かもしれないが，形式検証に不慣れなプログラマにとっては適切な不変条件を明記しなければならないことが大きなハードルになる可能性はあると考えられる．

```
method Triple(n: nat) returns (r: nat) {
  var i: nat := 0;
  var s: nat := 0;
  while i < n
  {
    s := s + 3;
    i := i + 1;
  }
  assert s == i * 3; // 検証不能
  r := s;
}
```

図 3 引数の値の 3 倍を計算するループ（未完成）

なお，図 4 のループは単純であるので，ループを制約付きホーン節 (CHC) に変換することにより，Spacer などのツールを用いて不変条件を生成できる．Spacer は SMT ソルバである Z3 に組み込まれており，CHC およびループ実行後に満たすべき条件を入力すると（簡便なループであれば）ループ不変条件を生成する．実際，図 3 のループに対し，次の整数変数の不等式が不変条件として得られる．

```
(((3 * i) + (-1 * s)) < 1)
&& (((3 * i) + (-1 * s)) > -1)
```

これは `s == i * 3` と同値である．

3 提案：ループ記述のためのユーザ支援

3.1 ツールに求められるものの検討

図 3 のループは非常に単純であるが，適切なループ不変条件を与えられない限り，ループ内での処理に関する検証は行えない †1．例えば，ループボディ

†1 仮に図 3 のループ直後の `assert` 節を省いたとすると，Dafny の検証器はエラーを検出しないが，ループ通過後の変数 `s` および `i` の相互関係を静的に論じることはできない．

```
method Triple(n: nat) returns (r: nat) {
  var i: nat := 0;
  var s: nat := 0;
  while i < n
    invariant s == i * 3
  {
    s := s + 3;
    i := i + 1;
  }
  assert s == i * 3; // OK
  r := s;
}
```

図 4　引数の値の 3 倍を計算するループ（完成版）

```
function Triple_rec(n: nat,
                    i: nat, s: nat): (nat, nat)
  requires i <= n
  decreases n - i
{
  if i < n then
    Triple_rec(n, (i + 1), (s + 3))
  else
    (i, s)
}
```

図 5　引数の値の 3 倍を計算するループ（関数化）

```
method Triple_test1()
{
  var n' := 10; // 参照される変数の初期化
  var i' := 0; // 参照される変数の初期化
  var s' := 0; // 参照される変数の初期化
  assert
    var (i, s) := Triple_rec(n', i', s');
    s == i * 3; // 確認したかった論理式
}
```

図 6　ループと同じ振る舞いをする関数を用いた検証

が実行されなかったときに `s == i` が成り立つかどうか，ガード `i < n` が正しいかどうか，ボディの `s := s + 3;` が妥当かどうかなど，簡単なことさえ静的な確認ができない．

これに対し，図 3 のループを図 5 のように再帰関数で表現し直せば，確認したかった論理式の真偽を `assert` 文を用いて検査することができる．図 6 では，図 5 の関数をループの代わりに用いて，ループ中で参照される変数 `n`, `i`，`s` の初期値がそれぞれ 10，0，0 であったときにループ通過後に `s == i * 3` が成り立つかどうかを確認する様子を示している．

```
method Triple_testn(n: nat) returns (r: nat)
{
  var i: nat := 0;
  var s: nat := 0;
  assert
    var (i', s') := Triple_rec(n, i, s);
    s' == i' * 3; // これは検証できない
  r := s;
}
```

図 7　関数化したものを用いた検証

```
lemma Triple_rec_Correct(n: nat, i: nat, s: nat)
  requires i <= n
  requires s == i * 3
  ensures
    var (i', s') := TripleLoop_rec2(n, i, s);
    s' == i' * 3
  decreases n - i
{
  if i < n {
    Triple_rec_Correct(n, i + 1, s + 3);
  }
}

method Triple_testn(n: nat) returns (r: nat)
{
  var i: nat := 0;
  var s: nat := 0;
  Triple_rec_Correct(n, i, s); // これが必要
  assert
    var (i', s') := Triple_rec(n, i, s);
    s' == i' * 3;
  r := s;
}
```

図 8　関数化したものを用いた検証 (2)

本研究では，このように，ループ中で用いる変数の値をいくつか固定したときにループ実行後に特定の条件が成立するかどうかを手軽に確認できる仕組みの構築を目指す．

本来ならば，図 7 のように，ループ中で参照される変数が取り得るあらゆる値（ここでは任意の自然数 `n`）に対しての検証を静的にできるとよいが，関数 `Triple_rec` の引数 `n` が定数でない場合には再帰関数を展開することができない．これに対処するには，図 8 のように，関数 `Triple_rec` の性質を明示的に論じた補題を用いる必要がある．つまり，ループ不変条件と同等な情報をプログラマが明記する必要がある．

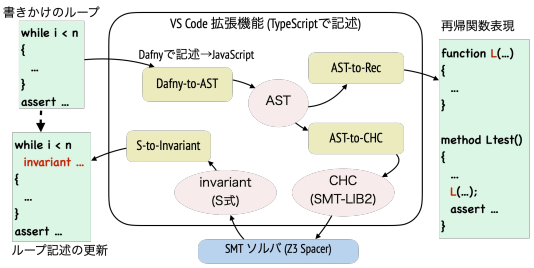

図 9　TAMESU の概略図

従って，(Spacer のような外部ツールを用いた自動生成に頼る以外の方法では) 最終的にはループ不変条件を明記せざるを得ない．

本研究で試作するシステムでは，前述の，いくつかの変数を固定したときに特定の条件が成立することを手軽に確認できる機能に加えて，ループ構文を CHC に変換して Spacer を用いて不変条件の自動生成を試みる機能も導入し，プログラマへの利便性を図る．

3.2　TAMESU の概要

提案ツールでは，不変条件を記述しきれていない書きかけのループ記述に対し，(1) 再帰関数への変換に基づく簡易的な検証機能，および，(2) 既存のツール (Spacer) を用いた不変条件自動生成機能を提供する．

ユーザは，VS Code で記述中のプログラムについて，`while` ループとそのループの実行後に成り立つことを期待する命題 (`assert` 文) を記述する．提案ツールは，ユーザからの指示に従い，ループの再帰関数への変換と，その関数を用いた簡易的な検証用メソッド生成を行う．これらは VS Code 上の別ペインで表示され，元のループを眺めながら，変数に具体的な値を当てはめてターゲットの論理式の成立・不成立を確認できる．

CHC 経由で Spacer によって不変条件が推論できる場合には，提案ツールは `while` ループの冒頭に `invariant` 節として推論結果を自動的に挿入する．

3.3　システム構成および制御フロー

提案機能を実現する VS Code 拡張機能 TAMESU のシステム構成の概略を図 9 に示す．図 9 中，Dafny パーサ (Dafny-to-AST) は Dafny で記述し JavaScript に変換して使用する．拡張機能の他の部分は TypeScript で記述している．

CHC は SMT-LIB2 形式で生成し，外部プロセスの Z3 へ渡し，15 秒のタイムアウト付きで検証・推論を実行する．不変条件は S 式で得られる．これを Dafny 記述に変換し，元のループ構文中に自動挿入する．

3.4　実装方式

純粋関数としてのパーサ実装　トランスレータは外部ライブラリに依存せず，Dafny の純粋な関数を用いてパーサコンビネータを独自実装している．データ型 `Result<T>` を利用してパースの成功・失敗を伝播させ，数式や論理式，文，宣言を再帰的に解析する．

タプルを利用した複数状態変数の末尾再帰化　ループ内で更新される変数が複数存在する場合 (例えばカウンタ `i` とアキュムレータ `s` など)，システムは AST から更新対象の変数を特定する．特定した複数の変数を一つのタプル型 (例: `(nat, nat)`) としてまとめることで，副作用のない純粋な末尾再帰関数への変換を実現している．

CHC へのエンコードと S 式のデコード　ループの初期状態 (Initial state)，遷移 (Transition)，およびユーザの指定する `assert` 文に基づく安全性 (Safety/Goal) の 3 つのホーン節を構築する．Z3 が生成した S 式は，TypeScript 側に実装したトランスレータによって，元の変数名を復元しながら Dafny の記法へと変換される．

3.5　実行例

図 3 のコードに対し，図 5 および図 6 が自動生成され，分割表示ペイン内に表示される．ここでループの振る舞いを確認できる．

またこの例であれば，図 3 のループ中に以下の `invariant` 節が自動的に挿入される (`s==3*i` と等価)．

```
invariant (3 * i - s < 1) && ↩
(3 * i - s > -1)
```

提案ツールを用いれば，プログラマは，ループ実行後に成り立つことが期待される命題を `assert` 節で記述するだけで，ループ不変条件を完成させる前にルー

```
method SquareRoot(N: nat) returns (r: nat)
    ensures r * r <= N < (r + 1) * (r + 1)
{
    r := 0;
    var s := 1;
    while s <= N
        invariant r * r <= N
        invariant s == (r + 1) * (r + 1)
    {
        s := s + 2*r + 3;
        r := r + 1;
    }
}
```

図 10　自然数平方根

プの処理内容を静的に確認することができるようになる．また，コードの書き換え（不変条件の挿入）と検証用コードの提示（再帰関数）を UI 上で明確に分離することにより，プログラマの思考を妨げないインクリメンタルな開発体験（「ちょっと書いて試す」アプローチ）を実現できる．

一方で，乗算を含むような非線形算術のループ（例：べき乗計算など）においては，Z3 Spacer による不変条件の推論が 15 秒のタイムアウト内に完了しない，あるいは反例あり（unsat）として適切に公式化できないケースが確認されている．

4　議論

4.1　テストによる検証と形式的証明の間を埋める

本システムは，ループを再帰関数で表現することにより，例えば `n = 10` のような具体的な値を代入して部分的に実行・検証を行う「具体値による単体テスト」の機能を提供するものといえる．完全な形式証明が困難な複雑なループであっても，テストによる検証へとシームレスに移行（フォールバック）可能な，実用性の高い支援環境の実現につながり得ると思われる．

4.2　ループ不変条件主導の実装

ループの実装は，仕様に基づいて導出することが自然で効率的な場合もある．例えば，自然数の平方根を計算するループ（図 10）や n 番目のフィボナッチ数の自乗を計算するループ（図 11）が挙げられる．これらは，プログラマがループボディの記述を直感に

```
function Fib(i: nat): nat
{
    if i == 0 then 0
    else if i == 1 then 1
    else Fib(i - 1) + Fib(i - 2)
}

method FibSquared(n: nat) returns (x: nat)
    ensures x == Fib(n) * Fib(n)
{
    x := 0;
    var i, y, k := 0, 1, 0;
    while i != n
        invariant 0 <= i <= n
        invariant x == Fib(i) * Fib(i)
        invariant y == Fib(i + 1) * Fib(i + 1)
        invariant k == 2 * Fib(i) * Fib(i + 1)
    {
        x, y, k := y, x + k + y, k + y + y;
        i := i + 1;
    }
}
```

図 11　フィボナッチ数の自乗

頼って導いたり，ループボディの記述を見ただけで反復処理の内容を汲み取ったりすることは難しいと思われる．我々のツールはこのようなループを記述するための直接的なサポートはできないかもしれないが，適切なインターフェースを用意すれば，紙と鉛筆の代わりに試行錯誤に使える可能性はある．

5　関連研究

プログラム検証において，ループ不変条件の記述は開発者にとって最大の障壁の一つと言われる．この負担を軽減するため，不変条件の自動生成に関する研究が長年行われてきた．例えば，Houdini [2] に代表されるあらかじめ用意された雛形に基づくテンプレートベースの手法や，近年ではプログラムの検証条件を制約付きホーン節（CHC）へと変換し，SMT ソルバを用いて解くアプローチ [3] [1] が広く用いられている．しかし，非線形算術や複雑なヒープ操作を含むループに対しては，最先端の自動推論ソルバであってもタイムアウトや推論失敗に陥ることが多いのが現状である．本研究は，Spacer による自動生成を組み込みつつ，推論が失敗した場合でも開発者が手動で検証を進められるフォールバック手段を提供している点で，完

全自動化を目指す既存ツールとは異なる実践的なアプローチをとっているといえる．

命令型プログラムのループ構造を等価な末尾再帰関数へ変換する手法自体は，プログラム意味論や形式検証ツールの内部処理（バックエンドにおける検証条件生成など）において標準的な手法として知られている．しかし，既存の検証系の多くはこれを「証明器向けの内部表現」として不可視に扱っているのに対し，本研究の TAMESU は，再帰化された純粋関数を「開発者が VS Code 上で具体値を与えて単体テストするためのインタフェース」として直接的に UI 上へ提示している点が特徴的である．これにより，完全な不変条件を書き上げる前段階でのインクリメンタルな動作確認が可能となっている．

本研究が目指す「少し書いて動かしながら整える」という対話的な実装過程は，ソフトウェア工学における継続的検証（Live Verification） [4] の思想に深く根ざしている．Dafny 自体が，IDE とバックグラウンド検証器を密結合させることでこの分野を牽引してきたが，未完成のループ記述に対しては静的解析の恩恵を受けにくいという課題があった．提案手法は，書きかけのループの検証を純粋関数の評価へと変換することで，Dafny の Live Verification の恩恵を「ループ仕様の試行錯誤の段階」にまで拡張するものと位置づけられる．

6 まとめ

形式検証向けプログラミング言語 Dafny において，「少し書いて動かしながら整える」というインクリメンタルな開発に近い体験ができるツール TAMESU を設計した．CHC への変換と Spacer による解析をバックグラウンドで行う機能も用意した．様々なコード例を用いた定性的・定量的な評価，および，対応するデータ型の拡充等が今後の課題である．

参 考 文 献

[1] Bjørner, N., Gurfinkel, A., McMillan, K., and Rybalchenko, A.: *Horn Clause Solvers for Program Verification*, Springer International Publishing, 2015, pp. 24–51.

[2] Flanagan, C. and Leino, K. R. M.: Houdini, an Annotation Assistant for ESC/Java, *Formal Methods Europe (FME)*, Springer, 2001, pp. 500–517.

[3] Komuravelli, A., Gurfinkel, A., and Chaki, S.: SMT-Based Model Checking for Recursive Programs, *Computer Aided Verification (CAV)*, Springer, 2014, pp. 17–34.

[4] Leino, K. R. M. and Wüstholz, V.: The Dafny Integrated Development Environment, *Formal Methods in Computer-Aided Design (FMCAD)*, IEEE, 2014, pp. 9–15.

[5] M. Leino, K. R.: Accessible Software Verification with Dafny , *IEEE Software*, Vol. 34, No. 06(2017), pp. 94–97.