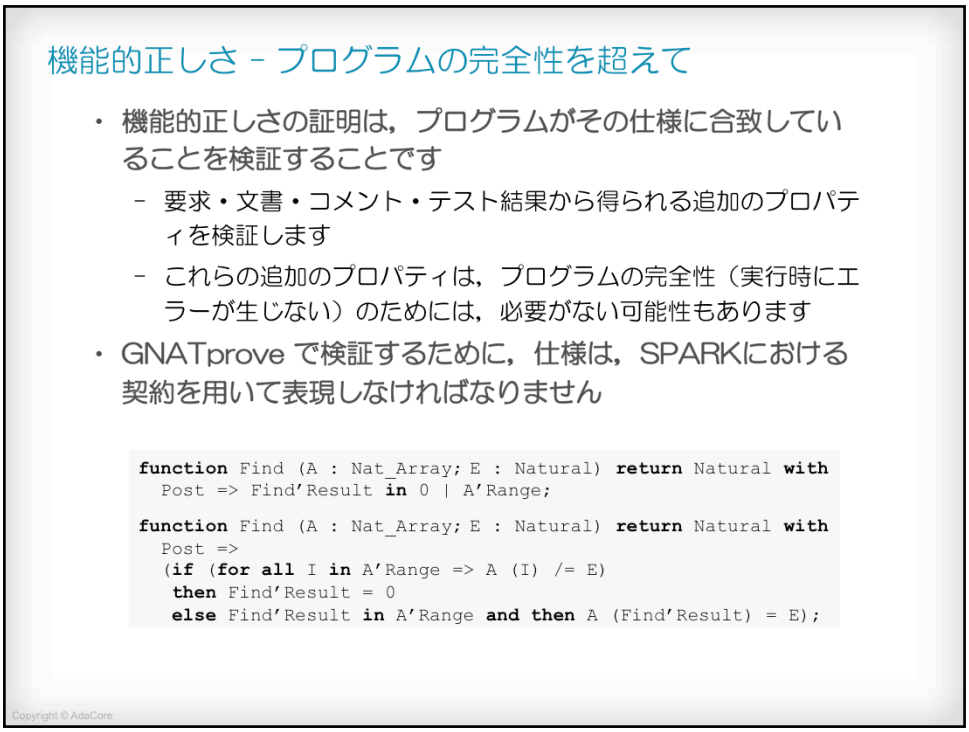
この講義は，プログラムの機能的正確さのためのものです．

複雑なプログラムプロパティを記述し，検証するために必要となる進んだ証明に関する特徴について考え，結果を示します．



プログラムの正しさは，プログラムがその仕様に合致しているということです．機能的正しさにおいて，サブプログラムの入力と出力の関係に，特に関心を持っています．時間やメモリ消費は含まれません．

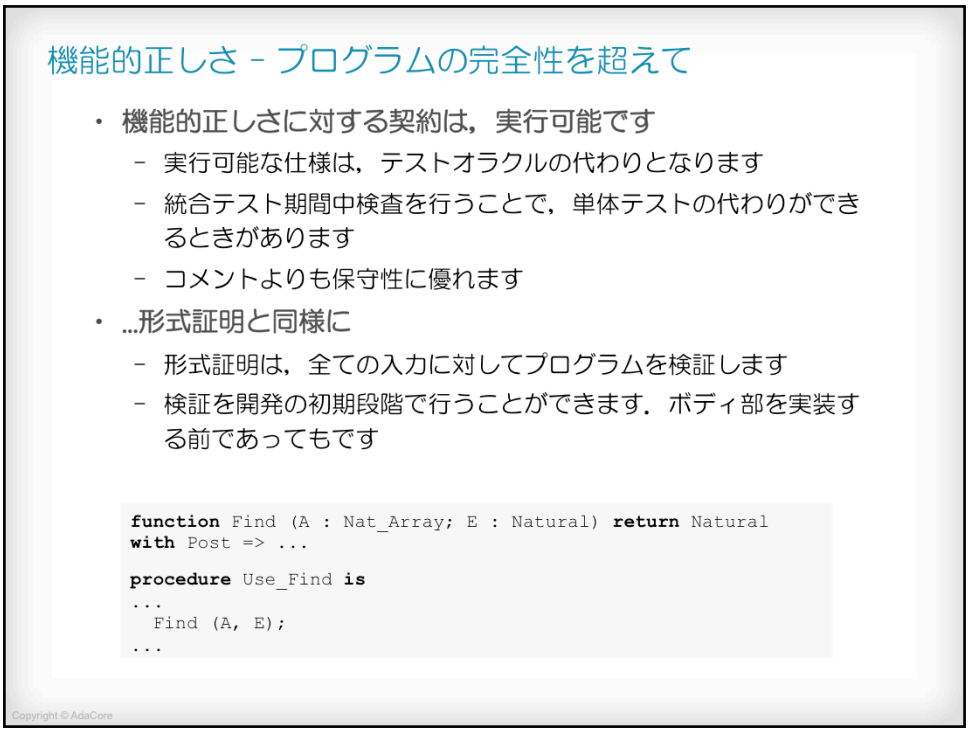
プログラムに示されたプロパティは，プログラムの完全性のために必要とされるより，一層強い記述になっています．

（何らかの規格の）認証プロセスにおいて，システムの仕様から，プロパティを導出します．或いは，より非形式的な場所から情報を得るかもしれません：プログラムの文書，コード中のコメントあるいは，テストオラクル（例えばテストケース）です．

例えば，関数 Find の結果を利用するときに，実行時エラーが送出されないことを保証することは，次のことが分かれば十分です．ゼロではない場合，つねに A の範囲にある．

しかし，プログラムが意味あるものであるためには，Find に関してより複雑なプロパティを検証したくなる場合があります．例えば， E が A 中には存在しない場合にのみゼロを返し，存在する場合は，インデクスを返すといったことです．

Find に対してそうしたように，もし，これらの仕様がサブプログラムまたはパッケージに対する契約として表現されれば，GNATprove は，厳密にそれらを検証することができます．



契約の形式を用いて，プログラムの仕様を部分的にでも記述することにはメリットがあります．

第一に，これら契約をテスト期間中に実行することができます．このことで，早い段階で，プログラムと仕様の差異を検出できるため，コードの保守性が向上します．

十分に正確であれば，契約はテストオラクルとして，テストに合格・不合格かを決めるのに用いることができます．

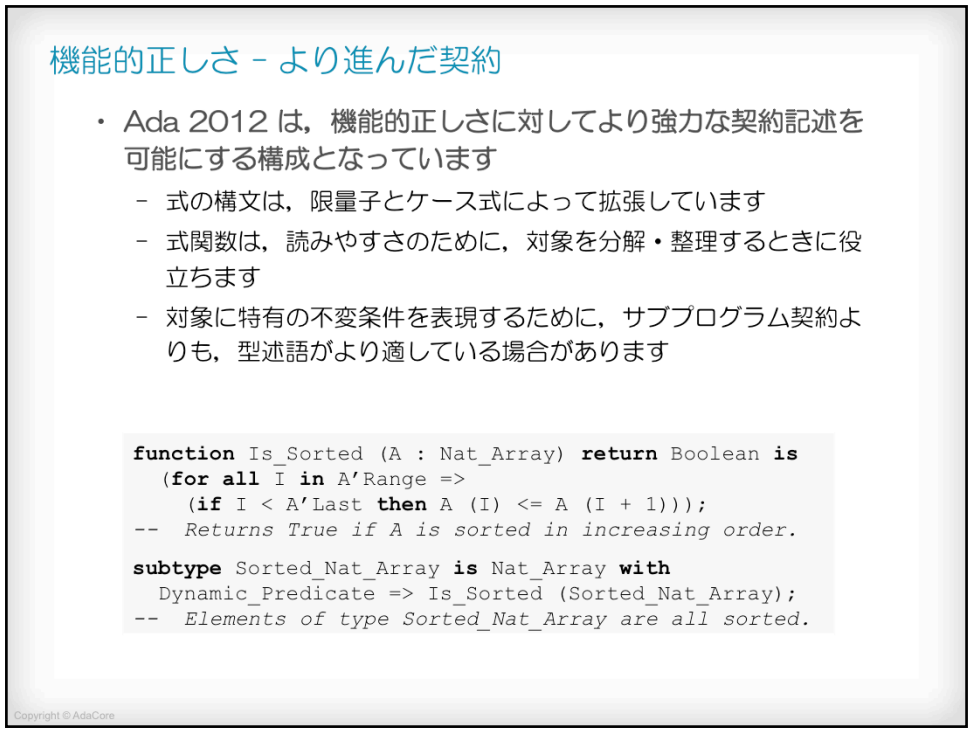
このように，完全なコードを実行しているあいだ，特定のユニットの出力を検証するのに役立ちます．

あるコンテキストにおいては，このことは，表明を動作可能にして，統合テストを実行することにより単体テストを代替できるかもしれません．

次に，コードが SPARK で記述されていれば，これら契約は，GNATprove によって，形式的に検証することができます．

形式検証は，全ての可能な実行を検証するという利点を持っています．これは，動的検証では通常不可能なことになります．

例えば，テスト中，Find の事後条件は，Find が呼ばれたときの全ての入力の組に対して，動的に検査をします．しかし，Find が形式的に検証されていれば，その事後条件は，全ての可能な入力に関してチェックされたことになります．静的検証は，開発期間中，テストに先立ち実施することもできます．静的検証は，サブプログラム単位で細かく動作するからです．例えば，Use_Find は，Find のボディ部を記述する前ですら，形式検証することができます．



機能的正しさに対する契約は，プログラムの完全さに対する契約以上のことを意味しています．Ada 2012 で導入された新しい形式の表現を，利用する必要がある場合があります．

特に，限量子付き表現（範囲中の「全て」，或いは「少なくともーつ」といったプロパティを記述するための表現）が，配列に対するプロパティを記述するときに手軽に使えるようになりました．

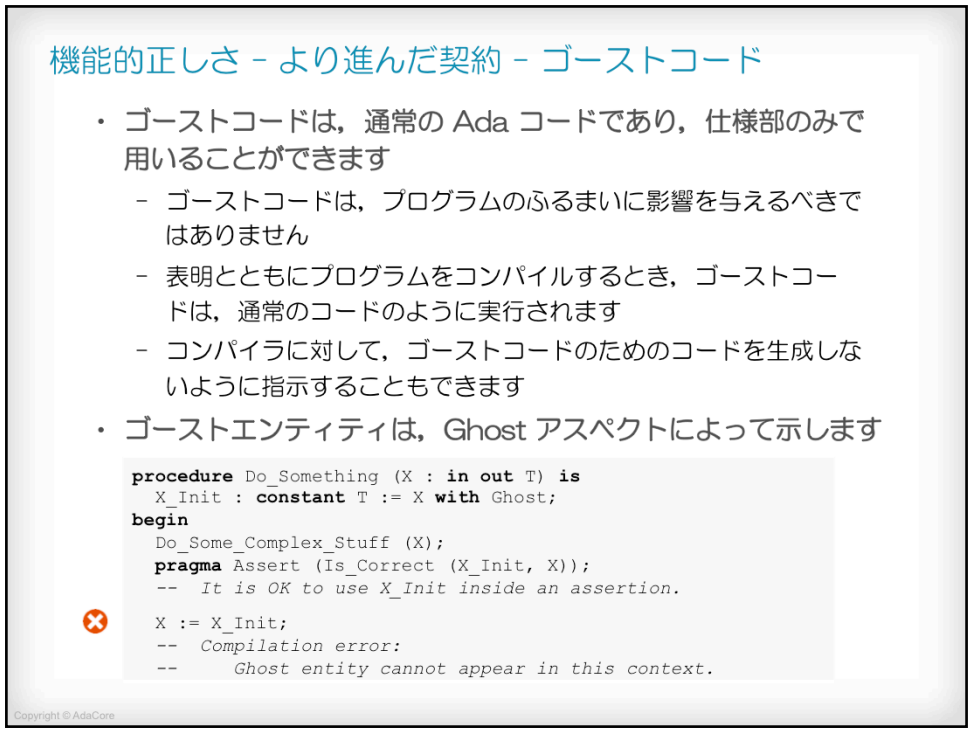
契約は複雑になったので，その可読性を向上させるために，新しい抽象を導入することは有益です．式関数は，この目的のためには良い手段です．式関数の本体は，パッケージの仕様部にあるからです．

さいごになります．サブプログラムのプロパティというより，データに関する不変条件である幾つかのプロパティを，サブプログラムの契約として表現するのは煩わしいかもしれません．

ある型の全てのオブジェクトが持つべき型述語（Type predicate）が，この目的にはより合っています．

例に示すように，副型 Sorted_Nat_Array は，配列変数に型づけられており，プログラムの中では，整列されている状態であるべきです．配列が整列していると記述することは，より複雑な限量子を用いた，より複雑な表現を必要とします．従って，我々の例題では可読性を向上させるために，式関数として表現し，このプロパティを取り除いています．Is_Sorted の本体は，パッケージの仕様部にあります．これにより，パッケージの利用者が必要とするときに，正しい意味についての正確な知識を得ることを可能にしています．

（訳注）副型（Subtype）元となった型が取り得る値に制約を加えた型．型に対する操作は，元になった型から継承する



我々が記述しようとするプロパティがより複雑になるにつれ，仕様化の目的のためだけに用いるエンティティに対して，ある必要性が生じます．

資格付与のためのプロセス中で，これらの新しいプロパティが，プログラムのふるまいに影響を与えない，さらに云えば最終的なコードからは取り除ける，というのがこの必要性です．

ゴーストコードと呼ばれる概念は，SPARK 2014 で Ghost アスペクトとして上記の必要性をサポートしています．

これにより，あらゆる通常のエンティティ（変数，型，サブプログラム，パッケージ…）に対して注釈をつけることが可能です．

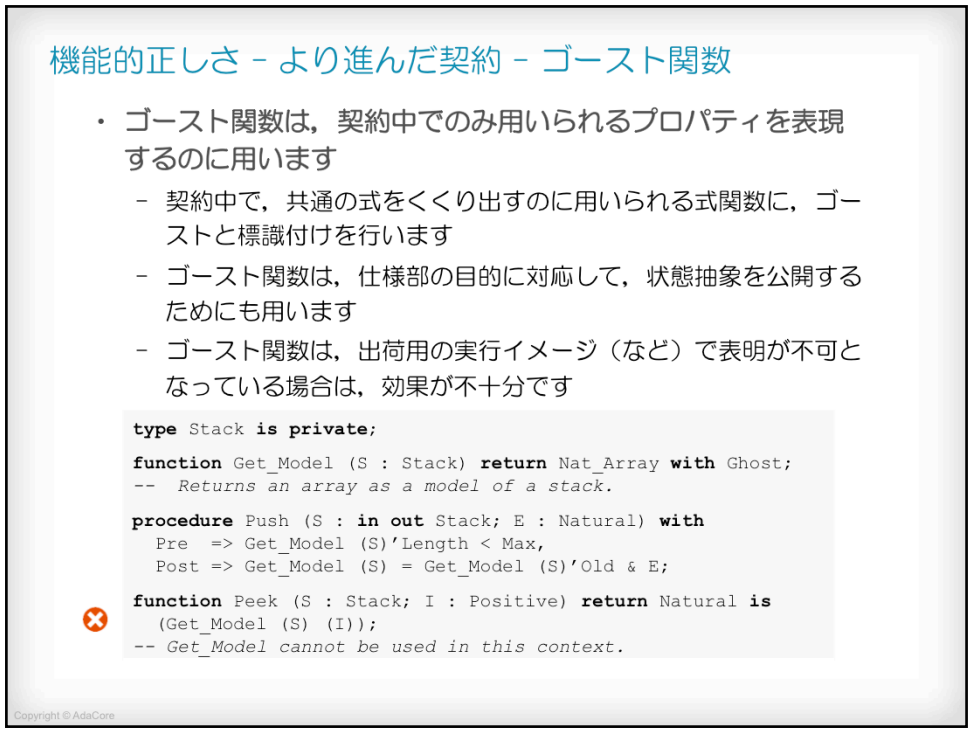
もし，エンティティが Ghost と標識付けされていた場合，GNATprove は，そのエンティティがプログラムのふるまいに影響を与えないことを保証します．

動的に契約をテストできるようにするために，表明を動作可能にしてコンパイルした場合，通常のコードのように，ゴーストコードを実行します．コンパイラが，ゴーストエンティティに対してコードを生成しないように指示することも可能です．

この例にあるように，Do_Something サブプログラムは，X の初期値をゴース定数である X_Init に保持します．Do_Some_Complex_Stuff を呼び出し演算を行うことで，X の値が期待通り修正されたかを確認するために，表明において，この変数を参照することができます．

しかし，X_Init は，例えば，X の初期値を保存するといった目的で，通常のコードで用いるべきではありません．

ここでの例では，X_Init の（通常コードでの）利用により，コンパイラは不正であると示します．より複雑なゴーストコードと通常コード間の干渉のケースは，GNATprove を動作させることでのみ，検知することができます．



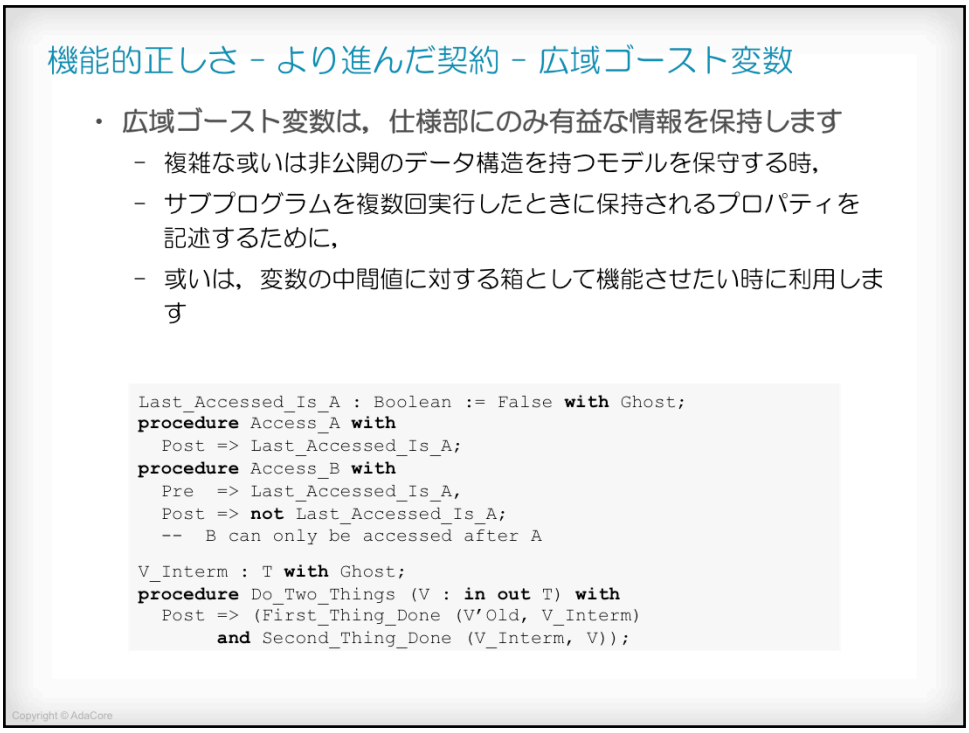
機能的正しさのための契約を記述するときに，仕様部のみで利用可能な関数を記述することは一般的です．

例えば，単純化や共通パターンをくくり出すために用いられる契約中の式関数は，通常 Ghost とマークします．

しかし，ゴースト関数は，読みやすさを向上させるだけではありません．実際のプログラムにおいて，抽象のために，関数仕様に必要な情報が，パッケージ仕様部内でアクセスできないということは良くあります．

この情報をパッケージのユーザに参照可能とすることは，一般的に抽象化原則に違反することは明らかです．ゴースト関数は，通常のクライアントコードでは利用できない情報にアクセスすることを可能とするための扱いやすい方法です．

ここでの例題では，Stack 型は，非公開です．手続き Push の期待されるふるまいを指定できるように，この抽象と S 中に保持されている要素の値へのアクセスを開示する必要があります．このため，Get_Model 関数を導入します．スタックのモデルとしての配列を返す関数です．ただし，通常のStack パッケージにおけるユーザコードにおいて，通常のコードによって，関数 Peek がそうであるように，スタックの抽象を壊すことはしたくないと考えます．関数を Ghost とすることで，問題なくゴールを達成することができます．さらに，サブプログラム Get_Model が，最終的なコードでは決して使用されないことを保証することができます．



しばしば起きることではありませんが，仕様部が広域変数に追加の情報を持たせる必要があるときがあります．

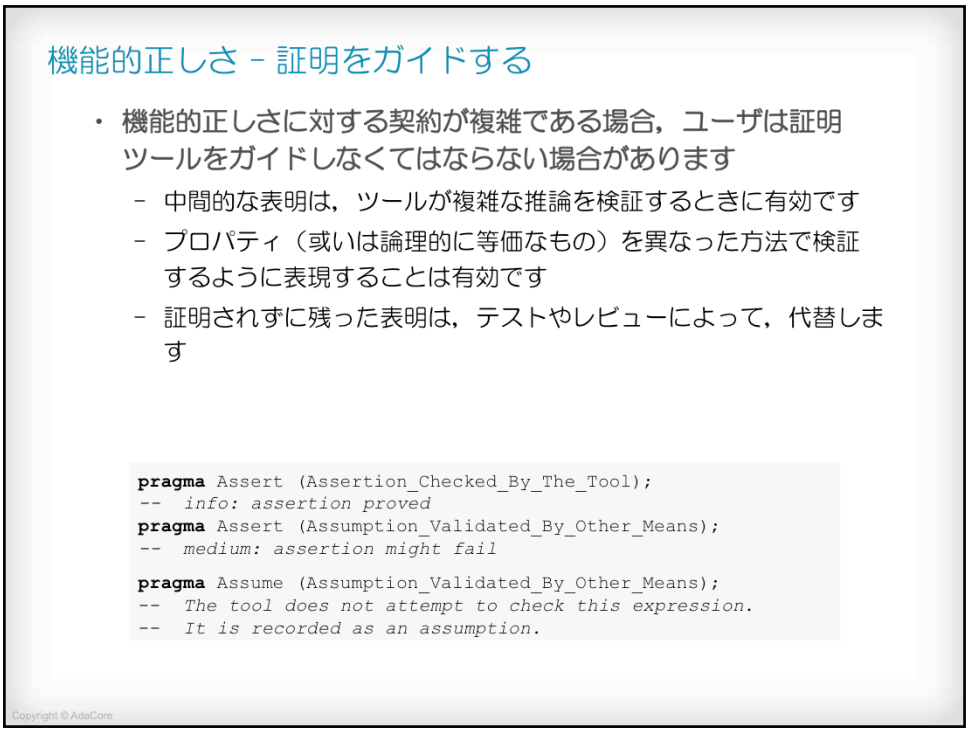
この情報は，通常のコードでは必要ではないので，この広域変数は，Ghost とマークされ，コンパイラは，この記述を取り除くこともできます．

これらの変数は，様々な理由から利用されますが，もっとも良くあるのは，モデルを提供することによって，複雑な或いは非公開の広域データ構造に対して更新を行うプログラムを記述する場合です．

単純な時制に関するプロパティを記述するために，サブプログラムの前回実行に関する情報を，広域変数に持たせることができます．例で，二つの手続きがあり，一つは状態 A にアクセスし，他方は，状態 B にアクセスします．広域変数 Last_Accessed_Is_A を，中間で A にアクセスしない限りは，2 度 B にアクセスできないことを示すために用いています．

サブプログラムの要求が，実行されるべきアクション列として，期待するふるまいを表現する例です．この種の仕様をより容易に記述するために，広域ゴースト変数を用いることができ，プログラム中の変数における中間値を保持することができます．

例えば，手続き Do_Two_Things を，広域変数 V_Interm を用いて，二つのステップで記述します．ここで，V_Interm は，二つの実行の間で，V の中間値を保持します．この記述がいつも実行できるわけではありません．Ada における限量子はループパターンに限定されているので，SPARK でいつも使えるわけではありません．更に，変数の値を追加することは，証明器が効果的に契約を検証する助けとなります．



機能的正しさに関するプロパティは，プログラムの完全性の証明以上に複雑です．従って，GNATprove が，プロパティは正しいけれど，直ちに検証できないということが起こり得ます．

デバッグの技術で証明できない場合があることは，プログラムの完全性の証明において説明しました．ここれでは簡単に振り返ります．

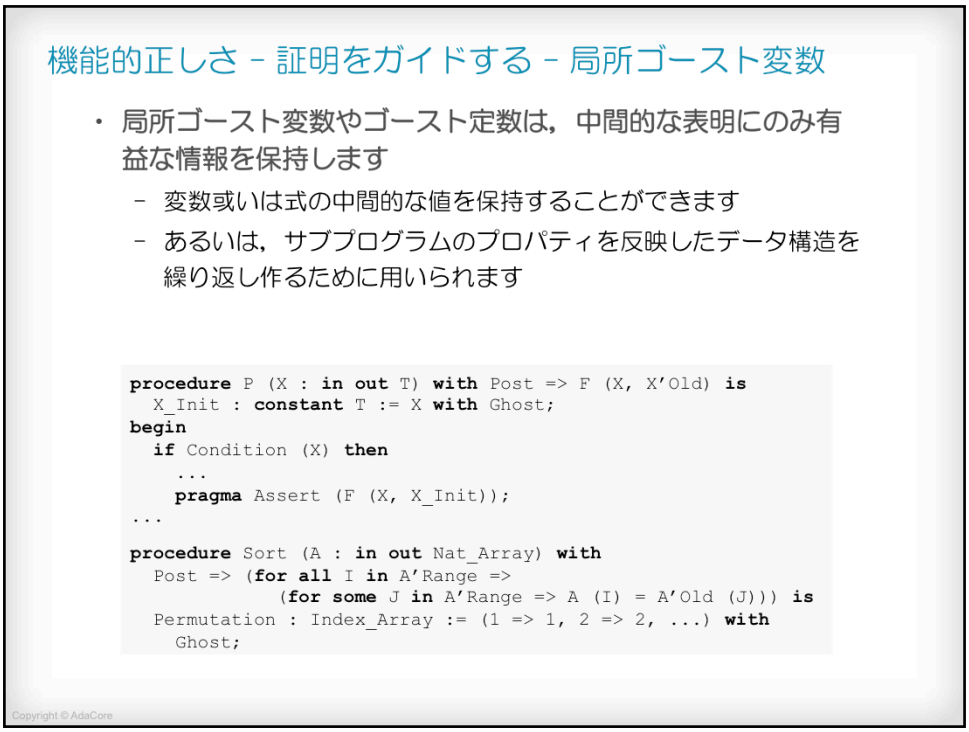
このコースでは入念に調べることはせず，合理的な時間で GNATprove が証明できない，しかし正当なプロパティが残っているケースの結果を良くすることに注目します．

ここでのケースでは，証明を完了させるため，或いは，簡単にレビュー可能な仮定に分解するために，ユーザが GNATprove をガイドする場合です．

この目的のために，複雑な証明を分解して小さなステップにするための表明を付け加えます．特に，中間的なステップとして，望んでいるプロパティと論理的に同等なバージョンを証明することは良い考えです．このとき，証明器のためにプロパティを単純化します．例えば，異なるケースを分割したり，関数の定義をインライン化します．

GNATprove が中間的表明を扱えない場合があります．情報が不足している，可能な情報の量により対処できなくなる，といった理由からです．（証明されずに）残った表明は，実行でき，レビューできるので，テストのような他の手段によって検証できます．

ユーザは，GNATprove に対して，無視するように指示することもできます．例にあるように，仮定に変更したり，プラグマ Annotate を用いて検査を正当化することによってです．この両者のケースで，表明が有効であれば，実行時にやはり検査を行います．



仕様の場合と同様に，中間的な表明内部での表現を強化するために，ゴーストコードを利用することができます．

特に，表明内で使用することを目的とした局所変数や定数は良く用いられます．

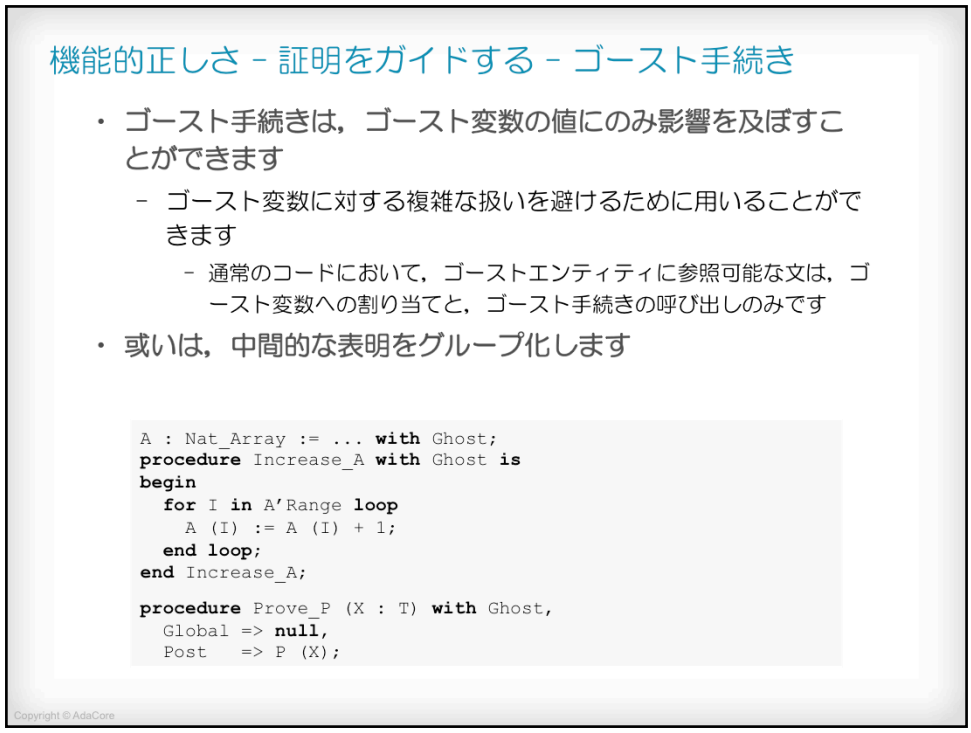
多くの場合，これらの変数は，変数或いは式の前回値を記憶するために用いられます．これは，仮定で参照するためです．

特に，サブプログラムの事後条件の外側にある 'Old 属性を用いてアクセスすることができないパラメータや式の初期値を参照するときに有益です．

例では，手続き P の事後条件を GNATprove の負荷をとりのぞくために，if 文の全ての分岐それぞれに表明を加えています．これらの表明中で，P の事後条件とは異なり，パラメータ X の初期値にアクセスするために 'Old 属性を使用することができず，この初期値をもとめるために，局所ゴースト定数 X_Init を導入し，その助けを借ります．

局所ゴース変数は，サブプログラムの複雑なプロパティを確認するためのデータ構造を作るといったより複雑なことのためにも用いることができます．例題においては，手続き Sort が新しい要素を作らないと云うこと証明したい．即ち，成立が行われた前後で，配列 A の要素に変化がないことを示したいとします．

ここでのプロパティは，整列するように呼び出されたあと，A は呼び出される前の順列と同じであるということを保証するには，十分ではないことに注意して下さい．さらに，証明器が検証を行うには複雑になっています．限量子の選言的判断が含まれているからです．また，GNATprove のために，各インデクス I に対して，期待しているプロパティを持っているインデクス J を保持しています．



ゴースト手続きは，通常の変数の値に影響を与えることができません．従って，多くの場合，ゴースト変数の処理を行うか，中間的な表明の組をグループ化することになります．ゴースト変数の扱いないしはゴースト手続き中の表明を取り除くことには，幾つかの利点があります．

第一に，表現力を高めます．コンパイラによるゴーストコードの削除を単純化するために，通常のコード中に存在し得るのは，ゴースト変数への割り当てとゴースト手続き呼び出しのみです．

次に，サブプログラムの機能的ふるまいに役立たない複雑なコードを隠すことにより，可読性を向上させます．

最後に，そうしなければ文脈が分かりにくくなる表明を除くことによって GNATprove への負荷を下げます．

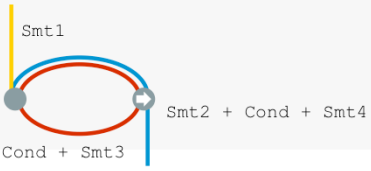
例えば，X を引数にして，手続き Prove_P の 呼び出すと，証明すべき対象として，検証に必要となる可能で重要な表明の組ではなく，P (X) を追加するだけです．更に，P の証明は一度だけ行われます．また，検証中，不必要な情報がないことにより，証明の実行が容易になります．

また，もし Prove_P が十分に検証されなくても，小さな範囲のみ考えればよいので，検証しきれなかった残りの仮定も，より容易にレビューすることができます．

機能的正しさ - 証明をガイドする - ループを扱う

- GNATprove は，ループのあるサブプログラムを幾つかの部分に分けます．制御フローを循環を含まない形にするためです
  - サブプログラムの開始からループ文まで
  - ループの最初からループの終了まで（一回文）
    - 脱出条件は成立しないと仮定する
    - ループ中で更新される初期値は不明である
  - 各脱出パス毎に，ループの開始からサブプログラムの終了まで
    - 脱出条件は成立すると仮定する
    - ループ中で更新される初期値は不明である

```
Stmt1;
loop
   Stmt2;
   exit when Cond;
   Stmt3;
end loop;
Stmt4;
```



GNATprove が証明を完了するために，ほぼ常にユーザの注釈を必要とするのは，プログラムがループを含んでいる場合です．

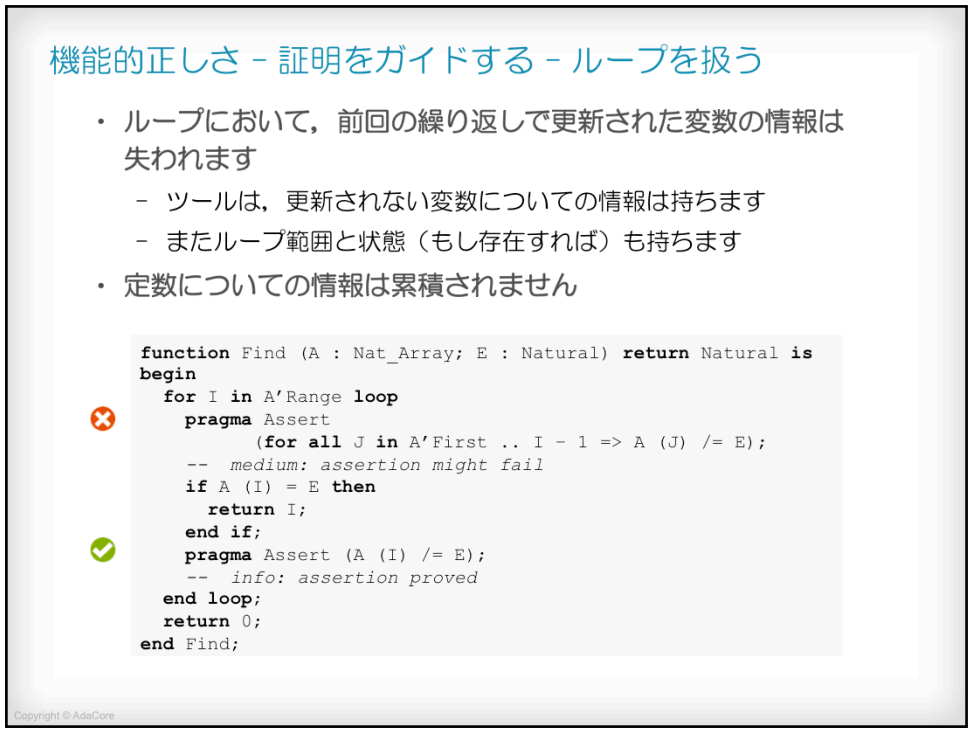
実際，GNATprove で利用している検証技術は，サブプログラムフローにおける循環を扱いません．従って，幾つかの非周期的な部分に分割することで，ループが循環を含まない形にします．

例にある単純なループを見て下さい．一つの脱出条件を持ちます．図にあるように，制御フローは三つの部分に分かれます．最初は黄色い線です．最初からループ文までです．

次に，ループ自身を 2 つに分けます．赤い線は，ループ本体の実行を示します．即ち，ループ脱出条件を満足しない実行です．

青い線は，ループの最後の実行を示しています．脱出条件が成立しており，サブプログラムの残りへアクセス可能です．赤と青で示した部分は，明らかに，黄色い部分の実行後に生じます．

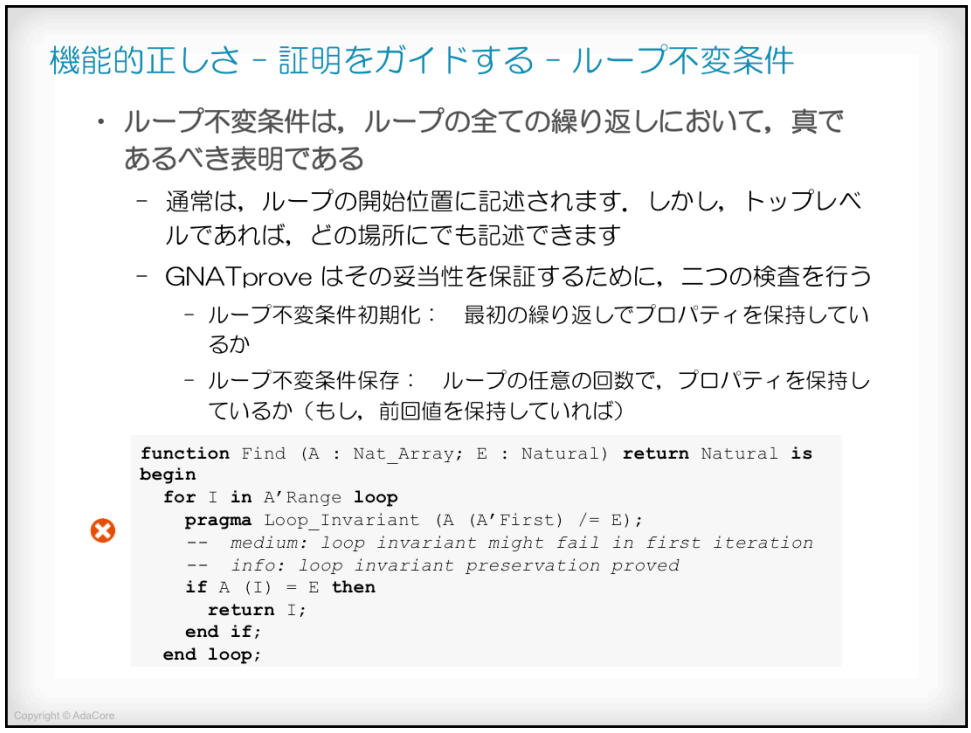
ループにおいて，アクセスした変数を如何に変更したかを知る方法がないので，GNATprove は，これら部分に入ったときに，知っていることを全てただ忘れます．ループ中で更新されない定数や変数の値は，もちろん保持されます．



（前ページで示した）この特別なループに対する扱いによって，ループを含むサブプログラムを検証する時に，GNATprove は不正確さの影響を受けます．

具体的に言えば，ループ中で更新される変数の値に依存したプロパティを検証することができないということです．また，定数の値や変更されない変数に関する情報は忘れないけれども，ループを用いて，それらに関する新しい情報を引き出すことはできません．

例えば，Find 関数では，配列 A が保持している要素 E の場所を示すインデクスを探します．各繰り返しで，ループが継続している間，インデクス I のとき 配列 A に保持されている値が E ではないことを，GNATprove は知っています．しかし，I より小さなインデクス全てに関して，それが真であることを導き出すための情報を，累積しているわけではありません．



（前ページの）限界を乗り越えるために，ループ不変条件という形式で，ツールに付加的な情報を与えることができます．

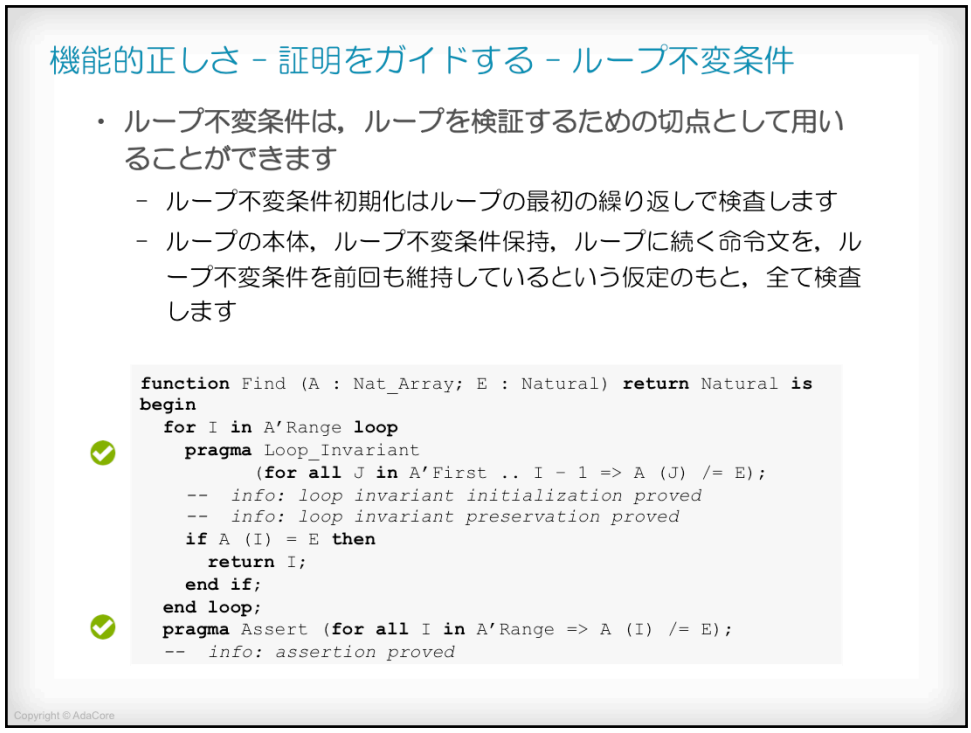
SPARK 2014 では，ループ不変条件は，ループの各繰り返しで保持すべきブール式になります．他の全ての表明と同様に，表明を実行可能にしてプログラムをコンパイルすることで実行時に検査することが可能になります．

他の表明と比較すると，ループ不変条件は，証明のための扱い方が異なっています．ループ不変条件の証明は，2段階で行われます：最初に，GNATprove は，最初の繰り返しで保持しているものを検査します．次に，任意の繰り返しで，前回保持しているものが，引き続き保持されているかを検査します．

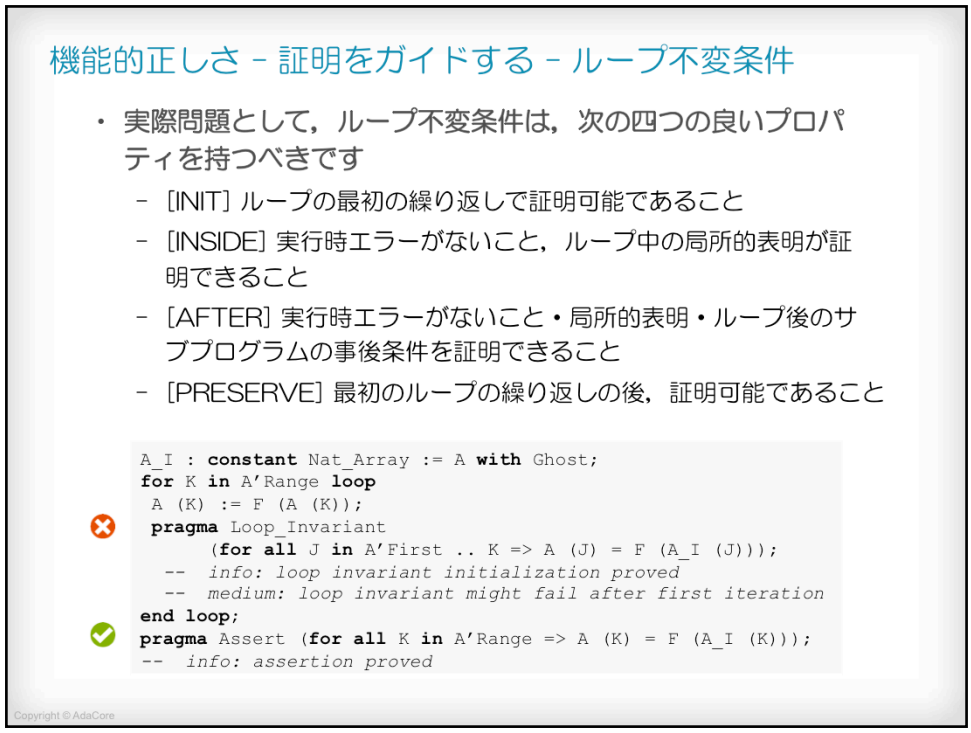
例において，Find 関数に次のループ不変条件を追加しています：「配列 A の最初の要素は，E ではない」 この不変条件を検査するために，GNATprove は 2 つの検査を行います．最初の検査は，ループの最初の繰り返しで表明が維持されているかです．これはツールは検証されたとはしないでしょう．実際，配列 A の最初の要素が，E と異なることに理由はありません．しかし，2番目の検査は成功するでしょう．

実際，もし最初の A の要素が，ある繰り返し回数において E ではなく，次も E ではないと演繹することは簡単です．

もし，不変条件をループの最後に移動すると，GNATprove は検証に成功することに注意して下さい．



ループ不変条件は，ループ全体の複雑なプロパティを検証できるばかりか，GNATprove が他のプロパティ（例えば実行時エラーがないこと）を，ループ本体とループに続く命令文に渡って検証するためにも使用します．より正確には，実行時検査および，ループの本体ないしはループに続く命令文から他の表明の検査を行うとき，この検査に先立ち，最後のループ不変条件は，維持されていると仮定しています．例えば，Find 関数において，GNATprove は，ループ終了後，全ての A の要素は，Eとは異なると検証できます．ループ不変条件は，最後のループの繰り返しにおいて，維持されるからです．



良いループ不変条件を作ることは，きわめて難しい．この作業を容易にするために，ループ不変条件の良いプロパティを考察します．最初に，ループ不変条件は，ループの最初の繰り返しで証明可能であるべきです．このプロパティを達成するために，ループ不変条件の初期化は，Program Integrity コース（No.3：プログラムの完全さの証明）にある戦略を用いて，失敗した場合は証明のデバッグを試みます（pp.12-17）．

次に，ループ不変条件は，実行時エラーがないことを証明するに足りるだけ十分に正確であるべきです．ループの本体，およびループに続く命令文においてそうであるべきです．このようにするためには，不変条件中で記述されていないループ中で更新される変数に関する全ての情報を，ツールは忘れてしまうということを覚えておくべきです．特に，不変条件中に，ループのフレーム状態と呼ばれるものを含むように注意すべきです．ループによって更新されることのない複合変数の部分を保持するように記述することになります．最後に，ループ不変条件は，ループの継続的な繰り返しを通じて保存されるものを証明するのに十分正確であるべきです．これは，一般には，もっとも手際を必要とする箇所です．ループ不変条件の維持がどうしてGNATprove で証明されないかを理解するために，どの点で表明が機能しなくなるかを知るために，ループ本体のあちこちに局所的な表明を置くことはしばしば役に立ちます．

例にあるループを見ます．配列 A において，関数 F を全ての要素に適用します．ループのあと，各配列の要素は，前回の繰り返しで同じインデクスを持つ A に保持された値に対して，F を適用した結果です．プロパティを特定するために，ループに先立ち，配列 A の値をゴースト変数 A_I にコピーします．ループ不変条件として，K より小さなインデクスに関して，期待したように配列は変更されています．このループ不変条件は，良いループ不変条件の 4 つの良いプロパティを持っているでしょうか?

GNATprove をたちあげて見ます．INIT は満足しています．不変条件の初期化は証明されています．INSIDEとAFTERでは，実行時エラーは報告されておらず，ループ後の表明は成功裏に検証されています．しかし，PRESERVEプロパティで失敗します．最初の繰り返しの後，不変条件が保持されていると証明できなかったと，GNATprove は報告します．この失敗した証明を調査すると，ループの最初の繰り返しでは，A(K)はA_I(K）と等しいので，GNATprove は検証できないという問題があることが分かります．実際，配列 A はループ中で更新され，最初の繰り返し以降で分かることは，不変条件中で述べられていることです．残念なことに，現在のインデクスのあとの A の値について不変条件は何も云っていません．ループのフレーム状態を見失っています．K より大きな全てのインデクス J に関


Quiz
Copyright © AdaCore

正しいですか 1/10

はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
package Ring_Buffer with SPARK_Mode is
  function Valid_Model (M : Nat_Array) return Boolean;

  function Get_Model return Nat_Array with Ghost,
    Post => Valid_Model (Get_Model'Result)
      and Get_Model'Result'First = 1
      and Get_Model'Result'Length in Length_Range;

  procedure Push_Last (E : Natural) with
    Pre  => Get_Model'Length < Max_Size,
    Post => Get_Model = Get_Model'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Get_Model return Nat_Array is
    Result : Nat_Array (1 .. Length);
  begin
    ...
```

正しいですか 1/10

はい

```
package Ring_Buffer with SPARK_Mode is
  function Valid_Model (M : Nat_Array) return Boolean;

  function Get_Model return Nat_Array with Ghost,
    Post => Valid_Model (Get_Model'Result)
    and Get_Model'Result'First = 1
    and Get_Model'Result'Length in Length_Range;

  procedure Push_Last (E : Natural) with
    Pre  => Get_Model'Length < Max_Size,
    Post => Get_Model = Get_Model'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Get_Model return Nat_Array is
    Result : Nat_Array (1 .. Length);
  begin
```

Get_Model は，仕様部のみで使われているので，これは正しい．Get_Model の呼び出しは，バッファの中身をコピーすることに注意して下さい．これらは最終的なコードにおいては，コンパイラによって自動的に取り除かれます．

正しいですか　2/10

はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
package Ring_Buffer with SPARK_Mode is
  type Model_Type (Length : Length_Range := 0) is record
    Content : Nat_Array (1 .. Length);
  end record with Ghost;
  Model : Model_Type with Ghost;

  function Valid_Model return Boolean;

  procedure Push_Last (E : Natural) with
    Pre  => Valid_Model and Model.Length < Max_Size,
    Post => Valid_Model and Model.Content = Model.Content'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Valid_Model return Boolean is
    (Length = Model.Length and then ...);

  ...
end Ring_Buffer;
```

正しいですか 2/10

いいえ

```
package Ring_Buffer with SPARK_Mode is
  type Model_Type (Length : Length_Range := 0) is record
    Content : Nat_Array (1 .. Length);
  end record with Ghost;
  Model : Model_Type with Ghost;

  function Valid_Model return Boolean;

  procedure Push_Last (E : Natural) with
    Pre  => Valid_Model and Model.Length < Max_Size,
    Post => Valid_Model and Model.Content = Model.Content'Old & E;

  ...
end Ring_Buffer;

package body Ring_Buffer with SPARK_Mode is
  Content : Nat_Array (1 .. Max_Size);
  First   : Index_Range;
  Length  : Length_Range;

  function Valid_Model return Boolean is
    (Length = Model.Length and then ...);
```

ゴースト変数である Model は，通常の関数 Valid_Model の返り値に影響を与えることができません．Valid_Model は，仕様部のみで使用できます．Ghost とマークすることができます．


正しいですか 3/10
はい
（チェックアイコンをクリックする）
いいえ
（エラーの場所をクリックする）
Model : Model_Type with Ghost;
procedure Pop_When_Available (E : in out Natural) with
Pre => Valid_Model,
Post => Valid_Model,
Contract_Cases =>
(Model.Length > 0 => E & Model.Content = Model.Content'Old,
others => Model = Model'Old and E = E'Old);
procedure Pop_When_Available (E : in out Natural) is
begin
if Length > 0 then
Model := (Length => Model.Length - 1,
Content => Model.Content (2 .. Model.Length));
E := Content (First);
Length := Length - 1;
First := (if First < Max_Size then First + 1 else 1);
end if;
end Pop_When_Available;
Copyright © AdaCore

（訳注）スライド中のゴースト文というのは，自身への値の割り当て等の文で，非ゴースト文に影響を与えない文のこと

正しいですか 4/10

はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) with
  Pre             => Valid_Model,
  Post            => Valid_Model,
  Contract_Cases =>
    (Model.Length > 0 => E & Model.Content = Model.Content'Old,
     others           => Model = Model'Old and E = E'Old);

procedure Pop_When_Available (E : in out Natural) is
begin
  if Model.Length > 0 then
    Model := (Length  => Model.Length - 1,
              Content => Model.Content (2 .. Model.Length));
  end if;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

正しいですか 4/10

いいえ

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) with
  Pre             => Valid_Model,
  Post            => Valid_Model,
  Contract_Cases =>
    (Model.Length > 0 => E & Model.Content = Model.Content'Old,
     others           => Model = Model'Old and E = E'Old);

procedure Pop_When_Available (E : in out Natural) is
begin
  if Model.Length > 0 then
    Model := (Length  => Model.Length - 1,
              Content => Model.Content (2 .. Model.Length));
  end if;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

Model に関するテストは，自身の値を更新するのに用いられているときですら，許されていません．実際，コンパイラによってゴーストコードの除去を単純化するために，通常のコードにおいて，ゴーストと考えられる文のみがゴースト変数の割り当てとゴースト手続き呼び出しができます．

正しいですか 5/10

はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) is
  procedure Update_Model with Ghost is
  begin
    if Model.Length > 0 then
      Model := (Length  => Model.Length - 1,
                Content => Model.Content (2 .. Model.Length));
    end if;
  end Update_Model;

begin
  Update_Model;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

正しいですか 5/10

はい

```
Model : Model_Type with Ghost;

procedure Pop_When_Available (E : in out Natural) is
  procedure Update_Model with Ghost is
  begin
    if Model.Length > 0 then
      Model := (Length  => Model.Length - 1,
                Content => Model.Content (2 .. Model.Length));
    end if;
  end Update_Model;

begin
  Update_Model;

  if Length > 0 then
    E := Content (First);
    Length := Length - 1;
    First := (if First < Max_Size then First + 1 else 1);
  end if;
end Pop_When_Available;
```

ここでは全て正しいです．Model は，それ自身がゴースト手続きである Update_Model の内側でのみアクセスできます．更に，Update_Model に対して契約を追加する必要がありません．実際，局所的手続きなので，GNATprove によってインライン化されます．


正しいですか
6/10
はい
（チェックアイコンをクリックする）
いいえ
（エラーの場所をクリックする）
function Max_Array (A, B : Nat_Array) return Nat_Array with
Pre => A'Length = B'Length;
function Max_Array (A, B : Nat_Array) return Nat_Array is
R : Nat_Array (A'Range) := (others => 0);
J : Integer := B'First;
begin
for I in A'Range loop
if A (I) > B (J) then
R (I) := A (I);
else
R (I) := B (J);
end if;
J := J + 1;
end loop;
return R;
end Max_Array;
Copyright © AdaCore

正しいですか 6/10

はい

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

プログラムは正しい．残念なことに，GNATproveは，J が B のインデクス範囲にいるということを検証するのに失敗します．実際，ループの本体を検査するときに，GNATprove は，前回ループで変更されている可能性があるので J の現在値を全て忘れます．より正確な結果を得るためには，ループ不変条件を提供する必要があります．

正しいですか 7/10

はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    pragma Loop_Invariant (J in B'Range);
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

正しいですか 7/10

はい

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre => A'Length = B'Length;

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
  J : Integer := B'First;
begin
  for I in A'Range loop
    pragma Loop_Invariant (J in B'Range);
    if A (I) > B (J) then
      R (I) := A (I);
    else
      R (I) := B (J);
    end if;
    J := J + 1;
  end loop;
  return R;
end Max_Array;
```

ループ不変条件は，今回はループ本体で，実行時エラーが生じないことを検証できます．残念なことに，GNATprove は，不変条件がループの最初の繰り返しのあと妥当であるということを検証に失敗します．実際に，任意の繰り返しで，J が B の範囲にいるということをだけでは，次の繰り返しでもそうであると示すには十分ではありません．より正確な不変条件が必要です．J をループインデクス I の値にリンクすることです．例えば，次になります：J = I - A'First + B'First.

正しいですか 8/10

はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post =>
   (for all K in A'Range =>
         Max_Array'Result (K) = Natural'Max (A (K), B (K)));

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I =>
                R (K) = Natural'Max (A (K), B (K)));
    if A (I) > B (I) then
      R (I) := A (I);
    else
      R (I) := B (I);
    end if;
  end loop;
  return R;
end Max_Array;
```

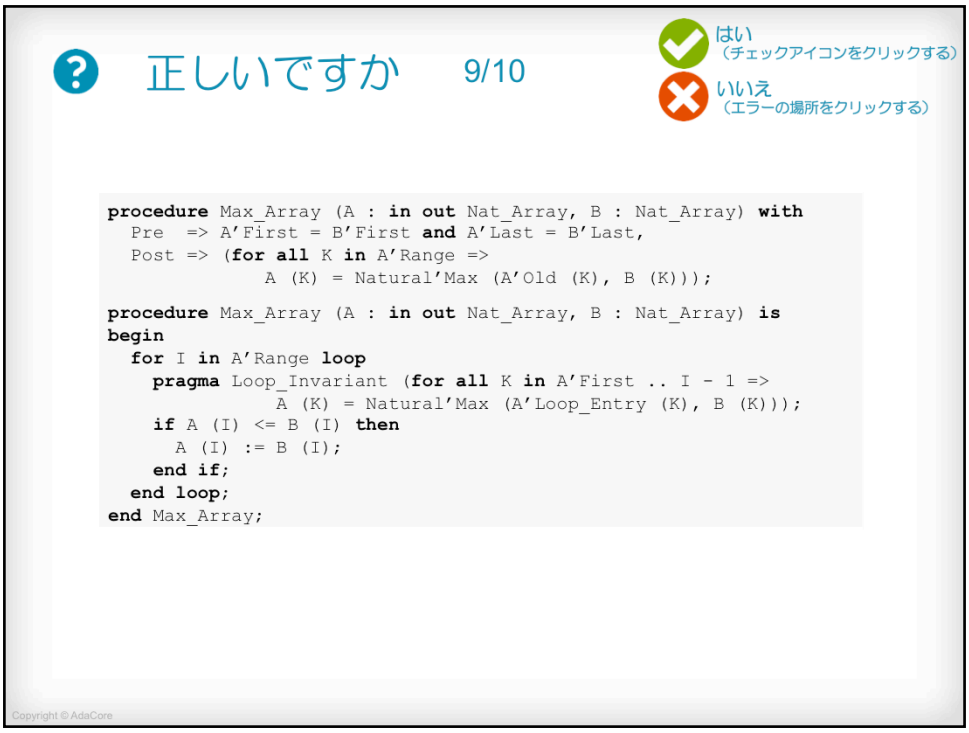
正しいですか 8/10

いいえ

```
function Max_Array (A, B : Nat_Array) return Nat_Array with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post =>
   (for all K in A'Range =>
         Max_Array'Result (K) = Natural'Max (A (K), B (K)));

function Max_Array (A, B : Nat_Array) return Nat_Array is
  R : Nat_Array (A'Range) := (others => 0);
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I =>
                R (K) = Natural'Max (A (K), B (K)));
    if A (I) > B (I) then
      R (I) := A (I);
    else
      R (I) := B (I);
    end if;
  end loop;
  return R;
end Max_Array;
```

このプログラムは正しい．しかし，ループ不変条件が正しくありません．関数 Max_Array で表明付きで実行することによって検査できます．実際，各ループの繰り返しで，配列 R は，インデクス毎に，配列 A と B の小さくない値を持ちます．ただし，I -1 までになります．I 番目は，まだ計算していません．


正しいですか
9/10
はい
（チェックアイコンをクリックする）
いいえ
（エラーの場所をクリックする）
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
Pre => A'First = B'First and A'Last = B'Last,
Post => (for all K in A'Range =>
A (K) = Natural'Max (A'Old (K), B (K)));
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
for I in A'Range loop
pragma Loop_Invariant (for all K in A'First .. I - 1 =>
A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
if A (I) <= B (I) then
A (I) := B (I);
end if;
end loop;
end Max_Array;
Copyright © AdaCore

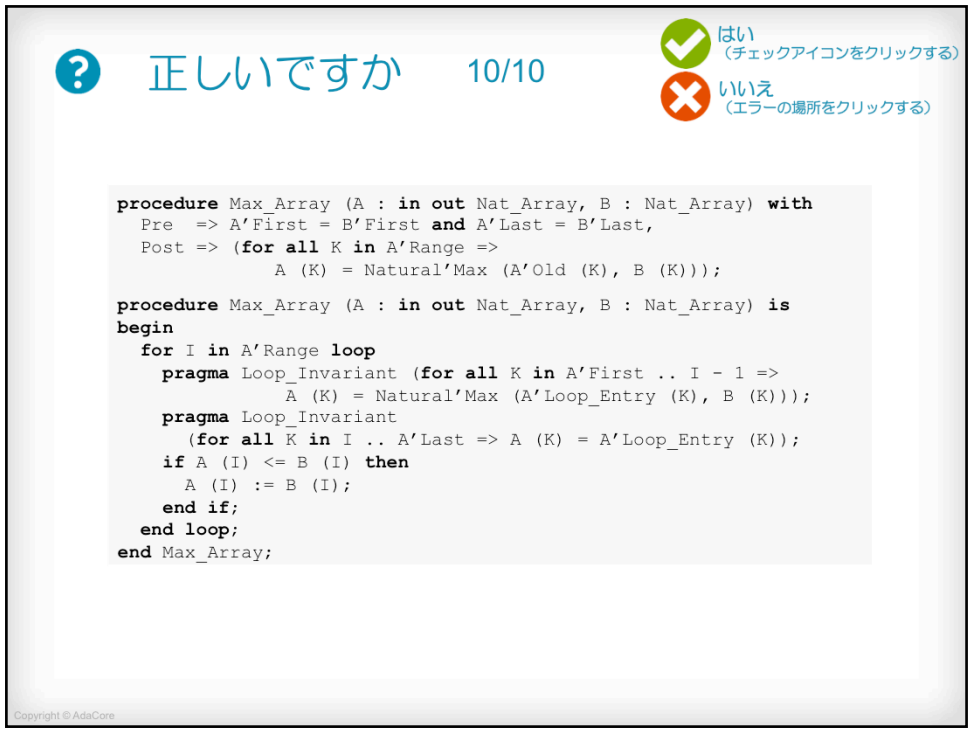
正しいですか 9/10

はい

```
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
             A (K) = Natural'Max (A'Old (K), B (K)));

procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I - 1 =>
                             A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
    if A (I) <= B (I) then
      A (I) := B (I);
    end if;
  end loop;
end Max_Array;
```

今回，このプログラムは正しい．残念なことに，GNATprove は，最初の繰り返しのあと，GNATprove は，ループ不変条件が妥当である，ことを検証することができません．I が前回の繰り返しで保存された後で，配列 A に保存されている値が分からないからです．ループで変更されない複合変数の部分は，フレーム状態と呼ばれます．ループ不変条件を使用する時は，フレーム状態を忘れないように注意が必要です．


正しいですか 10/10
はい
（チェックアイコンをクリックする）
いいえ
（エラーの場所をクリックする）
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
Pre => A'First = B'First and A'Last = B'Last,
Post => (for all K in A'Range =>
A (K) = Natural'Max (A'Old (K), B (K)));
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
for I in A'Range loop
pragma Loop_Invariant (for all K in A'First .. I - 1 =>
A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
pragma Loop_Invariant
(for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
if A (I) <= B (I) then
A (I) := B (I);
end if;
end loop;
end Max_Array;
Copyright © AdaCore

正しいですか 10/10

はい

```
procedure Max_Array (A : in out Nat_Array, B : Nat_Array) with
  Pre  => A'First = B'First and A'Last = B'Last,
  Post => (for all K in A'Range =>
             A (K) = Natural'Max (A'Old (K), B (K)));

procedure Max_Array (A : in out Nat_Array, B : Nat_Array) is
begin
  for I in A'Range loop
    pragma Loop_Invariant (for all K in A'First .. I - 1 =>
                             A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
    pragma Loop_Invariant
      (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
    if A (I) <= B (I) then
      A (I) := B (I);
    end if;
  end loop;
end Max_Array;
```

今回，不足していたフレーム状態を追加しました．プログラムは，GNATproveで，正しく検証できます．


ADACORE
university.adacore.com
Copyright © AdaCore