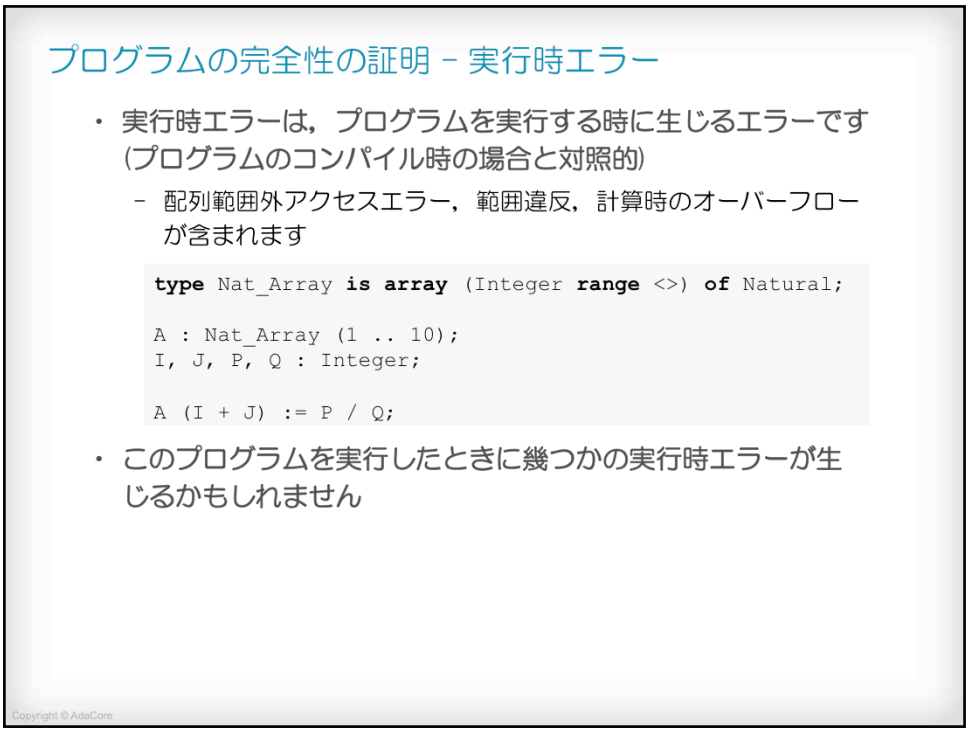
GNATproveを用いて，プログラム分析を行う 2つ目の講義にようこそ．

この講義は，2つ目の分析タイプであるプログラムの完全性の証明についての講義です．

確実性を期すために，この分析は，フロー分析で得られた条件に依存しています．もし，プログラムにまだ解決していないフロー分析のメッセージがあれば，先に進むべきではありません．

プログラムの完全性証明を行う主たる目的は，プログラム実行中の実行時エラーがないことを保証することです．



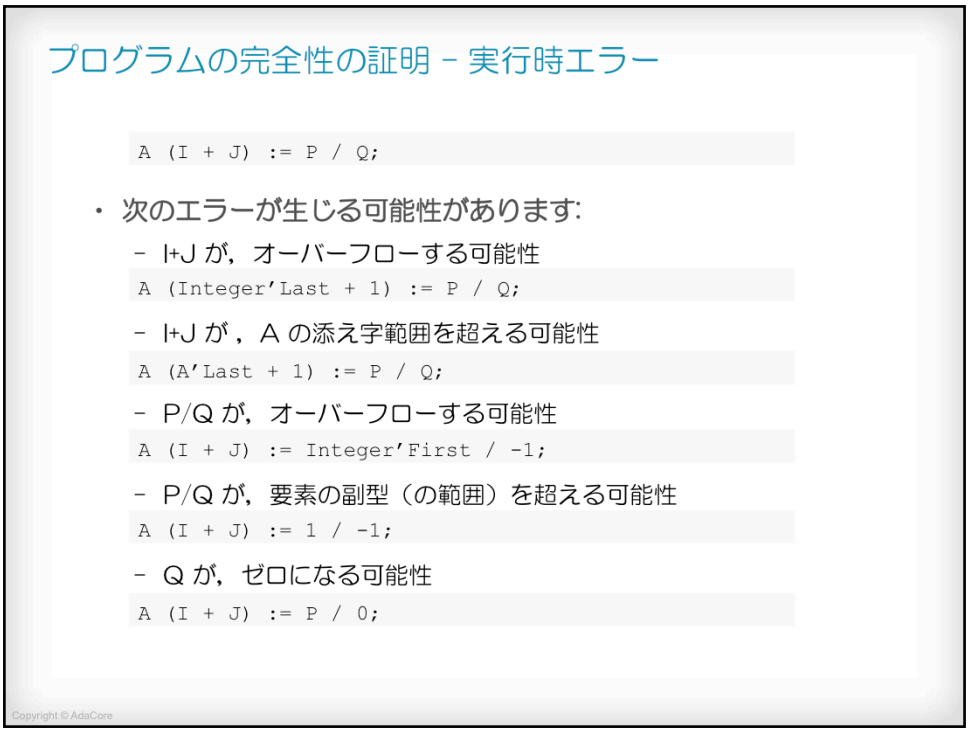
プログラムの実行中に生じるかもしれないが，コンパイル時には見つからない潜在的なエラーがつねにあります．

これらのエラーは，実行時エラーと呼ばれ，GNATprove の分析のターゲットとなります．

さまざまな実行時エラーがあります．配列に対する範囲外アクセス，サブタイプの範囲違反，計算時のオーバーフロー，そしてもっとも良くあるのが，ゼロ割です．

このスライドにあるコード例を見て下さい．自然数による配列 A I+J番目のセルに，P/Qをセットする割り当て文を見ましょう．

もし，I, J, P, Q の値を知らなければ，このコードを実行する時に起きるからもしれないエラーが何かが分かりません．



実際，重要な幾つかの場合があります．

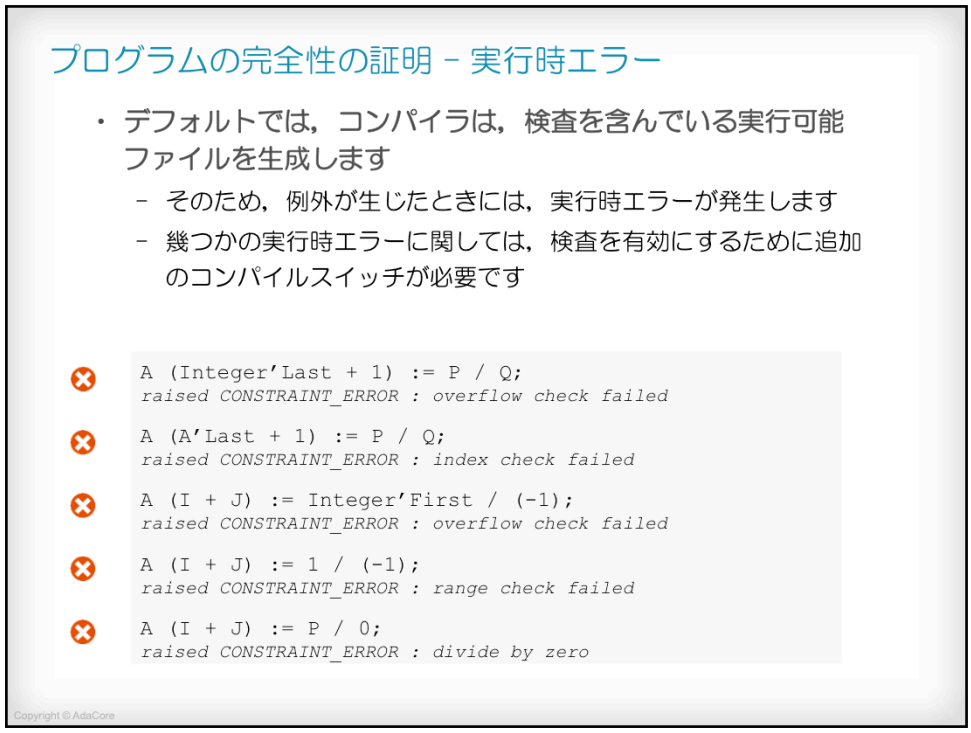
第一に，I + J がオーバーフローする可能性です．例えば，I が整数値の最大値で，J が正の数の場合です．

次に，I + J が，配列 A の添え字の範囲外になる可能です．

割り当ての他の側面としては，割り算がオーバーフローする可能性です．しかし，P が整数の最初の数で，Q が -1 というとても特別な場合です．符号付き整数型の非対称な範囲のためです．

配列が自然数を含んでいる場合，負数を保持使用するときにもエラーになります．

最後に，Q がゼロといった場合の割り算も許されません．

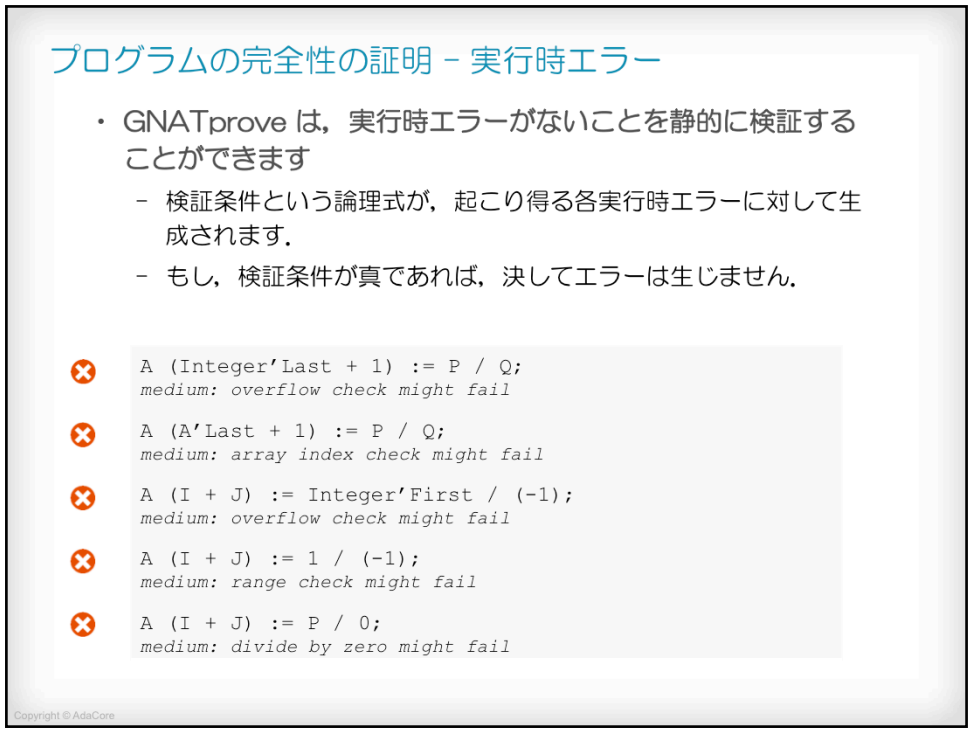


これら全てのランタイムエラーに対して，コンパイラは，実行可能ファイル中に，検査を埋め込みます．不斉な状態に到達しないことを確認する，或いは確認したときに問題があることが分かった場合，例外を送出します．

このスライドには，異なった割り当て文に対して，検査で問題があったときにどういう例外が送出されるかを示しています．

これらの実行時検査にはコストが掛かります．プログラムサイズと実行時間という意味のコストです．

コストがゼロになることはなく，コストは文脈に依存します．実行時にエラーが生じると静的に確認できるならば，これら検査を取り除くことが望ましい．



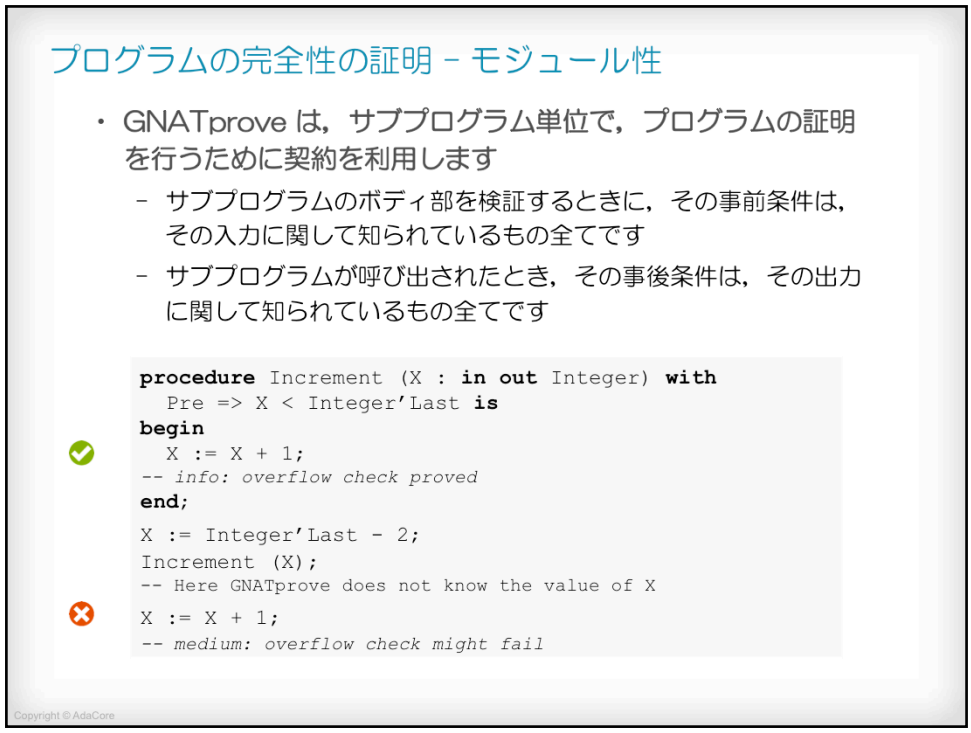
ここにあるのは，実行時にエラーが生じないことを静的に示すために GNATprove が用いる分析です．

より正確には，GNATprove は，プログラム中の全ての命令文の意味を論理的に解釈します．

この解釈を用いて，GNATproveは，論理式を生成します．また，可能な検査に対して名前付け検証条件を生成します．ここでの可能な検査とは，コードの妥当性確認を意味します．

次に，検証条件は，自動証明器に渡されます．

プログラムに対する全ての検証条件が証明器によって検証されたならば，このプログラムは実行時に，エラーが生じないということを意味します．



拡張性の点から，GNATprove は，サブプログラム単位でプログラムの証明を行います．

このため，各サブプログラムの入力・出力状態を適切にまとめるために，事前・事後条件を用います．

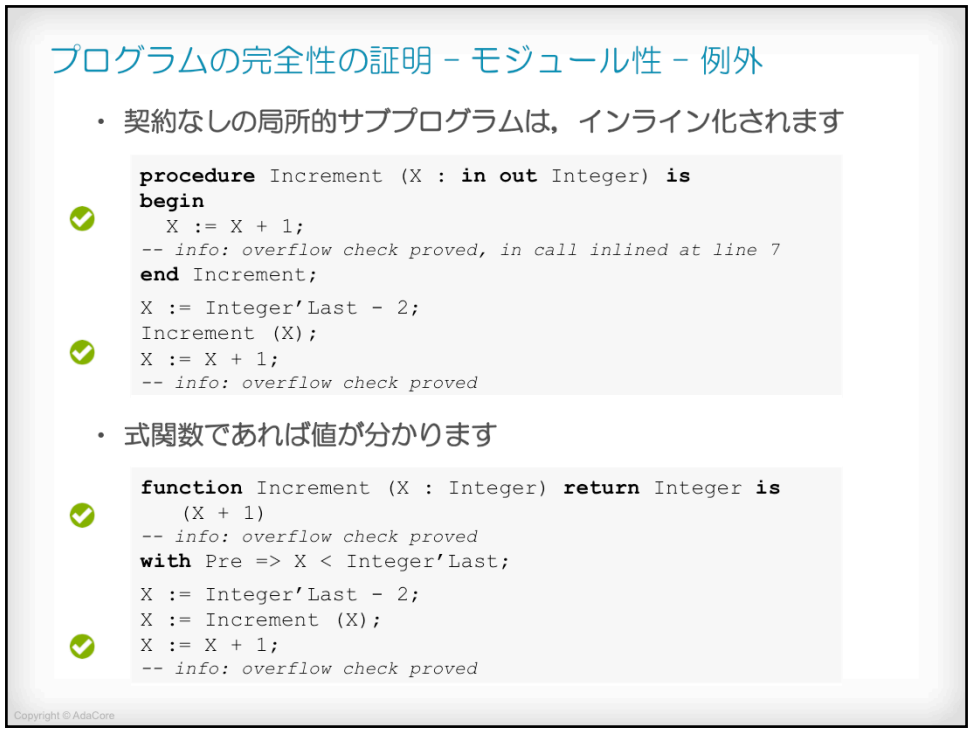
より正確には，サブプログラムのボディ部を検証する場合に，GNATproveは，次を仮定します．サブプログラムの事前条件に記述されている内容を除き，パラメータやアクセスするグローバル変数の取り得る初期値は分からないということです．

事前条件が与えられなければ，仮定はありません．スライドにある Increment のボディ部に示されているコードは，事前条件を与えられている例です．パラメータ X は，Integer'Last より小さいという事前条件の制約を持つとして，正しく検証されます．

同様に，サブプログラムが呼ばれたときに，GNATprove は，その out と in out パラメータおよび広域変数は，事後条件に合うように更新されると仮定します．

例えば，Increment は事後条件を持たないので，GNATproveは，呼び出しの後，Integer'Lastより小さいと云うこと知ることができません．

それゆえ，次の加算がオーバーフローしないということを証明できません．



GNATproveによって，モジュール性が強制されない場合が二通りあります．

契約がない局所的サブプログラムが，十分に単純であればインライン化することができます．しかし，それらは再入可能であったり，複数のリターンポイントを持つべきではありません．

もし，Increment から契約を取り除いた場合，インライン化の基準に適合します．

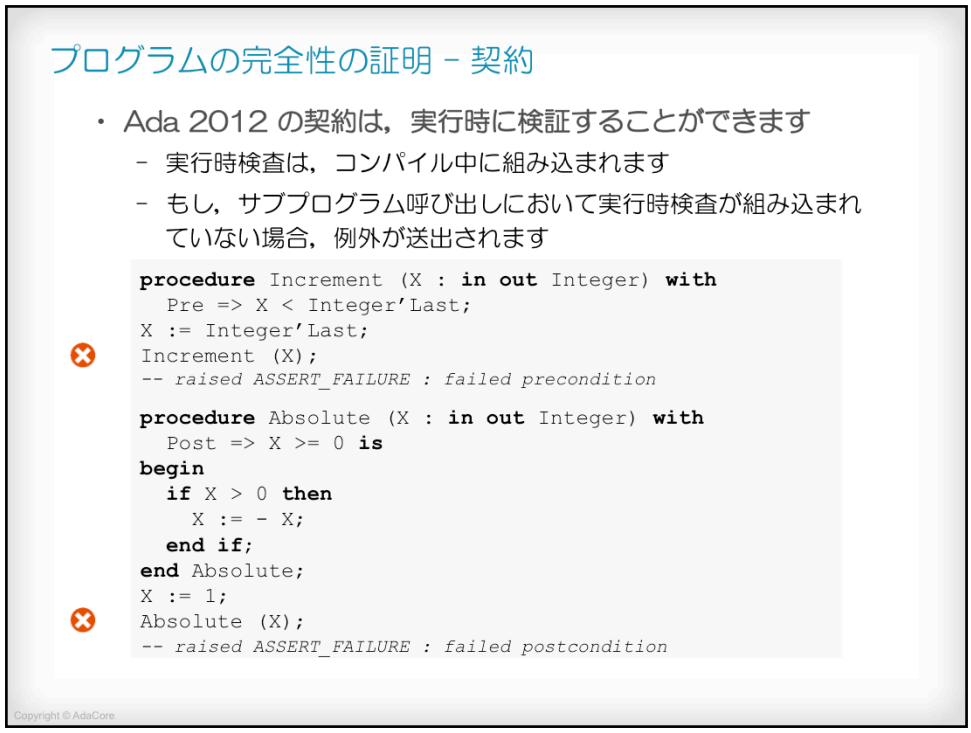
GNATprove は，あたかも X が直接値を増やすかのように Increment への呼び出しを捉えるので，続いて行う加算のいずれかで，オーバフローが生じないと云うことを正しく検証することができます．

他のケースは，式関数に関係しています．

もし，ある関数が，契約の有無によらず，式関数として定義されていたならば，その結果の値を示す事後条件を持っているかのごとく，扱われます．

スライドの例では，Increment は，式関数で置き換えられ，GNATproveは，続く加算においてオーバフローの検査を正しくできます．

（訳注）式関数（expression function）単一の式で関数を記述する．本体を別に必要としない



Ada 2012 における契約は，形式検証に完全に適合していますが，主として，実行時に検査がなされるように設計されています．

実行時の契約の検証を行うコードは，適切なスイッチを用いて，コンパイラにより生成することができます．スイッチは，-gnata です．

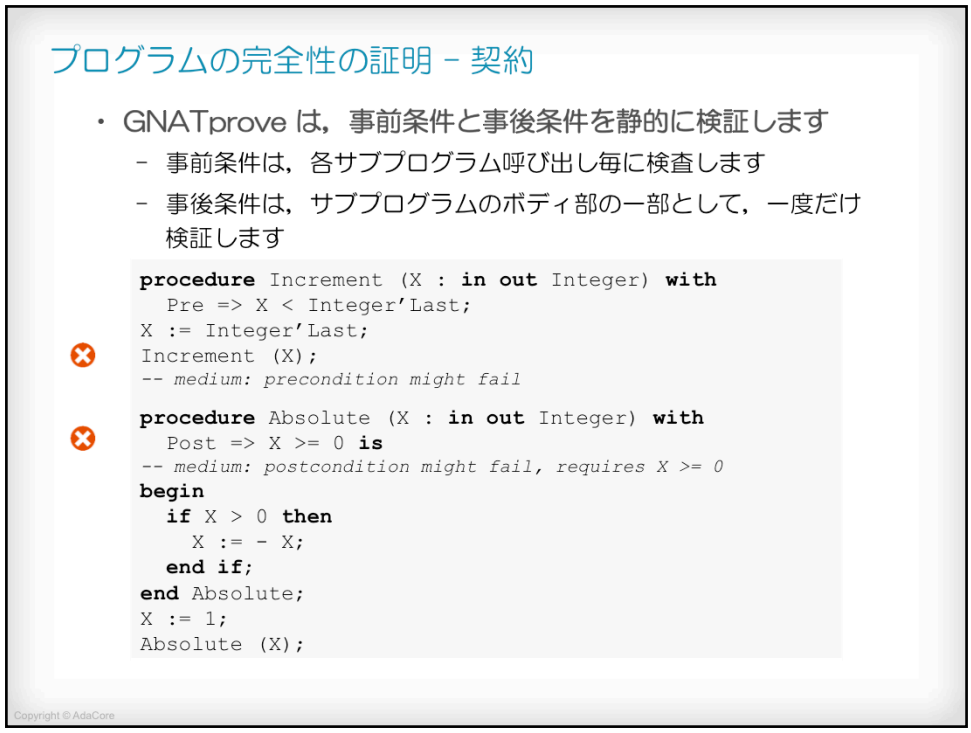
もし，Ada 2012 の契約が，あるサブプログラム呼び出しで保持されていないならば，assert_failure という例外が送出されます．

これは，開発とテストにおいて，特に便利です．しかし，表明の実行，特に事前条件は，不斉な状態に到達することを避けるために，（出荷のために）最終的にプログラムを準備しているあいだも，保持することが望ましいことになります．

例えば，もし，Integer‘Last を引数にして Increment を呼び出すと，プログラムの広域状態の不斉な更新を避けるために，ボディ部が開始する前にプログラムの実行は失敗します．

同様に，誤って実装された関数 Absolute の呼び出しにおいて，0 以外の入力では，呼び出し側は，負数を受け取ります．
（訳注：絶対値として事後条件は合っていますが，結果はこの事後条件に違反しています）

初期に検知することで，修正が容易となり，デバッグを進めることができます．



分析が妥当であることを保証するために，GNATprove は，事前条件と事後条件を静的に検証します．

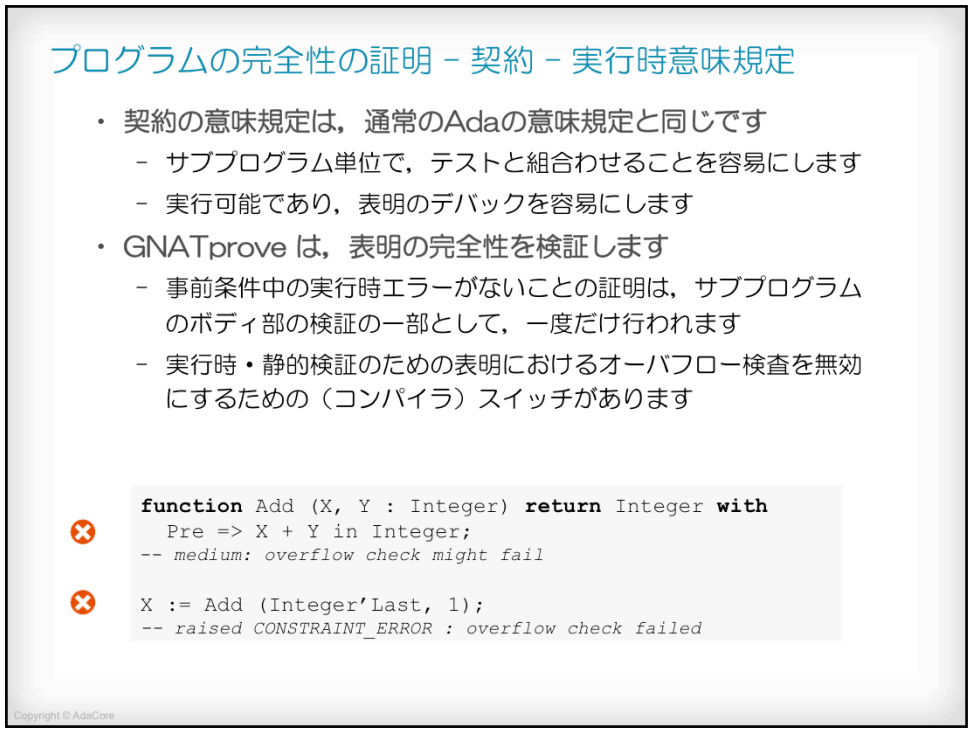
契約の実行時意味規定と同様に，事前条件は，サブプログラムが呼び出されるたびに検証されます．

一方で，事後条件は，サブプログラムのボディ部の検証の一部として，一度だけモジュール単位で検証されます．

私たちの例では，GNATprove は，二つのエラーを検知し，都度報告します．

事前条件に関しては，Increment が不適切に呼び出されるまで待つ必要があります．事前条件は，実際，呼び手に対する契約だからです．

一方で，手続き Absolute において，事後条件が，その条件を満足するのに必要な入力を持っていないということを検知するために，呼び出しを待つ必要はありません．



Ada 2012 では，契約中の式は，ブール式における規則的な意味を持ちます．

特に，実行時エラーが計算中に起きる可能性があります．

表明のデバッグ・テストと静的検証の組合わせを，共に促進するために，GNATproveは同様の意味規定を持ちます．

プログラムの証明において，契約の実行中，エラーが生じないことを確認します．

この意味規定は，時には負担です．特に，オーバーフロー検査においては，そうです．

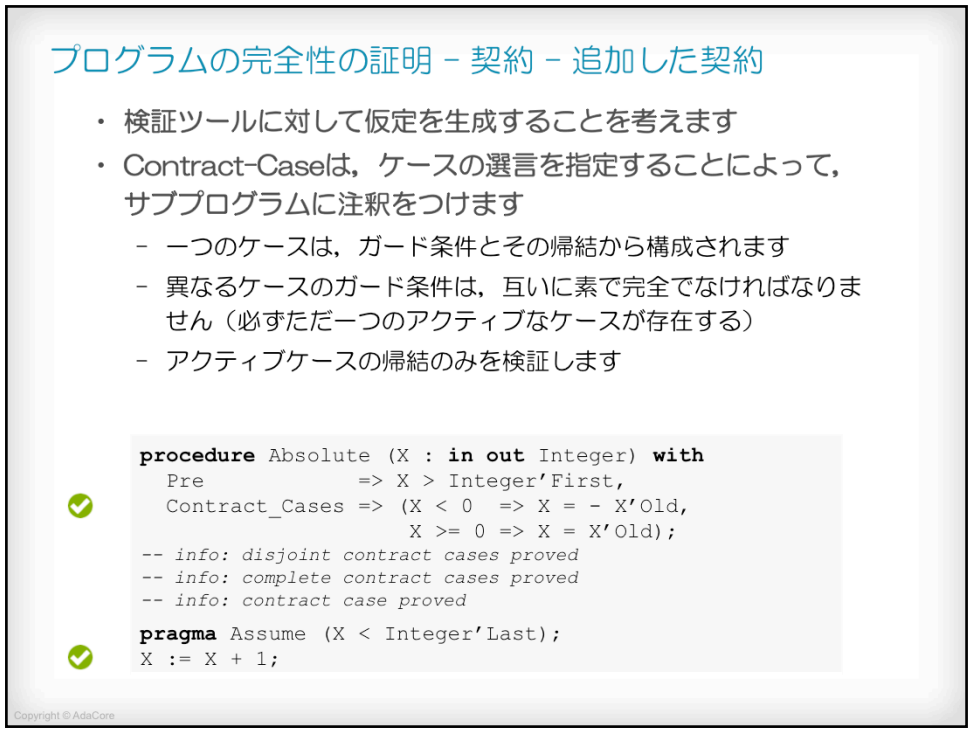
例えば，オーバフローを避けるように関数 Add に対して適切な事前条件を指定しようと試みてみます．X と Y の加算を計算するときに，ボディ部におけるオーバーフローを避けます．

残念なことに，表明中の式は，通常の Ada の意味規則なので，GNATprove は，Add の事前条件を検査するときにエラーを送出される可能性があると指摘します．

これは，正しいふるまいです．既にみたように，引数が Integer'Last と1の場合にそうなります．

一方で，文脈に従って，GNATprove が加算に関する数学的意味規定を用いることで，Addのボディ部において実行時にエラーが送出されないことを適切に検証することが望ましいと考えることもできます．

-gnato というコンパイラスイッチを用いると，このふるまいになります．このスイッチにより，コードと表明中で，オーバフローモードであることを独立に設定することができます．オーバーフロー検査の数を減らすか，完全に取り除くためです．



これまでに見てきたように，契約は GNATprove の主要な特徴です．

事前条件と事後条件をサポートしています．表明と同様に，プログラマ Assert や型述語もサポートされます．

形式検証のために新しい契約を導入しています．例えば，新しいプラグマ Assume は，実行時の表明ですが，プログラムの証明のための仮定として導入しています．即ち，ツールによって検証をすることなく真と見なせるブール式です．

この特徴は，有益ですが，注意して使用する必要があります．

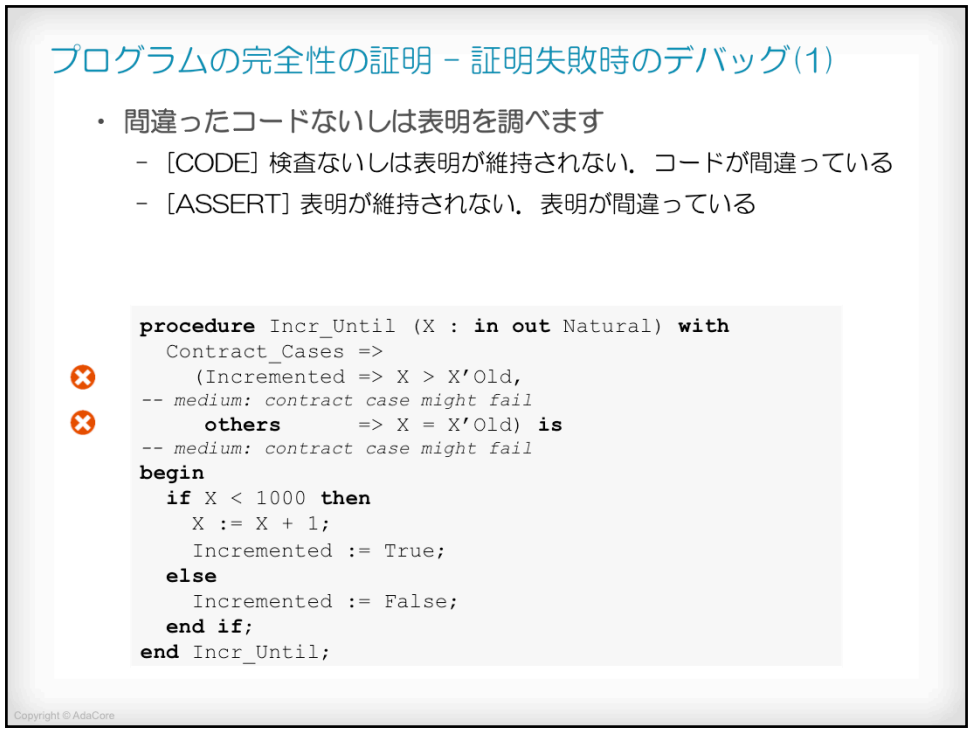
GNATproveが導入した他の構成要素として，Contract_Casesアスペクトがあります．

ケースの選言を用いてサブプログラムのふるまいを規定することができます．contract-case の各要素は，実際ガードによるそれ自身が契約となっています．しかし，サブプログラムの入力のみを参照でき，サブプログラムの呼び出し前に評価され，結果を得ます．

サブプログラムの呼び出し毎に，ガード条件が真と見なせるただ一つのケースがなければなりません．また，その結果は，サブプログラムの終了時まで保持する必要があります．

GNATprove において，Contract_Cases の選言性，完全性と同様に，妥当性は，サブプログラムの事前条件の文脈において，一度だけ検証されます．

（訳注）型述語（type predicate） Predicate を用いたユーザ型の定義



もし，GNATprove が，プログラムを検証中にエラーを報告した場合，そこには様々な理由があります．

プログラムにエラーがあるかもしれません．情報が不足しているためにプロパティを検証できない，或いは，GNATprove が用いている証明器が完全に正当な検証条件を評価できないかもしれません．

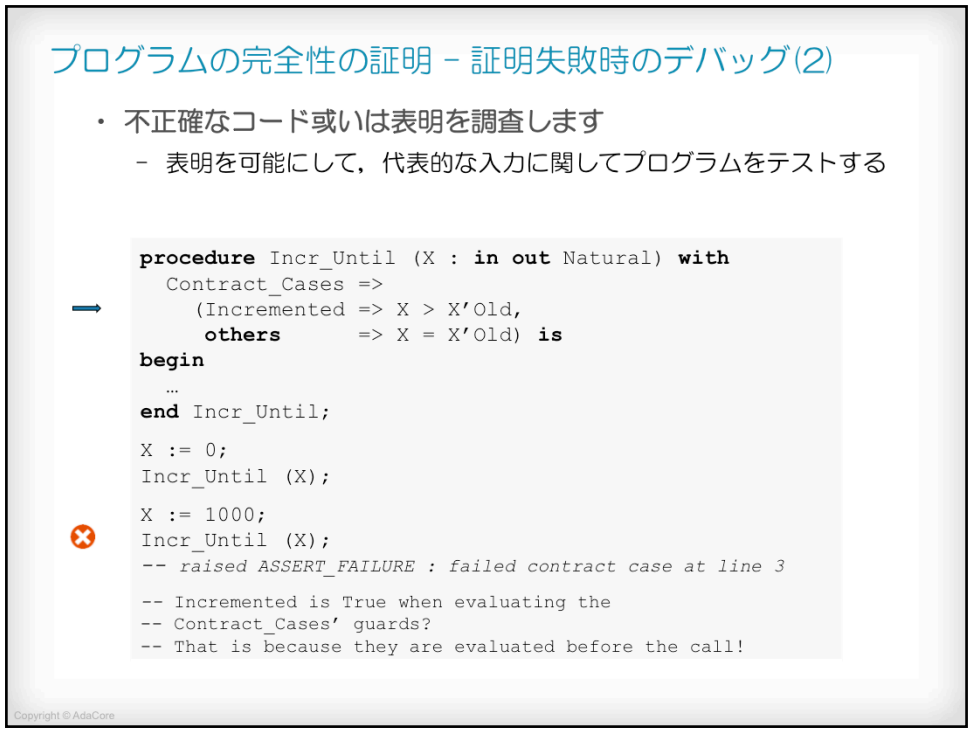
この講義の残りの部分では，証明の失敗をデバッグするための，時に手の込んだ作業について説明します．

最初は，本当にプログラムに誤りがある場合です．

二つの可能性があります．

コードが不正確であるか，ありがちですが，仕様が間違っている場合です．

例を見てみます，手続き Incr_Until にエラーがあり，Contract_Cases が証明できません．



表明は実行可能です．表明を使用するようにすることで，代表的な入力の組に関して，プログラムをテストすることは有益です．

この方法で，コードと表明にあるバグを見つけやすくなります．

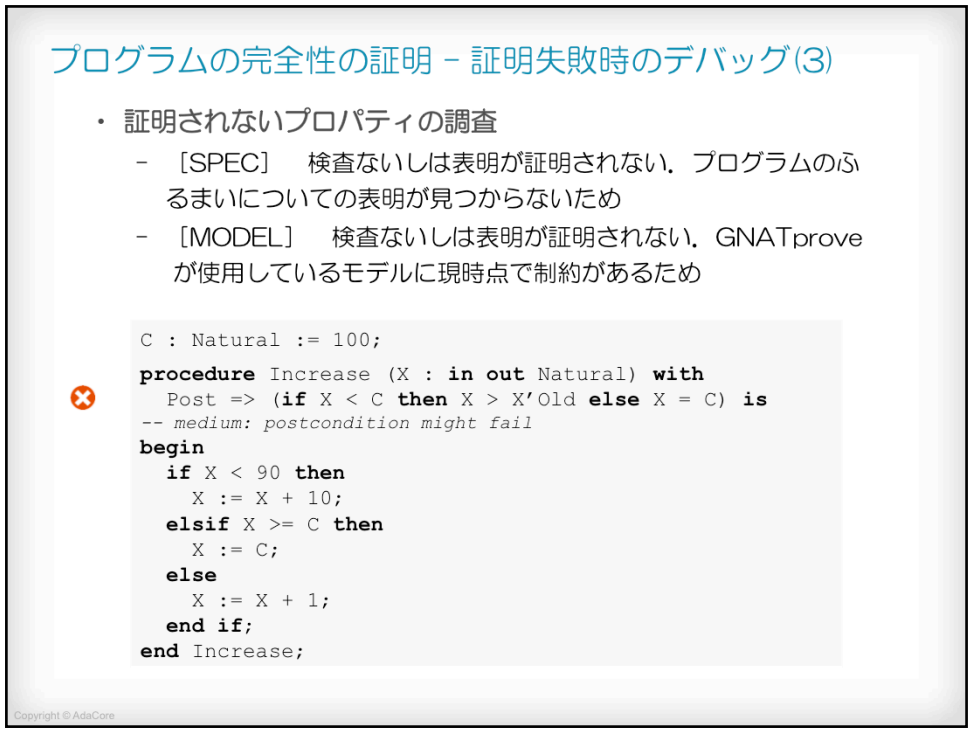
例えば，1000より大きな入力で Incr_Until をテストすると，実行時に例外が送出されます．

最初の契約ケースが失敗したことを示しています．そしてそのことは，Incremented が真であることを意味しています．

それでも，呼び出しの後，再度 Incremented の値を印字すると，それが False であることが分かります．期待と違うわけです．

なるほど，契約ケースのガード条件は呼び出し前に評価され，仕様にエラーが含まれています．

修正するためには，X < 1000 を最初の契約ケースのガード条件とするか，if 式を用いた通常の事前条件を用いるべきです．



コードと表明が正しいときでさえ，GNATproveは，プロパティに対して，未証明検証条件を出力するときがあります．

これは二つの理由から生じます．

第一に，コード中の表明に欠落があり，プロパティが未証明になっている場合です．

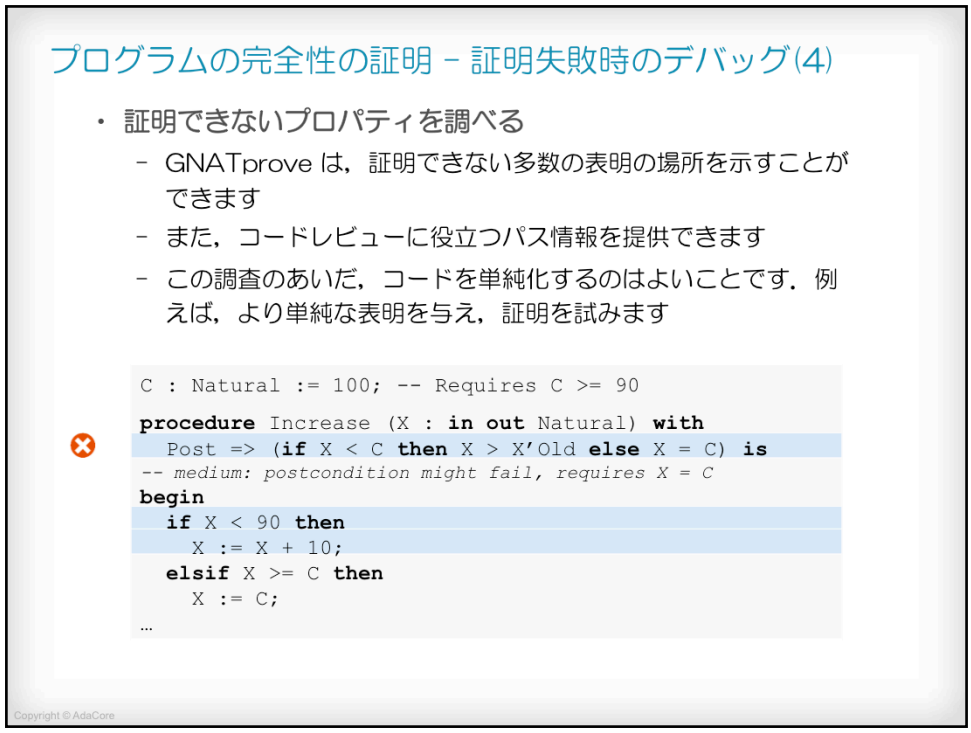
特に，これは分析のモジュール性によって生じることがあります．モジュール性によって，ツールは明示的に注釈されたプロパティを保持するのみとなるからです．

第二に，GNATprove が用いているプログラムの論理モデル中に情報が欠落している場合があるということです．

浮動小数点演算，文字列リテラルといった特徴を支援するのが困難である，ということから生じがちです．

例において，Increase の事後条件に対する GNATproveによる検証は，証明されません．

これは次のことを示しています．もし，パラメータ X がある値 C より小さかったならば，その値は手続きによって増加します．もしそれが，Cより大きくても，その値はCにとどまります．



適切なオプションを用いたとき，失敗した検証条件に対して，GNATprove は，付加的な情報を示します．

特に，状態が複雑であれば，ツールは，失敗した条件の該当箇所を特定します．

例において，GNATprove は，X = C を証明できないと示します．それは，X が C より大きい場合だということを意味します．

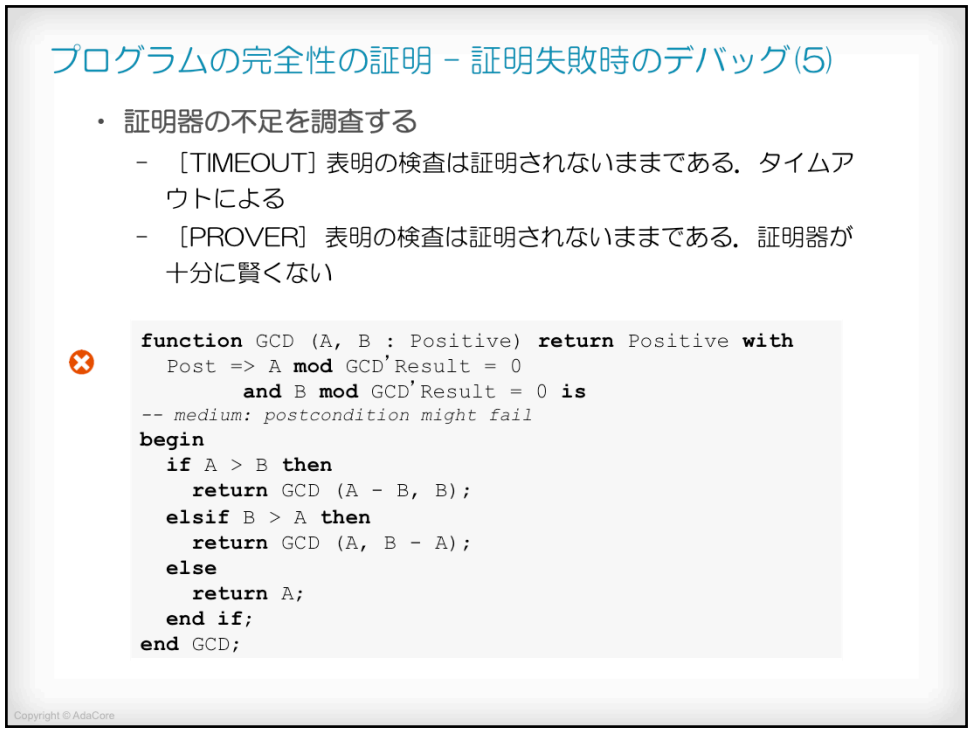
コードレビューに役立つ他の情報があります．

もし，GNATbench ないしは GPS を使用していれば，GNATprove は，失敗状態に至るパスをハイライト表示することができます．

ここでは，最初は if 文の最初の分岐です．結果として，X が C よりも大きいときと，X が 90 より小さいときは，Increase の事後条件を，GNATprove が証明できないことが分かります．

実際に，このケースでは，我々の事後条件は，維持されません．

しかし，おそらく，C の値が変化することは期待できません，或いは，すくなくとも 90 を下回ることはありません．この場合，単に C が定数であるか，Increase に対して，事後条件を加えることになります．



最後に，GNATprove がプロパティに対して，完全に正当な検証条件を提供するケースがあります．そして，それは，ツール実行後のステージで自動証明器によって証明されることはありません．

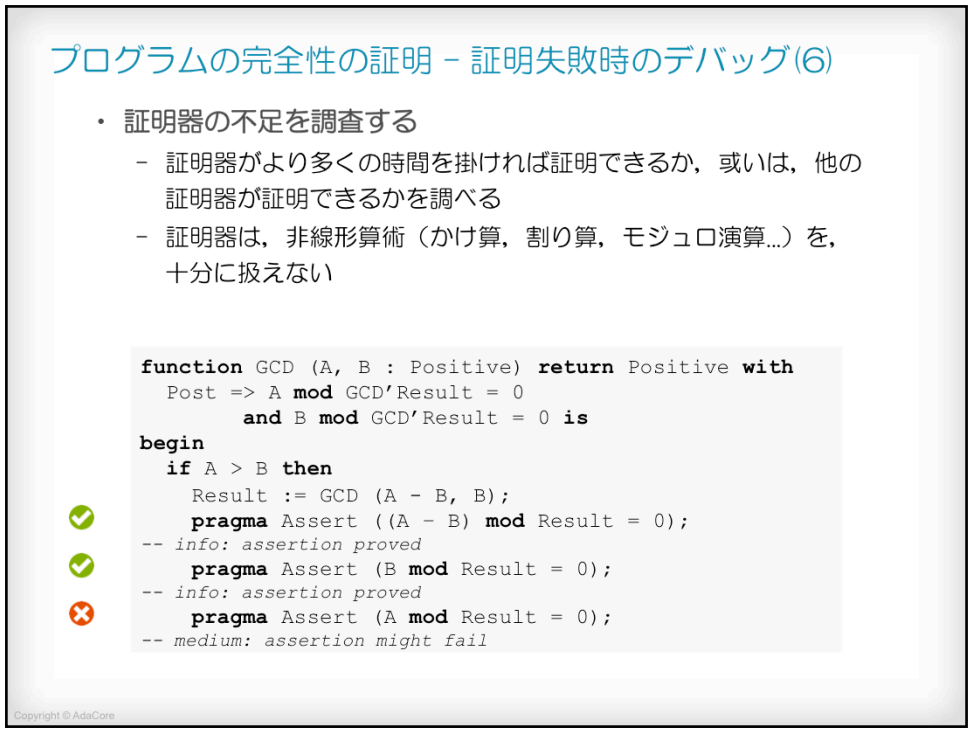
これはきわめて普通に生じます．

なるほど，GNATprove は，検証条件を一階述語論理で生成します．しかし，それは決定可能ではありません．特に，算術と組合わせる場合はです．

時には，自動証明器が，多くの時間を必要としている場合もあります．しかし，ときには，証明器は，ほぼ直ちに答えを求めることをあきらめたり，最終的な答えに到達する前に無限ループに陥る場合があります．

例えば，例にある GCD 関数の事後条件において，ユークリッドアルゴリズムを用いて，二つの正の値の GCD 値を計算しますが，GNATproveのデフォルトの設定では，検証することができません．

（訳注）GCD = Greatest Common Divisor, 最大公約数



試すべき最初のことは，各検証条件下で，証明器が費やすことの出来る許可時間の最大値を増やすことである．これには，次のGNATproveのオプション --timeout を使用するか，GPS中のダイアログボックスを使用する

我々の例では，1分で負荷が高くなり，また相対的に高い値であり，助けにはなりません．

もし他の自動証明器を持っていれば，GNATprove の --prove オプションを用いるか GPS のダイアログボックスで指定します．

例にある事後条件では，z3, Alt-ergo, CVC4 の各証明器を用いましたが，答えには到達しませんでした．

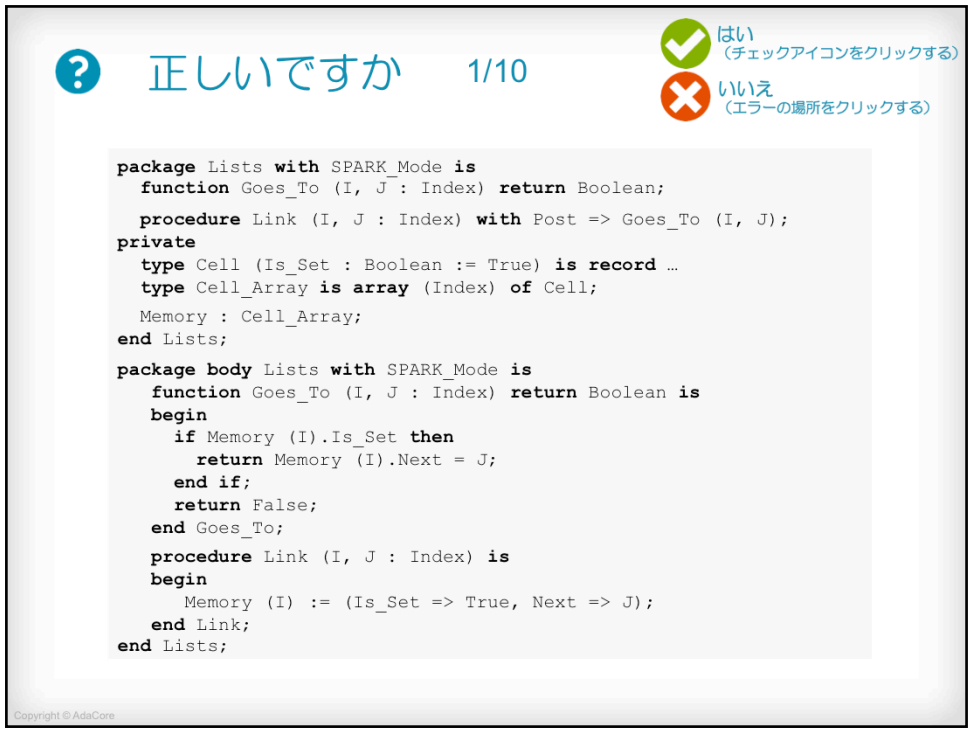
問題をよりよく理解するためには，証明を単純化するために中間の表明を追加し，問題を引き起こす部分を特定した．どうしてプロパティが証明されないかをレビューによって理解しようと試みるときに，しばしば良いアイデアとなります．

ここで，もし A - B と B が Result によって除することができ，Aがそうであれば，証明器は検証できません．

これには驚くかもしれません．しかし，かけ算，モジュロ-演算，累乗といったものを含む場合は，証明器にとって困難な問題であり，Alt-Ergo, Z3, CVC4 といった汎用目的のどの証明器を用いても，実際上うまく扱うことができません．

この講義のスライドのセクションはこれで終わりです．ここからクイズセクションに行きます．

正解は10点です．クイズの終わりで，総点数を見てみましょう．



スライドにあるコードをレビューして，このコードが正しいと思えば，YES をクリック，そうでなければ，間違っていると思う行をクリックしてください．

一度選択を行ったら，SUBMIT ボタンを押して下さい．

正しいですか 1/10

はい

```
package Lists with SPARK_Mode is
   function Goes_To (I, J : Index) return Boolean;

  procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
  type Cell (Is_Set : Boolean := True) is record …
  type Cell_Array is array (Index) of Cell;

  Memory : Cell_Array;
end Lists;

package body Lists with SPARK_Mode is
   function Goes_To (I, J : Index) return Boolean is
   begin
     if Memory (I).Is_Set then
       return Memory (I).Next = J;
     end if;
     return False;
   end Goes_To;

   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;
end Lists;
```

これは正しい．しかし，GNATprove では検証されません．Goes_Toは事後条件を持たず，結果から分かることはありません．

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 Lists with SPARK_Mode is
   function Goes_To (I, J : Index) return Boolean;

   procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
   type Cell (Is_Set : Boolean := True) is record …
   type Cell_Array is array (Index) of Cell;

   Memory : Cell_Array;

   function Goes_To (I, J : Index) return Boolean is
       (Memory (I).Is_Set and then Memory (I).Next = J);
end Lists;

package body Lists with SPARK_Mode is
    procedure Link (I, J : Index) is
    begin
       Memory (I) := (Is_Set => True, Next => J);
    end Link;
end Lists;
```

正しいですか 2/10

はい

```
package Lists with SPARK_Mode is
   function Goes_To (I, J : Index) return Boolean;
   procedure Link (I, J : Index) with Post => Goes_To (I, J);
private
   type Cell (Is_Set : Boolean := True) is record …
   type Cell_Array is array (Index) of Cell;
   Memory : Cell_Array;
   function Goes_To (I, J : Index) return Boolean is
       (Memory (I).Is_Set and then Memory (I).Next = J);
end Lists;
```

```
package body Lists with SPARK_Mode is
    procedure Link (I, J : Index) is
    begin
       Memory (I) := (Is_Set => True, Next => J);
    end Link;
end Lists;
```

Goes_To は式関数です．従って，（式関数）本体を証明のために利用可能です

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）

```
package Stacks with SPARK_Mode is
  type Stack is private;

  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      return;
    end if;

    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

正しいですか 3/10

いいえ

```
package Stacks with SPARK_Mode is
  type Stack is private;
  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      return;
    end if;
    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

Push の事後条件は，stack が一杯ではないときに Push が呼ばれたときのみ，真となります．

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）

```
package Stacks with SPARK_Mode is
  type Stack is private;

  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;

package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      raise Is_Full_E;
    end if;

    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

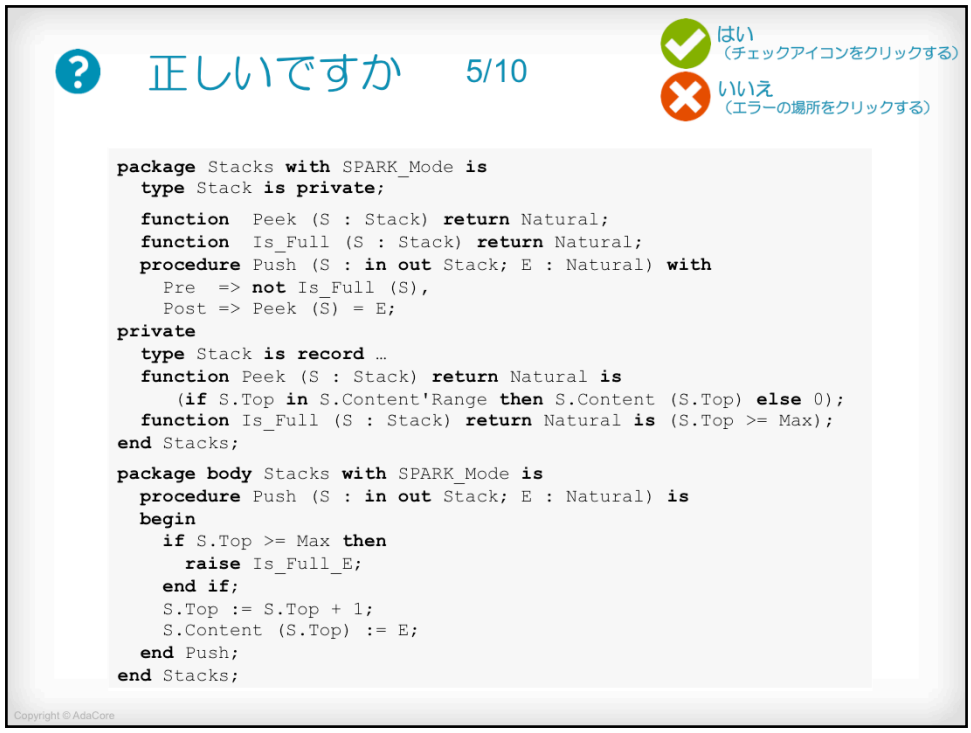
正しいですか 4/10

いいえ

```
package Stacks with SPARK_Mode is
  type Stack is private;
  function  Peek (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
end Stacks;
```

```
package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      raise Is_Full_E;
    end if;
    S.Top := S.Top + 1;
    S.Content (S.Top) := E;
  end Push;
end Stacks;
```

GNATprove は，正常終了へのパスのみを考えるため，Push の事後条件を検証することができます．しかし，Is_Full_E 例外が，実行時に送出され，エラーとなる可能性があるため，警告を出します．


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）
package Stacks with SPARK_Mode is
type Stack is private;
function Peek (S : Stack) return Natural;
function Is_Full (S : Stack) return Natural;
procedure Push (S : in out Stack; E : Natural) with
Pre => not Is_Full (S),
Post => Peek (S) = E;
private
type Stack is record …
function Peek (S : Stack) return Natural is
(if S.Top in S.Content'Range then S.Content (S.Top) else 0);
function Is_Full (S : Stack) return Natural is (S.Top >= Max);
end Stacks;
package body Stacks with SPARK_Mode is
procedure Push (S : in out Stack; E : Natural) is
begin
if S.Top >= Max then
raise Is_Full_E;
end if;
S.Top := S.Top + 1;
S.Content (S.Top) := E;
end Push;
end Stacks;
Copyright © AdaCore

正しいですか 5/10

はい

```
package Stacks with SPARK_Mode is
  type Stack is private;

  function  Peek (S : Stack) return Natural;
  function  Is_Full (S : Stack) return Natural;
  procedure Push (S : in out Stack; E : Natural) with
    Pre  => not Is_Full (S),
    Post => Peek (S) = E;
private
  type Stack is record …
  function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
  function Is_Full (S : Stack) return Natural is (S.Top >= Max);
end Stacks;
```

```
package body Stacks with SPARK_Mode is
  procedure Push (S : in out Stack; E : Natural) is
  begin
    if S.Top >= Max then
      raise Is_Full_E;
    end if;
    …
```

事前条件により，ここでは，GNATprove は，Is_Full_E例外が実行時に送出されることはないと，検証します．

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）

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  with
    Pre => Memory (First) + Offset in Memory'Range
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

正しいけれど，GNATproveでは検証されません．GNATprove は Read_One を自身で分析します．特定の場合に，事前条件中でオーバーフローが生じるかもしれないと云うことを警告します．

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）

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  with
    Pre => Memory (First) <= Memory'Last - Offset
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```

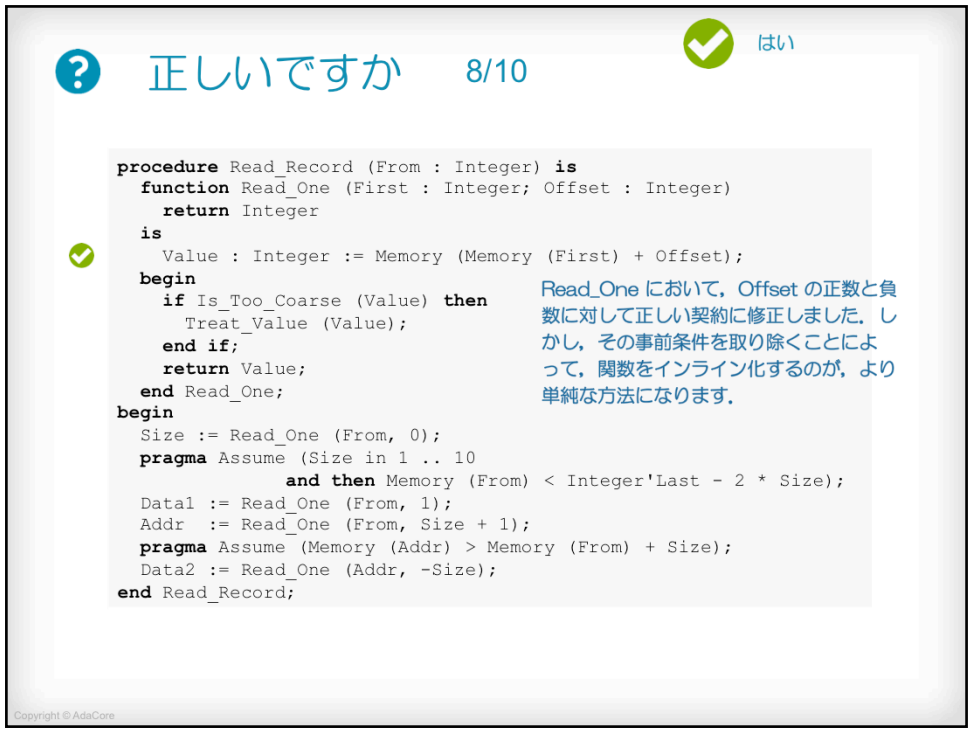
残念なことに，Read_Oneの事前条件を正しくしようとする試みは失敗しています．例えば，Memory(First) が Integer'Last であったり Offset が負数であった場合に実行時にオーバーフローが生じます．

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）

```
procedure Read_Record (From : Integer) is
  function Read_One (First : Integer; Offset : Integer)
    return Integer
  is
    Value : Integer := Memory (Memory (First) + Offset);
  begin
    if Is_Too_Coarse (Value) then
      Treat_Value (Value);
    end if;
    return Value;
  end Read_One;
begin
  Size := Read_One (From, 0);
  pragma Assume (Size in 1 .. 10
                 and then Memory (From) < Integer'Last - 2 * Size);
  Data1 := Read_One (From, 1);
  Addr  := Read_One (From, Size + 1);
  pragma Assume (Memory (Addr) > Memory (From) + Size);
  Data2 := Read_One (Addr, -Size);
end Read_Record;
```



Read_One において，Offset の正数と負数に対して正しい契約に修正しました．しかし，その事前条件を取り除くことによって，関数をインライン化するのが，より単純な方法になります．

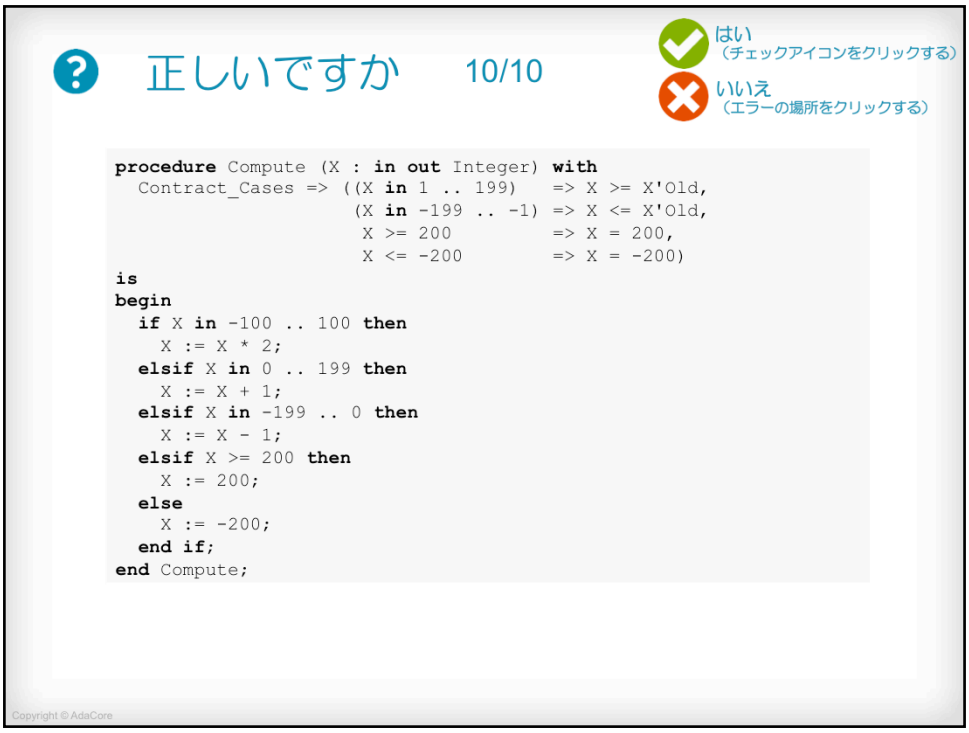
正しいですか 9/10

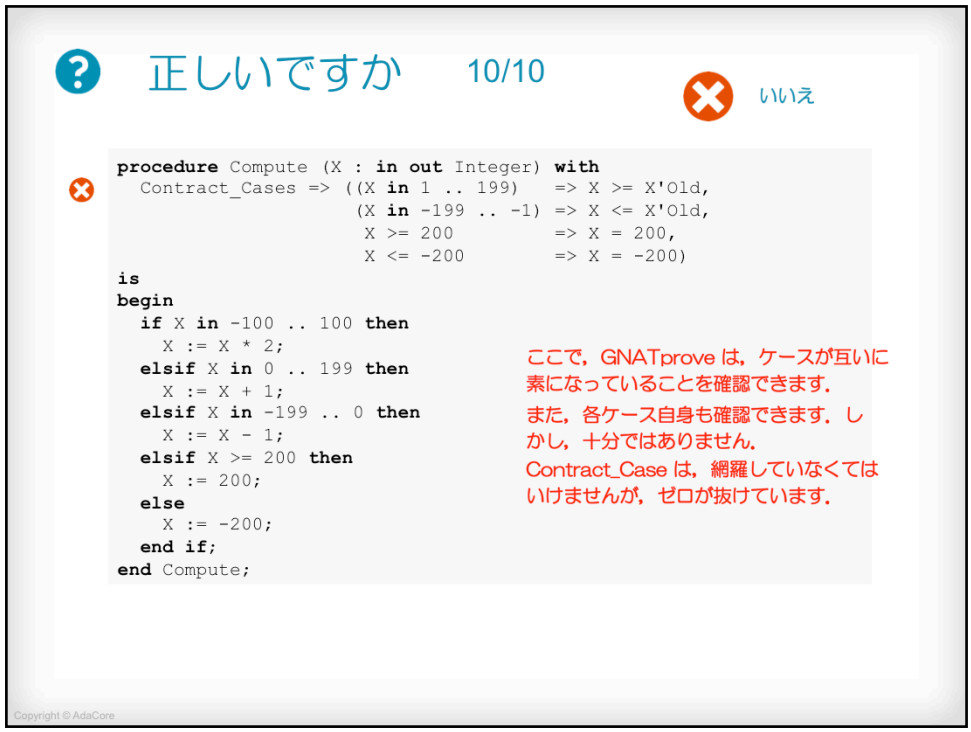
はい
（チェックアイコンをクリックする）

いいえ
（エラーの場所をクリックする）

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
                     (X in 0 .. 199)    => X = X'Old + 1,
                     (X in -199 .. 0)   => X = X'Old - 1,
                      X >= 200          => X = 200,
                      others            => X = -200)
is
begin
  if X in -100 .. 100 then
    X := X * 2;
  elsif X in 0 .. 199 then
    X := X + 1;
  elsif X in -199 .. 0 then
    X := X - 1;
  elsif X >= 200 then
    X := 200;
  else
    X := -200;
  end if;
end Compute;
```

手続き Compute の契約中で，中身に重複があります．これは，ケース文のように互いに素になるべきである Contract_Cases の意味規則の点からは正しくありません．


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 Compute (X : in out Integer) with
Contract_Cases => ((X in 1 .. 199) => X >= X'Old,
(X in -199 .. -1) => X <= X'Old,
X >= 200 => X = 200,
X <= -200 => X = -200)
is
begin
if X in -100 .. 100 then
X := X * 2;
elsif X in 0 .. 199 then
X := X + 1;
elsif X in -199 .. 0 then
X := X - 1;
elsif X >= 200 then
X := 200;
else
X := -200;
end if;
end Compute;
Copyright © AdaCore



ここで，GNATprove は，ケースが互いに素になっていることを確認できます．
また，各ケース自身も確認できます．しかし，十分ではありません．Contract_Case は，網羅していなくてはいけませんが，ゼロが抜けています．


ADACORE
university.adacore.com
Copyright © AdaCore