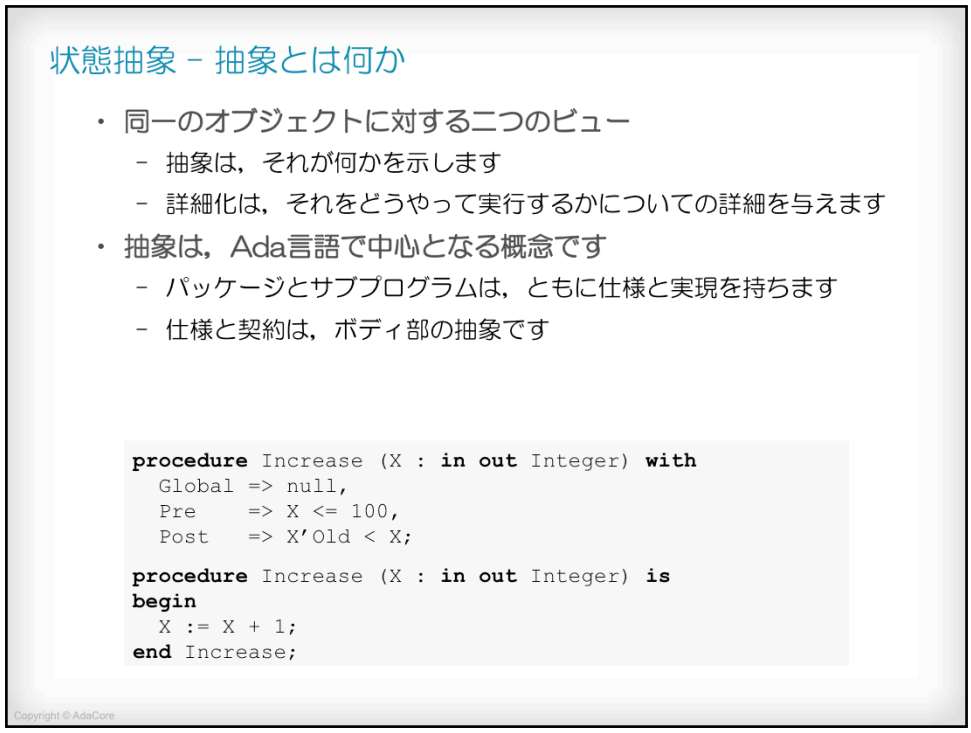
抽象は，プログラミングにおいて重要な概念です．実装とコードの保守を劇的に単純化するからです．

それは，特に，SPARK と モジュール分析に最も適しています．

このコースでは，状態抽象とは何か，SPARK で如何に記述できるかを説明します．

情報フローとプログラムプロパティの証明という点から，GNATprove の分析にどのような影響があるかを明らかにします．



抽象原則は，プログラミング言語において，共通に用いられています．それは，同一のオブジェクトに対して二つのビューを持つことに帰着します．即ち，抽象と詳細化です．

抽象ビューというのは，通常仕様と呼ばれます．対象を粗く記述します．サブプログラム仕様は，通常，それがどのように呼ばれるか，いくつパラメータを持っているか，どういう型からなるか…ということを説明します．同様に，何をするか，返り値は何か，そのパラメータの一つを更新する…

Ada 2012 で採用されている契約ベースのプログラムでは，契約をサブプログラム仕様に追加します．契約は，より詳細にサブプログラムのふるまいを記述するために用います．サブプログラムが実際にどのように動作するかの詳細は，その詳細化 (refinement) ビューが受け持ちます．即ち，実装です．

ここにあるコードを見てみましょう．サブプログラム Increase の仕様は，100以下の整数型の変数であるべき単一の引数によって呼び出されるべきであることを示しています．これによって，厳密に引数の値を増やすことを保証します．

（訳注）詳細化と訳した原語は，refinement．詳細化の他に洗練と訳す場合も多いが，「練り上げて，磨き上げて」という意味はなく，ここでは，詳細化を訳語としてとっています

状態抽象 – 抽象はなぜ役立つのか

- 仕様は，ユーザが依拠しているものを要約します
  - ある抽象の実装は，他の実装に依拠すべきではありません
- 抽象は，実装と検証を単純にします
  - ふるまいを知ることのみが，その抽象の利用者にとって必要なことです
- 抽象は，保守とコードの再利用を容易にします
  - オブジェクトの実装への変更は，そのオブジェクトのユーザに影響しません

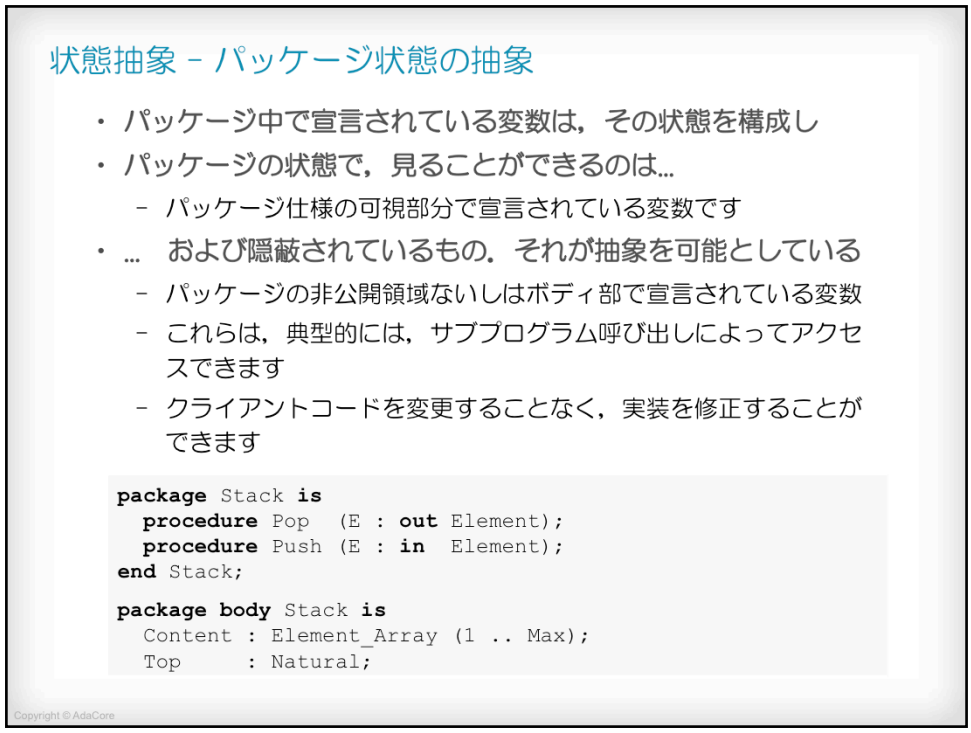
```
procedure Increase (X : in out Integer) with
  Global => null,
  Pre    => X <= 100,
  Post   => X'Old < X;

while X <= 100 loop      --  The loop will terminate
  Increase (X);          --  Increase can be called safely
end loop;
pragma Assert (X = 101); --  Will this hold ?
```


サブプログラム実装の良い抽象を得るために，その仕様は，オブジェクトの利用者が何に依拠することができるかを正確に要約すべきです．言い換えると，ユーザコードは，仕様部に記述がないにも関わらず，オブジェクトの実現が持つふるまいに依拠すべきではありません．

例えば，Increase を呼び出す人は，常に引数の値を確実にふやすということを仮定できます．ここに示したユーザコードの一部において，ループが必ず終了することになっている，ということを意味します．また，Increase の実装は，ループ中で呼ばれたときに実行時エラーを生じないことも仮定しています．一方で，Increase の実装が変わったら，表明は維持できなくなる可能性があります．

基本原則に従うことで，抽象によって多くの利益を得ることができます．まず，抽象によって，プログラムの実装と検証が容易になります．オブジェクトを使用するときに，その仕様を理解するだけで十分であることはよくあります．さらに，保守とコードの再利用が簡単になります．オブジェクトの実装が，このオブジェクトを使用するコードに影響しないからです．



サブプログラムは，抽象からメリットを得る唯一の対象ではありません．パッケージの状態，即ち，その中で定義されている永続的な変数の組は，外部ユーザからは隠蔽されています．

この抽象の形式は，状態抽象と呼ばれ，ボディ部の変数やパッケージの密閉部における変数の定義により構成しています．これら変数には，サブプログラム呼び出しを通じてのみアクセスすることができます．例えば，ここに示す Stack パッケージは，Pop と Push 手続きを用いて修正可能である Stack オブジェクトに対する抽象です．

配列を用いて実装しているという事実は，パッケージのユーザには無関係であり，ユーザのコードに影響を与えることなく，変更することができます．

（訳注1）private は，Ada 言語において伝統的には「密閉」と訳されてきました（今でも，JIS X に残っています）．しかし，Ada 以後普及してきた言語の流儀に従うと少し重い訳語であり，ここでは，単に非公開としています．

（訳注2） Ada言語はオブジェクト指向を意識した言語として，最初に国際規格になっています．オブジェクト指向に代表的な概念に，カプセル化と情報隠蔽があります．仕様部とボディ部の分離は前者が相当します．ここでの非公開領域・隠蔽は，後者の情報隠蔽に関係しています．

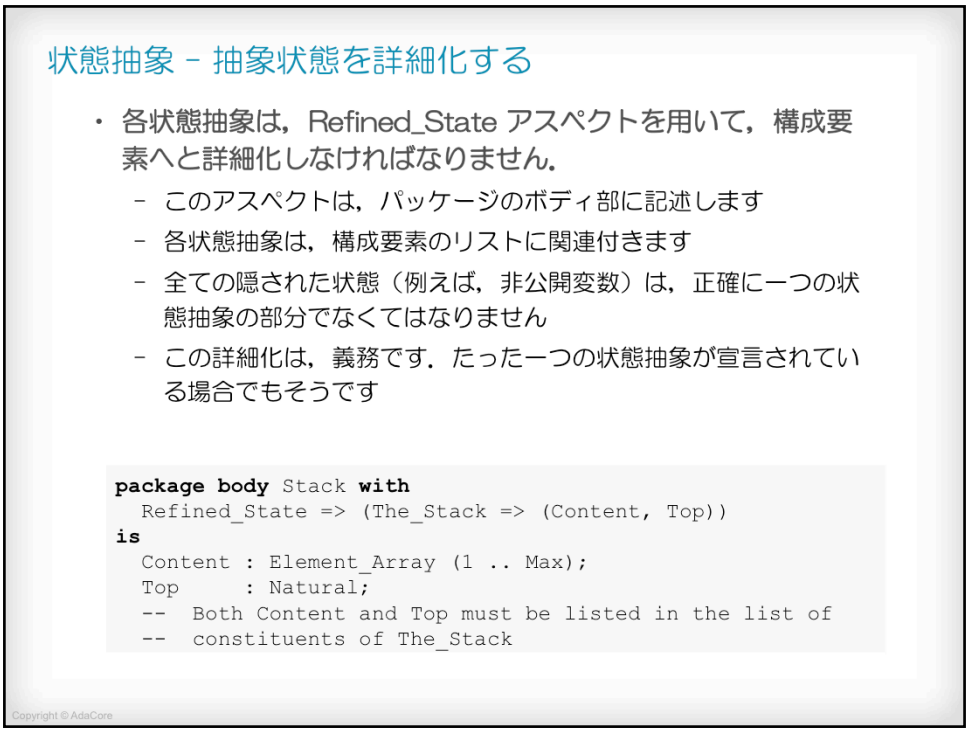
隠された状態は，プログラムのふるまいに影響を与えるので，SPARK では，宣言することができます．

このために，名前付き状態抽象を，Abstract_Stateアスペクトを用いて導入することができます．

これは，隠れた状態を持つパッケージにとっても，必須ではありません．幾つかの状態抽象が，単一のパッケージの隠れた抽象に対して，或いは，全く隠れた状態を持たないパッケージに対して導入されています．ただし，SPARK は別名付けを認めないので，異なる状態抽象は具体的な変数の互いに素な組を常に参照しなくてはならないことに注意してください．

また状態抽象は，変数ではないことに注意してください．それは，型を持たないし，（ボディ部ないしは契約中の）式の内部で使用することができません．

Stack パッケージの例において，最初のコード片のようにパッケージの全ての隠れた状態に対する状態抽象を定義するか，二番目のように，各隠れた変数毎に定義するかは，選択できます．



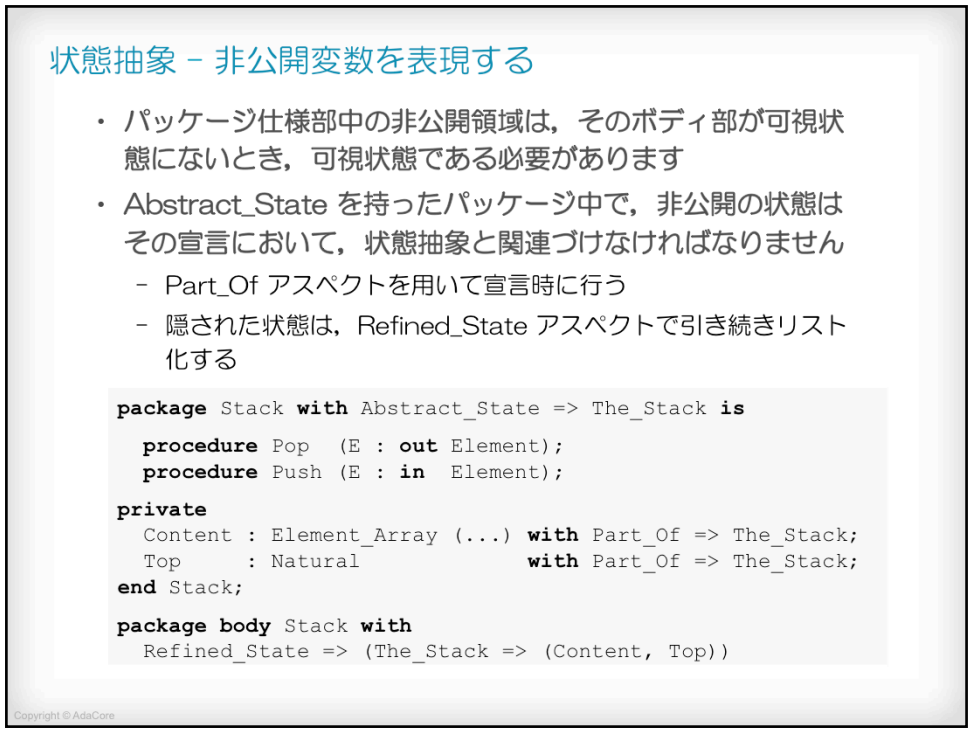
パッケージに対して，一度抽象状態が定義されたら，Refined_State アスペクトを用いて，構成要素へと詳細化することになります．

Refined_State アスペクトは，詳細化前のパッケージがボディ部を必要としない場合でも，パッケージのボディ部に置かなくてはなりません．

パッケージに対して宣言された各状態抽象に対して，詳細化した状態は，この状態抽象によって表現される変数の組をリスト化します．

もし，状態抽象が，あるパッケージに対して記述された場合，全ての隠れた変数が状態抽象の部分に含まれなくてはならない，という意味で完全である必要があります．

例えば，Stack のパッケージボディ部上で，The_Stack 状態抽象にリンクした Refined_State アスペクトを加えなければなりません．パッケージの隠された全ての状態を加える必要があるためです．Content と Top が対象になります．



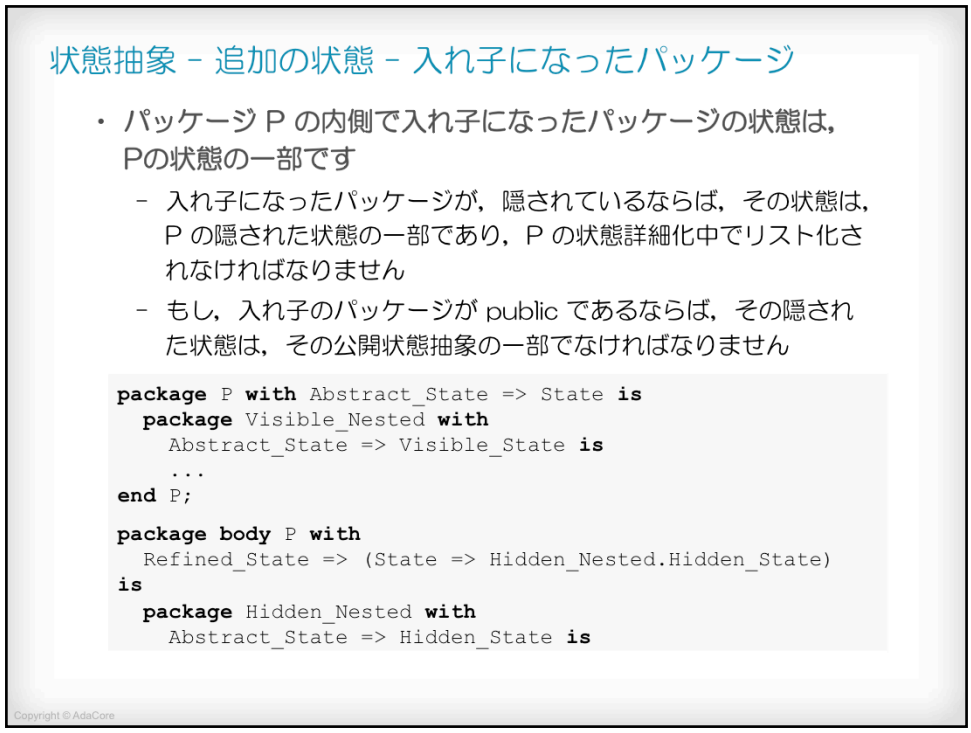
状態抽象は，全ての変数が可視状態であるパッケージのボディ部で常に詳細化されます．

パッケージの仕様部のみが利用可能であるとき，どの状態抽象に非公開変数が属しているかを記述する必要があります．

これは，変数の宣言において，Part_Of アスペクトを用いることで行われます．Part_Of 注釈は必須です：もし，パッケージが抽象状態注釈を持っているならば，非公開領域で定義された全ての隠された状態は，状態抽象とリンクされなければなりません．

例えば，もし，Stack のボディ部ではなく，（仕様部の）非公開領域で Content と Top を定義することを選択した場合，その宣言に Part_Of を付け加える必要があります．また，Stack で定義されている唯一の状態抽象であったとしても，状態抽象 The_Stack と関連づけます．

この記述を行っても，Stack のボディ部中で，Refined_State アスペクト中でリスト化することは必要です．

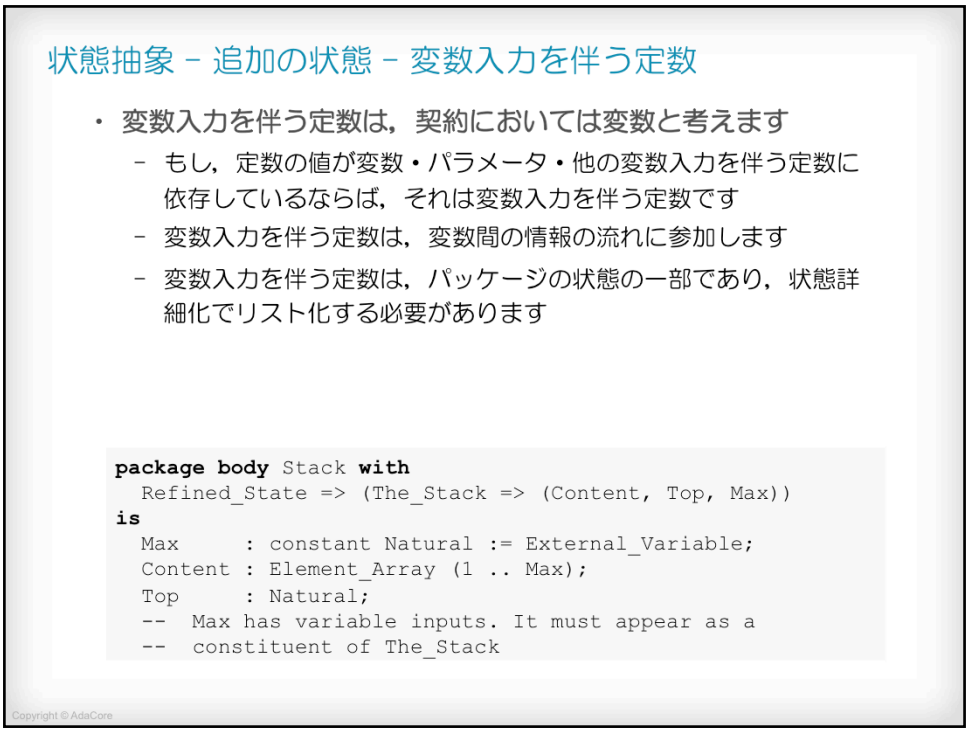


これまで，隠された変数についてのみ説明してきました．しかし，変数のみが，パッケージ状態の唯一の構成要素ではありません．

もし，パッケージ P が入れ子のパッケージを含んでいて，入れ子のパッケージの状態が，P の状態の一部だとします．結果として，もし入れ子のパッケージが隠された状態にあるならば，その状態は，P の隠された状態の一部であり，P の状態詳細化においてリスト化されないといけません．

例では，このことを示しています．Hidden_Nested パッケージの隠された状態が，P の隠された状態の一部となっています．

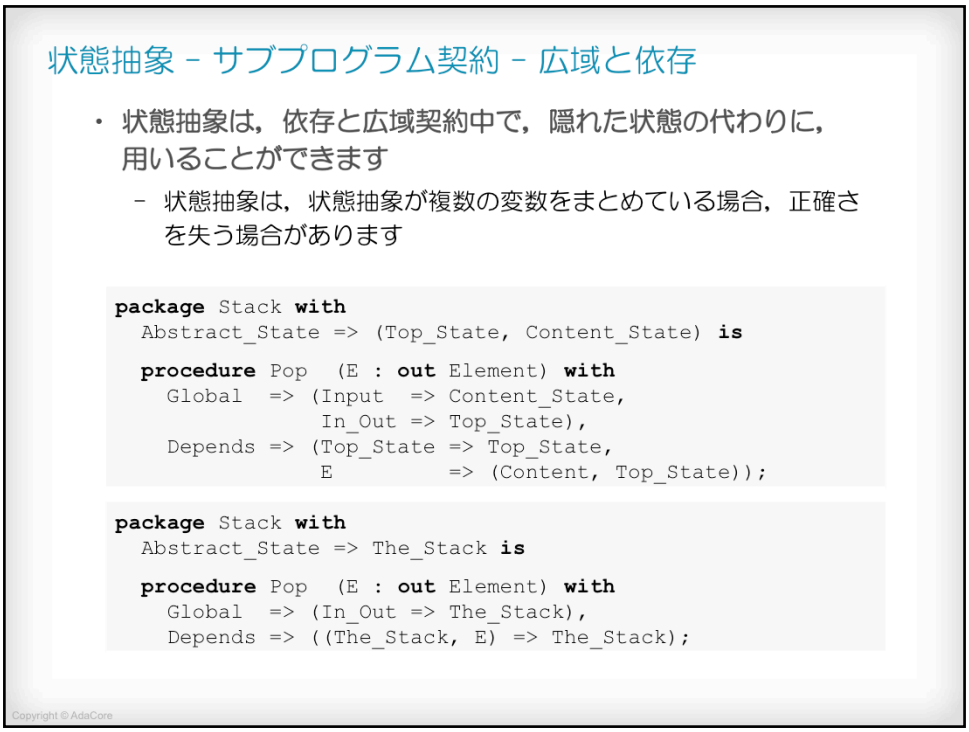
Hidden_Nested の可視状態は，P の隠された状態でもあることに注意して下さい．また，もし P が可視状態にある入れ子のパッケージを含んでいるならば，入れ子のパッケージの状態は，P の隠された状態の部分ではありません．特に，その隠された状態はそれ自身の宣言で，別々状態抽象で宣言されるべきです．Visible_Nested にその例があります．



状態抽象の他の可能な構成要素は，変数入力を伴う定数である．変数入力のある定数とは，その値が変数ないしはサブプログラムパラメータに依存する定数を指しています．これらはフロー解析中では，通常変数として扱われます．プログラムの中で，変数間の情報の流れに参加しているからです．このように，変数入力を伴う定数は，変数のようであり，パッケージ状態の一部と考えることができます．

もし，状態抽象が，パッケージとして記述され，変数入力を伴う定数が，パッケージ中で宣言されているならば，状態抽象の詳細化の中で記述されるべきです．一方で，変数入力を伴わない定数は，情報の流れには参加せず，状態詳細化には現れることはできないことに注意が必要です．

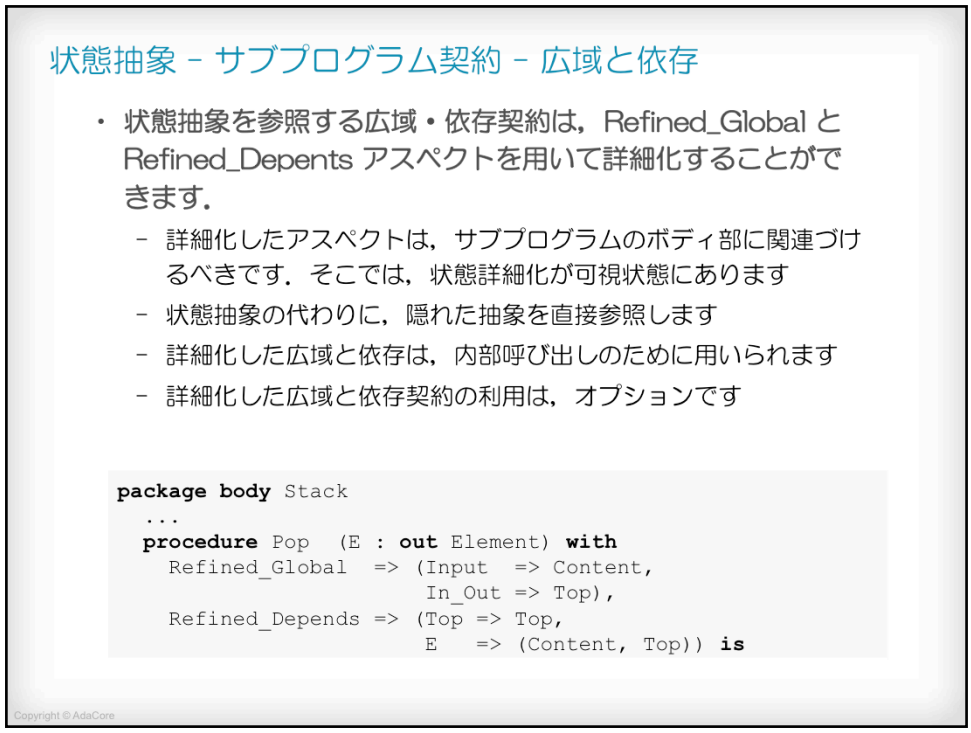
例では，要素の最大値を示す Max は，スタック内で値を保存でき，外部のパッケージから来る変数によって，初期化されます．変数入力なので，Max は，The_Stack の状態抽象の一部でなくてはなりません．



隠れた変数は，サブプログラム呼び出しを通じてアクセスするしかないので，サブプログラム契約は，状態抽象がプログラムの実行中に，如何に更新されるかを文書化することが大切になります．

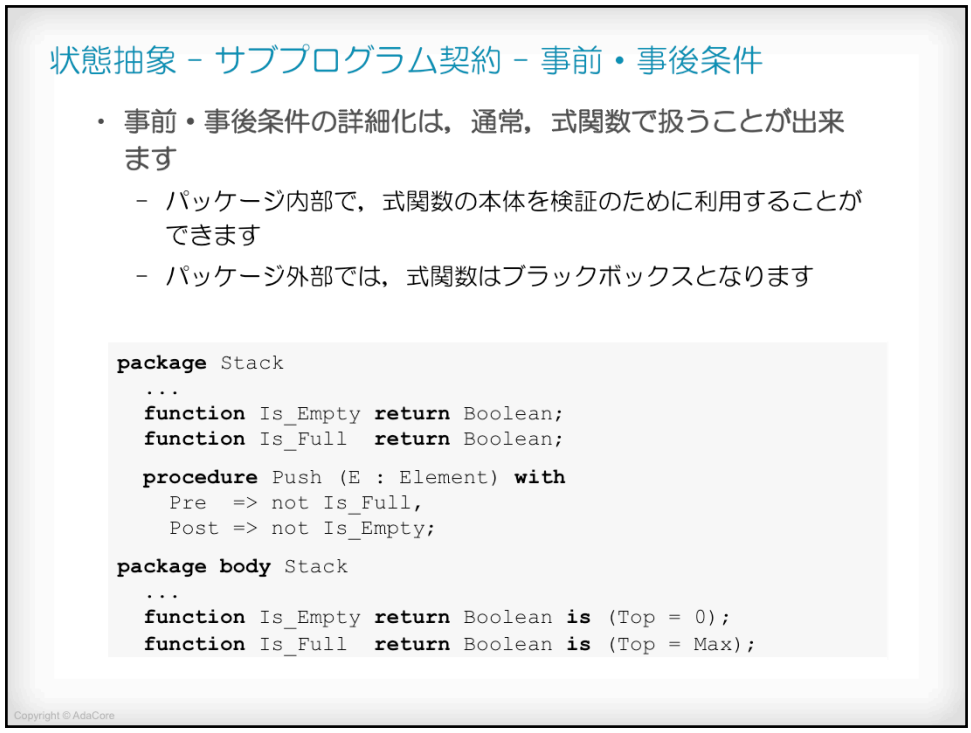
最初に，サブプログラムによってアクセスされる状態抽象が何であるか，異なった変数間で値はどのように流れるかについて，広域契約と依存契約に記述が必要です．状態抽象を参照している広域契約と依存契約は，可視できる変数を参照する契約と比べると，正確さに欠けている場合があります．状態変数に集約された隠れた変数の異なるモードが，単一のモードにまとめられる場合があるからです．

例えば，Pop 手続きは，隠れた変数である Top の値を更新し，Content の値を変えません．もし，二つの状態抽象を，二つの変数で表現すると，この契約は維持されます．一方で，もし，たった一つの状態抽象にまとめられると，Content は値を保持し，The_Stack が更新されるのみといった情報を失うことになります．ここでは，正確さを失っても，問題にはなりません．それが抽象の本質です．しかし，ユーザは，注釈が無意味にならないように，無関係な隠れた状態がまとめられないように注意しなくてはなりません．



もし，全体として，状態抽象を扱う不正確な契約が，パッケージの利用者にとって完全に筋の通ったものだとしても，広域契約と依存契約は，パッケージのボディ部内側で，可能な限り正確になるようにすべきです．この理由から，SPARK は，詳細化契約の記法を導入しています．

これらは，正確な契約です．サブプログラムのボディ部に記述し，状態詳細化は可視です．これらの契約は，通常の広域契約・依存契約とまった同じように見えますが，パッケージの隠れた状態を直接に参照している点で異なります．パッケージのボディ部で，サブプログラムが呼ばれると，これらの詳細化された契約が用いられます．詳細化前のものではありません．従って，検証を可能な限り正確にすることができます．詳細化広域と詳細化依存は，オプションです．もしユーザが記載しなければ，ツールは，パッケージの実装を検査するために，詳細化広域と詳細化依存に相当するものを計算します．



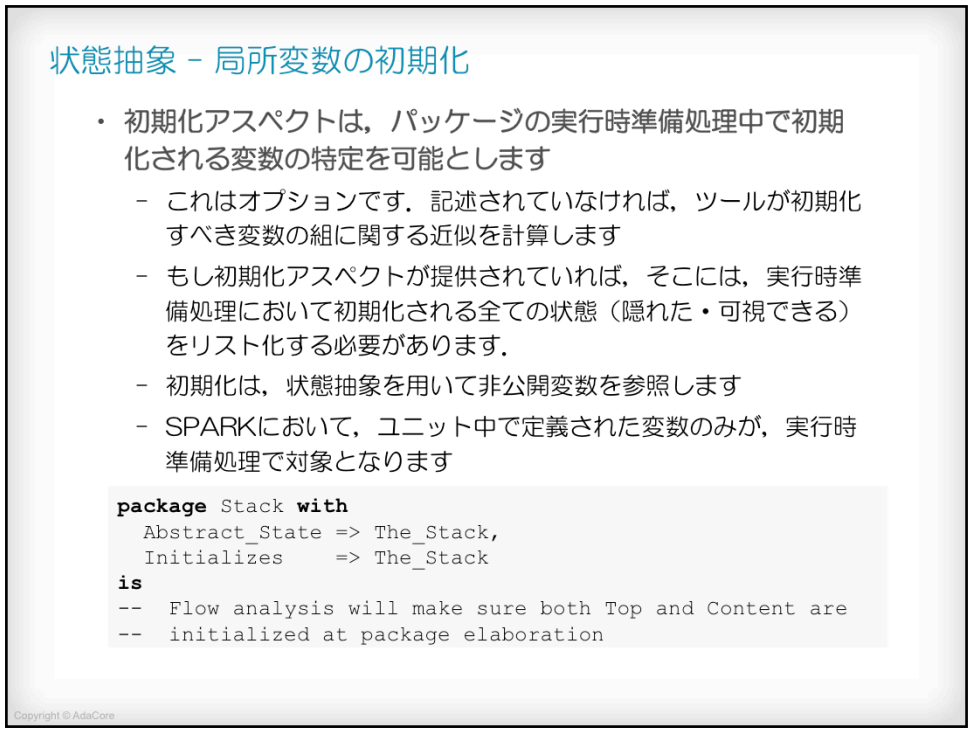
サブプログラムの機能的プロパティは，通常，事前条件と事後条件を用いて記述します．これら契約は，ブール式なので，直接に状態抽象を参照することができません．

この制限に対する処置として，隠れた変数の値を調べるために関数が定義されます．これらの関数は，他のサブプログラム契約における状態抽象の代わりに用いることができます．これが，例で記述していることです．スタックの状態にアクセスする二つの関数を定義しています．Is_Empty と Is_Full です．これによって，手続き Push を記述しています．

広域契約と依存契約に関しては，隠れた変数が可視状態にあるとき，機能的契約のより正確なビューを持つことは有益です．これは式関数によって実現することができます．式関数の本体は，GNATprove において契約のように働くので，実装が可視状態にあるとき，契約のより正確なバージョンを自動的に得ることができます．

抽象を超えてよりパッケージの実装を保証するために，より制約を加えた契約によって検証する必要性があるとします．これは，Refined_Post アスペクトを用いて実現することができます．

このアスペクトは，サブプログラムのボディ部に置かれ，サブプログラムの内部的な呼び出しをより強く保証するために用いることができます．使用する場合，詳細化した事後条件は，サブプログラムの事後条件を含んでいる必要があります．これは，GNATprove によってチェックされます．GNATprove は，実際サブプログラムのボディ部に含まれているときでさえ，詳細事後条件が弱すぎる場合，事後条件が成立しなかったと報告します．SPARK は事前条件に対して，同様の記法を持っていないことに注意して下さい．

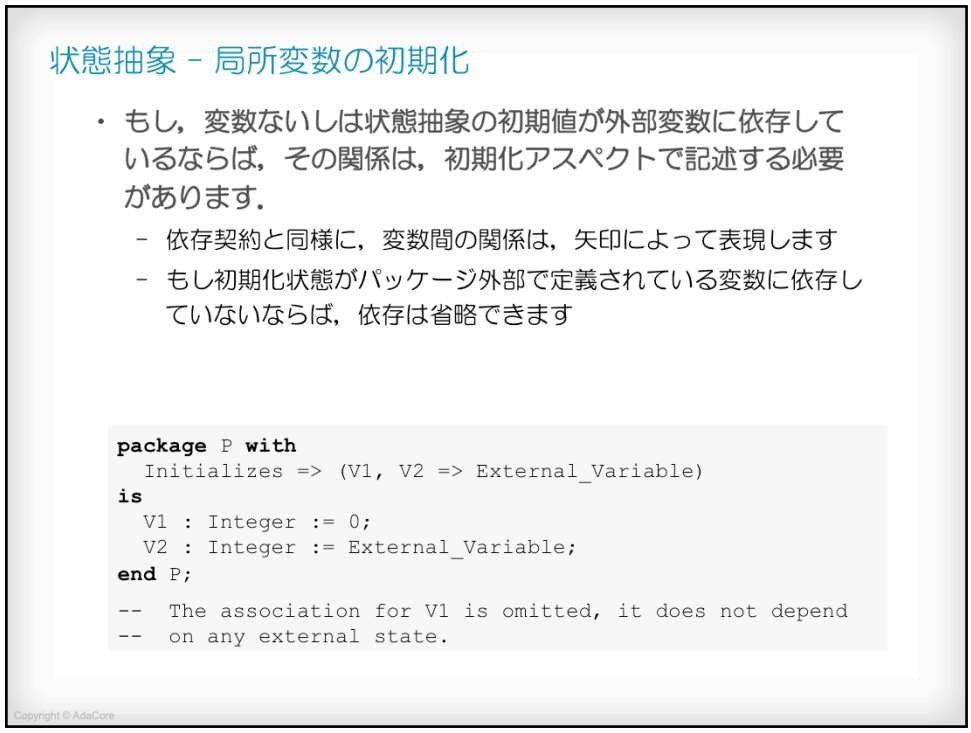


フロー分析の一部として，GNATprove は変数の初期化が適切に行われているかを検査します．従って，フロー分析は，パッケージの実行時準備処理において，初期化された変数がどれであるかを知る必要があります．初期化アスペクトを，パッケージの可視変数の組と状態抽象を特定するために用います．これらは，実行時準備処理において，初期化されます．

初期化アスペクトは，ユニット中で定義されていない変数を参照することはできないことに注意して下さい．SPARK 2014 では，パッケージは，パッケージ内部で宣言されるやいなや変数の初期化のみを行うからです．

初期化アスペクトは，オプションです．ユーザが記述しなければ，GNATproveが計算します．

（訳注） 実行時準備処理 = elaboration，実行（execution）の一部．宣言，宣言部等の処理を行う



フロー分析は，変数間の依存を検査することができるので，状態の初期化を通して，情報の流れが分かります．初期化アスペクトは，また，この目的のためにも使用されます．変数ないしは状態抽象の初期値が，可視変数の値に依存している，他のパッケージの状態抽象に依存しているならば，この依存は，初期化契約でリスト化する必要があります．変数の初期値が依存するエンティティのリストは，矢印を用いて変数と関連づけます．

ここでの例では，P の初期化アスペクトにおいて，V2 の初期値が External_Variable の値に依存していることを示しています．（初期化アスペクトとは異なり）V1 はその初期値が外部の変数には依存していないことに注意して下さい．従って，この依存は，明示的には，次の記述になります：V1 => null

初期値の依存性は，初期化アスペクトが与えられない場合，ツールによって計算できます．一方で，もし初期化アスペクトがあるパッケージに与えられた場合，完全であるべきです．即ち，パッケージの全ての初期化状態はリスト化される必要があります．外部依存についても同様です．


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 Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer is
    Capacity : constant Natural := External_Variable;
    ...
  end Ring_Buffer;

  ...
end Communication;

package body Communication with
  Refined_State => (State => Ring_Buffer.Capacity) is
...
```

正しいですか 1/10

いいえ

```
package Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer is
    Capacity : constant Natural := External_Variable;
    ...
  end Ring_Buffer;

  ...
end Communication;

package body Communication with
  Refined_State => (State => Ring_Buffer.Capacity) is
...
```

ここで，Capacity は，Communication の非公開部で定義されています．従って，Part_Of アスペクトを用いて，宣言時に State とリンクすべきです．

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 Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer with
    Abstract_State => (B_State with Part_Of => State)
  is
    ...
  private
    Capacity : constant Natural := External_Variable with
      Part_Of => B_State;
    ...
  end Ring_Buffer;

  ...
end Communication;
```

正しいですか 2/10

はい

```
package Communication with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
  ...
private
  package Ring_Buffer with
    Abstract_State => (B_State with Part_Of => State)
  is
    ...
  private
    Capacity : constant Natural := External_Variable with
      Part_Of => B_State;
    ...
  end Ring_Buffer;

  ...
end Communication;
```

このプログラムは正しい．GNATprove は検証可能です．

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 Counting with Abstract_State => State is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
   ...
end Main;
```

正しいですか 3/10

はい

```
package Counting with Abstract_State => State is
   ...
end Counting;

package body Counting with
   Refined_State => (State => (Black_Counter, Red_Counter))
is
   Black_Counter, Red_Counter : Natural;

   procedure Reset_Black_Count is
   begin
      Black_Counter := 0;
   end Reset_Black_Count;
   ...
end Counting;

procedure Main is
begin
    Reset_Black_Count;
    ...
```

Reset_Black_Count に対して生成される広域契約は，(In_Out => State) です．

このプログラムは，初期化されていないデータを読みません．しかし，GNATprove はこの事実の検証に失敗します．状態抽象で説明したように，フロー分析は，状態抽象の点から，サブプログラムの影響を計算します．従って，手続き Reset_Black_Count は State の値を読むと見なします．

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 Counting is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
  ...
end Counting;

package body Counting is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
   ...
end Main;
```

正しいですか 4/10

はい

```
package Counting is
  procedure Reset_Black_Count;
  procedure Reset_Red_Count;
  ...
end Counting;

package body Counting is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count is
  begin
    Black_Counter := 0;
  end Reset_Black_Count;
  ...
end Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
   ...
end Main;
```

ここでは，状態抽象は示されていません．GNATprove は，変数の点から推測を行い，問題なくデータの初期化を証明します．

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 Counting with Abstract_State => State is
  procedure Reset_Black_Count with Global => (In_Out => State);
  procedure Reset_Red_Count   with Global => (In_Out => State);
  procedure Reset_All         with Global => (Output => State);
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count with
    Refined_Global => (Output => Black_Counter) is ...

  procedure Reset_Red_Count with
    Refined_Global => (Output => Red_Counter) is ...

  procedure Reset_All is
  begin
    Reset_Black_Count;
    Reset_Red_Count;
  end Reset_All;
end Counting;
```

正しいですか 5/10

はい

```
package Counting with Abstract_State => State is
  procedure Reset_Black_Count with Global => (In_Out => State);
  procedure Reset_Red_Count   with Global => (In_Out => State);
  procedure Reset_All         with Global => (Output => State);
  ...
end Counting;

package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
  Black_Counter, Red_Counter : Natural;

  procedure Reset_Black_Count with
    Refined_Global => (Output => Black_Counter) is ...

  procedure Reset_Red_Count with
    Refined_Global => (Output => Red_Counter) is ...

  procedure Reset_All is
  begin
    Reset_Black_Count;
    Reset_Red_Count;
```

フロー分析は，内部呼び出しに対して，広域契約の詳細化バージョンを利用します．
Rest_Allが，適切に State を初期化することを検証することができます．
Refined_Global と Global 注釈は，必須ではありません．ツールによって計算することもできます．

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）

```
package Stack with Abstract_State => The_Stack is
  pragma Unevaluated_Use_Of_Old (Allow);

  type Element_Array is array (Positive range <>) of Element;
  Max : constant Natural := 100;
  subtype Length_Type is Natural range 0 .. Max;

  procedure Push (E : Element) with
    Post => not Is_Empty and
    (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);

  function Peek     return Element with Pre => not Is_Empty;
  function Is_Full  return Boolean;
  function Is_Empty return Boolean;
end Stack;

package body Stack with
  Refined_State => (The_Stack => (Top, Content)) is
  Top     : Length_Type := 0;
  Content : Element_Array (1 .. Max);

  procedure Push (E : Element) is ...;
  function  Peek     return Element is (Content (Top));
  function  Is_Full  return Boolean is (Top >= Max);
  function  Is_Empty return Boolean is (Top = 0);
end Stack;
```

正しいですか 6/10

いいえ

```
package Stack with Abstract_State => The_Stack is
  pragma Unevaluated_Use_Of_Old (Allow);

  type Element_Array is array (Positive range <>) of Element;
  Max : constant Natural := 100;
  subtype Length_Type is Natural range 0 .. Max;

  procedure Push (E : Element) with
    Post => not Is_Empty and
    (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);

  function Peek     return Element with Pre => not Is_Empty;
  function Is_Full  return Boolean;
  function Is_Empty return Boolean;
end Stack;

package body Stack with
  Refined_State => (The_Stack => (Top, Content)) is
  Top     : Length_Type := 0;
  Content : Element_Array (1 .. Max);

  procedure Push (E : Element) is ...;
  ...
```

Push の事後条件に，コンパイルエラーがあります．実際，The_Stack は状態抽象であり，変数ではありません．式中で使用することはできません．

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）

```
package Stack with Abstract_State => The_Stack is
   ...
   type Stack_Model is private;

   procedure Push (E : Element) with
     Post => not Is_Empty and
     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

   function Peek      return Element with Pre => not Is_Empty;
   function Is_Full   return Boolean;
   function Is_Empty  return Boolean;
   function Get_Stack return Stack_Model;
private
   type Stack_Model is record
     Top     : Length_Type := 0;
     Content : Element_Array (1 .. Max);
   end record;
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
    F : Element := Peek;
begin
    Push (E);
    pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

正しいですか 7/10

はい

```
package Stack with Abstract_State => The_Stack is
  ...
  type Stack_Model is private;

  procedure Push (E : Element) with
    Post => not Is_Empty and
    (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

  function Peek      return Element with Pre => not Is_Empty;
  function Is_Full   return Boolean;
  function Is_Empty  return Boolean;
  function Get_Stack return Stack_Model;
private
  …
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
   F : Element := Peek;
begin
   Push (E);
   pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

このプログラムは正しい．しかし，GNATprove は，Use_Stack 中の表明を検証することができません．実際，Get_Stack が式関数であっても，そのボディ部は，Stack のボディ部外では，不可視になります．

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）

```
package Stack with Abstract_State => The_Stack is
   ...
   procedure Push (E : Element) with
     Post => not Is_Empty and
     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
   ...
private
   ...
   Top     : Length_Type := 0          with Part_Of => The_Stack;
   Content : Element_Array (1 .. Max) with Part_Of => The_Stack;

   function Peek      return Element     is (Content (Top));
   function Is_Full   return Boolean     is (Top >= Max);
   function Is_Empty  return Boolean     is (Top = 0);
   function Get_Stack return Stack_Model is ((Top, Content));
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
    F : Element := Peek;
begin
    Push (E);
    pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

正しいですか 8/10

はい

```
package Stack with Abstract_State => The_Stack is
   ...
   procedure Push (E : Element) with
     Post => not Is_Empty and
     (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);
   ...
private
   ...
   Top     : Length_Type := 0          with Part_Of => The_Stack;
   Content : Element_Array (1 .. Max) with Part_Of => The_Stack;

   function Peek      return Element     is (Content (Top));
   function Is_Full   return Boolean     is (Top >= Max);
   function Is_Empty  return Boolean     is (Top = 0);
   function Get_Stack return Stack_Model is ((Top, Content));
end Stack;

procedure Use_Stack (E : Element) with Pre => not Is_Empty is
    F : Element := Peek;
begin
    Push (E);
    pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

GNATproveは，Use_Stackの表明を検証することができます．
Get_Stackのボディ部への可視性を持っているからです．

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）

```
package External_Interface with
  Abstract_State => File_System,
  Initializes    => File_System
is
  procedure Read_Data (File_Name : String; Data : out Data_Record)
  with Global => File_System;
end External_Interface;

package Data with Initializes => (Data_1, Data_2, Data_3) is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    ...
```

正しいですか 9/10

いいえ

```
package External_Interface with
  Abstract_State => File_System,
  Initializes    => File_System
is
  procedure Read_Data (File_Name : String; Data : out Data_Record)
  with Global => File_System;
end External_Interface;
```

```
package Data with Initializes => (Data_1, Data_2, Data_3) is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
```

Data_1 の初期値と File_System 間の依存は，Data の初期化アスペクトでリスト化されるべきです．

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）

```
package Data is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    ...
  end;
end Data;

procedure Use_Data is
  X : Data_Type_1 := Data_1;
begin
  ...
```

正しいですか 10/10

はい

```
package Data is
  pragma Elaborate_Body;
  Data_1 : Data_Type_1;
  Data_2 : Data_Type_2;
  ...
end Data;

pragma Elaborate_All (External_Interface);
package body Data is
begin
  declare
    Data_Read : Data_Record;
  begin
    Read_Data ("data_file_name", Data_Read);
    Data_1 := Data_Read.Field_1;
    ...
  end;
end Data;

procedure Use_Data is
  X : Data_Type_1 := Data_1;
begin
  ...
```

Data は初期化アスペクトを持っていないので，GNATprove は，その実行時準備処理中で，初期化された変数の組を計算します．その結果，Data_1が常に Use_Data 内で初期化されることを保証することができます．


ADACORE
university.adacore.com
Copyright © AdaCore