CS_313 High Integrity Systems/ CS_M13 Critical Systems

Course Notes

Additional Material

Chapter 2: SPARK Ada

Anton Setzer
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/∼csetzer/lectures/critsys/11/index.html

October 20, 2011

Variant Records

- Variant record means that we have a record, s.t. the type of one field depends on the value of some other type.
- Example:

```
type Gender is (Male, Female);

type Person(Sex: Gender:= Female) is
  record
    Birth: Date;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Children: Integer;
    end case;
  end record;
```

Variant Records

- In the above example the type Gender is defined as a type having two elements, namely Male and Female.
- Person is a type, which has a field Sex, Birth, and depending on the field Sex either a field Bearded or a field Children.
- By default, Person.Gender = Female.
- We can have elements of type Person and of type Person(Male).
  - If John: Person(Male), then John.Sex=Male.

Variant Records

- Whether the field of a variant record accessed is in the variant used **cannot** always **be checked at compile time**.
  - For instance, if we have

        a: Person ,

    a code which accesses

        a.Bearded

    compiles, even if it is clear that

        a.Sex=Female .
  - But this will cause a run time error.
  - In case of

        a: Person(Female) ,

    a warning is issued at compile time if

        a.Bearded

    is accessed.

Example

```
(For simplicity Date = Integer)
John: Person(Male);
Tom : Person;

begin
  John:= (Male,1963,False);
  -- John.Gender:= Female; -- would cause compile error
  Tom:= (Male, 1965,False);
  Tom.Children := 5; -- Compiles okay but runtime error.
  -- Tom.Sex := Female; -- would cause compile error
```

Variant Records

- Variant records are a restricted form of **dependent types** (see module on interactive theorem proving).
  - In **dependent type theory,** as introduced there, such kind of constructs can be used in a **type safe way.**

Object Orientation in Ada

- Object orientation in Ada consists of
  - Tagged types,
  - Class-wide types with dynamic dispath.

Tagged Types

- Record types can be extended.
  - But only if they had been declared to be **tagged**.
  - Tagged means that each variable is associated with a tag which identifies which type it belongs to.
  - This is necessary in case we have a class-wide type (see below) to decide which instance of a function is used.
    - We might define a function which takes an element of one type and a function which takes as argument an element of an extended type.

Example

```
type Student is tagged
  record
    StudentNumber   :   Integer;
    Age             :   Integer;
  end record;

type Swansea_Student is new Student with null record;
--  We extend Student but without adding a new component
--  We could have extended it by a new field as well.
```

Example

- Swansea_Student is a subtype of Student.
- Any function having as argument Student can be applied to a Swansea_Student as well.
- We can override a function for Student by a function with argument Swansea_Student.
- Note that which function to be chosen can be decided at compile time, since it only depends on the (fixed) type of the argument.

Class-Wide Types

- Associated with a tagged type such as Student above is as well a Class-wide type.
  - Denoted by Student'Class.
- An element of Swansea_Student is not an element of Student, but can be converted into an element of Student as follows
  A : Swansea_Student := ...
  B : Student = Student(A);
- However an element of Swansea_Student is an element of Student'Class:
  C : Student'Class = A;

Dynamic Dispatch

- Assume an element A : Student'Class
- Assume a function
  function f (X : Student) return ..
- Assume this function is overridden for Swansea_Student:
  function f (X : Swansea_Student) return ..
  - Without this function the function
    function f (X : Student) return ..
    would be applicable to X : Swansea_Student as well.
    Since it is overriden, the new function is the one to be applied.

Dynamic Dispatch

- We can apply f to A : Student'Class.
  - If A was originally an element of Student, the first version of the function is applied.
  - If A was originally an element of Swansea_Student, the second version of the function is applied.
  - At compile time it is usually not known, which of the two cases applies, therefore the decision which function to choose depends on the **tag** of *A*.
    - The tag tells which type it originally belongs to.
  - This is called **dynamic dispatch** or **late binding**.

Class-Wide Types and Java/C++

- In Java we could say we have only class-wide types.
- In C++ we have as well only class-wide types, but one can control subtyping by using the keyword **virtual**:
  - Only **virtual** methods have late binding.
  - Only virtual methods can be overridden.

Class-Wide Types

- Problem of inheritance: properties are inherited remotely, which makes it difficult to verify programs.
  - If one has a class-wide type A with subtype B, and two different functions f(x:A) and f(x:B), then one
    - might expect that a call of f(a) for a:A refers to the first definition,
    - but in fact, if a:B it will refer to the second definition.
    - That redefinition could have been done by a different programmer in a different area of the code.
- However elements of a subtype in the sense of the restriction of the range of a type (e.g. Integer restricted to 0 . . . 20) can be assigned to elements of the full type.

Object-Orientation in Ada

- Ada's concept of object-orientation is restricted.
  - Ada allows only to form record types, and class-wide types.
  - So instead of
    - having a method f of a class C with parameters x1:A1 ,. . ., xn:An, and then writing O.f(x1 ,. . ., xn) for a method call for object O: C,
    - one has to introduce a polymorphic function f with arguments X: C'Class,x1:A1 ,. . ., xn:An, and then to write f(O,x1 ,. . ., xn) for the call of this function.

Object-Orientation in Ada

- **Disadvantage:** The definition of the functions can be defined completely separted from the definition of the class.
- **Advantage:** More flexibility since one doesn't have to decide for a function, to which object it belongs to.

No Additional Material

For this subsection no additional material has been added yet.

SPARK Ada Concepts

Details about restrictions on subtyping in SPARK Ada.

- No derived types (essentially a new name for an existing type or a subrange for an existing type).
- No type extension (extension of a record by adding further components).
- No class-wide types (see slides on object-orientation in Subsection a). Therefore no **late binding** (dynamic dispatch, called dynamic dispatching in Ada).

No Additional Material

For this subsection no additional material has been added yet.

No Additional Material

For this subsection no additional material has been added yet.

No Additional Material

For this subsection no additional material has been added yet.

No Additional Material

For this subsection no additional material has been added yet.