

32. Type constraints

This clause describes how to use type constraints to restrict the declared type of a field to one of its **like** or **when** subtypes for a given context. A constraint prefixed with the **type** modifier is both (a) enforced by the generator (like a regular constraint) and (b) presupposed at compile time for purposes of type checking. From the generation point of view, the specified type is not associated with the subtype yet at the **pre_generate()** phase and thus the **pre_generate()** method of this type is not called. However, unlike a regular constraint, this does not cause a compile-time type mismatch error.

Example

<<Remove this example? --AW >>

In this example, two struct types have a fixed correlation between their **when** subtypes:

```
type firm: [HONDA, FORD, MERCEDES];

struct engine {
    const manufacturer: firm;
};

struct car {
    const manufacturer: firm;
    my_engine: engine;

    // this constraint declares subtype correlation
    keep type my_engine.manufacturer == manufacturer;
};
```

Now consider the following code:

```
extend FORD engine {
    change_plugs() is {
        //...
    };
};

extend FORD car {
    fix() is {
        my_engine.change_plugs();
    };
};
```

Even though field **my_engine** is declared to be of type **engine**, in the context of method **fix()** you can access method **change_plugs()** (declared under **FORD engine**). Without the **type** modifier, this expression would cause a compile time error.

32.1 keep type

Purpose	Refine the type of a field to one of its subtypes for the specified context												
Category	Struct member												
Syntax	<pre> keep type [me.]<i>const-field-name</i> is a <i>type</i> keep type [me.]<i>const-field-name</i>.<i>property-name</i> == [me.]<i>my-property-name</i> keep for each [(<i>item-name</i>)] in <i>list-field-name</i> { ... type <i>item-name</i> is a <i>type</i>; ... } keep for each [(<i>item-name</i>)] in <i>list-field-name</i> { ... type <i>item-name</i>.<i>property-name</i> == [me.]<i>my-property-name</i>; ... } </pre>												
Parameters	<table border="1"> <tr> <td><i>field-name</i></td> <td>The name of a struct field in the enclosing struct.</td> </tr> <tr> <td><i>type</i></td> <td>The name of a struct or unit type.</td> </tr> <tr> <td><i>property-name</i></td> <td>The name of an enumerated or Boolean field.</td> </tr> <tr> <td><i>my-property-name</i></td> <td>The name of a field of the same type as the <i>property-name</i> in this constraint.</td> </tr> <tr> <td><i>item-name</i></td> <td>An optional name used as a local variable referring to the current item in the list. The default is it.</td> </tr> <tr> <td><i>list-field-name</i></td> <td>The name of a field of typelist of struct (or unit) in the enclosing struct.</td> </tr> </table>	<i>field-name</i>	The name of a struct field in the enclosing struct.	<i>type</i>	The name of a struct or unit type.	<i>property-name</i>	The name of an enumerated or Boolean field.	<i>my-property-name</i>	The name of a field of the same type as the <i>property-name</i> in this constraint.	<i>item-name</i>	An optional name used as a local variable referring to the current item in the list. The default is it .	<i>list-field-name</i>	The name of a field of typelist of struct (or unit) in the enclosing struct.
<i>field-name</i>	The name of a struct field in the enclosing struct.												
<i>type</i>	The name of a struct or unit type.												
<i>property-name</i>	The name of an enumerated or Boolean field.												
<i>my-property-name</i>	The name of a field of the same type as the <i>property-name</i> in this constraint.												
<i>item-name</i>	An optional name used as a local variable referring to the current item in the list. The default is it .												
<i>list-field-name</i>	The name of a field of typelist of struct (or unit) in the enclosing struct.												

You can put a type constraint either on a field of a struct type or on a list field of a struct type. The declaration is similar to a regular constraint inside a **keep** struct member, or, in the list case, inside a **keep for each** construct, with the **type** keyword prefixing the expression.

The **type** keyword is a constraint modifier syntactically analogous to **soft**. However, unlike **soft**, it can modify only specific constraint expressions and can appear only in restricted contexts.

The type correlation can be constant or, when the correlated types are **when** subtypes, variable. The former case is expressed using the **is a** operator. In the latter case the determinant property (the **when** determinant) of the referenced struct is equated to a determinant property of the same type in the declaring struct type.

Typically the static type of a field-access expression is determined according to the type of the field as it was initially declared in the struct type of *instance-expression* (or in one of its supertypes). Type constraints tying the static type of *instance-expression* with a subtype of the field's declared type can change this rule.

If the context in which the field-access occurs requires the subtype, the field-access is automatically down-cast. In this case a runtime check is added to ensure that the casting is justified, and an error is issued if it is not. The runtime check involves a minor overhead, not more than that required by the **as_a()** operator.

Note that

- In the Boolean expression following **type**, operators other than == and **is a** are not allowed. For example, the following is not allowed:

```
keep type TRUE => engine is a FORD engine; // not allowed
```

- The **for each** clause must occur immediately after **keep**. For example, the following is not allowed:

```
keep my_doors.size() > 4 => for each in my_doors { // not allowed
    type it is a small door;
}
```
- Type constraints can equate only constant fields, so the **const** keyword must appear in the declaration of fields involved in equality constraints.
- Type constraints in general affect code from that point onwards. This includes type constraints that appear inside a **for each** clause, in which case other expressions in the same scope after the declaration (but not before it) can assume automatic casting.
- Type constraints cannot appear inside a **gen** action.
- The **soft** keyword cannot be used with type constraints.

Syntax Example

```
keep type f.p1 == p1;
keep for each in lf {
    type it is a B S1;
};
```

Example 1

<<Remove this example? --AW >>

This examples illustrates two of the four forms of allowed syntax.

```
type property_t: [A, B, C];

struct S1 {
    const p1: property_t;
};

struct S2 {
    const p1: property_t;
    f: S1;
    lf: list of S1;
    keep type f.p1 == p1;
    keep for each in lf {
        type it is a B S1;
    };
};

extend sys {
    my_S2: S2;
};
```

Example 2

<<Remove this example? --AW >>

Here is a simple case where two struct types have a fixed correlation between their **when** subtypes:

```
type firm: [HONDA, FORD, MERCEDES];

struct engine {
    const manufacturer: firm;
};

struct car {
    const manufacturer: firm;
```

```
1 my_engine: engine;  
2  
3 // this constraint declares subtype correlation  
4 keep my_engine.manufacturer == manufacturer;  
5  
6 };  
7  
8
```

9 Now consider the following code:

```
10 extend FORD engine {  
11     change_plugs() is {  
12         //...  
13     };  
14 };  
15  
16 extend FORD car {  
17     fix() is {  
18         my_engine.change_plugs();  
19     };  
20 };  
21 };  
22
```

23 Even though in the context of method “fix()”, the field “my_engine” is in fact of type FORD engine (due to
24 the generation constraint), a compile time error is issued on the call to change_plugs(). To fix it, you have to
25 use the **as_a()** operator.
26

27
28 By adding the **type** modifier to the constraint definition above, for example:

```
29     keep type my_engine.manufacturer == manufacturer;  
30
```

31 the field-access in method “fix()” is automatically cast to FORD engine and passes static type checks.
32
33

34 32.2 Type constraints and struct fields

35
36 Automatic casting of a struct-reference field is performed in any context that requires it, including:

- 37 — Struct-member access
- 38 — Assignment
- 39 — Parameter passing

40
41
42
43
44 *Example*

45
46 <<Remove this example? --AW >>
47
48

49 The type “big packet” is correlated with “info cell” on a field called “my_cell”. This is done with the
50 following constraint:

```
51 extend big packet {  
52     keep type my_cell is a info cell;  
53 };  
54
```

55
56 Assume also that field “data” is declared under “info cell”. Here are some examples of automatic casts:

```
57 var bp: big packet;  
58 gen bp;  
59 var lc: info cell = bp.my_cell;  
60 print bp.my_cell.data;  
61 bp.size = small;  
62 print bp.my_cell.data;  
63 var p: packet;  
64 gen p keeping { it.size == big; };
```

```
print p.my_cell.data;
```

32.3 Type constraints and list fields

When the type relation is one-to-many, in other words, when a list field is concerned, automatic casting is applied not to the list itself but to its elements. Automatic casting affects list operators whose result type is the element type, such as indexing (the [] operator) and **pop()**. It also affects the iteration variable inside the **for each** construct, both in procedural and in constraint contexts.

Example 1

<<Remove this example? --AW >>

Here is a complete but simple example with a one-to-many type relation, showing the different contexts of the automatic casting it allows:

```
type color_t: [red, blue];

struct door {
    const color: color_t;
    when blue door {
        is_metallic: bool;
    };
};

struct car {
    const external_color: color_t;
    doors[4]: list of door;
    keep for each (d) in doors {
        type d.color == external_color;
    };
};

// illustrating 4 cases of automatic casting
extend blue car {
    keep for each in doors {
        it.is_metallic => index < 2;
    };

    foo() is {
        print doors[0].is_metallic;

        var bd: blue door = doors.top();

        for each (d) in doors {
            check that d.is_metallic;
        };
    };
};
```

Example 2

<<Remove this example and the xref to it? --AW >>

This example shows that type constraints do not work on lists, but only on their elements. In the extension to “blue car”, the assignment of a “list of door” to a “list of blue door” results in a load time error, despite the type constraint that keeps the color of each door in the “list of door” the same as the external color of the car.

```

1 type color_t: [red, blue];
2 struct door {
3   const color: color_t;
4   when blue door {
5     is_metallic: bool;
6   };
7 };
8
9 struct car {
10  const external_color: color_t;
11  doors[4]: list of door;
12  keep for each (d) in doors {
13    type d.color == external_color;
14  };
15 };
16
17 extend blue car {
18   foo() is {
19     var lc : list of blue door;
20     var c : blue door;
21     //   lc = doors; -- gives load time error
22     c = doors[0];
23   };
24 };
25
26
27

```

32.3.1 Multiple type constraints

A field's type may be restricted by more than one type constraint with respect to different "when" dimensions (determinant fields). Consider the example in . It is possible that, along with the manufacturer property of car and engine, there is another property that determines subtypes:

Example

<<Remove this example? --AW >>

```

37 type fuel_t: [OIL, DIESEL];
38
39
40 extend engine {
41   const fuel: fuel_t;
42 };
43
44 extend car {
45   const fuel: fuel_t;
46   keep type fuel == my_engine.fuel;
47 };
48
49

```

Note that automatic casting is possible for both properties, but not for the conjunction (unless an explicit type constraint allows it). For example, the field of a variable of type HONDA car can be cast to HONDA engine, DIESEL car to DIESEL engine, but FORD OIL car would not be automatically cast to FORD OIL engine.

32.4 Type constraints and like subtypes

Type constraints work just as well for **like** subtypes of the declared type of the field. They apply to the two "is a" forms of the **keep type** struct member.

Note that type constraints with **like** subtypes cannot make the actual **like** type of a generated field dependent on a **when** determinant. In other words, they may not figure under a **when** subtype if they affect a field not declared in the same subtype. This is an error: the constraint is unenforceable.

Example

<<Remove this example? --AW >>

The following simplistic code illustrates the effect of like narrowing type constraint in contrast to a regular constraint, and the restriction described above:

```
struct dog {  
};  
  
struct poodle like dog {  
};  
  
struct person {  
    my_pet: dog;  
};  
  
struct child like person {  
    keep my_pet is a poodle; // constraint is flagged as  
        // having no generatable elements at compile time  
        // and ignored at generation time  
  
    keep type my_pet is a poodle; // constraint is enforced  
  
    size: [SMALL, BIG];  
    when SMALL child {  
        keep type my_pet is a poodle; // error - cannot make  
            // like narrowing depend on when determinant  
  
        my_other_pet: dog;  
        keep type my_other_pet is a poodle; // constraint is enforced  
    };  
};
```

Without the type modifier, `keep my_pet is a poodle` is non-generatable.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65