

12.2.3 Defining Type Constraints

You can 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.

Example

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 typemy_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.

Type constraints are discussed in the following sections:

- [keep type on page 12-32](#)
- [“Type Constraints and Struct Fields” on page 12-36](#)
- [“Type Constraints and List Fields” on page 12-37](#)
- [“Multiple Type Constraints” on page 12-38](#)
- [“Type Constraints and Like Subtypes” on page 12-39](#)

12.2.3.1 **keep type**

Purpose

Refine the type of a field to one of its subtypes for the specified context

Category

Struct member

Syntax

keep type [**me.**]*field-name* **is a type**

keep type [**me.**]*field-name.property-name* **==** [**me.**]*my-property-name*

keep for each [(*item-name*)] **in** *list-field-name* {

...
 type *item-name* **is a type**;

...
 }

keep for each [(*item-name*)] **in** *list-field-name* {

...
 type *item-name.property-name* **==** [**me.**]*my-property-name*;

...
 }

Syntax Example

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

Parameters

<i>field-name</i>	The name of a struct field in the enclosing struct.
<i>type</i>	The name of an enumerated or Boolean 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 [<i>child-</i>] <i>property-name</i> .
<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 list field in the enclosing struct.

Description

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.

Notes

- 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.

Example 1 Example

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

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;
  my_engine: engine;

  // this constraint declares subtype correlation
  keep 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 in the context of method “fix()”, the field “my_engine” is in fact of type FORD engine (due to the generation constraint), a compile time error is issued on the call to change_plugs(). To fix it, you have to use the **as_a()** operator.

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

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

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

12.2.3.2 Type Constraints and Struct Fields

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

- Struct-member access
- Assignment
- Parameter passing

Example 3

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

```
extend big packet {
    keep type my_cell is a info cell;
};
```

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

```
var bp: big packet;
gen bp;
var lc: info cell = bp.my_cell;
print bp.my_cell.data;
bp.size = small;
print bp.my_cell.data;
var p: packet;
gen p keeping { it.size == big; };
```

```
print bp.my_cell.data;
```

12.2.3.3 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 4

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 5

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.

```
<'
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;
  };
};
extend blue car {
  foo() is {
    var lc : list of blue door;
    var c : blue door;
  //   lc = doors; -- gives load time error
    c = doors[0];
  };
};
'>
```

12.2.3.4 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 “[Example 2](#)” on page 12-35. It is possible that, along with the manufacturer property of car and engine, there is another property that determines subtypes:

Example 6

```
type fuel_t: [OIL, DIESEL];

extend engine {
  const fuel: fuel_t;
```

```
};

extend car {
    const fuel: fuel_t;
    keep type fuel == my_engine.fuel;
};
```

Note 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.

12.2.3.5 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.

Notes

- Type constraints with **like** subtypes are supported only by IntelliGen. In Pgen they are treated in the same way as they would be without the **type** modifier (that is, they cause a contradiction at generation time).
- 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. They are flagged as errors at compile time.

Example 7

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
};
};
```

Note Without the type modifier, `keep my_pet is a poodle` is non-generatable. See [Table 5-1 on page 5-7](#) in the *IntelliGen User Guide*.