Towards Inheritance Via Types

- Type inclusion (aka subtyping)
  - A type A is said to be included in, or is a subtype of, another type B when all the values of type A are also values of type B.
  - That is, when A and B are interpreted as sets of values, $A \subseteq B$.
  - Equivalently, the type B is said to be a supertype of type A.
  - We will write this relation as $A \sqsubseteq B$.

Towards Inheritance Via Types

- Type inclusion (aka subtyping)
  - A type A is said to be included in, or is a subtype of, another type B when all the values of type A are also values of type B.
  - That is, when A and B are interpreted as sets of values, $A \subseteq B$.
  - Equivalently, the type B is said to be a supertype of type A.
  - We will write this relation as $A \sqsubseteq B$.
- Examples of subtypes.
  - [Function types]

$$D \to R \sqsubseteq D' \to R' \equiv D' \sqsubseteq D \wedge R \sqsubseteq R'$$

or, $\frac{D' \sqsubseteq D \quad R \sqsubseteq R'}{D \to R \sqsubseteq D' \to R'}$.

  - [Record types]
    $\mathbf{REC}(a_1 : t_1, \dots, a_n : t_n, \dots, a_m : t_m) \sqsubseteq \mathbf{REC}(a_1 : u_1, \dots, a_n : u_n) \equiv \forall i \in [1, n].\, t_i \sqsubseteq u_i$.

Towards Inheritance Via Types

- Type inclusion (aka subtyping)
  - A type A is said to be included in, or is a subtype of, another type B when all the values of type A are also values of type B.
  - That is, when A and B are interpreted as sets of values, $A \subseteq B$.
  - Equivalently, the type B is said to be a supertype of type A.
  - We will write this relation as $A \sqsubseteq B$.
- Examples of subtypes.
  - [Function types]

    $$D \to R \sqsubseteq D' \to R' \equiv D' \sqsubseteq D \wedge R \sqsubseteq R'$$

    or, $\frac{D' \sqsubseteq D \quad R \sqsubseteq R'}{D \to R \sqsubseteq D' \to R'}$.
  - [Record types]
    $\mathbf{REC}(a_1: t_1, \dots, a_n: t_n, \dots, a_m: t_m) \sqsubseteq \mathbf{REC}(a_1: u_1, \dots, a_n: u_n) \equiv \forall i \in [1, n].\, t_i \sqsubseteq u_i$.
- Subtyping on record types corresponds to the concept of inheritance (subclasses) in programming languages.

Name Spaces, Take 4: Classes + Inheritance

- By the typing judgment rule for records, `Point3D` ⊑ `Point2D`.
  - Ignoring field ordering issue.

```
typedef struct _Point2D {
  int x, y;
  int get_x(struct _Point2D*);
  int get_y(struct _Point2D*);
  double dist(struct _Point2D*);
} Point2D;

typedef struct _Point3D {
  int x, y, z;
  int get_x(struct _Point3D*);
  int get_y(struct _Point3D*);
  int get_z(struct _Point3D*);
  double dist(struct _Point3D*);
} Point3D;
```

Name Spaces, Take 4: Classes + Inheritance

- By the typing judgment rule for records, `Point3D` ⊑ `Point2D`.
  - Ignoring field ordering issue.
- We indicate this in the source language with special syntactic sugar.

```
typedef struct _Point2D {
  int x, y;
  int get_x(struct _Point2D*);
  int get_y(struct _Point2D*);
  double dist(struct _Point2D*);
} Point2D;

typedef struct _Point3D {
  int x, y, z;
  int get_x(struct _Point3D*);
  int get_y(struct _Point3D*);
  int get_z(struct _Point3D*);
  double dist(struct _Point3D*);
} Point3D;
```

```
class Point2D {
  int x, y;
  int get_x(void);
  int get_y(void);
  double dist(void);
};

class Point3D extends Point2D {
  int z;
  int get_z(void);
  double dist(void); // overrides
};
```

Name Spaces, Take 4: Classes + Inheritance

- By the typing judgment rule for records, `Point3D` ⊑ `Point2D`.
  - Ignoring field order issue.
- We indicate this in the source language with special syntactic sugar.
  - The members common to `Point2D` and `Point3D` are inherited from the supertype.

```
typedef struct _Point2D {
  int x, y;
  int get_x(struct _Point2D*);
  int get_y(struct _Point2D*);
  double dist(struct _Point2D*);
} Point2D;

typedef struct _Point3D {
  int x, y, z;
  int get_x(struct _Point3D*);
  int get_y(struct _Point3D*);
  int get_z(struct _Point3D*);
  double dist(struct _Point3D*);
} Point3D;
```

```
class Point2D {
  int x, y;
  int get_x(void);
  int get_y(void);
  double dist(void);
};

class Point3D extends Point2D {
  int z;
  int get_z(void);
  double dist(void); // overrides
};
```

Name Spaces, Take 4: Classes + Inheritance

- By the typing judgment rule for records, `Point3D` ⊑ `Point2D`.
  - Ignoring field order issue.
- We indicate this in the source language with special syntactic sugar.
  - The members common to `Point2D` and `Point3D` are inherited from the supertype.
  - Additional (`z`, `get_z`) and overridden members (`dist`) are explicitly indicated in `Point3D`.

```
typedef struct _Point2D {
  int x, y;
  int get_x(struct _Point2D*);
  int get_y(struct _Point2D*);
  double dist(struct _Point2D*);
} Point2D;

typedef struct _Point3D {
  int x, y, z;
  int get_x(struct _Point3D*);
  int get_y(struct _Point3D*);
  int get_z(struct _Point3D*);
  double dist(struct _Point3D*);
} Point3D;
```

```
class Point2D {
  int x, y;
  int get_x(void);
  int get_y(void);
  double dist(void);
};

class Point3D extends Point2D {
  int z;
  int get_z(void);
  double dist(void); // overrides
};
```

Name Spaces, Take 4: Classes + Inheritance

- By the typing judgment rule for records, Point3D ⊑ Point2D.
  - Ignoring field order issue.
- We indicate this in the source language with special syntactic sugar.
  - The members common to Point2D and Point3D are inherited from the supertype.
  - Additional (z, get_z) and overridden members (dist) are explicitly indicated in Point3D.
  - The type signatures of the instance methods now have an *implicit* object reference (this or self) as the first argument.

```
typedef struct _Point2D {
  int x, y;
  int get_x(struct _Point2D*);
  int get_y(struct _Point2D*);
  double dist(struct _Point2D*);
} Point2D;

typedef struct _Point3D {
  int x, y, z;
  int get_x(struct _Point3D*);
  int get_y(struct _Point3D*);
  int get_z(struct _Point3D*);
  double dist(struct _Point3D*);
} Point3D;
```

```
class Point2D {
  int x, y;
  int get_x(void);
  int get_y(void);
  double dist(void);
};

class Point3D extends Point2D {
  int z;
  int get_z(void);
  double dist(void); // overrides
};
```

Run-Time Memory View

1. How are objects and classes laid out in memory?
2. How are names mapped to methods?
3. What are the rules for name visibility?

```
class Point2D {
  int x, y;
  int get_x(void);
  int get_y(void);
  double dist(void);
};
class Point3D extends Point2D {
  int z;
  int get_z(void);
  double dist(void); // overrides
};
Point2D p1, p2; Point3D q = new();
```