Type Definitions in C

A type definition is a synonym for a type expression. In C they are introduced using the `typedef` keyword. Type definitions are useful:

- as abbreviation:
  ```c
  typedef struct { int x; int y; } point_t;
  ```
- to construct recursive types:
  ```c
  struct list {  
    int info;  
    struct list* next;
  }  
  ```

Possible declaration in C:
```c
typedef struct list list_t;
```
Type Definitions in C

The C grammar distinguishes `typedef-name` and `identifier`. Consider the following declarations:

```c
typedef struct { int x, y } point_t;
point_t origin;
```

Relevant C grammar:

- `declaration` → `(declaration-specifier)^+ declarator`;
- `declaration-specifier` → `static | volatile ... typedef`;
  | `void | char | char ... typename`
- `declarator` → `identifier | ...`

**Problem:**
- parser adds `point_t` to the table of types when the `declaration` is reduced
- parser state has at least one look-ahead token
- the scanner has already read `point_t` in line two as `identifier`

Type Definitions in C: Solutions

**Relevant C grammar:**

- `declaration` → `(declaration-specifier)^+ declarator`;
- `declaration-specifier` → `static | volatile ... typedef`;
  | `void | char | char ... typename`
- `declarator` → `identifier | ...`

Solution is difficult:
- try to fix the look-ahead inside the parser
- add a rule to the grammar:

  ```c
  typename → identifier
  ```

Type Definitions in C: Solutions

**Relevant C grammar:**

- `declaration` → `(declaration-specifier)^+ declarator`;
- `declaration-specifier` → `static | volatile ... typedef`;
  | `void | char | char ... typename`
- `declarator` → `identifier | ...`

Solution is difficult:
- try to fix the look-ahead inside the parser
- add a rule to the grammar:

  ```c
  typename → identifier
  ```

- register type name earlier
Chapter 3: Type Checking

Goal of Type Checking

In most mainstream (imperative / object oriented / functional) programming languages, variables and functions have a fixed type. For example: `int, void*, struct { int x; int y; }`.

Types are useful to:
- manage memory
- to avoid certain run-time errors

In imperative and object-oriented programming languages a declaration has to specify a type. The compiler then checks for a type correct use of the declared entity.
Type Expressions

Types are given using type *expressions*. The set of type expressions \( T \) contains:

- **base types**: `int, char, float, void, ...`
- **type constructors** that can be applied to other types

Example for type constructors in C:

- **structures**: `struct { t_1 x_1 \ldots t_k x_k }`
- **pointers**: `t *`
- **arrays**: `[t] n`
  - the size of an array can be specified
  - the variable to be declared is written between \( t \) and \( [n] \)
- **functions**: `[t_1 \ldots t_k] f()`
  - the variable to be declared is written between \( t \) and \( (t_1, \ldots, t_k) \)
  - in ML function types are written as: \( t_1 \ldots \ast t_k \rightarrow t \)

Type Checking

**Problem:**

- **Given**: A set of type declarations \( \Gamma = \{ t_1 x_1 ; \ldots ; t_m x_m ; \} \)
- **Check**: Can an expression \( e \) be given the type \( t \)?

**Example:**

```c
struct list { int info; struct list* next; }; 
int f(struct list* l) { return l; }; 
struct { struct list* c;}* b; 
int* a[11];
```

Consider the expression:

```
+a[f(b->c)]+2;
```
Type Checking using the Syntax Tree

Check the expression \( a \{ f \{ b \to c \} \} + 2 \):

[Diagram of syntax tree with nodes and edges]

**Idea:**
- traverse the syntax tree bottom-up
- for each identifier, we lookup its type in \( \Gamma \)
- constants such as 2 or 0.5 have a fixed type
- the types of the inner nodes of the tree are deduced using typing rules

Type Systems

**Formally:** consider *judgements* of the form:

\[ \Gamma \vdash e : t \]

// (in the type environment \( \Gamma \) the expression \( e \) has type \( t \))

**Axioms:**
- **Const:** \( \Gamma \vdash c : t_c \) (\( t_c \) type of constant \( c \))
- **Var:** \( \Gamma \vdash x : \Gamma(x) \) (\( x \) Variable)

**Rules:**
- **Ref:** \( \frac{\Gamma \vdash e : t}{\Gamma \vdash e \& e : \Gamma(e)} \)
- **Deref:** \( \frac{\Gamma \vdash e : \Gamma(e)}{\Gamma \vdash e^\dagger : \Gamma(e)} \)

Type Systems for C-like Languages

More rules for typing an expression:

**Array:**
\[
\frac{\Gamma \vdash e_1 : t \times \Gamma \vdash e_2 : t}{\Gamma \vdash e_1[e_2] : t}
\]

**Array:**
\[
\frac{\Gamma \vdash e_1 : t \times \Gamma \vdash e_2 : t}{\Gamma \vdash e_1[e_2] : t}
\]

**Struct:**
\[
\frac{\Gamma \vdash e : \text{struct}\{ t_1, a_1; \ldots; t_m, a_m \};}{\Gamma \vdash e.a_i : t_i}
\]

**App:**
\[
\frac{\Gamma \vdash e : t_1, \ldots, t_m \quad \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_m : t_m}{\Gamma \vdash e(e_1, \ldots, e_m) : t}
\]

**Op:**
\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\]

**Explicit Cast:**
\[
\frac{\Gamma \vdash e : t_1 \quad t_1 \text{ can be converted to } t_2}{\Gamma \vdash (t_2) e : t_2}
\]

Example: Type Checking

Given expression \( a \{ f \{ b \to c \} \} + 2 \) and

\( \Gamma = \{\)

\[
\text{struct list ( int info; struct list* next; );}
\]

\[
\text{int f(struct list* l);}\]

\[
\text{struct \{ struct list* c;\} \* c;}
\]

\[
\text{int * c[ll];}
\]

\( \}
\)
Example: Type Checking

Given expression `a[f(b->c)] + 2`:

```
  +
 /   
int  int
   /
int *
 |
int [ ]
 
( )
int

int (struct list *) f

struct { struct list * c; }

struct { struct list * c; } *
```

Equality of Types

Summary of Type Checking

- Choosing which rule to apply at an AST node is determined by the type of the child nodes
- determining the rule requires a check for \( \sim \) equality of types

**type equality** in C:

- \( \sim \) the compiler could re-order the fields of \( A \) and \( B \) independently (not allowed in C)
- to extend an record \( A \) with more fields, it has to be embedded into another record:

```
struct B {
    struct A;
    int field_of_B;
}
```

- after issuing `typedef int C;` the types \( C \) and `int` are the same

Structural Type Equality

Alternative interpretation of type equality (does not hold in C):

*semantically*, two types \( t_1, t_2 \) can be considered as **equal** if they accept the same set of access paths.

Example:

```
struct list {  struct list1 {  int info;
    int info;
    struct list next;
    struct list1 next;
  }
}
```

Consider declarations `struct list * l` and `struct list1 * l`.
Both allow

```
    l->info  l->next->info
```

but the two declarations of \( l \) have unequal types in C.

Algorithm for Testing Structural Equality

Idea:

- track a set of equivalence queries of type expressions
- if two types are syntactically equal, we stop and report success
- otherwise, reduce the equivalence query to a several equivalence queries on (hopefully) simpler type expressions

Suppose that recursive types were introduced using type definitions:

```
typedef A t
```

(we omit the `T`). Then define the following rules:
Rules for Well-Typedness

Example:

```c
typedef struct { int info; A *next; } A
typedef struct { int info; struct { int info; B *next; } *next; } B
```

We ask, for instance, if the following equality holds:

```c
struct { int info; A *next; } = B
```

We construct the following deduction tree:

Example:

```c
typedef struct { int info; A *next; } A
typedef struct { int info; struct { int info; B *next; } *next; } B
```

We ask, for instance, if the following equality holds:

```c
struct { int info; A *next; } = B
```

We construct the following deduction tree:

Proof for the Example:
Implementation

We implement a function that implements the equivalence query for two types by applying the deduction rules:

- if no deduction rule applies, then the two types are **not equal**
- if the deduction rule for expanding a type definition applies, the function is called recursively with a potentially larger type
- in case an equivalence query appears a second time, the types are **equal by definition**

Termination

- the set $D$ of all declared types is finite
- there are no more than $|D|^2$ different equivalence queries
- repeated queries for the same inputs are automatically satisfied
- termination is ensured

Overloading and Coercion

Some operators such as `+` are **overloaded**:

- `+` has several possible types
  - for example: `int` + `(int, int)`, `float` + `(float, float)`
  - but also `float*` + `(float*, int)`, `int*` + `(int, int*)`
- depending on the type, the operator `+` has a different implementation
- determining which implementation should be used is based on the type of the **arguments** only

Coercion: allow the application of `+` to `int` and `float`.

- instead of defining `+` for all possible combinations of types, the arguments are automatically **coerced**
  - conversion is usually done towards more general types i.e. `5 + 0.5` has type `float` (since `float >= int`)
  - coercion may generate code (e.g. converting `int` to `float`
Subtypes

On the arithmetic basic types char, int, long, etc. there exists a rich subtype hierarchy

- $t_1 \leq t_2$ means that the values of type $t_1$
  - form a subset of the values of type $t_2$;
  - can be converted into a value of type $t_2$;
  - fulfill the requirements of type $t_2$;
  - are assignable to variables of type $t_2$.

Example:
assign smaller type (fewer values) to larger type (more values)

$$
\begin{align*}
  t_1 & \rightarrow x; \\
  t_2 & \rightarrow y; \\
  y & = x;
\end{align*}
$$

Rules for Well-Typedness of Subtyping

$\begin{align*}
  t & \leq t' \\
  s & \leq t \text{ s.t. } s = t
\end{align*}$

typedef $A$ $s$

Rules and Examples for Subtyping

- struct $\{s_j, a_{j1}, \ldots, s_j, a_{jm_j}\}$
- struct $\{t_k, a_{k1}, \ldots, t_k, a_{km_k}\}$

Examples:

- struct $\{\text{int } a; \text{ int } b;\}$
- int (int)
- float (float)
- float (int)
Rules and Examples for Subtyping

```
s_0 (s_1, ..., s_m) ⊑ t_0 (t_1, ..., t_m)
```

Examples:

- \(\text{struct} \{ \text{int} a; \text{int} b; \} \subseteq \text{struct} \{ \text{float} a; \}\)
- \(\text{int} (\text{int}) \nsubseteq \text{float} (\text{float})\)
- \(\text{int} (\text{float}) \nsubseteq \text{float} (\text{int})\)

**Definition**

Given two function types in subtype relation

\(s_0 (s_1, ..., s_n) \subseteq t_0 (t_1, ..., t_n)\) then we have

- **co-variance** of the return type \(s_0 \leq t_0\) and
- **contra-variance** of the arguments \(s_i \geq t_i\) for \(1 < i < n\)

Subtypes: Application of Rules (I)

Check if \(S_1 \leq R_1\):

- \(R_1 = \text{struct} \{ \text{int} a; R_1 (R_1) f; \}\)
- \(S_1 = \text{struct} \{ \text{int} a; \text{int} b; S_1 (S_1) f; \}\)
- \(R_2 = \text{struct} \{ \text{int} a; R_2 (S_2) f; \}\)
- \(S_2 = \text{struct} \{ \text{int} a; \text{int} b; S_2 (R_2) f; \}\)

Subtypes: Application of Rules (II)

Check if \(S_2 \leq S_1\):

- \(R_1 = \text{struct} \{ \text{int} a; R_1 (R_1) f; \}\)
- \(S_1 = \text{struct} \{ \text{int} a; \text{int} b; S_1 (S_1) f; \}\)
- \(R_2 = \text{struct} \{ \text{int} a; R_2 (S_2) f; \}\)
- \(S_2 = \text{struct} \{ \text{int} a; \text{int} b; S_2 (R_2) f; \}\)

Subtypes: Application of Rules (III)

Check if \(S_2 \leq R_1\):

- \(R_1 = \text{struct} \{ \text{int} a; R_1 (R_1) f; \}\)
- \(S_1 = \text{struct} \{ \text{int} a; \text{int} b; S_1 (S_1) f; \}\)
- \(R_2 = \text{struct} \{ \text{int} a; R_2 (S_2) f; \}\)
- \(S_2 = \text{struct} \{ \text{int} a; \text{int} b; S_2 (R_2) f; \}\)
Discussion

- for presentational purposes, proof trees are often abbreviated by omitting deductions within the tree
- structural sub-types are very powerful and can be quite intricate to understand
- Java generalizes records to objects/classes where a sub-class $A$ inheriting from base class $O$ is a subtype $A \leq O$
- subtype relations between classes must be explicitly declared
- inheritance ensures that all sub-classes contain all (visible) components of the super class
- a shadowed (overwritten) component in $A$ must have a subtype of the the component in $O$
- Java does not allow argument subtyping for methods since it uses different signatures for overloading