1 Koncepty

W 1987 próbowano projektować szablony z odpowiednimi interfejsami. Chciano by szablony:

- były w pełni ogólne i wyraziste
- by nie wykorzystywały większych zasobów w porównaniu do kodowania ręcznego
- by miały dobrze określone interfejsy

Długo nie dało się osiągnąć tych trzech rzeczy, ale za to osiągnięto:

- *kompletność Turinga*[1]
- lepszą wydajność (w porównaniu do kodu pisanego ręcznie)
- kiepskie interfejsy (praktycznie *typowanie kaczkowe czasu kompilacji*)[2]

Brak dobrze określonych interfejsów prowadzi do spektakularnie złych wiadomości błędów. Dwie pozostałe właściwości uczyniły z szablonów sukces.

Rozwiązanie problemu specyfikacji interfejsu zostało, przez Alexa Stepanova nazwane **konceptami**. **Koncept** to zbiór wymagań argumentów szablonu. Można też go nazwać systemem typów dla szablonów, który obiecuje znacząco ulepszyć diagnostyki błędów i zwiększyć siłę ekspresji, taką jak przeciążanie oparte na konceptach oraz częściowa specjalizacja szablonu funkcji.

Koncepty (*The Concepts TS*[3]) zostały opublikowane i zaimplementowane w wersji 6.1 kompilatora GCC w kwietniu 2016 roku. Fundamentalnie to predykaty czasu kompilacji typów i wartości. Mogą być łączone zwykłymi operatorami logicznymi (`&&`, `||`, `!`)

1.1 Podstawy konceptów

Reprezentacja definicji szablonu w C++ to zazwyczaj drzewa wyprowadzania[4]. Używając identycznych technik kompilatora, możemy przekonwertować

[1] (ang. Turing Completness) umiejętność do rozwiązania każdej odpowiedzi. Program, który jest kompletny według Turinga może być wykorzystany do symulacji jakiejkolwiek 1-taśmowej maszyny Turinga

[2] (ang. duck typing) rozpoznanie typu obiektu, nie na podstawie deklaracji, ale przez badanie metod udostępnionych przez obiekt

[3] (ang. The Concepts Technical Specification) Specyfikacja techniczna konceptów

[4] (ang. Parse Trees)

koncepty do drzew wyprowadzania. Posiadając to możemy zaimplementować sprawdzanie konceptów jako abstrakcyjne drzewo dopasowań[5]. Wygodnym sposobem implementowania takiego dopasowywania jest generowanie i porównywanie zestawów wymaganych funkcji i typów (zwane *zestawami ograniczeń*) z definicji szablonów i konceptów.

Definicja konceptu to zestaw równań *drzewa AST*[6] z założeniami typu. Koncepty dają dwa zamysły:

1. w *definicjach szablonu*, koncepty działają jak reguły osądzania typowania. Jeśli *drzewo AST* zależy od parametrów szablonu i nie może być rozwiązane przez otaczające środowisko typowania, wtedy musi się pojawić w strzegących ciałach konceptów. Takie zależne *drzewa AST* są domniemanymi parametrami konceptów i zostaną rozwiązane przez sprawdzanie konceptów w momentach użycia.

2. w *użyciach szablonów*, koncepty działają jak zestawy predykatów, które argumenty szablonu muszą spełniać. Sprawdzanie konceptów rozwiązuje domniemane parametry w momentach inicjalizacji.

1.2 Ulepszenie programowania generycznego

```
double pierwiastek(double d);
double d = 7;
double d2 = pierwiastek(d);
vector<string> v = {"jeden", "dwa"};
double d3 = pierwiastek(v);
```

Mamy funkcję `pierwiastek`, która jako parametr przyjmuje zmienną typu `double`. Jeśli dostarczymy taki typ, wszystko będzie w porządku, ale jeśli damy inny typ od razu otrzymamy pomocną wiadomość błędu.

```
template<class T>
void sortuj(T &c){
    //kod sortowania
}
```

Kod funkcji `sortuj` zależy od różnych właściwości typu `T`, takiej jak posiadanie operatora []

```
vector<string> v = {"jeden", "dwa"};
sortuj(v);
//OK: zmienna v ma wszystkie syntaktyczne właściwości
wymagane przez funkcję sort
```

[5](ang. Abstract Tree Matching)
[6](ang. Abstract Syntax Tree)

```
double d = 7;
sortuj(d);
//Błąd: zmienna d nie ma operatora []
```

Mamy kilka problemów:

- wiadomosc błędu jest niejednoznaczna i daleko jej do precyzyjnej i pomocnej, tak jak : "Błąd: zmienna d nie ma operatora []"
- aby użyć funkcji `sortuj`, musimy dostarczyć jej definicję, a nie tylko deklaracje. Jest to różnica w sposobie pisania zwykłego kodu i zmienia się model organizowania kodu
- wymagania funkcji dotyczące typu argumentu są domniemane w ciałach ich funkcji
- wiadomość błędu funkcji pojawi się tylko podczas inicjalizacji szablonu, a to może się zdarzyć bardzo długo po momencie wywołania
- Notacja `template<typename T>` jest powtarzalna, bardzo nielubiana.

Używając konceptu, możemy dotrzeć do źródła problemu, poprzez poprawne określanie wymagań argumentów szablonu. Fragment kodu używającego konceptu `Sortable`:

```
void sortuj(Sortable &c);//(1)
vector<string> v = {"jeden", "dwa"};
sortuj(v);//(2)
double d = 7;
sortuj(d);//(3)
```

(1) - akceptuj jakąkolwiek zmienną c, która jest `Sortable`
(2) - OK: `v` jest kontenerem typu `Sortable`
(3) - Błąd: d nie jest `Sortable` (`double` nie dostarcza operatora [], itd.

Kod jest analogiczny do przykładu `pierwiastek`. Jedyna różnica polega na tym, że:

- w przypadku typu `double`, projektant języka wbudował go do kompilatora jak określony typ, gdzie jego znaczenie zostało określone w dokumentacji
- zaś w przypadku `Sortable`, użytkownik określił co on oznacza w kodzie. Typ jest `Sortable` jeśli posiada właściwości `begin()` i `end()` dostarczające losowy dostęp do sekwencji zawierającej elementy, które mogą być porównywane używając operatora `<`

Teraz otrzymujemy bardziej jasny komunikat błędu. Jest on generowany natychmiast w momencie gdzie kompilator widzi błędne wywołanie (`sortuj(d);`)

Cele to zrobienie:

- kodu generycznego tak prostym jak nie-generyczny
- bardziej zaawansowanego kodu generycznego tak łatwym do użycia i nie tak trudnym do pisania

1.3 Używanie konceptów

Koncept to predykat czasu kompilacji (coś co zwraca wartość boolowską). Np. argument typu szablonu `T` mógłby mieć wymagania żeby być:

- iteratorem `Iterator<T>`
- iteratorem losowego dostępu `Random_access_iterator<T>`
- liczbą: `Number<T>`

Notacja `C<T>`, gdzie `C` to koncept a `T` to typ, to wyrażenie znaczące "prawda jeśli `T` spełnia wszystkie wymagania C, a nieprawda w przeciwnym wypadku."

Podobnie, możemy określić, że zestaw argumentów szablonu musi spełniać predykat, np. `Mergeable<In1, In2, Out>`. Taki predykaty wielu typów są niezbędne do opisywania biblioteki STL i wielu innych. Są bardzo ekspresywne i łatwo kompilowalne (tańsze niż obejścia metaprogramowania szablonów). Można oczywiście definiować własne koncepty i można tworzyć biblioteki konceptów. Koncepty pozwalają na przeciążanie i eliminują potrzebę wielokrotnego doraźnego metaprogramowania i kodu *scaffoldingu*[7] z metaprogramowania, co znacznie upraszcza metaprogramowanie, a także programowanie generyczne.

1.4 Określanie interfejsu szablonu

```
template<typename S, typename T>
    requires Sequence<S> &&
    Equality_comparable<Value_type<S>, T>
Iterator_of<S> szukaj(S &seq, const T &value);
```

[7] metaprogramistyczna metoda budowania aplikacji bazodanowych. To technika wspierana przez niektóre frameworki MVC, w których programista może napisać specyfikację opisującą sposób wykorzystania bazy danych aplikacji. Kompilator używa tej specyfikacji, aby wygenerować kod, który aplikacja może wykorzystać do odczytu, tworzenia, aktualizacji i usuwania wpisów bazy danych

Powyższy szablon przyjmuje dwa argumenty typu szablonu. Pierwszy argument typu musi być typu `Sequence` i musimy być w stanie porównywać elementy sekwencji ze zmienną `value` używając operatora `==` (stąd `Equality_comparable<Value_type<S>, T>`). Funkcja `szukaj` przyjmuje sekwencję przez referencję i `value` do znalezienia jako referencję `const`. Zwraca iterator.

Sekwencja musi posiadać `begin()` i `end()`. Koncept `Equality_comparable` jest zaproponowany jako koncept standardowej biblioteki. Wymaga by jego argument dostarczał operatory `==` i `!=`. Ten koncept przyjmuje dwa argumenty. Wiele konceptów przyjmuje więcej niż jeden argument. Koncepty mogą opisywać nie tylko typy, ale również związki między typami.

Użycie funkcji `szukaj`:

```
void test(vector<string> &v, list<double> &list){
    auto a0 = szukaj(v, "test");(1)
    auto p1 = szukaj(v, 0.7);(2)
    auto p2 = szukaj(list, 0.7);(3)
    auto p3 = szukaj(list, "test");(4)

    if(a0 != v.end()){
      //Znaleziono "test"
    }
}
```

1) OK 2) Błąd: nie można porównać string do double 3) OK 4) Błąd: nie można porównać double ze string

1.5 Notacja skrótowa

Gdy chcemy podkreślić, że argument szablonu ma być sekwencją, piszemy:

```
template<typename Seq>
    requires Sequence<Seq>
void algo(Seq &s);
```

To oznacza, że potrzebujemy argumentu typu `Seq`, który musi być typu `Sequence`, lub innymi słowy: Szablon przyjmuje argument typu, który musi być typu `Sequence`. Możemy to uprościć:

```
template<Sequence Seq>
void algo(Seq &s);
```

To znaczy dokładnie to samo co dłuższa wersja, ale jest krótsza i lepiej wygląda. Używamy tej notacji dla konceptów z jednym argumentem. Np.

moglibyśmy uprościć funkcję `szukaj`:

```
template<Sequence S, typename T>
    requires Equality_comparable<Value_type<S>, T>
Iterator_of<S> szukaj(S &seq, const T &value);
```

Upraszcza to składnię języka. Sprawia, że nie jest zbyt zagmatwana.

1.6 Definiowanie konceptów

Koncepty, takie jak `Equality_comparable` często można znaleźć w bibliotekach (np. w `The Ranges TS`), ale koncepty można też definiować samodzielnie:

```
template<typename T>
concept bool Equality_comparable = requires (T a, T b){
    { a == b } -> bool; //(1)
    { a != b } -> bool; //(2)
};
```

Koncept ten został zdefiniowany jako szablonowa zmienna. Typ musi dostarczać operacje `==` i `!=`, z których każda musi zwracać wartość `bool`, żeby być `Equality_comparable` . Wyrażenie `requires` pozwala na bezpośrednie wyrażenie jak typ może być użyty:

- `{ a == b }`, oznajmia, że dwie zmienne typu `T` powinny być porównywalne używając operatora `==`
- `{ a == b} -> bool` mówi że wynik takiego porównania musi być typu `bool`

Wyrażenie `requires` jest właściwie nigdy nie wykonywane. Zamiast tego kompilator patrzy na wymagania i zwraca `true` jeśli się skompilują a `false` jeśli nie. To bardzo potężne ułatwienie.

```
template<typename T>
concept bool Sequence = requires(T t) {
    typename Value_type<T>;
    typename Iterator_of<T>;

    { begin(t) } -> Iterator_of<T>;
    { end(t) } -> Iterator_of<T>;

    requires Input_iterator<Iterator_of<T>>;
    requires Same_type<Value_type<T>,
```

```
    Value_type<Iterator_of<T>>>;
};
```

Żeby być typu `Sequence`:

- typ `T` musi mieć dwa powiązane typy: `Value_type<T>` i `Iterator_of<T>`. Oba typy to zwykłe *aliasy szablonu*[8]. Podanie tych typów w wyrażeniu `requires` oznacza, że typ `T` musi je posiadać żeby być `Sequence`.
- typ `T` musi mieć operacje `begin()` i `end()`, które zwracają odpowiednie iteratory.
- odpowiedni iterator oznacza to, że typ iteratora typu `T` musi być typu `Input_iterator` i typ wartości typu `T` musi być taka sama jak jej wartość typu jej iteratora. `Input_iterator` i `Same_type` to koncepty z biblioteki.

Teraz w końcu możemy napisać koncept `Sortable`. Żeby typ był `Sortable`, powinien być sekwencją oferującą losowy dostęp i posiadać typ wartości, który wspiera porównania używające operatora `<`:

```
template<typename T>
concept bool Sortable = Sequence<T> &&
Random_access_iterator<Iterator_of<T>> &&
Less_than_comparable<Value_type<T>>;
```

`Random_access_iterator` i `Less_than_comparable` są zdefiniowane analogicznie do `Equality_comparable`

Często, wymagane są relacje pomiędzy konceptami. Np. koncept `Equality_comparable` jest zdefiniwoany by wymagał jeden typ. Można zdefiniować ten koncept by radził sobie z dwoma typami:

```
template<typename T, typename U>
concept bool Equality_comparable = requires (T a, U b) {
    { a == b } -> bool;
    { a != b } -> bool;
    { b == a } -> bool;
    { b != a } -> bool;
};
```

To pozwala na porównywanie zmiennych typu `int` z `double` i `string` z `char*`, ale nie `int` z `string`.

[8] ALIAS SZABLONU