0.1 Rozszerzanie algorytmów

Rozwiązaniem jest dodanie kolejnego przeciążenia, które jako parametr przyjmuje kontener asocjacyjny[1].

```
template<Associative_container A,
    Same_as<key_type_t<T>> T>
bool czyIstnieje(const A &assoc, const T &value) {
   return assoc.find(value) != assoc.end();
}
```

Ta wersja funkcji `czyIstnieje()` ma tylko dwa ograniczenia: `A` musi być
typu `Associative_container`, a typ `T` musi być taki sam jak typ klucza `A` (`key_type_t<A>`). W przypadku kontenerów asocjacyjnych po prostu wyszukujemy wartość przy użyciu `find()`, a następnie sprawdzamy, czy znaleźliśmy ją przez porównanie z `end()`. To prawdopodobnie szybsze rozwiązanie niż wyszukiwanie sekwencyjne. W przeciwieństwie do wersji `Sequence`, `T` nie musi być typu `Equality_comparable`. Wynika to z faktu, że dokładne wymagania `T` są określone przez kontener asocjacyjny, a wymogi te są zwykle określane przez osobny komparator lub funkcję haszującą.

Koncept `Associative_container`:

```
template<typename S>
concept bool Associative_container() {
  return Regular<S> && Range<S>() &&
    requires {
      typename key_type_t<S>;
      requires Object_type<key_type_t<S>>;
    } &&
    requires (S s, key_type_t<S> k) {
```

[1](ang. associative container) grupa szablonów klas w standardowej bibliotece *C++*, która implementuje uporządkowane tablice asocjacyjne. Kontenery zdefiniowane w obecnej wersji standardu: set, map, multiset, multimap, unordered set, unordered multiset, unordered map, unordered multimap.

```
        { s.empty() } -> bool;
        { s.size() } -> int;
        { s.find(k) } -> iterator_t<S>;
        { s.count(k) } -> int;
    };
}
```

Kontener asocjacyjny jest typu `Regular`, definiuje `Range` elementów, ma `key_type` (który może różnić się od wartości `value_type`), a także zestaw operacji, w tym `find()`, itd.

Podobnie jak poprzednio w przypadku `Sequence`, nie jest to wyczerpująca lista wymagań dla kontenera asocjacyjnego. Nie dotyczy wstawiania i usuwania, a także wyklucza szczególne wymagania dotyczące iteratorów `const`. Ponadto nie opisaliśmy dokładnie tego, jak oczekujemy, że zachowają się funkcje `size()`, `empty()`, `find()` i `count()`.

Ten koncept dotyczy wszystkich kontenerów asocjacyjnych z biblioteki standardowej *C++* (`set`, `map`, `unordered_multiset`, itp.). Obejmuje również te niestandardowe, zakładając, że narażają interfejs. Na przykład przeciążenie to będzie działało dla wszystkich kontenerów asocjacyjnych typu `Q(QSet<T>, QHash<T>)`.

Aby używać konceptów do rozwijania algorytmów, należy zrozumieć, jak kompilator wybiera pomiędzy wersją `Sequence` a `Associative_container`. Innymi słowy, co się dzieje gdy wywoływana jest funkcja `czyIstnieje()`

```
std::vector<int> v { ... };
std::set<int> s { ... };

if (czyIstnieje(v, 42)) // (1)
    //...
if (czyIstnieje(s, 42)) // (2)
    //...
```

(1) - wywołuje przeciążenie `Sequence`

(2) - wywołuje przeciążenie `Associative_container`

Dla każdego wywołania `czyIstnieje` kompilator określa, która funkcja jest wywoływana na podstawie podanych argumentów. Nazywa się to *rozwiązaniem przeciążenia*[2]. Jest to algorytm, który próbuje znaleźć jedną najlepszą funkcję (wśród jednego lub więcej kandydatów), aby ją wywołać na podstawie podanych argumentów. Oba wywołania funkcji odnoszą się do szablonów, więc kompilator wykonuje dedukcję argumentów szablonu, a potem formuje specjalizację deklaracji w oparciu o wyniki. W obydwu przypadkach dedukcja i zastąpienie powiodą się w zwykły i przewidywalny sposób, dlatego w każdym punkcie wywołania musimy wybrać jedną z dwóch specjalizacji. W tym miejscu ograniczenia wchodzą w grę. Tylko funkcje których ograniczenia są spełnione mogą być wybrane przez rozwiązanie przeciążenia. Aby określić, czy ograniczenia funkcji są spełnione, zastępujemy dedukowane argumenty szablonu powiązanymi ograniczeniami deklaracji szablonu funkcji, a następnie oceniamy wynikowe wyrażenie. Ograniczenia są spełnione, gdy substytucja się powiedzie, a wyrażenie okaże się prawdziwe.

W pierwszym wywołaniu, dedukowane argumenty szablonu to `vector<int>` i `int`. Argumenty te spełniają ograniczenia `Sequence`, ale nie tych `Asociative_container`, ponieważ `vector` nie ma `find()` lub `count()`. Dlatego kandydat `Asociative_container` zostaje odrzucony, pozostawiając tylko kandydata `Sequence`. W drugim wywołaniu, dedukowane argumenty to `set<int>` i `int`. Rozwiązanie jest odwrotne do poprzedniego: `set` nigdy nie jest `Sequence`, ponieważ brakuje mu operacji `front()` i `back()`, tak więc kandydat jest odrzucany, a rozwiązanie przeciążenia wybiera kandydata `Asociative_container`. To działa, ponieważ ograniczenia obu przeciążeń są wystarczająco wyczerpujące, aby zapewnić, że kontener spełnia ograniczenia jednego szablonu lub drugiego, ale nie obu. Sytuacja jest nieco bardziej interesująca, jeśli chcemy dodać więcej przeciążeń tego algorytmu. Możemy rozszerzyć algorytm dla konkretnych typów lub szablonów, tak jak mogliśmy to zrobić bez konceptów. Zasadniczo możemy określić prawidłowe definicje funkcji dla tych typów. Jeśli będziemy mieli szczęście, wiele z tych nowych przeładowań będzie miało identyczne definicje.

Ogólnie rzecz biorąc, możemy kontynuować rozszerzanie definicji algo-

[2](ang. overload resolution)

rytmu generycznego przez dodanie przeciążeń, które różnią się tylko ich ograniczeniami. Są trzy przypadki, które trzeba wziąć pod uwagę podczas przeładowywania z konceptami:

1. Rozszerzać definicję poprzez dostarczenie przeciążenia, które działa dla zupełnie innego zestawu typów. Ograniczenia tych nowych przeładowań byłyby wzajemnie wykluczające lub miałyby minimalną ilość nakładania się na istniejące ograniczenia.

2. Dostarczać zoptymalizowaną wersję istniejącego przeciążenia, specjalizując ją w podzbiorze swoich argumentów. Wymaga to utworzenia nowego przeciążenia, które ma silniejsze ograniczenia niż jego bardziej ogólna forma.

3. Dostarczać uogólnioną wersję, która jest zdefiniowana w kategoriach ograniczeń współużytkowanych przez jedno lub więcej istniejących przeładowań.

Jeśli ograniczenia nie są rozłączne z wieloma kandydatami, mogą być opłacalne. Kompilator musi określić najlepszego kandydata na wywołanie. Jeśli jednak kompilator nie może określić najlepszego kandydata, rozwiązanie jest niejednoznaczne. Gdy w pierwszym algorytmie `czyIstnieje()` zmieni się wymaganie `Sequence` zamiast tylko `Range`. To zminimalizuje ilość nakładania się, a zatem i prawdopodobieństwo dwuznaczności.

Ograniczenia rozłączne nie gwarantują, że połączenie będzie niedwuznaczne. Możemy na przykład spróbować zdefiniować kontener, który spełnia wymagania zarówno `Sequence` i `Associative_container`. W tym przypadku oba przeciążenia byłyby opłacalne, ale przeciążenie nie jest z natury lepsze od innych. Chyba że dodamy nowe przeciążenia, aby dostosować się do tego rodzaju struktury danych, wynik byłby niejednoznacznym rozwiązaniem.

`Sequence` i `Associative_container` tak naprawdę mają pokrywające się ograniczenia. Oba wymagają konceptu `Range`. Możemy rozważyć te przeciążenia jako przykład trzeciego przypadku. To wskazuje, że może istnieć algorytm, który można zdefiniować w odniesieniu do wymagań przecinających. Ale to nie jest takie proste.

Drugi przypadek jest ważną cechą programowania generycznego w języku *C++* i jest podstawą optymalizacji typów w bibliotekach generycznych. Ograniczenie subsumpcji pozwala na optymalizację generycznych algorytmów opartych na interfejsach dostarczonych przez ich argumenty.