0.1 Semantyczne udoskonalanie

W niektórych przypadkach udoskonalenia są czysto semantyczne. Nie dostarczają operacji, które kompilator może wykorzystać do odróżnienia przeciążeń. W rzeczywistości ten problem pojawia się w standardowej hierarchii iteratorów: *iteratory input* i *iteratory forward* dzielą dokładnie te same zestawy operacji.

Pojęciowo *iterator input* jest iteratorem reprezentującym pozycję w strumieniu wejściowym. Ponieważ jest zwiększany, poprzednie elementy są konsumowane. Oznacza to, że wcześniej dostępne elementy nie są już dostępne przez iterator lub dowolną jego kopię. W przeciwieństwie do tego, *iterator forward* nie konsumuje elementów przy zwiększaniu. Wcześniej dostępne elementy mogą być uzyskane dzięki kopiom. Jest to zazwyczaj określane mianem właściwości *multipass*. Jest to czysto semantyczna własność.

```
template<typename I>
concept bool Input_iterator() {
  return Regular<I>() && requires (I i) {
    typename value_type_t<I>;
    { *i } -> value_type_t<I> const&;
    { ++i } -> I&;
  };
}

template<typename I>
concept bool Forward_iterator() {
  return Input_iterator<I>();
}
```

Wszystkie wymagania składniowe są zdefiniowane w koncepcie `Input_iterator`. Koncept `Forward_iterator` zawiera tylko `Input_iterators`. Innymi słowy, zestaw wymagań `Forward_iterator` jest dokładnie taki sam, jak `Input_iterator`. Jeśli próbujemy zdefiniować przeciążenia wymagające tych konceptów, wynik byłby zawsze dwuznaczny (ani lepszy od drugiego). Zróżnicowanie pomiędzy tymi konceptami jest tak naprawdę przydatny. Na

przykład jeden z konstruktorów `vector` ma bardziej wydajną implementację *iteratorów forward* niż dla *iteratorów input*.

```
template<Object_type T, Allocator_of<T> A>
class vector {
  template<Input_iterator I>
    requires Same_as<T, value_type_t<I>>()
  vector(I first, I limit) {
    for ( ; first != limit; ++first)
      push_back(*first);
  }

  template<Forward_iterator I>
    requires Same_as<T, value_type_t<I>>()
  vector::vector(I first, I limit) {
    reserve(distance(first, limit));
      // 1 allocation
    insert(begin(), first, limit);
  }
  // ...
```

To nie zadziała, jeśli kompilator nie może odróżnić `Forward_iterator` z `Input_iterator`.

Można to naprawić dodając nowe wymagania składniowe do `Forward_iterator`, które odnoszą się do jego rangi w hierarchii iteratorów. To tradycyjnie zostało zrobione przy użyciu *tag dispatch*. Łączenie *etykiety klasy*[1] z typem iteratora w celu wybrania odpowiedniego przeciążenia. Ten skojarzony typ to `iterator_category`. Zmieniony `Forward_iterator` może wyglądać tak:

```
template<typename I>
  concept bool Forward_iterator() {
    return Input_iterator<I>() && requires {
      typename iterator_category_t<I>;
      requires Derived_from<I,
        forward_iterator_tag>();
```

[1](ang. tag class) Pusta klasa w hierarchii dziedziczenia

```
    };
  }
```

Dzięki tej definicji wymagania `Forward_iterator` zaliczają wymagania `Input_iterator`, a kompilator może rozróżnić powyższe przeciążenia. Jako dodatkowa zaleta, używanie *iteratorów random access* będzie jeszcze bardziej wydajne bo `distance()` wymaga tylko jednej operacji całkowitej.

Jako inny przykład, *C++17* dodaje nową kategorię iteratorów: *iteratory contiguous*. *Iterator contiguous* jest *iteratorem random access*, którego obiekty odwoławcze są przydzielane w sąsiednich obszarach pamięci, których adresy rosną wraz z każdym przyrostem iteratora. Powoduje to otwarcie drzwi na wiele optymalizacji pamięci na niższym poziomie. Jest to oczywiście zupełnie czysta semantyka. Jeśli chcemy zdefiniować nowy koncept, musimy ją odróżnić od `Random_access_iterator`. Na szczęście właśnie zdefiniowaliśmy maszynę, aby to zrobić.

```
template<typename I>
concept bool Contiguous_iterator() {
  return Random_access_iterator<I>() && requires {
    requires Derived_from<I,
    contiguous_iterator_tag>();
  };
}
```