다형적 객체 타입 기술을 위한 메커니즘

이민규, 한동수
한국정보통신대학원대학교

A Mechanism for The Expression of Polymorphic Object Type

Minkyu Lee, Dongsoo Han
Information and Communications University

요 약

객체지향 프로그래밍에서 다형성이란 객체가 하나 이상의 객체 타입을 가질 수 있다는 것을 말한다. 이처럼 객체가 하나 이상의 타입을 가지게 되면 다양한 문맥에서 객체를 사용할 수 있게 되어 재사용성을 달성할 수 있게 된다. 그러나 이러한 객체의 다형성은 정적인 타입 검사를 어렵게 하여 실행 시 타입 오류를 유발할 가능성을 높이는 원인으로 작용한다. 본 논문에서는 객체 타입을 기술하기 위한 문맥에서 단일 타입만을 기술했던 제약에서 벗어나 여러 타입의 조합으로 다형적 타입을 기술할 수 있게 함으로써 문맥에 맞는 객체 타입을 기술할 수 있도록 하는 메커니즘을 제안한다. 그리고 이 메커니즘이 어떻게 사용될 수 있는가에 대한 예제들도 함께 소개한다.

1. 서론

프로그래밍 언어에서 일반적으로 타입(type)은 집합으로 이해된다. 어떤 객체가 A라는 타입을 가진다는 것은 A라는 집합에 속한다라고 이해할 수 있다. 객체지향 프로그래밍에서는 객체에 대한 타입을 주로 다루기 때문에 객체 타입(object type)이라는 용어를 사용한다[1]. 객체 타입 역시 인스턴스(instance)들의 집합으로 이해된다.

객체지향 언어들에서는 다형성(polymorphism)을 제공하여 재사용 가능한 객체를 개발할 수 있도록 해준다. 다형성이란 객체가 하나 이상의 타입을 가지는 것을 허용하는 것을 말하는데 객체가 여러 개의 타입을 동시에 지닌다는 말은 객체가 여러 집합에 동시에 속한다는 말로 이해할 수 있다. 즉, 객체 *x*가 *A*, *B*의 타입을 동시에 지닌다면 *A*와 *B*를 집합으로 볼 때 그림 1에서처럼 집합 *A*와 집합 *B*에 모두 속한다고 할 수 있다.

이러한 다형성을 가지는 객체들을 정의하고 만들어내는 것은 용이하지만 여러 타입을 동시에 지니는 객체만을 요구하는 문맥을 기술하는 것은 그다지 용이하지 않다. 즉, 그림 1에서처럼 *x*와 같은 객체들만 요구하는 문맥을 기술하려면 *A*와 *B*의 교집합 부분에 해당하는 타입 *T*를 기술하고 객체 *x*의 정적 타입을 *T*로 두어야 하는 번거로운 작업들이 필요하다.

본 논문에서는 *T*와 같은 타입을 정의하지 않고도 *A*, *B* 타입의 조합으로써 그러한 문맥을 기술할 수 있도록 하는 메커니즘인 Object Type Expression을 도입하였다. 2장에서는 객체 타입에 대한 설명과 객체의 기능적 분류에 대해 설명하고 3장에서는 Object Type Expression이 왜 필요하며 어떻게 사용될 수 있는지 그리고 이것에 대한 서브 타입 규칙이 어떻게 정의될 수 있는지를 보이고 이어 4장에서 결론을 맺는다.

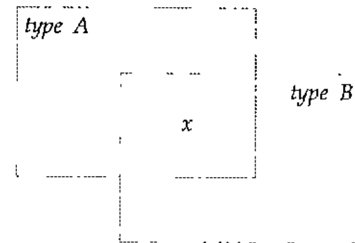

그림 1 *type A*와 *type B*를 모두 가지는 객체 *x*

2. 객체 타입과 객체의 기능적 분류

객체 타입(object type)이란 객체가 가지게 되는 타입을 의미한다. 그림 1에서 객체 *x*는 *A*, *B* 타입 모두를 가지는 것을 보여주고 있는데 이러한 객체를 다형적 객체(polymorphic object)라 부르고 객체 *x*를 형식적으로는 다음과 같이 표현할 수 있다. 여기서 $A <: B$는 *A*는 *B*의 서브 타입임을 의미하는 것이고 $T(x)$는 객체 *x*의 타입을 의미한다.

$$T(x) <: A \ \& \ T(x) <: B$$

C++[2], Eiffel[3]과 같은 많은 객체지향 언어들은 객체 타입을 직접 기술할 수 없고 클래스(class)와 객체 타입을 혼용하고 있다. 그러나 Java[4], Sather[5] 등과

같은 최근의 객체지향 언어들에서는 객체 타입을 기술할 수 있는 방법을 제공하고 있다. 특히 Java에서는 interface를 통하여 객체 타입을 기술할 수 있게 한다.

클래스를 정의함에 있어서 그것이 만족하는 객체 타입은 하나 이상이 되는 경우가 빈번하게 나타난다 특히, 객체들을 기능적으로 분류하기 위하여 많은 객체 타입을 미리 정의해 두고 새로운 클래스를 정의할 때 그것들을 구현하도록 하고 있다. 대표적인 것으로 Equality, Runnable, Clonable, Comparable, Observable, Serializable 등과 같은 것들이 있는데 주로 -able 이라는 접미사를 붙여 이름을 정하는 것이 일반적이다.

예를 들어 Integer 라는 정수의 클래스를 정의하고자 할 때 당연히 정수는 동등 비교가 가능해야하고, 서로 대소 비교도 가능해야 한다. 따라서 Equality, Comparable의 타입을 가지게 되는데, 만약 정수를 객체 수준에서 복제하고자 한다면 Clonable해야 하고 네트워크를 통한 전송 혹은 디스크 등의 저장 장치에 저장될 수 있으려면 Serializable해야 한다. 따라서 Integer 클래스의 객체 *i*는 그림 2와 이 설명될 수 있다.

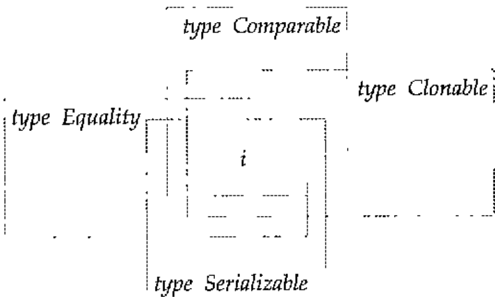

그림 2 객체의 *i*의 타입

그리고 객체 *i*의 타입은 형식적으로 다음과 같이 표현되어질 수 있다.

$$T(i) <: Equality \;\&\; T(i) <: Comparable \;\&\; T(i) <: Clonable \;\&\; T(i) <: Serializable$$

or

$$T(i) <: Equality, Comparable, Clonable, Serializable$$

3. Object Type Expression

3.1 다형적 객체 타입 기술의 문제점

우선 Set이라는 자료 구조를 정의한다고 가정하자. 어떤 객체가 집합의 원소인가를 검사하기 위해서는 동등 비교가 불가피 하므로 객체의 원소 타입은 최소한 Equality 타입을 가져야 한다. 따라서 Set 클래스는 다음과 같이 선언될 수 있다.

```
class Set<Equality T> {
  ...
  boolean is_member(T a) {
    ... a.equals(..) ...
  }
}
```

Set 클래스의 객체가 복제될 때 그것의 원소 객체들까지 모두 복제되도록 하는 기능(deep clone)을 추가하기 위해 ClonableSet 클래스를 정의하였다.

```
class ClonableSet<??? T> implements Clonable {
  ...
  boolean is_member(T a) { ... a.equals(..) ... }
  ClonableSet<T> clone() { ... elem.clone(..) ... }
}
```

여기서 clone이라는 메소드는 자기 자신을 복제하여 돌려주는 일을 수행하기 때문에 자신과 동일한 타입인 ClonableSet<T>으로 기술하였다.

다음으로 고려해야 할 사항은 ClonableSet의 원소 타입을 무엇으로 지정해 줄 것인가 하는 점이다. 원소 타입은 우선 동등 비교가 가능해야 한다는 것은 이미 설명하였고 복제(deep clone)를 구현하기 위해서는 원소 그 자체도 복제가 가능해야 한다. 그렇다면 타입 파라미터인 T는 어떤 타입이어야 하는가?

타입 파라미터 T에 Equality 타입을 실제 파라미터로 넘겨주면 원소 객체들의 동등 비교만을 보장하게 되므로 원소 객체에 clone() 메소드를 호출할 수 없다. 따라서 ClonableSet의 clone() 메소드를 호출하기 위해서는 RTTI(Run Time Type Identification)를 사용해 Clonable 타입을 가지는 지를 검사해야 하고 불가피하게 타입 캐스팅(type casting)을 사용해야 한다. 반대로 Clonable 타입을 넘겨주어도 마찬가지 상황이 된다.

이러한 상황을 종합해 볼 때 타입 파라미터 T가 요구하는 정확한 타입을 표현하면 다음과 같다.

$$T <: Equality, Clonable$$

결국 ClonableSet의 원소 타입은 동시에 Equality와 Clonable 타입을 동시에 가져야 한다. 다시 말해 원소 타입은 Equality와 Clonable의 서브 타입이어야 한다는 것이다. ClonableSet의 원소로 사용될 수 있는 객체의 영역을 그림으로 나타내면 그림 3에서 채워진 부분이 된다.

그림 3 ClonableSet의 원소 객체들이 속한 영역

원소 타입이 Equality와 Clonable 모두를 만족해야 하므로 ClonableSetElementType 이라는 객체 타입을 따로 다음과 같이 선언하였다. 그리고 이 타입을 ClonableSet의 원소 타입으로 지정하였다.

```
objecttype ClonableSetElementType
  extends Equality, Clonable {}

class ClonableSet<ClonableSetElementType T>
    implements Clonable { ... }
```

ClonableSetElementType을 구현한 모든 클래스의 객체들이 ClonableSet의 원소로 사용될 수 있게 된다. 그러나 여기에도 문제가 있다. 우선 Integer, Float 클래스가 아래와 같이 정의되어 있다고 가정하자.

```
class Integer implement Equality, Clonable, ...
class Float implement Equality, Clonable, ...
```

비록 Integer, Float가 Equality, Clonable을 만족하지만

ClonableSetElementType을 직접 구현한 것이 아니므로 ClonableSet의 원소로 사용할 수 없게 된다. 이것은 지나친 제약이 되어서 기존에 구성된 클래스의 객체들을 전혀 사용할 수 없게 되는 단점이 있다.

3.2 subtypeof

3.1 절에서 제시한 여러 문제점들을 극복하기 위하여 여러 객체 타입들의 조합으로 어떤 문맥에 필요한 객체들의 타입을 표현할 수 있도록 subtypeof 라는 키워드를 도입하여 사용하고자 한다. 이것을 사용하면 ClonableSet은 다음과 같이 정의될 수 있다.

```
class ClonableSet<subtypeof(Equality, Clonable) T>
    implements Clonable
{
    ...
    boolean is_member(T a) { ... a.equals(..) ... }
    MyType clone() { ... elem.clone(..) ... }
}
```

subtypeof를 사용함으로써 ClonableSet의 원소 타입으로 사용 가능한 타입을 정확하게 표현해 줌으로써 그러한 조건을 만족하지 못하는 타입의 객체가 원소로 사용되는 것을 정적으로 검사하여 막아줄 수 있게 된다.

다른 경우를 살펴보자. 하나의 객체를 인자로 받아서 그 객체를 원하는 개수만큼 복제를 한 다음 그것을 쓰레드로 동시에 실행하는 함수를 정의하려고 한다. 쓰레드는 Runnable 객체 타입의 연산인 run()을 실행하면 시작되는 것으로 가정한다.

```
void makeClonesAndRun(??? obj, Integer num) {
  for (Integer i=1; i<=num; I++) {
    ??? t = obj.clone();
    t.run();
  }
}
```

이와 같은 경우에도 마찬가지의 문제가 발생하게 되는데 함수 makeClonesAndRun의 인자 obj의 타입과 t의 타입이 그러하다. 형식 인자 obj로 들어오게 될 객체와 t 객체는 Clonable과 Runnable 타입을 가져야 한다. makeClonesAndRun 함수의 obj와 t의 정확한 타입을 표현하면 다음과 같다.

$T(obj) <: Clonable, Runnable, T(t) <: Clonable, Runnable$

이 때 subtypeof를 사용하여 기술해주게 되면 역시 개념적으로 이해하기 쉬울 뿐만 아니라 정적 타입 검사가 가능해지게 된다. subtypeof를 사용하여 다시 작성한 makeClonesAndRun의 새로운 버전은 다음과 같다.

```
void makeClonesAndRun
  (subtypeof(Clonable, Runnable) obj, Integer num)
{
  for (Integer i=1, i<=num; i++) {
    subtypeof(Clonable, Runnable) t = obj.clone();
    t.run();
  }
}
```

subtypeof는 객체의 타입을 좀 더 정확하게 기술할 필요가 있을 때에 유용할 뿐만 아니라 전체적인 객체 타입의 계층 구조도 단순화 할 수 있다. 위 예에서 다루었던 ClonableSet의 경우 subtypeof를 사용하지 않는다면 ClonableSetElementType 이라는 객체 타입을 정의하고 사용하여야 한다. 이러한 방식은 그림 4와 같이 계층 구조를 복잡하게 만들게 되고 이러한 Set의 변형 자료구조들이 많이 생길수록 복잡해지면서도 객체들의 재사용성이 떨어지게 된다.

그러나 subtypeof를 사용할 수 있는 경우라면 ClonableSetElementType과 같은 객체 타입을 정의할 필요가 없고 몇몇의 객체 타입의 조합으로 그것을 표현할 수 있기 때문에 객체 타입들의 계층구조를 더욱 간단하게 만들 수 있다.

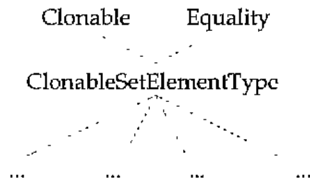

그림 4 복잡화된 객체 타입들의 계층구조

3.3 서브 타입 규칙

다형적 객체 타입을 기술하기 위해 새로 도입한 subtypeof에 대한 서브 타입 규칙을 다음과 같이 정의하였다. 어떤 객체의 타입이 T'라고 가정하고 $subtypeof(T_i)$의 서브 타입이 되려면 각각의 T_i의 서브 타입이어야 한다.

$$T' <: subtypeof(T_i)_{1 \le i \le k} \text{ if } k \ge 1 \text{ and for each } T_i :> T'.$$

4. 결론

본 논문에서는 여러 객체 타입들의 조합으로 다형적 객체 타입을 기술할 수 있는 메커니즘인 Object Type Expression을 소개하고 이것이 어떠한 경우에서 필요하며 또 잘 적용될 수 있는지를 소개하였다. 다형적 객체 타입을 표현해야 하는 경우는 종종 발생하는데 표현할 수 있는 방법이 없는 경우 정적 객체 타입 검사를 힘들게 할 뿐 아니라 객체 타입들의 계층 구조도 복잡하게 만들 우려가 있다.

앞으로 더 연구되어야 할 사항은 Java, Sather 등과 같이 subtypeof를 도입하기에 적절한 언어를 선택한 후 그것을 확장하여 다형적 객체 타입을 기술할 수 있도록 subtypeof를 추가하는 연구가 더 진행되어야 할 것이다. 또한 subtypeof가 어떠한 경우에 잘 적용될 수 있는지 그리고 subtypeof를 사용하여 어떻게 객체지향 프로그래밍을 더 쉽게 할 수 있는지에 관한 연구도 함께 필요할 것이다.

참고문헌

[1] James Martin, James J. Odell, *Object-Oriented Method: A Foundation*, Prentice Hall, 1995.

[2] Bjarne Stroustrup, *The C++ Programming Language 3rd ed.*, Addison-Wesley, 1997.

[3] Bertrand Meyer, *Eiffel: The Language*, Prentice-Hall, 1992.

[4] Ken Arnold, James Gosling, *The Java Programming Language 2nd ed.*, Addison-Wesley, 1998.

[5] Benedict Gomes, Holger Klawitter, David Stoutamire, Boris Vaysman, *Sather 1.1: A Language Manual*, http://www.icsi.berkeley.edu/~sather.