2000년 한국정보처리학회 추계 학술발표논문집 제7권 제2호

포괄적 정의의 특수화를 지원하는 GJ 언어의 확장에 관한 연구

양우석, 권기항
동아대학교 컴퓨터공학과
e-mail:phoenix@ce.donga.ac.kr

A Study on the Extension of GJ for supporting Generic Definition Specialization

Wooseok Yang, Keehang Kwon
Dept. of Computer Engineering, Dong-A University

요약

GJ는 인자적 다형성(Parametric Polymorphism)을 사용할 수 있도록 Java를 확장한 언어이다. 인자적 다형성은 포괄적 프로그래밍을 가능하게 할 뿐만 아니라 객체지향성의 이론적 결함을 메워줄 수 있는 언어 기능이다. 그러나 일반성이 높아지면 상대적으로 효율성은 감소하기 때문에 반대로 일반성을 제약하여 효율성을 높여야만 할 때가 많다. C++에서는 특정한 타입에 대해서는 특화된 코드가 사용될 수 있도록 하여 효율성의 손실을 메울 수 있는 기능을 제공하고 있다. 본 논문에서는 포괄적 프로그래밍으로 일반성과 재사용성을 높이고 타입 인수의 특수화를 통해 일반성에서 오는 효율성 손실을 막기 위한 방법으로, GJ의 의미론을 확장하고 그 실용적인 타당함을 보이기 위한 몇 가지 예제를 기술하였다. 그리고 이러한 기능적 확장에 따르는 이점과 단점을 열거하였다.

1. 서론

GJ[1]는 인자적 다형성(Parametric polymorphism)을 사용할 수 있도록 자바를 확장하였다. 이러한 기능은 정적형검사를 가능하게 하면서 데이터형에 종속적이지 않는 프로그램을 작성할 수 있도록 하여 포괄성을 크게 향상시킬 수 있도록 하였다. 즉, 타입을 인수로서 받을 수 있도록 자바의 문법(Grammar)을 확장하여 프로그램 구조와 코드가 같은 경우에는 한번의 정의로 타입 정보를 제외한 구현 부분을 재사용할 수 있도록 하였다.

그러나 GJ가 제공하는 포괄성은 재사용성을 향상시킬 수 있는 반면에, 특정 타입에만 적합하고 효율적인 프로그램을 추가할 수 있는 방법이 없다. 즉, 컴파일러가 제공하는 재사용성이 높은 프로그램을 얻을 수 있지만 부분적으로 특정 타입에 대하여 프로그래머가 원하는 효율적인 프로그램을 점진적으로 추가할 수 있는 언어적 기능을 가지고 있지 않다. C++[2]는 템플릿을 통해서 포괄적 프로그램을 얻는 동시에, 효율성 손실을 막기 위하여 템플릿 특수화(Template specialization)를 제공하고 있다.

게다가, 현재 GJ는 타입 인수의 인자로서 참조 의미론(Reference semantics)을 가지는 타입만 가능하게 제약을 하고 있다. 그러나 GJ는 자바를 확장한 언어이므로, 참조 의미론을 가지는 타입뿐만 아니라 값 의미론(Value semantics)을 가지는 타입도 가지고 있다. 이러한 제약적인 문제에 대한 해결책이 필요하다.

이런 문제점을 보완하고자 GJ의 타입 인수의 특수화를 제안하고자 한다. 타입 인수의 특수화를 통해서 C++의 템플릿 특수화와 같은 효과를 얻는 동시에, 불완전한 GJ의 의미론을 완성을 시키고자 한다.

본 논문에서는 이러한 타입 인수 특수화를 문법적인 변화를 주어서 하고자 하는 것은 아니다. GJ와의 언어적 호환성을 유지하기 위하여 문법적인 변화는 주지 않고 의미론적으로 추가를 하고자 한다. 즉, GJ가 가지고 있는 의미론에 추가적으로 특수화를 위한 의미론만을 추가하여 GJ와의 코드 호환성을 유지하면서 변화를 주고자 하는 것이다.

2. GJ 언어와 인자적 다형성

GJ는 인자적 다형성을 사용할 수 있도록 Java를 확장한 언어이다. 이러한 기능은 정적형검사를 가능하게 하면서 데이터형에 종속적이지 않는 프로그램을 작성할 수 있도록 하여 포괄성을 크게 향상시킬 수 있다.

이러한 언어적 기능은 Java로서 구현할 수 없는 것은 아니다. Java로 이러한 구현을 얻기 위해서는 프로그래머의 노력이 몇 배로 들게 된다. GJ는 그 기능을 언어적 편의로 제공해준다. 컴파일 시에는 GJ의 확장된 부분은 Java의 구현으로 변환하여 이루어진다.

2.1 인자적 다형성

GJ는 인자적 다형성을 사용할 수 있도록 Java를 확장한 언어다. 다음과 같이 Pair를 인자적 다형성을 이용하여 정의하면 멤버 변수의 타입에 제한적이지 않는 구현의 재사용성을 얻을 수가 있다.

```
class Pair<T> { // Pair 클래스 : 인자적 다형성
        protected T first;
        protected T second; // Pair : (first, second)
        ......
}
```

일반적인 인자적 다형성은 타입 인수에 아무런 제약을 줄 수가 없다. 타입 인수에 제약을 줄 수 없는 다형성은 타입 인수가 적절히 가져야할 조건이 있음에도 불구하고 그 조건에 만족하지 않는 타입을 인수로서 줄 수 있다. 이러한 결함을 해결하기 위해 서브타입 다형성(Subtype polymorphism)과 인자적 다형성을 적절히 결합하여 타입 인수에 적절한 제한을 주는 언어 기능을 제한적 다형성(Bounded polymorphism)이라고 한다.

```
interface Ord { ...... } // Pair 클래스 : 제한적 다형성
class Int implements Ord {
        ......
        public boolean less(Object x) {
                if (x instanceof Int)
                    return value < ((Int)x).getValue();
                else ......
}
class Pair<T implements Ord> { ......
        public T max() { ...... }
}
```

위의 예제는 앞 절에서 소개한 Pair에 순서쌍의 두 값을 비교하여 큰 값을 돌려주는 max라는 메서드를 추가하였다. 이 메서드에 적용하려는 값의 타입은 적어도 대소비교가 가능해야 한다. 위의 예제에서 first와 second를 정렬이 가능한 타입으로 제한을 주기 위해 Ord라는 인터페이스(Interface)를 구현(Implements)한 타입만으로 형인자를 제한한다.

Int 클래스의 less 메서드와 같이 양인자가 동일한 타입만 가능한 연산자를 이진연산자(binary method)라고 한다. 이러한 연산자는 서브타입 다형성 이론의 변성 규칙에 따르면 하부 메서드의 인자형은 상부 메서드의 인자형과 동일하거나 더 일반적인 타입이 되어야 하지만 이러한 이진 연산자인 경우에는 적용할 수 없다는 것이 증명되었다.[3] 이러한 난점을 서브타입 다형성과 인자적 다형성의 적절한 결합으로 해결하고자 한다. 아래의 예제에서 인자적 다형성을 가진 인터페이스를 작성하여 인수에 제약을 주는 방식으로 해결하려는 언어 기능을 F-제한적 다형성(F-Bounded polymorphism)이라고 한다.

```
interface Ord<T> { ...... } // Pair 클래스 : F-제한적 다형성
class Int implements Ord<Int> {
        ......
        public boolean less(Int x) {
                return value < x.getValue();
        ......
}
class Pair<T implements Ord<T>> {
        ......
        public T max() { ...... }
}
```

2.3 GJ 언어에서 인자적 다형성의 문제점

GJ가 제공하는 포괄성은 재사용성을 향상시킬 수 있는 반면에, 특정 타입에만 적합하고 효율적인 프로그램을 추가할 수 있는 방법이 없다. 즉, 컴파일러가 제공하는 재사용성이 높은 프로그램을 얻을 수 있지만 부분적으로 특정 타입에 대하여 프로그래머가 원하는 효율적인 프로그램을 점진적으로 추가할 수 있는 언어적 수단이 없다.

인자적 다형성은 컴파일러에 의해서 타입 인수에 적용되는 모든 타입에 대해서 단일한 프로그램을 생성하여 제공하는 방식이다. 이러한 언어 기능은 일반성과 재사용성을 얻는 동시에 정적형검사가 가능하다. 그러나 보통 일반성이 높아지면 상대적으로 효율성은 감소하므로 객체지향성의 함수재정의와 같이 필요에 따라서 더 좋은 프로그램을 사용할 수 있는 가능성이 요구된다.

게다가 GJ는 타입 인수로 Integer와 같이 참조 의미론을 사용하는 타입만 가능하다. 왜냐하면 타입 인수를 최상위 타입인 Object로 대체하는 방법으로 변환을 하는 동형 변환을 사용하고 있기 때문이다. 그러나 자바와 GJ는 값 의미론을 쓰는 int와 같은

타입도 존재하면서도 그 타입은 타입인수로서 사용할 수 없으므로 그러한 타입에 대한 재사용성은 얻을 수 없다. 이러한 결함을 보완하게 되면 GJ의 완전한 의미론과 재사용성을 얻을 수 얻을 수 있다.

그리고 포괄적 프로그래밍으로 제공되는 재사용성이 높은 라이브러리에도 특정 타입에 대한 효율성이 입증된 알고리즘이 있다면 그것을 사용할 수 있는 기능을 제공함으로써 일반성과 재사용성을 높이고 효율성의 손실을 막을 수 있는 프로그래밍이 가능하게 된다.

3. GJ 언어에서의 타입 인수의 특수화(Specialization)

인자적 다형성은 정적형검사를 가능하게 하면서 데이터형에 종속적이지 않는 프로그램을 작성할 수 있도록 하여 포괄성을 크게 향상시킬 수 있다. 즉, 컴파일러가 자동으로 필요한 타입 캐스팅과 타입 검사를 해주기 때문에 프로그래머에게 일반성과 재사용성을 얻을 수 있는 좋은 추상화 수단을 제공한다. 그러나 특정 타입에 대한 효율성이 입증된 알고리즘이 있다면 GJ의 인자적 다형성을 제약하여 그것을 점진적으로 추가할 수 있는 가능성이 요구된다. 본 논문에서는 포괄적 프로그래밍으로 일반성과 재사용성을 높이고 타입 인수의 특수화를 통해서 오는 효율성 손실을 막기 위한 방법으로, GJ의 의미론을 확장하고 실용적인 타당함을 보이기 위한 몇 가지 예제를 기술한다.

3.1 타입 인수의 특수화

타입 인수의 특수화는 인자적 다형성으로 작성된 포괄적 프로그램으로 일반성과 재사용성을 얻으면서 특정한 형이나 형의 생김새에 따라 특화된 프로그램을 점진적으로 추가하여 사용하는 기능이다. 다음은 GJ에서 본 문법의 예이다.

클래스 선언
ModifiersOpt class Identifier TypeParametersOpt SuperOpt InterfacesOpt ClassBody

메서드 선언
ModifiersOpt TypeParametersOpt Type MethodDeclarator ThrowsOpt

TypeParametersOpt : GJ의 타입 인수 또는 특정한 형

위에 제시한 문법은 기존의 GJ와 코드 호환성을 유지하기 위하여 기존의 문법적 형태를 변경하거나 새로운 의미를 부여할 수 있는 키워드를 주지 않는다. 이러한 이유로 기존의 GJ 의미론을 확장하여 사용하려고 한다. 클래스 또는 메서드를 정의할 때에 TypeParametersOpt를 특정한 형으로 기술하여 프로그래밍을 하게 되면 그러한 타입에 한해서 특수화된 프로그램을 사용하게 한다. 이러한 클래스의 메서드를 사용시에는 다음과 같은 절차를 가진다.

- 타입 추론 및 해당 클래스와 메서드의 존재 여부 검색.
- 검색 후에 해당되는 타입으로 특수화된 프로그램의 존재 여부 검색.
- 특수화된 프로그램이 존재한다면 그 프로그램을 사용하도록 한다.
- 특수화된 프로그램이 없다면 기존의 GJ 의미론에 따라서 포괄적 프로그램을 사용한다.

3.2 타입 인수의 특수화를 적용한 예

타입 인수의 특수화에 관한 확장된 의미론을 기술하였다. 이 절에서 그런 특수화의 예제를 들고자 한다.

```
interface Iterator{
        public boolean hasNext();
        public Object next();
}
class Forward_Iterator implements Iterator{ ...... }
class Random_Iterator implements Iteraotr{ ...... }
```

*Iterator*는 복합데이터에 포함된 모든 원소들을 일정한 순서에 의해 열거할 수 있는 프로그램이다. hasNext는 다음 원소가 존재여부를 확인하는 메서드이며, next는 다음 원소를 추출하는 메서드이다. 그리고 Iterator 인터페이스에서 구현을 한 Forward_Iterator는 단방향으로만 열거가 가능한 Iterator이고 Random_Iterator는 임의로 열거가 가능한 Iterator이다.

```
class Utils{
        public static <T implements Iterator> int
        distance(T it){
                int count = 0;
                while(it.hasNext()){
                        count++;
                        it.next();
                }
                return count;
        ......
```

위의 함수는 현재 Iterator가 열거하고 있는 위치와 끝 사이의 거리를 알려주는 기능을 한다. 이 때에 임의의 순서로 열거가 가능한 Iterator는 위의 프로그램보다 더욱 효율적인 프로그래밍이 가능하다. 위

의 프로그램이 N개의 원소에 대해서 O(N)만큼의 시간 복잡도가 발생한다면 아래의 예로서 특수화를 한다면 O(1)만큼의 시간 복잡도가 발생한다. 아래의 예제는 Random_Iterator에 대해서 타입 인수를 특수화한 예이다.

```
class Utils {
        ......
        public static <Random_Iterator>
                int distance (Random_Iterator it){
                        return it.end() - it.start();
                }
        ......
```

end는 Iterator의 끝의 인덱스를 돌려주고 start는 현재 위치의 인덱스를 돌려주는 메서드이다.

3.3 값 의미론을 가지는 지원

Java는 int와 Integer처럼 타입을 두 가지로 구분해서 다른 의미를 부여한다. int형은 값 의미론을 가지는 타입이며 Integer형은 참조 의미론을 가지는 타입이다. GJ도 마찬가지로 Java의 확장된 언어임에도 불구하고 타입 인수로는 Integer와 같은 참조 의미론이 가능한 타입만 가능하다. 이러한 결점을 타입 인수의 특수화를 통해서 해결함으로써 완전한 GJ의 의미론과 재사용성을 얻을 수 있다.

```
class Vector<T> { ...... }
class Vector<int> { ...... }
```

이렇게 int형으로 특수화를 하면 타입 변환이 없으므로 좀 더 효율적으로 동작하게 된다. 그리고 값 의미론을 쓰는 타입도 타입 인수로 사용할 수 있으므로 클래스로 둘러싸는 오버헤드가 발생하지 않는다.

3.4 타입 인수의 특수화가 주는 이점 및 단점

타입 인수의 특수화는 일반성에서 오는 효율성 손실을 막을 수 있다. 즉, 특정한 형이나 형의 생김새에 따라 특수화된 프로그램은 포괄적으로 작성된 프로그램보다 효율적이라는 것은 일반적인 사실이다. 그리고 점진적으로 추가가 가능하므로, 차후에 더욱 효율적인 프로그램으로 개발하기가 용이하다.

게다가 타입 인수의 특수화를 통해서 GJ의 타입 인수로 값 의미론을 쓰는 타입을 쓸 수 있도록 함으로써 완전한 GJ의 의미론과 재사용성을 얻을 수 있다.

반면에, 특수화를 하기 위한 추가적인 의미론과 특수화의 타입 정보를 처리한다는 것은 컴파일러의 복잡성을 증가시키는 요인이 된다.

4. 결론 및 향후과제

본 논문에서는 인자적 다형성을 사용할 수 있도록 Java를 확장한 언어인 GJ의 인자적 다형성의 결점을 지적하였다. 이에 해결책으로 타입 인수의 특수화를 제시하여 GJ의 의미론을 확장하고 그 실용적인 타당함을 보이기 위한 몇 가지 예제를 기술하였으며 그러한 특수화가 주는 이점과 단점을 기술하였다. 또한 타입 인수의 특수화를 통해 GJ의 타입 인수로서 값 의미론을 쓰는 타입을 사용할 수 있음을 보였다.

그러나 GJ에 기존의 문법과 의미론을 유지하면서 새로운 의미를 추가하려는 복잡성은 남아있다. 차후에 계속 연구가 진행되어 타입 인수의 특수화를 위한 의미론의 정형화와 타입 추론 엔진 그리고 그러한 기능을 실제 지원하는 컴파일러에 관한 연구가 더욱 진행되어야 한다.

참고문헌

[1] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler, "Making the future safe for the past : Adding Genericity to the Java™ Programming Language", 1998

[2] Bjarne Stroustrup, "The Design and Evolution of C++", Addison-Wesley, 1994

[3] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gray T. Leavens and Benjamin Pierce, "On Binary Method", 1994

[4] Bjarne Stroustrup, "The C++ Programming Language" - 3rd Ed., Addison-Wesley, 1997

[5] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler, "GJ : Extending the Java™ Programming language with parameters", , 1998

[6] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler, "GJ Specification", 1998

[7] Martin Odersky and Philip Wadler, "Pizza into Java : Translating theory into practice", 1997