□ 기술해설 □

의미론적 정의로부터 컴파일러의 자동생성

한양대학교 도경구*

목 차

1. 서 론
2. 컴파일러 생성기의 일반적인 구조
3. action 의미론에 의한 컴파일러 자동생성
4. 부분 계산에 의한 컴파일러 자동생성
5. 결 론

1. 서 론

프로그램을 컴퓨터로 실행하기 위해서는 그 프로그램을 처리할 수 있는 프로세서가 필요하다. 그러한 프로세서로는 인터프리터가 있는데, 인터프리터는 원시 프로그램과 입력 데이터가 주어졌을 때 실행하여 바로 그 결과를 준다. 그러나 인터프리터는 구현이 용이한 반면에, 원시 프로그램을 실행할 때마다 그 원시 프로그램 자체도 함께 처리해야 하는 추가 시간이 소요되기 때문에 효율적이지 못하다. 따라서 보통 고급언어로 쓰여진 원시프로그램들은 더 빨리 수행될 수 있는 기계어 등으로 된 목적 프로그램으로 컴파일 시키며, 그 목적 프로그램은 원래의 프로그램을 인터프리터를 통해서 실행하는 것 보다 빠른 속도로 실행될 수가 있다. 이렇게 고급언어로 쓰여진 프로그램을 목적 프로그램으로 번역하는 프로그램을 컴파일러라고 한다.

현재 널리 쓰이는 거의 모든 컴파일러들은 원시언어와 목적언어의 명세서에 따라서 직접 구현자들에 의해서 구현이 된다. 구현하고자 하는 언어의 명세서는 대부분 현존하는 언어가 그렇듯이 구문을 제외하고는 자연어로 표현되어 있다. 그러나 자연어는 그 자체의 모호함 때문에 구현하는 사람들에게 설계자의 의도가 잘못 전달되기 쉬운 단점이 있다. 또한 이렇게 해서 구현된 컴파일러의 정확도를 형식적으로 증명하는 일은 더욱 불가능하다. 게다가 직접 작성된 컴파일러는 거의 대부분의 경우 범용이 되지 못하고, 목적 프로그램에 따라서 기존의 컴파일러를 다시 작성하거나 많은 부분을 수정해야 하는 등의 문제도 있다. 이러한 단점을 근본적으로 보완하기 위해서는 구현하고자 하는 원시 언어와 목적 언어의 형식적 정의(formal definition)로부터 직접 자동적이며, 체계적으로 컴파일러를 생성할 수 있어야 한다. 컴파일러를 자동으로 생성할 수 있게 되면 언어를 설계하는 과정에서 설계된 언어가 설계자의 의도에 부합하는지를 수시로 시험해볼 수 있고, 생성된 컴파일러의 정확도 증명도 가능하게 된다.

프로그래밍언어를 형식적으로 정의하는 대표적인 기법들로는 표시적 의미론(denotational semantics) [38,37], 공리적 의미론(axiomatic semantics) [15,18], 동작적 의미론(operational semantics) [35,23,14] 등을 들 수 있다. 이 기법들은 제각기 나름대로의 장점을 지니고 있기는 하지만, 컴파일러의 자동 생성에 직관적으로 가장 적합한 기법은 표시적 의미론이다. 최근의 일련의 연구[17,8]에 따르면 동작적 의미론만을 이용하여서도 만족할 만큼 효율적인 컴파일러 및 추상 계산기를 자동으로 생성할 수 있는 것으로 밝혀졌다. 그러나 본 논문에서는 전통적으로 연구되어왔던, 표시적 의미론을

*정 회 원

기초로한 형식적 정의로부터 컴파일러를 자동으로 생성하는 문제에 관해서 살펴보고자 한다. 제2절에서는 컴파일러 생성기의 일반적인 구조와 전통적인 표시적 의미론에 의한 접근방법의 문제점에 대해서 살펴보고, 다음 제3절에서는 표시적 의미론을 개선한 action 의미론에 의한 컴파일러 생성기에 대해서 알아보고, 제4절에서는 부분 계산에 의한 컴파일러 자동생성에 대해서 설명하고, 마지막으로 제5절에서 결론을 짓는다.

2. 컴파일러 생성기의 일반적인 구조

일반적으로 컴파일러는 크게 분석(analysis)하는 부분과 합성(synthesis)하는 부분으로 나뉘어진다. 더 세분하면 그림 1과 같이 분석부분은 형태 분석(lexical analysis), 구문 분석(syntax analysis), 그리고 타입과 같은 고정 데이터를 분석하는 의미 분석(semantic analysis)으로 나뉘어지고, 합성부분은 번역기, 그리고 목적코드 생성기로 나뉘어진다.

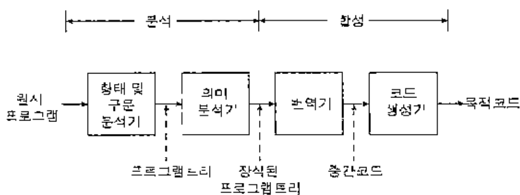


그림 1 컴파일러의 구조

컴파일러의 앞부분을 구성하는 형태 분석 및 구문 분석은 프로그램의 구문(syntax)을 처리하는 부분으로, regular expression과 context-free grammar에 의해 정확하게 형식적으로 정의할 수 있고, 또한 각각 finite automata 및 pushdown automata를 통해서 구현할 수도 있다. 따라서, 컴파일러에서 형태 및 구문을 처리하는 부분들은 자동 생성이 비교적 용이하고, 실질적으로 Lex와 Yacc같은 프로그래밍 도구들은 컴파일러 작성을 위한 도구로서 널리 쓰이고 있다. 이러한 도구들은 토큰의 형태를 정의하는 regular expression 및 구문을 정의하는 context-free grammar를 각각 입력으로 받아서, 거의 직접 작성된 것만큼 효율적인 형태분석기(lexer) 및 구문분석기(parser)를 각각 자동으로 생성할 수 있다.

의미분석은 타입이나 scope과 같이 프로그램을 실행하지 않고도 알 수 있는 고정된 정보를 처리하는 부분으로, 분석된 결과는 프로그램에 장식하여 컴파일러의 후반부에서 더 효과적인 목적프로그램을 생성하는데 이용한다. 의미분석의 자동화는 형태 및 구문분석과는 달리 그리 간단하지 않다. 왜냐하면 구현하고자 하는 언어의 특성에 따라서 분석할 수 있는 정도가 달라지기 때문이다. 예를 들면 static scope의 규칙을 사용하는 언어는 변수들의 scope 분석이 가능하지만, dynamic scope의 규칙을 사용하는 언어는 불가능하다. 또한 타입을 static하게 결정할 수 있는 언어는 타입의 분석이 가능하지만, 그렇지 않은 경우는 불가능하다. 따라서 이 부분을 모든 형태의 언어들을 다 수용할 수 있도록 자동으로 처리하기 위해서는 컴파일러 생성기 자체가 프로그래밍언어의 정의를 분석하여 그 언어의 성질을 판별할 수 있는 능력이 있어야 한다.

컴파일러의 후반부를 구성하는 코드 생성 부분은 프로그래밍언어 의미의 정의에 좌우된다. 일반적으로 형식적 의미론에 의한 정의는 프로그램의 의미가 메타언어로 표시가 된다. 따라서 컴파일러 생성기에 의해서 생성된 컴파일러는 프로그램을 그 메타언어 프로그램으로 팽창하고, 그 팽창된 메타언어 프로그램은 목적코드로 번역이 된다. 그러므로 그 메타언어가 프로그래밍언어의 성질을 잘 반영해 줄 수 있도록 설계되었는가, 또한 목적코드를 효율적으로 생성할 수 있도록 설계되었는가의 여부는 컴파

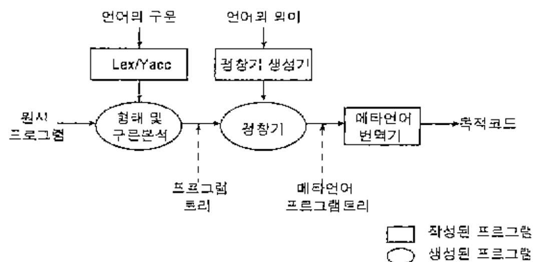


그림 2 컴파일러 자동 생성기의 일반적인 구조

일러 생성기의 성능을 좌우하는 중요한 열쇠가 된다.

컴파일러 자동 생성기와 생성된 컴파일러의 일반적인 구조는 그림 2와 같다. 그림에서 직사각형으로 표시된 부분은 컴파일러 생성기에 속한 부분을 가리키고, 동그라미로 표시된 부분은 컴파일러 생성기에 의해서 생성된 프로그램을 나타낸다.

컴파일러를 자동 생성하는 것이 직접 컴파일러를 작성하는데 비해서 좋다고 생각되는 점은 다음과 같다.

(1) 언어의 설계자의 의도가 형식적 정의를 통해서 정확하게 반영이 되고, 그 의도가 컴파일러 생성기에 의해서 생성된 컴파일러에 그대로 전달된다.

(2) 언어의 설계가 변경되었을 경우에 컴파일러 프로그램을 변경하는 것보다는 프로그램의 정의를 변경하는 것이 훨씬 수월하다.

(3) 일단 컴파일러 생성기가 정확하다고 증명이 되면, 생성된 컴파일러의 정확도는 증명할 필요가 없다.

(4) 언어의 설계시 언어의 정의가 완성되면 바로 컴파일러를 생성하여 프로그램을 실행함으로써, 언어가 원래의 의도대로 설계되었는지를 확인하는 도구로 사용될 수 있다.

표시적 의미론의 경우, 프로그램의 의미는 λ-calculus 라는 메타언어를 통해서 표시가 되는데, 결국 컴파일러 생성기에 의해서 생성된 컴파일러는 프로그램을 λ-calculus 프로그램으로 번역하고, 결국은 그 프로그램은 λ-reducer에 의해서 더욱 최적화된 프로그램으로 변환한다. 사실상 1979년 Peter Mosses의 SIS(Semantics Implementation System)[26]는 이 아이디어를 그대로 구현한 최초의 컴파일러 자동생성 시스템이다. 그러나 생성된 컴파일러는 일반 상업용에 비해서 수만 배나 느린 목적코드를 생성하였고, 결과적으로 그 이후 이를 향상시키 위해서 많은 연구가 이루어졌다[19,34,39]. 그러나 오래고 꾸준한 연구에도 불구하고 그 결과는 그다지 만족스럽지 못했고, 생성된 컴파일러는 직접 작성된 컴파일러에 비해서 그 성능이 현저하게 뒤졌다. 실패의 가장 큰 원인으로는, 메타언어인 λ-calculus는 이론적으로는 잘 정립이 되어 있어서 프로그램의 성질에 대해서 분석하고 추론하는데는 적합하였지만, 프로그램의 기본적인 개념을 표현하기에는 자연스럽지 못하였고, 특히 그 언어들이 어떻게 효과적으로 구현될 것인가에 대해서 언급하기에 부적합하였다.

그 이후 Peter Lee는 그의 PhD 논문[24]에서 언어를 고정적 의미(static semantics)와 유동적 의미(dynamic semantics)로 구분하여 정의하는 high-level 의미론을 이용하여 과거의 시스템보다는 훨씬 수행 능력이 우수한 컴파일러를 생성할 수 있는 Mess라는 시스템 제작에 성공하였다. 결국 그의 아이디어는 다음 장에서 다룰 action 의미론과 맥락을 같이하지만, 그의 high-level 의미론은 효율적인 컴파일러 생성에 중점을 두었기 때문에 순수한 프로그래밍언어의 정의를 위한 체계라고는 보기 어려운 단점이 있다.

3. action 의미론에 의한 컴파일러 자동생성

표시적 의미론의 단점을 보완하기 위해서 최근에 고안된 action 의미론[27,28,29]은 소규모의 언어가 아닌 실질적인 언어를 정식으로 표현할 수 있을 뿐 아니라, 한 번 정의된 의미는 다른 비슷한 언어의 정의에도 재사용 될 수 있는 등 모듈화가 용이하고, 프로그래밍에 필수적으로 많이 쓰이는 개념들을 action이라는 용어들을 통해서 제공하기 때문에 구현자들에게 어떻게 구현해야 한다는 힌트도 제시할 수 있어서, 프로그래밍 언어 설계자뿐만 아니라, 구현자, 사용자, 그리고 언어의 성질을 연구하고자 하는 사람들에게까지도 아주 유망한 도구로서 점차 인정되어 가고 있다. 이러한 action 의미론의 우월성을 등에 업고, 이 기법을 이용한 컴파일러 자동 생성에 대한 연구가 최근 활발히 진행되고 있다. 지금까지의 연구 결과에 의하면 표시적 의미론에 근거한 컴파일러 자동생성에 비해서 컴파일러 생성이 더욱 체계적으로 이루어질 수 있을 뿐만 아니라, 생성된 컴파일러도 효율 면에서 성능이 우수한 것으로 나타났다.

action 의미론에서는 action 메타언어로 의미를 표시하는데, 정보처리 및 계산을 하는데 필요한 연산자들 — 즉 수치의 전달, 연산, 바인딩 설정 및 검색, 저장 장소 할당 및 조작 등 — 을 나타내는 기본 action들과, 그 기본 action들을 복합 action으로 합성하는 action 결합자들로 구성되어 있다. action 의미론적 프로그래밍 언어의 정의는 전형적으로 Abstract Syntax, Semantic Function, 그리고 Semantic Entities의 세 부분으로 나누어지는데, Semantic Function은 Abstract Syntax에서 정의된 각 프로그램 구문의 구조를 그 의미에 해당하는 action으로 팽창(expand or map)하고, Semantic Entities는 Semantics Function에서 쓰이는 여러 가지 보조 action들을 정의한다. 그림 3은 덧셈만 할 수 있는 아주 간단한 탁상용 계산기 언어의 정의이다.

action 의미론을 기초로한 컴파일러 자동생성 시스템에서 팽창기 생성기는 action 의미론으로 쓰여진 프로그래밍 언어의 정의를 입력으로 받아서 action 팽창기를 생성한다. 이 생성된 action 팽창기는 원시 프로그램을 action 프로그램으로 팽창시키고, action 컴파일러는 그 action 프로그램을 목적코드로 컴파일 한다.

```
1. Abstract Syntax
   • Expression=Numoral
                  | [Expression "+" Expression]
   • Numeral=[digit+].
2. Semantic Function
    includes : Action Notation
   • evaluate— :: Expression→action
    (1) evaluate N : Numeral=
          give the natural number of N.
    (2) evaluate[E1 : Expression "+" E2
              : Expression]=
              |evaluate E1 and evaluate E2
              then
              |give the sum of (the given natural#1,
                                the given natural#2).
   • natural number of_ :: Numeral→natural
    (3) the natural number of [d : digit+]=
         decimal of string of[d].
3. Semantic Entities
   includes : Data Notation
```

그림 3 action 의미론에 기초한 간단한 덧셈용 언어의 정의

여기서 action 컴파일러는 시스템의 효율성을 높이는데 중요한 역할을 한다. 수동으로 컴파일러를 작성하는 경우에는, 동작적 의미론(operational semantics)을 기초로 정의된 action과 목적 언어를 참조로 하여 구현할 수 있다. 이 경우 작성된 컴파일러의 정확성 여부는 수학적으로 증명이 가능하다. 그러나, 효율적인 목적코드 생성을 위해서는 여러 가지의 static 분석 및 프로그램 최적화 단계를 거쳐야 한다. 실제로 Peter Orbaek은 Oasis 시스템에서 다양한 데이터 흐름 분석을 통하여 거의 최적화된 C-코드와 비교할 만큼 효율적인 목적코드를 생성하는 action 컴파일러를 만들었다[32].

action 컴파일러는 다음절에서 다룰 부분 계산을 이용하여 자동으로 생성할 수 도 있다. Bondorf 와 Palsberg는 부분 계산기인 Similix[3]를 통해서 양질의 action 컴파일러를 생성할 수 있음을 보였다[4]. 그러나 Similix가 Scheme으로 작성된 관계로 컴파일된 목적코드가 Scheme 프로그램에 국한될 뿐 아니라, 양질의 목적코드를 생성하기 위해서는 Similix의 본질을 잘 이해하여 입력인 action 인터프리터를 binding-time이 향상될 수 있도록 수동적으로 수정해야 한다는 단점이 있다.

action 의미론을 기초로한 최초의 시스템으로는 Glasgow 대학의 David Watt 그룹이 개발한 Actress가 있다[5]. 이 시스템은 다양한 변환 규칙을 사용하여 팽창된 action 프로그램을 대폭 변환함으로서 목적코드의 괄목할 만한 speed-up을 이루었다[30,31]. 실제로 이 시스템은 전통적인 표시적 의미론을 기초로한 시스템들에 비해서, 생성된 컴파일러가 만들어 낸 목적 프로그램은 약 100배 가량 성능이 향상된 반면, 일반적인 상업용 컴파일러와 비교해서는 아직도 100배 가량 차이가 있었다.

또 하나의 흥미 있는 시스템으로서, Jens Palsberg의 Cantor 시스템[33]은 정확도를 증명 가능한 컴파일러를 생성함으로서, 그 복잡성으로 인하여 거의 불가능하다고 여겨져 왔던 컴파일러의 정확도 증명을 비교적 평이하게 할

수 있음을 보여 주었다. 또한 위에서 언급한 Oasis는 이 Cantor에 접합되어, 거의 상업용 컴파일러의 최적화된 목적코드의 성능에 근접하는 (그의 논문의 실험 결과에 의하면 단지 3배 정도만 느림) 코드를 생성하는 컴파일러를 생성할 수 있게 되었다. 그러나 그 시스템은 모든 흐름 분석들이 원시 프로그램이 일단 action 프로그램으로 변환된 이후에 사용되기 때문에, 너무 많은 시간을 컴파일 하는데 소비하는 단점이 있다.

Doh와 Schmidt는 category-sorted algebra를 기초로한 action을 골격으로 하여[13], 컴파일러를 합성하는 방법을 고안했다[9,10]. 각 action마다 분석 함수, 즉 typing 함수 및 binding-time 함수를 할당하여, 프로그래밍 언어의 action 의미론적 정의가 주어졌을 때 typing 분석 및 binding-time 분석이 유추되는데, 이 분석들은 컴파일러 합성시 각각 type checker 및 static semantics로 구현되게 된다. 이렇게 자동으로 유추된 type checker는 합성된 컴파일러가 원시 프로그램의 type error를 감지하는데 사용되고, static semantics는 action 프로그램의 compile-time 계산 부분을 제거하는데 사용된다. 이후 계속된 연구에서 Doh와 Schmidt는 더 체계적인 컴파일러 합성을 가능하게 하기 위하여 각종 요약 해석을 inference rule의 형태로 표현함으로서, 각종 action 의미론적 정의 분석이 "symbolic type expression"을 이용하여 "proof tree"를 구축함으로서 성취될 수 있는 기틀을 마련하였다[11]. 이를 바탕으로, 최근에 Doh는 다음절에서 다룰 부분 계산(partial evaluation) 기술을 이용하여 action을 부분 계산을 통해서 변환하는 방법을 제시하였다[12]. 이 방법은 Actress에서 사용된 방법보다는 더욱 체계적이고 자동적이긴 했지만, 요약 해석(abstract interpretation)의 본질적인 문제점으로 인하여 Actress에 비해서 변환 성능은 떨어졌다. 그러나 특기할 만한 결과 중의 하나는 이 기법은 action 의미론적 정의 자체를 부분 계산할 수 있게 scale-up 될 수 있기 때문에, 필요한 각종 분석들이 원시 프로그램이 주어지기 전, 즉 컴파일러를 생성할 때 행해질 수 있다는 것이다. 다시 말하면, 그 만큼 컴파일 하는 시간을 절약할 수 있다는 것을 뜻한다.

4. 부분 계산에 의한 컴파일러 자동생성

일반적으로 프로그램 p가 입력 in을 받아서 그 결과 out을 출력한다고 하면, 이는 다음과 같이 함수 형태로 표시할 수 있다.

p (in) = out

여기에서 프로그램 p의 입력을 알고 있는 고정된 데이터 s와 모르는 유동적 데이터 d로 나누면 다음과 같이 된다.

p (s, d) = out

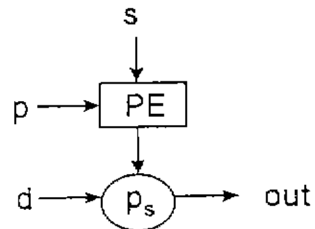


그림 4 부분 계산기의 구조

이 때 고정된 데이터 s 관해서 프로그램 p를 전문화하는 프로그램 변환 기술을 부분 계산이라 한다[21,7]. 부분 계산기 PE는 프로그램 p와 고정된 데이터 s를 입력으로 받아서 s에 관해서 전문화된 프로그램 ps를 생성하는데, 그 프로그램 ps에 유동적 데이터 d가 주어지면 p에 (s,d)가 동시에 주어졌을 때에 얻어질 결과와 같은 결과를 얻게 되는데 다음과 같이 식으로 나타낼 수 있다.

PE (p, s) = ps   where $p_s(d) = out$

이 식을 그림으로 표시하면 그림 4와 같이 되는데, 여기서 부분 계산에 의해서 생성된 프로그램 ps 는 일반적으로 원시 프로그램 p 보다 복잡하지만, 전문화되었으므로 수행 속도면에서 더욱 효율적일 수가 있다.

1970년대 초, Futamura는 위의 p가 인터프리터일 때, 부분 계산을 통해서 컴파일러 및

컴파일러 생성기도 생성할 수 있음을 이론적으로 증명하였다[16]. 다음 식과 같이 인터프리터 int는 프로그램 source와 그 프로그램의 입력인 in을 입력으로 받아서 out을 결과로 출력한다.

int (source, in) = out

여기에서 int를 고정된 데이터인 source에 대해서 부분 계산을 하면 전문화된 프로그램 target이 출력이 되는데, target은 유동적인 데이터 in을 받으면 out을 출력하게 된다.

PE (int, source) = target
where target (in) = out

즉 부분 계산은 프로그램 source를 프로그램 target으로 컴파일 하는 효과를 얻게 된다. 이를 Futamura의 첫 번째 투영이라고 한다. 여기서 다시 PE 자체를 프로그램으로 보고, PE를 int에 대해서 부분 계산을 하면 다음과 같이 컴파일러를 얻을 수 있고, 이를 Futamura의 두 번째 투영이라고 한다.

PE (PE, int) = compiler
where compiler (source) = target

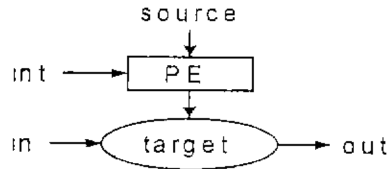
여기서 다시 PE를 자신인 PE에 대하여 부분 계산을 하면 다음과 같이 컴파일러 발생기 cogen을 얻을 수 있으며, 이를 Futamura의 세 번째 투영이라고 한다.

PE (PE, PE) = cogen
where cogen (int) = compiler

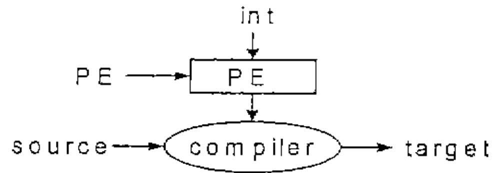
이와같은 Futamura의 투영들은 그림 5와 같이 나타낼 수 있다.

이 Futamura의 투영들은 이론적으로는 증명이 용이하였음에도 불구하고, 실질적인 PE의 구현은 1980년대 중반에 가서야 최초의 부분 계산기(Mix 라고 부름)를 통해서 처음 성공적으로 이루어졌다[20]. 그 이후 지금까지[22] 개발된 대표적인 부분 계산 시스템으로는 ANSI 표준 C 언어를 위한 C-Mix[1], Scheme 언어를 위한 Schism[6]과 Similix[3], Prolog 언어를 위한 Mixtus[36] 와 LogiMix[25], Standard ML을 위한 SML-Mix[2] 등을 들수가 있다.

[1차 투영]



[2차 투영]



[3차 투영]

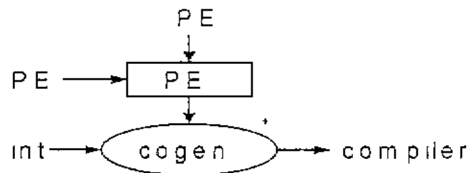


그림 5 Futamura's 투영

5. 결 론

본 논문에서는 프로그래밍 언어의 형식적 정의로부터 컴파일러를 자동 생성하는 방법론과, 그 방법론에 의해서 구현된 여러 시스템들에 대해서 살펴보았다. 이러한 컴파일러 자동 생성 시스템은 설계, 명세서 작성, 모형화, 그리고 구현의 4 단계로 구성되어있는 프로그래밍 언어 개발 절차를 자동으로 진행하게 해서, Pleban과 Lee가 제창한 프로그래밍 언어 설계자를 위한 워크벤취 시스템[24]의 실현이 가능하게 된다. 그렇게 되면 생성된 컴파일러가 설계상의 명세와 일관성을 유지할 수 있고, 설계자는 자신이 설계한 언어를 시험해 볼 수있게 되어서, 신뢰성있는 컴파일러를 제작하는데 큰 일익을 담당하게 되는 것이다.

참고문헌

[1] Andersen L.O., Program analysis and specialization for the C programming language. DIKU, Department of Computer Science, University of Copenhagen. DIKU Report No. 94/19, 1994.

[2] Birkedal L., Welinder, M., Partial Evaluation of Standard ML. DIKU, Department of Computer Science, University of Copenhagen. DIKU Report No. 93/22, 1993.

[3] A. Bondorf, Similix Manual, System Version 5.0, technical report, DIKU, University of Copenhagen, Denmark, 1993.

[4] Anders Bondorf and Jens Palsberg, Generating action compilers by partial evaluation, Journal of Functional Programming, to appear.

[5] Deryck Brown, Hermano Moura, and David Watt, Actress : an action semantics-directed compiler generator, Proc. 4th Int. Conf. on Compiler Construction, Paderborn, Lecture Notes in Computer Science 641, pp.95-109, Springer-Verlag, 1992.

[6] C. Consel, New insights into partial evaluation : The schism experiment, in H. Ganzinger(ed.), ESOP "88, 2nd European Symposium on Programming, Nancy, France, March 1988 (Lecture Notes in Computer Science, vol. 300), pp.236-246, Berlin : Springer-Verlag, 1988.

[7] C. Consel and O. Danvy, Tutorial notes on partial evaluation, in Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993, pp. 493-501, ACM, New York : ACM, 1993.

[8] Stephan Diehl. Semantics-Directed Generation of Compilers and Abstract Machines, PhD Thesis, University Saarbuecken, Germany, 1996.

[9] Kyung-Goo Doh and David Schmidt, Extraction of strong typing laws from action semantics definitions, Proc. European Symp. on Programming, Rennes, Lecture Notes in Computer Science 582, pp.151-166, Springer-Verlag, 1992.

[10] Kyung-Goo Doh and David Schmidt, Action semantics-directed prototyping, Computer Languages, 19(4), pp.213-233, 1993.

[11] Kyung-Goo Doh and David Schmidt, The facets of action semantics : some principles and applications, Proc. 1st Int. Workshop on Action Semantics, Edinburgh, Scotland, BRICS Notes Series NS-94-1, pp.1-15, Univ. of Aarhus, 1994.

[12] Kyung-Goo Doh, Action transformation by partial evaluation, Proc. ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, pp.230-240, June 1995.

[13] Susan Even and David Schmidt, Category-sorted algebra-based action semantics, Theoretical Computer Science, 77, pp.73-96, 1990.

[14] M. Felleisen and R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical Computer Science, 103 : 235-271, 1992.

[15] R.W. Floyd, Assigning meanings to programs, Mathematical Aspects of Computer Science (ed. J.T. Schwartz), American Mathematical Society, Providence, RI, pp. 19-32, 1967.

[16] Y. Futamura, "Partial evaluation of computation process - an approach to a compiler-compiler', Systems, Computers, Controls, 2(5) : 45-50, 1971.

[17] John Hannan, Operational semantics-directed compilers and machine architectures. ACM Transactions on Programming Languages and Systems, 16(4), 1994.

[18] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of ACM 12, pp576-581, 1969.

[19] Neil Jones, Semantics-Directed Compiler

Generation, Lecture Notes in Computer Science 94, Springer-Verlag, 1980.

[20] Neil D. Jones, P.Sestoft, and H. Sondergaard, An experiment in partial evaluation : The generation of a compiler generator, In J.-P. Jouannaud (ed.), Rewriting Technique and Applications, Dijon, France, Lecture Notes in Computer Science 202, pp.124-140, Springer-Verlag, 1985.

[21] Neil D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall International Series in Computer Science, 1993.

[22] Neil D. Jones, MIX ten years after, Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, pp.24-38, 1995.

[23] Giles Kahn, Natural semantics, 4th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 247, Springer-Verlag, Berlin, pp.22-39, 1987.

[24] Peter Lee, Realistic Compiler Generation, Foundation of Computing Series, MIT Press, 1989.

[25] T. Mogensen and A. Bondorf, LogiMix : A self-applicable partial evaluator for Prolog, in K.-K. Lau and T. Clement (eds.), LOPSTR 92. Workshops in Computing, Berlin : Springer-Verlag, January 1993.

[26] Peter Mosses, SIS - semantics implementation system : reference manual and users guide, Technical Report DAIMI MD-30, Aarhus Univ., Aarhus, Denmark, 1979. No longer available.

[27] Peter Mosses, Action Semantics, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[28] Peter Mosses, A tutorial on action semantics, Notes for Formal Methods Europe, Barcelona, October 1994.

[29] Peter Mosses, Theory and practice of action semantics, To be presented at MFCS'96, available at http : //www.brics.dk/Projects/AS.

[30] Hermano Moura, Action Notation Transformation, PhD Thesis, Univ. of Glasgow, 1993.

[31] Hermano Moura and David Watt, Action transformations in the Actress compiler generator, Proc. 5th Int. Conf. on Compiler Construction, Edinburgh, Lecture Notes in Computer Science 786, pp.1-15, Springer-Verlag, 1994.

[32] Peter Orbaek. OASIS : an optimizing action-based compiler generator, Proc. 5th Int. Conf. on Compiler Construction, Edinburgh, Lecture Notes in Computer Science 786, pp.16-30, Springer-Verlag, April 1994.

[33] Jens Palsberg, Provably Correct Compiler Generator, PhD Thesis, Aarhus Univ., 1992.

[34] Lawrence Paulson, A semantics-directed compiler generator, Proc. 9th Annual ACM Symp. on Principles of Programming Languages, pp.224-233, 1982.

[35] Gordon Plotkin, A structural approach to operational semantics, Technical Report, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[36] D. Sahlin, "The Mixtus approach to automatic partial evaluation of full Prolog', in S. Debray and M. Hermenegildo (eds.), Logic Programming : Proceedings of the 1990 North American Conference, Austin, Texas, October 1990, pp. 377-398, Cambridge, MA : MIT Press, 1990.

[37] David Schmidt, Denotational Semantics, W.C. Brown, Dubuque, Iowa, 1986.

[38] J. Stoy, Denotational Semantics - the Scott and Strachey approach to programming language theory, MIT Press. Cambridge, Mass., 1973.

[39] Mitchell Wand, A semantic prototyping system, Proc. ACM SIGPLAN Symp. on

Compiler Construction, pp.213-221, 1984.

도 경 구

1980 한양대학교 산업공학과, 학사
1987 미국 Iowa State University 전자계산학, 석사
1992 미국 Kansas State University 전자계산학, 박사
1993~95 일본 會津大學 講師
1995~현재 한양대학교 전자계산학과 조교수
관심분야 : 프로그래밍언어 의미론, 프로그래밍언어 분석 및 구현, 부분 계산, 함수형 언어 프로그래밍

KOREA-JAPAN Joint Workshop on Algorithm and Computation

- 일 자 : 1996년 8월 23~24일
- 장 소 : 한국과학기술원 전산학과
- 관련분야 : Automata, Languages, Computability
  Combinatorial/Graph/Geometric/Randomized Algorithms
  VLSI/Parallel Algorithms, Networks/Distributed Algorithms
  Theory of Learning/Robotics, Number Theory/Cryptography
  Graph Drawing, Computational Logic
- 주 최 : 한국정보과학회 컴퓨터이론 연구회
  위원장 장직현 교수(서강대 전산학과)
  jchang@alglab.sogang.ac.kr
- 문 의 : 신찬수(KAIST 전산학과)
  cssin@jupiter.kaist.ac.kr
  Tel : 042-869-3553 Fax : 042-869-3510