하스켈로 배우는
함수형 프로그래밍

KANSU PROGRAMMING JISSEN NYUMON
by Noriyuki Ookawa

하스켈로 배우는 함수형 프로그래밍

초판 1쇄 발행 2015년 8월 21일

지은이 오카와 노리유키
옮긴이 정인식
펴낸이 장성두
펴낸곳 제이펍

출판신고 2009년 11월 10일 제406-2009-000087호
주소 경기도 파주시 문발로 141 뮤즈빌딩 403호
전화 070 – 8201 – 9010 / **팩스** 02 – 6280 – 0405
홈페이지 www.jpub.kr / **이메일** jeipub@gmail.com

편집부 이민숙, 이 슬, 이주원 / **소통·기획팀** 민지환, 현지환
표지디자인 미디어픽스 / **본문디자인** 북아이
용지 에스에이치페이퍼 / **인쇄** 해외정판사 / **제본** 광우제책사

ISBN 979-11-85890-29-6 (93000)
값 30,000원

하스켈로 배우는
함수형 프로그래밍

간결하고 올바른 코드 작성법을 배운다

Jpub 제이펍

※ 드리는 말씀

- 이 책에 기재된 내용을 기반으로 한 운용 결과에 대해 저자, 역자, 소프트웨어 개발자 및 제공자, 제이펍 출판사는 일체의 책임을 지지 않으므로 양해 바랍니다.
- 이 책에 등장하는 각 회사명, 제품명은 일반적으로 각 회사의 등록 상표 또는 상표입니다. 본문 중에는 ™, ©, ® 마크 등이 표시되어 있지 않습니다.
- 이 책에서 사용하고 있는 제품 버전은 독자의 학습 시점이나 환경에 따라 책의 내용과 다를 수 있습니다.
- 본문 중 일본 내의 실정에만 국한되어 있는 내용이나 그림은 일부를 삭제하거나 국내 실정에 맞도록 변경하였으니 참고 바랍니다.
- 책 내용과 관련된 문의사항은 옮긴이나 출판사로 연락해 주시기 바랍니다.
  - 옮긴이: insik8463@gmail.com
  - 출판사: jeipub@gmail.com

차 례

Chapter 5

모나드 — 문맥을 지닌 계산을 다루기 위한 장치 _ 278

Chapter 6

추천하는 개발/설계 테크닉

"함수형/Haskell식"의 프로그램 설계/구현, 사고 _ 342

Chapter 7

Haskell에 의한 제품 개발의 길 — 패키지와의 교제 _ 418

옮긴이 머리말

새로운 프로그래밍 언어를 배운다는 것은 저자의 표현처럼 새로운 도구를 자신의 도구함에 넣는 것처럼 무언가를 새로 얻은 뿌듯함을 느끼게 한다. 하지만 이 책은 다른 프로그래밍 언어와는 달리 묘한 매력(?)이 있다.

대부분의 프로그래밍 언어 학습서들은 해당 언어의 역사와 "Hello World"라는 문장을 출력하는 기본적인 프로그램 소스로 책을 시작한다. 역자 또한 이 책을 접할 때 그럴 것이라 예상하였다. 하지만 저자도 머리말에서 언급하고 있듯이, 이 책은 처음부터 다소 난도가 높은 언어 구조와 다른 명령형 언어와의 비교로 시작하고 있다. 앞부분이 프로그래밍 언어의 이론적인 부분에 치중하고 있어서 일일이 이해하고 나가는 데 다소 시간이 걸릴지는 모르겠다. 하지만 여러분이 이를 이해한다면 오히려 함수형 언어의 장단점을 명확히 알 수 있게 될 것이다. 이 책은 특히 수학적인 개념의 함수를 다른 프로그래밍 언어와는 다르게 절차적인 부분과 나누어 명확히 구분하고 있다. 이러한 점은 더 안전한 부품화와 디버깅 작업의 최소화를 극대화한다는 점에서 매력적이다.

이 책은 객체지향 프로그래밍이나 구조적 프로그래밍을 이미 기본적으로 이해하고 있는 프로그래머들이 함수형 프로그래밍을 제대로 이해할 수 있도록 기존 프로그래밍 방법론과 비교 분석하여 함수형 프로그래밍 사고방식을 지니게 해 준다. 따라서 이 책은 초급 프로그래머보다는 다른 언어를 이미 꽤 다뤄 본 경력 있는 프로그래머와 함수형 언어를 공부하려는 학생들에게 체계적인 이론과 실제 도입을 위한 응용력을 기를 수 있도록 도움을 줄 것이다.

부디 이 책으로 많은 분이 함수형 프로그래밍에 대해 깊이 있고 폭넓은 이해를 얻기 바란다.

감사의 글

또 한 권의 책을 번역하게 해 주신 하나님께 감사드린다.

그리고 책의 출간에 관여하신 모든 분께 감사의 마음을 전한다. 특히, 이 책의 교정과 편집, 여러 지원을 아낌없이 해 주신 장성두 실장님과 배규호 님의 수고에 감사의 말씀을 드린다.

끝으로, 사랑하는 나의 아내와 하은, 시온에게도 이 책의 출간에 앞서 기쁨을 함께 나누고 싶다. 이제 여름 휴가철이다! 시원한 해변이나 계곡으로 힐링을 위한 여행을 떠나고 싶다!

일본 동경에서

역자 **정인식**

머리말

'함수형 프로그래밍'은 함수형 언어 프로그래머뿐만 아니라 대부분의 프로그래머도 많이 사용한다.

주요 명령형 언어의 최신 버전에서도 함수형 프로그래밍의 에센스를 도입한 기능들이 존재한다. 즉, 함수형 언어를 사용하지 않는 사람이라 할지라도 사용하는 언어의 새로운 기능을 잘 도입해서 개발을 계속하려면 함수형 프로그래밍을 배울 필요가 있다.

함수형 프로그래밍은 그 이름대로 '함수'를 기본 단위로 구성한 함수의 조합으로 프로그램을 구성한다. 함수형 프로그래밍이라는 스타일 자체는 아마도 실용적인 언어라면 어떤 언어로도 구현 가능할 것이다. 하지만 아무래도 함수형 프로그래밍을 위해 만들어진 함수형 언어가 함수형 프로그래밍에 제격일 것이다.

주변에서 들어본 적이 있을지도 모르겠지만, 함수형 프로그래밍에는 다양한 소문이 존재한다.

- "간결한 코드로 작성할 수 있다"
- "버그가 발생하기 어려워서 안전하다"
- "병렬화하기 쉽다"

아니 땐 굴뚝에 연기가 날 리 없겠지만, 이러한 소문들이 생겨나는 데에는 그 나름의 이유가 존재한다.

특히, 이 책이 집필된 2014년에는 널리 사용되는 오픈 소스 소프트웨어의 심각한 취약점이 연달아 발견되었다. 혹시 여러분 중에도 이에 대응하느라 분주했던 경험이 있을지도 모르겠다.

물론, 'if'문으로도 전혀 할 수 없는 것은 아니지만, 복잡한 사양(specification)임에도 코드를 간결하게 유지하기 쉬운 언어, 아니면 프로그래머의 사소한 실수도 민감하게 잘 감지해 내는 안전한 언어가 사용되었다면, 아마 몇 개의 취약점은 막아 낼 수 있었을지 모른다. 만약 위와 같은 특성의 소문이 진실이라면, 함수형 프로그래밍은 전 세계 사람들이 취약점에 대응하느라 낭비한 시간을 절약할 수 있었을 것이다.

이 책은 함수형 프로그래밍과 함수형 언어란 도대체 어떤 것인지를 명확하게 알려준다. 그리고 이 책 속에서는 앞에서 언급한 바와 같은 소문이 생기게 된 이유를 설명한다. 최종적으로, 여러분의 도구함 속에 함수형 프로그래밍의 기초라는 도구를 하나 더 추가해 준다. 이 도구는 실제로 함수형 언어를 사용하는 경우뿐만 아니라, 여러분이 일반적으로 사용하는 언어에 이르기까지 다양하게 활용 가능한 도구가 될 것임이 틀림없다.

함수형 프로그래밍이나 함수형 언어에 관한 서적은 이미 서점에 몇 권 나와 있다. 이 책에서는 함수형 언어의 설명에 Haskell(하스켈)을 사용한다. 그리고 함수형 프로그래밍의 기초 외에도 다음의 두 가지에 주력해서 설명한다.

- 함수형 언어에서의 설계 방법/사고 방법
- 다른 언어와 함수형 언어의 비교 대조

이 책의 대상 독자는 지금까지 다른 언어를 나름대로 사용해 보았으며 앞으로 함수형 프로그래밍/함수형 언어에도 흥미가 있는 프로그래머와 학생들이다.

이 책에서는 잘 알려진 언어와 비교하여 함수형 언어가 갖는 장단점이 명확히 드러나도록 하였다. 단순한 비교만이 아니라 함수형 언어의 특정한 유효 기능을 별개의 주요 언어에 도입하려고 했을 때 어떻게 하면 좋은지도 설명한다.

참고로, '함수형 프로그래밍에서의 사고 방법'은 잘 알려진 구조적 프로그래밍이나 객체지향 프로그래밍의 사고 방법과는 꽤 다르다.

다른 함수형 프로그래밍 관련 서적의 경우, '어떻게 생각해서 작성하면 좋은가'에 대한 설계나 사고 방법의 설명이 머릿속에 확 와 닿지 않았을지도 모르겠다. 작성 방법을 이해했어도 생각이 뒤따르지 않으면 프로그램을 작성하기 어렵다. 따라서 이 책에서는 함수형 언어를 가지고 함수형 프로그래밍을 할 때 어떤 식으로 생각을 진

행해 나가야 함수형 언어 본연의 간결한 코드가 되는지를 설명한다.

부디 독자 여러분이 이 책을 통해 안전한 프로그램을 만드는 지식을 손에 넣어 함수형 언어를 가지고, 또는 여러분이 사용할 언어에 이 책의 지식을 도입함으로써 더 나은 미래를 구축하는 데 보탬이 되었으면 한다.

감사의 글

이 책을 집필하는 데 특히 히비노 케이 님과 후지무라 다이스케 님에게 유익한 조언을 받았다. 마음속 깊이 감사드린다.

또한, Haskell을 이용한 기술 도입, 관련 연구회 개최, 이 책의 집필 등에 대해 너그럽게 이해해 준 나의 직장 아사히 넷과 그곳의 동료 여러분, 그리고 스킬 향상에 도움되는 연구회를 개최하여 적극적으로 함수형 프로그래밍을 학습하고 있는 프로그래밍 연구회 회원들, 특히 Haskell과 관련된 많은 분들에게 진심으로 감사의 말씀을 전하고 싶다.

오카와 노리유키

이 책의 구성

이 책의 각 장은 다음과 같이 구성되었다.

● 0장 **[입문] 함수형 프로그래밍** — "함수"의 세계

제0장은 도입부다. 함수형 프로그래밍과 그 실현을 위한 함수형 언어는 도대체 어떠한 것인지를 설명한다. 이와 더불어 함수형 프로그래밍 분야의 사람들과 대화가 가능한 정도의 기본적인 개념과 특징, 용어, 역사 등을 소개한다. 다양한 함수형 언어와 해당 언어로 작성된 제품 소개 등도 덧붙인다.

● 1장 **[비교를 통해 발견하기] 함수형 프로그래밍** — C/C++, JavaScript, Ruby 그리고 Haskell

1장에서는 동일한 문제 설정하기에서 주요 언어 몇 가지와 Haskell을 비교함으로써 Haskell(또는 동등한 함수형 언어의 기능)의 기능이 프로그래밍에 미치는 효과를 파악한다. 아직 이 시점에서는 Haskell의 문법을 설명하지 않으나, 비교와 대조로 Haskell 코드를 설명한다. 처음으로 Haskell를 접하는 사람은 일단 1장을 건너뛰고 다음 장인 2장부터 5장까지를 읽은 다음에 되돌아오는 것도 좋다.

● 2장 **타입과 값** — "타입"은 기본 중의 기본

● 3장 **함수** — 함수 적용, 함수 합성, 함수 정의, 재귀 함수, 고차 함수

● 4장 **평가 전략** — 지연 평가와 적극 평가

● 5장 **모나드** — 문맥을 지닌 계산을 다루기 위한 장치

2장에서 5장까지는 Haskell을 비롯한 많은 함수형 언어의 안전성에 크게 공헌한 '타입', Haskell 코드를 읽고 쓰기 위한 기본 문법과 Haskell과 다른 언어에서 실제로 계산이 진행되는 구조적 차이점, 다른 명령형 언어에서 도입되기 시작한 모나드라는 특징적 기능에 대해 상세히 설명한다.

● 6장 **추천하는 개발/설계 테크닉** — "함수형/Haskell식"의 프로그램 설계/구현, 사고

6장에서는 Haskell의 추천 설계법에 대해 그 사고 과정을 알 수 있도록 단계별로 설명한다. 무엇이 Haskell식의 코드이며, 어떻게 생각하면 그러한 코드가 작성 가능한지를 파악할 수 있을 것이다. 이 책을 읽는 독자가 함수형 언어에 대한 경험이 있다면 5장까지를 건너뛰고 바로 6장부터 읽기 시작하는 것도 좋다.

● 7장 **Haskell에 의한 제품 개발의 길** — 패키지와의 교제

7장에서는 실제로 Haskell을 사용하여 제품을 만들 때 어떤 코드 베이스나 라이브러리를 관리해 나가면 좋은지에 대해 다루고 있다. 여러분의 프로그램을 오픈 소스로 공개하거나 업무에서 Haskell을 사용하거나 할 경우에 도움이 될 것이다.

● 부록

함수형 언어를 사용할 수 있는 프로그래밍 콘테스트 사이트와 실력 향상을 위해 읽어 두어야 할 참고문헌을 소개한다.

이 책에 필요한 사전 지식

이 책에서는 다음과 관련된 사전 지식이 필요하다.

- **주요 명령형 언어에 대한 경험과 지식**
  - Java, C언어, C++, Ruby, Python, JavaScript 등
  - Java 8, C++ 11까지 접해 봤다면 더 좋음
- **구조화 프로그래밍이나 객체지향 프로그래밍의 기본 사항**
  - GoF(Gang of Four)의 디자인 패턴 등
- **유닉스 계열의 OS나 윈도우즈의 기본 조작**
  - 명령 프롬프트에서의 조작
  - 각 언어의 개발, 실행 환경

위의 첫 번째, 두 번째는 비교 및 개념의 대비를 위해 필요하다. 세 번째는 Haskell이나 다른 언어의 샘플 코드를 실제로 테스트해 보기 위해서 필요하다.

베타리더 후기

김용균(이상한모임)

함수형 프로그래밍의 전반적인 배경과 더불어 Haskell의 문법과 실제 적용까지 짚어 주는 책입니다. 같은 코드를 명령형 프로그래밍 언어로도 구현/비교하고 있어서 특히 명령형 프로그래밍에 익숙한 사람이 함수형 프로그래밍을 이해하는 데 도움이 될 것 같습니다. 또한, 그 비교를 통해 함수형 프로그래밍이 지닌 장점을 확연하게 파악할 수 있어 좋았습니다.

김종욱(한국기술교육대학교)

이 책에는 함수형 프로그래밍이 무엇인지, 그리고 이를 통해 독자가 얻을 수 있는 것이 무엇인지가 잘 요약되어 있습니다. 함수형 언어인 Haskell을 이용하여 다양한 예제를 구현한 것은 이 책만이 가지는 독특한 묘미이며, 또한 예제를 독자 스스로 구현하며 익히게 하는 방식 역시 함수형 프로그래밍을 처음 접하는 독자에게 큰 도움이 될 것이라 생각됩니다.

김지헌(이노트리)

이 책은 조금 어려웠습니다. 책이 어렵다기보다는 '함수형 프로그래밍'이 어렵다는 느낌적(?) 느낌을 받았습니다. 그래도 인내력을 발휘하며 한 장 한 장 읽다가 '재귀 함수', '지연 평가', '무한의 데이터'에 들어서면서 나도 몰래 '와우!'를 외쳤습니다. 함수형 프로그래밍에 살짝 맛을 봤으니 천천히 함수형 프로그래밍을 음미하면서 내 것으로 만들어 갈까 합니다. 처음은 어렵지만 꾸준히 하다 보면 분명 내 것이 될 거라 믿기에…….

송영준(줌인터넷)

함수형 패러다임은 프로그래밍 세계에 영향을 끼치고 있습니다. Java와 C++에서도 그 대세에 따라서 Lambda 표기법을 지원한 지 몇 년이 지났습니다. 과거 스몰토크가 많은 언어의 객체지향 설계에 영향을 준 것처럼 Haskell도 많은 언어의 함수형 설계에 영향을 줄 것입니다. 이 책에서는 가장 순수한 함수형 언어 중 하나로 평가받는 Haskell로 여러분을 함수형 프로그래밍의 세계로 인도할 것입니다. 비록 내용은 좀 어렵지만요. :)

장효진(다음카카오)

함수형 프로그래밍이 주목받는 이유에 관한 최신 이야기를 들을 수 있어서 좋았습니다. 기존의 명령식 프로그래밍에 익숙하지만 본인이 작성하는 프로그램에서 무결함을 달성하는 데 다소 버거움을 느끼는 독자라면, 저자가 소개하는 세련된 프레임이 신선한 자극이 될 것이라 생각합니다.

차성호

이 책을 읽기 전에 함수형 책을 몇 권 봤습니다만, 대부분 다른 패러다임의 개발자들 위주로 설명되어 있어서 친절하지만 사고방식을 바꾸기가 쉽지는 않았습니다. 이 책은 명령식 프로그래밍에 익숙한 개발자들이 이해하기 쉽도록 비교를 위한 예제 코드를 보여 주고, 함수형 프로그래밍을 개념부터 시작해서 수학적 정의를 하듯이 설명하고 있습니다. 그래서 다른 함수형 언어를 대충은 알고 있지만 익숙하지 않은 저 같은 사람들이 (그래도 쉽지는 않겠지만) 사고방식을 전환하는 데 도움이 될 것입니다.

제이펍은 책에 대한 애정과 기술에 대한 열정이 뜨거운 베타리더들로 하여금
출간되는 모든 서적에 사전 검증을 시행하고 있습니다.

CHAPTER

0

[입문] 함수형 프로그래밍

"함수"의 세계

“함수”의 세계

최근 들어 “함수형 프로그래밍” 및 “함수형 언어”라는 단어가 자주 언급되고 있다. “개발 효율이 좋고 버그가 적다”라는 캐치프레이즈와 함께 함수형 프로그래밍에 관한 기사도 눈에 띄게 많아졌는가 하면 이미 함수형 언어에 의해 만들어진 제품도 출시되고 있다(자세한 내용은 나중에 설명).

본래 전산학에서 연구 대상이었던 함수형 프로그래밍은 더 이상 일부 연구자들만의 전유물이 아니다. 함수형 언어는 제품 개발을 위해 충분히 실용적인 선택 요소 중 하나가 되었고, 이제는 잘 알려진 명령형 언어들과 함께 어깨를 나란히 하여 우리 앞에 존재한다.

그렇다면 “함수형 프로그래밍”이란 실제로 무엇일까? 그리고 “함수형 언어”의 “함수”란 무엇일까? 이것은 명령형 언어에서 말하는 함수와 같은 것일까? 이와 다른 어떤 차이가 있을까? 정말로 개발 효율이 좋아지거나 버그가 적어질 수 있다면, 그 근거는 도대체 어디에 있는 것일까?

이 장에서는 함수형 프로그래밍의 세계를 조망해 보기 위해, 우선 함수형 프로그래밍 및 함수형 언어의 특징을 명확히 하고자 한다.

0.1 함수형 프로그래밍, 그 전에
실용 프로그램에서 활용하는 강점 알기

함수형 프로그래밍이나 함수형 언어를 배우는 데 있어서 우선 동기를 명확히 하자. 실리를 이길 수 있는 명분이란 좀처럼 없을 테니 말이다.

함수형 프로그래밍을 통해 얻을 수 있는 개선

먼저 함수형 프로그래밍과 함수형 언어를 배워 사용하면 어떤 일이 가능할까? 그 개요를 알아보도록 하자.

▶ **코드양이 적어진다**

어느 정도의 기능을 갖춘 프로그램을 일반적인 수준으로 작성해 보면, 훨씬 적은 코드양으로 끝마칠 수 있다는 점에 놀라게 될 것이다. 이것은 함수형 프로그래밍의 경우, 고도로 유용한 추상화 능력을 가진 것으로 알려진 "수학"이라는 분야의 개념을 그대로 프로그래밍에 적용할 수 있기 때문이다. 코드양이 적다는 것은 무엇보다 높은 유지 보수성에 직접적인 영향을 미친다.

▶ **최적화하기 쉽다**

최적화는 이미 많은 프로그래머들의 노력을 통해 얻어지는 일이 아니라 컴파일러(compiler)가 자동으로 해 주는 일이다. 따라서 처리계가 최적화를 위해 사용할 수 있는 유용한 특징을 많이 갖추고 있을수록 프로그램에서 효율을 신경 쓰지 않아도 고효율의 프로그램이 자동으로 생성될 수 있다. 여기에서도 함수형 프로그래밍의 고급 추상화 능력이 도움이 된다.

▶ **병행/병렬화하기 쉽다**

CPU의 진화의 경우, 과거에는 단순히 동작 주파수가 빨라지는 식이었지만, 현재는 코어가 늘어나는 방향으로 가고 있다. 즉 어떤 프로그램이라도 CPU가 새롭게 바뀌면 왠지 모르게 빠르게 처리되던 과거와는 달리, 병행/병렬화된 프로그램이 아니라면 CPU를 새롭게 바꾸어도 빨라지지 않는다. 제대로 병행/병렬화된 프로그램이라면 (향후 언급) 프로그램의 변경 없이 코어를 충분히 다 사용하게 될 것이다.

▶ **버그가 발생하기 어렵다**

일부 함수형 언어에서는 꽤 고도의 제약 조건을 "타입(type)"으로 표현하여 그 제약 조건이 지켜지고 있는지를 컴파일 시에 체크할 수 있도록 하였다. 자신이 작성한 라이브러리를 다른 사람이 사용할 경우, 또는 몇 년 전 자신이 작성한 프로그램을 유지보수할 경우 등에 잘못된 사용법이 발생하지 않도록 미연에 방지할 수 있다.

▶ **문서가 적어진다**

고도의 제약 조건이 지켜지는지 컴파일 시에 검사할 수 있다는 것은 그 제약 조건에 대해 문서화가 필요 없다는 의미를 내포한다. 지켜야 할 제약 조건을 일부러 문서에 설명할 정도라면 그 시간에 제약 조건을 표현하기 위한 프로그래밍을 하는 편이 더 좋다. 문서로 다시 사용자에게 타입 정보를 알려 주는 것보다 검사를 통해 금지시키는 것이 좋기 때문이다.

● ● ● ● ● ● ● ● ●

일단 실용 프로그램에 종사하는 사람 중에서 위와 같은 항목에 흥미 없는 사람은 없을 것이다. 이후에서는 함수형 프로그래밍과 함수형 언어를 시작하면서 위와 같은 항목에 대해서 적절히 다루어 보겠다.

0.2 함수란 무엇인가?
명령형 언어의 함수와 무엇이 다른가?

함수형 프로그래밍이나 함수형 언어에서 가장 특징적인 것은 "함수(function)"다. 우선은 이것들이 취급하는 함수란 도대체 어떠한 것인지에 대해서 살펴보도록 하자.

함수형 프로그래밍에서 함수

함수형 프로그래밍에서 "함수"란 무엇일까? 명령형 언어[주1]에서 함수는 특정 명령/절차의 나열에 붙은 라벨[주2]에 지나지 않는다. 이에 반해 함수형 프로그래밍에서 함수는 "주어진 입력 값만으로 단지 하나의 출력되는 값을 결정하는 규칙"이라는 수학적 의미의 함수다.[주3]

예를 들어, 다음의 say.cpp의 say는 함수형 프로그래밍의 함수가 아니다.

```
// C++
// say.cpp
// 문자열을 표준 출력으로 표시한다
void say(const std::string& something) {
    std::cout << something << std::endl; // 외부로 입출력을 실시한다
}
```

위의 코드에서 say는 외부(표준입출력)로 문자 출력을 한다. 외부로의 입출력은 당연히 외부의 상황에 의존하므로 항상 같은 결과가 나온다고 단정할 수 없다.

또 하나의 예를 보자.

```
# Ruby
# current.rb
# 현재 시각을 취득한다
def current
  Time.now # 주어진 입력값 이외에서 출력이 결정되고 있다
end
```

주1 컴퓨터가 실행해야 할 명령문의 나열에 의한 프로그램을 기술하는 언어다. 절차형 언어라고도 부른다. C언어나 Java, Perl이나 Ruby, Python 등 현재 주류인 언어는 거의 명령형 언어다.

주2 덧붙이자면, 여기서의 라벨이란 어셈블러에서 이야기하는 "라벨"을 뜻한다. 즉, 여기서는 단순히 명령열의 선두, 또는 명령열의 블록을 식별하기 위한 "이름"이라고 생각해도 좋다.

주3 함수형 프로그래밍 및 함수형 언어와 수학과의 관계는 나중에 다시 이야기하겠다(0.6절의 "함수형 언어의 추상화").

위의 소스에서 current는 외부에서 현재 시각을 가져온다. current에는 아무것도 인수를 부여하지 않으므로, 주어진 입력값만으로 단 하나의 출력값이 결정되는 수학적 의미의 함수라면 항상 같은 값이 되어야만 한다. 하지만 실제로 current는 때에 따라 결과가 매번 다르다.

다음의 show.cpp의 show는 함수형 프로그래밍의 함수라고 말해도 상관없을 것이다.

```
// C++
// show.cpp
// 정수 값 n을 문자열로 변환한다
// show(1234) => "1234"
std::string show(const int n) {
    std::ostringstream oss;
    oss << n;
    return oss.str();
}
```

위의 코드에 있어서 show는 주어진 숫자에 대해서만 그 값의 문자열 표현을 얻는다. show는 동일 숫자를 부여하면 언제 어디서나 동일한 문자열 표현을 얻을 수 있다. show는 수학적인 의미에서 함수다.

```
# Ruby total.rb
# numbers 내부 값을 전부 더한다
# total([1,2,3]) => 6
def total(numbers)
  numbers.inject(:+)
end
```

위의 코드에서 total은 주어진 숫자의 열만을 통해 그 합계를 얻는다. total은 언제 어디서 사용해도 동일 숫자의 열을 부여하면 동일한 합계를 얻을 수 있다. total은 수학적인 의미에서 함수다.

이후 이 책에서는 단순히 “함수”라고만 적혀 있을 경우, 수학적인 의미에서 함수를

의미하는 것으로 간주하겠다. 이에 반해 함수가 아닌 것, 예를 들어 명령형 언어에서의 "(이른바)함수"는 이 책의 경우, "절차"라고 부르겠다.

부작용

프로그래밍에서 "상태"를 참조하거나 "상태"에 변화를 줌으로써 다음 번 이후의 결과에까지 영향을 미치는 효과를 **부작용**(side effect)이라고 한다.

예를 들어, 대입은 부작용이다.[주4] 변화하는 변수의 내용의 경우, 그 시점에서 어떠한 값을 갖고 있느냐는 하나의 "상태"로 표현될 수 있다. 그리고 이것은 다음 참조 시에 이전과 다른 "상태"의 결과를 얻을 수 있을지도 모른다.

예를 들어, 입출력은 언어의 외부 세계[주5]의 "상태"에 간섭하므로 부작용이 된다. 파일 쓰기 처리는 물론이며, 파일의 내용을 읽어 오는 처리라고 할지라도 물리 디스크로부터 읽기 처리를 하는 물리적인 동작[주6]을 발생시킬 수 있다. 그리고 그 동작에 의해 물리 디스크 자신이 파손될지도 모른다. 그렇게 되면, 그 후 동일한 읽기 처리를 한다 해도 동일한 결과를 얻지 못한다.[주7]

부작용을 지닌 절차는 앞서 언급한 "함수"가 아니다. 함수형 프로그래밍에 대해 배운 이상, 프로그래밍이 대상으로 하는 처리 내용에 대해 그림 0.1과 같은 부작용을 구별하여 생각하는 것이 중요하다.

주4 특히, 전역 변수의 참조/변경은 가장 성가신 부작용이다.

주5 즉, 처리의 세계가 아닌 우리가 사는 현실 세계.

주6 자기 헤드 구동이나 플래터(platter)의 회전.

주7 그전에 I/O 오류가 발생할 것이다.

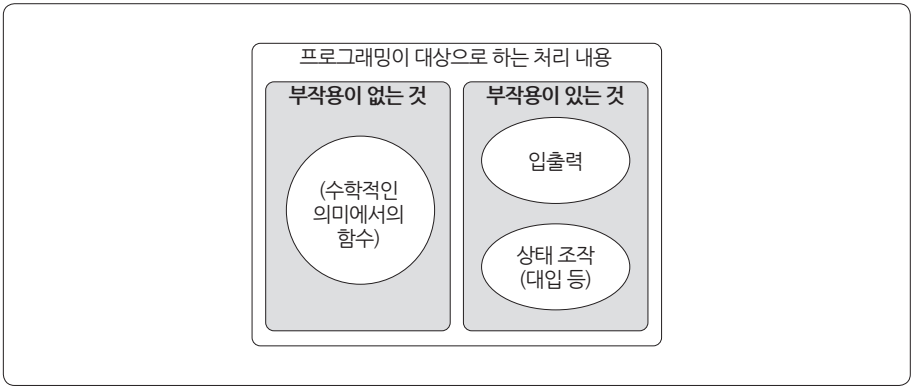

그림 0.1 프로그래밍이 대상으로 하는 처리

0.3 함수형 프로그래밍이란 무엇인가?

"프로그램이란 함수다"라는 관점

함수형 프로그래밍(functional programming)이란, 값을 가지고 앞 절에서 설명한 함수를 적용해 계산을 진행하는 프로그래밍 스타일이다.

프로그래밍의 패러다임

프로그래밍에서 프로그래밍 패러다임(programming paradiam)이란, 프로그램이나 그것에 대상이 되는 문제 자체를 "어떤 식으로 볼 것인가"라는 것이다.[주8] 대개 한 개의 언어는 한 개의 패러다임을 갖는다.[주9]

프로그래밍에 국한하지 않고 사물을 다른 관점에서 파악한다는 것은 어려운 일이

주8 "망치를 갖고 있는 사람에게 모든 것은 못으로 보인다"라는 의미에 가깝다.

주9 복수의 패러다임으로 프로그래밍할 수 있는 언어를 멀티 패러다임이라고 부른다.

다. 함수형 프로그래밍이 어렵다는 인상을 주는 이유는 많은 프로그래머가 초창기에 배운 언어의 프로그래밍 패러다임과 함수형 프로그래밍 패러다임이 다르기 때문이다.[주10]

그럼 명령형 및 객체지향의 패러다임을 확인한 후에 함수형 프로그래밍의 패러다임이란 어떠한 것인지 살펴보자.

● 명령형 프로그래밍의 패러다임

명령형 프로그래밍의 경우 "프로그램은 컴퓨터가 수행해야 할 명령의 나열"이다. 프로그래밍은 명령을 적절한 순서로 나열하는 것으로서 나열한 명령을 순서대로 실행함에 따라 문제를 해결하는 기능을 실현한다.

● 객체지향 프로그래밍의 패러다임

객체지향 프로그래밍의 경우 "프로그램은 객체와 그것들과의 메시징"이다. 프로그래밍은 문제에 어떤 객체가 있는지를 찾아내고 정의하는 것이며, 객체를 생성/관리하여 객체끼리 메시지를 교환함으로써 문제를 해결하는 기능을 실현한다.

● 함수형 프로그래밍의 패러다임 — 프로그램은 "함수"다

한편, 함수형 프로그래밍에서는 "프로그램은 '함수'다"라고 생각한다. 그리고 커다란 프로그램은 작은 프로그램의 조합으로 구성된다. 큰 프로그램은 큰 함수, 작은 프로그램은 작은 함수라고 했을 때, 프로그램의 조합은 **함수 합성**(function compostion)이 된다. 여기서 말하는 함수 합성도 수학에서의 함수 합성과 동일한 것으로, 어떤 함수 f와 함수 g가 있을 때 h(x)=f(g(x))라는 결과를 산출해 내는 새로운 함수 h를 만드는 것을 말한다.

함수 h가 함수 f와 함수 g를 합성한 것일 때, h를 f ◦ g라고 쓰고, (f ◦ g)(x)=f(g(x))로 표현할 수 있다. 이렇게 조립된 함수를 적용함으로써 문제를 해결하는 기능을 실

주10 바꿔 말하면, 그 정도의 이유밖에 없다고 할 수 있다.

현한다.

"프로그램이 작은 부품의 조합으로 구성된다"라는 것은 특정 언어를 사용하고 있는 프로그래머일지라도 동일하게 인식할 것이다. 이때, 조합의 용이성/부품으로서의 독립성을 **모듈화**(modularity)라고 부른다. 함수형 프로그래밍은 함수와 함수 합성이 가지는 특성에 따라 모듈화의 성능 향상을 보장한다.

함수가 갖는 모듈화
— "프로그램을 구성하는 부품"의 독립성

함수형 프로그래밍에서 **함수**는 앞에서 언급한 바와 같이 입력만으로 출력이 결정되어 그 외의 요인에 의존하지 않는, 즉 **부작용**이 없으므로 "프로그램을 구성하는 부품"으로서 독립성이 양호하다.

함수에 입력 가능한 값 전체로 이루어진 집합을 **정의역**(domain), 출력 가능한 값 전체로 이루어진 집합을 **치역**(codomain)이라고 한다. 예를 들어, 문자열의 길이를 구하는 함수라면 정의역은 문자열 전체, 치역은 자연수[주11] 전체다.

x값이 함수 f의 정의역에 포함되어 있으면 x에 f를 적용할 수 있으며, 함수 f의 치역이 함수 g의 정의역에 들어가 있으면 함수 f와 g는 문제없이 합성할 수 있다.

함수형 프로그래밍에서 함수에 해당하지 않는 것은 일반적으로 모듈화에 좋지 않다. 예를 들어, 치역과 정의역이 일치하는 함수 f와 g가 있다고 하자. 즉, f(g(x))이든 g(f(x))이든 호출 자체에는 문제가 없다. 단, f는 내부에서 특정 리소스의 초기화를 실시하며 g는 내부에서 f와 같은 리소스를 초기화된 것으로 가정하여 이용하려고 한다.

즉, 함수인 것처럼 f와 g를 사용하지만, 이것들이 실은 절차였다는 사실이다. 이때, f를 적용한 후 g를 적용하는 조합이 가능하지만, g를 적용한 후 f를 적용하는 식의 조합은 할 수 없다. 따라서 치역과 정의역으로만 함수를 부품으로 결합할 수 없기

주11 0을 포함한다.

에 f와 g 사이에는 문맥[주12]이 존재한다.

이번 절의 서두에서 언급한 바와 같이 함수형 프로그래밍에서는 절차가 아닌 함수를 처리한다. 따라서 "수학적인 의미에서 함수만 사용하기=절차를 취급하지 않기"라는 제약이 있게 된다.

0.4 함수형 언어란?
함수가 1급(first class) 대상이다? 대입이 없다?

계속해서 함수형 프로그래밍을 실현하는 함수형 언어(functional language)란 어떠한 것인가에 대해 살펴보도록 하자.

함수형 언어이기 위한 조건

일반적으로 함수형 언어란 **함수가 1급(first class) 대상인 언어**를 말한다. "1급 대상이다"라는 것은 해당 언어에서 단순한 값, 즉 아래와 같은 특징이 있다는 것이다.

- 리터럴(literal)이 있다.
- 실행시간에 생성할 수 있다.
- 변수에 넣어서 취급할 수 있다.
- 절차나 함수에 인수로서 제공할 수 있다.
- 절차나 함수의 결과로서 반환할 수 있다.

함수가 1급의 대상이라는 의미는 별다른 것[주13]을 하는 것이 아니라 "함수를 값과

주12 여기에서는 정의역과 치역만이 아닌 배후에 숨겨진 무언가를 "문맥"이라고 부른다. 함수형 언어에는 문맥을 프로그래밍할 수 있는 것도 있는데, 예를 들어 F#의 경우, "컴퓨테이션식(compuation expression)", Haskell이라면 "모나드(monad)"로 취급한다. 모나드는 제5장에서 다루겠다.

주13 함수 포인터를 취급하거나 함수 객체를 만들거나 하는 것.

동일하게 취급하는 기능을 갖는 것"을 말한다. 즉, 함수와 값에 구분이 거의 없다.

함수가 1급 대상이라는 것이 어떤 것인지 앞 절의 항목에 대해 각각 설명하겠다.

● 함수 리터럴이 있다

문자열이나 값에 리터럴이 있듯이 함수형 언어에서는 함수에도 리터럴이 주어진다. 자주 나타나는 것이 람다식(λ식, 나중에 설명)에 의한 것이다. 언어에 따라 람다식의 기법은 다르므로 여기서는 이해를 위해 조금 설명하겠다. 예를 들어, 인수에 1을 더하는 함수는 람다식 (λ x. x + 1)으로 나타내는 식이다. 2를 이 함수에 적용하면 (λ x. x + 1)(2)로 2+1이 된다.

● 함수를 실행시간에 생성할 수 있다

함수형 언어에서는 함수를 실행시간에 생성할 수 있는데 그 방법이 다양하다. 함수 합성에 의해 만들어질 수도 있고, 부분 적용(뒤에 설명)이나 고차 함수(뒤에 설명)에 의해 만들어질 수도 있다. 람다식에 의해서도 만들어질 수 있다.

● 함수를 절차나 함수에 인수로서 제공할 수 있다

정의역이 특정 함수의 집합이 되는 함수를 만들 수 있다. 예를 들면, 함수를 인수로 취하고 그 함수에 1을 적용한 결과가 되는 함수를 만들 수 있다. (λ f. f(1))와 같은 식으로 할 수 있다.

● 함수를 절차나 함수의 결과로서 반환할 수 있다

치역이 특정 함수의 집합이 되는 함수를 만들 수 있다. 예를 들어, 값을 하나 가져다가 그 값을 더하는 함수가 되도록 함수를 만들 수 있다. 람다식으로 표현하면 (λ a. (λ b. a + b))와 같은 식이다. 실제로, 1에 이 함수를 적용하면 (λ a. (λ b. b + a))(1)이 (λ b. b + 1)이라는 결과가 된다. 즉, 1을 더하는 함수다.

하지만 최근에는 기존의 명령형 언어에서도 함수형 언어의 특징을 도입하려는 언어도 존재하므로 이 요구 사항만을 가지고서 함수형 언어의 특징이라고 하는 것은

어렵다.

예를 들어, JavaScript 등은 이전부터 함수가 1급의 대상으로 되어 있으며, Java나 C++도 각각 Java8과 C++11에서 람다식을 도입하는 등 점차 함수형 언어의 기능을 도입하려는 부분이 있는 것 같다.

물론 함수형 언어의 특징을 가져온 언어로도 함수형 프로그래밍을 할 수 있다. 그러나 이는 활용하기 쉬운 것 그리고 유용한 것과는 별개의 문제[주14]라고 생각한다.

함수형 언어와 명령형 언어

프로그래밍에 있어서 처음부터 함수형 언어를 시작한 사람은 비율상 그리 많지 않을 것이다. 많은 사람들이 명령형 언어를 먼저 접할 것이라고 생각하는데, 함수형 언어와 명령형 언어 사이에는 큰 차이가 있다.

명령형 언어에서는 결과를 달성하기 위해 CPU 또는 처리계의 저수준의 동작을 주된 부분으로 작성할 필요가 있다. 구체적으로 말하자면 다음과 같다.

- 값을 어디에 보관할 것인가(대입).
- 어디로부터 값을 가져올 것인가(참조).
- 다음에 어떤 절차로 진행할 것인가(절차의 호출).

함수형 언어에서는 특정 결과를 달성하기 위해 그 결과의 성질만을 선언한다. 그리고 그 성질의 충족 자체는 처리계[주15]에 달려 있다.

"선언적이다"라는 것은 출력의 성질이 어떤 것인가에 주목하고 그것만을 기술하는 것이다. 예를 들어, 정렬을 처리함에 있어서 "배열의 첫 번째 요소와 그 다음 요소를 비교하여 다음의 요소가 작으면 교체, 그러고 나서 그 후의 처리(이하 생략)"와 같은 식으로 작성된다면 선언적이지 않다고 말할 수 있다. 만일 "배열의 특정 위치의 요소는 그 이후의 요소보다도 작아야 한다"라고만 기술한다면 이는 선언적이 된다.

예를 들어, 많은 명령형 언어에서는 메모리 모델을 직접적으로 취급한다. 그 대표

주14 C언어로도 객체지향 프로그래밍을 할 수 있다는 점과 동일하다고 말해도 과언이 아닐 것이다.

주15 컴파일러, 인터프리터(interpreter), 런타임(runtime) 등.

적인 동작의 하나로서 "변수에 값 대입"이 있다. 이것은 그림 0.2와 같이 저장 매체를 다루는 언어가 충분히 추상화해 주지 않아서 메모리에 대한 매우 저수준의 조작이 대입이라는 언어의 기본 조작으로 프로그래머에게 보이는 것이다. 처리 절차를 기술한다는 이유로 값을(메모리 또는 레지스터 등의) 어디에 둘 것인지를 강하게 의식시키거나 또는 그것을 서술하도록 요구하는 것이다.

명령형 언어를 배울 때는 변수를 "값을 집어넣는 상자"의 개념으로 배우는 경우가 많을 것이다. 이 상자에는 값을 넣는 횟수에 특별한 제한이 있는 것은 아니기에 한 번 값을 대입한 변수에 다른 값을 대입하는 "파괴적인 대입 조작"이 가능하다.[주16] 변수에 들어 있는 값을 고쳐 쓸 수 있다는 것은 "변수에 어떠한 값이 들어 있는지, 또는 집어넣을 것인지"라는 상태를 취급하는 것이다. 즉, 부작용 항목에서도 설명했듯이 대입은 부작용이라고 생각할 수 있다.

이에 반해 함수형 언어에는 변수로의 대입이 없거나 혹은 매우 한정되어 있어 기본적으로 한번 변수의 값을 정하면 바꿀 수 없는 "속박(binding)"밖에 없다. 대입 기능을 갖고 있는 함수형 언어에서도 대입과 속박은 명확하게 구별하여 처리된다. 변수에 들어 있는 값은 그 변수가 사용되는 범위에서는 변화하지 않는다. 즉, 대입과 같이 상태를 취급할 필요가 없다.

함수형 언어에 익숙하지 않은 명령형 언어 프로그래머들에게 대입이 없다는 것은 사용하기 어려운 제약처럼 보일 수 있다. 왜냐하면 대입을 한 번밖에 할 수 없다는 것은 모든 변수에 C언어로 말하는 const 등의 상수화를 거쳐 프로그램을 작성하라는 것과 같은 것이기 때문이다.[주17]

주16 변수에 값을 넣는 조작이 대입이라고 할 때, 이때까지 들어 있던 값을 엎어쓰는 것은 적극적 대입 또는 재대입이라고 부르기도 한다. 하지만, 본질적으로 역할이 다르지는 않기에 특별히 구별하지 않는 경우도 있다.

주17 그러한 프로그래머야말로 함수형 언어에 익숙해진 후에 명령형 언어로 되돌아와 보기를 바란다. 모든 변수에 대해 const 등의 상수화 키워드를 붙이지 않으면 못 배길 것이다.

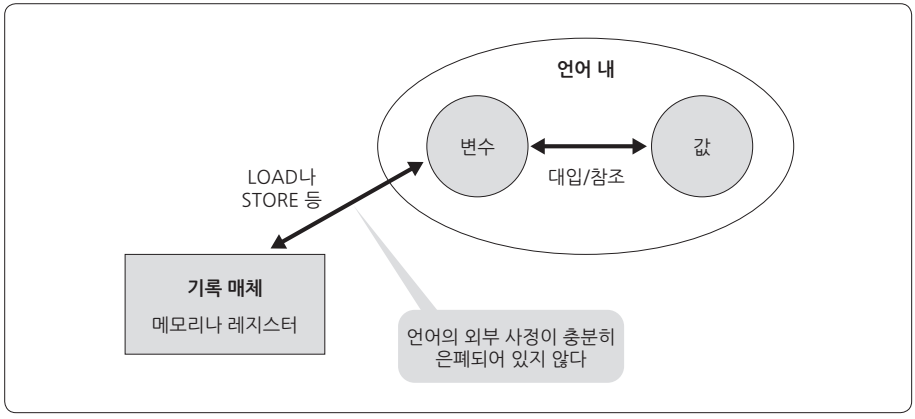

그림 0.2 메모리 모델과 대입

● 대입이 없기에 얻을 수 있는 것

대입이 없기에 속박밖에 없다는 말은 다음과 같을 것이다.

- 값이 들어 있지 않은 경우[주18]가 없다(=값이 항상 들어 있다).
- 값이 변해 버리는 일이 없다(=값이 변하지 않는다).

값이 들어 있으므로 NULL 체크와 같은 본질에서 벗어난 처리를 작성할 필요가 없으며, 값이 변하지 않으니까 정말로 주목하고 싶은 처리 전후에서 무엇을 해도 신경 쓸 필요가 적게 된다.[주19]

속박이라는 제약상, 값이 "들어 있는" 것과 "변하지 않는" 것은 거의 같은 것이다. 만약 어떤 시점에서 변수가 "들어 있지 않은"에 상당하는 특정 상태였다면 그 상태가 "변하지 않기" 때문에 마지막까지 그 변수에 값이 들어가지 않게 된다. 즉, 그 변수는 처음부터 끝까지 그 범위 내에서 쓸모가 없다. 위에서 언급한 포인트에서 "들어가 있다"와 "변하지 않는다"를 나누어 예를 든 것은, 대입과 미초기화가 있는 언어에서 이

주18 Ruby의 nil, Java의 null, 또는 디폴트 값이 없는 미초기화의 변수 값

주19 변수명이 shadowing[보다 좁은 스코프(scope)를 갖는 동일 변수가 있는 것]되어 있지는 않은지 주의할 필요가 있지만, 대개는 처리계가 경고를 해 준다.

두 가지 성질에는 명확한 차이가 있기 때문에 그것들의 속박만을 개별적으로 비교하기 위해서다.

특히 대형 프로그램을 여러 사람이 개발할 경우 자신의 코드를 작성하는 것보다 다른 사람의 코드를 읽을 기회가 더 많아진다. 사람의 주의력은 관심 있는 것 외에는 쉽게 놓칠 수 있으므로 언어의 기능으로서 제약을 보장한다는 점은 코드를 리딩하는 단계를 매우 편리하게 한다.

또한 처리계에 있어서도 속박한 변수의 값이 변하지 않는 성질은 매우 유용하며 대입을 허용한 경우보다도 고도의 최적화를 기대할 수 있다. 아주 간단한 예이지만, 대입이 있는 언어에서 아래와 같이 코드에서는 기술되어 있어도 중간에 생략된 부분의 코드에 따라 x와 y는 동일하지 않을 수도 있다.

```
x = a + b;
<중간 생략>
y = a + b;
```

즉, 실제로 생략된 부분에서 a나 b에 파괴적인 대입을 하지 않은 채 프로그래머가 x와 y가 같다는 것을 알고 있다 해도, x와 y가 동일한 것을 이용한 최적화가 필요한 경우에 처리계에서는 중간에 생략된 부분의 분석을 해야 한다. 하지만 속박밖에 없는 언어라면 a도 b도 한번 속박되면 변하지 않는 것이므로, 프로그래머나 처리계에 있어서 x와 y가 동일하다는 것은 쉽게 알 수 있다.

그림 0.3은 파괴적인 대입의 유무가 코드 리딩(code reading) 시의 시선 이동에 미치는 차이를 간단한 예제로 표현하고 있다. a == b의 a값을 쫓아가 봤을 때, 파괴적인 대입이 있는 경우, a값이 최종적으로 어떻게 되었는지를 끝까지 쫓아가 봐야만 알 수 있다. a 값이 먼저 대입되고 나서 a == b에서 실제로 사용할 때까지 몇 번이나 그 내용이 변경되었는지 모른다. 그에 반해 파괴적인 대입이 없는 경우는 a가 먼저 속박된 위치를 찾아보면 그것이 그대로 a의 정의이며, 이후에는 변하지 않는 것이 보증된다.

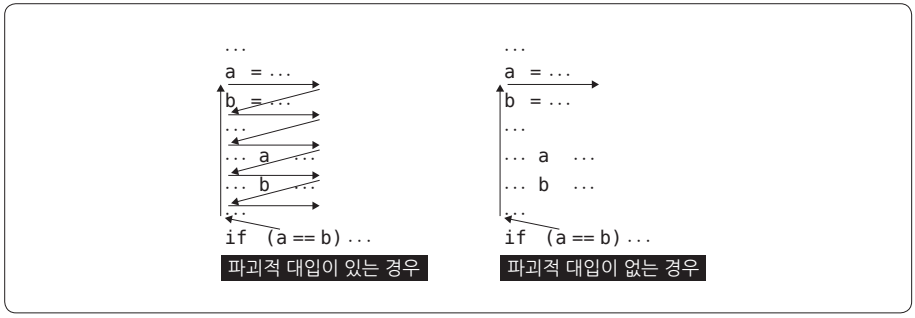

그림 0.3 코드 리딩 시의 시선 이동

Column 다양한 함수형 언어

다양한 함수형 언어를 소개하면 다음과 같다(알파벳순).

Agda URL http://wiki.portal.chalmers.se/agda/pmwiki.php

스웨덴의 찰머스공과대학(Chalmers University of Technology)에서 개발된 함수형 언어. 타입 중에서도 강력한 **의존 타입**이라는 구조를 갖고 있어 수학적 정리 및 증명을 할 수 있다. 뒤에서 설명하는 Haskell과 유사한 문법을 가지고 있다.

Clean URL http://wiki.clean.cs.ru.nl/Clean

네이메헌 라드바우드 대학(Radboud Universty Nijmegen)에서 개발된 순수 함수형 언어.

Miranda라는 함수형 언어를 바탕으로 한 문법을 가지고 있다. **고유형**이라는 기능을 통해 부작용을 취급한다. 개발 환경에는 상용 버전 및 비(非)상용 버전이 있다.

Clojure URL http://clojure.org/

리치 히키(Rich Hickey)에 의해 만들어진 JVM(Java VM)에서 작동하는 함수형 언어. LISP 계열 중 하나다. LISP 자체 함수형 언어로 분류되는 경우가 있지만, 최근에는 함수형 언어를 분류하는 기준의 제약이 다소 느슨해졌다. 이에 대해 Clojure는 보다 더 함수형 언어의 방향으로 기능이 개선되고 있다.

Coq URL http://coq.inria.fr/

프랑스 국립 정보학 자동 제어 연구소(INRIA, Instiut National de Recherche en Infromatique Enautomatique)가 개발한 정리 증명 지원계의 **순수 함수형 언어**. OCaml의 영향을 받아, OCaml로 구현되어 있다. Agda와 같은 **의존 타입**의 구조를 갖고 있어, 함수를 작성하고 함수가 만족해야 할 성질을 기술한 후 그 성질에 대해 정리 증명을 할 수 있다. tactic이라는 증명을 위한 강력한 기능을 제공하고 있어, tactic 명령을 나열하여 증명을 진행할 수 있다. 또한 입증된 함수를 다른 언어로 출력할 수도 있다.

Erlang URL http://www.erlang.org/

에릭손(Ericsson)사가 만든 함수형 언어. 병행/분산 처리를 강하게 의식한 언어로 설계되어 있으며, **경량 프로세스**라고도 불리는 자체 스레드 간의 메시징[주a]에 의해 주로 처리한다. 경량 프로세스 자체는 동일하게 위치된 노드와 원격 노드를 의식하지 않고 **투명**[주b]하게 처리할 수 있다. **핫 스와프(hot swap)**,[주c] **내장애성**[주d] 등에도 배려가 잘 되어 있다.

F# URL http://msdn.microsoft.com/ko-kr/vstudio/hh388569

마이크로소프트(Microsoft)가 OCaml을 바탕으로 개발하고 있는 .NET Framework에서의 함수형 언어. Visual Studio 2010부터 추가되어 있다. 컴퓨테이션식(compuation expression)이라는 것으로 부작용을 제어할 수 있다.

Haskell URL http://www.haskell.org/

함수형 언어의 개방형 표준으로 만들어진 순수 함수형 언어. Clean처럼 Miranda를 바탕으로 한 문법을 가지고 있다. **모나드**(뒤에 설명)라는 구조를 가지고 부작용을 취급한다.

Idris URL http://www.idris-lang.org/

세인트앤드루스 대학(University of St. Andrews)의 에드윈 브래디(Edwin Brady)에 의해 개발된 의존 타입을 지닌 함수형 언어. Haskell과 비슷한 문법과 언어 기능과 Coq가 갖고 있는 tactic을 가지고 있다.

OCaml URL http://caml.inria.fr/ocaml/

INRIA가 개발한 ML 계열의 언어. C언어로 작성한 경우와 손색이 없는 정도의 동작 속도를 기대할 수 있다. camlp4라는 전처리 프로세서도 강력하여 구문 확장이 가능하다.[주e]

OCaml은 원래 Caml이라는 ML의 방언에 대해 객체지향적인 기능을 도입한 것이다. 그러나 신뢰성을 중시하는 OCaml 프로그래머일수록 그에 상응하는 이유가 없는 한 객체지향적인 부분은 사용하지 않는 것 같다. 객체지향 패러다임을 안전하게 취급하는 것은 본질적으로 어렵다. 즉, 버그 없이 처리한다는 것은 사람들에게 매우 어려운 일이다.

Scala URL http://www.scala-lang.org/

마틴 오더스키(Martin Odersky)에 의해 개발된 JVM에서 동작하는 함수형 언어. 함수형과 객체지향의 통합을 목적 중 하나로 내걸고 있다. Java의 라이브러리를 사용할 수도 있으므

주a 임의의 값을 메시지로 하여 별도의 경량 프로세스 간의 송수신을 할 수 있다.

주b 여기에서는 경량 프로세스가 동작하고 있는 노드가 차이가 있든 없든 간에 동일하게 처리한다는 의미다.

주c 가동 중의 모듈을 정지하지 않고 바꿀 수 있다

주d 오류에 따라 프로그램 전체의 동작이 정지하지 않도록 할 수 있고, 오류 발생 시의 로깅(logging)이나 프로세스의 재기동을 하기 쉽다는 점을 의미한다.

주e 여기까지 된다면 너무나 강력하다는 견해가 있다.

로 Java 프로그램과 연계시키기 쉬워 기존의 Java 프로그램을 부분적으로 Scala로 치환해 나간다는 소리를 듣고 있다.

다른 함수형 언어의 초보자를 위한 스터디 그룹[주f]에 나가 본 결과, "Haskell은 처음이지만 Scala를 사용한 적이 있다"라는 사람들이 꽤 있었다.

Standard ML(SML)

Standard ML(SML)[주g]은 나름대로 표준화한 ML. SML 계열의 언어로서는 너무 많은 처리계 구현이 존재하고 있어 각각에 SML의 사양과 호환성을 유지하면서 확장을 실시하고 있다.

주f "고독의 Haskell"이라는 스터디 그룹이 있다.

주g "The Definition of Standard ML"(Robin Milner/Robert Harper/David Mac Queen/Mads Tofte, MIT Press, 1997)

0.5 함수형 언어의 특징적인 기능
타입의 유무, 정적/동적, 강약

18페이지의 칼럼에서는 다양한 함수형 언어를 소개하였다. 여기에서는 "함수형 언어의 특징적인 기능"이라는 제목으로 "typed(타입이 있는)와 untyped(타입이 없는)", "정적 타입과 동적 타입", "순수", "타입 검사", "강한 타입과 약한 타입", "타입 추론", "의존 타입" 그리고 "평가 전략"을 다루겠다.

typed와 untyped

함수형 언어를 분류하는데 타입에 관한 기능이 차지하는 비중은 매우 크다. 먼저 typed(**타입이 있는**)와 untyped(**타입이 없는**)로 크게 나눌 수 있다.

typed는 기존의 명령형 언어에서도 자주 봐 왔던 것이다. 타입이 맞지 않는다는

것을 특정 타이밍에 감지하고 이에 대해 무언가 오류를 표시한다.

또한, typed는 정적 타입과 동적 타입으로 나뉘는데 이것들에 대해서는 다음 항에서 설명하겠다.

반대로 untyped는 도대체 무슨 뜻인지 좀처럼 감이 오지 않을 것이다.

untyped는 모든 값에 대해 전혀 타입을 구별하지 않음에도 계산이 가능하다는 것이다.

예를 들어, 덧셈과 덧셈을 덧셈하는 등 그다지 의미를 알 수 없는 계산을 생각해 보자. typed는 일반적으로 숫자와 계산은 구분되어 있기 때문에 오류가 발생한다. untyped는 수와 계산의 구분도 없기 때문에 계산 결과가 만들어진다. 단, 그 결과로 완성된 것이 도대체 무엇인지에 대해서는 특별히 신경쓰지 않는다. 이것을 잘 이해할 수 있도록 하는 것은 우리 인간의 일이다.

현재 실용적 단계에 있는 언어는 typed 방식이 거의 대부분이다. typed라는 것은 어떠한 **타입 시스템**(type system)이 존재하고 있다는 것이다. 타입 시스템은 프로그램에 타입을 부여함으로써 그 행동을 보장하기 위한 프레임워크(framework)이다. 앞으로 타입에 관해서 무언가 언급되는 경우, 그것이 타입 시스템의 프레임워크의 일부라고 이해해서 읽어 주길 바란다.

정적 타입과 동적 타입

typed의 함수형 언어는 크게 정적 타입 언어와 동적 타입 언어가 있다. 타입 검사를 컴파일 시에 하는 것이 **정적 타입**(static typing)이고, 정적 타입을 갖는 것이 정적 타입 언어(statically typed language)다. 그리고 타입 검사를 실행시간에 실시하는 것이 **동적 타입**(dynamic typing)으로, 동적 타입을 갖는 언어를 동적 타입 언어(dynamically typed language)라 한다.[주20]

주20 명령형 언어에서도 정적 타입 언어와 동적 타입 언어의 구별은 있다. C언어나 Java 등의 경우가 정적 타입 언어이고, Perl이나 Python 등은 동적 타입 언어다.

순수

순수(pure)란, 동일한 식은 언제 평가해도 같은 결과가 되는 **참조 투명성**(referential transparency)이라는 성질을 갖고 있다는 것이다.

순수하다면 어떤 변수를 참조했을 때의 결과가 항상 동일하다. 또 순수하다면 함수를 값에 적용했을 때의 결과가 항상 동일하다. 순수한 함수형 언어는 특별히 순수 함수형 언어(purely functional language)라고 한다.

부작용이 없다면 참조 투명성이 된다. 부작용이 있는 식을 작성할 수 있는 언어는 순수하지 않다.[주21]

참조 투명성을 지키지 않는 함수로서 다음과 같은 C언어의 함수 foo를 생각해 보자.

```
int foo() {
  static int n = 0;
  return ++n;
}
```

foo()라는 식은 foo 자체가 호출된 수를 반환한다. 당연히 매번 결과가 다르다. n이 상태이며 그것의 참조와 값의 변환이 있을 수 있다. 즉, 이것은 부작용이 있다.

참조 투명성을 지키지 않는 변수를 생각해 보자. 생각해 보자고 말은 했지만, C언어의 변수 값은 대입으로 쉽게 다시 엎어 써 버리므로 변수 단위로 보면 대체로 참조 투명성이 없다. 그러므로 여기에서는 대입을 하지 않고도 값이 바뀌어 버리는 식을 생성하도록 극단적인 상황도 제공해 보았다. 다음과 같은 C언어의 변수 foo를 생각해 보자.

```
volatile int *foo = 0xDEADBEEF;
```

주21 즉, 거의 모든 언어는 순수하지 않다.

이 코드는 임베디드 프로그래밍에서 0xDEADBEEF번지에 매핑되어 있는 하드웨어 신호를 읽어들이기 위한 부분이다. *foo라는 식으로 그 시점에서의 신호 상태를 취득할 수 있는데, 당연히 *foo라는 식은 사용될 때마다 결과가 다를 수 있다. 이는 값을 참조할 때마다 외부와의 입출력을 발생시키기[주22] 때문이다.

참조 투명성을 갖는 변수는 간단하다. 다음과 같은 C언어의 변수는 참조 투명성을 갖는다.

```
const int foo = 0;
```

이 변수 foo를 참조하는 foo라는 식[주23]이 언제나 0이다.

참조 투명성을 갖는 함수는, 즉 수학적인 의미의 함수다. 다음과 같은 C언어의 함수 foo는 참조 투명성이 있다.

```
int foo() {
  return 1;
}
```

이 함수 foo를 호출하는 foo()라는 식은 언제나 1이 된다.

타입 검사

타입 검사(type checking)는 프로그램(함수)으로서 "타입"[주24]에 일관성이 있는지를 컴파일 시에 검사해 주는 기능이다. 검사에 실패하는 식이 있는 경우 일반적으로 컴파일 오류로 처리된다.

주22 그리고 그렇게 되는 것에 의미가 있다.

주23 C언어의 경우, "꽤나 지저분(dirty)한 짓을 하지 않는" 한 변경은 없다.

주24 타입에 대해서는 뒷장에서 자세히 설명한다.

강한 타입과 약한 타입

언어 사양(language specification)에서 정의되지 않은 동작을 발생시키지 않는 것을 안전성(safety)이라고 한다. 여기서 말하는 "언어 사양에서 정의되지 않은 동작"의 예를 들면, C언어 등에서 "전혀 관계없는 메모리 영역에서 잘못된 데이터를 읽어 버리는 경우"를 말할 수 있다. 해당 프로그램은 분명 바람직하지 않은 상태로 돌입하게 될 것이다.

이 안전성에 대해 타입 검사를 성공했다면 안전성이 보장되는 타입을 **강한 타입**(strong typing), 반대로 타입 검사에 성공하더라도 안전성이 보장되지 않는 타입을 **약한 타입**(weak typing)이라고 한다. 정적 타입의 명령형 언어는 약한 타입인 경우가 많다.[주25] 타입 언어라면 강한 타입 쪽이 "타입"의 고마움을 느낄 수 있을 것이다.

타입의 추론

타입 추론(type inference)은 명시적으로 부여되는[주26] 타입의 정보로부터 암시적인[주27] 부분의 타입을 추론해 주는 기능이다.

예를 들어, "어떤 비교 가능한 타입의 값의 열을 가지고 그 열에 있는 값 중에서 최댓값을 반환하는" 함수 maximum과 "정수 값을 가지고 문자열 표현(문자열)을 반환하는" 함수 show가 있다고 하자. 함수 maximum이 취하는 것은 비교만 된다면 무슨 열이라도 상관없지만, 함수를 합성하여 maximum · show를 취하면 maximum은 show의 결과인 문자의 열을 취하므로 함수 maximum · show는 "정수 값을 가진 문자를 반환하는" 함수가 된다. "어떤 비교 가능한 타입"이 "문자"로 타입 추론되었기 때문이다.

주25 동적 타입 언어에서 타입이 맞지 않은 경우, 강한 타입은 타입 오류를 발생시키지만, 약한 타입은 타입의 변환을 시도하여 어떻게든 동작하려고 한다.

주26 프로그래머가 부여했다는 의미.

주27 프로그래머가 타입의 기술을 생략했다는 의미.

Column **약한 타입은 무엇을 위한 것인가?**

이것에 관해서는 필자도 명확한 이유를 파악한 것은 아니어서 확신은 없지만, 약한 정적 타입의 경우, '있다'보다는 '아직 남아 있다'라고 생각한다.

예를 들어, 당초의 C언어에서는 타입이 어셈블러 레벨로 변화될 때나 메모리에 값을 저장할 때에 어떤 명령을 지정하는 것이 적절한지, 얼마나 많은 공간을 점유하는 값인지를 나타내기 위한 주석 정도의 의미밖에 갖질 않았다. 타입에 관한 이론이 발전해 감에 따라 타입의 역할이 다시 검토되고, 기왕 타입을 붙이는 경우라면 강한 타입이 나오게 된 것이다.

컴파일 시에 체크하지 않는 동적 타이핑의 경우, "비록 의도하지 않은 행동을 하고 있다고 해도 가능한 한 오류로 멈추지 않고 동작하기 위해서"라는 목적으로(일단, 그 좋고 나쁨은 제외하자) 약한 타입을 필요로 하게 되었다.

단, 일반적으로 컴파일을 통과해야 하는 정적 타입의 경우, 뭔가 강한 의도가 있어 "약한 타입"이 존재하는 것은 아니라고 생각한다.

의존 타입

의존 타입(dependent type)은 다른 타입에 의존한 타입이나 값에 의존한 타입을 만들 수 있는 기능이다. 예를 들어, 타입A와 타입B가 동일할 때에만 그 타입에 값이 존재하는 타입이나, 길이를 타입 수준에서 갖는 리스트 타입과 같은 보다 강한 제약을 갖는 타입을 만들 수 있다.

평가 전략

평가 전략이란, 프로그래밍 언어에서 "어떠한 순서로 식을 평가할까?"라는 규칙을 말한다. **적극 평가**(eager evaluation) 또는 **지연 평가**(lazy evaluation) 등이 있다. 적극 평가에서는 인수가 전달되기 전에 평가된다. 지연 평가에서는 필요할 때까지 평가하지 않는다. 함수형 언어에 한정되지 않고 현재 대부분의 언어는 적극 평가를 사용한다. 일부 순수 함수형 언어가 지연 평가를 사용하고 있을 뿐이다.

다음의 함수 tarai는 재귀적으로 호출되는 "다라이 함수(다케우치 함수)"라고 불리는 함수다.

```
int tarai(int x, int y, int z) {
    return (x <= y)
        ? y
        : tarai(tarai(x - 1, y, z),
                tarai(y - 1, z, x),
                tarai(z - 1, x, y));
}
```

이 함수에서는 x가 y보다 작으면 z값은 볼 필요도 없이 결과가 결정난다. 따라서 적극 평가의 언어로 순수히 구현하면 z의 계산 때문에 느려지고, 지연 평가의 언어로 구현하면 z의 계산이 필요 없다고 판단해 생략되므로 빨라진다는 특징적인 함수로 되어 있다.

물론 어떠한 경우에도 지연 평가가 더 빨라진다는 것은 아니다. 평가 대기의 상태[주28]를 대량으로 생산할 수 있으므로 적극 평가가 빠른 경우가 압도적으로 많을 것이다. 그럼에도 지연 평가가 존재하는 이유는 지연 평가의 경우가 언어로서는 수학적으로 자연스럽게 기술하기 쉬운 편이고, 실제로 끝까지 평가해 보면 오류가 되지만, 생략되는 경우라면 오류에 관계없이 계산이 진행된다는 점 등의 유리한 점도 있기 때문이다.

주요 함수형 언어와 명령형 언어의 기능 리스트.

주요 함수형 언어와 명령형 언어에 대해 구비된 기능을 표 0.1에 정리하여 두었다.[주29]

주28 최종적으로 평가되지 않는다고 해도.

주29 표 안에 ○가 많다고 해서 좋다든가 추천한다는 뜻이 아니다. 툴은 적재적소에 사용되어야 한다.

표 0.1 주요 함수형 언어와 명령형 언어의 기능

언어	type	순수	타입 추론	의존 타입
Agda	강함, 정적	○	○	○
Clean	강함, 정적	○	○	X
Coq	강함, 정적	○	○	○
F#	강함, 정적	X	△	X
Haskell	강함, 정적	○	○	△
Idris	강함, 정적	○	○	○
OCaml	강함, 정적	X	○	△
Scala	강함, 정적	X	△	X
SML	강함, 정적	X	○	X
Clojure	강함, 동적	X	–	–
Erlang	강함, 동적	X	–	–
C	약함, 정적	–	X	–
C++(C++11)	약함, 정적	–	△	–
Java	강함, 정적	–	△	–
JavaScript	약함, 동적	–	–	–
Perl	약함, 동적	–	–	–
Python	강함, 동적	–	–	–
Ruby	강함, 동적	–	–	–

X: 지원 안 함, O: 지원 , 세모: 부분 지원

0.6 왜 지금 함수형 언어인가?
추상화, 최적화, 병행/병렬화

지금 함수형 언어를 채용하는 이유에는 여러 가지가 있다. 여기서는 "추상화", "최적화", "병행/병렬화" 등 현대적인 프로그래밍에 있어서 개발 효율과 실행 효율에 영향을 주는 중요한 관점에서 함수형 언어를 사용하는 이유를 살펴보자.

함수형 언어의 추상화 — 수학적인 추상화란?

프로그래밍에서 **추상화**(abstraction)는 중요한 기술이다. 모듈의 재사용성을 강화하고, 구현과 인터페이스를 분리해 두는 적절한 추상화에 의해 프로그램은 유연하게 유지된다. 프로그래머가 갖고 있는 추상화 능력에서 차지하는 비율도 크지만, 언어 기능에 의해 가능한 추상화 또한 무시하지 못하는 것도 사실이다.

함수형 언어에서 추상화할 때 함수형 프로그래머 모두가 명확하게 좋다고 말하는 방향성이 있다. 그것은 그룹 이론과 범주론[주30]이라는 수학적 방향으로의 추상화다.

여기서 좋은 추상화는 무엇인지에 대해 생각해 보자. 필자가 생각하는 추상화의 장점에 대한 평가 기준은 다음과 같다.

- 많은 문제에 대해 범용적으로 적용할 수 있다.
- 추상화 후의 세계에서 실시할 수 있는 조작이 풍부하다.

범용적으로 적용할 수 없는 것은 원래부터 추상화한다고 말할 수 없다. 따라서 전자에 대해서는 당연한 기준일 것이다. 후자에 대해서는 어떨까? 추상화한 후 누군가 위에서 많은 강력한 조작을 할 수 있는가 하는 점은 의외로 고려되지 않았을 것이라 생각한다.

주30 그룹 이론과 범주론은 수학적인 구조를 다루기 위한 분야다. 함수형 언어에서는 수학적인 의미에서 함수를 취급하므로 수학적인 구조를 취급하는 방법이 그대로 적용되기 쉬운 경향이 있다.

수학은 실로 많은 문제를 처리할 수 있으며 문제를 한번 수학의 세계로 추상화할 수 있다면 컴퓨터의 역사보다 긴 세월에 걸쳐 축적된 풍부하고 강력한 수학의 세계의 성과를 모두 그대로 적용할 수 있는 상태가 된다. 이것은 앞서 언급한 좋은 추상화의 평가 기준 두 가지를 충분히 충족하고 있다.

혹시, 수학 실력 때문에 걱정인가? 수학 실력이 별로 좋지 않은 사람이라도 너무 걱정할 필요 없다. 모든 함수형 프로그래머가 수학적인 지식이 없으면 함수형 언어를 사용할 수 없다는 뜻이 아니다.[주31]

수학적으로 명료한 추상화된 프로그램을 작성하고 싶거나 라이브러리를 제공하려는 경우를 제외하고는 선배들이 준비한 명확히 추상화된 라이브러리를 사용할 수 있다. 또한 자신이 수학적 의미를 잘 몰라도 특정 인터페이스와 타입의 제약에 따라 작성하는 것만으로도 마음대로 수학적으로 명료한 추상화를 할 수 있는 경우도 있다.

함수형 언어의 최적화

프로그래밍 언어 및 프로그래머에게 있어서 **최적화**(optimization) 또한 매우 중요한 요소다. 최적화가 강력할수록 프로그래머는 효율을 신경 쓰지 않고 간결성에만 유의해서 의미 있는 코드를 작성하면 된다. 최적화가 강력할수록 코드의 표현 이상으로 충분히 효율 좋은 동작을 해 줄 것이기 때문이다.[주32]

특히 예전처럼 컴퓨터의 성능이 낮은 시대와는 달리 컴퓨터의 성능이 크게 향상된 요즈음에서는 컴파일러 등의 처리계에 최적화를 맡기는 것이 좋을 만큼, 프로그래머가 편해질 수 있는 가능성이 높아졌다.

앞서 언급한 바와 같이 함수형 언어에서는 수학적 추상화를 한다. 수학적으로 명확한 세상을 경유함으로써 최적화도 수학의 성과를 이용하는 것이 가능하다.

주31 필자의 경우도 그다지 수학적 지식이 많지 않다.

주32 예를 들어, 사람의 힘에 의한 최적화의 경우, 캐시에 올라갈 수 있는 데이터 양을 생각한 후에 그 단위로 데이터를 구분하여 처리하도록 코드를 작성해야 하지만, 그것은 캐시 크기가 바뀌면 함께 변경해야 한다. 이것은 실제 대상으로 하고 있는 문제에 대한 알고리즘의 본질과는 또 다른 이야기다. 사람의 힘으로 무리하게 최적화함으로써 "대상의 문제를 해결하는 것"과 "처리를 빨리하는 것"의 두 가지 과제를 동시에 해결하려는 코드를 기술하는 것이므로 전체적으로 이해하기 어려워 유지 보수 또한 어렵다.

그럼, 1에서 n까지의 자연수를 모두 더하는 함수 totalFrom1To를 생각해 보자. 예를 들어, C언어에서는 다음과 같이 될 것이다.

```
// C(C99)
int totalFrom1To(int n) {
    int result = 0;
    for (int i = 1; i <= n; result += i++);
    return result;
}
```

result에 더해지는 수 i를 1부터 증가시키면서 더해 나가, i가 n이 될 때까지 반복시키면 된다.

이에 반해 Haskell로 다음과 같이 작성했다고 치자.

```
-- Haskell
totalFrom1To = sum . enumFromTo 1
```

이 코드에서는 두 개의 함수, 리스트의 모든 내용을 더하여 전체 합의 값으로 하는 함수 sum과 1부터 n까지의 리스트를 만드는 함수 enumFromTo 1을 함수 합성하고 있다. Haskell 코드의 예와 같이, 일단 중간 리스트를 만들고 나서 생성한 리스트를 하나 하나 읽어서 처리하는 부분을 작성하면 "리스트를 만드는 만큼의 처리가 아까운 것은 아닌지?"라고 생각할지도 모르겠다. 하지만 이러한 처리는 최적화를 통해, C언어의 코드와 동일하게 중간 리스트를 만들지 않는 처리로 하는 것이 가능하다. 이것은 일종의 구조(이번 예에서는 리스트)를 만드는 함수와 특정 구조(이번 예에서는 리스트)를 넣어서 처리하는 함수가 합성할 때 중간 구조를 만들지 않는 함수로 변환 가능하다는 수학의 세계의 성과가 반영되어 있기 때문이다.

만약 이러한 최적화가 없는 경우, C언어의 코드와 같은 처리를 기대한다면 다음과 같이 기술하게 될 것이다.

```
-- Haskell
totalFrom1To = auxTotalFrom1To 0 where
```

```
auxTotalFrom1To result 0 = result
auxTotalFrom1To result n = auxTotalFrom1To (result+n) (n-1)
```

여기에서는 이 코드에 대한 설명은 하지 않는다. 중요한 것은 이번 예에서 일단 리스트를 통한 코드의 예가 프로그래머에게 있어 파악하기 좋다는 것이다. "1부터 n까지의 리스트를 만들고" 나서 "만든 리스트의 내용을 전부 더한다"라는 코드는, "result에 더해지는 수 i를 1부터 증가시키면서 n까지 더하는" 코드보다도 부품화되어 있다. 부품마다 제대로 작동하는지 확인하는 것도 간단할 것이다.

프로그래머에게 있어 파악하기 좋다는 것은 특별히 프로그램을 짧게 작성할 수 있거나 하는 것이 아니다.[주33] "프로그램의 정확성(올바른 결과를 반환하는 것)"이 사람의 눈에도 명확하다는 것을 의미한다. 그러나 명확하게 올바른 코드는 종종 효율[주34]이 좋지 않다. 함수형 언어의 최적화 메커니즘은 언어가 갖는 강한 제약과 수학적 추상화를 이용하여 올바른 것이 분명하지만, 비효율적인 프로그램으로부터 동일하면서도 더 효율적인 프로그램으로 변환할 수 있다는 것이다.

그림 0.4는 위의 코드에 대한 최적화가 되지 않은 경우[주35]에 계산해 나가는 과정을 보여 준다. 마찬가지로 그림 0.5는 최적화가 된 경우에 계산해 나가는 과정을 보여 준다. 모두 1부터 5까지의 숫자를 더한 결과를 계산하고 있지만, 최적화가 된 동작에서는 도중에 리스트를 만드는 작업을 하지 않는 만큼 시간 효율도 공간 효율도 개선되고 있다.

물론 다른 언어에도 있는 저수준의 최적화 또한 별도로 적용된다. 여기에서 설명한 최적화는 보다 추상적인 단계에서의 최적화다. 그림 0.6은 언어가 갖는 최적화를 요약하고 있다. 함수형 언어는 추상화의 항목에서도 언급했듯이 수학적 추상화가 좋다고 판단해서 이용하므로 다른 언어에서도 이루어지는 일반적인 최적화 기법 외에 수학적 추상화를 이용한 최적화 단계를 가질 수 있다. 당연하겠지만, 언어가 갖는 성

주33 그러한 면도 있지만.

주34 시간 효율(실행 속도)이나 공간 효율(메모리 사용량)이 대표적인 효율의 지표다.

주35 즉, 코드에서 이해한 그대로의 동작을 할 것이라고 생각했을 경우.

질이 수학적으로 깨끗할수록 수학적인 최적화를 적용하기 쉽다.

단, 최적화할 수 있는 것이 많다고 해서 반드시 효율이 좋은 것이 생성되는 것은 아니라는 점은 기존의 명령형 언어와 동일하다.

특히 현재의 CPU는 명령형 언어[주36]를 빨리 동작시키는 것에 중점을 두고 있다. 물론 본래 CPU가 있으면 이를 위해 기계어라는 명령형 언어가 존재하고 있으므로 명령형 언어 쪽이 CPU에, 좋게 말하면 좀 더 가까운 위치에, 나쁘게 말하면 추상도가 낮은 위치에 있으므로 당연하기는 하다. 따라서 단순히 명령형 언어와 비교하면 실행 속도에 있어서 함수형 언어에 불리한 점이 있다는 것 또한 사실이다. 더욱이 최적화를 포함해도 원하는 성능에 이르지 못했을 경우, 일반적으로 말하는 고급 최적화[주37]를 걸 정도로 튜닝의 난이도가 올라간다.

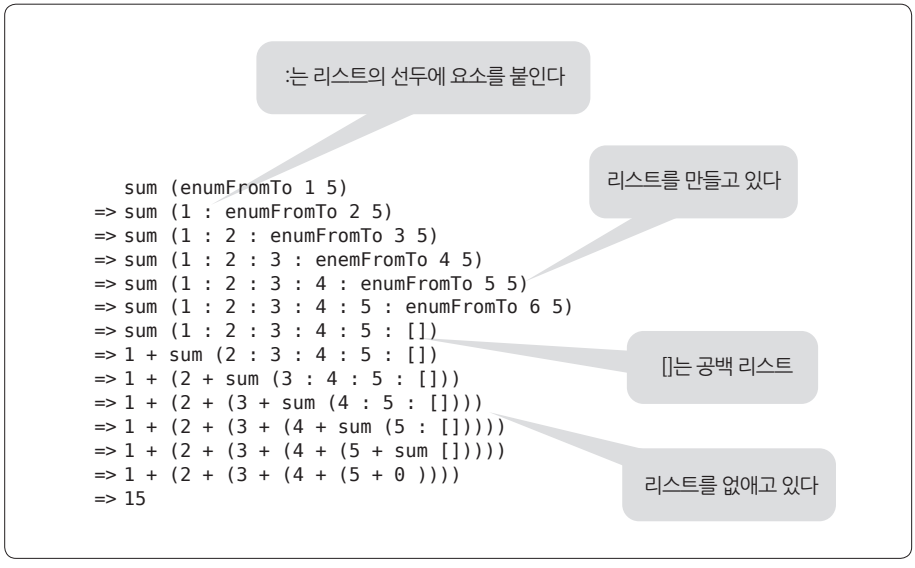

그림 0.4 최적화 없음

주36 꼭 집어 말하자면 C언어.

주37 최적화 메커니즘에 따르지 않고 어찌되었든 간에 사람에게 있어서 분명하지 않은 변환을 시행하는 것들.

```
   sum (enumFromTo 1 5)
=> 1 + sum (enumFromTo 2 5)
=> 1 + (2 + sum (enumFromTo 3 5))
=> 1 + (2 + (3 + sum (enumFromTo 4 5)))
=> 1 + (2 + (3 + (4 + sum (enumFromTo 5 5))))
=> 1 + (2 + (3 + (4 + (5 + sum(enumFromTo 6 5)))))
=> 1 + (2 + (3 + (4 + (5 + 0))))
=> 15
```

리스트를 만들지 않고 계산이 진행 가능

그림 0.5 최적화 있음

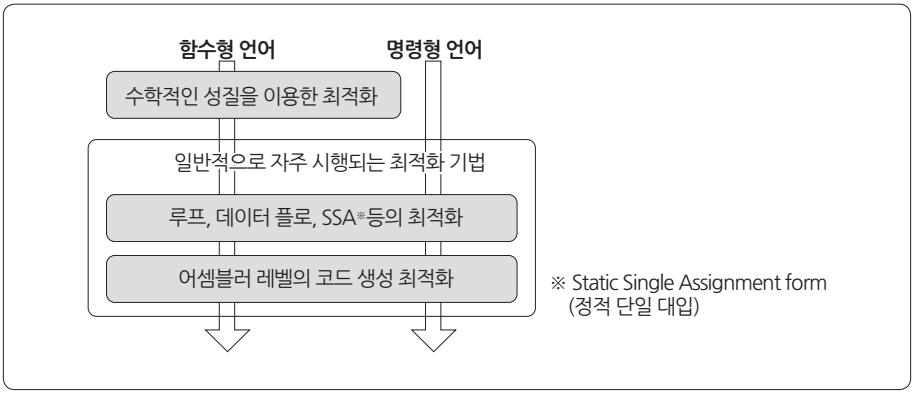

그림 0.6 이용할 수 있는 최적화 메커니즘

그러나 어떤 언어에 있어서도 언어의 수학적 성질이 개선되는 것은 거의 없지만, 다른 한편에서는 연구를 통한 최적화 기법이 진보하고 있다.

다른 언어에서 적용할 수 있는 최적화 기법[주38]이라도 특정 언어에서는 언어 그 자체의 성질이 그다지 좋지 않아서(순수하지 않은) 적용할 수 없는 최적화 기법이라는 것도 있을 것이다. 추상도가 높고 언어의 성질이 비교적 좋다는 점에서 함수형 언어는 명령형 언어에 비해 최적화의 성장 가능성이 높다고 말할 수 있다.

주38 예를 들어, 순수한 언어로 참조 투명성을 이용하여 동일한 인수라면 재계산 없이 앞에서 계산한 결과를 부여하거나 하는 최적화.

함수형 언어와 병행/병렬 프로그래밍

여러분이 가지고 있는 CPU는 몇 코어인가?[주39] 해마다 CPU의 코어 수는 증가하고 있으며, HT(Hyper-Threading)를 포함하여 10개 이상의 병렬 처리를 할 수 있는 환경도 이미 상품화되고 있다. 그런데 프로그램 쪽은 어떨까? 대부분의 경우, 동일 프로그램이라고 해도 보다 주파수가 큰 CPU에서 작동시키면 기대되는 처리의 속도가 빨라질 것이다. 하지만 동일 프로그램이라고 해도 보다 코어가 많은 CPU에서 동작시키면 그만큼 기대가 될 정도로 속도가 빨라질까?

● 병행/병렬이라는 개념과 프로그래밍의 어려움

프로그램이 멀티 코어 CPU를 충분히 활용하기 위해서는 적절하게 병행/병렬 프로그래밍되어 있지 않으면 안 된다. 그러나 병행/병렬 프로그래밍은 일반적으로 쉽지 않다. 예를 들어, 병렬 실행에 있어서 어떤 특정 처리의 열과 다른 처리의 열이 서로 어떻게든 잘 혼합하여 실행되었다 하더라도 전체적으로 기대하는 동작으로 나타나야 하기 때문이다. 서로를 혼합하는 방법의 패턴은 일반적으로 방대하고, 그중 극소수에서 기대하는 동작을 하지 않는다면 재현성[주40]이 매우 낮은 버그로 다시 표면화될 수 있다.

일반적으로 그림 0.7과 같이 물리적으로 여러 작업을 동시에 수행할 수 있는 것을 **병렬**(parallel)이라고 말하고, 그림 0.8과 같이 논리적으로 여러 작업을 동시에 실행할 수 있거나 실행 상태를 여러 개 유지할 수 있는 것을 **병행**(concurrent)이라고 한다. 병렬이라면 병행할 수 있다. 예를 들어, 싱글 코어 환경에서 스레드 전환(시분할)으로 여러 작업을 외관상 동시에 수행하는 것처럼 보이는 것은 병행 실행을 말한다.

주39 참고로, 필자의 메인 환경은 6코어(HT/Hyper-Threading으로 12개의 스레드)다(이 책의 원고 집필 시점 2014년 10월).

주40 동일한 조건에서 동일한 것을 한다면 같은 결과를 얻을 수 있는 성질.

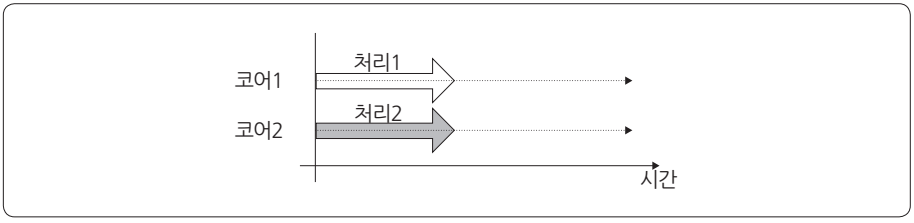

그림 0.7 병렬

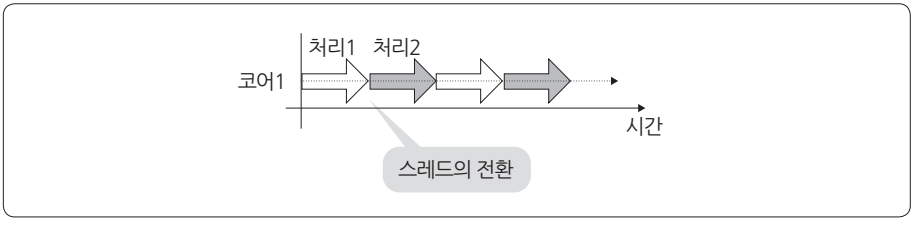

그림 0.8 병행

● 목적으로부터 고려하는 병행/병렬 프로그래밍

실제로 대상으로 하는 문제와 그 해결책으로서 프로그래밍에 주목하면, 병행/병렬 프로그래밍이 목적으로 하는 방향성의 차이를 알 수 있다. 병행 프로그래밍을 하는 이유는 여러 태스크를 동시에 수행할 수 있다는 점 때문이다. 이에 반해 병렬 프로그래밍에서는 명확하게 태스크의 고속화를 목적으로 하고 있다. 병행 프로그래밍에서는 여러 태스크가 같은 모양이어야 한다는 필요와 결과가 결정적[주41]이어야 할 필요가 특별히 없다. 주목하는 문제에 대해 동시에 실행되면 좋겠다는 조건밖에 없기 때문이다.

예를 들어, 웹 애플리케이션은 병행 프로그래밍되어 있다고 말할 수 있다. 웹 애플리케이션의 경우, 여러 태스크는 요청에 대한 응답의 생성이며, 요청은 동시에 처리되지만 응답의 내용은 타이밍에 따라 비결정적[역주]이다. 그림 0.9는 액세스 카운터를 갖는 웹 애플리케이션에 대해 거의 동시에 여러 요청이 발생하는 경우다. 적절히 병행 프로그래밍되어 있으면 이러한 여러 요청은 병행하게 처리된다.

주41 같은 응답을 얻을 것.

역주 비결정적이란 말은 같은 응답을 얻지 못하는 것을 말한다.

그러나 액세스 카운터 부분은 순서대로 처리될 필요가 있으므로, 실제 어떤 요청에 대해 먼저 카운트 업을 할지에 따라 각 요청에 대한 응답 내용은 비결정적으로 바뀐다.

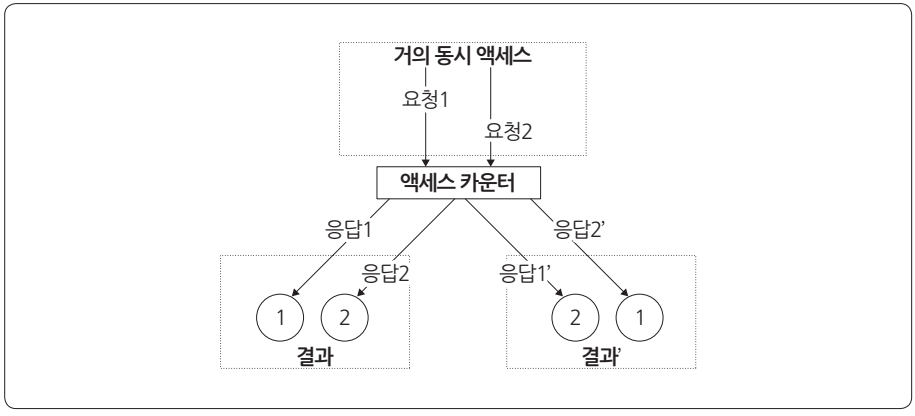

그림 0.9 웹 애플리케이션에 의한 요청의 병행 처리

병렬 프로그래밍에서는 원래 (순차로 행해지는) 태스크의 고속화를 목적으로 하므로 결과가 결정적인 것까지 기대할 경우가 있다. 즉, 싱글 코어에서 실행했을 때와 멀티 코어에서 실행했을 때에 다른 결과가 되기도 하는 것은 실제로 사용하기에는 곤란하다. 예를 들어, 무언가 계산하는 동안에 매우 큰 배열의 내용을 전부 더하는 식의 합계 처리가 있다고 치자. 이 합계 처리를 병렬화하면 그림 0.10과 같이 배열을 코어 수만큼으로 나누어서 각 코어에서 각각 담당한 만큼을 더한 후에 마지막으로 각 코어의 결과를 모아서 더하게 된다. 사실은 이 병렬화에서는 덧셈이 결합 법칙 a+(b+c)=(a+b)+c를 만족하지 못하는 값의 배열에 적용한 경우, 결정적이지 않다. IEEE 754 부동 소수점 숫자 등에서는 바로 덧셈이 결합 법칙을 만족하고 있지 않으므로 분야[주42]에 따라서는 이 병렬 알고리즘에서 문제가 될 수 있다.

주42 작은 오차가 큰 오차를 발생시키고 그것이 치명적인 영향을 끼치는 분야로는 금융이나 항공 우주 과학 분야가 있다.

● 병행 프로그래밍의 어려움 — 결합 상태, 데드록

일부 함수형 언어는 경량 스레드와 경량 프로세스라는 매우 낮은 비용의 스레드 메커니즘[주43]과 이를 제어하기 위한 우수한 스레드 컨트롤러와 I/O 매니저를 가지고 있다.

병행 프로그래밍은 일반적으로 **멀티 스레드 프로그래밍**(multithreaded programming)에 의해 실현한다. 따라서 병행 프로그래밍의 어려움은 멀티 스레드 프로그래밍의 어려움이다.

멀티 스레드 프로그래밍이 어려워 주로 문제가 되는 경우는 여러 스레드가 동일한 리소스에 거의 동시에 액세스할 때 리소스가 예기치 않은 상태가 되어 버리는 **경쟁 상태**(race condition)다. 일반적으로 mutex 잠금[주44] 등을 이용하여 리소스에 접근할 때 크리티컬 섹션(critical section: 임계 영역) 내로 국한하는 등 적절하게 경쟁 상태를 일으키지 않도록 배타적 제어[주45]를 할 필요가 있다.

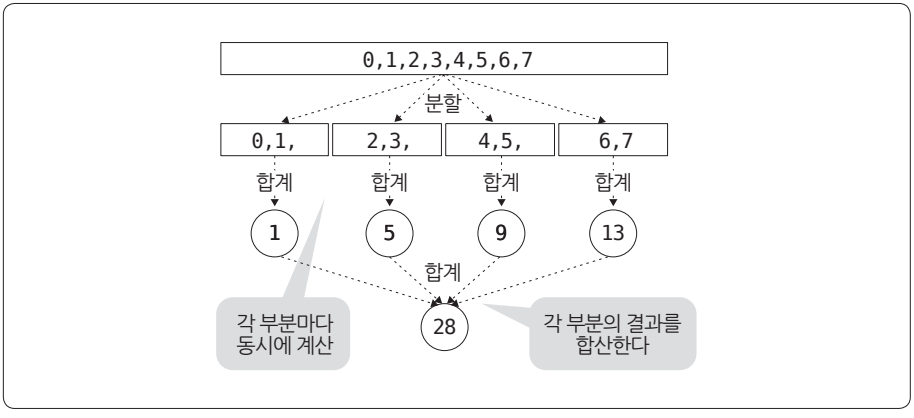

그림 0.10 병렬 합계

주43 사용자 공간에서 동작하며 문맥(context)이 작아서 스위칭 또한 빠르다는 특징이 있다.

주44 동시에 한 개의 처리밖에 크리티컬 섹션(임계 영역)에 들어가지 못하도록 하기 위한 상호 배타 메커니즘. 잠금(lock)을 취득한 문맥의 처리만이 크리티컬 섹션 내에서 실행할 수 있다.

주45 동시에 이용되는 것을 막기 위한 것.

그러나 기존의 잠금에 의한 리소스 제어는 매우 복잡하고, 병행 프로그래밍의 트러블의 근원점이 되고 있다. 예를 들어, 다음과 같은 것이 대표적인 트러블일 것이다.

- 잠금해야 할 리소스인 것을 사양/설계/기존 코드 등으로부터 파악하지 못하고 잠금해야 하는 것을 잊어버렸다.
- 잠금한 리소스를 사용한 후, 잠금을 푸는 것을 잊어버렸다.
- 잠금할 필요가 없거나 아니면 당초 필요가 있었지만 할 필요가 없어진 리소스에 대해 여전히 잠금을 하고 있어 성능이 올라가지 않는다.
- 잠금을 취할 범위가 쓸데없이 커서 성능이 올라가지 않는다.
- 스레드 두 개가 서로 잠금 상태인 리소스를 서로 취하는 바람에 멈춰 버리는 데드록.

그리고 이에 기인하는 대부분의 문제가 재현성이 없는 버그 리포트(bug report)로 올라와 있다.

결과적으로 좀처럼 재현하지 않는 문제를 재현하기 위한 단순한 작업을 오로지 운에 맡기고 반복하는 데에 프로그래머의 귀중한 시간이 소요된다.

특히 위험한 것은 **데드록**(deadlock)이다. 데드록은 프로그램이 다음 조건을 만족하는 것만으로 발생할 수 있다.

- 잠금해야 할 두 개 이상의 리소스가 있다.
- 이미 한 개 이상의 리소스를 잠금한 채로, 별도의 리소스의 잠금을 요구하며 기다리는 일이 있다.
- 이미 잠금되어 있는 리소스를 다시 취할 수 없다.
- 리소스의 잠금을 취하는 순서가 정해지지 않았다.

멀티 스레드 프로그래밍에서는 처음 세 가지 조건은 비교적 어쩔 수 없는 일이라 손댈 수 없어, 마지막 조건을 잡기 위해 리소스의 잠금 순서를 정해 둠으로써 대처하곤 한다.

그러나 이 조치는 분명히 사람의 주의력에 의존한다. 뒤늦게 개발에 뛰어든 프로그래머에게 프로그램 전체에서 몇 개 있는지도 명확하지 않은 리소스의 잠금에 관한 순서를 코드에서 파악하게 하는 것은 가혹하다고 말할 수 있다. 그렇다고 해서 도큐먼트로 관리한다면 드물게 우스갯소리[주46]로 나오는 "스프레드 시트로 관리되는 변

수 관리 테이블"과 같은 존재가 되어 버리는 것은 아닐까 싶다. 게다가 어떤 리소스를 잠그고 있는 절차 중에 다른 리소스를 잠글 필요가 있는 절차를 부르려고 한다면, 결과적으로 리소스 두 개의 잠금 순서가 정해진 잠금 순서에 어긋난 경우 이 호출은 하면 안 된다. 호출하려는 절차 중에 있다고 해도 무엇을 어떤 순서로 잠갔는지 모두 쫓아가 검증할 수 없다면, 일반적으로 부담 없이 하는 절차 호출조차 위험한 것이다. 사실상 잠금을 포함한 절차끼리는 결합할 수 없다고 말해도 좋다.

일부 함수형 언어는 STM(Software Transactional Memory)이라는 DB의 트랜잭션(transaction)[주47] 및 그 리트라이(retry) 제어와 비슷한 메커니즘을 갖추고 있어 비교적 높은 실행 효율로 리소스의 배타적 처리를 쉽게 할 수 있다. 리소스의 배타 처리 중에는 실시해야 할 처리가 제한되므로 안전하다.

● 병렬 프로그래밍의 기여 — 참조 투명성의 보증

함수형 언어는 명령형 언어와의 비교에서도 언급했듯이 파괴적인 대입 조작을 할 수 없거나 매우 제한적인 조건에서만 허용된다. 특히 순수 함수형 언어에서는 함수가 언제 어떻게 평가해도 같은 결과가 된다. 즉, 그 처리만 병렬화하여도 같은 결과가 된다는 것을 알고 있다.

중요한 것은 "알고 있다"라는 것이다. 예를 들어, 굳이 함수형 언어가 아니라도 참조 투명성을 만족하는 처리만 있으면 아무런 생각 없이 병렬화하여도 전혀 문제가 없다. 그러나 정말 참조 투명성을 만족하는 처리인지, 그리고 훗날 누군가가 처리에 손을 댔다고 해도 참조 투명성을 계속해서 만족하는지 등, 일일이 프로그래머가 사람의 주의력의 한계 내에서 보장해야 한다는 사실이다.

크기에 있어 일정 수준의 처리라면 컴파일할 때에 참조 투명성을 만족하는지를 처리계가 자동으로 판단할 수 있을 것이다. 하지만 대부분의 언어에서는 만족하지 못하고 있다고 해도 컴파일 오류가 발생하지 않는다. 여기서의 "알고 있다"는 "처리계

주46 이제는 우스갯소리가 아닐지도 모르겠다.

주47 상태의 무결성을 갖는 목적 등, 여러 처리를 불가분의 관계로 실행하여 그들 모두가 문제없이 실행되었을 경우에만 반영시키는 처리.

의 기준에 있어서 언어 기능으로서의 판단"이라는 것이다.

담당한 프로그래머가 있으므로 때문에 괜찮다고 말할 수 있을지도 모르겠지만, 프로그래머 자신이 인간으로서 한계가 있는 기억력으로 언제까지 기억할 것인지, 또 코드에 손을 대는 다른 프로그래머도 마찬가지로 이해하고 있는지는 전혀 보장되지 않는다. 이에 대해 어차피 프로그래머가 언어의 제약을 깨뜨리는 것이 불가능하다면, 처리계가 처리의 성질을 판단할 수 있어 성질을 만족하지 않을 때에 컴파일 오류로 처리해 누가 처리에 손을 대도 성질을 보장할 수 있다.

실수 없이 작성할 수 있다면, 그리고 실수 없이 유지 관리해 나갈 수 있다면, 현대 컴퓨터 아키텍처에서는 명령형 언어 쪽이 빠른 경우가 많다. 진정으로 속도가 요구되는 부분이라면 Fortran 등의 병렬 계산 능력 쪽이 아직 우위성도 있다고 할 수 있다. 그러나 한번 작성하고 버려도 좋다고 확실히 말할 분야의 프로그램이라면 몰라도 대부분의 경우, 프로그램은 유지 보수 등의 업데이트를 통해 사용자에게 제공해야 한다. 사양상에서 올바른 동작을 다시 정하고 현실적인 비용과 시간 내에서, 그리고 가능하면 프로그래머에게 과도한 부담이 가지 않는 범위 내에서 프로그램에 손을 댈 필요가 있는 것이다. 병렬화 부분이 문제없이 동작하기 위한 성질[주48]로 처리계를 보장할 수 있다는 사실은 분명 프로그래머에게 큰 도움이 될 것이다.

Column **함수형 언어와 정리 증명**

소프트웨어나 하드웨어의 사양/설계/검증을 수학적인 엄밀함을 갖고서 처리하는 형식 기법이 있다. 정리 증명(theorem proving)은 형식 기법(formal method)의 한 분야로 시스템이나 프로그램, 한 개의 함수 등 사양대로 동작하는 것을 테스트에 의존하지 않고 통째로 증명하는 것이다. 테스트에서는 테스트를 통과한 경우만 올바로 동작한다고 생각한다. 결과적으로 경계값과 같은 코너 케이스(corner case)의 테스트 누락 등이 발생하여 버그로 나타나면 프로그래머의 시간을 빼앗기게 된다. 이에 반해 사양대로인 것이 증명되면, 어떠한 경우의 입력에 대해서도 올바로 동작하는 것이 보증된다.

함수형 언어는 정리 증명과 궁합이 좋은 것으로 알려져 있다. 18페이지의 칼럼에서 소개한 Coq와 Agda, Idris 등도 정리 증명을 할 수 있는 함수형 언어다.

주48 예를 들어, 병렬 라이브러리에 대한 이용자가 제공하는 처리가 "수학적인 의미의 함수여야만 한다" 등의 성질.

일반적으로 정리 증명이 아직 간단하지는 않다. 함수형 언어에 익숙한 프로그래머라도 정리 증명을 제대로 사용할 수 있는 프로그래머는 아마도 많지 않을 것이다. 그만큼 정리 증명을 하려면 비용이 소요된다.

그러나 세상에는 시스템이 버그를 만들 경우 인명과 관련된 분야가 있다. 예를 들어, 항공 우주, 의료, 원자력 등이 그러하다. 또한 암호화를 비롯한 보안 분야의 경우, 완벽하다는 것으로 가치가 인정된다. 이러한 분야에서의 형식 기법은 비용에 걸맞은 효과를 발휘할 가능성이 있다. 그 밖에도 여러 미션 크리티컬(mission critical: 핵심 업무 수행)한 분야에서 정리 증명을 비롯한 형식 기법의 효과는 점차 인정을 받으며 나아가게 될 것이다.

0.7 함수형 언어와 함수형 프로그래밍의 관계
강력한 성과를 끌어내기 위해서는 어떻게 하면 좋은가?

함수형 프로그래밍을 위해서 함수형 언어를 사용하는 편이 좋은 이유가 있다. 여기에서는 그 이유를 확인한다.

함수형 프로그래밍의 도입 — 명령형이라도 활용할 수 있는 기법

함수형 프로그래밍은 이를 위한 몇 가지 제한 사항을 지키면 아마도 많은 언어에서 가능할 것이다. 실제로 함수형 프로그래밍을 배운 후 명령형 언어를 사용하게 되더라도 그 언어에서 함수형 프로그래밍을 사용해 이득이 될 기회가 있다는 점을 알게 될 것이다.

다른 언어를 사용하더라도 함수형 프로그래밍에서 얻은 지식을 반영하여 안전한 프로그램을 목표로 하는 것은 가능하다.[주49] 함수형 프로그래밍에 대한 제약을 지키

주49 현실 문제, 새롭게 시작하는 프로젝트가 아닌 한, 함수형 언어를 사용하지 않은 업무에 있어서 함수형 언어를 사용하기 시작하는 것은, 기존의 자산이나 속박/교육 비용 등과 비추어 보면 고려해 볼 수 있는 것이다.

며 가능한 한 대입을 하지 않고, 혹여나 해야 할 경우에도 최소한의 범위에 한해 함수의 적용을 기반으로 처리를 기술할 수 있도록 하면 되는 것이다.[주50]

함수형 언어에 의한 함수형 프로그래밍의 도입

위와 같은 함수형 프로그래밍을 하려면 역시 함수형 언어가 적합하다고 말할 수 있다. 함수형 프로그래밍을 위한 제약을 지키고 규율에 맞는 프로그램을 작성하는 것을 보통은 함수형 언어가 체크해 주기 때문이다.

예를 들어, "입출력을 포함한 처리를 작성해서는 안 된다"라는 제한된 스코프(scope)를 정의할 수 있으며, 실제로 그 안에서 입출력을 포함한 처리를 작성할 수 없는 기능이 있다면 이것은 매우 유용한 기능이다. 트랜잭션처럼 롤백(rollback)이 발생할 수 있는 처리에서는 입출력 처리와 같이 실행하면 되돌릴 수 없는 것은 사용해서는 안 된다. 그런 부분이 앞서 언급한 제약의 범위로 있으면 안전한 프로그래밍을 할 수 있다.

그러나 이러한 기능이 없는 언어의 경우, 입출력 처리를 포함하지 않는다는 제약을 프로그래머에게 지키게 하려면 그 취지를 문서나 주석으로 작성하는 것밖에 방법이 없고, 정말로 입출력이 적혀 있지 않은지 코드를 일일이 살펴보는 것 이외에는 확인할 수 없는 것이 현실이다.

18페이지의 칼럼에서 소개한 함수형 언어는 함수형 프로그래밍을 간단히 하기 위해, 그리고 함수형 프로그래밍을 실시한 결과로부터 보다 강력한 성과를 도출하기 위해 기능을 발전시켜 온 언어들이다.[주51]

주50 실제, 필자는 명령형 언어를 사용할 경우 이렇게 코딩하는 버릇이 생겼다.

주51 이 점에 대해서는 0.9절 "함수형 언어를 채용하는 장점"에서 좀 더 상세하게 다루겠다. 이와 함께 참조하길 바란다.

0.8 함수형 언어의 역사
과거를 알고 미래 탐구하기

함수형 언어가 걸어온 역사를 살펴봄으로써 향후의 발전 방향을 알아보겠다.

함수형 언어의 발자취 – 지금까지

표 0.2는 함수형 언어의 등장 연표다.

함수형 언어의 배경인 **람다 계산(λ계산)**은 1930년대에 알론조 처치(Alonzo Church)에 의하여 고안되었다.[주52] 람다 계산은 그 자체가 튜링 완전(turing completeness), 즉 만능 튜링 머신(universal turing machine)과 같은 계산 능력을 가졌다. 더 직관적으로 말한다면 다른 잘 알려진 언어와 같은 표현 능력이 있다는 것을 나타낸다.

함수형 언어로서 첫 번째 실제적인 구현이라 할 수 있는 LISP[주53]는 1958년에 나타났다. LISP는 "S식"[주54]이라는 간단하고 강력한 기법을 도입했다. S식에서는 프로그램 자체를 데이터로 취급하는 것이 매우 용이하다. 파생 언어인 Scheme과 Common Lisp, Clojure를 포함하여 오늘날에도 열렬한 팬이 있는 언어다. Emacs를 사용하는 사람이라면 Emacs LISP라는 LISP 파생 언어를 사용하고 있을 것이다.

1966년, ISWIM[주55]라는 추상 프로그래밍 언어가 고안되었다. ISWIM은 람다 계산의 코어를 명령형 언어에서 구문 정의한 것으로 구현조차 주어지지 않았지만, 나중에 ML, SASL[주56] 계열의 함수형 언어 구문에 큰 영향을 주었다.

주52 A. Church "A set of postulates for the foundation of logic" (Annals of Mathematics, Series 2, 33:346–366, 1932)

주53 J. McCarthy "Recursive functions of symbolic expressions and their computation by machine" (Communications of the ACM, 3:184–195, 1960)

주54 S식은 "심벌"과 "S식의 그룹"만으로 이루어진 매우 간단한 식으로 트리 구조의 데이터로서 취급한다. 괄호가 대량으로 나온다.

주55 P.J. Landin "The next 700 programming languages" (Communications of the ACM, 9(3):157–166, 1966)

주56 Untyped의 순수 함수형 언어다.

표 0.2 함수형 언어 관련 연표

시기	언어
1930년대	람다 계산(λ계산)
1958년	LISP
1966년	ISWIM
1970년대	ML
1972년	SASL
1975년	Scheme
1981년	KRC
1984년	Common Lisp
1985년	Miranda
1987년	Clean
1990년	Haskell 1.0
1990년	SML
1991년	Coq(CoC로부터 개명)
1996년	OCaml(당시 Objective Caml)
1998년	Erlang(오픈 소스화)
1990년대	Agda
2003년	Scala
2005년	F#
2007년	Clojure
2012년	Idris 0.9

※ 18페이지의 칼럼 "다양한 함수형 언어"와 같이 참조.

1970년대 초반에 ML이 함수형 언어에 "정적 타입"을 도입하였다.[주57] ML은 그 후

주57 R.Milner "A theory of type polymorphism in programming" (Journal of Computer and System Sciences, 17(3):348–375, 1978)

많은 함수형 언어에 강한 영향을 주고 있다. SML과 OCaml, F#은 ML계의 언어이므로 문법을 계승하고 있으며, 다른 정적 타입의 함수형 언어도 타입에 의한 검증을 계승하고 있다.

1985년, 최초의 상용 순수 함수형 언어인 Miranda[주58]가 등장하였다. Miranda 자체는 SASL, KRC[주59]라는 함수형 언어의 후계 언어로 지연 평가를 제공하고 있다. Miranda의 문법을 계승하고 있는 언어도 많아, Haskell, Clean 등이나 Agda, Idris로 이 흐름은 계속되었다.

1984년부터 개발이 계속되고 있었던 CoC가 1991년 Coq로 개명되었다. Coq는 함수형 언어이지만 튜링 완전이 되지 않도록 제한해서 설계하였다. 제한이 없을 경우, 프로그램의 정지성[주60]을 나타내는 것이 어렵기 때문이었다. 정리 증명 지원계로서 프로그램의 정지성을 보여 줄 범위의 것으로만 쓸 수 있도록 제한하였던 것이다. Agda와 Idris도 같은 의도에서 유사한 제한이 도입되어 있다.

함수형 언어의 발자취 – 앞으로

여기서는 앞 절에서의 함수형 언어의 진화를 근거로 앞으로의 발전 방향에 대해, 그리고 함수형 언어가 과연 보급될 것인지에 대해 생각해 보겠다.

● 진화의 방향

함수형 언어는 지금까지보다 강한 제약을 둠으로써 안전한 프로그램을 쓸 수 있도록 하고 모듈화를 향상시키기도 하였다. 이 진화의 경향은 과거에 명령형 언어도 걸어왔던 것처럼 다음과 같은 것이 동일한 진화의 경향이라고 말할 수 있다.

- 구조화 프로그래밍의 도입
- 캡슐화의 도입

주58 URL http://miranda.org.uk/

주59 SASL 기반에 기능 추가된 함수형 언어다.

주60 무한 루프에 빠지지 않고 유한 시간에 실행을 중지하는 프로그램의 성질.

1968년 데이크스트라(Dijkstra)는 지금까지 goto문의 폐해에 대해 언급되어 왔던 위험성과 회의를 정리하여, 인간이 처리의 진척을 파악할 수 있는 제어 구조에 의한 프로그래밍이 중요하다는 것, 바로 구조적 프로그래밍을 제창하였다. 지금까지 사용되어 온 "뭐든지 할 수 있는 대신에 위험한" 것을 없애려고 한 것이다.

객체지향에 있어 객체 안의 데이터와 동작을 은폐하는 캡슐화는 객체의 내부 상태를 직접 만지지 않고 메시징에 의해서만 제어하려고 한다. 내부 상태를 직접 객체 밖에서 건드리는 경우, 객체로서는 있을 수 없는 상태에 빠지는 버그를 만들어 넣기 쉽다. 객체지향 언어에서는 객체 안의 것에 가시성을 지정하여 제한할 수 있도록 함으로써, 지금까지 사용되어 온 "뭐든지 할 수 있는 대신에 위험한" 것을 배제하려고 한다.

이러한 명령형 언어가 걸어온 사례를 봐도 강한 제약을 부여함으로써 사물을 편리하고 안전하게 하려는 접근의 유용성에 대해 일정한 이해를 얻을 수 있었다. 여기서 말하는 제약은 두 종류가 있다.

- 언어 기능으로서 부과된 제약
- 프로그래머가 부여할 수 있는 제약

전자는 예를 들어, 순수 함수형 언어에서 참조 투명성을 만족시켜야 하는 것이고, 후자는 예를 들어, 명령형 언어에서 변수에 const 등의 상수화 키워드를 붙이는 것이다.

현재의 함수형 언어는 일반적으로 명령형 언어보다도 전자의 제약이 매우 강한 경우가 많다.[주61]

이에 반해 후자의 제약은 언어상의 제약: 기술 능력[역주] 내에서 강한 것부터 약한 것까지 자유롭게 선택할 수 있다. 제약을 편리하게 제공할 수 있다고 말하는 편이 좋을지도 모른다. 필요한 곳에 필요한 정도의 제한성을 가지고 안전하게 프로그래밍할

주61 0.9절 "함수형 언어를 채용하는 장점"의 "제약의 충족을 체크해 주는 장점"에서도 좀 더 상세히 다루겠으니 같이 참조하길 바란다.

역주 코딩 방식

수 있다고 말할 수 있다. 언어상의 제약에서 서술 능력이 풍부할수록 더 복잡하고 섬세하고 강력한 제약을 기술할 수 있는 것은 말할 필요가 없다.

예를 들어, 여러분이 라이브러리를 만들면서 "이 부분은 라이브러리의 사용자가 정의하여 부여하는" 처리가 있다고 치자. 그때 여러분의 라이브러리가 제대로 동작하기 위해서는 사용자가 부여한 처리 안에서 "입출력은 사용할 수 있지만, 무엇이든 사용할 수 있는 것은 아니고, 기껏해야 파일 읽기밖에 허용되지 않는다"라는 조건을 붙여야 한다는 것을 알았다고 하자. 프로그래머가 부여할 수 있는 제약이 약한 언어에서는 이러한 세세한 조항을 강제하기는 어려울 것이다.

사용할 수 있는 방법이라고는 기껏해야 조항을 문서나 주석으로 남겨 두어 실제로 조건을 만족하는 처리를 사용자가 부여해 주길 기대하는 것뿐이다. 이에 비해 프로그래머가 줄 수 있는 제약이 강한 언어에서는 이러한 세세한 조항을 강제할 수 있으며 사용자는 실제로 이 조건을 만족하는 처리밖에 못 한다.

앞으로의 함수형 언어는 언어상의 제약의 기술 능력을 높여서, 그 제약을 언어나 라이브러리가 파악하고 사용할 수 있는 방향으로 나아갈 것이다. 예를 들어, 의존 타입은 제약의 기술 능력을 크게 향상시킨다. 앞서 언급한 바와 같이 이미 Agda, Coq, Idris 등은 의존 타입을 가지고 있으며, Haskell이나 OCaml 등에도 의존 타입의 기능을 도입하는 등 방향의 변경이 종종 있다. 그리고 강한 제약은 프로그래머가 작성한 프로그램을 더욱 효율적인 것으로 자동 변환하거나 자동으로 병렬화하는 등의 최적화와 프로그램의 정확성 증명에 이용되어 갈 것이다. 예를 들어, 동일한 처리를 하는 라이브러리 함수가 두 개 있는데, 하나는 그냥 주어진 인수로부터 순차 실행하여 계산해 주는 함수라고 상상하고, 다른 하나는 같은 인수 이외에 어떠한 제약을 나타내는 인수를 취하도록 되어 있어 프로그래머가 직접 병렬 프로그래밍하지 않아도 취득한 제약을 이용해 알아서 병렬화해서 병렬 실행해 주는 함수라고 상상해 볼 수 있겠다. 물론, 여기서 사용하는 제약을 일부러 프로그래머가 제공하지 않아도 제약에 만족함을 자동으로 검출하는 방향으로도 발전해 나갈 것이다.

● 보급 가능성

함수형 언어는 생각 이상으로 이미 보급되어 있으며, 관련 일자리도 사실상 증가

하는 추세다.[주62] 특히 이익에 민감하고 원래부터 수학적인 것들과 궁합이 좋은 금융 분야에서의 채용이 많다고 한다.

그러나 현재의 Java처럼 많은 사람들이 "누구라도 그럭저럭 할 수 있는" 상태로 되는 것을 보급 기준으로 본다면, 급격히 그 수준까지 도달할 것이라고는 생각하지 않는다. 이유는 다음의 두 가지 때문이다.

- 거의 대부분의 경우, 함수형 언어는 "그럭저럭 할 수 있는 상태(또는 그렇게라도 할 수 있는 상태)"를 허용하지 않는다.
- 아직 명령형 언어부터 프로그래밍을 시작하는 사람이 많다.

전자에 대해 함수형 언어에서는 "그럭저럭"도 할 수 있지 않으면 실행조차도 못할 정도로 언어의 취급 문제에 대한 올바른 이해를 프로그래머에게 요구한다. 이것은 버그가 적어지는 이유 중 하나이기도 하지만, "누구나 그럭저럭할 수 있는" 상태이며 문제없이 동작하는 분야라면 그냥 방해가 되는 제한으로밖에 보이지 않으므로 좀처럼 보급이 어려울 것이다.

후자에 대해서는 좀 더 단순하여, 명령형 언어부터 시작할 경우, 지금까지 설명해 온 명령형 언어와 함수형 언어의 차이에서 "어렵다", "사용하기 어렵다"라고 생각하는 사람이 많기 때문이다. 단순히 아직 명령형 언어의 일이 더 많고, 일부러 함수형 언어를 기억하지 않아도 문제를 해결할 수 있다. 아무래도 기술적 관심 이외에 함수형 언어를 사용할 강한 동기가 없다고 말하는 사람이 많을 것이다. 사람이 많음은 당연히 어떤 제품을 만들 때의 언어 선택에도 영향을 미친다. 그러한 이유로, 후자의 이유도 단순하긴 하지만 비교적 치명적인 요인이다.

그러나 최근 개발 도구는 안전성과 품질에 엄격한 부분을 쉽게 해결하도록 도와주고 있어 전자의 이유가 무너지고 있다. 사람의 목숨이나 보안에 관계되는 분야에서 실제로 동작시켜 보는 것밖에 안전성을 검증할 방법이 없는 코드 기반을 계속해서 유지보수하는 식이라면, 그것은 더 이상 "문제없이 작동한다"라고 말할 범위에 들어가지 못한다.

주62 60페이지 및 339페이지의 칼럼도 참고해 보길 바란다.

어느 정도 여유가 있는 조직이나 프로젝트 등에서 검증에 손이 많이 가는 단점이 원래부터 많지 않을 경우는 맨 파워로 어느 정도는 처리할 수 있겠지만, 완전히 검증을 마칠 수는 없거나 마감이 다가오는 등의 이유가 있는 경우의 검증은 비교적 줄어들기 쉽다. 이러한 사정에서는 언어로서의 성질이 좋으므로 시스템의 정확성을 검증하기 위한 각종 형식 기법과의 궁합도 좋은 함수형 언어는 이점이 있다. 사람의 생명이나 보안에 손상을 입히는 간과할 수 없는 위험[주63]이 있다고 판단되는 분야일수록 함수형 언어는 존재감이 커질 것이다.

0.9 함수형 언어를 채용하는 장점
선언적일 것, 제약의 충족 체크, 타입과 타입 검사, 타입 추론

함수형 언어를 사용함으로써 우리들은 어떠한 장점을 얻을 수 있는지 정리해 보았다.

선언적인 것의 장점

선언적이라는 것은 출력의 성질이 어떤 것인가에 주목하여 그것만 기술하도록 한다는 것이었다. 선언적이라는 것은 프로그램을 보다 잘 추상화하여 본질적으로 설명하려는 것과 파악하려는 것 이외의 사소한 것들로부터 프로그래머를 해방시킨다. 프로그래머는 결과가 어떤 성질을 만족하는가에 대해서만 주목하여 그 성질을 써 내려가기만 하면 된다.

예를 들어, 평균이 필요할 경우에는

주63 지속적인 사업이 곤란한 상태에 빠지거나 막대한 손해 배상 청구를 당하는 등.

```
숫자열에 대한 평균을 구하는 절차는
  전체 합계를 유지하는 수를 0으로 초기화
  요소 수를 유지하는 수를 0으로 초기화
  숫자열의 선두부터 하나씩 숫자를 꺼내어 아래의 처리를 한다.
    전체 합계를 유지하는 수에 값을 더한다.
    요소 수를 유지하는 수에 1을 더한다.
  전체 합계를 유지하는 수/요소 수를 유지하는 수의 결과를 나타낸다.
```

라는 절차적인 기술을 하는 것보다는

```
"숫자열에 대한 평균"을 만족해야 하는 성질은 "숫자열의 전체 합계/숫자열의 요소 수"다
```

라고 선언적으로 작성하는 편이 파악하기 쉽다.

성질을 만족하는 결과를 올바르게 얻을 수 있다면, 결과를 얻는 방법[주64]에 대해서는 무엇이든 좋다. 반대로 이 방법의 부분에 대한 결정을 프로그래머가 명시하지 않고 언어 측에 전달함으로써 최적화에 자유도를 갖게 할 수 있다고도 말할 수 있다. 절차적으로 작성되어 있을 경우 절차로부터 성질을 발견하지 않으면 성질을 파괴하지 않고 최적화가 가능한지 알 수 없다. 선언적으로 작성되어 있다면 지켜야 할 성질은 처음부터 나타나게 되어 있다.

제약의 충족 여부를 체크해 주는 장점

앞서 언급한 대로 함수형 언어에서는 다음과 함께 프로그래밍을 실시한다.

- 언어 기능으로서 부과된 제약
- 프로그래머가 부여할 수 있는 제약

그리고 많은 함수형 언어는 그 제약이 지켜지고 있는지를 프로그래머에게 기대하

주64 예를 들어, 변수나 분기, 루프 등을 어떻게 사용하여 결과를 얻을지에 대한 것.

지 않는다. 언어 자체가 제약을 지키고 있는지를 체크[주65]하고 만일 지켜지지 않으면 그것을 (가능하면 버그로 나타나기 전) 표면화시킨다.[주66] 따라서 언어 수준에서 사물을 편리하고 안전하게 기술할 수 있는 범위가 커진다. 프로그래머는 이 체크 메커니즘에 맡기고 다른 곳에 집중할 수 있다.

예를 들어, 가장 대표적인 제약인 "참조 투명하다"라는 것, 즉 "순수한 언어"에 주목해 보자. 순수한 언어라는 것은 수학적인 의미에서의 함수밖에 인정하지 않는 언어다. 수학적인 의미에서의 함수가 아닌 함수를 본래부터 쓸 수 없으므로 그러한 함수를 작성하거나 혹은 그런 함수로 나중에 수정하는 식으로 제약이 파괴되는 일은 발생할 수 없다.

제약의 충족을 체크해 주는 것은 프로그래머 단독으로 봐도 매우 유용하지만, 다수에 의한 개발의 경우 더욱더 그 유용성은 높아진다. 왜냐하면 언어로 체크되는 것이라면 코딩 제약이나 문서로 멤버에게 제약을 지키도록 할 필요가 없기 때문이다. 결과적으로 코딩 규약은 적어지고 작성해야 할 문서도 줄어 든다. 그리고 코드 기반에 대한 이러한 문서들이 적다는 것은 프로젝트에 대해 멤버가 새로 추가될 때 기존 코드 기반에 대한 학습 비용과 실수에 대한 위험이 적어진다고 할 수 있다.

타입과 타입 검사가 있는 경우의 장점

타입 검사를 갖는 언어에 주목하자. 앞서 언급했듯이 함수형 프로그래밍에서는 그림 0.11과 같이 함수 f의 치역이 함수 g의 정의역에 들어가 있으면, 함수 f와 g는 문제없이 합성할 수 있다.

그러나 정의역에 포함되어 있는지를 프로그래머가 체크하는 것은 힘들다. 물론 그런 귀찮은 일은 하지 않을 것이다. 강한 타입은 함수의 치역과 정의역에 적절한 타입을 제공함으로써 정의역에 주어진 타입이 정의역이 되고 치역에 주어진 타입이 치역이 포함되도록 해 준다. 그 결과 "함수 f의 치역이 함수 g의 정의역에 포함되어 있다는 것"은 "함수 f의 치역에 주어진 타입이 함수 g의 정의역에 주어진 타입과 맞는 것"

주65 타입 검사 등에 의한 체크.

주66 예를 들어, 컴파일 오류라는 형태로 표면화할 수 있다.

이라는 타입 검사의 문제가 된다. 이를 통해, 함수 f와 함수 g가 합성 가능한 함수인지를 프로그래머가 일일이 체크하지 않아도 된다. 합성할 수 없는 함수라면, 타입 검사에 실패하여 알려 주기 때문이다. 예를 들어, 다음과 같은 코드를 고려해 보자.

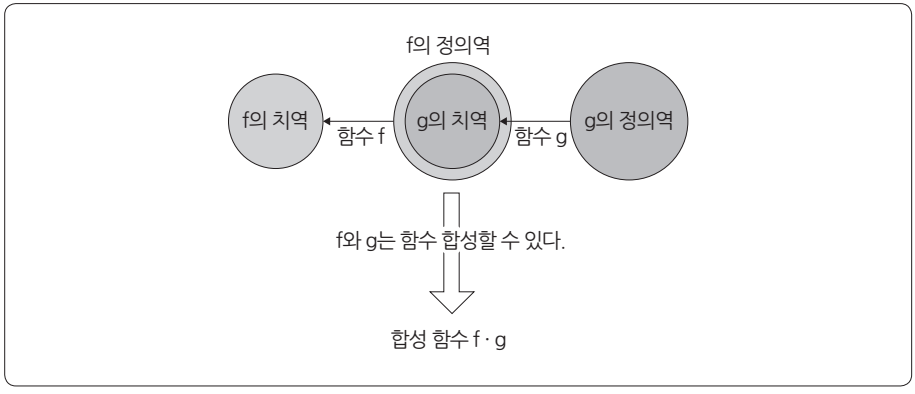

그림 0.11 함수 합성

```
# compose.rb
def f(x)
  if x < 1
    0
  else
    "foo"
  end
end

def g(n)
  n+1
end

p g(f(0)) # OK
p g(f(1)) # NG
```

아마도 이러한 코드를 작성하는 프로그래머는 없을 것이다. 그러나 여기에서는 실제로 OK행까지 실행 가능하게 하는 것 자체를 문제 삼고 있다. NG행에서 일어나는

오류는 바로 g의 정의역 내에 f의 치역이 들어가 있지 않아서 생기는 NG다. 이 샘플 코드에서 잘되지 않는 부분이 명확히 보이는 이유는, 위의 코드가 매우 작으며 상당히 자의적인 코드이기 때문이라는 점에 유의하기 바란다.

실제로 코드가 방대하고 게다가 f와 g를 서로 다른 사람이 작성하는 식의 프로젝트의 경우, 보다 복잡한 객체를 f가 분리해서 만드는 사태가 발생할 수 있고, 그런 후에 명확히 잘못된 실행 경로를 판별하는 것은 쉬운 일이 아니다. 그리고 해당 경로를 다른 영향 없이 수정 및 유지 보수할 수 있는 경우는 더 드물며 이를 위한 충분한 시간을 확보할 수도 없다.

"단위 테스트를 작성해서 한번 돌려보면 된다." 물론 그것도 좋은 방법이다. 그러나 코드가 변경되는 경우에는 테스트를 재작성해야 하고, 경로가 전부 망라되어 있는지 확인하기 위한 비용이 계속 발생한다는 것을 각오해야 한다. 타입 검사가 있으면 쉽게 알 수 있음에도 불구하고 말이다.

물론 테스트가 쓸모없다거나 불필요하다는 이야기가 아니다. 함수형 언어도 개별적으로 테스트 프레임워크는 가지고 있다. 대상으로 하는 문제에 대해 검사 기법으로서 적합한지의 여부는 고려할 여지가 있으며, 취할 수 있는 검사 기법의 선택이 많음과 강력함은 언어 선택의 시점에서 거의 결정적이다.

타입 추론의 장점

타입 추론은 겉으로 드러난 타입의 정보로부터 겉으로 드러나지 않은 부분의 타입까지 추론해 주는 기능이다.

변수의 타입을 일일이 기술하는 것은 누구나 귀찮게 느껴질 것이다.[주67] 하지만 정적 타입 언어를 학습하면 강하게 실감할 것이다. 타입은 값 또는 처리에 만족해야 하는 성질에 대해 의미를 부여하기 위해 사용할 수 있어, "프로그램의 정확성"을 보장하기 위한 중요한 요소다. 타입을 결정한다는 것은 구현에 앞서서 구현을 충족시킬 적절한 성질을 결정하는 설계 행위에 해당된다. 따라서 타입을 기술하는 것은 올바

주67 함수형 또는 명령형에 국한하지 않고 정적 타입의 언어의 단점으로 꼽히는 항목 중 하나로서 "명확히 알고 있는 타입이라도 일일이 선언해야 한다는 점"이 가장 손꼽히는 단점이다.

른 프로그램을 작성하는 데 결코 소홀할 수 없는 행위인 것이 확실하다.

타입 추론은 모든 타입을 프로그래머가 일일이 선언하는 부담에서 프로그래머를 해방시켜 준다. 추론으로부터 유도되는 타입에 대해서는 프로그래머가 타입을 기술하지 않아도 된다. Java에 도입된 다이아몬드 연산자와 C++에 도입된 auto 등, 함수형 언어가 아니라도 타입 추론[주68]이 도입되고 있는 것은 역시 프로그래머에게 도움이 된다는 증거일 것이다.

그리고 앞서 언급한 "타입은 값과 처리의 만족해야 할 성질에 대해 강하게 의미를 부여한다"라는 것과 함께 생각하면, 타입 추론이 더욱 강력한 잠재력을 가지고 있음을 알 수 있다. 즉, 타입을 유추해 준다는 것은 프로그래머를 대신해서 필요한 성질을 추론해 준다는 장점이다.

0.10 이 책에서 다루는 함수형 언어
Haskell의 특징, 구현, 환경 구축

이 절에서는 이 책에서 주로 다루는 Haskell과 그 구현, 환경 구축에 대해 알아두어야 할 기본 사항을 간단하게 정리해 두겠다.

Haskell이 갖는 특징적인 기능

이 책에서는 주로 Haskell을 다루어 설명하고 있다. 최근의 함수형 언어이며 사용자도 어느 정도 있으며, 제품의 이용 사례 또한 충분히 있다.[주69] 그리고 함수형 언

주68 강력한 것은 아니다.

주69 다음과 같은 참고가 있다.

URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

URL http://www.dataists.com/2010/12/ranking-the-popularity-of-programming-langauges/

URL http://langpop.com/

URL https://www.haskell.org/haskellwiki/Haskell_in_industry

어 중 개방형 표준을 목표로 만들어진 언어로 함수형 언어의 특징적인 기능을 가지고 있기 때문이다. 이 책에서 다루는 Haskell의 특징적인 기능을 잘 파악해 두면 다른 함수형 언어를 배울 때에도 쉽게 접근할 수 있을 것이다. Haskell에는 다음과 같은 특징이 있다.

- 순수
- 정적 타입
- 강한 타입
- 타입 검사
- 타입 추론
- 지연 평가

규격으로서 Haskell 98, Haskell 2010 등이 있다.

Haskell의 구현

Haskell의 구현은 여러 가지가 있지만, 현재의 주류는 GHC/TheGlasgow Haskell Compiler다. 이 책에서도 GHC를 이용한다. 여기에서는 GHC뿐만 아니라 다른 처리계도 간단히 소개해 두겠다.

▶ **GHC/ The Glasgow Haskell Compiler** URL **http://www.haskell.org/ghc/**

GHC는 컴파일러 및 대화식 인터프리터를 갖는 시스템. Windos, Mac OS X, Unix계(Linux 및 BSD 등)에서 동작한다. Haskell98, Haskell 2010을 지원할 뿐 아니라, 선진적인 다수의 언어 확장 기능을 갖고 있다. 현재 제공되는 Haskell 라이브러리의 대다수가 GHC를 대상으로 하고 있어 사실상의 표준(defacto standard)이다.

▶ **UHC/Utrecht Haskell Compiler** URL **http://foswiki.cs.uu.nl/foswiki/UHC**

UHC는 위트레흐트 대학(Utrecht University)에 의한 컴파일러 구현이다. Haskell 98를 지원하는 한편, 독자적으로 실험적인 확장 기능을 갖고 있다. Windows, Mac OS X, Unix계 등에서 동작한다.

▶ JHC URL http://repetae.net/computer/jhc/

JHC는 좀 미첨(John Meacham)에 의한 효과적인 프로그램의 생성을 목적으로 한 컴파일러다. Haskell 98을 지원하는 한편, FFI(Foreign function interface)계의 확장 기능을 갖고 있다. 컴파일러에서는 일단 C언어 소스 코드를 경유시킨다. 또, 런타임 자체도 3000행 정도의 C언어로 작성되어 있으므로 GHC에 비하여 매우 풋프린트가 작은 바이너리가 생성된다. 즉, 작은 풋프린트는 임베디드 시스템에 어울리는 특성이기도 하다. JHC를 fork한 Ajhc라는 프로젝트[주70]에서는 마이컴 포트를 Haskell 프로그램으로 제어하는 것도 실시하고 있다.

Haskell 환경의 구축

Haskell의 프로그래밍 환경을 구축하려면, Haskell Platform(http://www.haskell.org/platform/)을 사용하는 것이 일반적이다. Haskell Platform은 GHC를 주축으로 보통 자주 사용되는 라이브러리가 포함되어 있으며, GHC 중에서도 일반적인 목적이라면 Haskell Platform을 사용할 것을 권장하고 있다. 대략 연 2회 정도 Haskell Platform의 버전업이 이루어지고 있다.

위의 URL로부터 여러분의 환경에 맞는 Haskell Platform을 다운로드하여 인스톨하거나, 특정 Linux 환경이라면 Haskell Platform의 패키지가 있으므로 인스톨하면 된다. 이 책에서는 자세한 순서를 설명하지 않는다. 인스톨 관련 정보는 위의 URL로부터 찾을 수 있으므로 적절히 참고하길 바란다.

이후의 언급에 사용하는 것은 "Haskell Platform 2014.2.0.0"이다. 단, Haskell Platform 버전에 따라 크게 변화가 있는 부분은 이 문서에서 다루지 않으므로 별로 신경 쓰지 않아도 된다.

에디터에 대해서는 원하는 것을 사용하길 바란다. 물론 Haskell용 모드가 있는 에디터가 더 좋기는 하다. 왜냐하면 Haskell은 Python과 같이 인덴트 위치로 코드 블록을 표현하는 언어이므로, 다음의 들여쓰기 위치가 어디가 될 것인지 파악하여 결정해 주는 에디터가 아니면 비효율적이다. Emacs와 Vim에 ghc-mod라는 툴을 함께 사용하는 사람이 많으며, 최근 Sublime Text가 더 좋다는 이야기도 자주 듣는다.

주70 URL http://ajhc.metasepi.org/

● 대화식 인터프린터 GHCi의 기본적인 사용법

대화식 인터프리터를 기동시켜 보자. Haskell Platform을 인스톨한 환경에서 터미널을 기동시켜 보길 바란다. 터미널에서 ghci라고 입력하고 Enter를 누르면 대화식 인터프리터 GHCi가 기동한다.

```
$ ghci
```

인터프리터 자체의 GHCi 조작 명령어 도움말을 보려면, :?를 입력하고 Enter를 누른다. GHCi는 자체 디버거(debugger)이기도 하므로 디버깅(debugging)계의 명령도 포함되어 있다. 인터프리터를 종료하려면 :q라고 입력하고 Enter를 누른다.

당연히 파일로 작성한 프로그램을 로드할 수도 있다. Haskell 소스 코드 파일의 확장자는 일반적으로 "hs"다. foo.hs라는 파일로 프로그램을 작성하였다면, GHCi를 시작할 때 foo.hs를 런타임의 인수로 전달하면 로드된다.

```
$ ghci foo.hs
```

혹은 이미 GHCi가 기동되어 있다면 :l foo를 입력하고 Enter를 눌러서 로드할 수 있다.

```
$ ls
foo.hs
$ ghci
> :l foo
```

단, 문법이 틀리거나 타입 검사에 실패하는 프로그램이 작성되어 있다면 어떤 방법으로도 로드에 실패한다. foo.hs를 변경했다면 다시 로드할 필요가 있다. 다시 로드하려면 :r를 입력하고 Enter를 누른다.

● 컴파일러 GHC의 기본적인 사용법

컴파일러를 사용해 보자. Haskell Platform를 인스톨한 환경에서 터미널을 기동한다. 터미널에서 ghc라고 입력하고 Enter를 누르면 컴파일러가 기동한다.

```
$ ghc
```

그렇긴 하지만, 소스 코드를 건네지 않았으므로 아무 일도 일어나지 않는다.

foo.hs를 컴파일하고 링크하려면 ghc --make foo라고 입력하고 Enter를 누른다.

```
$ ls
foo.hs
$ ghc --make foo
$ ls
foo foo.hs foo.hi foo.o
```

"foo"라는 실행 바이너리가 생성된다.

또한 실행 바이너리를 생성하지 않고 스크립트 실행이 가능하다. 스크립트를 실행하려면 runghc 명령을 사용한다. 터미널에서 runghc foo를 입력하고 J를 누르면 foo.hs가 실행된다. runghc를 shebang에서 실행되도록 Haskell 스크립트를 작성할 수 있다.[주71]

대부분의 컴파일러와 동일하게 ghc에도 많은 옵션이 있다.[주72]

표 0.3에 자주 사용하는 옵션을 정리하였다. 이 책에서는 자세히 설명하지 않겠지만, 이 책의 단계에서는 -O와 -Wall 옵션을 붙여서 실행하면 그다지 이상한 일은 발생하지 않을 것이다. 그 외의 추가 옵션과 상세 설명은 앞서 언급한 가이드를 참조하길 바란다.

주71 그다지 추천하지 않는다.

주72 최근 GHC의 사용자 가이드는 다음과 같다.

URL http://www.haskell.org/ghc/docs/latest/html/users_guide/

표 0.3 자주 사용하는 옵션

옵션	설명
--help, -?	도움말을 표시한다
--make	의존성 해석을 하면서 빌드해 준다. 디폴트 모드다
-e *expression*	Expression을 평가한다
--version, -V	GHC의 버전을 표시한다
-v *N*	레벨 N의 중복 출력을 한다
-W	표준과 함께, 몇 가지 경고를 활성화한다
-Wall	몇 개의 경고를 빼고, 의심스러운 모든 경고를 활성화한다
-Werror	경고를 오류로 한다
-package p	인스톨된 패키지 p를 사용하도록 한다
-hide-package p	인스톨된 (필요 없는) 패키지 p를 사용하지 않도록 한다
-O0	최적화를 비활성화한다
-0, -01	그다지 시간을 들이지 않고 최적화를 한다
-02	시간이 걸려도 최적화를 한다
-rtsopts	런타임 시스템 옵션 처리를 활성화한다
-threaded	스레드화 런타임에 링크한다
-debug	디버그용 런타임에 링크한다
-prof	프로파일을 활성화한다

0.11 정리

함수형 언어의 개요로서 다양한 함수형 언어와 개념/특징을 소개하였다.

나머지 장의 경우, 이 책에서는 Haskell을 사용하여 설명하고 있는데, 어느 정도 Haskell을 이해하고 나면 다른 함수형 언어를 습득하는 것이 그다지 어렵지 않을 것이다. 물론 이것은 Haskell이 다른 언어에 비해 어렵기 때문이라는 뜻이 아니다. 특히 명령형 언어만 사용해 본 사람들이 많으므로 일단 함수형적인 시점을 도입해 두기만 한다면 다른 함수형 언어를 배우는 것도 그다지 어렵지는 않게 된다.

직장에서 프로그램에 관여하고 있는 사람들 중에서는 Haskell 또는 다른 함수형 언어를 조속히 도입할 만한 상황이 아닌 사람도 많을 것이다. 소스 코드까지 납품할 성과물에 포함되어 있고 개발 언어에 대한 지정도 있는 아웃소싱이거나 C언어가 아직도 강한 임베디드 분야이거나, 또는 기존 언어의 코드베이스가 거대하여 전환하자는 결정조차도 할 수 없는 경우 등, 언어 선택은 여러 가지 사정에 영향을 받는다. 물론 그 상황을 바꿀 수 있다면 이상적이긴 하지만, 현실적으로는 어려운 경우가 더 많다.

하지만 Haskell을 사용할 수 없는 환경이더라도 Haskell 또는 다른 함수형 언어의 지식을 배우는 것은 결코 낭비가 아니다. "왜 특정 언어의 기능이 다른 언어에는 없는가?"와 같이, 지금 여러분이 주로 관여하고 있는 언어와의 비교를 통하여 Haskell 이외의 언어에서도 이 책에서 얻은 지식을 적절히 피드백함으로써, Haskell을 사용할 수 있는 환경에 있는 사람은 물론, 즉시 사용할 수 없는 환경에 있는 사람에게도 프로그래밍 능력의 향상을 기대할 수 있을 것이다.

Column **현재 함수형 언어가 채택되어 있는 분야/제품**

함수형 언어는 연구나 특정 목적에서만 사용하고 있는 것이 아니라, 이미 충분히 실용 레벨로 사용되고 있다. 따라서 채택하고 있는 분야라고 했을 때, 실은 모든 분야에 사용되고 있다는 것을 유념하길 바란다. 굳이 말하자면 임베디드 분야에 대해서는 아직 약하다고 말할 수 있을 정도다. 그중에서도 특히 "금융", "코드 분석 도구", "컴파일러", "웹 서비스"

등에서 채택 사례를 자주 듣는다. 참고로, 임베디드와 마찬가지로 플랫폼의 제약으로 개발 언어로 채택되기 어려운 부분이 있기도 하지만, "게임" 분야에서도 사용되고 있다고 한다. 다양한 사례가 여기 있다.

Twitter URL https://twitter.com/

원래 트위터(Twitter)는 루비 온 레일스(Ruby on Rails) 기반의 시스템이었지만, JVM상으로 기반을 서서히 전환하였다. Ruby의 GC(Garbage Collector)[주a] 성능이 증가하는 부하를 따라잡지 못하게 된 것이 주된 이유였던 것이다. JVM에서 동작하는 함수형 언어인 Scala를 채용하고 있다. RPC 프레임워크 Finagle, 메시지 큐 Kerstrel 등 다수의 Scala 프레임워크를 사용하고 있다. Finagle은 트위터 사가 공개 및 관리하고 있다.

LinkedIn URL http://www.linkedin.com/

대량 검색 쿼리를 처리하기 위해, Scala로 구현된 프레임워크 "Norbert"를 이용하고 있다. Norbert는 ZooKeeper,[주b] Netty,[주c] Protocol Buffers[주d]"를 래핑(wrapping)하고 있으며, 클러스터 애플리케이션을 쉽게 구축할 수 있는 프레임워크다.

Foursquare URL http://foursquare.com/

원래는 LAMP 구성에서 가동하고 있던 위치 정보 SNS서비스다. 처음에는 비(非)엔지니어에 의해 PHP로 작성되었던 것 같다.[주e]

단 3개월 만에 전체의 90%가 Scala의 웹 프레임워크인 Lift로 치환되었다. 나머지 10%는 아이폰용의 변경으로 이것도 이후 2개월 만에 완료한 것 같다. 원래 어느 정도의 PHP 코드였는지는 정보를 얻을 수 없었지만, 치환 후의 단계에서 14000행의 Scala 코드와 5000행의 마크업(markup)으로 만들어졌다고 한다.

astamuse(파텐토뷰로) URL http://astamuse.com/

Scala + Lift를 메인으로 만들어진 기술 정보 데이터베이스 서비스. Scala로 특허 정보 등의 자연 언어 분석 및 데이터 마이닝(data mining)을 위한 대규모 일괄 처리를 실시하였고, 웹 서비스 구축은 Lift로 실시하였다고 한다.

comnus URL http://comnus.com/

Scala + Lift로 만들어진 SNS 서비스

주a 가베지 콜렉터(GC, Garbage Collector). 불필요한 메모리 영역의 검출 및 해제를 실시하는 메커니즘.

주b 설정 정보 집중 관리를 위한 고(高)신뢰 서버.

주c 비동기 이벤트 구동 네트워크 애플리케이션 프레임워크.

주d 바이너리 시리얼라이즈(binary serialize) 형식과 시리얼라이저(serializer).

주e URL https://docs.google.com/presentation/d/1y-uLNBl2cAaoFjXJl9FdvK2NnsiiBOteXAH5hkogZMo/present#slide=id.i0

Tabbles URL http://tabbles.net/

F#으로 만들어진 파일 관리자. 개발자 중 한 사람이 말하길, "F# 이외(특히 C#)를 사용하는 것은 생각할 수 없다. 로직(logic)의 읽고 쓰기가 어렵고, 좀 더 조기에 관리 및 유지 보수 불가에 직면했을 것이다. 아마도 C#에서는 3배의 코드 길이, 4배의 행수, 보다 많은 실행 시 버그에 시달리게 되어 전혀 만들 수 없었을 것이다"라고 했다.[주f]

WebSharper URL http://www.websharper.net/

F#의 웹 애플리케이션 프레임워크. F#으로 JavaScript/마크업 등의 프런트엔드(front-end)에서부터 백엔드(back-end)까지 작성할 수 있다.

Bump URL http://bu.mp/

스마트 폰과 스마트 폰 간이나 스마트 폰과 PC 간에서 가볍게 접촉시킴으로써 데이터를 교환할 수 있는 인기 애플리케이션. 누적 다운로드 수 1억 건 이상으로 개발한 Bump Technologies는 2013년 9월에 구글에 인수되었다.[주g] 초기에는 Python으로 개발되어 이미 개발 팀이 숙련되어 있었던 상황이었기에 빠르게 개발을 진행할 수 있었다. 그러나 규모가 커지면서 간과할 수 없는 문제의 증대나 코드의 비대화가 발생하였다. 그래서 그들은 Haskell에 눈을 돌렸다. 학습 비용은 적지 않게 있었지만, 결함 제거에 보낸 시간을 감안하면 시간이 절약되고 있는 것 같다. 특히 서비스에 기여하고 있는 점으로 신뢰성과 확장성을 들 수 있다.[주h]

COBOL의 리버스 엔지니어링 툴(NTT 데이터)

사양서가 없는 대규모 COBOL 레거시 소프트웨어 시스템에서 COBOL 코드를 리버스 엔지니어링하여 사양서를 만들어 내는 툴. 이유는 잘 모르겠지만, 인도에는 Haskell 사용자가 많다고 하는데, 인도인을 포함한 개발자 10여 명에 의해서 개발되었다고 한다. Haskell은 Haskell로 구현된 Strafunski[주i]라는 강력한 구문 분석 툴을 보유하고 있어, 함수형 언어의 특성과 함께, 프로그램의 분석 툴과 컴파일러 등의 작성에 비교적 장점이 있다. 이 툴도 StraFunski를 이용하고 있는 것 같다.[주j]

주f URL http://blogs.msdn.com/b/dsyme/archive/2010/07/08/tabbles-organize-your-files-written-in-f.aspx

주g URL http://blog.bu.mp/post/61411611006/bump-google

주h URL http://download.fpcomplete.com/bump-fp-complete-study.pdf

주i URL https://code.google.com/p/strafunski/

주j 참고: URL http://itpro.nikkeibp.co.jp/article/COLUMN/20130112/449224/
상기 툴의 관련 서비스도 릴리스되어 있다.
참고: URL http://www.nttdata.com/jp/ja/news/release/2013/042402.html

BancMeasure(신일철주금 솔루션즈)

Haskell로 개발된 회계 지원 패키지 소프트웨어. Haskell 습득 기간을 포함해 10명이서 6개월 만에 완성했다고 한다.[주k] 학습 내용은 표본이 되는 코드가 있으면 실제로 그다지 문제없이 진행할 수 있었다고 한다. 그러나 문제 해결 등을 위해 Haskell을 잘 이해하고 있는 사람이 최소한이라도 필요했다고 한다. 실제로는 Coq에 의한 정리 증명까지 하고 싶었지만, 거기까지는 습득이 곤란해서 난관에 봉착했던 것 같다.

manaba(아사히 넷) URL http://manaba.jp/

교육기관용 학습관리 시스템이다. 일본의 일부 대학생들은 어쩌면 지금 현재 이용하고 있을지도 모르겠다. 원래 Perl로 개발된 시스템이지만, 출석 관리/설문 기능에 대해서는 독립적으로 Haskell을 가지고 신규 개발하였다. 브라우저와 웹 서버 간의 양방향 통신이 가능한 WebSocket에 의하여 실시간으로 답변 상태를 확인할 수 있다. 함수형 언어로 현대적인 기능을 사용한 웹 애플리케이션을 만드는 것도 향후 일반적으로 추세가 되지 않을까 생각한다.

LexiFi URL http://www.lexifi.com/

OCaml에 의한 금융 상품의 통합 개발 환경. MLFi(Modelling Language for Finance)라는 전용의 금융 상품 기술 언어도 OCaml로 만들어져 있다. MLFi는 콤비네이터리(combinatory)의 조합으로 복잡한 계약이나 금융 상품을 정의할 수 있도록 되어 있다. 또한 정의의 의미적인 체크도 컴파일러가 수행하여 불합리한 상품을 정의해도 이를 감지할 수 있다.

XenServer의 툴들(Citrix Systems, Inc.) URL http://www.citrix.com/

잘 알려진 가상화 서버 Xen의 툴들은 13만 행의 OCaml로 작성되어 있다.

Coherent PDF Command Line Tools URL http://www.coherentpdf.com/

OCaml로 작성된 PDF 편집 툴이다. CamlPDF라는 PDF 라이브러리도 공개되어 있다.

Field Reports(필드 웍스) URL http://www.field-works.co.jp/

OCaml로 작성된 LL언어용 PDF 전표 툴. 위의 CamlPDF을 이용하고 있는 것 같다.

SCAWAR(Punch Wolf Game Studios) URL http://www.punchwolf.com/scawar/

Scala로 작성된 안드로이드(Android) 단말용의 슈팅 게임. 백엔드 부분이나 주변 툴이 아니라 게임 자체에 함수형 언어를 사용하는 예는 그리 많지 않다. 상용 게임 분야에서는 오락실의 거치형이 대부분 전용 개발 체인이며, 브라우저 게임에서는 JavaScript나 Flash에 속박되기 쉬우므로 개발에 있어 선택의 여지가 없었을 수도 있다. 안드로이드 게임에서는 Scala나 Clojure 등이 앞으로 활약할 수 있는 여지가 있다.

주k URL http://itpro.nikkeibp.co.jp/article/COLUMN/20130112/449224/

찾아보기

기호 및 숫자

A

ㅂ

ㅅ

ㅇ

ㅈ

ㅊ

ㅋ

ㅌ

ㅍ

ㅎ