2 장

모양과 뜻

프로그래밍 언어의 겉모양은 문법구조(*syntax*)를 말한다. 프로그래밍 언어의 속뜻은 의미구조(*semantics*)를 말한다. 그 두가지를 알지 못하면 그 언어를 사용할 수 없다.

제대로 생긴 프로그램을 어떻게 만들고 그 프로그램의 의미가 무엇인지에 대한 정의를 모르고는 우리는 그 언어를 사용할 수 없다. 더군다나 그 정의들은 애매하지 않고 혼동이 없어야 한다.

2.1 문법구조

문법구조(*syntax*)는 프로그래밍 언어로 프로그램을 구성하는 방법이다. 제대로 생긴 프로그램들의 집합을 만드는 방법이다. 귀납적인 규칙으로 정의된다. 이 귀납적인 규칙들로 만들어지는 프로그램은 나무구조를 갖춘 이차원적인 모습이라고 할 수 있다. 왜 나무구조인가?

Example 15 정수식을 만드는 귀납규칙을 생각해 보자.

$$\begin{array}{rcll} E & \rightarrow & n & (n \in \mathbb{Z}) \\ & | & E + E & \\ & | & - E & \end{array}$$

첫번째 규칙은 임의의 정수는 정수식이라고 한다. 하나의 정수만 말단으로 가지고 있는 나무가 되겠다. 두번째 규칙은 두개의 정수식을 가지고 +를 이용해서 정수식을 만들 수 있다고 한다. 이것이 나무 두개를 양쪽으로 매달고 있고 뿌리에는 "+"를 가지고 있는 나무가 되겠다. 마지막 규칙은 하나의 정수식을 가지고 −를 이용해서 정수식을 만드는 것이다. 이것도 나무 하나를 매달고 있고 뿌리에는 "-"를 가지고 있는 나무가 된다. □

Example 16 간단한 명령형 언어를 생각해 보자. 이 언어로 짤 수 있는 프로그램 C는 명령문(command)으로서 다음의 방법으로 만들어 진다:

$$
\begin{array}{rcl}
C & \rightarrow & \texttt{skip} \\
& | & x \texttt{ := } E \\
& | & \texttt{if } E \texttt{ then } C \texttt{ else } C \\
& | & C \texttt{ ; } C
\end{array}
$$

이고 E는 위의 예에서 정의한 정수식이라고 하자. 이 방법들로 만들 수 있는 명령문들은 모두 나무 구조물 들이다. "skip"만 있는 명령문은 그것만 말든으로 가지고 있는 나무가 되겠다. 두번째 규칙은 정수식 나무를 오른편에, 하나의 변수를 왼편에, 루트 노드는 ":="임을 표시하고 있는 나무가 된다. 세번째 규칙은 세개의 나무(정수식 나무와 두개의 명령어 나무)를 하부 나무로 가지고 있고 루트 노드에는 "if"문 임을 표시하고 있는 나무가 된다. 마지막 규칙은 두개의 명령어 나무를 하부 나무로 가지고 있고 루트 노드에는 순차적인 명령문(";")임을 표시하고 있는 나무가 된다. □

프로그램을 만들 때 필요한 최소한의 정보를 가지고 있는 가장 간단한 문법을 써 보자. 예 15와 예 16에서 보여준 규칙들에는 규칙을 읽는 사람을 돕기 위해서 필요 이상의 정보가 있기도 하다. 다음이 필요한 정보만 가지고 있는 더

욱 날씬한 규칙일 것이다:

$$\begin{array}{rcll} C & \rightarrow & \& & \\ & | & = x\ E & \\ & | & ?\ E\ C\ C & \\ & | & ;\ C\ C & \end{array}$$

$$\begin{array}{rcll} E & \rightarrow & n & (n \in \mathbb{Z}) \\ & | & +\ E\ E & \\ & | & -\ E & \end{array}$$

긴 심볼을 쓸 필요도 없고, 심볼이 중간에 끼일 필요도 없고, 사이 사이에 "then"이나 "else"같은 장식이 있을 필요도 없다. 오직 각 규칙마다 다른 간단한 심볼을 쓰면 그만이다. 아니면 아예 그러한 심볼이 필요없어도 된다. 우리가 각 규칙마다 구분할 수 만 있다면:

$$\begin{array}{rcll} C & \rightarrow & \& & \\ & | & x\ E & \\ & | & E\ C\ C & \\ & | & C\ C & \end{array}$$

$$\begin{array}{rcll} E & \rightarrow & n & (n \in \mathbb{Z}) \\ & | & E\ E & \\ & | & -\ E & \end{array}$$

하지만, 이렇게 까지 프로그램 만드는 방법을 최대한 요약해서 보여줄 필요는 없다. 문법 규칙들을 보고 무엇을 만드는 방법인지를 힌트해 주는 규칙들을 사용하게된다.

2.1.1 요약된 혹은 구체적인

지금까지 보아온 문법규칙들을 "요약된 문법구조"(*abstract syntax*)라고도 하는데, 요약된 문법구조(*abstract syntax*)와 구체적인 문법구조(*concrete syntax*)의 차이는 프로그램을 만들 때 사용하는 규칙이냐, 읽을 때 사용하는 규칙이냐, 에 달려있다. 프로그램을 만들 때 사용하는 문법은 지금까지 보아온 귀납적인 방법이면 충분하다. 만든 결과는 나무구조를 가진 이차원의 구조물이다. 반면에 프로그램을 읽을 때의 문법은 더 복잡해 진다. 왜냐하면, 대게 만들어진 프로그램을 표현할 때는 나무를 그리지 않고 일차원적인 글자들의 나열이 되고 이러한 글자나 소리의 일차원적인 실을 받아서, 쓰거나 말한 사람의 머리에 그려져 있었을 2차원의 나무구조를 복원시키는 규칙들이 되기 때문이다.

구체적인 문법(*concrete syntax*)은 얼마나 구체적일까? 나무구조가 아니라 일차원의 실로 표현되는 다음의 정수식 프로그램을 생각해 보자:

```
-1+2
```

위의 것은 다음 두개의 나무구조중 하나를 일차원으로 펼친것이다:

$$\langle\langle -\ 1\rangle\ +\ 2\rangle \qquad -\langle 1\ +\ 2\rangle$$

어느것인가? 프로그램 실 "`-1+2`"를 둘 중의 어느 프로그램으로 복구시킬 것인가? 요약된 문법

$$\begin{array}{rcll} E & \rightarrow & n & (n \in \mathbb{Z}) \\ & | & E\ \texttt{+}\ E & \\ & | & \texttt{-}\ E & \end{array}$$

으로는 둘 중에 어느 구조로 복구시켜야 할 지 알 수 없다. 두가지 구조를 모두 구축할 수 있다.

정수식의 문법이 다음과 같다면 어떨까?

$$\begin{array}{rcll} E & \rightarrow & n & (n \in \mathbb{Z}) \\ & | & E \texttt{ + } E & \\ & | & \texttt{ - } F & \\ F & \rightarrow & n & \\ & | & \texttt{ (} E \texttt{)} & \end{array}$$

위의 규칙이 말하는 것은, 음의 부호가 앞에 붙은 정수식인 경우는, 두가지 밖에 없다: 말단 자연수이거나, 괄호가 붙은 정수식이거나. 따라서, 프로그램 실 "-1+2"의 경우는 $\langle\langle -1\rangle + 2\rangle$의 구조밖에는 없다.

구체적인 문법구조는 일차원의 프로그램실을 받아서 혼동없이 2차원의 프로그램 구조물을 복구하는 데 사용되는데, 이 문법은 프로그램 복원(*parsing*) 혹은 프로그램의 문법검증(*parsing*)이라는 과정의 설계도가 된다. 그 과정은 우리가 일상적으로 프로그램을 쓰면(문자들의 1차원 실로) 컴퓨터가 그 구조물을 자동으로 복원하는 과정이다.

우리는 구체적인 문법을 더 이상 다루지않는다. "프로그램"이라고 하면, 요약된 문법 규칙들로 만들어진 이차원의 나무구조를 뜻한다. 비록 이곳에 적는 프로그램들이 그 나무구조를 직접 그리지는 않았더라도, 그 2차원의 구조물이 무엇인지 혼동되지 않도록 적절히 괄호를 쓰거나 할 것이다.

2.2 의미구조

의미구조(*semantics*)는 프로그램이 뜻하는 바를 정의한다. 프로그램이 뜻 하는 바는 무엇인가? "1+2"라는 프로그램은 무엇을 뜻하는가? 결과값 3을 뜻하는가, 1과 2을 더해서 결과를 계산한다, 는 과정을 뜻하는가? 3을 뜻한다고 정의하는 스타일, 뜻하는 바 궁극을 수학적인 구조물의 원소로 정의해 가는 스타일을 "궁극을 드러내는 의미구조"(*denotational semantics*)라고 한다. 반면에 프로그램의 계산 과정을 정의함으로써 프로그램의 의미를 정의해 가는 스타일을

"과정을 드러내는 의미구조"(*operational semantics*)라고 한다.

다양한 스타일의 의미구조 정의법은 모두 충분히 엄밀하다. 그리고 각 스타일마다 프로그램 의미의 성질을 증명하는 다양한 기술이 개발되어 있다.

그럼 왜 여러가지 스타일을 살펴볼 필요가 있을까? 그것은 다양한 의미구조 정의 방식과 증명기술들이 모두 종합적으로 사용되는 것이 흔하기 때문이다. 어떤때는 궁극의 스타일이 적절하고 어떤때는 과정을 드러내는 스타일이 적절하다. 각 경우마다 우리가 드러내고 싶은 디테일의 수준 만큼만 표현해 주는 의미구조 방식을 취사선택하게 된다.

2.2.1 궁극을 드러내는

궁극을 드러내는 의미구조(*denotational semantics*)는 프로그램의 의미를 전통적인 수학의 세계에서 정의한다. 프로그램이 뜻 하는 바를 수학의 물건으로 정의하는 것이다. 영어 "denotational semantics"를 "지칭하는 바를 드러내는 의미구조"라고 직역할 수 있지만, 그러면 그 고유 특징을 드러내지 못하기 때문에 "궁극을 드러내는"이라고 번역하였다. "denotational semantics"는 프로그램의 의미를 결정하는 한 방법을 뜻하는 고유명사이고 그 방법의 특징은, 수학의 세계에서 의미하는 바 그 궁극을 드러내는 것이다.

특히, 프로그램 부품들의 의미들이 전체 프로그램의 의미를 구성한다. 이러한 이유로 조립식 의미구조(*compositional semantics*)라고 불리기도 한다. 조립식 의미구조가 좋은 이유는 명백하다. 프로그램의 의미가 쉽게 결정된다. 프로그램 생김새만 잘 뜯어서 그 부품들을 파악하면, 그 부품들의 의미를 가지고 전체 프로그램의 의미가 결정된다.

예를 들어보자. 아래와 같은 명령형 언어(*imperative language*)를 생각해보

자. 우리가 늘 알고 있는 명령형 언어라고 보면 된다.

$$
\begin{array}{rcll}
C & \to & \texttt{skip} & \\
 & | & x \texttt{ := } E & \\
 & | & \texttt{if } E \texttt{ then } C \texttt{ else } C & \\
 & | & C \texttt{ ; } C & \\
E & \to & n & (n \in \mathbb{Z}) \\
 & | & x & \\
 & | & E \texttt{ + } E & \\
 & | & \texttt{-} E &
\end{array}
$$

프로그램은 명령문이 되고, 명령문은 기계의 메모리에 정수값들을 저장시키면서 일을 진행한다. 프로그램의 변수는 메모리의 주소를 뜻하기도 하고 (지정문 "x := E"의 왼편에 사용되는 경우) 그 주소에 저장된 정수값을 뜻하기도 한다(계산식 E에서 사용되는 경우).

이렇게 간단한 의미구조를 수학적으로 모델하기는 어렵지 않다. 우선, 명령어 C의 의미는 메모리 상태를 변화시키는 함수로 정의한다: 메모리를 받아서 메모리를 내어놓는. 메모리도 또 다른 함수로 정의한다: 프로그램 변수를 받아서 그 변수가 가지는 정수값을 내어놓는. 프로그램 변수는 프로그램에서 사용하는 변수 이름이다.

그리고, 이런 물건들이 어느 집합에 소속되는 지를 정의하자. 그러한 집합을 의미공간(*semantic domain*)이라고 한다. 아래 $Memory$, $Value$, Var, $Memory \to Memory$, $Memory \to Value$등이 의미공간이 되겠다:

$$
\begin{array}{rcl}
M \in Memory & = & Var \to Value \\
z \in Value & = & \mathbb{Z} \\
x \in Var & = & ProgramVariable \\
\text{명령문 } C\text{의 의미 } [\![C]\!] & \in & Memory \to Memory \\
\text{계산식 } E\text{의 의미 } [\![E]\!] & \in & Memory \to \mathbb{Z}
\end{array}
$$

이제, 프로그램의 의미는 프로그램의 각각의 경우에 대해서 위에 마련한 의미 공간의 원소들을 가지고 다음과 같이 정의된다:

$$\begin{aligned}
[\![\texttt{skip}]\!]\, M &= M \\
[\![x \texttt{ := } E]\!]\, M &= M\{x \mapsto [\![E]\!]\, M\} \\
[\![\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2]\!]\, M &= \mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![C]\!]M\ \mathit{else}\ [\![C_2]\!]M \\
[\![C_1 \texttt{ ; } C_2]\!]\, M &= [\![C_2]\!]\,([\![C_1]\!]\, M) \\
[\![n]\!]\, M &= n \\
[\![E_1 \texttt{ + } E_2]\!]\, M &= ([\![E_1]\!]\, M) + ([\![E_2]\!]\, M) \\
[\![\texttt{- } E]\!]\, M &= -([\![E]\!]\, M)
\end{aligned}$$

조립식인 것이 보이는가? 보기 좋다. 모든 프로그램의 의미는 그 부품들의 의미만을 가지고 구성되어진다. 예로, if-문의 의미는 그 부품들의 의미들로 정의되어 있지 않은가:

$$[\![\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2]\!] = \cdots [\![E]\!] \cdots [\![C_1]\!] \cdots [\![C_2]\!].$$

2.2.1.1 조립식일 수 있는 이유: 의미공간 이론(*domain theory*)

그런데, 이렇게 부품들의 의미로 전체의 의미를 구성하는 상식적인 방식이 일사천리 가능하지는 않다. 명령형 언어에서 반복문이 가능하다고 하면, 반복문의 의미를 반복문을 구성하는 부품들의 의미만으로 정의하는 것이 간단치 않게된다.

조립식으로 의미를 결정하기가 난처한 경우를 살펴보자. 명령문중에 반복문으로 while문을 쓸 수 있다고 하자.

$$C \ \rightarrow\ \cdots \mid \texttt{while } E \texttt{ do } C$$

while문은 조건식 E의 값이 0이면 아무것도 안하고 끝나지만, 조건식의 값이 0이 아닐 때는 명령문 C를 실행하고 난 후 다시 같은 while문을 반복하게 된다.

따라서 아래와 같이 while문의 의미를 정의해 보려고 할 수 있겠다:

$$[\![\texttt{while } E \texttt{ do } C]\!]M = \mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![\texttt{while } E \texttt{ do } C]\!]([\![C]\!]M)\ \mathit{else}\ M$$

조립식인가? 그렇지 않다. $[\![\texttt{while } E \texttt{ do } C]\!]$가 $[\![E]\!]$와 $[\![C]\!]$만 가지고 정의되지 않고 자기자신 $[\![\texttt{while } E \texttt{ do } C]\!]$도 사용되고 있다.

따라서 위의것은 $[\![\texttt{while } E \texttt{ do } C]\!]$의 정의가 아니다. 이것은 단순히 $[\![\texttt{while } E \texttt{ do } C]\!]$에 대한 방정식일 뿐이다. 그 방정식의 해가 바로 $[\![\texttt{while } E \texttt{ do } C]\!]$의 정의일 것이다.

그렇다면 그 해는 무엇일까? 해는 과연 있을까? 항상 있을까? 있다면 유일하게 있을까? 이에 대한 모든 답은 모두 예, 라고 할 수 있다. 이유는 의미방정식에서 사용하는 모든 물건들(원소들과 연산자들)이 소속된 의미공간(*semantic domain*)이 좀 특별하기 때문이다.

그러한 의미공간에 대한 이론이 의미공간 이론(*domain theory*)인데, 이 이론은 다음을 확인해 준다:

> 모든 컴퓨터 프로그램(모든 계산 가능한 함수들)의 의미(의미방정식의 해)는 의미공간 이론에서 규정하는 성질의 집합안에서 유일하게 존재하고 그것은 이러이러하다.

의미공간 이론이 규명해 낸 그러한 집합은 CPO(*complete partial order set*)라는 성질을 만족하는 집합이다. 그래서, 모든 프로그램의 의미방정식들에 쓰이는 것들은 모두 어떤 CPO의 원소들이고, 특히 방정식에서 사용하는 연산자들은 모두 CPO에서 CPO로 가는 연속함수(*continuous function*)로 정의된다. 그러면, 의미방정식의 해는 유일하게 항상 어떤 CPO안에 존재하게 된다.

프로그램 C의 의미 $[\![C]\!]$에 대한 의미방정식은 항상 다음과 같고

$$\begin{aligned} [\![C]\!] &= \mathcal{F}([\![C]\!]) \\ \text{여기서 } [\![C]\!] &\in D \qquad (\text{어떤 CPO}) \\ \text{그리고 } \mathcal{F} &\in D \to D \quad (D\text{에서 } D\text{로가는 연속함수들의 CPO}) \end{aligned}$$

($[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$의 의미방정식에 해당하는 $\mathcal{F}$는 무엇?) 이 방정식의 해는 항상 연속함수 $\mathcal{F}$의 최소고정점(*least fixpoint*)으로 정의된다. 연속함수의 최소고정점은 CPO의 성질과 연속함수의 조건때문에 항상 유일하게 존재한다. 이 내용은 다음절에서 다루기로하자.

아뭏튼, 이렇게 최소 고정점(*least fixed point*)이라는 수학적 깃법을 통해서 궁극의 의미가 조립식으로 가능해 지는 이유로해서, 궁극적인 의미구조를 고정점 의미구조(*fixpoint semantics*)라고 불리기도 한다.

2.2.1.2 CPO, 연속함수, 최소고정점

프로그램의 수학적인(궁극적인) 의미는 CPO(*complete partial order*)라는 공간의 원소로 정의된다. 어느 집합이 CPO가 되려면, 그 집합의 원소들 간에 어떤 순서가 있고(임의의 두 원소간에 순서가 있을 필요는 없다, 따라서 *"partial" order*), 모든 원소보다 아래에 있는 밑바닥 원소(주로 $\perp$으로 표현한다)가 항상 있고, 그 순서를 가지고 일렬로 줄을 세울 수 있는 원소들(*chain*)이 있다면 그 줄에 있는 모든 원소들 보다 위에 있으면서 가장 작은 원소(LUB, *least upper bound*)를 항상 가지고 있는 집합이다.

CPO D_1에서 CPO D_2로 가는 함수 $f : D_1 \to D_2$가 연속 함수란 D_1의 체인(*chain*)의 LUB를 보전해 주는 함수이다. 체인의 LUB을 취하고 함수를 적용한 결과가 함수를 그 체인의 원소들에 각각 취하고 그 결과를 LUB한 것과 일치하는 함수이다:

$$\forall chain\ X \subseteq D_1.f(\bigsqcup X) = \bigsqcup_{x \in X} f(x).$$

CPO에서 CPO로 가는 연속함수 f는 항상 최소 고정점 $fix\,f$이 유일하게 있고, 그것은

$$\perp \sqcup f(\perp) \sqcup f(f(\perp)) \sqcup \cdots = \bigsqcup_{i \in \mathbb{N}} f^i(\perp)$$

이다.(왜?)

CPO성질을 만족하는 집합들은 매우 다양하게 만들 수 있다. 집합을 가지고

CPO를 만들 수 있고, 만들어 논 CPO들을 가지고 조합해서 새로운 구조의 CPO를 만들 수 있다. 집합에 밑바닥 원소를 하나 추가하고 올려붙인($x \sqsubseteq y$ iff $x = y \vee x = \bot$) 집합은 CPO이다. CPO와 CPO의 데카르트 곱(*Cartesian product*)도 CPO이다. CPO와 CPO의 출신을 기억하는 합(*separated sum*)도 CPO이다. CPO에서 CPO로 가는 연속함수들의 집합도 CPO이다. (이렇게 CPO들을 가지고 구축한 CPO들의 원소 사이의 순서는 부품 CPO들의 순서를 가지고 정의되는데, 그 순서를 어떻게 정의해야 결과가 CPO가 될까?)

Example 17 다음과 같이 정수의 집합 $\mathbb{Z}$에서 올려붙인 집합 $\mathbb{Z}_\bot = \mathbb{Z} \cup \{\bot\}$을 생각하자. $\mathbb{Z}$ 원소들 사이의 순서는 없고, 순서는 오직 $\bot$과 $\mathbb{Z}$ 사이에만 존재한다: $\forall x \in \mathbb{Z}.\bot \sqsubseteq x$. 모든 체인은 유한한 길이를 가지고 있으면 맨 꼭대기의 원소가 그 체인의 LUB이 된다. 따라서 $\mathbb{Z}_\bot$은 CPO이다. □

Example 18 CPO D_1과 D_2의 데카르트 곱(*Cartesian product*)

$$D_1 \times D_2 = \{\langle x, y\rangle \mid x \in D_1, y \in D_2\}$$

은 CPO가 된다. 이 곱집합의 원소들 사이의 순서를 조립식(*component-wise*)으로 했을 때이다:

$$\langle x, y\rangle \sqsubseteq \langle x', y'\rangle \text{ iff } x \sqsubseteq_{D_1} x' \wedge y \sqsubseteq_{D_2} y'.$$

이때, 최소의 원소는 $\langle \bot_{D_1}, \bot_{D_2}\rangle$가 된다. □

Example 19 CPO D_1과 D_2의 합

$$D_1 + D_2 = \{\langle x, 1\rangle \mid x \in D_1\} \cup \{\langle x, 2\rangle \mid x \in D_2\} \cup \{\bot\}$$

은 CPO가 된다. 이 합집합의 원소들 사이의 순서는, $\bot$이 가장 작고, 그외에는

고향이 같은 원소들끼리 그 고향에서의 순서로 정했을 때이다:

$$\langle x, 1\rangle \sqsubseteq \langle x', 1\rangle \quad \text{iff} \quad x \sqsubseteq_{D_1} x'$$
$$\langle x, 2\rangle \sqsubseteq \langle x', 2\rangle \quad \text{iff} \quad x \sqsubseteq_{D_2} x'$$

□

여기서 잠깐, 인자가 x인 연속함수를 표현하는 방법을 이야기 하고 가자. 연속함수는 람다계산법(*lambda calculus*)에서 사용하는 함수를 표현하는 방법을 빌려서, "$\lambda x.$함수 내부" 로 표현하도록 하자. 예를들어, 1을 더하는 연속함수는 "$\lambda x.x + 1$"로 표현한다.

Example 20 CPO D_1에서 D_2로 가는 모든 연속함수의 집합 $D_1 \to D_2$ 은 CPO가 된다. 연속함수들 사이의 순서를 조립식(*point-wise*)으로 정의했을 때이다:

$$f \sqsubseteq g \text{ iff } \forall x \in D_1. f(x) \sqsubseteq_{D_2} f(y).$$

가장작은 원소는 $\lambda x.\bot_{D_2}$가 된다. □

의미방정식에서 사용하는 모든 물건들은 CPO의 원소이다. 연산자들은 모두 CPO에서 CPO로 가는 연속함수들이고, 연산자가 아닌 물건들도 모두 CPO의 원소들이다. 따라서 모든 의미방정식의 해는 항상 어떤 연속함수 $\mathcal{F}$의 고정점임을 나타내고 있고:

$$X = \mathcal{F}(X)$$

위의 방정식의 해는 $\mathcal{F}$의 최소 고정점 $fix\mathcal{F}$로 정의할 수 있다:

$$fix\mathcal{F} = \sqcup_{i \in \mathbb{N}} \mathcal{F}^i(\bot)$$

자 이제, CPO와 연속함수의 최소 고정점이라는 장치를 이용해서 while-문의 의미가 어떻게 조립식이 되는 지 보자.

우선, 우리가 직관적으로 구성한 의미공간들은 모두 CPO가 되야 한다:

$$\begin{array}{rcll} M \in \mathit{Memory} & = & \mathit{Var} \to \mathit{Value} & \text{연속함수 CPO} \\ z \in \mathit{Value} & = & \mathbb{Z}_\perp & \text{올려붙인 CPO} \\ x \in \mathit{Var} & = & \mathit{ProgramVariable}_\perp & \text{올려붙인 CPO} \\ \text{명령문 } C\text{의 의미, 연속함수 } [\![C]\!] & \in & \mathit{Memory} \to \mathit{Memory} & \text{연속함수 CPO} \\ \text{계산식 } E\text{의 의미, 연속함수 } [\![E]\!] & \in & \mathit{Memory} \to \mathbb{Z}_\perp & \text{연속함수 CPO} \end{array}$$

그 위에서 while-문의 의미방정식은

$$[\![\texttt{while } E \texttt{ do } C]\!]M = \mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![\texttt{while } E \texttt{ do } C]\!]([\![C]\!]M)\ \mathit{else}\ M$$

이다. 메모리를 받아서 메모리를 내어놓는 연속함수의 표현($\lambda M.\cdots$)으로 다시 쓰면,

$$[\![\texttt{while } E \texttt{ do } C]\!] = \lambda M.\mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ [\![\texttt{while } E \texttt{ do } C]\!]([\![C]\!]M)\ \mathit{else}\ M.$$

위의 방정식을

$$X = \mathcal{F}(X)$$

의 모양으로 보면, X는 $[\![\texttt{while } E \texttt{ do } C]\!]$에 해당하고, 연속함수 $\mathcal{F}$는

$$\lambda X.(\lambda M.\mathit{if}[\![E]\!]M \neq 0 \mathit{then} X([\![C]\!]M)\mathit{else}M) \in (\mathit{Memory} \to \mathit{Memroy}) \to (\mathit{Memory} \to \mathit{Memroy})$$

에 해당하게 되므로, while-문의 의미 $[\![\texttt{while } E \texttt{ do } C]\!]$는 위 함수의 최소 고정점으로 정의된다:

$$\begin{array}{rcl} [\![\texttt{while } E \texttt{ do } C]\!] & = & \mathit{fix}\mathcal{F} \in \mathit{Memory} \to \mathit{Memory} \\ & = & \mathit{fix}(\lambda X.(\lambda M.\mathit{if}\ [\![E]\!]M \neq 0\ \mathit{then}\ X([\![C]\!]M)\ \mathit{else}\ M)) \end{array}$$

조립식인 것이 보이는가? $[\![\texttt{while } E \texttt{ do } C]\!]$는 $[\![E]\!]$와 $[\![C]\!]$를 가지고 조립되어있다.

▶

2.2.1.3 완전히 요약된 의미구조(*fully abstract semantics*)

궁극의 스타일(*denotational semantics*)에서 한가지 재미있는 이슈: 사용하는 수학의 세계가 우리가 프로그램 의미로 관찰하고 픈 세계와 정확히 일치하느냐?

우선, 이 질문이 왜 재미있는 질문인가? 프로그램 최적화 방법을 고안했다고 하자. 그 방법은 주어진 프로그램을 마사지해서 보다 효율적인 프로그램으로 변환해 주는 방법이라고 하자. 과연 그 최적화 방법이 맞는지 알고 싶으면, 오리지날 프로그램과 그 방법으로 변환된 프로그램이 같은 일을 하는 지 확인해야 한다. 같은 일을 하는지 알아보는 것은, 프로그램의 의미가 같은지 확인하는 것이다. 그런데, 같은 프로그램들이 프로그램의 의미 정의를 따라 살펴보면 다른 것으로 나온다면, 그 의미 정의를 가지고 변환방법이 맞다고 증명하는데 쓰기는 어렵다. 사용하기에 너무 촘촘한 도구(formalism)인 것이다.

예를들어, `1+2`와 `1+1+1`의 계산 결과는 3으로 같다. 따라서 두 식은 같은 의미를 가진다. 과연 우리가 정의한 수학의 세계에서 두 식의 의미가 같은 원소로 지정되는가? 정의한 의미대로 두 프로그램의 의미를 따라가 봤더니 다른 두 개의 원소로 지정된다면 그 수학의 세계는 우리의 의도 이상으로 정교한 것이다. 같았으면 하는 원소들을 다르게 가지고 있으므로.

우리의 의도와 일치하는 수학의 세계를 구축한 궁극의 의미구조를 "불필요한 디테일이 완전히 제거된" 의미구조(*fully abstract semantics*)라고 한다. 같은 일을 하는 프로그램은 수학의 의미 세계에서도 같고, 거꾸로 수학의 의미세계에서 동일한 프로그램들이 실제로도 같은 일을 하는 프로그램이라면 그 궁극의 의미는 완벽한 해탈의 경지가 된 것이다: 불필요한 디테일이 완전히 제거된 의미구조가 된 것이다.

그런데, 이 질문에 대한 답이 어떻게 보면 쉬운것이, "불필요한 디테일"을 뭐로 정하느냐에 따라, 고민할 것도 없는 것이 된다. 예를 들어, 우리가 사용

하는 의미구조가 정의하는 바가 우리가 필요로하는 디테일의 정확한 모습이다라고 해 놓으면, 의미가 다르면 다른 것이 되는 것이고, 다른 것은 의미가 다르다고 정의되는 것이다.

◀

2.2.1.4 고정점에 대해서 증명하기: 고정점 귀납법(*fixpoint induction*)

궁극의 방식으로 정의한 프로그램의 의미는 CPO위에서 정의된 연속함수의 최소 고정점이다. 따라서 프로그램의 의미가 어떠어떠 하다는 성질은 어떤 최소 고정점이 어떠어떠 하다는 성질이다. 그 성질을 증명하는 것은, 최소 고정점의 성질을 증명하는 것이다.

CPO X에서 X로 가는 연속함수 f의 최소 고정점은

$$S = \{\bot_X, f\bot_X, f(f\bot_X), \cdots\}$$

의 최소 웃뚜껑(*least upper bound*) $\sqcup S$이다. 그런데 S 집합의 원소들은 다음의 두 귀납규칙에 의해 만들어진다:

$$\frac{}{\bot_X} \qquad \frac{x}{f(x)}$$

혹시, 최소 고정점의 성질에 대해서 증명하려면, S 집합의 모든 원소들이 그 성질을 만족하는지를 증명하면 되는 게 아닐까? 더군다나 그 증명은 흔한 귀납법이 되겠다:

- $\bot_X$이 그 성질을 만족하는 지 보인다.
- x가 그 성질을 만족하면 $f(x)$도 그 성질을 만족하는 지를 보인다.

이 증명이 이루어지면 집합 S의 모든 원소들이 그 성질을 만족하는 것이 된다.

그런데, 그렇다고 집합 S의 최소 웃뚜껑(*least upper bound*) $\sqcup S$가 그 성질을 만족하는 걸까? 그렇지는 않다. $\sqcup S$는 집합 S의 바깥의 원소가 되는 경우가 많다. 예를 들어, CPO $\mathbb{N} \cup \{\infty\}$를 생각해 보자. 원소들의 순서는 자연수의 크기 순서로 하고 ∞는 모든 자연수보다 큰 원소가 되겠다. 이때 $\mathbb{N}$ 집합을 생각해 보자. 그 집합의 최소 웃뚜껑(*least upper bound*) ∞는 $\mathbb{N}$에 있지 않다.

이러한 문제를 극복하면서, 위의 두가지를 증명하면 최소고정점에 대한 증명이 되려면, 증명하고자 하는 성질이 특별난 경우면 된다. 그러면, 위와 같이 최소고정점을 구성하는 집합 S의 모든 원소가 그 성질을 만족하면 항상 그 최소 웃뚜껑(*least upper bound*)는 그 성질을 만족하게 된다.

성질이 특별난 경우란 무엇인가? 최소고정점 귀납법을 사용해서 증명할 수 있는 성질들을 품에넣는 성질(*inclusive assertion*)들이라고 하는데, 어떤 성질이 "품에넣는" 성질인지 확인하기는 까다롭다. 대신에, 다음의 문법으로 만들어 지는 성질 AP는 모두 최소고정점 귀납법으로 증명해도 되는 성질들이다[MNV72]:

$$\begin{aligned} AP &\rightarrow AP \wedge AP \mid \forall \vec{y}.EP \\ EP &\rightarrow EP \vee EP \mid Q(y) \mid f(y) \sqsubseteq g(y) \end{aligned}$$

여기서 Q는 1차 논리식(*first order predicate*)이고 f와 g는 연속함수여야 한다.

감싸는 성질의 모두가 이렇게 만들어지는 것은 아니지만 우리가 프로그램 분석에서 증명할 대부분은 모두 위의 방식으로 만들 수 있는 것들이다.

2.2.2 과정을 드러내는

과정을 드러내는 의미구조(*operational semantics*)는 프로그램이 실행되는 과정을 정의한다. 프로그램의 실행되는 과정이라? 실행의 과정을 어느 수준 표현해야 할까? 계산과정에서 삼성의 칩들 사이에 일어나는 전기신호들의 흐름을 일일이 표현해야 할까? 아니면 1이 1을 계산하고 2가 2를 계산하고 +가 $1+2=3$이라는 등식을 적용하는 것으로 표현해야 할까? 아니면 실행되는 과정을, 프로그램과 그 계산결과를 결론으로 하는 논리적인 증명의 과정이라고 표현해야 할까?

의미구조를 정의하는 목표에 맞추어 그 디테일의 정도를 정하게된다. 대게는 상위의 수준, 가상의 기계에서 실행되는 모습이거나, 기계적인 논리 시스템의 증명으로 정의하게 된다.

과정을 드러내는 의미구조도 궁극을 드러내는 스타일(*denotational semantics*)처럼 충분히 엄밀하다. 다른 것이 있다면,

- 조립식(compositional)이 아닐 수 있다: 프로그램의 의미가 그 프로그램의 부품들의 의미들로만 구성되는 것은 아니다. 그렇게 되면 좋지만, 아니어도 상관없다.

- 하지만 귀납적(inductive)이다: 어느 프로그램의 의미를 구성하는 부품들은 귀납적으로 정의된다. 이 귀납이 프로그램의 구조만을 따라 되돌기도 하지만(이렇게 되면 조립식이 된다), 프로그램 이외의 것(의미를 드러내는 데 사용되는 장치들)을 따라 되돌기도 한다.

2.2.2.1 큰 보폭으로

프로그램의 실행이 진행되는 과정을 큰 보폭으로 그려보자(*big-step semantics*). 논리 시스템의 증명 규칙들로 표현된다. 증명하고자 하는 바 는, 명령문 C와 정수식 e에 대해서 각각

$$M \vdash C \Rightarrow M' \quad \text{와} \quad M \vdash e \Rightarrow v$$

이다. $M \vdash C \Rightarrow M'$는 "명령문 C가 메모리 M의 상태에서 실행이 되고 결과의 메모리는 M'이다"로 읽으면 된다. $M \vdash e \Rightarrow v$는 "정수식 e는 메모리 M의 상태에서 정수값 v를 계산한다"로 읽고. 모든 규칙들을 쓰면 이렇게 된다:

$$\frac{}{M \vdash \texttt{skip} \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash x \texttt{ := } E \Rightarrow M\{x \mapsto v\}}$$

$$\frac{M \vdash C_1 \Rightarrow M_1 \quad M_1 \vdash C_2 \Rightarrow M_2}{M \vdash C_1 \,;\, C_2 \Rightarrow M_2}$$

$$\frac{M \vdash E \Rightarrow 0 \quad M \vdash C_2 \Rightarrow M'}{M \vdash \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \Rightarrow M'}$$

$$\frac{M \vdash E \Rightarrow v \quad M \vdash C_1 \Rightarrow M'}{M \vdash \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \Rightarrow M'}\ v \neq 0$$

$$\frac{M \vdash E \Rightarrow 0}{M \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M}$$

$$\frac{M \vdash E \Rightarrow v \quad M \vdash C \Rightarrow M_1 \quad M_1 \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M_2}{M \vdash \texttt{while}\ E\ \texttt{do}\ C \Rightarrow M_2}\ v \neq 0$$

$$\frac{}{M \vdash n \Rightarrow n}$$

$$\frac{}{M \vdash x \Rightarrow M(x)}$$

$$\frac{M \vdash E_1 \Rightarrow v_1 \quad M \vdash E_2 \Rightarrow v_2}{M \vdash E_1 \texttt{ + } E_2 \Rightarrow v_1 + v_2}$$

$$\frac{M \vdash E \Rightarrow v}{M \vdash \texttt{-}\ E \Rightarrow -v}$$

위의 규칙들은 논리시스템의 증명규칙이라고 이해해도 된다. 명령 프로그램 C의 의미는 임의의 메모리 M과 M'에 대해서 $M \vdash C \Rightarrow M'$를 증명할 수 있으면 그 의미가 되겠다. 대게는 M이 비어있는 메모리일때 C를 실행시키는 과정을 나타내고 싶으므로, $\emptyset \vdash C \Rightarrow M'$의 증명이 가능하면 C의 의미가 된다. 그것이 증명 불가능하면 C의 의미는 없는 것이다.

Example 21 x := 1 ; y := x + 1 의 의미는 $\emptyset$메모리에 대해서 다음과 같은 증명으로 표현된다. 위의 명령문을 C라고 하자.

$$\frac{\dfrac{}{\emptyset \vdash \texttt{1} \Rightarrow 1}}{\emptyset \vdash x \texttt{ := 1} \Rightarrow \{x \mapsto 1\}} \qquad \frac{\dfrac{\{x \mapsto 1\} \vdash x \Rightarrow 1 \qquad \{x \mapsto 1\} \vdash \texttt{1} \Rightarrow 1}{\{x \mapsto 1\} \vdash x \texttt{ + 1} \Rightarrow 2}}{\{x \mapsto 1\} \vdash y \texttt{ := } x \texttt{ + 1} \Rightarrow \{x \mapsto 1, y \mapsto 2\}}$$
$$\emptyset \vdash C \Rightarrow \{x \mapsto 1, y \mapsto 2\}$$

□

이렇게 의미구조를 정의하는 방법을 "자연스런"(*natural semantics*), "구조적인"(*structural operational semantics*), 혹은 "관계형"(*relational semantics*) 의미구조라고도 불린다.

- "자연스럽"다고 하는 이유는 아마도 "추론 규칙"꼴로 구성되어 있고, "자연스런 추론규칙"(*natural deduction rules*) 이라고 불리는 추론 규칙이 있어서 그런 이름을 붙였을 것이다.

- "구조적"이라고 하는 이유는 두가지 정도다. 지금 돌아보면 당연한 모습처럼 보이지만, 이러한 스타일로 계산과정을 드러내는 의미구조 방식은 당시의 흔한 방식보다는 더욱 짜임새가 있었기 때문이다. 당시의 흔한 방식은 가상의 기계를 정의하고 프로그램이 그 기계에서 어떻게 실행되는 지를 정의한다. 이러다 보니, 기계의 실행과정중에 프로그램이 부자연스럽게 조각나면서 기계 상태를 표현하는 데 동원되기도 하고, 프로그램의 의미를 과도하게 낮은 수준의 실행과정으로 세세하게 표현하게 된다. (2.2.2.4절에서 자세히 다룸). 이러한 옛 방식을 "구현을 통해서 정의하기"(definition by implementation), " 어떻게 구현하는 지 보임으로서 무엇인지 정의하기"라고 하는데, 이것보다는 확연히 "구조적"이지 않은가. 프로그램의 구조마다 추론규칙이 정의되고, 계산과정은 그 추론규칙들이 레고블락이 되어 만들어내는 하나의 증명이 되는 것이다.

  "구조적"이라고 부르는 다른 이유는 의미규칙들이 한 집합(프로그램과 의미장치들간의 관계쌍들의 집합)을 귀납적으로 정의하는 방식이고, 그 귀납이 프로그램이나 의미장치들의 구조를 따라 흐르기 때문이다.

- "관계형"이라고도 하는 이유는 위의 추론규칙들이 관계된 짝들의 집합을 정의하는 귀납적인 방법으로도 바라볼 수 있기 때문이다. 관계된 짝들이란 $M \vdash C \Rightarrow M'$ 인 $\langle M, C, M' \rangle$와 $M \vdash E \Rightarrow v$인 $\langle M, E, v \rangle$들이다. 위의 규칙들에 의해서 그러한 관계를 맺을 수 있는 M, C, M'과 M, E, v들이 모아지는 집합.

2.2.2.2 작은 보폭으로

프로그램의 실행을 작은 보폭으로 정의해 보자(*small-step semantics*).

$$\frac{}{(M, \texttt{skip}) \to (M, \text{done})}$$

$$\frac{(M, E) \to (M, E')}{(M, x \texttt{ := } E) \to (M, x \texttt{ := } E')}$$

$$\frac{}{(M, x \texttt{ := } v) \to (M\{x \mapsto v\}, \text{done})}$$

$$\frac{(M, E) \to (M, E')}{(M, \texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2) \to (M, \texttt{if } E' \texttt{ then } C_1 \texttt{ else } C_2)}$$

$$\frac{}{(M, \texttt{if } 0 \texttt{ then } C_1 \texttt{ else } C_2) \to (M, C_1)}$$

$$\frac{}{(M, \texttt{if } n \texttt{ then } C_1 \texttt{ else } C_2) \to (M, C_1)}\ n \neq 0$$

$$\frac{(M, C_1) \to (M', C_1')}{(M, C_1 \texttt{ ; } C_2) \to (M', C_1' \texttt{ ; } C_2)}$$

$$\frac{}{(M, \texttt{while } E \texttt{ do } C) \to (M, \texttt{if } E \texttt{ then } C \texttt{ ; while } E \texttt{ do } C \texttt{ else skip})}$$

$$\frac{}{(M, \text{done} \texttt{ ; } C_2) \to (M, C_2)}$$

$$\frac{(M, E_1) \to (M, E_1')}{(M, E_1 \texttt{ + } E_2) \to (M, E_1' \texttt{ + } E_2)}$$

$$\frac{(M, E_2) \to (M, E_2')}{(M, v_1 \texttt{ + } E_2) \to (M, v_1 \texttt{ + } E_2')}$$

$$\frac{}{(M, v_1 \texttt{ + } v_2) \to (M, v_1 + v_2)}$$

이러한 스타일을 변이과정 의미구조(*transition semantics*)라고도 한다. 이 방식에서 재미있는 것은 문법적인(겉모양을 구성하는) 물건들과 의미적인(속뜻을 표현하는) 물건들이 한 데 섞이고 있다는 것이다. 이것이 문제될 것은 없다. 것 모양을 구성하는 것들과 속 뜻을 구성하는 것들이 반드시 다른 세계에서 멀찌감치 떨어져 있어야 한다는 것은, Tarski라는 수학자가 시작한 전통일

뿐이다.

프로그램 이외의 의미장치 없이 프로그램만을 가지고도 변이과정 의미구조(*transition semantics*)를 정의할 수도 있다. 예를들어, 메모리를 뜻하는 M이라는 프로그램 이외의 의미장치 없이. 예를 들어, 간단한 정수식 프로그램들의 의미는 프로그램을 다시쓰는 과정으로 정의된다: $1+2+3$ 다시쓰면 $3+3$ 다시쓰면 6. 프로그램 이외에 다른 장치가 사용되지 않는다.

2.2.2.3 문맥구조를 통해서

프로그램 실행을 프로그램을 다시 쓰는 과정으로 정의해 보자. 예를 들어, $1+2+3$ 다시쓰면 $3+3$ 다시쓰면 6. 이렇게 프로그램을 다시 써 가는 과정이 프로그램의 실행이라고 볼 수 있다(*transition semantics*).

두가지를 정의해야 한다: 프로그램의 어느 부분을 다시 써야(계산해야) 하는가? 다시 써야할 부분을 무엇으로 다시 써야 하는가?

프로그램의 어느 부분을 다시 써야(계산해야) 하는가? 이 질문에 대한 답은 "실행문맥"(*evaluation context*)에 의해서 정의된다. 실행문맥의 정의에 따라서 현재의 프로그램을 구성하다보면, 다시 써야 할 부분(계산해야 할 속 부분)이 결정된다. 재미있는 것은 그 정의가 문법적으로 가능하다는 것이다.

다음의 정의를 보자. 실행문맥을 가지고 있는 프로그램 K를 정의한다. K안에는 []가 딱 하나 있다. 그 곳이 프로그램에서 다시 써야 할 부분, 먼저 실행되야 할 부분이 된다. 그러한 부분을 품은 프로그램을 강조하기 위해 "$K[]$"라고 쓰고, 그 빈칸에 들어있는 (다시 써야할) 프로그램 부분 C 까지 드러내어 "$K[C]$"라고 표현한다.

지금까지 예로 사용되어 온 언어에서, 다시 써야 할 부분 []을 내포한 실행

문맥이 문법적으로 다음과 같이 정의된다:

$$\begin{array}{rcl} K & \rightarrow & [] \\ & | & x \texttt{ := } K \\ & | & K \texttt{ ; } C \\ & | & \text{done} \texttt{ ; } K \\ & | & \texttt{if } K \texttt{ then } C \texttt{ else } C \\ & | & \texttt{while } K \texttt{ do } C \\ & | & K \texttt{ + } E \\ & | & v \texttt{ + } K \\ & | & \texttt{-} K \end{array}$$

즉, 지정문(assignment command)에서는 다음에 실행되야 할 부분 K는 오른쪽 식에 있고

$$x \texttt{ := } K$$

순서문에서(sequential command) 뒤의 명령문이 다음에 실행되야 할 부분 이라면, 앞의 명령문은 시행이 이미 끝났을 때 만이고

$$\text{done} \texttt{ ; } K$$

덧셈식에서 오른쪽 식이 다음에 실행되야 할 부분이라면, 왼쪽 식의 결과는 나와있어야 한다

$$v \texttt{ + } K.$$

다시 써야할 부분을 무엇으로 다시 써야 하는가? 이제 이 질문에 대한 답은 다시쓰기 규칙(*rewriting rule*)에 의해서 정의된다. 우선, 다시 쓸 곳은 다시 쓰면 되고

$$\frac{(M, C) \rightarrow (M', C')}{(M, K[C]) \rightarrow (M', K[C'])}$$

$$\frac{(M, E) \to (M, E')}{(M, K[E]) \to (M, K[E'])}$$

속에서 어떻게 다시 쓰여지는 가 하면:

$$\begin{array}{rcll}
(M, x \texttt{ := } v) & \to & (M\{x \mapsto v\}, \text{done}) & \\
(M, \text{done} \texttt{ ; } \text{done}) & \to & (M, \text{done}) & \\
(M, \texttt{if } 0 \texttt{ then } C_1 \texttt{ else } C_2) & \to & (M, C_1) & \\
(M, \texttt{if } v \texttt{ then } C_1 \texttt{ else } C_2) & \to & (M, C_2) & (v \neq 0) \\
(M, \texttt{while } 0_E \texttt{ do } C) & \to & (M, \text{done}) & \\
(M, \texttt{while } v_E \texttt{ do } C) & \to & (M, C \texttt{ ; while } E \texttt{ do } C) & (v \neq 0) \\
(M, v_1 \texttt{ + } v_2) & \to & (M, v) \quad (v = v_1 + v_2) & \\
(M, \texttt{- } v) & \to & (M, -v) & \\
(M, x) & \to & (M, M(x)) &
\end{array}$$

이다.

Example 22 x := 1 ; y := x + 1 의 의미를 문맥구조를 통해서 알아보면,

$$\frac{(\emptyset, x \texttt{ := } 1) \to (\{x \mapsto 1\}, \text{done})}{(\emptyset, [x \texttt{ := } 1] \texttt{ ; } y \texttt{ := } x \texttt{ + } 1) \to (\{x \mapsto 1\}, \text{done} \texttt{ ; } y \texttt{ := } x \texttt{ + } 1)}$$

다음은,

$$\frac{(\{x \mapsto 1\}, x) \to (\{x \mapsto 1\}, 1)}{(\{x \mapsto 1\}, \text{done} \texttt{ ; } y \texttt{ := } [x] \texttt{ + } 1) \to (\{x \mapsto 1\}, \text{done} \texttt{ ; } y \texttt{ := 1 + 1})}$$

다음은,

$$\frac{(\{x \mapsto 1\}, 1 \texttt{ + } 1) \to (\{x \mapsto 1\}, 2)}{(\{x \mapsto 1\}, \text{done} \texttt{ ; } y \texttt{ := } [1 \texttt{ + } 1]) \to (\{x \mapsto 1\}, \text{done} \texttt{ ; } y \texttt{ := } 2)}$$

다음은,

$$\frac{(\{x \mapsto \mathtt{1}\}, y := \mathtt{2}) \to (\{x \mapsto \mathtt{1}, y \mapsto \mathtt{2}\}, \text{done})}{(\{x \mapsto \mathtt{1}\}, \text{done} ; [y := \mathtt{2}]) \to (\{x \mapsto \mathtt{1}, y \mapsto \mathtt{2}\}, \text{done} ; \text{done})}$$

다음은

$$(\{x \mapsto \mathtt{1}, y \mapsto \mathtt{2}\}, \text{done} ; \text{done}) \to (\{x \mapsto \mathtt{1}, y \mapsto \mathtt{2}\}, \text{done}).$$

□

2.2.2.4 가상의 기계를 통해서

어떤 가상의 기계가 정의되어 있고 프로그램의 의미는 그 프로그램이 그 기계에서 실행되는 과정으로 정의된다. 기계에서 실행되는 과정은 기계상태가 매 스텝마다 변화되는 과정이 된다.

예를 들면, 정수식의 의미가 한 기계의 실행과정으로 다음과 같이 정의된다. 변수가 없는 정수식만을 생각해 보자:

$$E \ \to \ n \mid E + E \mid - E$$

정의하는 기계는 소위 말하는 "스택머신"이다. 그 기계는 스택 S와 명령어들 C로 구성되어 있다:

$$\langle S, C \rangle$$

스택은 정수들로 차곡차곡 쌓여있다:

$$\begin{array}{rcll} S & \to & \epsilon & (\text{빈 스택}) \\ & \mid & n.S & (n \in \mathbb{Z}) \end{array}$$

명령어들은 정수식이나 그 조각들이 쌓여있다:

$$\begin{array}{rcll} C & \rightarrow & \epsilon & (\text{빈 명령어}) \\ & | & E.C & \\ & | & \texttt{+}.C & \\ & | & \texttt{-}.C & \end{array}$$

기계 작동과정의 한 스텝은 다음과 같이 정의된다:

$$\begin{array}{rcll} \langle S, n.C\rangle & \rightarrow & \langle n.S, C\rangle & \\ \langle S, E_1 \texttt{ + } E_2.C\rangle & \rightarrow & \langle S, E_1.E_2.\texttt{+}.C\rangle & \\ \langle n_2.n_1.S, \texttt{+}.C\rangle & \rightarrow & \langle n.S, C\rangle & (n = n_1 + n_2) \\ \langle n.S, \texttt{-}.C\rangle & \rightarrow & \langle -n.S, C\rangle & \end{array}$$

정수식 E의 의미는 $\langle \epsilon, E\rangle \rightarrow \cdots$ 가 된다.

또다른 스타일로는, 명령어들을 정수식과는 다른, 기계 고유의 명령어들로 정의할 수 도 있다:

$$\begin{array}{rcll} C & \rightarrow & \epsilon & (\text{빈 명령어}) \\ & | & \texttt{push}\, n.C & (n \in \mathbb{Z}) \\ & | & \texttt{pop}.C & \\ & | & \texttt{add}.C & \\ & | & \texttt{rev}.C & \end{array}$$

그리고, 기계 작동과정의 한 스텝도 다음과 같이 정의된다:

$$\begin{array}{rcll} \langle S, \texttt{push}\, n.C\rangle & \rightarrow & \langle n.S, C\rangle & \\ \langle n.S, \texttt{pop}.C\rangle & \rightarrow & \langle S, C\rangle & \\ \langle n_1.n_2.S, \texttt{add}.C\rangle & \rightarrow & \langle n.S, C\rangle & (n = n_1 + n_2) \\ \langle n.S, \texttt{rev}.C\rangle & \rightarrow & \langle -n.S, C\rangle & \end{array}$$

C에 있는 첫 명령어를 수행하면서 기계상태가 변하는 과정이다.

그리고, 이제 정수식들이 스택머신의 어떤 명령어들로 변환되는 지를 정의하면 된다:

$$\begin{aligned} [\![n]\!] &= \texttt{push}\, n \\ [\![E_1 \texttt{ + } E_2]\!] &= [\![E_1]\!].[\![E_2]\!].\texttt{add} \\ [\![\texttt{- } E]\!] &= [\![E]\!].\texttt{rev} \end{aligned}$$

어떻게 정의하던간에, 이러한 기계의 과정이 매우 임의적이라고 생각될 수 있다. 가상의 기계를 어떻게 디자인 하는가? 그 기계의 디테일은 어느 레벨에서 정해야 하는가? 그에 대한 답은 프로그램의 의미를 정의하는 목적, 그에 따라 결정될 것이다.

대게 가상기계를 사용해서 프로그램의 의미를 정의하는 것은 프로그래밍 언어를 구현하는 단계에서 사용된다: 프로그래밍 언어의 번역기(*compiler*)나 실행기(*interpreter*)를 구현할 때. 프로그래밍 언어의 성질을 궁리하거나 그 언어로 짜여진 프로그램들의 관심있는 성질을 분석할 때에 사용하는 의미구조로 사용되는 예는 드믈다. 가상의 기계는 대게가 이와같은 분석에 필요이상으로 디테일(기계의 볼트 넛트들)이 드러나기 때문이다.

가상기계를 정의하는 방법을 가이드하는 프레임웍은 없지만, 가상 기계 디자인은 이렇게 하는 것이다, 라는 모범을 보이는 사례가 있다. ML을 실행하는 "Categorical Abstract Machine"이라는 가상기계를 고안한 과정을 서술한 논문[CCM87]을 읽어보기 바란다. 대상 프로그래밍 언어의 구조와 의미를 따라 신중히 궁리하면서 가상기계를 디자인한 멋진 경우이다.