5.2 Economic/Optimal Translation

우리가 정의한 tcoML-to-SECK 컴파일 과정은 주어진 tcoML 프로그램 e에 대해서 하나의 SECK 코드를 결정해 주지만, 같은 일을 하는 SECK 코드는 그 밖에 여럿이 있을 수 있으므로 같은 일을 하면서 그 실행 비용이 가장 경제적인 SECK 코드로 컴파일하는 것이 필요하다. SECK 코드의 실행중에 소모하는 시간과 메모리량을 생각하여 될 수 있으면 경제적인 SECK코드를 만들어 내야 하는 것이다.

우선, SECK 코드의 실행중에 소모하는 시간과 메모리 비용을 다음과 같이 정의하고 경제적인 코드를 만드는 예를 살펴보자:

	time	memory
`FRAME`	1	0
`DEFRAME`	1	0
`FETCH`(n)	1	0
`QUOTE`(v)	1	1
`ADD`	1	0
`READ`	1	1
`WRITE`	1	0
`COND`(C_1,C_2)	1	0
`E, FTN`(C)	1	|E|
`APP`	2	0
`RETURN`	1	0
`UNROLL`	1	0

Example 1 다음의 tcoML 프로그램을 생각해 보자:

```
1 + 2
```

SECK 코드로 변환하면

```
QUOTE(1).QUOTE(2).ADD
```

위의 코드가 실행되는 시간은 3, 메모리 소모는 2. 같은 값을 만들어 내는 SECK 코드

```
QUOTE(3)
```

의 실행 시간은 1, 메모리 소모는 1.□

Example 2 다음의 tcoML 프로그램을 생각해 보자:

```
(fn x => x+1) read
```

SECK 코드로 변환하면

$$\texttt{FTN(FETCH(0).QUOTE(1).ADD.RETURN).READ.APP}$$

위의 코드가 실행되는 시간은 8, 메모리 소모는 2:

$$\begin{array}{ll} & ([],[],\texttt{FTN(FETCH(0).QUOTE(1).ADD.RETURN).READ.APP},[]) \\ \to & ((\texttt{FETCH(0).QUOTE(1).ADD.RETURN},[]),[],\texttt{READ.APP},[]) \\ \to & (v.(\texttt{FETCH(0).QUOTE(1).ADD.RETURN},[]),[],\texttt{APP},[]) \\ \to & \cdots \end{array}$$

같은 일을 하는 SECK 코드로는:

$$\texttt{READ.FRAME.FETCH(0).QUOTE(1).ADD.DEFRAME}$$

실행시간은 6, 메모리 소모는 2. 같은 일을 하는 SECK 코드:

$$\texttt{READ.QUOTE(1).ADD}$$

의 실행시간은 3, 메모리 소모는 2. □

경제적인 SECK코드를 만들어 내는데에는 주어진 tcoML 프로그램을 표면적으로 (혹은 소극적으로) 분석하면 어느정도는 달성할 수 있다. 예를 들어 tcoML 프로그램의 식중에

$$\texttt{(fn x =>}\ e_1\texttt{)}\ e_2$$

이 있으면, 위의 식을

$$\texttt{let x =}\ e_2\ \texttt{in}\ e_1$$

로 바꾸면 된다.

그렇지만 다음의 경우는 위와같은 변환이 불가능하다:

$$\texttt{fn x =>}\ \cdots\ \texttt{(x 10)}\ \cdots$$

변수 x 가 어느 함수를 가지게 될 지는 알 수 없으므로 어쩔 도리 없이 FTN과 APP 를 가지는 SECK 코드를 만들어 낼 수 밖에 없다.

위의 예에서 변수 x에 전달되는 함수들을 미리 예측해 볼 수 있다면? 그래서, 주어진 함수 호출식

$$e_1\ e_2$$

에서 e_1이 실행중에 가질 수 있는 함수가 오직 하나의 함수

$$\texttt{fn x =>}\ e$$

라는 사실이 밝혀진다면, 위의 함수 호출식을

$$\texttt{let x = } e_2 \texttt{ in } e$$

로 바꾸어서 경제적인 코드로 변환시킬 수 있다. 단, 이때 위의 함수 `fn x => `e가 정의될 때의 환경은 호출되는 현재의 환경과 같거나 $fv(e) = \{\texttt{x}\}$ 이어야 한다. Why?

위와 같은 변환을 "function inlining"이라고 하는데, 얼마나 경제적일까?

$e_1\ e_2$		`let x =` e_2 `in` e
$T(e_1) + T(e_2) + 2 + T(e) + 1$	(시간)	$T(e_2) + 1 + T(e) + 1$
$M(e_1) + \lvert\texttt{E}\rvert + M(e_2) + M(e)$	(메모리)	$M(e_2) + M(e)$

함수 뿐 아니라 보통 식

```
x + 1
```

에서도, 변수 `x` 가 가질 수 있는 값이 3 뿐이라면, 위의 덧셈식을 곧바로 `QUOTE(4)`으로 바꿀 수 있다. 이러한 변환을 "constant propagation"이라고 한다.

조건문

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

에서, 조건식 e_1이 가지는 값이 항상 true라면 위의 식을 곧바로 e_2로 대체시킬 수 있다. 이때 e_3 파트는 필요없는 코드가 된다. 이러한 변환을 "dead-code elimination"이라고 한다.

위와 같이 주어진 tcoML 프로그램의 내부 식들이 실행중에 가질 수 있는 값들을 최대한 미리 예측해 내고, 이것을 바탕으로 경제적인 SECK코드를 만들어 낼 수 있다. 다음장에서는 프로그램 식들이 실행중에 가질 수 있는 성질을 (여기서는 값들을) **안전하고 신속하게** 예측하는 기술을 살펴보도록 한다. 이러한 기술을 "static analysis", "compile-time program analysis", "semantic-based program analysis", 혹은 "data-flow analysis"등으로 불리워진다.