5.3.2 Compile-time Analysis: Our Process

프로그램이 실행중에 가질 수 있는 성질(e.g., 식들이 계산하는 값은 어떤 값일까? 식들이 계산중에 메모리를 얼마나 잡아먹을까? 식들이 혹시 발생시킬 수 있는 예외상황이 있을까?)을 미리 분석하는 기술은 어떤 것이든지 다음의 두가지 과정을 거친다:

1. 프로그램으로부터 연립방정식(혹은, 관계식)을 도출한다.
2. 그 연립방정식을 푼다.

우리가 관심있는 성질은 각 프로그램식들이 가질 수 있는 함수값들의 집합인데, 컴파일러가 이것을 정확하게 예측하는 것은 불가능하다. 함수값은 프로그램의 실행 의미식(Section 2)에서 정의한 바와 같이, 함수의 **텍스트**와 그 함수가 정의될 때의 **환경**으로 정의되고 정의될 때의 환경은 실행중에 결정되기 때문에 [3] 무한히 많은 함수값들이 만들어질 수 있고 이 값들을 컴파일러가 **정확하게** 예측해낼 수 는 없다.[4]

하지만, 우리가 함수의 텍스트 부분만을 고려한다면 무한히 많을 수 있는 실행중의 함수값들을 유한한 시간내에 **안전하게** 예측하는 것은 가능해 진다. 프로그램에 정의된 함수 텍스트의 갯수가 유한하고, 그 유한한 것을 가지고 실행중에 만들어질 수 있는 무한한 함수값들을 **요약**할 수 있기 때문이다.[5]

그럼, 우리의 분석에서 연립 방정식을 구성하는 변수들(풀어야 할 "X"들)은 무엇이 되야할까? 프로그램에 있는 변수들과 그 외의 식 마다 변수 하나씩을 설정하자. 즉, 우리의 분석은 프로그램 변수나 식들이 가지는 함수 텍스트를 예측하는 연립 방정식을 도출해서 그 해를 찾도록 하자.

연립방정식을 구성할 부품들이 다음과 같은 집합제약식(set constraint)로 표현된다:

$$X \supseteq Y, \quad X \supseteq \{\lambda x.e\}, \quad \text{혹은} \quad X \supseteq \{f\lambda x.e\}.$$

왜 "부품" 이냐? 프로그램 식들을 훑으면서 방정식을 구성하는 조각조각들이 위와 같은 집합제약식의 꼴로 표현되기 때문이다. 프로그램 전체를 모두 훑은 후에야 모아진 부품들을 가지고 연립방정식이 만들어 진다. 예를 들어서,

$$\begin{array}{rcl} X & \supseteq & Y \\ X & \supseteq & \{\lambda x.e\} \\ X & \supseteq & Z \end{array}$$

[3] 다음의 함수 g를 보자: `g = fn x => (fn y => x + y)`. 이 함수는 불려질 때마다 새로운 함수를 만들어 낸다: 1을 인자로 받으면 1을 더하는 함수 $(\langle \lambda y.x+y, \{x:1\}\rangle)$ 를 만들고, 2를 인자를 받으면 2를 더하는 함수 $(\langle \lambda y.x+y, \{x:2\}\rangle)$ 를 만들어낸다. 함수 g가 끝없이 다른 정수 값을 인자로 받으면서 불려지면? 그만큼 다른 함수들이 실행중에 만들어 지는 것이다.

[4] Undecidable. If such a compiler exists, then by using the compiler we can solve the halting problem; let's define `(fn y => y 1)(p 1)` and use the compiler to analyze the `y`. If it concludes `y` has no value then we decide `p` does not halt on input `1`.

[5] This theory is called *abstract interpretation*. This theory has been developed for the last 20 years since the seminal paper by Patrick Cousot in 1977.

와 같이 X에 대한 조각들이 모아졌으면 그 조각들로 부터 만들어지는 X에 대한 방정식은 다음이 된다:

$$X = Y \cup \{\lambda x.e\} \cup Z.$$

Figure 3에서는 우리가 도출할 연립방정식의 부품들을 모으는 규칙을 정의하고 있다.

Example 3 다음의 프로그램들에 대해서 연립방정식을 도출해 보자. See in class.□

```
1)      let
          f = fn x => x+1
          g = fn x => x+2
        in
          (if read > 0 then f else g) read
        end
2)      let
          rec f = fn n => if n=0 then 1 else n*f(n-1)
        in
          f read 1
        end
3)      let
          f = fn x => (fn y => x+y)
        in
          (f read) ((f read) read)
        end
4)      let
          f = fn x => (fn y => x y)
        in
          (f (fn x => x+1)) ((f (fn x => x+2)) read)
        end
```

주어진 프로그램 $\wp$ 에 이 규칙들을 적용해서 ($\wp \triangleright \mathcal{C}$) 모아진 방정식의 부품 $\mathcal{C}$ 들을 가지고 연립방정식을 만들면 다음의 꼴이 되고:

$$\begin{array}{lcl} X_1 & = & \mathcal{F}_1(X_1, \cdots, X_n) \\ X_2 & = & \mathcal{F}_2(X_1, \cdots, X_n) \\ \vdots & & \vdots \\ X_n & = & \mathcal{F}_n(X_1, \cdots, X_n) \end{array}$$

위의 연립방정식을 만족시키는 해답이 우리 분석의 결과가 된다.

$$c \;\triangleright\; \{X_e \supseteq \emptyset\}$$

$$x \;\triangleright\; \{X_e \supseteq X_x\}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1}{\texttt{fn}\, x \texttt{ => } e_1 \;\triangleright\; \{X_e \supseteq \{\lambda x.e_1\}\} \cup \mathcal{C}_1}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1 \qquad e_2 \;\triangleright\; \mathcal{C}_2}{e_1\; e_2 \;\triangleright\; \{X_e \supseteq X_{e'}, X_x \supseteq X_2 | \lambda x.e' \in X_1\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1 \qquad e_2 \;\triangleright\; \mathcal{C}_2}{\texttt{let}\; x \texttt{ = } e_1 \;\texttt{in}\; e_2 \;\triangleright\; \{X_x \supseteq X_1, X_e \supseteq X_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1 \qquad e_2 \;\triangleright\; \mathcal{C}_2}{\texttt{let rec}\; f \texttt{ = fn } x \texttt{ => } e_1 \;\texttt{in}\; e_2 \;\triangleright\; \{X_f \supseteq \{\lambda x.e_1\}, X_e \supseteq X_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1 \qquad e_2 \;\triangleright\; \mathcal{C}_2 \qquad e_3 \;\triangleright\; \mathcal{C}_3}{\texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 \;\triangleright\; \{X_e \supseteq X_2, X_e \supseteq X_3\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1 \qquad e_2 \;\triangleright\; \mathcal{C}_2}{e_1\; op\; e_2 \;\triangleright\; \{X_e \supseteq \emptyset\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$\texttt{read} \;\triangleright\; \{X_e \supseteq \emptyset\}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1}{\texttt{write}\; e_1 \;\triangleright\; \{X_e \supseteq X_1\} \cup \mathcal{C}_1}$$

$$\frac{e_1 \;\triangleright\; \mathcal{C}_1}{\texttt{(}e_1\texttt{)} \;\triangleright\; \{X_e \supseteq X_1\} \cup \mathcal{C}_1}$$

그림 3: Rules for Deriving Set Constraints for Estimating Function Values

그 해답은 바로 다음의 함수 $\mathcal{F}$의 고정점 (fixpoint) 과 같다:

$$\begin{array}{rcl}\mathcal{F}(X_1,\cdots,X_n) & = & (\mathcal{F}_1(X_1,\cdots,X_n),\\ & & \mathcal{F}_2(X_1,\cdots,X_n),\\ & & \vdots\\ & & \mathcal{F}_n(X_1,\cdots,X_n))\end{array}$$

헌데, 위의 연립방정식의 해답(혹은 위 함수 $\mathcal{F}$의 고정점)이 반드시 하나만 있지는 않다. 예를들어, 다음의 연립 방정식을 생각해 보자:

$$\begin{array}{rcl}X_1 & = & \{1\}\cup X_2\\ X_2 & = & X_1\cup X_3\\ X_3 & = & X_2\end{array}$$

위에서 X_i가 가질 수 있는 값이 $\{1,2,3\}$의 부분집합이라고 하면, 위의 방정식의 해는 $X_1 = X_2 = X_3 = \{1\}$, $X_1 = X_2 = X_3 = \{1,2\}$등 여럿이 있을 수 있다.

우리는 그 해답들 중에서 **가장 작은** 해답을 원한다. 왜? 가장 작은 해답만 하더래도 분석의 안전성을 보장해 주기때문이다. 분석의 안전성은 다음으로 정리되고 분석을 디자인하는 사람은 이것을 증명해야한다:

Theorem 7 (Analysis' Safety) *Let $\mathit{Values}_\wp(e_i)$ be the set of values that expression e_i of program $\wp$ will have at run-time. Let $\mathit{Sol}(\mathcal{F})$ be a solution of the equation $\mathcal{F}(X_1,\cdots,X_n) = (X_1,\cdots,X_n)$. Then,*

$$\forall\langle\lambda x.e,\sigma\rangle \in \mathit{Values}_\wp(e_i).\lambda x.e \in \mathit{Sol}(\mathcal{F})(X_i).$$

연립방정식의 가장작은 해는 함수 $\mathcal{F}$의 가장 작은 고정점 (least fixpoint)에 해당한다. 가장작은 해는 항상 존재하는 것일까? 즉, 함수 $\mathcal{F}$ 의 가장 작은 고정점은 항상 존재하는 것일까? 존재한다면 그놈은 어떤 놈일까? 다음이 이러한 질문의 답을 정리해준다:

Theorem 8 (Least Fixpoint) *Continuous function $F : A \to A$ over complete partial-order set A has the least fixpoint*

$$\bigcup_{n\geq 0} F^n(\perp)$$

where $F^0(x) = x$ and $\perp$ is the least element in A.

If the set A is a finite power set, ordered by the set-inclusion, then A is a complete partial-order set. F is then continuous if it is monotonic.

Our set of equations derived from the input program $\wp$ are composed with monotonic operator $\cup$ (the set-union) and the set that our unknowns (X_i's) range is the powerset (with the set-inclusion order) of the function texts in $\wp$ hence a finite set. Thus, the least fixpoint of the $\mathcal{F}$ exists and is computed as:

$$\bigcup_{n\geq 0} \mathcal{F}^n(\emptyset, \cdots, \emptyset).$$

Now, the question. When should the iteration stop? Deriving an efficient algorithm is a homework to come and an important component that you have to implement in your optimizing tcoML compiler.

6 Typed Intermediate Language: tySECK Machine

지구상의 모든 컴퓨터가 초고속 넷트웍으로 연결되면 지금까지와는 판이한 컴퓨팅 환경이 만들어진다. 이러한 컴퓨팅 환경에서는 프로그램 실행중에 필요한 코드부품이나 데이타들이 한 컴퓨터에 모여있지 않고 전 세계의 컴퓨터에 흩어져 있게 된다. 프로그램 부품들은 그 부품에 특화된 기술을 갖춘 팀에의해 최상의 품질로 넷트웍을 통해 지체없이 제공될 수 있기 때문이다. 모든 프로그램은 지구위를 광속으로 돌아다니면서 필요한 곳에서 필요한 자원(코드부품, 데이타, CPU, 메모리등)을 적시에 동원하여 작업을 끝마치는 것이 일상적이 되는 것이다.

이러한 컴퓨팅 환경에 최적으로 적응하는 코드를 만들기 위해서는 지금까지 생각지 못한 많은 문제들이 해결되야한다. 물론, 현재의 컴퓨팅환경(desktop computing) 에서도 실행코드의 부품과 데이타를 적시에 동원해서(디스크로부터 불러들여서) 프로그램을 실행하는 것은 가능해왔던 일이다. 그러나 미래의 전 지구적 컴퓨팅 환경에서 문제가 되는 것은, 그러한 일들이 전세계의 모든 컴퓨터를 묶은 초고속 넷트웍을 타고 **엄청난 양과 빈도로** 발생하게 된다는 것이다. 전세계 모든 코드가 광속으로 지구위를 움직이면서 사용자에게 최적의 성능을 제공하려고 경쟁하는 환경에서 그러한 일들이 일어나는 것이다. **이러한 환경에서 최적으로 수행되는 코드**는 다음의 안전성 문제를 해결하고 있어야 한다.

- 코드의 안전성 보장: 넷트웍을 통해 외부에서 제공될 불특정 다수의 코드가 안전하다는 것을 미리 검증한 후에 실행을 허락해야 한다.[6] 코드를

[6] 비유하자면, 전세계의 전화는 하나의 넷트웍으로 연결되어있다. 전화통화의 내용이 무엇이 되야 할 지 모르는 상태에서 벨만 울리면 무조건 받아도 되는가? 전화로는 목소리가 들려올 뿐이다. 그 내용이 전화를 고장내거나 듣고 있는 사람을 미치게 할 수 는 없다. 하지만, 컴퓨터로 전달되는 프로그램들은 컴퓨터를 미치게 만들수 있다.

> 제공하는 측에서 안전성을 **검증**해줘야 할 뿐 아니라, 받아든 측이 그 검증내용을 다시 **확인**할 수 있도록 해야 하는데, 그 안전성의 조건에는 진행중인 프로그램을 파행적으로 중단시키지 말아야한다는 기본적인 조건(type safety)뿐 아니라, 자원(CPU, 메모리)을 어느 이상은 사용하지 말아야 하는 등의 광범위한 제약조건이 모두 포함된다.

요즈음 얘기되는 agent code라는 것, Java의 applet으로 그 단초가 선보여진 것등이 그 얘가 된다.

현재 우리의 타겟언어인 SECK 코드에 타입 시스템을 덧붙여서 SECK 코들를 받아든 호스트에서 실행하기 전에 (수화기를 들기전에) 그 코드의 안전성을 미리 검증할 수 있도록 만들어 보자. 이러한 언어를 tySECK 이라고 부르도록 하고, tcoML 에서 tySECK 으로의 컴파일러 과정을 밝혀가도록 하자.

다음의 도표는 전지구적 컴퓨팅환경에서 컴파일러와 코드사용자의 역할을 정리한 것이다:

Code provider (compiler)	Code consumer (user)
tcoML 검증	tySECK 검증
tcoML → tySECK	tySECK → k86
tySECK 송신	SECK 혹은 k86 실행

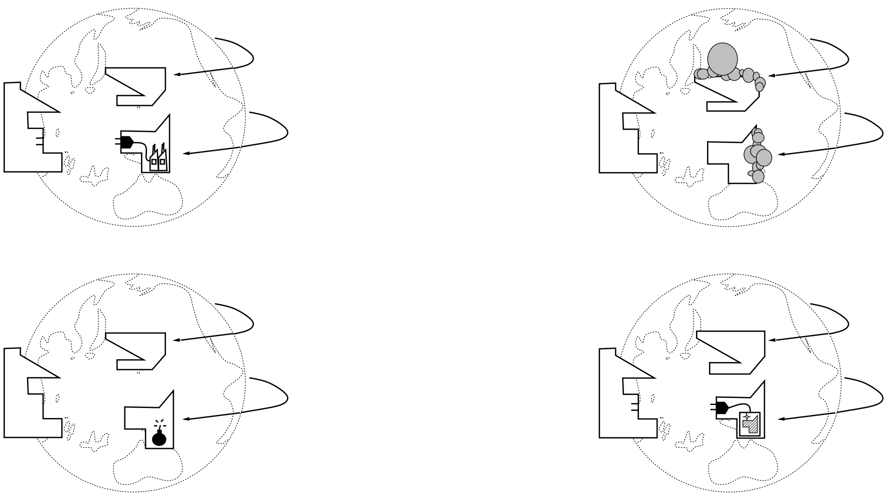

그림 4: 전 지구적 소프트웨어 부품의 문제에 대한 개념도