The tcoML Language

Kwangkeun Yi
http://cs.kaist.ac.kr/~kwang
KAIST

CS420/Spring 1999

1 Syntax

The syntax of our source language tcoML is:

e	::=	*const*	constant
	|	*id*	identifier
	|	`fn` *id* `=>` *e*	function
	|	*e e*	application
	|	`let` *bind* `in` *e* `end`	local block
	|	`if` *e* `then` *e* `else` *e*	branch
	|	*e op e*	infix binary operation
	|	`read`	input
	|	`write` *e*	output
	|	`(`*e*`)`	
bind	::=	*id* `=` *e*	value binding
	|	`rec` *id* `=` *e*	recursive value binding
op	::=	`+` | `-` | `*` | `/` | `=` | `<` | `>` | `and` | `or`	
const	::=	`true` | `false` | *string* | *num*	
id			alpha-numeric identifier
string			string
num			integer

1.1 Identifiers

Alpha-numeric identifiers are [`a-zA-Z`][`a-zA-Z0-9_'`]*. Identifiers are case sensitive: `z` and `Z` are different. The reserved words cannot be used as identifiers.

```
fn let in end if then else read write rec and or true false
```

1.2 Strings/Numbers/Comments

Strings begin and end with ". Inside the two double quotes, any sequence of characters excepting the new-line and the " characters can appear: [ˆ"ˆ\n]*. Null string "" is possible.

Numbers are integers, optionally prefixed with ~(for negative integer): ~?[0-9]$^+$.

A comment is any character sequence within the comment block (* *). The comment block can be nested.

1.3 Precedence/Associativity

The binary operators (op) precedence in decreasing order is as follows. Operators in the same set have identical precedence.

$$\{\texttt{and}, \texttt{*}, \texttt{/}\}, \{\texttt{+}, \texttt{-}, \texttt{or}\}\}, \{\texttt{=}, \texttt{<}, \texttt{>}\}$$

All the above operators are left associative. Function applications are left associative.

1.4 Syntactic Restrictions

Each value binding id = e within `rec`, e must be of the form `fn` x `=>` e. That is, only function values can be recursively declared.

2 Dynamic Semantics

$$\begin{array}{rclll}
x, f & \in & Id & & \text{identifiers} \\
v & \in & Val & = \; Num \cup String \cup Bool \cup Fvals & \text{values} \\
n & \in & Num & & \\
\langle \rho, \sigma \rangle & \in & Fvals & = \; Fexpr \times Env & \text{function values} \\
\sigma & \in & Env & = \; Id \xrightarrow{\text{fin}} Val & \text{environments} \\
\rho & \in & Fexpr & ::= \; \lambda x.e & \text{function defs} \\
 & & & \mid \;\; f\lambda x.e & \text{rec function defs}
\end{array}$$

Notations: we write $\{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\}$ for an environment σ. The domain $Dom(\sigma)$ is $\{x_1, \cdots, x_n\}$. We write $\sigma(x)$ for v if $x \mapsto v \in \sigma$. If $x \mapsto v \notin \sigma$ then $\sigma(x)$ is not defined. We write $\sigma[x \mapsto v]$ for

$$\sigma \cup \{x \mapsto v\} \quad \text{if } x \notin Dom(\sigma)$$

and

$$(\sigma \setminus \{\sigma(x)\}) \cup \{x \mapsto v\} \quad \text{if } x \in Dom(\sigma).$$

The semantics rules precisely defines how relations of the form

$$\sigma \vdash e \Rightarrow v$$

to be inferred. The relation is read "expression e computes value v under environment σ."

Definition 1 (Program's Semantics) *A program e's semantics is defined to be the inference tree of relation $\emptyset \vdash e \Rightarrow v$ for some v. If there is no such v, then the expression has no meaning.*

[Const] $$\sigma \vdash c \Rightarrow c$$

[Id] $$\frac{\sigma(x) = v}{\sigma \vdash x \Rightarrow v}$$

[Fun] $$\sigma \vdash \texttt{fn}\ x\ \texttt{=>}\ e \Rightarrow \langle \lambda x.e, \sigma \rangle$$

[App] $$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad \sigma'[x \mapsto v_2] \vdash e \Rightarrow v}{\sigma \vdash e_1\, e_2 \Rightarrow v}$$

[RecApp] $$\frac{\begin{array}{c}\sigma \vdash e_1 \Rightarrow \langle f\lambda x.e, \sigma' \rangle \qquad \sigma \vdash e_2 \Rightarrow v_2 \\ \sigma'[x \mapsto v_2][f \mapsto \langle f\lambda x.e, \sigma' \rangle] \vdash e \Rightarrow v\end{array}}{\sigma \vdash e_1\, e_2 \Rightarrow v}$$

[Let] $$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v}{\sigma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow v}$$

[RecLet] $$\frac{\sigma \vdash e_1 \Rightarrow \langle \lambda x.e, \sigma' \rangle \qquad \sigma[f \mapsto \langle f\lambda x.e, \sigma' \rangle] \vdash e_2 \Rightarrow v}{\sigma \vdash \texttt{let rec}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow v}$$

[IfT] $$\frac{\sigma \vdash e_1 \Rightarrow \texttt{true} \qquad \sigma \vdash e_2 \Rightarrow v}{\sigma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}$$

[IfT] $$\frac{\sigma \vdash e_1 \Rightarrow \texttt{false} \qquad \sigma \vdash e_3 \Rightarrow v}{\sigma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}$$

[Op] $$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad op(v_1, v_2) = v}{\sigma \vdash e_1\, op\, e_2 \Rightarrow v}$$

[Read] $$\sigma \vdash \texttt{read} \Rightarrow n$$

[Write] $$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash \texttt{write}\ e \Rightarrow v}$$

[Paren] $$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash \texttt{(}e\texttt{)} \Rightarrow v}$$

3 Static Semantics: Type System

위의 장에서는 tcoML 프로그램이 실행된다는 것이 무엇을 뜻하는 지를 정확하게 정의하였다. 위의 규칙들을 이용해서 주어진 프로그램 e가 어떤 값 v에 대해서

$$\emptyset \vdash e \Rightarrow v$$

의 관계를 가진다는 것이 증명되면 "프로그램 e는 값 v를 계산해 낸다"고 한다. 그러한 관계가 증명되는 과정(proof tree 혹은 inference tree)은 프로그램이 실행되는 과정에 그대로 대응된다.

이 장에서는 프로그램이 실행되는 과정을 **미리** 검증해 볼 수 있는 규칙들을 생각해 본다. 이러한 규칙에 의해서 주어진 프로그램이 검증될 때에만 우리는 그 프로그램을 목적코드로 컴파일하고 싶은것이다. 컴파일러는 이제 문법이 맞는 프로그램뿐 아니라 의미있는 (문제없이 실행될 수 있는) 프로그램만 컴파일해야 하는 것이다.

이러한 규칙이 갖추어야 할 조건을 정리해보자:

- 검증된 프로그램은 실행중에 잘못되지 않는다. 즉, 임의의 프로그램 e에 대해서 검증이 성공하면 $\emptyset \vdash e \Rightarrow v$를 만족하는 v가 존재한다.
- 검증은 항상 유한시간내에 마쳐져야 한다.
- 가능한한 많은 의미있는 프로그램들이 검증될 수 있어야 한다.

이러한 규칙을 만드는 가장 훌륭한 방법이 타입이론(type theory)으로 부터 나왔다. [1]

여러분들은 본 강의를 통해서 타입이론이 프로그래밍 언어의 디자인과 구현에 어떻게 어디까지 쓰이고 있고 미래가 어떻게 될 것인지를 부분적으로나마 엿보게 될 것이다. 미리 몇 마디 덧붙이면, "첨단 프로그래밍 이론과 실제는 타입이론의 주석(footnote)에 불과한 것이고 현재의 첨단은 항상 고전을 정확하고 철저하게 이해하는 사람들에 의해서만 만들어진다는 사실을 여러분들은 깊이 이해하기 바란다."

---

[1] 타입이론은 1908년 B. Russel 이 집합에서의 모순(Russel's paradox)을 피하는 방안으로 고안된 것을 비롯해서, A. Church에 의해 1940년도에 Lambda Calculus에 적용되서 "typed lambda terms always reduce into their normal forms"라는 정리(Normalization Theorem)가 만들어 진것이 프로그래밍에 적용되는 다리가 되었다. 타입이론이 프로그래밍에 적용되는 역사를 부분적으로 일별하면 다음과 같다: simply-typed lambda calculus (Church 1940), type inference (Hindley 1969), plymorphic lambda calculus (Reynolds 1972), subtypes (Reynolds 1980), existential abstract types (Mitchell and Plotkin 1985), dependent type (MacQueen 1986), linear type (Girard 1987), SML (Milner 1990), proof-carrying code (Necula and Lee 1997), typed assembly language (Morriset 1998), etc. 각각의 수박은 수업시간에 핥는다.

우리의 tcoML에서 타입은 상당히 단순하다:

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\textit{Type}} \\
\tau & ::= & i & \text{integer type} \\
 & | & b & \text{boolean type} \\
 & | & s & \text{string type} \\
 & | & \tau \to \tau & \text{function type}
\end{array}
$$

타입규칙은 다음의 관계를 결정해주는 규칙들이다:

$$\Gamma \vdash e : \tau$$

위의 관계를 다음과 같이 읽자 "프로그램 식 e는 Γ라는 환경에서 타입 τ를 가진다." 타입 환경 Γ는 다음과 같은 테이블이다:

$$\Gamma \in \textit{TypeEnv} = \textit{Id} \xrightarrow{\text{fin}} \textit{Type} \qquad \text{type environment}$$

Definition 2 (Program's Type) *A program e has type τ iff relation $\emptyset \vdash e : \tau$ is proved. If there is no such τ, then the expression has no type.*

다음이 프로그램 식 e 의 타입을 결정하는 규칙들이다:

[Num] $$\Gamma \vdash n : i$$

[Bool] $$\Gamma \vdash \texttt{true} : b \qquad \Gamma \vdash \texttt{false} : b$$

[String] $$\Gamma \vdash \mathit{string} : s$$

[Fun] $$\frac{\Gamma[x:\tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e : \tau_1 \to \tau_2}$$

[App] $$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

[Let] $$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x:\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2}$$

[RecLet] $$\frac{\Gamma[x:\tau_1] \vdash e_1 : \tau_1 \qquad \Gamma[x:\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2}$$

[If] $$\frac{\Gamma \vdash e_1 : b \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

[Op] $$\frac{\Gamma \vdash e_1 : i/b \qquad \Gamma \vdash e_2 : i/b}{\Gamma \vdash e_1\ \mathit{op}\ e_2 : i/b}$$

[Read] $$\Gamma \vdash \texttt{read} : i$$

[Write] $$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{write}\ e : \tau}$$

[Paren] $$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{(}e\texttt{)} : \tau}$$

3.1 Type Soundness

타입 규칙이 갖추어야 할 조건중 첫째 것은, 검증된 프로그램은 실행중에 잘못되지 않는다는 것이다. 이것을 타입규칙의 안전성이라고 한다. 우리가 정의한 타입시스템이 안전하다는 것을 증명하자.

안전성 증명을 용이하게 하기 위해서 프로그램이 실행되는 매 스텝을 다음의 관계로 정의한다:

$$e \longmapsto e'$$

위의 관계는 "프로그램 e가 한 스텝 후에 프로그램 e'로 변한다"로 읽는다. 예를 들어

$$\texttt{1 + 2 * 3} \longmapsto \texttt{1 + 6} \longmapsto \texttt{7}$$

과 같이 식이 변해가면서 최종 값을 만들어내는 과정을 표현하는 것이다. 프로그램 식이 변하는 까닭은 프로그램의 부분식들이 계산되는 순서에 따라 값으로 변하기 때문이다.

프로그램의 부분식들이 계산되는 순서를 결정하는 방법은 *실행의 맥*(evaluation context)이라는 기발한 방법으로 가능하다:

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\text{실행의 맥 evaluation context}} \\
C & ::= & [] & \text{빈칸} \\
 & | & C\ e & \\
 & | & v\ C & \\
 & | & \texttt{if}\ C\ \texttt{then}\ e\ \texttt{else}\ e & \\
 & | & \texttt{let}\ x\ \texttt{=}\ C\ \texttt{in}\ e & \\
 & | & C\ op\ e & \\
 & | & v\ op\ C & \\
 & | & \texttt{write}\ C & \\
 & | & \texttt{(}C\texttt{)} &
\end{array}
$$

$C[]$는 빈칸이 있는 프로그램 식을 표현한다. 위의 C의 정의에 의하면 $C[]$에서 빈칸은 오직 하나뿐이다. $C[e]$ 는 그 빈칸에 식 e가 든 것을 표현한다. 위에서 v는 값을 프로그램 식으로 표현한 것이다:

$$
\begin{array}{lcll}
v & ::= & const & \text{constant value} \\
 & | & \texttt{fn}\ x\ \texttt{=>}\ e & \text{function value} \\
 & | & f\ \texttt{fn}\ x\ \texttt{=>}\ e & \text{recursive function value}
\end{array}
$$

위에서 재귀함수 값 "f `fn` x `=>` e"는 다음의 식을 간단하게 표현한 것이다:

$$\texttt{let rec}\ f\ \texttt{=}\ \texttt{fn}\ x\ \texttt{=>}\ e\ \texttt{in}\ f\ \texttt{end}.$$

어느정도 눈치챘겠지만, C에서 빈칸이 있을 수 있는 자리가 바로 다음에 수행해야 할 식이 위치한 자리가 된다. 값이 계산된 장소(v가 있는 장소)에는

빈칸([])이 있을 수 없고, 계산순서가 v와 C의 적절한 위치를 통해서 표현되고 있다. 예를 들어, "$C\,e$"는 함수식 파트가 우선 계산되야 한다는 것을 표현하고 있고, "$v\,C$"에서는 인자 파트가 계산되는 경우는 항상 함수 파트가 값 v로 계산된 다음이다 라는 것을 표현한다. 위에서 let-rec binding에서는 C가 있

$$\texttt{let rec } x \texttt{ = } C \texttt{ in } e$$

와 같이

위에서 정의한 프로그램 실행의 한 스텝 $e \longmapsto e'$ 은 다음의 규칙에 의해서 정의된다:

$$\frac{r \to c}{C[r] \longmapsto C[c]}$$

즉, e에서 바로 다음에 계산해야 할 식이 r이고 ($e = C[r]$) 그 r이 c가 될 때 ($r \to c$) e는 e' ($e' = C[c]$) 으로 한 스텝 진행하는 것이다. 규칙 $r \to c$는 다음과 같다:

$$\begin{array}{rcll}
(\texttt{fn } x \texttt{ => } e)\,v & \to & e \,/\, \{v/x\} & \\
(f \texttt{ fn } x \texttt{ => } e)\,v & \to & e \,/\, \{v/x, (f \texttt{ fn } x \texttt{ => } e)/f\} & \\
\texttt{let } x \texttt{ = } v \texttt{ in } e & \to & e \,/\, \{v/x\} & \\
\texttt{let rec } x \texttt{ = } v \texttt{ in } e & \to & e \,/\, \{x\,v/x\} & \\
\texttt{if true then } e_1 \texttt{ else } e_2 & \to & e_1 & \\
\texttt{if false then } e_1 \texttt{ else } e_2 & \to & e_2 & \\
v_1 \mathit{op}\, v_2 & \to & v & \text{where } v = \mathit{op}(v_1, v_2) \\
\texttt{write } v & \to & v & \\
(v) & \to & v &
\end{array}$$

Definition 3 (Program's Semantics) *A program e computes v iff $e \overset{*}{\longmapsto} v$.*

Fact 1 $\emptyset \vdash e \Rightarrow v$ *iff* $e \overset{*}{\longmapsto} v$.

Definition 4 (Program Goes Wrong) *If program e stucks at e' because no value v exists such that $e' \overset{*}{\longmapsto} v$, then the program e goes wrong, writing $e \overset{*}{\longmapsto}$ wrong.*

Theorem 1 (Type's Weak Soundness) *For any program e of tcoML, if $\emptyset \vdash e : \tau$ then not $e \overset{*}{\longmapsto}$ wrong.*

Theorem 2 (Type's Strong Soundness) *For any program e of tcoML, if $\emptyset \vdash e : \tau$ and $e \overset{*}{\longmapsto} \bullet$ then $\emptyset \vdash \bullet : \tau$.*

Proof. We prove that if $\emptyset \vdash e : \tau$ and $e \longmapsto e'$ then $\emptyset \vdash e' : \tau$. Then by induction on the number of evaluation steps, the theorem holds. The proof is by induction on the evaluation context C of $C[r] = e$ and by using Lemma 2.

Base case: $C[] = []$. True by Lemma 2.

Inductive case: $C[] = C_1[]\,e'$. By the assumption $\emptyset \vdash C_1[r]\,e' : \tau$ and by rule [App],

$$\begin{array}{rl} \emptyset \vdash C_1[r] : \tau' \to \tau & (0) \\ \emptyset \vdash e' : \tau' & (1) \end{array}$$

By induction hypothesis for (0),

$$\emptyset \vdash C_1[c] : \tau' \to \tau.$$

Hence by (1) and rule [App]

$$\emptyset \vdash C_1[c]\,e' : \tau.$$

Other cases are similarly done. □

Note that the strong soundness implies the weak soundness because the *wrong* has no type.

Lemma 1 *If* $\Gamma[x : \tau_1] \vdash e : \tau$ *and* $\Gamma \vdash v_1 : \tau_1$ *then* $\Gamma \vdash e \,/\, \{v_1/x\} : \tau$.

Proof. By induction on e. This Lemma is necessary to prove Lemma 2. □

Lemma 2 *If* $\emptyset \vdash r : \tau$ *and* $r \to c$ *then* $\emptyset \vdash c : \tau$.

Proof. By case analysis on r.

Case $r = (f\ \texttt{fn}\ x\ \texttt{=>}\ e)\,v$. Because $\emptyset \vdash r : \tau$,

$$\begin{array}{rl} \emptyset \vdash f\ \texttt{fn}\ x\ \texttt{=>}\ e : \tau' \to \tau & (0) \\ \emptyset \vdash v : \tau'. & (1) \end{array}$$

By (0) and rule [RecLet],

$$\{f : \tau' \to \tau, x : \tau'\} \vdash e : \tau. \qquad (2)$$

By (1), (2), and Lemma 1

$$f : \tau' \to \tau \vdash e/\{v/x\} : \tau$$

and by (0) and Lemma 1

$$\emptyset \vdash e/\{v/x, f\ \texttt{fn}\ x\ \texttt{=>}\ e/f\} : \tau.$$

Other cases are similarly done. □

4 Type Checking

After we made sure that our type system was sound, we have to implement the type system. That is, we need a program `typecheck` such that

$$\texttt{typecheck}(e) = \tau \quad \text{iff} \quad \emptyset \vdash e : \tau.$$

Determining for given a program e whether $\emptyset \vdash e : \tau$ or not is reduced to the provability in a simple logical system called *unification logic*.

The unification logic formulas for type terms t are:

$$\begin{array}{lcll}
\multicolumn{4}{l}{\textit{Formula}} \\
\phi & ::= & t \doteq t & \\
& | & \phi \wedge \phi & \\
& | & \exists \alpha . \phi & \\
& & & \\
\multicolumn{4}{l}{\textit{TyTerm}} \\
t & ::= & i & \text{int type} \\
& | & b & \text{bool type} \\
& | & s & \text{string type} \\
& | & t \to t & \text{function type} \\
& | & \alpha & \text{type variable}
\end{array}$$

The proof rules for the unification logic formulas are:

$$\vdash t \doteq t \qquad \frac{\vdash \phi/\{t/\alpha\}}{\vdash \exists \alpha . \phi} \qquad \frac{\vdash \phi_1 \quad \vdash \phi_2}{\vdash \phi_1 \wedge \phi_2}$$

Therefore, a solution of a set of unification logic formulas is a substitution of type terms for type variables that will satisfy all the $\doteq$. The most general such subsitution is achieved by Robinson's most general unifier (mgu).

We define an algorithm T that collects unification formulas:

$$T : \textit{Expr} \times \textit{TyEnv} \times \textit{TyTerm} \to \textit{Formula}$$

such that

$$\vdash T(e, \emptyset, \tau) \quad \text{iff} \quad \emptyset \vdash e : \tau.$$

$$
\begin{array}{rcl}
T(n,\Gamma,t) & = & t \doteq i \\
T(s,\Gamma,t) & = & t \doteq s \\
T(b,\Gamma,t) & = & t \doteq b \\
T(x,\Gamma,t) & = & t \doteq \Gamma(x) \\
T(\texttt{fn}\ x\ \texttt{=>}\ e,\Gamma,t) & = & \exists\alpha.\exists\beta.\, t \doteq \alpha \to \beta\ \wedge\ T(e,\Gamma[x:\alpha],\beta) \\
T(e_1\ e_2,\Gamma,t) & = & T(e_1,\Gamma,\alpha \to t)\ \wedge\ T(e_2,\Gamma,\alpha) \\
T(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\Gamma,t) & = & \exists\alpha.\, T(e_1,\Gamma,\alpha)\ \wedge\ T(e_2,\Gamma[x:\alpha],t) \\
T(\texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\Gamma,t) & = & \exists\alpha.\, T(e_1,\Gamma[x:\alpha],\alpha)\ \wedge\ T(e_2,\Gamma[x:\alpha],t) \\
T(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\Gamma,t) & = & \exists\alpha.\, T(e_1,\Gamma,b)\ \wedge\ T(e_2,\Gamma,\alpha)\ \wedge\ T(e_3,\Gamma,\alpha) \\
T(e_1\ \texttt{+}\ e_2,\Gamma,t) & = & T(e_1,\Gamma,i)\ \wedge\ T(e_2,\Gamma,i)
\end{array}
$$

Theorem 3 $\vdash T(e,\Gamma,\tau)$ *iff* $\Gamma \vdash e : \tau$.

Proof. By induction on e.

Case $e = \texttt{fn}\ x\ \texttt{=>}\ e'$:

$$
\begin{array}{lll}
 & \vdash T(\texttt{fn}\ x\ \texttt{=>}\ e',\Gamma,\tau) & \\
\text{iff} & \tau = \tau_1 \to \tau_2 & \text{and} \\
 & \vdash T(e',\Gamma[x:\tau_1],\tau_2) & \\
\text{iff} & \Gamma[x:\tau_1] \vdash e' : \tau_2 & \text{by I.H.} \\
\text{iff} & \Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e' : \tau_1 \to \tau_2 & \text{by rule [Fun]}
\end{array}
$$

□

Our type checking algorithm `typecheck` can be implemented as follows

$$\texttt{typecheck}(e) = (mgu \circ T)(e,\emptyset,\alpha)(\alpha)$$

If the unification fails the program is ill-typed.

One drawback of the above implementation inside a compiler is that it is hard to report the region of the input program that causes, if any, a type error.

A rather more direct-style implementation is the following algorithm:

$$W : \mathit{Expr} \times \mathit{TyEnv} \to \mathit{TyTerm} \times \mathit{Subst}$$

$$
\begin{array}{rcl}
W(n,\Gamma) & = & (i,\{\}) \\
W(x,\Gamma) & = & (\Gamma(x),\{\}) \\
W(\texttt{fn}\ x\ \texttt{=>}\ e,\Gamma) & = & \text{let } (t,S) = W(e,\Gamma[x:\alpha]) \quad \alpha \text{ new} \\
 & & \text{in } (\alpha/S \to t, S) \\
W(e_1\, e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma) \\
 & & \quad\ (t_2,S_2) = W(e_2,\Gamma/S_1) \\
 & & \quad\ S_3 = mgu(t_1/S_2, t_2 \to \alpha) \quad \alpha \text{ new} \\
 & & \text{in } (\alpha/S_3, S_3S_2S_1) \\
W(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma) \\
 & & \quad\ (t_2,S_2) = W(e_2,\Gamma[x:t_1]/S_1) \\
 & & \text{in } (t_2, S_2S_1) \\
W(\texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma[x:\alpha]) \quad \alpha \text{ new} \\
 & & \quad\ S_2 = mgu(t_1, \alpha/S_1) \\
 & & \quad\ (t_2,S_3) = W(e_2,\Gamma[x:\alpha]/(S_2S_1)) \\
 & & \text{in } (t_2, S_3S_2S_1) \\
W(e_1\texttt{+}e_2,\Gamma) & = & \text{let } (t_1,S_1) = W(e_1,\Gamma) \\
 & & \quad\ (t_2,S_2) = W(e_2,\Gamma/S_1) \\
 & & \quad\ S_3 = mgu(t_1/S_2, t_2, i) \\
 & & \text{in } (i, S_3S_2S_1)
\end{array}
$$

Now it becomes clear where the type-error occurs: where the unfication mgu fails.

Theorem 4 (W's Soundness) *If* $W(e,\Gamma) = (\tau, S)$ *then* $\Gamma/S \vdash e : \tau/S$.

Proof. By induction on e. □

Theorem 5 (W's Completeness) *If* $\Gamma/P \vdash e : \tau/P$ *then* $W(e,\Gamma) = (\tau, S)$ *and* $P = TS$ *for some* T.

Proof. By induction on e. □

5 Intermediate Language: SECK Machine

$$
\begin{array}{rcll}
Cmd & ::= & \texttt{FRAME} \mid \texttt{DEFRAME} & \\
 & \mid & \texttt{FETCH(}n\texttt{)} & \\
 & \mid & \texttt{QUOTE(}p\texttt{)} & \\
 & \mid & \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \mid \texttt{EQ} \mid \texttt{LT} \mid \texttt{GT} & \\
 & \mid & \texttt{READ} \mid \texttt{WRITE} & \\
 & \mid & \texttt{COND(}Cmd^{+}\texttt{,}Cmd^{+}\texttt{)} & \\
 & \mid & \texttt{FTN(}Cmd^{+}\texttt{)} & \\
 & \mid & \texttt{APP} & \\
 & \mid & \texttt{RETURN} & \\
 & \mid & \texttt{UNROLL} & \\
p & ::= & n & \text{integer} \\
 & \mid & \mathit{true} \mid \mathit{false} & \text{boolean} \\
 & \mid & s & \text{string}
\end{array}
$$

SECK machine's state is a tuple

$$(S, E, C, K)$$

where

$$
\begin{array}{rcll}
\text{stack} & S & \in & \mathit{Stk} = \mathit{Value}\ \text{list} \\
\text{environment} & E & \in & \mathit{Env} = \mathit{Value}\ \text{list} \\
\text{code} & C & \in & \mathit{Code} = \mathit{Cmd}\ \text{list} \\
\text{continuation} & K & \in & \mathit{Cont} = (\mathit{Env} \times \mathit{Code})\ \text{list} \\
\text{value} & v & \in & \mathit{Value} = \mathit{Int} \cup \mathit{Bool} \cup \mathit{String} \cup \mathit{Code} \times \mathit{Env}
\end{array}
$$

SECK machine's execution is defined as the sequence of machine states linked by the following $\rightarrow$ transition:

$$
\begin{array}{rcl}
(v.S,\ E,\ \texttt{FRAME}.C,\ K) & \rightarrow & (S,\ v.E,\ C,\ K) \\
(S,\ v.E,\ \texttt{DEFRAME}.C,\ K) & \rightarrow & (S,\ E,\ C,\ K) \\
(S,\ E \text{ as } v_0.\cdots.v_n.E',\ \texttt{FETCH(}n\texttt{)}.C,\ K) & \rightarrow & (v_n.S,\ E,\ C,\ K) \\
(S,\ E,\ \texttt{QUOTE(}v\texttt{)}.C,\ K) & \rightarrow & (v.S,\ E,\ C,\ K) \\
(v_2.v_1.S,\ E,\ \texttt{ADD}.C,\ K) & \rightarrow & (v_1 + v_2.S,\ E,\ C,\ K) \\
(S,\ E,\ \texttt{READ}.C,\ K) & \rightarrow & (\mathit{get}().S,\ E,\ C,\ K) \\
(v.S,\ E,\ \texttt{WRITE}.C,\ K) & \rightarrow & (v.S,\ E,\ C,\ K) \\
(\mathit{true}.S,\ E,\ \texttt{COND(}C_1\texttt{,}C_2\texttt{)}.C,\ K) & \rightarrow & (S,\ E,\ C_1.C,\ K) \\
(\mathit{false}.S,\ E,\ \texttt{COND(}C_1\texttt{,}C_2\texttt{)}.C,\ K) & \rightarrow & (S,\ E,\ C_2.C,\ K) \\
(S,\ E,\ \texttt{FTN(}C'\texttt{)}.C,\ K) & \rightarrow & ((C',\ E).S,\ E,\ C,\ K) \\
(v.(C',\ E').S,\ E,\ \texttt{APP}.C,\ K) & \rightarrow & (S,\ v.E',\ C',\ (E,\ C).K) \\
(S,\ E,\ \texttt{RETURN},\ (E',\ C).K) & \rightarrow & (S,\ E',\ C,\ K) \\
(S,\ v.E',\ \texttt{UNROLL}.C,\ K) & \rightarrow & (S,\ v.(\texttt{UNROLL}.C,\ E').E',\ C,\ K)
\end{array}
$$

5.1 Translation from tcoML to SECK

Each expression e of lexical depth ℓ is translated into SECK code C. We write this translation relation as

$$L, \ell, e \;\triangleright\; C.$$

The relation says that

- tcoML expression e is at lexical depth ℓ,
- e's free variable x's lexical depth is $L(x)$, and
- tcoML expression e is translated into SECK machine code C.

The generated SECK code C for tcoML expression e satisfies the invariant that the SECK machine, after executing the C and right before continuation, has the expression e's value on top of its stack S. The L is a table from identifiers to their lexical depths.

Rules for determining this translation relation are in Figure 1. For tcoML program e, its $SECK$ code is C such that

$$\emptyset, 0, e \;\triangleright\; C.$$

5.1.1 Correctness of the Translation

Theorem 6 *For tcoML program e, let $SECK$ code C be such that $\emptyset, 0, e \;\triangleright\; C$. Then $\emptyset \;\vdash\; e \;\Rightarrow\; v$ iff $([], [], C, []) \stackrel{*}{\to} (v, [], [], [])$.*

Let $L, \ell, e \;\triangleright\; C_e$ occurs during $\emptyset, 0, p \;\triangleright\; C$. Then followings are true:

Fact 2 *Expression e's lexical depth ℓ is the number of frames in E right before executing the code C_e.*

Fact 3 *The ℓ frames are those that were pushed in by function calls or entering let-blocks.*

Fact 4 *Calling a function (`fn x =>` $\cdots$) or entering a let-block (`let` [`rec`] `x =` $\cdots$) pushes one frame for the bound variable x.*

Fact 5 *Variable x's lexical depth $L(x)$ is the index of its frame in the current E.*

$$[\mathrm{C}_\triangleright] \quad L, \ell, \mathit{const} \ \triangleright\ \texttt{QUOTE(}\mathit{const}\texttt{)}$$

$$[\mathrm{X}_\triangleright] \quad L, \ell, x \ \triangleright\ \texttt{FETCH(}\ell - L(x)\texttt{)}$$

$$[\mathrm{FN}_\triangleright] \quad \frac{L[x \mapsto \ell+1], \ell+1, e \ \triangleright\ C}{L, \ell, \texttt{fn}\ x\ \texttt{=>}\ e \ \triangleright\ \texttt{FTN(}C\texttt{.RETURN)}}$$

$$[\mathrm{APP}_\triangleright] \quad \frac{L, \ell, e_1 \ \triangleright\ C_1 \qquad L, \ell, e_2 \ \triangleright\ C_2}{L, \ell, e_1\ e_2 \ \triangleright\ C_1.C_2.\texttt{APP}}$$

$$[\mathrm{LET}_\triangleright] \quad \frac{L, \ell, e_1 \ \triangleright\ C_1 \qquad L[x \mapsto \ell+1], \ell+1, e_2 \ \triangleright\ C_2}{L, \ell, \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \ \triangleright\ C_1.\texttt{FRAME}.C_2.\texttt{DEFRAME}}$$

$$[\mathrm{RLET}_\triangleright] \quad \frac{L[f \mapsto \ell+1][x \mapsto \ell+2], \ell+2, e_1 \ \triangleright\ C_1 \qquad L[f \mapsto \ell+1], \ell+1, e_2 \ \triangleright\ C_2}{L, \ell, \texttt{let rec}\ f\ \texttt{=}\ \texttt{fn}\ x\ \texttt{=>}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \ \triangleright\ \begin{array}{l}\texttt{FTN(UNROLL.}C_1\texttt{.RETURN).}\\ \texttt{FRAME.}C_2\texttt{.DEFRAME}\end{array}}$$

$$[\mathrm{IF}_\triangleright] \quad \frac{L, \ell, e_i \ \triangleright\ C_i}{L, \ell, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \ \triangleright\ C_1.\texttt{COND(}C_2\texttt{,}\ C_3\texttt{)}}$$

$$[\mathrm{ADD}_\triangleright] \quad \frac{L, \ell, e_i \ \triangleright\ C_i}{L, \ell, e_1\ \texttt{+}\ e_2 \ \triangleright\ C_1.C_2.\texttt{ADD}}$$

$$[\mathrm{OR}_\triangleright] \quad \frac{L, \ell, e_i \ \triangleright\ C_i}{L, \ell, e_1\ \texttt{or}\ e_2 \ \triangleright\ C_1.\texttt{COND(QUOTE(}\mathit{true}\texttt{),}C_2\texttt{)}}$$

$$[\mathrm{AND}_\triangleright] \quad \frac{L, \ell, e_i \ \triangleright\ C_i}{L, \ell, e_1\ \texttt{and}\ e_2 \ \triangleright\ C_1.\texttt{COND(}C_2\texttt{,QUOTE(}\mathit{false}\texttt{))}}$$

$$[\mathrm{IN}_\triangleright] \quad L, \ell, \texttt{read} \ \triangleright\ \texttt{READ}$$

$$[\mathrm{OUT}_\triangleright] \quad \frac{L, \ell, e \ \triangleright\ C_1}{L, \ell, \texttt{write}\ e \ \triangleright\ C_1.\texttt{WRITE}}$$

그림 1: Translation from tcoML to SECK

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

`let x =` e_2 `in` e

로 바꾸어서 경제적인 코드로 변환시킬 수 있다. 단, 이때 위의 함수 **fn x =>** e 가 정의될 때의 환경은 호출되는 현재의 환경과 같거나 $fv(e) = \{\mathtt{x}\}$ 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)	$T(e_2) + 1 + T(e) + 1$
$M(e_1) + \lvert\mathtt{E}\rvert + M(e_2) + M(e)$	(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

`if` e_1 `then` e_2 `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.

5.3 Compile-time Analysis of the Run-Time Values

5.3.1 A Source Transformation: Our Goal

분석을 얘기하기 전에, 분석된 결과를 가지고 우리가 목표로하는 프로그램 변환과정을 정확하게 정의하도록 하자.

tcoML 프로그램 $\wp$ 가 주어졌을 때 $\mathcal{A}_\wp$ 는 프로그램 $\wp$ 에 나타나는 모든 함수식들이 실행중에 가질 수 있는 함수들을 예측한 결과라고 하자. 예를들어 다음의 식이 프로그램에 있다면

```
let
   rec f = fn x => e
in
   ··· (f e') ···
end
```

위의 (f e') 에서 함수식 f가 가지는 함수는 위에서 f라는 이름으로 정의된 재귀함수가 된다.

다음의 예를 보면

```
let
  g = fn f => f 10
in
  ···
end
```

위에서 f 10 에서 함수식 f가 가지는 함수는 g가 호출될 때 인자로 전달되는 함수들이 된다.

따라서 $\mathcal{A}_\wp$를 다음의 테이블이라고 할 수 있다:

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\mathcal{A}_\wp : \mathit{Expr} \to 2^{\mathit{Fexpr}}} \\
\mathit{Fexpr} & ::= & \lambda x.e & \text{functions in } \wp \\
 & | & f\lambda x.e & \text{rec functions in } \wp
\end{array}
$$

분석결과 $\mathcal{A}_\wp$를 가지고 프로그램 $\wp$에 있는 함수 호출식들을 Figure 2의 규칙을 이용해서 좀더 경제적인 식들로 변환할 수 있다[2]:

$$e \xrightarrow{\text{rewrite}} e' \xrightarrow{\text{rewrite}} e'' \xrightarrow{\text{rewrite}} \cdots \xrightarrow{\text{rewrite}} e^*.$$

[2]변환규칙을 적용해가면 유한번만에 끝나는지를 확인해보라.

$$\frac{\mathcal{A}_e(e_1) = \{\lambda x.e\} \qquad fv(e) = \{x\}}{e_1\ e_2 \stackrel{\text{rewrite}}{\longrightarrow} \texttt{let x = } e_2 \texttt{ in } e \texttt{ end}}$$

$$\frac{\mathit{Pure}(e_1)}{\texttt{let x = } e_1 \texttt{ in } e_2 \texttt{ end} \stackrel{\text{rewrite}}{\longrightarrow} e_2/\{e_1/\texttt{x}\}}$$

$$\mathit{Pure}(c)$$

$$\frac{\forall(\lambda y.e \in \mathcal{A}_e(x)).\mathit{Pure}(e) \qquad \forall((f\ \lambda y.e) \in \mathcal{A}_e(x)).\mathit{Pure}(f) \Rightarrow \mathit{Pure}(e)}{\mathit{Pure}(x)}$$

$$\frac{\mathit{Pure}(e)}{\mathit{Pure}(\texttt{fn x => } e)}$$

$$\frac{\mathit{Pure}(e_1) \qquad \mathit{Pure}(e_2)}{\mathit{Pure}(\texttt{let x = } e_1 \texttt{ in } e_2 \texttt{ end})}$$

$$\frac{\mathit{Pure}(f) \Rightarrow \mathit{Pure}(e_1) \qquad \mathit{Pure}(e_2)}{\mathit{Pure}(\texttt{let rec f = fn x => } e_1 \texttt{ in } e_2 \texttt{ end})}$$

$$\frac{\mathit{Pure}(e_1) \qquad \mathit{Pure}(e_2)}{\mathit{Pure}(e_1\ e_2)}$$

$$\frac{\mathit{Pure}(e_1) \qquad \mathit{Pure}(e_2) \qquad \mathit{Pure}(e_3)}{\mathit{Pure}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)}$$

$$\frac{\mathit{Pure}(e_1) \qquad \mathit{Pure}(e_2)}{\mathit{Pure}(e_1\texttt{+}e_2)}$$

$$\frac{e \stackrel{\text{rewrite}}{\longrightarrow} e'}{E[e] \stackrel{\text{rewrite}}{\longrightarrow} E[e']}$$

그림 2: tcoML 프로그램 변환 규칙

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.

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

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

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

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

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

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

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

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

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

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

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

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 $F\colon A\to A$ 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 컴퓨팅 환경에서 컴파일러와 코드사용자의 역할을 정리한 것이다:

Code provider (compiler)	Code consumer (user)
tcoML 검증	tySECK 검증
tcoML $\rightarrow$ tySECK	tySECK $\rightarrow$ k86
tySECK 송신	SECK 혹은 k86 실행

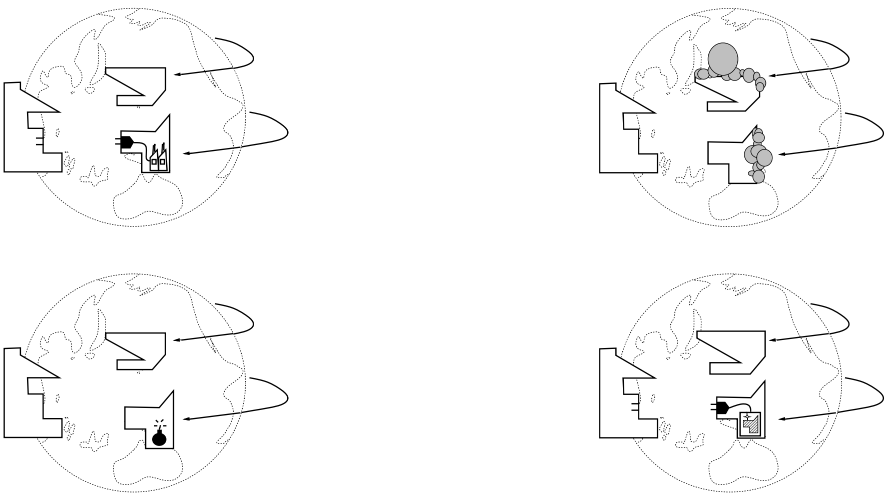

그림 4: 전 지구적 소프트웨어 부품의 문제에 대한 개념도

6.1 tySECK Syntax and Semantics

tySECK command is identical to SECK command except that first, `FTN` and `UNROLL` command has the type of its corresponding function and second, the `UNROLL` has as a sub-command the body code of the function to unroll:

$$
\begin{array}{rcl}
\mathit{tyCmd} & ::= & \texttt{FRAME} \mid \texttt{DEFRAME} \\
& \mid & \texttt{FETCH(}n\texttt{)} \\
& \mid & \texttt{QUOTE(}p\texttt{)} \\
& \mid & \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \mid \texttt{EQ} \mid \texttt{LT} \mid \texttt{GT} \\
& \mid & \texttt{READ} \mid \texttt{WRITE} \\
& \mid & \texttt{COND(}\mathit{tyCmd}\texttt{,}\mathit{tyCmd}\texttt{)} \\
& \mid & \texttt{FTN(}\tau_1 \to \tau_2\texttt{, }\ \mathit{tyCmd}\texttt{)} \\
& \mid & \texttt{FTN(}\mathit{rec}\,\tau_1 \to \tau_2\texttt{,}\mathit{tyCmd}\texttt{)} \\
& \mid & \texttt{APP} \\
& \mid & \texttt{RETURN} \\
& \mid & \texttt{UNROLL(}\mathit{rec}\,\tau_1 \to \tau_2\texttt{,}\mathit{tyCmd}\texttt{)} \\
& \mid & \mathit{tyCmd}^+ \\
n & ::= & \text{non-negative integer} \\
p & ::= & \text{integer, boolean, or string}
\end{array}
$$

tySECK 코드 수행중에 만드는 값들의 타입 τ는 다음과 같다:

$$
\begin{array}{rcll}
\tau & ::= & i \mid b \mid r & \text{int, bool, string type} \\
& \mid & \tau \to \tau & \text{function type} \\
& \mid & \mathit{rec}\,\tau \to \tau & \text{recursive function type}
\end{array}
$$

tySECK machine's transition semantics is identical to the SECK cases except:

$$
\begin{array}{rcl}
(S,\ E,\ \texttt{FTN(}\tau_1 \to \tau_2, C'\texttt{)}.C,\ K) & \to & ((C',\ E).S,\ E,\ C,\ K) \\
(S,\ E,\ \texttt{FTN(}\mathit{rec}\,\tau_1 \to \tau_2, C'\texttt{)}.C,\ K) & \to & ((C',\ E).S,\ E,\ C,\ K) \\
(S,\ v.E',\ \underbrace{\texttt{UNROLL(}\mathit{rec}\,\tau_1 \to \tau_2\texttt{, }\ C'\texttt{)}.C}_{C''},\ K) & \to & (S,\ v.(C'', E').E',\ C,\ K)
\end{array}
$$

6.2 tcoML-to-tySECK translation

The tcoML-to-tySECK translation is the same as before except that first, we add types for function codes and second, the `UNROLL` has as a sub-command the body code of the function to unroll:

$$[\text{RLET}_\triangleright] \quad \frac{\begin{array}{c} L[f \mapsto \ell+1], \ell+1, e_2 \ \triangleright\ C_2 \\ L[f \mapsto \ell+1][x \mapsto \ell+2], \ell+2, e_1 \ \triangleright\ C_1 \end{array}}{\begin{array}{l} L, \ell, \texttt{let rec } f : \tau_1 \to \tau_2 \texttt{ = fn } x \texttt{ => } e_1 \texttt{ in } e_2 \texttt{ end} \\ \triangleright \\ \texttt{FTN(UNROLL(}\mathit{rec}\,\tau_1 \to \tau_2, C_1\texttt{)}.C_1\texttt{.RETURN).FRAME.}C_2\texttt{.DEFRAME} \end{array}}$$

$$[\text{FN}_\triangleright] \quad \frac{L[x \mapsto \ell+1], \ell+1, e \ \triangleright\ C}{L, \ell, \texttt{fn } x \texttt{ => } e : \tau_1 \to \tau_2 \ \triangleright\ \texttt{FTN(}\tau_1 \to \tau_2, C\texttt{.RETURN)}}$$

6.3 tySECK's Type System

각 tySECK 명령어 C 의 타입은 다음의 관계로 결정한다:

$$s, e, k \vdash C \ : \ s', e', k'$$

위의 관계는 다음과 같이 읽자: "tySECK 의 기계상태 타입 s, e, k에서 *tyCmd* C 가 수행이 마쳐지면 새로운 기계상태 타입 s', e', k'가 된다." 여기서 s, e, k 는 각각 SECK 기계의 S (stack), E (environment), K (continuation)의 타입을 나타내고 다음과 같이 정의한다:

s	$::=$	ϵ	empty stack type
	$\mid$	$\tau.s$	stack type with τ on top
	$\mid$	α	type for arbitrary/abstract stack
e	$::=$	ϵ	empty environment type
	$\mid$	$\tau.e$	environment type with τ on top
k	$::=$	ϵ	empty continuation type
	$\mid$	$(e, C).k$	continuation type with e and *tyCmd* C on top
	$\mid$	β	type for arbitrary/abstract continuation

tyCmd 의 타입을 결정하는 규칙들이 Figure 5에 정의되어 있다.

Our tySECK's type system is sound:

Theorem 9 (Soundness) *Let* $s, e, k \vdash C \ : \ s', e', k'$.
For all $S, E, K \sim s, e, k$*:*

- *(Progress) transitions from* (S, E, C, K) *are either always possible or finish with* (S', E', nil, K').
- *(Subject Reduction) if the transitions finish* $(S, E, C, K) \xrightarrow{*} (S', E', nil, K')$ *then* $S', E', K' \sim s', e', k'$.

Proof. We prove the subject reduction part by the structure induction on C. (That is, the induction hypothesis is that the theorem holds for sub-commands of C.) The progress part can be similarly proved.

• Base case: $i.i.s, e, k \vdash \texttt{ADD} : i.s, e, k$. Let $v_1.v_2.S, E, K \sim i.i.s, e, k$. By definition,

$$(v_1.v_2.S, E, \texttt{ADD}, K) \rightarrow (v_1 + v_2.S, E, nil, K).$$

Clearly, $v_1 + v_2.S, E, K \sim i.s, e, k$. Other base cases are similarly straightforward.

• Case $b.s, e, k \vdash \texttt{COND(}C_1\texttt{, }C_2\texttt{)} : \tau.s, e, k$. Let $S, E, K \sim s, e, k$. By rule [IF] in Figure 5, $s, e, k \vdash C_1 : \tau.s, e, k$ holds. Hence by induction hypothesis, if $(S, E, C_1, K) \stackrel{*}{\rightarrow} (S', E', nil, K')$ then $S', E', K' \sim \tau.s, e, k$. Thus, the same holds if we start from $(true.S, E, \texttt{COND(}C_1\texttt{,}C_2\texttt{)}, K)$ because

$$(true.S, E, \texttt{COND(}C_1\texttt{,}C_2\texttt{)}, K) \rightarrow (S, E, C_1, K).$$

• Case $s, e, k \vdash \texttt{FN(}\tau_1 \rightarrow \tau_2\texttt{, }C\texttt{)} : \tau_1 \rightarrow \tau_2.s, e, k$. Let $S, E, K \sim s, e, k$. By definition,

$$(S, E, \texttt{FN(}\tau_1 \rightarrow \tau_2\texttt{, }C\texttt{)}, K) \rightarrow ((C, E).S, E, nil, K).$$

We need to show that $(C, E) \sim \tau_1 \rightarrow \tau_2$. By rule [FN] in Figure 5, $C = C'\texttt{.RETURN}$ and $\alpha, \tau_1.e, \beta \vdash C' : \tau_2.\alpha, \tau_1.e, \beta$. By induction hypothesis, for any $S \sim \alpha$, $K \sim \beta$, and $v_1 \sim \tau_1$, if $(S, v_1.E, C', K) \stackrel{*}{\rightarrow} (S', E', nil, K')$ then $S' \sim \tau_2.\alpha$, $E' \sim \tau_1.e$, and $K' \sim \beta$. That is, the conditions of rule [F1] holds, hence $(C, E) \sim \tau_1 \rightarrow \tau_2$.

• Case $\tau_1.(\tau_1 \rightarrow \tau_2).s, e, k \vdash \texttt{APP} : \tau_2.s, e, k$. Let $v_1.(C, E_0).S, E, K \sim \tau_1.(\tau_1 \rightarrow \tau_2).s, e, k$. By definition

$$(v_1.(C, E_0).S, E, \texttt{APP}, K) \rightarrow (S, v_1.E_0, C, (E, nil).K).$$

By rule [F1] in Figure 6, $C = C'.\texttt{RETURN}$ and if $(S, v_1.E_0, C', (E, nil).K) \stackrel{*}{\rightarrow} (S', E', nil, K')$ then $S' \sim \tau_2.s$, $E' \sim \tau_1.e$, $K' \sim (e, nil).k$ with $S, E, K \sim s, e, k$. Thus we can let $S' = v_2.S''$ and $K' = (E'', nil).K''$ with $v_2 \sim \tau_2$, $S'' \sim s$ and $K'' \sim k$. Hence if $(S, v_1.E_0, C'.\texttt{RETURN}, (E, nil).K)$ finishes, the state right before the terminal state is

$$(v_2.S'', E', \texttt{RETURN}, (E'', nil).K''),$$

which finishes in the next state

$$(v_2.S'', E'', nil, K'').$$

By definition, $v_2.S'' \sim \tau_2.s$, $E'' \sim e$, and $K'' \sim k$.

• Other cases are similary proved. □

6.4 Type Checking

Given a tySECK code C, we need to check if s, e, k exists such that

$$\epsilon, \epsilon, \epsilon \vdash C : s, e, k.$$

The tySECK code's type-checking is a straightforward implementation of the rules in Figure 5. For example, the type-checker

```
typeVerify:  stackTy * envTy * contTy -> code -> stackTy * envTy * contTy
```

would be defined as follows:[7]

```
exception SECKtyError
fun typeVerify (CONS(tau, s),e,k) FRAME = (s, CONS(tau,e), k)
  | typeVerify (CONS(INT, CONS(INT, s)), e, k) ADD = (CONS(INT,s), e, k)
  | typeVerify (CONS(BOOL,s), e, k) COND(C1, C2) =
      let
        val (tau::s', e', k') = typeVerify (s, e, k) C1
        val (tau'::s'', e'', k'') = typeVerify (s, e, k) C2
      in
        if equivT(tau,tau') andalso
           equivS(s, s') andalso equivS(s', s'') andalso
           equivE(e, e') andalso equivE(e', e'') andalso
           equivK(k, k') andalso equivK(k', k'')
        then (CONS(tau,s), e, k)
        else raise SECKtyError
      end
  | typeVerify (s, e, k) (FN(FNty(tau1,tau2), C)) =
      let
        val C'::RETURN = C
        val (tau2'::alpha', tau1'::e', beta') =
            typeVerify (alpha, tau1::e, beta) C'
      in
        if equivT(tau1,tau1') andalso equivT(tau2,tau2') andalso
           equivT(alpha,alpha') andalso equivT(beta,beta') andalso
           equivE(e,e')
        then (FNty(tau1,tau2)::s,e,k)
        else raise SECKtyError
  |
  ...
```

[7] You can use the unification as in type-checking the tcoML expressions.

[FR] $\tau.s, e, k \vdash \texttt{FRAME} : s, \tau.e, k \qquad s, \tau.e, k \vdash \texttt{DEFRAME} : s, e, k$

[OP] $i.i.s, e, k \vdash \texttt{ADD} : i.s, e, k \qquad b.b.s, e, k \vdash \texttt{AND} : b.s, e, k$

$i.i.s, e, k \vdash \texttt{EQ} : b.s, e, k \qquad b.b.s, e, k \vdash \texttt{EQ} : b.s, e, k$

[FE] $s, e \text{ as } \tau_0.\cdots.\tau_n.e', k \vdash \texttt{FETCH(}n\texttt{)} : \tau_n.s, e, k$

[QU] $$\frac{\mathit{type}(c) = \tau}{s, e, k \vdash \texttt{QUOTE(}c\texttt{)} : \tau.s, e, k}$$

[IO] $s, e, k \vdash \texttt{READ} : i.s, e, k \qquad s, e, k \vdash \texttt{WRITE} : s, e, k$

[IF] $$\frac{s, e, k \vdash C_1 : \tau.s, e, k \qquad s, e, k \vdash C_2 : \tau.s, e, k}{b.s, e, k \vdash \texttt{COND(}C_1\texttt{, }C_2\texttt{)} : \tau.s, e, k}$$

[FN] $$\frac{\begin{array}{c} C = C'.\texttt{RETURN} \\ \alpha, \tau_1.e, \beta \vdash C' : \tau_2.\alpha, \tau_1.e, \beta \end{array}}{s, e, k \vdash \texttt{FN(}\tau_1 \to \tau_2\texttt{, } C\texttt{)} : (\tau_1 \to \tau_2).s, e, k}$$

[RF] $$\frac{\begin{array}{c} C = \texttt{UNROLL(}\mathit{rec}\,\tau_1 \to \tau_2, C'\texttt{)}.C'.\texttt{RETURN} \\ \alpha, \tau_1.\mathit{rec}\,\tau_1 \to \tau_2.e, \beta \vdash C' : \tau_2.\alpha, \tau_1.\mathit{rec}\,\tau_1 \to \tau_2.e, \beta \end{array}}{s, e, k \vdash \texttt{FN(}\mathit{rec}\,\tau_1 \to \tau_2\texttt{, } C\texttt{)} : (\mathit{rec}\,\tau_1 \to \tau_2).s, e, k}$$

[AP] $\tau_1.(\tau_1 \to \tau_2).s, e, k \vdash \texttt{APP} : \tau_2.s, e, k \qquad \tau_1.(\mathit{rec}\,\tau_1 \to \tau_2).s, e, k \vdash \texttt{APP} : \tau_2.s, e, k$

[RE] $s, e, e'.k \vdash \texttt{RETURN} : s, e', k$

[UN] $$\frac{\alpha, \tau_1.\mathit{rec}\,\tau_1 \to \tau_2.e, \beta \vdash C : \tau_2.\alpha, \tau_1.\mathit{rec}\,\tau_1 \to \tau_2.e, \beta}{s, \tau_1.e, k \vdash \texttt{UNROLL(}\mathit{rec}\,\tau_1 \to \tau_2\texttt{, } C\texttt{)} : s, \tau_1.(\mathit{rec}\,\tau_1 \to \tau_2).e, k}$$

[SE] $$\frac{s, e, k \vdash \mathit{cmd} : s', e', k' \qquad s', e', k' \vdash C : s'', e'', k''}{s, e, k \vdash \mathit{cmd}.C : s'', e'', k''}$$

[NI] $s, e, k \vdash \mathit{nil} : s, e, k$

그림 5: tySECK's Type System

[NT] $$nil \sim \epsilon$$

[VT] $$S \sim \alpha \qquad K \sim \beta$$

[ST] $$\frac{v \sim \tau \qquad S \sim s}{v.S \sim \tau.s}$$

[ET] $$\frac{v \sim \tau \qquad E \sim e}{v.E \sim \tau.e}$$

[KT] $$\frac{E \sim e \qquad K \sim k}{(E, C).K \sim (e, C).k}$$

[P1] $$n \sim i \qquad string \sim r$$

[P2] $$true \sim b \qquad false \sim b$$

[F1] $$\frac{\begin{array}{ll} & C = C'.\mathtt{RETURN} \\ \forall S, K, v_1 \sim \tau : & (S, v_1.E, C', K) \text{ does not stuck and} \\ & \text{if } (S, v_1.E, C', K) \stackrel{*}{\to} (S', E', nil, K') \\ & \text{then } S' \sim \tau_2.s, E' \sim \tau_1.e, K' \sim k \\ & \text{where } S \sim s, E \sim e, \text{ and } K \sim k \end{array}}{(C, E) \sim \tau_1 \to \tau_2}$$

[F2] $$\frac{\begin{array}{lll} \multicolumn{3}{c}{C = \mathtt{UNROLL(}rec\,\tau_1 \to \tau_2, C'\mathtt{)}.C'.\mathtt{RETURN}} \\ \multicolumn{3}{l}{(C, E) \sim rec\,\tau_1 \to \tau_2} \\ \Rightarrow & \forall S, K, v_1 \sim \tau : & (S, v_1.(C, E).E, C', K) \text{ does not stuck and} \\ & & \text{if } (S, v_1.(C, E).E, C', K) \stackrel{*}{\to} (S', E', nil, K') \\ & & \text{then } S' \sim \tau_2.s, E' \sim \tau_1.rec\,\tau_1 \to \tau_2.e, K' \sim k \\ & & \text{where } S \sim s, E \sim e, \text{ and } K \sim k \end{array}}{(C, E) \sim rec\,\tau_1 \to \tau_2}$$

그림 6: tySECK's Value Type

7 The Target Language: k86 Machine

The target machine's address space is not pre-divided into registers and the main memory. A particular memory architecture is programmed by declaring memory areas (using `AREA`) each of which is addressed by a pair of an area name and an offset in the area. Conventional machines that have registers `r0, r1` and etc. can be mimicked by declaring an area, say, `r` and using locations `r(0), r(1)` and so on.

A single memory location can have either an integer, a string (of any length), a label, or a location.

k86 programs are interpreted by our interpreter. The interpreter, after executing a k86 program, reports the execution cost of the program. The execution cost consists of the number of instruction executions, memory consumption, memory accesses, and etc. The interpreter is parameterized by a cost table that determines the costs of using particular areas. By using different cost tables we can measure a k86 program's execution cost on various memory architectures.

The syntax of the target language k86 is:

pgm	::=	$decl^+ instr^+$	program
decl	::=	`AREA` *id*	memory area declaration
instr	::=	`MOVE` *m m*	
	|	`ADD` *m m m*	
	|	`SUB` *m m m*	
	|	`MUL` *m m m*	
	|	`DIV` *m m m*	
	|	`TOZ` *m m*	
	|	`JMP` *m*	
	|	`JMPZ` *m m*	
	|	`JMPN` *m m*	
	|	`LAB` *label*	
	|	`READ` *m*	
	|	`WRITE` *m*	
m	::=	*integer* | "*string*" | *id* | *label*	
	|	*m*`(`*m*`)`	
	|	*m*`@`$^+$	
id, *label*			alpha-numeric identifier

For example, a summation program in k86:

```
AREA A          // sum 0 upto A
```

```
AREA S          // S has the result
LAB     START
        READ  A
        MOVE  0  S
LAB     LOOP
        ADD    S@  A@  S
        SUB    A@  1   A
        JMPZ   A@  EXIT
        JMP    LOOP
LAB     EXIT
        WRITE  S@
LAB     END
```

Let's define the meaning of k86 command.

$$
\begin{array}{lllll}
M & \in & Mem & = & Loc \to Val & \text{memory} \\
 & & Loc & = & Area \times Int & \text{location} \\
 & & Val & = & Loc \cup Int \cup Lab \cup String & \text{value} \\
pc & \in & Int & & & \text{program counter} \\
m & \in & Addr & & & \text{address term} \\
\ell & \in & Lab & & & \text{label}
\end{array}
$$

One execution step

$$(M, pc) \to (M', pc')$$

indicates that a memory M changes to M' after an instruction at pc is executed, and the next program counter becomes pc'. Notation $M[x \mapsto y]$ indicates a new memory that is equivalent to M except at x, where its value is y.

$$
\begin{array}{lll}
(M, pc : \texttt{MOVE}\ m_1\ m_2) & \to & (M[val(M,m_2) \mapsto val(M,m_1)], pc+1) \\
(M, pc : \texttt{ADD}\ m_1\ m_2\ m_3) & \to & (M[val(M,m_3) \mapsto val(M,m_1) \oplus val(M,m_2)], pc+1) \\
(M, pc : \texttt{SUB}\ m_1\ m_2\ m_3) & \to & (M[val(M,m_3) \mapsto val(M,m_1) \ominus val(M,m_2)], pc+1) \\
(M, pc : \texttt{MUL}\ m_1\ m_2\ m_3) & \to & (M[val(M,m_3) \mapsto val(M,m_1) \otimes val(M,m_2)], pc+1) \\
(M, pc : \texttt{DIV}\ m_1\ m_2\ m_3) & \to & (M[val(M,m_3) \mapsto val(M,m_1) \oslash val(M,m_2)], pc+1) \\
(M, pc : \texttt{TOZ}\ m_1\ m_2) & \to & (M[val(M,m_2) \mapsto cast(val(M,m_1))], pc+1) \\
(M, pc : \texttt{JMP}\ m) & \to & (M, pcfy(val(M,m))) \\
(M, pc : \texttt{JMPZ}\ m_1\ m_2) & \to & (M, \text{if } val(M,m_1) = 0 \text{ then } pcfy(val(M,m_2)) \text{ else } pc+1) \\
(M, pc : \texttt{JMPN}\ m_1\ m_2) & \to & (M, \text{if } val(M,m_1) < 0 \text{ then } pcfy(val(M,m_2)) \text{ else } pc+1) \\
(M, pc : \texttt{READ}\ m) & \to & (M[val(M,m) \mapsto get()], pc+1) \\
(M, pc : \texttt{WRITE}\ m) & \to & (M, pc+1) \\
(M, pc : \texttt{LAB}\ \ell) & \to & (M, pc+1) \\
(M, pc : \texttt{LAB END}) & \to & (M, pc)
\end{array}
$$

Auxiliary function $val(M, m)$ indicates a value represented by an address term m at a given memory M:

$$val: Mem \times Addr \rightarrow Val$$

$val(M, a)$	$= \langle a, 0 \rangle$	aread id
$val(M, x)$	$= x$	integer/string/label
$val(M, m_1(m_2))$	$= val(M, m_1) \oplus val(M, m_2)$	offset address
$val(M, m\texttt{@})$	$= M(val(M, m))$	dereference

Casting function $cast(v)$ casts a location to an integer by removing the area name from the location:

$$cast: Loc \rightarrow Int$$

$$cast\langle a, z \rangle = z.$$

$v_1 \oplus v_2$ is defined only when

- both v_1 and v_2 are integer. Then the $\oplus$ is the integer addition.
- both v_1 and v_2 are locations whose areas are identical. Then $v_1 \oplus v_2$ is the location in the area whose offset is the sum of v_i's offsets.
- one of them is an integer and the other is a location. Then the result is the location in the same area whose offset is incremented by the integer.

Similarly for $\ominus$,$\otimes$, and $\oslash$.

- "`AREA sp`" declares new memory area `sp`.
- "`MOVE 2 sp`" moves constant 2 to location $\langle \texttt{sp}, 0 \rangle$.
- "`MOVE "cs" sp`" moves string "`cs`" to location $\langle \texttt{sp}, 0 \rangle$.
- "`MOVE sp@ sp`" moves value at location $\langle \texttt{sp}, 0 \rangle$ to location $\langle \texttt{sp}, 0 \rangle$.
- "`MOVE sp(-1)@ sp`" moves value at $\langle \texttt{sp}, -1 \rangle$ to location $\langle \texttt{sp}, 0 \rangle$.

Function $pcfy(\ell)$ returns a label ℓ's program counter value, and $input()$ gets an integer from the outside world.

Interpretation (execution) of a program P is the sequence of states starting from an empty memory and the initial program counter at label `START`:

$$([], pcfy(\texttt{START})) \rightarrow \cdots$$

Note that the end of the program is when the program counter reaches the `END` label. Therefore any program must have the two labels (`START` and `END`).

8 Translation from tySECK to k86

We need to implement the semantics of tySECK codes by k86 intructions. That is, we need to implement the three components of tySECK machine (stack S, environment E, and continuation K) and we need to implement the tySECK values in k86 objects.

- S, E, and K are implemented as stacks in separate memory segments of k86 machine.

- Primitive values in tySECK are directly implemented as primitive values in k86, except that boolean values are implemented in k86 as integers: true as non-zero integers and false as zero.

- Function values (C, E) is implemented as a pointer to a pair whose first field has the code label and the second field the pointer to the environment. Similarly for continuation pair (E, C). The code label is the label at the beginning of the code.

- Now the question: how and where do we implement the function's environment part? Can the environment be implemented as a pointer to the current machine environment E? No, because machine environment and function value's environment have different life-spans. The environment when a function is called is in general different from the environment which was captured when the function is defined. The values in the current machine environment E can be expired (hence the memories that were used to implement the values are re-cycled) while the captured environment in functions can last. For example, consider the following program:

  ```
  let
   f = let x = 2 in fn y => x+y end              (1)
  in
   f z                                           (2)
  end
  ```

  The `f` has a function value whose environment part has the entry $[\mathtt{x} \mapsto 2]$. This entry in the machine environment is removed (by the `DEFRAME` command) when the `let`-block at line (1) is exited. Note that this entry has to be available when the function is called at line (2).

- Thus, function's environment has to be implemented in a separate memory area whose objects last "forever" without being expired. We call this area "heap."

We now know that four memory areas are necessary:

```
AREA S  // stack
AREA E  // environment
AREA K  // continuation
AREA H  // heap
```

Each area's first empty address is pointed to by each pointer:

```
AREA sp // stack pointer sp
AREA ep // environment pointer ep
AREA kp // continuation pointer kp
AREA hp // heap pointer hp
```

In defining the translation rule we use the following macros:

```
macro push(x, ptr)                    =  MOVE x ptr@
                                         ADD 1 ptr@ ptr
macro pop(ptr)                        =  SUB ptr@ 1 ptr
macro copy(REPEAT, from, to, n, END)  =  LAB REPEAT
                                          JMPZ n@ END
                                          JMPN n@ END
                                          MOVE from(n@)@ to(n@)
                                          SUB n@ 1 n
                                          JMP REPEAT
                                         LAB END
```

The translation function `codegen` is of type

```
codegen:  tyCmd -> k86Cmd * k86Cmd
```

The second output is for the k86 codes of functions that appear in the given tySECK command. Given a sequence of tySECK command C, `codegen(`C`)` will return two k86 code-sequences

$$(C_1,\ C_2)$$

from which the final k86 code would be

$$\texttt{LAB START}\ C_1\ \texttt{LAB END}\ C_2$$

with appropriate area declarations in front of the `START` label.

The `codegen` function would be defined something like in Figure 7. In the figure I omitted the second part when it's empty.

```
fun codegen FRAME =  push(sp@(-1),ep)
                     pop(sp)
  | codegen DEFRAME  = pop(ep)
  | codegen FETCH(n) = push(ep@(-n-1),sp)
  | codegen QUOTE(n) = push(n,sp)
  | codegen QUOTE(s) = push(s,sp)
  | codegen QUOTE(true) = push(1,sp)
  | codegen QUOTE(false) = push(0,sp)
  | codegen ADD =  ADD sp@(-1) sp@(-2) sp@(-2)
                   SUB sp@ 1 sp
  | codegen COND(C1, C2) =
      let
         val C1 = codegen C1
         val C2 = codegen C2
         val ELSE = newLabel()
         val EXIT = newLabel()
      in
            JMPZ sp@(-1) ELSE
            pop(sp)
            C1
            JMP EXIT
        LAB ELSE
            pop(sp)
            C2
        LAB EXIT
      end
  | codegen FTN(C) =
      let
         val C = codegen C
         val ENTRY = newLabel()
         val REPEAT = newLabel()
         val END = newLabel()
      in
       ( TOZ ep@ sp@           (* sp@@ has the current environment size *)
         push(sp@@, hp)        (* put the environment size at the head of
                                  the copied environment *)
         copy(REPEAT, ep@, hp@, sp@, END) (* copy the env into the heap *)
         ADD hp@ sp@@ hp       (* shift the hp@ by the env size *)
         ADD hp@ 2 hp          (* allocate the pair cell in the heap *)
         MOVE ENTRY hp@(-1)    (* second field has the code label *)
         MOVE hp@(-(sp@+3))
              hp@(-2)          (* first field has the ptr to the copied env *)
         push(hp@(-2),sp)      (* stack has the pointer to the pair *)
       , C)
      end
  | ...
```

그림 7: tySECK-to-k86 Translation Rules

8.1 Economic/Optimal Translation into k86: Challenges and Opportunities

The translation procedure in the previous section has a lot room for improvement.

- When a function value is made, taking one copy of the current environment into the heap area takes a long time. Under some conditions we don't have to copy the environment. The current environment can be just pointed to. What would be the conditions?

- When a function is called, the return state (the pair of the current environment and next command to continue after returning) is pushed into the continuation stack K. In particular, the environment part is copied. Under some conditions we can avoid this expensive copy process, too. What would be the conditions?

- When a recursive function is unrolled by the `UNROLL` command, the environment part E' is duplicated:

$$(S,\ v.E',\ \underbrace{\texttt{UNROLL(}rec\,\tau_1 \to \tau_2\texttt{,}\ \ C'\texttt{)}.C}_{C''},\ K) \to (S,\ v.(C'', E').E',\ C,\ K)$$

  I guess we can always avoid this duplicated copy. Do you agree?

Other than the above big chances for optimizations we can do many simple optimizations after the translation is done by examining a small number of tySECK or k86 codes in sequence. To enlist some,

- When we see a tySECK code sequence like

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

  we can replace it by

  `QUOTE(3)`

- Two instructions `move` m_1 `sp@` immediately followed by `move sp@@` m_2 can be replaced by a single command `move` m_1 m_2.