Challenge Set

SNU 4190.310, 2026 Spring

Kwangkeun Yi

Due: 6/22(Mon) 16:00 @ IN box, Rm428 Bldg302

- 올바른 답을 제출한 사람은 최종 성적이 상승합니다.
- 틀린 답을 제출한 사람은 최종 성적이 하강합니다.
- AI사용 관련:
  - AI와 협력해서 답을 만든 경우, AI와 주고받은 대화 일체를 함께 제출해야합니다.
  - AI와 협력하지 않은 경우, 답만 제출합니다.

Challenge 1 (1↑, 2↓) "증명 I"

강의 홈페이지의 강의 슬라이드 `5-1simple-type.pdf`에는 단순 타입 시스템의 안전성 증명에 필요한 세 개의 정리들("Progress", "Preservation", "Preservation under Substitution")이 있다. 이 정리들의 증명을 모두 완성하라. □

Challenge 2 (1↑, 2↓) "증명 M"

강의 홈페이지의 강의 슬라이드 `5-1simple-type.pdf`에는 단순 타입시스템의 온라인 알고리즘들 M과 W가 있다. M 알고리즘이 단순 타입 시스템의 충실한 구현임(강의 슬라이드에 명시한 성질)을 증명하라. □

Challenge 3 (1↑, 2↓) "재귀함수 = 함수 + 메모리 지정문"

단순 타입 시스템(simple type system)을 갖춘 다음의 언어를 생각하자. 적극적인 계산법(eager evaluation)으로 실행되는 언어이다.

e	::=	n	integer
	|	x	variable
	|	$\lambda x.e$	function
	|	$e\ e$	application
	|	`if0` $e\ e\ e$	application
	|	e`+`e	addition
	|	`ref` e	memory allocation
	|	e `:=` e	assignment
	|	`!` e	dereference

이제 위의 언어를 가지고 재귀함수를 흉내내는 방법을 고안하자. 골치거리는 단순 타입 시스템 때문에 Y-combinator등의 방식으로는 재귀함수 효과를 낼 수가 없다는 것이다. 왜냐하면 Y-combinator류의 식을 보면

$$\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

자기호출식("$x\,x$")이 있는데, 이 호출식이 단순 타입 시스템을 통과할 수 없기 때문이다. 자기호출식에 사용되는 x의 단순 타입(함수타입이면서 동시에 그 함수의 인자타입이어야 하는 단순 타입)은 존재하지 않는다. (다형 타입(polymorphic type)을 갖춘 타입 시스템이라면 가능할 수 있다. 그러한 x의 타입은 $\forall\alpha\forall\beta.\alpha \to \beta$ 이면 된다. 자기호출식의 왼쪽 x의 타입(type instantiation)은 $(\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2)$로 하고 오른쪽 x의 타입은 $\alpha_1 \to \alpha_2$로 생각할 수 있으므로.)

과연, 단순 타입 시스템을 갖춘 위의 언어로 재귀함수를 흉내낼 수 있는 방법은 없는 것일까? 가능하다. 한 예로 재귀함수 factorial를 어떻게 흉내낼 수 있을 지를 보여라. 제안한 방법은 단순 타입 시스템을 통과할 수 있어야 한다. □

Challenge 4 (2↑, 1↓) "재귀호출의 비용"

학생: "저는 프로그램 짤 때 재귀함수로 짜지 않습니다. 함수 호출은 일반적으로 시간과 메모리를 너무 많이 잡아먹는데, 재귀 호출은 그게 너무 많이 반복되잖습니까. 그래서 피하고 있습니다."

이 챌린지에서는, 위 학생의 이야기가 무슨 뜻인지, SM5 프로그램의 실행 과정을 관찰하면서 알아보고, 재귀함수호출이 가지는 비용을 줄이는 개선책을 찾아 나서보자. 현재 대부분의 고급 언어 실행기(interpreter)와 번역기(compiler)들이 사용하는 기술이다.

- 함수호출이 다른 명령에 비해 비용이 많이드는 이유:

  SM5의 실행과정을 보면 그 이유가 드러난다. 함수 `call` 명령어가 실행될 때마다, K (continuation) 파트에 함수가 끝나고 계속 진행할 내용을 쌓아갔다:

$$\begin{array}{rllll} & (l :: v :: (x, C', E') :: S, & M, & E, & \texttt{call} :: C, \quad K) \\ \Rightarrow & (S, & M[v/l], & (x, l) :: E', & C', \quad \underline{(C, E) :: K}) \end{array}$$

  그리고 K에 기록된 "계속할일/마저할일"은 함수가 끝나면 복원해 주었다:

$$\begin{array}{rllll} & (S, & M, & E, & \mathit{empty}, \quad (C, E') :: K) \\ \Rightarrow & (S, & M, & E', & C, \quad K) \end{array}$$

  즉, 함수의 호출은 호출이 끝나고 할 일을 K에 넣다 뺐다하는 과정이 필요하고, 함수 호출이 재귀적으로 일어나면 그러한 것들이 K에 계속해서 쌓이기 때문에 메모리나 자원을 많이 쓰는 셈인 것이리라. 예를 들어, factorial 함수 같은 경우 "`n * fac(n-1)`"이라는 재귀호출 부분 때문에 K에 쌓이는 것은 변수 `n`의 크기만큼이 될 것이다.

- 언제 어떻게 K에 넣다 뺐다 하는 것을 줄일 수 있을까?

  특이한 재귀함수의 호출인 경우 줄일 수 있다. 재귀함수 중에서 재귀호출이 끝나고 나서, 마저할 일이 다시 리턴하는 것(현재 호출을 끝내는 것) 밖에는 없는 꼴이 있다. 예를 들어,

```
procedure f(x) = if x<0 then 1 else f(x-1)
```

  에서, 재귀호출 `f(x-1)`이 끝난 후에 할 일은 아무것도 없다. 재귀호출한 현재 호출을 끝내는 것 이외에는. 이러한 재귀호출을 끝재귀호출(tail-recursive call)이라고 한다. 맨 마지막(tail)에 하는 일이 재귀호출밖에는 없다는 뜻이다. (한편 factorial은 끝재귀가 아니다. 재귀호출을 하고 나서 할일이 있다: 재귀호출 결과(`fac(n-1)`)에 `n`을 곱하는 일.)

이정도로 힌트는 마치기로 하고, 여러분이 할 일은 SM5에 명령어를 첨가해서 SM5x를 만들고, K--의 재귀호출을 SM5x로 번역할 때, 첨가한 새로운 SM5x 명령어를 이용해서 위에서 언급한 함수호출비용(K에 뭔가가 자꾸 쌓이는 현상)이 줄어들도록 하는 것이다.

제출할 프린트물은, Sm5x에서 첨가한 명령어의 의미정의와 새로운 `trans`함수의 정의이다. 코드에는 첨가된 명령어들의 정의를 코멘트로 잘 설명하도록 한다.

□

Challenge 5 (1↑, 2↓) "헨젤과 그레텔"

뿌리 노드가 하나 정해져있는 그래프를 입력으로 받아서, 그 뿌리에서 도달할 수 있는 모든 노드들을 마크하는 알고리즘. 이것이 Mark-and-Sweep이나 Stop-and-Copy라는 방식으로 메모리를 재활용할 때 필요한 기본 엔진이다.

이 엔진 기술이 교과서에 있는 기초적인 수준이라면(depth-first-traveral 또는 breadth-first-traversal) 메모리재활용에 사용하기에는 문제가 많다. 입력된 그래프의 노드 갯수에 비례하는 메모리가 따로 더 필요하기 때문이다.

수업시간에 언급한 "헨젤과그레텔" 방식에 기초해서 잘 고안한다면 메모리를 임의로 많이 사용하지 않는 엔진을 만들 수 있다. 이 방법으로는 세 개의 변수와 노드와 포인터(엣지)마다 한 비트만 더 있으면 늘 모두 마크할 수 있다. 노드의 비트는 마크여부를 기록하고, 포인터(엣지)의 비트는 다른 용도다. 이 비트들과 세 개의 변수들은 그래프를 훑으며 모두 방문하는데 알뜰하게 사용된다.

이 알고리즘을 C-비슷한 가상의 언어로 작성한 아래 코드에서 물음표한 줄을 완성하라.

```
// Each node has one visit bit.
// Each pointer has one flip bit.
//
// mark: node pointer × node pointer → unit
// mark(c,p) marks every node reachable from current node c and previous node p.
// Assume the graph is given.
// Assume that initially every mark bit and flip bit are 0.
// Assume that initial call is mark(R,null) where R is the root node of the graph.

mark(c,p) =
   c→visitBit = 1
   if moreToVisit then
     let c→next be an unflipped pointer whose target is not visited
         ?
         ?
         ?
         mark(tmp, c)
   else //no more next to visit from current, then go back to previous
     if moreToGoBack then
       let p→next be a flipped pointer
           ?
           ?
           ?
           mark(p, tmp)
    else return
// where
// moreToVisit = exists an unflipped pointer from c whose target is not visited
// moreToGoBack = exists a flipped pointer from p
```

위 코드에서 필요한 메모리는 세 개의 변수(c, p, tmp), 그리고 노드와 포인터마다 한 개의 비트뿐이다. 위 코드가 재귀호출을 하지만 모두 끝재귀호출(tail-recursive call)이므로 단순 반복문과 같다 (Challenge 4). 즉, 더 이상의 메모리 소모없이 단순 반복문으로 쉽게 변환될 수 있다. □

Challenge 6 (3↑, 1↓) "메모리는 설탕"

메모리 반응식(imperative operations)들은 모두 설탕들인가?

아래는 적극적 계산법(eager evaluation)으로 실행되는 람다 언어이다. 메모리 명령문들의 실행은 익히 알고 있는 방식이다.

e	::=	x	variable
	|	$\lambda x.e$	function
	|	$e\ e$	application
	|	`ref` e	malloc
	|	e `:=` e	assignment
	|	`!`e	dereference

강의에서 이야기 한 바, 주어진 프로그램에 있는 메모리 명령문들을 나머지 다른 방식의 식들로 녹여서 같은 일을 하는 프로그램으로 바꿀 수 있다. 그 방안을 생각해 보자.

- 위와 같은 메모리 반응 식들은 메모리를 필요로한다.
- 그리고, 메모리에 반응하는 순서가 중요해 진다.

따라서 변환할 때는

- 프로그램식들의 실행순서를 존중하면서
- 각 식들이, 이전 순서 식의 결과값과 메모리를 받아서, 결과로 현재 식의 값과 결과 메모리를 리턴하도록 하면 된다.

그래서,

- 모든 식들이 함수가 된다: 이전식의 값과 메모리를 받아서 현재 식의 값과 반응이 일어난 메모리를 결과로 내놓는.

이제 남은 두 개의 질문:

1. 순서대로 계산되는 과정을 함수로 표현하는 방법은?
2. 메모리를 다른 식으로 어떻게 표현?

1답 : 위와같은 변환(할일의 순서를 드러내서 함수로 표현하는 변환)은 여러가지가 있을 수 있다. 아주 기본적은 방법으로는 CPS-변환(continuation-passing-style transformation)이 있다.

2답 : 많은 방법이 있을 수 있다. 예를 들어, 짝으로:

$$\text{counter} \times \text{list of (주소}\times\text{값)}$$

표현할 수 있을 것이다. counter는 현재 메모리에 할당된 메모리 갯수.

프로그램 식 e의 메모리 설탕을 녹이는 룰을 찾아라. □

Challenge 7 (3↑, 1↓) "코드계산은 설탕"

메타프로그래밍(meta programming)은 프로그램 코드를 계산대상으로 하는 프로그래밍을 말합니다. 코드를 실행하면서 코드를 만들고 만든 코드를 실행하는 일을 맘껏 할 수 있는 프로그래밍입니다.

메타프로그래밍을 지원하는 언어는 이미 여럿 존재합니다. Lisp(Scheme)이라는 언어에 있는 "quasi-quotation"이 그런 예이고, C의 매크로나 C++와 Haskell의 템플렛(template)이 또 그런 예이고, C#이나 JavaScript, PHP, Python에서도 제한적이지만 코드를 스트링으로 만들고 실행하는 것이 가능한 언어들입니다.

다음과 같은 메타프로그래밍 언어를 생각합시다. 적극적인 계산법을 사용하는 언어이다.

$$\begin{array}{rcll} e & \to & \cdots & \\ & | & x & \\ & | & \lambda x.e & \\ & | & e\,e & \\ & | & \texttt{'}e & \text{코드 데이터} \\ & | & \texttt{,}e & \text{코드에 껴드는 코드만들기 식} \\ & | & \texttt{run}\,e & \text{만든 코드의 실행} \end{array}$$

메타프로그래밍 코드의 예를들면 다음과 같다:

- 값으로서의 코드

```
'(1+1)
```

- 값으로서의 코드인데 자유변수를 가질 수 있고

```
'(x+1)
```

- 코드를 조립하면서 자유변수가 묶이기

```
let y = '(x+1) in '(λx.,y)
```

- 코드 실행하기

```
run '(1+1)
```

메타프로그래밍은 특화된 코드를 미리 준비해놓고 사용하는 데 많이 쓰인다. 예를 들어, 자연수 두 개 x와 n을 받아서 x^n을 계산하는 다음의 재귀 함수 코드를 보자.

```
let power x n = if n=0 then 1 else x * (power x (n-1))
```

식 `(power 2 3)`은 함수를 3회 반복 호출하면서 2^3을 계산하게된다. 비용이 걱정된다.

이제 다음과 같은 코드를 생각하자. 가정으로 `power`함수가 여러가지 `x`에 대해서 주로 3승을 계산한다고 하자. 그러면 `x*x*x`라는 코드를 따로 준비해 놓고, 나중에 `x`를 2로 하고 위 코드를 실행하면 함수호출없이 2^3 을 계산한다. 이렇게 미리 특화된 코드를 만들어 놓으면 실행비용을 줄일 수 있다. 다음과 같이 메타프로그래밍으로 3승에 특화된 코드를 만들어놓고 사용하면 된다:

```
let spower n = if n=0 then '1 else '(x * ,(spower (n-1))) in
let pow3 = '(λx.,(spower 3)) in
(run pow3) 2 ··· (run pow3) 8 ···
```

사실, 메타언어에서 코드를 만들고 사용하는 식들은 모두 설탕이다.

메타설탕을 녹이는 번역규칙을 정의하라. 출발어와 도착어는 다음과 같다. 모두 적극적인 계산법(eager evaluation, call-by-value)으로 작동하는 언어들이다.

- 출발어는 핵심만 있는 메타프로그래밍언어다:

$$
\begin{array}{rcll}
e & \to & x & \\
 & | & \lambda x.e & \\
 & | & e\,e & \\
 & | & \texttt{'}e & \text{데이터로서 코드 } e \\
 & | & \texttt{,}e & \text{코드안에서 } e\text{의 결과코드 껴넣기} \\
 & | & \texttt{run}\,e & e\text{의 결과코드를 실행}
\end{array}
$$

- 도착어는 메타설탕이 없고 레코드가 추가된 언어다:

$$
\begin{array}{rcll}
e & \to & x & \\
 & | & \lambda x.e & \\
 & | & e\,e & \\
 & | & \{\} & \text{빈 레코드} \\
 & | & e\{\texttt{x = } e\} & \text{레코드 확장} \\
 & | & e\texttt{.x} & \text{레코드 필드값}
\end{array}
$$

- 힌트로서, 번역의 예를 들면 다음과 같다.
  - 코드값은 함수로 변환된다:

$$\texttt{'(1+1)} \longmapsto \lambda\rho.\texttt{1+1}$$

  - 자유변수는 레코드 필드값으로 변환된다:

$$\texttt{'(x+1)} \longmapsto \lambda\rho.(\rho\texttt{.x})\texttt{+1}$$

  이 힌트를 기반으로, 출발어의 각 식의 경우마다 조립식으로 설탕을 녹이는 규칙을 찾으라.

Challenge 8 (2↑, 1↓) "GC mini확인"

메모리 재활용기(garbage collector)는 올바른가?

강의에서 논의한대로, 자동 메모리 재활용기(garbage collector)의 원리는 이렇습니다:

- **원칙**: 프로그램의 진행을 멈추고 나서, 지금까지 할당된 메모리중에서 미래에 사용할 수 있는 메모리를 제외하고 나머지는 모두 재활용해야 한다.
- **사실**: 재활용을 완전하게(빠뜨리지 않고) 할 수 있는 방법은 없다. 있다면 그런 재활용기를 이용해서 *Halting Problem*을 풀 수 있기 때문이다.
- **양보**: 그러나 재활용을 안전하게는(빠뜨리는 것은 있으나) 할 수 있는 방법은 있다. 뭔고하니,

⋆ 실행할 식 E의 현재 환경(environment)으로부터 현재의 메모리를 통해 다다를 수 있는 모든 주소들은 앞으로 E를 실행중에 다시 사용될 가능성이 있다.

이것들만 빼고 재활용하자. 즉, 그렇게 해서 다다를 수 없는 메모리 주소들은, 과거에 할당되어서 사용되었으나 앞으로의 E를 실행하는데는 사용되지 않을 것이 분명하므로, 재활용해도 된다.

모든 현대 언어들의 메모리 재활용기는 위의 방식으로 구현되어 있습니다 (Java, ML, Scheme, Haskell, C#, Prolog, etc.)

이번 챌린지에서는 위의 주장 $\star$ 가 옳은지를 확인하는 것입니다.

위의 사실이 어떻게 엄밀한 정리(theorem)로 표현되는지를 살펴보시기 바랍니다. 우선, 그 정리에서 사용하는 용어를 정확히 정의해야 합니다. 프로그램은 어떤 언어로 짠 프로그램을 말하는지, 프로그램의 실행(semantics)은 무엇인지, 환경에서 다다를 수 있는 메모리는 무엇을 뜻하는 지에 대한 정의들.

대상 프로그래밍 언어는 D 라는 언어로 합시다. D 프로그램 실행의 정확한 정의는 아래와 같습니다.

[Now on in English]

e	::=	n	integer
	|	x	variable
	|	`{`x `:=` e`}`	record
	|	`{}`	nil record
	|	$e.x$	record field
	|	x`:=`e	assignment
	|	e`;`e	sequence
	|	`let` x e e	local block

x	$\in$	Var			variables
v	$\in$	Val	$=$	$Num + Record$	values
n	$\in$	Num			numbers
l	$\in$	Loc			locations
r	$\in$	$Record$	$=$	$Var \xrightarrow{\text{fin}} Loc$	records
σ	$\in$	Env	$=$	$Var \xrightarrow{\text{fin}} Loc$	environments
M	$\in$	$Memory$	$=$	$Loc \xrightarrow{\text{fin}} Val$	memories

Notation: $A \xrightarrow{\text{fin}} B$ is the set of functions from a finite subset of A to B. Let f be a function $\{a \mapsto 1, b \mapsto 2\}$, then we write $dom f$ for the domain $\{a, b\}$ of f. We write $f[2/a]$ for a new function $\{a \mapsto 2, b \mapsto 2\}$, and $f[3/c]$ for $\{a \mapsto 1, b \mapsto 2, c \mapsto 3\}$.

The semantics rules precisely defines the execution of D expressions; they define how relations of the form

$$\sigma, M \vdash e \Rightarrow v, M'$$

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

Definition 1 (Expression's semantics/execution) *An expression e's execution is defined to be the inference tree for* $(\sigma, M \vdash e \Rightarrow v, M')$. *If there is no such* $\sigma, M, v,$ *and* M', *then the expression has no meaning, no way to execute.*

In particular, program e's execution is the inference tree for $(\emptyset, \emptyset \vdash e \Rightarrow v, M')$ *for some* v *and* M'.

[Int] $$\sigma, M \vdash n \Rightarrow n, M$$

[Var] $$\frac{M(\sigma(x)) = v}{\sigma, M \vdash x \Rightarrow v, M}$$

[Rec] $$\frac{\sigma, M \vdash e \Rightarrow v, M' \qquad l \notin domM \cup domM'}{\sigma, M \vdash \texttt{\{}x\ \texttt{:=}\ e\texttt{\}} \Rightarrow \{x \mapsto l\}, M'[v/l]}$$

[NilRec] $$\sigma, M \vdash \texttt{\{\}} \Rightarrow \{\}, M$$

[Field] $$\frac{\sigma, M \vdash e \Rightarrow \{x \mapsto l\}, M' \qquad M'(l) = v}{\sigma, M \vdash e.x \Rightarrow v, M'}$$

[Assign] $$\frac{\sigma, M \vdash e \Rightarrow v, M' \qquad \sigma(x) = l}{\sigma, M \vdash x\ \texttt{:=}\ e \Rightarrow v, M'[v/l]}$$

[Seq] $$\frac{\sigma, M \vdash e_1 \Rightarrow v_1, M' \qquad \sigma, M' \vdash e_2 \Rightarrow v_2, M''}{\sigma, M \vdash e_1\ ;\ e_2 \Rightarrow v_2, M''}$$

[Let] $$\frac{\sigma, M \vdash e_1 \Rightarrow v_1, M' \qquad \sigma[l/x], M'[v_1/l] \vdash e_2 \Rightarrow v_2, M'' \qquad l \notin domM \cup domM'}{\sigma, M \vdash \texttt{let}\ x\ e_1\ e_2 \Rightarrow v_2, M''}$$

Definition 2 *reach*(σ, M) *is the set of locations in* M *that are reachable from*

the entries in σ. It is the smallest set that satisfies the two rules:

$$\sigma(x) \in reach(\sigma, M) \qquad \frac{l \in reach(\sigma, M) \qquad M(l) = \{x \mapsto l'\}}{l' \in reach(\sigma, M)}$$

Now the correctness of the garbage collector idea hinges on the following theorem.

Theorem 1 *In the inference of $(\sigma, M \vdash e \Rightarrow v, M')$, the set of used (read or written) locations in M is included in $reach(\sigma, M)$.*

The proof of the theorem needs the following lemma:

Lemma 1 *If $(\sigma, M \vdash e \Rightarrow v, M')$ is infered, then $reach(\sigma, M) \supseteq reach(\sigma, M') \cap domM$.*

Both the proofs of the theorem and the lemma can be done by induction on expression e. You are challenged to complete the two proofs. □

Challenge 9 (2↑, 1↓) "람다예보"

현대의 프로그래밍 언어들(OCaml, Scala, Rust, Haskell, Python, JavaScript, C++14, C#, F# 등)은 함수를 자유롭게 다룰 수 있게 해준다. 함수가 여타 다른 데이터와 다르지 않다.

이 챌린지는 그런 프로그램의 경우, 실행중에 무슨 일이 일어날 지를 미리 자동으로 예측하는 도구를 고안하는 것이다. 특히, 그런 프로그래밍 언어의 핵심만을 대상으로해서, 무슨 함수가 어디에서 호출되는 지를 예측하는 도구(프로그램)을 고안하자. 이런 도구를 통해서, 우리가 짠 프로그램이 실행중에 혹시나 우리가 생각하지 못한 행동을 할 지를 확인하는 데 도움을 받게 된다.[1]

아래의 적극적인(eager-evaluation, or call-by-value) 프로그래밍 언어를 생각

[1]우리가 만드는 물건이 생각대로 작동할 지를 미리 확인하려는 욕구는 모든 공학분야에 공통적이다. 따라서 프로그램 실행을 미리 예측하려는 것은 특별한 것이 아니다. 다른점은 여타 공학이나 과학과 달리 그 대상이 프로그램이라는 것 뿐이다. 우리는 프로그램을 대상으로 그 다이나믹스(실행상황)를 미리 예측하는 방법을 실현해 보는 것이다. 자연의 다이나믹스를 예측하는 물리학, 혹은 기계장치와 전기장치의 다이나믹스를 예측하는 기계공학과 전기공학과 같다.

하자. 수업에서 다룬 언어의 일부다.

$$
\begin{array}{rcll}
e & ::= & n & \text{integer} \\
 & | & x & \text{identifier} \\
 & | & \lambda x.e & \text{function} \\
 & | & f\ \lambda x.e & \text{recursive function} \\
 & | & e\ e & \text{application} \\
 & | & \texttt{ifz}\ e\ e\ e & \text{branch} \\
 & | & e\ \texttt{+}\ e & \text{addtion}
\end{array}
$$

프로그램에는 함수식들이 산재해 있는데, 우리가 예측할 것은 함수호출식마다 어떤 함수가 호출될 지를 예측하는 것이다. 예를들어 아래 식을 보면 다섯 개의 람다식이 있다.

$$(\lambda f.(\lambda x.(f\ 0)\ (x\ 10)))\ (\lambda y.(\lambda x.x + y)\ (\lambda z.z)$$

위의 식에서 함수호출식은 세 개가 있는데 ($f\ 0$, $x\ 10$, $(f\ 0)\ (x\ 10)$) 이 식들이 어떤 함수를 호출하게 될 지를 미리 예측하는 것이다.

일반적으로 프로그램의 모든 식마다 그 식이 계산 결과로 내놓게되는 함수들의 집합을 예측하면 될 것이다. 그 집합들에 관한 방정식을 세우고 풀 수 있으면 된다.

- 방정식의 변수(unknown): 주어진 프로그램의 모든 식마다 번호를 붙이고 프로그램 변수들은 모두 다르다고 하자. 방정식의 변수는 다음과 같다. 변수 X_i 는 i번 식의 실행결과로 나올 함수식들의 집합이다. 변수 X_x는 프로그램 변수 x가 가지게 될 함수식들의 집합이다.

- 연립 방정식을 세우는 규칙: 다음의 규칙을 써서 프로그램식을 한번 훑으면 방정식이 모여진다. 연립 방정식 E는 다음과 같은 꼴로 표현한다.

$$
\begin{array}{rcl}
E & ::= & \mathcal{X} \supseteq setexp \\
 & | & E \wedge E \\
setexp & ::= & \emptyset \\
 & | & \{\lambda x.e\} \\
 & | & \mathcal{X} \\
 & | & \mathcal{X}\ @\ \mathcal{Y} \\
 & | & \mathcal{X} \cup \mathcal{Y}
\end{array}
$$

위에서 $\mathcal{X}$, $\mathcal{Y}$는 방정식 변수 $X_1, X_2, \cdots$등을 뜻한다.

- 프로그램 e의 연립 방정식 E를 세우는 ($e \vdash E$ 라고 쓰자) 규칙을 e의 경우에 맞추어 정의하라.
- 방정식 풀기: 모인 연립 방정식

$$(X_1 \supseteq setexp_1) \wedge (X_2 \supseteq setexp_2) \wedge \cdots$$

을 풀면 된다. 연립 방정식을 푸는 방법은 방정식들($X_i \supseteq setexp_i$) 집합에서부터 시작해서 새롭게 알게되는 방정식들을 계속 추가하는 것이다. 더 이상 추가할 게 없을 때 까지. 어떻게 추가하면 될까? 규칙을 정의하라.

- 방정식의 해: 위의 방식으로 모두 모아가면 언젠가는 끝난다. 프로그램을 구성하는 식의 갯수와 함수식의 갯수는 유한하기 때문이다. 방정식 변수 X_i의 답은 위와같이 모은 방정식중에서 다음과 같은 명백한 꼴들을 모두 모으면 된다.

$$X_i \supseteq \{\lambda x.e\} \quad X_i \supseteq \{\lambda y.e'\} \quad \cdots$$

위에서 오른쪽에 있는 함수식들을 모은 집합

$$\{\lambda x.e,\ \lambda y.e',\ \cdots\},$$

이 집합이 식 e_i가 실행중에 가지는 함수식들을 모두 포함하게 된다. □

Challenge 10 (2↑, 1↓) "K-- + 예외상황처리"

숙제 6, "SM5"의 확장이다. 우선 K--가 확장되었다. 예외상황처리(exception handling)가 가능하도록:

$$\begin{array}{lllll} \text{식 } \textit{Expression} & E & \to & \cdots \\ & & | & \texttt{raise} \\ & & | & \texttt{try}\ E\ \texttt{handle}\ E \end{array}$$

의미는 Java나 ML에서 사용는 예외상황의 의미와 같다. 단, 하나의 예외상황만 있는 간단한 경우가 되겠다. (참고로, C의 `longjmp`/`setjmp`이 예외상황처리를 위한 것이다.)

- `raise`는 예외상황을 발동시킨다. 그러면, 정상적인 계산 진행을 멈추고 지금까지 오는 동안 장착한 가장 최근의 예외상황 처리식으로 돌아간다.
- `try` E_1 `handle` E_2 는 예외상황 처리식을 장착한다. 위의 식은 E_1만 있는 것과 다르지 않다. 단, 예외상황이 발생할 경우 E_2를 실행하도록 준비해 놓은

식이다.

의미는 다음과 같다. 우선 E_1을 계산한다. 계산 중에 예외상황이 발생하지 않는다면 E_1의 결과가 위의 `try`-식의 결과다. E_1 계산중에 예외상황이 발생하면, E_2를 계산하고 그 결과가 위의 `try`-식의 결과가 된다. 주의할 점은, E_2가 실행되는 환경(environment)은 현재의(예외상황 처리식의) 환경이다.

- 예를 들어,

  `try interest(`e`) handle interestzero(x)`

  은 `interest`를 호출한다. 이 프로시져가 다시 `divide`를 호출한다고 하자. `divide`는 계산중에 분모값이 0인경우 예외상황을 발생시키도록 되어있다고 하자.

  Case 1: `interest(`e`)`를 실행중에 예외상황이 발생하지 않고 정상 종료되면 그 결과과 위의 식의 결과가 된다.

  Case 2: `interest(`e`)`를 실행중에 분모값이 0인 경우가 발생하면, 모든 실행이 중단되고 곧바로 예외상황처리 식인 `interestzero(x)`를 계산하게 되고 그 결과가 위의 식의 결과가 된다. 이 때 `x`는 `try`-식의 환경에 있는 `x`여야한다.

- 또다른 예로,

```
let procedure bar(x) =
    ( let procedure k(n) = if x-n = 0 then raise else x+n
     in k(x) )
   procedure choo(x) = x+x
   procedure foo(x) = try bar(x+1) handle choo(x)
in write foo(10)
```

  위의 프로그램은 20을 프린트하게 된다.

이제, 위와 같이 정의된 예외상황을 SM5로 구동시키기 위해서, SM5를 확장해서 SM5eh를 고안하고, SM5eh로 번역하는 방안을 고안하라. SM5eh는 기존의 SM5를 그대로 놔두고 새로운 부품과 새로운 명령어를 첨가하는 것으로 정의하라. 번역 방법을 설명하고 디자인한 SM5eh의 정의를 리포트로 제출한다. □

Challenge 11 (3↑, 1↓) "예외검증기: 처리안되는 예외는 없슴"

다음과 같은 적극적인 계산법으로 실행되는 언어를 생각하자.

e	::=	x	variable
	|	$\lambda x.e$	function
	|	$e\ e$	application
	|	n	integer
	|	e + e	addition
	|	`if0` $e\ e\ e$	branch
	|	`raise`	exception raise
	|	e `handle` e	exception handling

"`raise`"는 예외상황을 발생시키는 명령이고, "e_1 `handle` e_2"는 e_1을 실행하다가 예외가 발생되면 e_2를 실행하게 된다. e_1의 실행중에 예외가 발생되지 않으면 e_1의 값이 전체 식의 결과값이 된다. "`if0`"식은 첫째 식의 값이 0이면 둘째 식을 계산하고 아니면 세째 식을 계산한다.

예를들어,

$$10 + 1 \texttt{ handle } 0$$

은 11을 계산한다.

$$((\lambda x.\texttt{if0}\ x\ \texttt{raise}\ 10)\ 0)\ \texttt{handle}\ 8$$

은 8을 계산한다.

$$((\lambda x.\texttt{if0}\ x\ (\texttt{raise handle}\ 1)\ 10)\ 0)\ \texttt{handle}\ 8$$

은 1을 계산한다.

$$(\lambda x.\texttt{if0}\ x\ 1\ raise)\ 1$$

은 발생한 예외를 처리하는 식이 없으므로, 실행이 값자기 멈춘다.

let-설탕을 써서 예를 더 들면,

```
let
  f = λx.if0 x raise 2
in
  ((f 0) handle λx.x) (f 1)
end
```

은 2를 계산한다.

```
let
  k = λf.f (λx.x)
in
  ((k λx.x) handle λx.x) (k λx.raise)
end
```

는 발생한 예외를 처리하지 못하고 실행이 값자기 멈춘다.

수업시간에 다룬 단순타입시스템(simple type system)을 확장하라. 그래서 확장된 타입 시스템을 통과한 프로그램은 실행중에 처리되지 않는 예외가 없다는 것이 보장되는. 그리고 그 타입 시스템을 안전하게 구현하는 알고리즘을 정의하라. 그래서 위에 예를 든 프로그램들에 대해서, 제대로 도는 프로그램들 네개중 적어도 세개는 받아들일 수 있어야 하고, 제대로 돌지 않는 프로그램들은 모두 받아들이지 말아야 한다. □

Challenge 12 (2↑, 1↓) "더 좋은 let-다형 타입 시스템"

수업시간에 강의한 대로, 다형 타입 추론규칙(let-polymorphic type inference rules)들은 메모리 반응을 일으키는 명령문이 있는 경우 다형타입을 만들 때 조심한다:

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \texttt{malloc}\, E : \tau\ loc}$$

$$\frac{\Gamma \vdash E : \tau\ loc}{\Gamma \vdash \texttt{!}E : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau\ loc \quad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 \texttt{ := } E_2 : \tau}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \texttt{ ; } E_2 : \tau_2}$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : GEN_\Gamma(\tau) \vdash E' : \tau}{\Gamma \vdash \texttt{let } x \texttt{ = } E \texttt{ in } E' : \tau}\ \neg expansive(E)$$

$$\frac{\Gamma \vdash E : \tau \quad \Gamma + x : \tau \vdash E' : \tau}{\Gamma \vdash \texttt{let } x \texttt{ = } E \texttt{ in } E' : \tau}\ expansive(E)$$

$$
\begin{array}{rcl}
expansive(n) & = & false \\
expansive(x) & = & false \\
expansive(\lambda x.E) & = & false \\
expansive(E_1\ E_2) & = & true \\
expansive(\texttt{let}\ x\ \texttt{=}\ E_1\ \texttt{in}\ E_2) & = & expansive(E_1) \vee expansive(E_2)
\end{array}
$$

- 위의 규칙보다 "더 좋은" 타입 추론 규칙을 디자인하고 더 좋은 이유를 설명하라.
- 새로 만든 규칙이 안전하다는 것을 증명하고,
- 기존의 방식으로는 타입체크가 되지 않지만 잘 도는 프로그램이 새로운 방식으로 타입체크되는 예를 보여야 한다.

□