Homework 8

SNU 4190.310, 2019 가을

Kwangkeun Yi

Due: 12/14(토), 24:00

이번 숙제의 목적은:

- M언어로 짜여진 프로그램이 무난히 실행될 수 있는지를 미리 확인해 주는 안전장치를 갖추어 보기.
- 즉, let-다형 타입 시스템(let-polymorphic type system)을 장착해서 편리하면서도 안전한 프로그래밍 환경을 완성해 보기. (프로그램으로 제출)
- 그리고 두 문제중 하나를 선택해서 해결하기:
  - 프로그램의 실행내용을 미리 예측하는 예측기를 고안해보기. (리포트로 제출)
  - 혹은, 프로그래밍 언어의 예외상황 같은 과격한 점프도 설탕임을 확인해보기. (프로그램으로 제출)

Exercise 1 (80pts) "저지방 고단백 M"

M 실행기 위에, let-다형 타입 시스템(let-polymorphic type system)을 장착하자. 예를들어, 아래와 같은 잘 도는 프로그램들을 생각하자. 단순 타입 시스템은 받아들이지 않는 프로그램들이다. 하지만 장착할 let-다형 타입 시스템은 모두 받아들여야 할 프로그램들이다.

TA가 제공하는 M 실행기의 틀 위에 let-다형 타입 시스템을 장착하라.

```
(* example 1: polymorphic toys *)

  let val I = fn x => x
```

```
      val add = fn x => x.1 + x.1
      val const = fn n => 10
  in
    I I;
    add(1, true) + add(2, "snu 310");
    const 1 + const true + const "kwangkeun yi"
  end

(* example 2: polymorphism with imperatives *)

  let val f = fn x => malloc x
  in
     let val a = f 10
         val b = f "pl"
         val c = f true
     in
       a := !a + 1;
       b := "hw7";
       c := !c or false
     end
  end

(* example 3: polymorphic swap *)

  let val swap =
          fn order_pair =>
             if (order_pair.1) (order_pair.2)
              then (order_pair.2)
              else (order_pair.2.2, order_pair.2.1)
  in
    swap(fn pair => pair.1 + 1 = pair.2, (1,2));
    swap(fn pair => pair.1 or pair.2, (true, false))
  end

(* S K I combinators *)
  let val I = fn x => x
      val K = fn x => fn y => x
      val S = fn x => fn y => fn z => (x z) (y z)
  in
   S (K (S I)) (S (K K) I) 1 (fn x => x+1)
  end
```

□

다음 두 문제중 하나를 선택해서 풀이를 제출하라.

Exercise 2 (60pts) "점프마저 설탕"

이번 숙제는 예외상황을 발생시키고 처리하는 설탕을 모두 녹이는 변환기를 만드는 것이다. 현대의 프로그래밍 언어의 많은 장치들은 대부분 설탕인데, 예외상황처리 장치들도 그렇다는 것을 확인해보자.

프로그래밍언어에서 goto/jump/break등은 실행순서를 변경시키는 명령이고, 이런 명령들을 모두 포섭하면서 그 이상으로 강력한 것이 예외상황처리(exception raise/handling)다.

아래의 언어가 출발어다. 적극적인 계산법(eager evaluation)으로 실행되는 언어다. 이전 숙제에서 다루었던 언어와 유사하다. 다른 점은 예외상황 발생(`raise`)과 처리(`handle`) 식이 첨가된 것이다.

e	::=	n	natural number
	|	x	identifier
	|	$\lambda x.e$	function
	|	$e\ e$	application
	|	`if` $e\ e\ e$	branch
	|	e `=` e	number equal check
	|	`raise` e	exception raise
	|	e `handle` $n\ e$	exception handling

예외상황처리식 "e_1 `handle` n e_2"은 식 e_1과 거의 같다. e_1을 계산하면서 상황에 따라 다르게 작동한다.

- e_1 계산이 정상적으로 끝나면 그 결과가 이 예외상황처리식의 결과가 된다. e_2는 예외상황이 발생할 경우를 위해 준비해 놓은 식(handler)이다. 예외상황이 발생하지 않았으므로 실행되지 않는다.
- e_1 계산중에 예외상황이 발생하면, 정상 진행을 멈추고 곧바로 현재의 예외상황 처리식으로 껑충 점프한다. 그리고 준비해 놓은 처리식(handler)이 처리할 수 있는 지 검토후 처리한다.
  - 그 예외상황 값이 n이면 현재의 처리식이 처리할 수 있다. 즉, 처리식(handler) e_2를 계산하고 그 결과가 현재의 예외상황 처리식의 결과가 된다.
  - 그 예외상황 값이 n이 아니면, 현재의 처리식으로는 그 예외상황을 처리할 수 없다. 이 경우 현재의 예외상황 처리식을 실행 순으로 감

싸고 있는 바로 이전의 예외상황 처리식으로 다시 껑충 점프해서 위의 과정을 반복한다. 이 과정으로 예외상황 처리식을 찾지못하면 프로그램 실행은 갑작스럽게 멈춘다.

예외상황을 발생시키는 식은 "`raise` e"이다. 우선 e를 계산한다. 결과는 자연수이어야 한다. 그 자연수 값을 가지고 예외상황을 발생시킨다. 그러면 정상적인 진행이 멈춰지고 이 예외상황을 처리할 처리식을 찾아 껑충껑충 점프해간다.

예를들어, 함수 `f`가 다음과 같이 정의되어 있다고 하자.

$$\texttt{f() = 1 + (}E\ \texttt{handle 99 0)} \tag{1}$$

다음과 같은 식을 생각하자

$$\texttt{f() handle 77 10} \tag{2}$$

식 (2)를 실행하자. 함수 `f()`가 실행된다. 그러면 함수 내부(식 (1))에서 E가 실행된다. 그 안에서 예외상황 `raise 77`이 발생했다고 하자. E를 감쌌던 첫번째 처리문(식 (1))으로 껑충 점프한다. 그 처리문은 예외값 99 만을 처리할 수 있을 뿐이다. 따라서 발생한 예외상황은 처리되지 못하고 다음 처리식으로 다시 껑충 점프한다. 실행순서에서 현재의 처리문을 감싸고 있던 처리식으로(식 (2)). 그 곳에 이르러서야 발생한 예외상황 77이 처리될 수 있다. 그래서 식 (2)의 실행값은 10이 된다.

숙제는, 위의 언어로 작성된 프로그램을 받아서 `raise`와 `handle`이 사라진 식으로 변환하는 함수:

```
removeExn: xexp -> xexp
```

를 정의하는 것이다. 변환된 결과 $\underline{e}$는 e와 같은 일을 해야 한다. 위 언어의 실행기 `run`으로 다음과 같이 실행시켜서 두 결과가 같아야 한다:

$$\texttt{run(}e\texttt{)} \;=\; \texttt{run(removeExn }e\texttt{)}$$

변환할 프로그램은 항상 자연수를 최종적으로 계산하는 프로그램으로 한정한다. 조교는 위 언어의 실행기 `run`과 `mexp`의 파서를 제공할 것이다.

```
type xexp = Num of int
          | Var of string
          | Fn of string * xexp
          | App of xexp * xexp
```

```
  | If of xexp * xexp * xexp
  | Equal of xexp * xexp
  | Raise of xexp
  | Handle of xexp * int * xexp
```

□

Exercise 3 (60pts) "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 숙제는 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 생각하자. 수업에서 다룬 언어의 일부다.

e	::=	n	integer
	|	x	identifier
	|	$\lambda x.e$	function
	|	$f\ \lambda x.e$	recursive function
	|	$e\ e$	application
	|	$\texttt{ifz}\ e\ e\ e$	branch
	|	$e\ \texttt{+}\ e$	addtion

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

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

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

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 재귀적으로 정의한 일부는 다음과 같다. 빈칸과 나머지 경우를 모두 정의해서 제출하라.

$$\frac{}{n_i \vdash X_i \supseteq \emptyset} \qquad \frac{}{x_i \vdash X_i \supseteq \boxed{\cdots}}$$

$$\frac{e_k \vdash E}{(\lambda x.e_k)_i \vdash (X_i \supseteq \{\lambda x.e_k\}) \ \wedge \ E} \qquad \frac{e_k \vdash E}{(f\ \lambda x.e_k)_i \vdash (X_i \supseteq \{\lambda x.e_k\}) \ \wedge \ \boxed{\cdots} \ \wedge \ E}$$

$$\frac{e_m \vdash E_1 \qquad e_n \vdash E_2}{(e_m\ e_n)_i \vdash (X_i \supseteq \boxed{\cdots}) \ \wedge \ E_1 \ \wedge \ E_2}$$

- 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($X_i \supseteq setexp_i$) 집합에서 부터 시작해서 새롭게 알게되는 방정식들을 계속 추가하는 것이다.

더 이상 추가할 게 없을 때 까지. 어떻게 추가하면 될까? 아래 규칙에 따라 추가한다.

$$\frac{\mathcal{X} \supseteq \mathcal{Y} \qquad \mathcal{Y} \supseteq \{\lambda x.e\}}{\mathcal{X} \supseteq \{\lambda x.e\}} \qquad\qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \qquad \mathcal{Y} \supseteq \emptyset}{\mathcal{X} \supseteq \emptyset}$$

$$\frac{\mathcal{X} \supseteq \mathcal{Y} \cup \mathcal{Z}}{\mathcal{X} \supseteq \mathcal{Y} \qquad \mathcal{X} \supseteq \mathcal{Z}} \qquad\qquad \frac{\mathcal{X} \supseteq \mathcal{Y} \,@\, \mathcal{Z} \qquad \mathcal{Y} \supseteq \{\lambda x.e_i\}}{X_x \supseteq \mathcal{Z} \qquad \mathcal{X} \supseteq X_i}$$

첫 세개의 규칙은 알게 된 사실을 단순히 전파하거나 잘게 녹이는 규칙이다. 네번째 규칙은 함수호출식의 경우 알게 된 사실을 전파하는 것이다. $\mathcal{X}$가 $\mathcal{Y}$-호출-$\mathcal{Z}$의 결과를 가지게 되고 ($\mathcal{X} \supseteq \mathcal{Y} \,@\, \mathcal{Z}$), $\mathcal{Y}$가 어떤 함수인지를 알게되면 ($\mathcal{Y} \supseteq \{\lambda x.e_i\}$), 함수 인자 X_x는 $\mathcal{Z}$을 가지게 되고 ($X_x \supseteq \mathcal{Z}$) 호출식 결과 $\mathcal{X}$는 그 함수의 몸통식 e_i의 결과 X_i를 가지게 된다 ($\mathcal{X} \supseteq X_i$).

- 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. □