Homework 7

SNU 4190.310, 2026 봄

Kwangkeun Yi

Due: 5/30(Sat), 24:00

이번 숙제의 목적은:

- 프로그램 식마다 마저 할 일을 프로그램 식으로 드러내는 변환기를 만들어 보기.
- 예외상황 처리(하던일 관두고 복귀하기)가 위의 마저할일 변환과 유사한 변환을 통해서 녹여 없앨 수 있는 설탕구조라는 것을 확인하기.
- 람다계산법의 중력권에서 디자인된 언어의 실행기 완성하기.
- 그 언어의 실행기 앞에, 믿을만한 단순 타입 시스템(sound simple type system)을 장착하기.

Exercise 1 (20pts) "마저 할 일 드러내기"

프로그램의 의미구조(실행과정)를 살펴보면, 매 식마다 그 식의 계산이 끝나면 마저 할 일이 무엇인지 정해져 있다. 예를 들어 다음의 덧셈식을 생각해 보자:

$$e_1 + e_2$$

식 e_1과 e_2의 계산이 끝나면 마저 할 일은 무엇인가? 두 결과값을 가져다가 더하는 것이다. 이 할 일을 함수로 표현할 수 있다. "두 결과값을 가져다가"는 함수의 인자로 받는 것으로 표현할 수 있고, 함수 내부에서는 이 인자를 가지고 마저 할 일을 표현하면 된다. 이런 함수 정도가 되지 않을까:

$$\lambda v_2.v_1 + v_2$$

여기서 v_1은 식 e_1의 결과를 받은 인자다. 또는, 더한 결과를 가지고 마저 할일이 함수 κ로 정해져 있다면, 이런 함수 정도일 것이다:

$$\lambda v_2.\kappa(v_1 + v_2)$$

일반적으로 각 식마다 그 식이 계산된 후 마저 할 일은 함수로 표현할 수 있다. 그 식의 결과를 인자로 받는 함수다. 이런 함수가 그 식이 계산된 후 마저 할 일이다.

이제 프로그램의 모든 식마다 그런 함수 – "마저할일(continuation)" – 를 생각하면서 실행순서를 드러낼 수 있다. 모든 식들을 변환해서, 각 식이 계산을 끝내고 진행해야 할 "마저할일"을 그 식의 인자로 받아서 계산을 진행하는 형태로 꾸밀 수 있다.

이런 변환을 "마저할일 변환(CPS(continuation-passing-style) transformation)"이라고 한다.[1] 원래 식을 e, 마저할일 변환된 식을 $\underline{e}$라고 쓰고 두 식의 차이를 타입 레벨에서 보이면 다음과 같다:

$$e : \mathit{Int}$$
$$\underline{e} : (\mathit{Int} \to \mathit{Result}) \to \mathit{Result}$$

원래식의 결과가 정수(Int)라면, 변환된 식은 정수를 받아서 최종 결과로 보내는 마저할일함수($\mathit{Int} \to \mathit{Reult}$ 타입의 함수)를 받아서 최종 결과(Result)를 내놓는 식이 된다.

위의 덧셈식을 마저할일 변환하는 규칙은 다음과 같다:

$$\underline{e_1 + e_2} = \lambda\kappa.\ \underline{e_1}(\lambda v_1.\ \underline{e_2}(\lambda v_2.\ \kappa(v_1 + v_2)))$$

위에서 인자로 받는 κ가 덧셈식이 끝나고 마저 할 일을 표현한 함수(continuation)이고, $\underline{e_1}$에 전달되는

$$\lambda v_1.\underline{e_2}(\lambda v_2.\kappa(v_1 + v_2))$$

은 e_1이 실행된 후 그 결과를 v_1에 받아 마저 할 일이고, $\underline{e_2}$에 전달되는

$$\lambda v_2.\kappa(v_1 + v_2)$$

은 e_2가 실행된 후 그 결과를 v_2에 받아 마저 할 일이다.

따라서, 프로그램 e를 마저할일 변환한 결과 $\underline{e}$에 마지막으로 마저 할 일($\lambda v.v$,

[1] 마저할일 변환을 하고 나면, 프로그램 텍스트에 프로그램의 각 식들이 계산되는 순서가 드러나게 된다.

즉 "더이상 일 없음")를 던저주면 원래의 식이 계산하는 결과를 내놓게 된다:

$$e \quad \equiv \quad \underline{e}(\lambda v.v)$$

아래의 적극적인(eager-evaluation, or call-by-value) 프로그래밍 언어를 생각하자. 수업에서 다룬 언어의 일부분이다.

e	::=	z	integer
	|	id	identifier
	|	`fn` id `=>` e	function
	|	`rec` id id `=>` e	recursive function
	|	$e\ e$	application
	|	`ifp` $e\ e\ e$	branch on positive
	|	e `+` e	addition
	|	`(`e`,`e`)`	pair
	|	e`.1`	first component
	|	e`.2`	second component

위의 언어로 짜여진 프로그램을 마저할일 변환하는 변환기 함수:

```
cps: nexp -> nexp
```

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

$$\texttt{run}(e) \ = \ \texttt{run}(\underline{e}(\texttt{fn v => v}))$$

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

```
type nexp = Num of int
          | Var of string
          | Fn of string * nexp
          | Fnr of string * string * nexp
          | App of nexp * nexp
          | Ifp of nexp * nexp * nexp
          | Add of nexp * nexp
```

```
        | Pair of nexp * nexp
        | Fst of nexp
        | Snd of nexp
```

□

Exercise 2 (30pts) "예외처리(하던 일 관두고 복귀하기)는 설탕"

위에서 다룬 언어에서 예외상황을 발생시키고 처리하는 식을 더한 언어를 생각하자:

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & \texttt{raise}\ e & \text{raise exn with a value} \\
 & | & e\ \texttt{handle}\ id\ e & \text{exn handing}
\end{array}
$$

예외상황 발생식 raise e는 해왔던 일을 멈추고 e를 계산한 값을 가지고 복귀한다. 어디로? 실행중 가장 최근에 장착된 예외처리기 식 "e_1 handle id e_2"의 e_2식으로 복귀한다. 즉, e_1 계산중에 그 예외상황이 발생한 것이다. 이 때 id는 예외상황이 발생할 때 계산한 값이 이곳으로 점프해와서 붙는 이름이고 그 유효범위는 e_2이다.

위 프로그램식을 받아서 예외상황 처리식을 제거하는 변환기 함수:

```
xcps: xexp -> xexp
```

를 정의하라. 변환된 결과 $\underline{e}$는 e와 타입이 다음과 같이 다르다

$$
\begin{array}{l}
e : \tau \\
\underline{e} : (\tau \to \mathit{Result}) \times (\tau' \to \mathit{Result}) \to \mathit{Result}
\end{array}
$$

변환된 식은 두 개의, 마저할일함수(continuation)을 받는다. 첫번째 함수는 정상적인 마저 할 일이다. 두번째 함수는, 예외상황이 발생했을 때 앞으로 할 일이다.

변환된 결과 $\underline{e}$는 e와 같은 일을 해야 한다. 위 언어의 실행기 xrun으로 다음과 같이 실행시켜서 두 결과가 같아야 한다:

$$\texttt{xrun}(e) \;=\; \texttt{xrun}(\underline{e}(\texttt{fn v => v}, \texttt{fn v => print ``uncaught exn''}))$$

변환할 프로그램은 항상 정수를 최종적으로 계산하는 프로그램으로 한정한다. 조교는 위 언어의 실행기 xrun과 xexp의 파서를 제공할 것이다.

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

```
| Fnr of string * string * xexp
| App of xexp * xexp
| Ifp of xexp * xexp * xexp
| Add of xexp * xexp
| Pair of xexp * xexp
| Fst of xexp
| Snd of xexp
| Raise of xexp
| Handle of xexp * string * xexp
```

□

Exercise 3 (20pts) "M 실행기"

수업시간에 구축해간 프로그래밍 언어에 약간의 간을 한 언어가 M 이다. M 언어의 실행기의 대부분이 제공될 것이다. 첨부된 M의 정의에 그 언어의 의미에 대한 모든 것이 있다. 이에 기초해서 나머지를 완성하라. □

Exercise 4 (40pts) "저지방 M"

수업시간에 구축해간 프로그래밍 언어에 약간의 간을 한 언어가 M 이다. 위에서 제작한 M 실행기 위에, 수업시간에 확인한 안전한 단순 타입 시스템(simple type system)을 설치하라.

그래서 M 실행기 여기저기 쌓여있는 내장지방을 제거하라. 실행중 타입 첵크(dynamic type check)로 시간을 지연시키는 기름기. 즉, 실행기 내부에서 식들이 계산하는 값들의 타입을 첵크하는 부분이 많이 사라질 수 있다.[2]

즉, 첨부되는 M의 정의에서 타입 시스템의 미흡한 부분을 완성하고, 그것의 충실한 타입 유추기를 구현해서 M 실행기 앞단에 장착하도록 한다. 그러한 M 실행기는 항상 잘 도는 건강한 프로그램만 실행하게 된다. 그래서 건강해진 밥상, M 실행기를 즐길 수 있기를.

맛점하세요! Bon appétit! Help yourself! 请吃好! □

[2] 그렇게 제거하고 나면, 실행기에서 값종류를 체크하는 부분(패턴식)이 완전하지 않다고 OCaml 컴파일러는 불평할 것이다. 그러나, M실행기 앞단에 장착한 타입검사가 안전하다면 M실행기 소스에 대한 OCaml 컴파일러의 그런 불평은 안심하고 무시해도 될 것이다.