Programming Language(Fall 2015) Challenge

2009-11841 최영진

December 22, 2015

Challenge 1 증명 완성 I

$$
\begin{aligned}
expression\ e \rightarrow\ & n \\
& \mid x \\
& \mid \lambda x.e \\
& \mid e\ e \\
& \mid e\ +\ e
\end{aligned}
$$

Progress

$\vdash e : \tau$ 이고 e 가 값이 아니면 반드시 진행 $e \rightarrow e'$ 한다. (P)

1. $e\ =\ n$ 인 경우, e가 값이므로 P는 참이다.
2. $e\ =\ x$ 인 경우, 다른 케이스로 치환되므로 다른 케이스가 모두 맞다면 참이다.
3. $e\ =\ \lambda x.e$ 인 경우, e가 값이므로 P는 참이다.
4. $e\ =\ e_1\ e_2$ 인 경우, $\vdash\ e_1\ e_2 : \tau$ 이므로 타입추론 규칙에 의해 $\vdash\ e_1 : \tau' \rightarrow \tau$ 이고 $\vdash\ e_2 : \tau'$ 이다. 따라서 귀납 가정에 의해서,
   (a) e_1이 값이 아니면 진행 $e_1 \rightarrow e_1'$하고, 이는 곧 프로그램 실행 $\rightarrow$의 정의에 의해 $(e_1\ e_2) \rightarrow (e_1'\ e_2)$ 과 같다.
   (b) 마찬가지로 e_1이 값이고 e_2가 값이 아니라면 진행 $e_2 \rightarrow e_2'$하고, 이는 곧 프로그램 실행 $\rightarrow$의 정의에 의해 $(e_1\ e_2) \rightarrow (e_1\ e_2')$과 같다.
   (c) e_1과 e_2가 모두 값이라면, $\vdash e_1 : \tau' \rightarrow \tau$ 일 수 있는 값 e_1은 오직 $\lambda x.e'$의 경우 뿐이다. 따라서 프로그램 실행 $\rightarrow$의 정의에 의해 $e_1\ e_2\ =\ (\lambda x.e')\ e_2 \rightarrow \{e_2/x\}\ e'$으로 진행한다.
5. $e\ =\ e_1\ +\ e_2$인 경우, 타입의 정의에 따라 $\vdash e_1 : \iota$ 이고, $\vdash e_2 : \iota$이다.
   (a) e_1이 값이 아니면 진행 $e_1 \rightarrow e_1'$하고, 이는 곧 프로그램 실행 $\rightarrow$의 정의에 의해 $e_1 + e_2 \rightarrow e_1' + e_2$ 과 같다.
   (b) 마찬가지로 e_1이 값이고 e_2가 값이 아니라면 진행 $e_2 \rightarrow e_2'$하고, 이는 곧 프로그램 실행 $\rightarrow$의 정의에 의해 $e_1 + e_2 \rightarrow e_1 + e_2'$과 같다.
   (c) e_1과 e_2가 모두 값이면 즉, ι 타입을 가지는 값이면, 이 둘을 더한 n을 값으로 가지는 e'으로 진행한다.

Preservation

$\vdash e = \tau$이고 $e \to e'$이면 $\vdash e' : \tau$.

1. $e = n$ 인 경우, e가 값이므로 $\to$로 진행하지 않는다.

2. $e = x$ 인 경우, 다른 케이스로 치환되므로 다른 케이스가 모두 맞다면 참이다. 치환이 타입을 보존하는 것은 아래의 "Preservation under substitution"에서 증명한다.

3. $e = \lambda x.e_1$ 인 경우, e가 값이므로 $\to$로 진행하지 않는다.

4. $e = e_1\ e_2$ 인 경우, $\vdash\ e_1\ e_2 : \tau$ 이므로 타입추론 규칙에 의해 $\vdash\ e_1 : \tau' \to \tau$ 이고 $\vdash\ e_2 : \tau'$ 이다. $e_1 e_2 \to e'$이라면 세 가지 경우밖에 없다.

   (a) $e_1 \to e_1'$이라서 $(e_1\ e_2) \to (e_1'\ e_2)$ 인 경우. 귀납 가정에 의해 $\vdash e_1' : \tau' \to \tau$. $\vdash e_2 : \tau'$ 이므로, 타입추론 규칙에 의해 $\vdash e_1' e_2 : \tau$.

   (b) e_1은 값이고 $e_2 \to e_2'$이라서 $(e_1\ e_2) \to (e_1\ e_2')$인 경우. $\vdash e_1 : \tau' \to \tau$이고, 귀납 가정에 의해 $\vdash e_2 : \tau'$이므로, 타입추론 규칙에 의해 $\vdash e_1 e_2' : \tau$.

   (c) e_1과 e_2가 모두 값이라면, $\vdash e_1 : \tau' \to \tau$인 값 e_1은 타입추론 규칙에 의해 $e_1 = \lambda x.e'$밖에는 없다. 즉 $e_1 e_2 = (\lambda x.e')v$이고, $(\lambda x.e')v \to \{v/x\}e'$이다. $\vdash \lambda x.e' : \tau' - > \tau$이라면 타입추론 규칙에 의해 $x : \tau' \vdash e' : \tau$이다. $\vdash v : \tau'$이므로, "Preservation under substitution(아래에서 증명)"에 의해 $\vdash \{v/x\}e' : \tau$이다.

5. $e = e_1 + e_2$인 경우, 타입의 정의에 따라 $\vdash e_1 : \iota$ 이고, $\vdash e_2 : \iota$이다.

   (a) e_1이 값이 아니라면 $e_1 \to e_1'$과 같이 진행하고(Progress), 귀납가정에 의해 $\vdash e_1' : \iota$이다. 따라서 타입추론 규칙에 의해 $\vdash e_1' + e_2 : \tau$이다.

   (b) e_1이 값이고 e_2가 값이 아니라면 $e_2 \to e_2'$과 같이 진행하고(Progress), 귀납가정에 의해 $\vdash e_2' : \iota$이다. 따라서 타입추론 규칙에 의해 $\vdash e_1 + e_2' : \tau$이다.

   (c) e_1과 e_2가 모두 값이면 즉, ι 타입을 가지는 값이면, 이 둘을 더한 n을 값으로 가지는 e'으로 진행한다. $e = e_1 + e_2$에서 τ는 ι이고, 이 n는 ι 타입이므로 $\vdash e' : \tau$이다.

Preservation under Substitution

$\Gamma \vdash v = \tau'$이고 $\Gamma + x : \tau' \vdash e : \tau$이면 $\Gamma \vdash \{v/x\}e : \tau$.

1. $e = n$ 인 경우, 치환할 것이 없으므로 성립한다.

2. $e = x$ 인 경우, x가 다른 것으로 치환되는데, 귀납 가정에 의해 치환은 타입을 보존하므로 다른 케이스들로 회귀한다. 따라서 다른 케이스들이 모두 맞다면 성립한다.

3. $e = \lambda x.e$ 인 경우, 항상 $y \notin \{x\} \cup FV v$인 $\lambda y.e'$로 간주할 수 있으므로 $\{v/x\}\lambda y.e' = \lambda y.\{v/x\}e'$. 따라서, 보일 것은 $\Gamma \vdash \lambda y.\{v/x\}e' : \tau$. (단 τ는 $\tau_1 \to \tau_2$로 설정)
   가정 $\Gamma + x : \tau' \vdash \lambda y.e' : \tau_1 \to \tau_2$으로부터 타입추론 규칙에 의해 $\Gamma + x : \tau' + y : \tau_1 \vdash e' : \tau_2$ 이고, $\Gamma \vdash v : \tau'$와 $y \notin FV(v)$ 으로부터 $\Gamma + y : \tau_1 \vdash v\tau'$ 이므로, 귀납 가정에 의해 $\Gamma + y : \tau_1 \vdash \{v/x\}e' : \tau_2$. 즉, 타입추론 규칙에 의해 $\Gamma \vdash \lambda y.\{v/x\}e' : \tau_1 \to \tau_2$.

4. $e = e_1\ e_2$ 인 경우, $\vdash\ e_1 : \tau_1$ 및 $\vdash e_2 : \tau_2$라고 하자. $\{v/x\}e = \{v/x\}e_1\{v/x\}e_2$이고 귀납 가정에 의해 $\vdash \{v/x\}e_1 : \tau_1$ 이고 $\vdash \{v/x\}e_2 : \tau_2$ 이다. 따라서 타입추론 규칙에 의해 $\Gamma \vdash \{v/x\}e_1\{v/x\}e_2 : \tau_1\tau_2$ 이고, 이는 보존된 타입 τ이다.

5. $e = e_1 + e_2$인 경우, 위와 같이 $\vdash\ e_1 : \tau_1$ 및 $\vdash e_2 : \tau_2$라고 하자. $\{v/x\}e = \{v/x\}e_1 + \{v/x\}e_2$ 이고 귀납 가정에 의해 $\vdash \{v/x\}e_1 : \tau_1$ 이고 $\vdash \{v/x\}e_2 : \tau_2$ 이다. 따라서 타입추론 규칙에 의해 $\Gamma \vdash \{v/x\}e_1 + \{v/x\}e_2 : \tau$이다.

Challenge 4 재귀호출의 비용

1. 끝재귀호출(Tail-recursive call)

Tail-recursive call의 정의는 재귀호출 후 할 일이 아무것도 없다는 뜻이다. 그렇다면 함수 호출 시점의 E는 더 이상 쓸 일이 없을 것이고, 원래 K에 저장하던 E를 저장할 필요가 없다. 함수의 E'로 교체하면서, 기존의 C는 (예전처럼 (C, E)로 저장할 필요가 없으므로) 함수의 C'에 붙이면 된다. 다만 이 모든 동작은 Tail-recursive function에서만 가능하다.

2. 첨가한 명령어

Tail-recursive call을 뜻하는 명령어 trcall 을 다음과 같이 정의한다. 다른 모든 부분은 기존 SM5와 같다.

$(l :: v :: (x, C', E') :: S, M, E, trcall :: C, K) = (S, M\{l \leftarrow v\}, (x, l) :: E', C'@C, K)$

3. 새로운 trans함수의 정의

어떤 함수가 Tail-recursive call인지 판단하려면 다음 두 가지 경우를 조사하면 된다.
1) 변환한 함수가 call로 끝나는 경우
2) 변환한 함수가 jtr로 끝나고, 분기 대상 중 적어도 하나가 call로 끝나는 경우
따라서 변환한 후 위와 같은 경우 call을 trcall로 바꿔주면 된다. trans의 LETF는 다음과 같이 바뀐다. (숙제 5번 모범답안 코드 참조)

```
| K.LETF (f, x, e1, e2) ->
(* trans function body *)
let tranced = trans e1 in
let last_cmd = List.hd (List.rev tranced) in
(* check the function is tl *)
let is_tl =
let rec is_last_call cmd =
match cmd with
| K.CALL -> true
| K.JTR(c1, c2) -> is_last_call(c1) || is_last_call(c2)
| _ -> false
in
is_last_call last_cmd in
(* if not tl, then just use CALL, else replace CALL with TRCALL *)
let body =
let rec replace cmdlist =
match cmdlist with
| [] -> []
| hd::tl ->
(match hd with
| K.CALL -> K.TRCALL :: (replace tl)
| _ -> hd :: (replace tl)
)
in
if not is_tl
then tranced
else replace tranced
in
(* then put it to legacy logic *)
let proc_body = (Sm5.BIND f :: body) @ [Sm5.UNBIND; Sm5.POP] in
[Sm5.PUSH (Sm5.Fn (x, proc_body)); Sm5.BIND f] @
trans e2 @
[Sm5.UNBIND; Sm5.POP]
```

Challenge 6 더 좋은 let-다형 타입 시스템

let-다형 타입 시스템의 문제는, 커버할 수 있는 범위가 기대하는 것보다 작다는 것이다. 다음 프로그램을 보자.

$$\begin{array}{l}\texttt{let}\ f\ =\ \texttt{malloc}\ (\lambda x.x)\\ \texttt{in}\\ \quad f\ \texttt{:=}\ (\lambda x.x+1)\ \texttt{;}\\ \quad (\texttt{!}f)\ \texttt{true}\end{array}$$

in 안에서 let에서 정의된 주소에 대입을 할 경우, 애초에 정의된 타입과 달라져서 런타임에 타입 에러가 발생할 수 있다. 이 문제를 해결하기 위해 let-다형 타입 시스템에서는 expansive 라는 추가 검증 방식을 도입하였다. 그러나 expansive는 APP이나 MALLOC인 경우 무조건 true를 리턴하기 때문에 안전(sound)하지만 불완전(uncomplete)한 부분이 크다.

따라서 다음과 같은 타입 시스템을 제안한다.
숙제 8 "람다 예보"의 연립방정식 풀이를 통해 어떤 함수호출식에서 호출될 수 있는 함수들의 집합을 구할 수 있다. 이 방식으로 APP에 들어올 수 있는 함수들을 구한 뒤, 그 함수가 generalize하기 안전(sound)하다면 정상적으로 타입 체킹을 하면 된다. 이와 같은 방식으로 다음 프로그램을 타입 체크할 수 있다.

$$\begin{array}{l}\texttt{let}\ f\ =\ (\lambda y.y)\ (\lambda x.x)\\ \texttt{in}\\ \quad (\texttt{!}f)\ \texttt{true ;}\\ \quad (\texttt{!}f)\ \texttt{1}\end{array}$$

위 프로그램은 *let E*1 *in E*2의 *E*1부분에 (e_1e_2) 형태의 APP expression이 있기 때문에 기존의 타입 시스템에서는 타입 체킹이 불가능하다. 그러나 실행은 잘 되는 프로그램이다. 람다 예보를 통해 오직 Identity function만 통과시키도록 하여도 위와 같은 프로그램은 처리할 수 있다.

Challenge 9 메모리는 설탕

참고자료

15년도 챌린지가 나오기 전에 14년도 챌린지 (http://ropas.snu.ac.kr/ kwang/4190.310/14/challenge.pdf) 를 봤는데, 14년도의 이 문제에는 다음 3번 질문, 답변이 있었습니다.

3. 그리고 나면, ref e, e := e, !e는 어떻게 표현?
프로그램 식 e가 변환된 것을 $\underline{e}$로 표현하면, 다음과 같이 메모리 설탕 을 녹여낼 수 있을 것이다:

$$\underline{\text{ref } e} = \lambda(v,(c,S)).\text{let } (v',(c',S')) = \underline{e}(v,(c,S)) \text{ in } (c',(c'+1,(c',v')::S'))$$

그런데 15년도 챌린지 문서에는 이 내용이 빠진 것을 봤습니다. 이미 읽어본 터라 3번 질의응답을 아는 상태로 문제를 풀었는데, 혹시 문제가 될까봐 말씀드립니다.

메모리 반응식 풀이

CPS 과제와 마찬가지로 e가 변환된 것을 $\underline{e}$로 표현하였으므로, 메모리 반응식 뿐 아니라 언어의 모든 expression에 대해 $\underline{e}$를 귀납적으로 정의해야 한다. $\underline{e}$는 이전 식의 값과 메모리 표현(counter * list of (주소 * 값))을 받아 새로운 값과 메모리 표현을 내놓는 식이다.

$$\underline{x} \quad = \quad \lambda(v,(c,S)).(x,(c,S))$$

$$
\begin{array}{rcl}
\underline{\lambda x.e} & = & \lambda(v,(c,S)). \\
 & & (\lambda x.\lambda y.(\underline{e}\ y),(c,S)) \\
\underline{e\ e'} & = & \lambda(v,(c,S)). \\
 & & \text{let } (v',(c',S')) = \underline{e}(v,(c,S)) \text{ in} \\
 & & \text{let } (v'',(c'',S'')) = \underline{e'}(v',(c',S')) \text{ in} \\
 & & (v'\ v''\ (v'',(c'',S''))) \\
\underline{\text{ref } e} & = & \lambda(v,(c,S)).\text{let } (v',(c',S')) = \underline{e}(v,(c,S)) \text{ in } (c',(c'+1,(c',v')::S')) \\
\underline{e := e'} & = & \lambda(v,(c,S)). \\
 & & \text{let } (v',(c',S')) = \underline{e}(v,(c,S)) \text{ in} \\
 & & \text{let } (v'',(c'',S'')) = \underline{e'}(v',(c',S')) \text{ in} \\
 & & (v'',(c''+1,(v',v'')::S'')) \\
\underline{!e} & = & \lambda(v,(c,S)). \\
 & & \text{let } (v',(c',S')) = \underline{e}(v,(c,S)) \text{ in} \\
 & & \text{let rec } f\ x\ S = \text{ match S with} \\
 & & |(l,v)::S'->\text{if } x=l \text{ then } v \text{ else } (f\ x\ S') \\
 & & |[]->\text{unbound error} \\
 & & \text{in} \\
 & & (f\ v'\ S',(c',S'))
\end{array}
$$

위와 같이 정의함으로써 메모리 반응식(ref, :=, !e)을 다른 식의 조합으로 표현할 수 있다. !e의 표현에는 ML의 문법인 match with 및 리스트 관련 값들을 사용할 수 있다고 가정하였는데, 이 역시 설탕으로 기본 식만으로 쓸 수 있을 것이다.