Challenge Set

SNU 4190.310, Fall 2015

이 광근

Due: 12/18(Fri), 16:00 to IN box, Rm428 Bldg302

올바른 답을 제출한 사람은 최종 성적이 상승합니다.

Challenge 1 (1단계 상승) "증명 완성 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 (2단계 상승) "증명 완성 II-M"
강의 홈페이지의 강의 슬라이드 `5-1simple-type.pdf`에는 단순 타입시스템의 온라인 알고리즘들 M과 W가 있다. 이 알고리즘이 단순 타입 시스템의 충실한 구현임(강의 슬라이드에 명시한 성질)을 증명하라. □

Challenge 3 (1단계 상승) "재귀함수 = 함수 + 메모리 지정문"
단순 타입 시스템(simple type system)을 갖춘 다음의 언어를 생각하자. 적

극적인 계산법(eager evaluation)으로 실행되는 언어이다.

$$
\begin{array}{rcll}
e & ::= & n & \text{integer} \\
& | & x & \text{variable} \\
& | & \lambda x.e & \text{function} \\
& | & e\ e & \text{application} \\
& | & \texttt{if0}\ e\ e\ e & \text{application} \\
& | & e\texttt{+}e & \text{addition} \\
& | & \texttt{ref}\ e & \text{memory allocation} \\
& | & e\ \texttt{:=}\ e & \text{assignment} \\
& | & \texttt{!}\ e & \text{dereference}
\end{array}
$$

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("xx")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 타입을 가진 위의 언어로 재귀함수를 흉내낼 수 있는 방법은 없는 것일까? 가능하다. 힌트는 메모리 반응식을 이용하는 것이다. 한 예로 재귀함수 factorial를 메모리 반응식들을 이용해서 어떻게 흉내낼 수 있을 지를 보여라. 제안한 방법은 단순 타입 시스템을 통과할 수 있어야 한다. □

Challenge 4 (1단계 상승) "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."

이번 숙제에서는, 위 학생의 이야기가 무슨 뜻인지, SM5 프로그램의 실행 과정을 관찰하면서 알아보고, 재귀함수호출이 가지는 비용을 줄이는 개선책을 찾아나서보자. 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 때 마다, K (continuation) 파트에 함수가 끝나고 계속 진행할 내용을 쌓아갔다:

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

  그리고 K에 기록된 "계속할 일"은 함수가 끝나면 복원해 주었다:

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

  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 함수같은 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 앞으로 할 일이란 것은 다시 return하는 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. 재귀호출 한 함수를 끝내는 것 이외에는. 이러한 재귀호출을 끝재귀호출(<u>tail-recursive call</u>)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, `Sm5x`에서 첨가한 명령어의 의미정의를 Exercise 1에 있는 꼴로 표현한 것과, 새로운 `trans`함수의 정의이다. 코드에는 첨가된 명령어들의 정의를 코멘트로 잘 설명하도록 한다.

□

Challenge 5 (3단계 상승) “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,

$\star$ 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 숙제에서는 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]

$$
\begin{array}{rcll}
e & ::= & n & \text{integer} \\
 & | & x & \text{variable} \\
 & | & \texttt{\{}x\ \texttt{:=}\ e\texttt{\}} & \text{record} \\
 & | & \texttt{\{\}} & \text{nil record} \\
 & | & e.x & \text{record field} \\
 & | & x\texttt{:=}e & \text{assignment} \\
 & | & e\texttt{;}e & \text{sequence} \\
 & | & \texttt{let}\ x\ e\ e & \text{local block}
\end{array}
$$

$$
\begin{array}{rclcll}
x & \in & \mathit{Var} & & & \text{variables} \\
v & \in & \mathit{Val} & = & \mathit{Num} + \mathit{Record} & \text{values} \\
n & \in & \mathit{Num} & & & \text{numbers} \\
l & \in & \mathit{Loc} & & & \text{locations} \\
r & \in & \mathit{Record} & = & \mathit{Var} \stackrel{\text{fin}}{\to} \mathit{Loc} & \text{records} \\
\sigma & \in & \mathit{Env} & = & \mathit{Var} \stackrel{\text{fin}}{\to} \mathit{Loc} & \text{environments} \\
M & \in & \mathit{Memory} & = & \mathit{Loc} \stackrel{\text{fin}}{\to} \mathit{Val} & \text{memories}
\end{array}
$$

Notation: $A \stackrel{\text{fin}}{\to} B$ 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 $\mathit{dom} f$ 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] $$\frac{\sigma, M \vdash e \Rightarrow v, M\text{'} \qquad \sigma(x) = l}{\sigma, M \vdash x \texttt{ := } e \Rightarrow v, M\text{'}[v/l]}$$

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

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

Definition 2 *$reach(\sigma, M)$ 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:*

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

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 6 (2단계 상승) "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. 이때, 새로 만든 규칙이 안전하다는 것을 증명할 필요는 없지만, 기존의 방식으로는 타입체크가 되지 않지만 잘 도는 프로그램이 새로운 방식으로 타입체크되는 예를 보여야 한다.□

Challenge 7 (2단계 상승) "K-- + exception"

숙제 5, "SM5"의 확장이다. 우선 K--가 확장되었다. 예외상황처리(exception handling)가 가능하도록:

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

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 정상종료되면 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 8 (2단계 상승) "처리안된 예외는 없슴"

다음과 같은 적극적인 계산법으로 실행되는 언어를 생각하자.

$$
\begin{array}{rclll}
e & ::= & x & & \text{variable} \\
 & | & \lambda x.e & & \text{function} \\
 & | & e\, e & & \text{application} \\
 & | & n & & \text{integer} \\
 & | & e \mathtt{+} e & & \text{addition} \\
 & | & \mathtt{if0}\, e\, e\, e & & \text{branch} \\
 & | & \mathtt{raise} & & \text{exception raise} \\
 & | & e\, \mathtt{handle}\, e & & \text{exception handling}
\end{array}
$$

"`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,

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

은 11을 계산한다.

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

은 8을 계산한다.

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

은 1을 계산한다.

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

은 발생한 예외를 처리하는 식이 없으므로, 실행이 갑자기 멈춘다.

let-설탕을 써서 예를 더 들면,

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

은 2를 계산한다.

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

는 발생한 예외를 처리하지 못하고 실행이 갑자기 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 9 (2단계 상승) "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.

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & \lambda x.e & \text{function} \\
 & | & e\,e & \text{application} \\
 & | & \texttt{ref}\ e & \text{malloc} \\
 & | & e \texttt{ := } e & \text{assignment} \\
 & | & \texttt{!}e & \text{dereference}
\end{array}
$$

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 여러 가지가 있을 수 있다. 아주 기본적은 방법으로는 CPS-변환(continuation-passing-style transformation)이 있다.

2답 : 많은 방법이 있을 수 있다. 예를 들어, 짝으로:

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

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

프로그램 식 e의 메모리 설탕을 녹이는 룰을 찾아라. □