Homework 7

SNU 4190.310, 2019 가을

Kwangkeun Yi

Due: 11/28(Wed), 24:00

이번 숙제의 목적은:

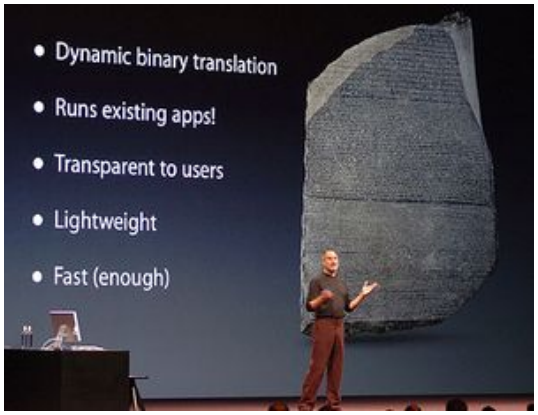
- 애플사에서 자랑했던 하위언어 자동번역 기술을 손수 이뤄보기. (특별한 기술이 아님을 겪고 자신감을 가지기)
- 프로그램식의 실행순서를 프로그램에 드러내는 변환기를 만들어 보기.
- 람다계산법의 중력권에서 디자인된 언어의 실행기 완성하기.
- 그 언어의 실행기 앞에, 믿을만한 단순 타입 시스템(simple type system)을 장착하기.

Exercise 1 (50pts) "SM5 Rozetta"

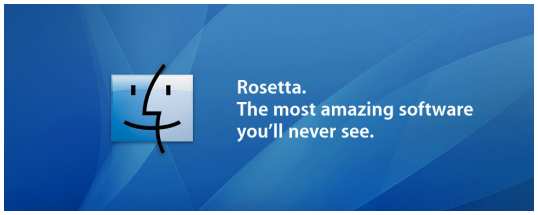
오늘날과 같은 Apple사의 발전에 핵심역할을 한 소프트웨어 중 하나가 기계어 번역(binary translation) 기술[1]이었다. 2005년 애플이 PowerPC 프로세서에서 Intel 프로세서로 갈아타는 선언을 한다. 같은 해 6월에 Apple's World Wide

[1] 정확하게는 "실행중 기계어 번역"(dynamic binary translation) 기술.

Developer Conference에서 잡스가 Rosetta라는 소프트웨어 기술을 소개한다:



소비자가 새로운 애플제품을 구매해도 PowerPC에서 작동하던 예전 소프트웨어를 아무 문제 없이 쓸 수 있게 해주는 기술이었다.



이 Rosetta 소프트웨어의 엔진은 애플이 "Transitive"라는 회사

TRANSITIVE

로부터 구입한 기계어 번역기였다. 당시 Transitive사는 영국 맨체스터에 본사를 둔 아주 작은 벤쳐회사 였다.[2]

이번 숙제는 이러한 번역기를 SM5에 대해서 만들어보는 것이다. SM5 프로그램을 Sonata 프로그램으로 번역하는 번역기이다.

Sonata가 SM5와 다른 점은 다음과 같다. 이외에는 모두 같다.

[2]Transitive사는 그 후 IBM에 인수됩니다. 당연히 돈방석에 앉았겠지요, 그 회사 연구원들과 설립자인 멘체스터 대학의 Alasdair Rawsthorne교수등.

- Sonata는 SM5 부품 중에서 K("continuation") 부품이 없다:

$$(S, M, E, C).$$

따라서, SM5에서 K부품을 건드렸던 `call` 명령의 의미와 빈명령문의 의미는 Sonata에서는 다음과 같다:

$$\begin{array}{rllll} & (l :: v :: (x, C', E') :: S, & M, & E, & \texttt{call} :: C) \\ \Rightarrow & (S, & M\{l \mapsto v\}, & (x, l) :: E', & C') \end{array}$$

$$\begin{array}{rllll} & (S, & M, & E, & \mathit{empty}) \\ \Rightarrow & (S, & M, & E, & \mathit{empty}) \end{array}$$

- 메모리는 스택에 저장할 수 있는 값들을 모두 저장할 수 있다. 즉,

$$\begin{array}{rcrcl} S & \in & \mathit{Stack} & = & \mathit{Svalue\ list} \\ M & \in & \mathit{Memory} & = & \mathit{Loc} \to \mathit{Svalue} \end{array}$$

여러분이 고안할 것은, 문제없이 돌아가는 SM5 코드를 받아서 Sonata 코드로 변환하는 함수

```
rozetta: Sm5.command -> Sonata.command
```

를 작성하는 것입니다. 모듈 `Sm5`와 모듈 `Sonata`는 제공됩니다. □

Exercise 2 (40pts) "앞으로 할 일 드러내기"

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

$$e_1 + e_2$$

식 e_1과 e_2의 계산이 끝나면 앞으로 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 앞으로 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 할일이 함수 κ

로 정해져 있다면, 이런 함수 정도일 것이다:

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

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

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

이런 변환을 "CPS(continuation-passing-style)-변환"이라고 한다.[3] 원래 식을 e, CPS-변환된 식을 $\underline{e}$라고 쓰고 두 식의 차이를 타입 레벨에서 보이면 다음과 같다:

$$\begin{aligned} e &: \mathit{Int} \\ \underline{e} &: (\mathit{Int} \to \mathit{Result}) \to \mathit{Result} \end{aligned}$$

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

위의 덧셈식을 CPS-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 앞으로 해야 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 해야할 일이고, $\underline{e_2}$에 전달되는

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

은 e_2가 실행된 후 해야할 일이다.

따라서, 프로그램 e를 CPS-변환한 결과 $\underline{e}$에 최종 할 일($\lambda v.v$)를 던저주면 원래

[3] CPS-변환을 하고 나면, 프로그램 텍스트에 프로그램의 각 식들이 계산되는 순서가 드러나게 된다.

의 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	::=	n	natural number
	|	id	identifier
	|	`fn` id `=>` e	function
	|	$e\,e$	application
	|	`rec` id id `=>` e	recursive function
	|	`ifzero` $e\,e\,e$	branch
	|	e+e	infix binary operation
	|	`(`e`,`e`)`	pair
	|	e`.1`	first component
	|	e`.2`	second component

위의 언어로 짜여진 프로그램을 CPS-변환하는 변환기 함수:

```
cps: mexp -> mexp
```

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

```
type mexp = Num of int
          | Var of string
          | Fn of string * mexp
          | App of mexp * mexp
          | Rec of string * string * mexp
          | Ifz of mexp * mexp * mexp
          | Add of mexp * mexp
```

```
| Pair of mexp * mexp
| Fst of mexp
| Snd of mexp
```

□

Exercise 3 (30pts) "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 (50pts) "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.[4]

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! □

[4] 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.