CPS로 바뀐 모듈과 CPS로 바뀌지 않은 모듈의 연동

Interoperation between CPS modules and Non-CPS modules

김정택, 이광근
전자전산학과 전산학전공
KAIST

요 약

지금까지 CPS 변환을 사용하기 위해서는 프로그램 전체에 적용해야 하는 것으로 알려져 왔다. CPS 변환을 하면 좋지않은 부분까지 바꾸어야 하기 때문에 CPS 변환 후에 적용해야 하는 유용한 기술들을 사용하지 힘든 경우도 있었다.

이 논문에서는 CPS 변환을 프로그램의 일부분에만 적용하는 방법을 제시하여 CPS 변환을 필요한 프로그램 부분에만 적용하고 나머지 부분과는 서로 연동을 통해서 접속할 수 있도록 한다.

이 연동 방법은 프로그램의 타입정보를 이용하여 정의되며, 프로그램 전체의 의미를 변화시키지 않는다.

1 문제제기

지금까지 CPS 변환을 사용할 경우 프로그램 전체에 적용을 해야 되는 것으로 알려져 왔다. 이로 인해 CPS 변환을 했을 때 유용한 기술과 CPS 변환을 하지 않았을 때 유용한 기술을 한 프로그램에 동시에 사용할 수 없었다.

이 논문에서는 프로그램의 일부 모듈만을 CPS 변환할 때 나머지 변환되지 않은 모듈과 서로 연동할 수 있는 방법을 제안하고 있다. 이 방법을 사용함으로서 CPS 변환을 적합한 모듈만을 CPS 변환함으로서 CPS 변환이 적합하지 않은 부분까지 바꿀 필요가 없게 된다.

2 사용되는 언어

2.1 개략적인 구문구조

이 논문에서 다루는 언어는 ML의 일부로, ML의 핵심부분이라고 할 수 있는 부분이다. 그리고, 매개변수를 전달할 때 값으로 계산한 후에 전달하고 함수를 값처럼 사용하여 매개변수로 넘겨주거나 함수의 결과 값으로 사용할 수 있다.

이 언어의 개략적인 구문구조를 나타내면 다음과 같다. (여기에서 κ는 특정 상수 값들을 생성할 수 있는 생성자를 나타낸다.)

e	::=	$\mathtt{1}$	기본 상수
		x	변수
		$\lambda x.\, e$	함수
		$\mathtt{fix}\, f\, \lambda x.\, e$	재귀 호출 함수
		$e_1\, e_2$	함수 호출
		$\mathtt{con}\, \kappa\, e$	상수 생성
		$\mathtt{decon}\, e$	상수의 인자 추출
		$\mathtt{case}\, e_1\, \kappa\, e_2\, e_3$	선택적 실행

"$\lambda x.\, e$"는 함수의 내용은 e이고 매개변수는 x인 함수를 나타낸다. "$\texttt{fix}\, f\, \lambda x.\, e$"는 f라는 이름을 가진 재귀호출이 가능한 함수이다. "$e_1\, e_2$"는 e_1을 계산해서 얻은 함수의 인자로 e_2의 값을 적용하여 함수를 호출한다.

e의 값을 계산하면 v가 된다고 할때 상수 값인 $\kappa \cdot v$는 "$\texttt{con}\, \kappa\, e$"를 계산하면 얻을 수 있다. 이와는 대칭적으로 상수 값인 $\kappa \cdot v$가 e를 계산하여 얻어질때 "$\texttt{decon}\, e$"를 이용해서 v 값을 얻을 수 있다.

"$\texttt{case}\, e_1\, \kappa\, e_2\, e_3$"를 이용하면 조건에 따라 계산을 다르게 할 수 있다. 먼저 e_1의 값을 계산한다. e_1의 값이 $\kappa \cdot v$가 되면, e_2의 값을 계산하여 전체 case 문의 값을 e_2의 값으로 한다. 그렇지 않은 경우에는 e_3의 값을 계산하여 전체 case 문의 값을 e_3의 값으로 한다.

2.2 동작으로 기술되는 프로그램의 의미

앞의 언어의 의미구조를 나타내기 위해서 구조적 동작의미구조(structural operational semantics [3])[1]를 사용한다. 그중에서도 Felleisen의 계산문맥방법(evaluation context [1])[2]을 사용하였다.

우선, 계산이 모두 끝난 것을 의미하는 것으로 값을 나타내는 v를 정의한다.

v	::=	$\texttt{1}$	기본 상수
	|	$\lambda x.\, e$	함수
	|	$\texttt{fix}\, f\, \lambda x.\, e$	재귀 호출 함수
	|	$\kappa \cdot v$	v를 인자로 가지는 상수

그리고, 코드 표현(expression)을 위의 값을 이용하여 확장한다.

계산할 부분의 프로그램 문맥을 나타내는 C 는 다음과 나타낼 수 있다.

C	::=	$[\,]$	hole
	|	$\texttt{con}\, \kappa\, C$	
	|	$\texttt{decon}\, C$	
	|	$C\, e$	
	|	$v\, C$	
	|	$\texttt{case}\, C\, \kappa\, e_1\, e_2$	

이 프로그램 문맥은 왼쪽에서 오른쪽의 순서대로 계산을 하고 함수를 호출할 때는 인자를 모두 계산하도록 되어 있다. 일반적으로 쓰이는 것과 같이 $C[e]$라고 나타내면 C에 있는 구멍인 $[]$를 e로 채워 넣은 것을 의미한다. 이 프로그램 문맥을 이용해서 임의의 코드 표현에 대한 계산 법칙을 정의한다.

$$\frac{e \to e'}{C[e] \;\mapsto\; C[e']}$$

계산할 부분인 e에 대한 한단계의 계산인 $e \to e'$는 그림 1에 정의되어 있다. 일반적인 경우와 같이 $[v/x]e$와 같은 코드 표현은 e에서 값이 결정되지 않은 변수 x의 값을 모두 v로 바꾼 새로운 코드 표현을 나타낸다.

이제 전체 프로그램의 의미구조에 대한 정의를 다음과 같이 할 수 있다.

정의 1 *정의되지 않은 변수가 없는 코드 표현 e의 의미구조는 계산 단계들의 연속된 나열로 정의된다.*

$$e \;\mapsto\; e_1 \;\mapsto\; e_2 \;\mapsto\; \cdots$$

위의 연속된 나열이 더이상 계산되지 않는 v라는 값으로 끝나는 경우 다음과 같이 나타낸다.

$$e \;\overset{*}{\mapsto}\; v$$

[1] 구문 구조에 맞추어 동작을 기술하는 의미구조

[2] 계산할 부분의 프로그램 문맥을 이용한 방법

$$
\begin{array}{rcll}
\texttt{con}\,\kappa\,v & \to & \kappa{\cdot}v & \\
\texttt{decon}\,\kappa{\cdot}v & \to & v & \\
(\lambda x.\,e)\,v & \to & [v/x]e & \\
(\texttt{fix}\,f\,\lambda x.\,e)\,v & \to & [v/x][\texttt{fix}\,f\,\lambda x.\,e/f]e & \\
\texttt{case}\,\kappa{\cdot}v\,\kappa\,e_1\,e_2 & \to & e_1 & \\
\texttt{case}\,\kappa{\cdot}v\,\kappa'\,e_1\,e_2 & \to & e_2 & (\kappa' \neq \kappa)
\end{array}
$$

그림 1: 계산 단계들의 정의

$$
\begin{array}{rcl}
T(1) & = & \lambda K.\,K\,(1) \\
T(x) & = & \lambda K.\,K\,(x) \\
T(\texttt{con}\,\kappa\,e) & = & \lambda K.\,T(e)\,(\lambda v.\,K\,(\texttt{con}\,\kappa\,v)) \\
T(\texttt{decon}\,e) & = & \lambda K.\,T(e)\,(\lambda v.\,K\,(\texttt{decon}\,v)) \\
T(\lambda x.\,e) & = & \lambda K.\,K\,(\lambda x.\,T(e)) \\
T(\texttt{fix}\,f\,\lambda x.\,e) & = & \lambda K.\,K\,(\texttt{fix}\,f\,\lambda x.\,T(e)) \\
T(e_1\,e_2) & = & \lambda K.\,T(e_1)\,(\lambda f.\,T(e_2)\,\lambda v.\,f\,v\,K) \\
T(\texttt{case}\,e_1\,\kappa\,e_2\,e_3) & = & \lambda K.\,T(e_1)\,(\lambda v.\,\texttt{case}\,v\,\kappa\,(T(e_2)\,K)\,(T(e_3)\,K))
\end{array}
$$

그림 2: CPS 변환 함수 T

3 CPS 변환

CPS 변환은 프로그램의 모든 코드 표현들을 나중의 할 일을 받는 함수 형태로 바꾸어 주는 방법이다. 또한 모든 코드 표현들이 자신의 결과 값을 넘겨 받은 나중의 할 일에 전달하도록 변환된다.

위의 언어에 대한 일반적인 CPS 변환을 그림 2와 같이 나타낼 수 있다 [2].

4 변환방식이 다른 모듈간의 연동

서로 변환방식이 다른 모듈간에 연동을 하기 위해서는 다음과 같은 일을 해주어야 한다.

- 첫째로 변환된 프로그램 부분을 변환되지 않은 곳에서 사용할 수 있도록 해주어야 한다. 즉, 변환되지 않은 곳에서 사용 할 수 있는 값들에 대해서는 외부에서 사용할 수 있는 형태로 만들어서 넘겨주어야 한다.
- 둘째로 변환되지 않은 부분을 변환된 부분에서 사용할 수 있어야 한다. 변환되지 않은 부분의 값을 사용할 때는 변환된 부분에서 사용할 수 있는 형태로 바꾸어서 사용해야 한다.
- 위의 두 경우에 함수값의 형태를 바꾸어 사용할 때는 각 함수의 인자도 적절히 바꾸어서 사용해야 한다.

각 부분의 타입정보를 이용하여 위의 조건에 맞는 변환 함수를 작성할 수 있다.

CPS 방식의 값을 Non-CPS 모듈에서 사용하도록 하는 함수를 *Marshal* 이라고 하고 Non-CPS 방식의 값을 CPS 모듈에서 사용하도록 하는 함수를 *Unmarshal* 이라고 하면 이 두 변환 함수를 다음과 같이 상호 재귀적으로 정의 될 수 있다.

Marshal : (non-CPS variables × non-CPS types) → CPS expressions

$$
\begin{aligned}
&\mathit{Marshal}(x, \iota) = x \\
&\mathit{Marshal}(x, \tau_1 \to \tau_2) = \\
&\qquad \lambda y_{cps}.\ \lambda K.\ (K\ \mathit{Marshal}(x\ (\mathit{Unmarshal}(y_{cps}, \tau_1)), \tau_2)) \\
&\mathit{Marshal}(x, \tau) = \\
&\qquad (\mathtt{fix}\, f\ \lambda y.\ \mathtt{case}\, y\, \kappa_1\ (\mathtt{con}\, \kappa_1\ (f\ (\mathtt{decon}\, y))) \\
&\qquad\qquad\qquad \mid \kappa_2\ (\mathtt{con}\, \kappa_2\ (\mathit{Marshal}(\mathtt{decon}\, y, \tau_1))))\ x
\end{aligned}
$$

이때 τ 는 일반적으로 다음의 재귀적은 상수 생성자라고 할 수 있다.
$\kappa_1 : \tau \to \tau$ and $\kappa_2 : \tau_1 \to \tau$.

Unmarshal : (CPS variables × non-CPS types) → non-CPS expressions

$$
\begin{aligned}
&\mathit{Unmarshal}(x_{cps}, \iota) = x_{cps} \\
&\mathit{Unmarshal}(x_{cps}, \tau_1 \to \tau_2) = \\
&\qquad \lambda y.\ \mathit{Unmarshal}(x_{cps}\ (\mathit{Marshal}(y, \tau_1))\ (\lambda v.\ v), \tau_2) \\
&\mathit{Unmarshal}(x_{cps}, t_1) = \\
&\qquad (\mathtt{fix}\, f\ \lambda y.\ \mathtt{case}\, y\, \kappa_1\ (\mathtt{con}\, \kappa_1\ (f\ (\mathtt{decon}\, y))) \\
&\qquad\qquad\qquad \mid \kappa_2\ (\mathtt{con}\, \kappa_2\ (\mathit{Unmarshal}(\mathtt{decon}\, y, \tau_1)))\ x_{cps}
\end{aligned}
$$

이때 τ 는 일반적으로 다음의 재귀적은 상수 생성자라고 할 수 있다.
$\kappa_1 : \tau \to \tau$ and $\kappa_2 : \tau_1 \to \tau$

5 결론

이 논문에서는 CPS 변환을 프로그램의 일부에만 적용하고 나머지 부분과 서로 연동할 수 있는 방법을 제시하였다.

이를 이용하면 프로그램의 CPS 변환이 필요한 부분만을 바꾸어 줌으로서 다른 부분까지 바꿀때 발생하는 불필요한 성능 저하를 막을 수 있다.

이때 사용한 CPS 변환은 일반적인 값을 이용한 호출을 사용한 언어에 해당하는 변환이지만, 다른 특별한 CPS 변환에도 쉽게 적용할 수 있다.

참고 문헌

[1] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[2] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[3] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.