예외상황 분석을 이용한 계산과정 전달 변환
(Continuation Passing Style Transformation after Exception Analysis)

김 정 택 † 이 광 근 ††
(Jung-Taek Kim) (Kwangkeun Yi)

요 약 이 논문의 목적은 ML 프로그램의 소스(source)를 수정하여 ML의 예외상황 처리기의 수행 속도를 개선하고자 하는 것이다. ML은 함수를 값으로 주고받을 수 있으며 타입을 이용하여 프로그램을 검사해 주는 언어이다. 이러한 ML의 예외상황 처리부분을 사용함으로서 프로그래머는 쉽게 자신의 프로그램의 예외적인 동작을 기술할 수 있다. 하지만, 이러한 예외상황을 처리하기 위해서는 많은 계산이 필요하기 때문에, 예외상황을 처리하는 프로그램 부분이 병목 현상을 일으키는 경우가 많다.

프로그램의 소스를 바꾸어서 예외상황 처리부분이 존재하지 않는 같은 동작을 하는 다른 프로그램으로 바꾸는 방법은 이미 알려져 있지만, 도리어 수행시간이 느려진다. 위의 바꾸는 방법은 "나중에 할 일을 넘겨주는 방식(Continuation Passing Style)"이라고 부르는 방식으로 프로그램의 소스를 바꾸는 방법을 조금 수정하여, 예외상황을 처리하는 부분을 위의 "나중에 할 일(continuation)"과 같은 방식으로 넘겨주어 예외상황 처리부분이 모두 사라지게 된다. 그러나, 이러한 방식은 모든 프로그램내 표현(expression)을 모두 위와 같은 방식으로 바꾸기 때문에, 이로 인해 발생하는 계산이 예외상황을 처리하는 계산보다 더 많아지게 된다.

이 논문에서는 이러한 단점을 개선하여 프로그램내에서 예외상황 처리부분을 없애는데 꼭 필요한 표현만을 정적분석을 사용하여 골라내어 이를 선택적으로 변환하는 방법을 사용한다.

Abstract ML's exception handling makes it possible to describe exceptional execution flows conveniently. Sometimes, current implementation of exception handling introduces unnecessary overhead. Our goal is to reduce this overhead by source-level transformation.

To this end, we transform source programs into variant of continuation-passing style(CPS), replacing handle and raise expressions by continuation-catching and throwing expressions, respectively. CPS-transforming every expression, however, introduces a new cost.

We therefore use an exception analysis to transform expressions selectively: if an expression is statically determined to involve exceptions then it is CPS-transformed; otherwise, it is left in direct style.

In this article, we formalize this selective CPS transformation, prove its correctness, and present possible improvement for our transformation.

1. 문제 제기

1.1 ML의 예외상황

ML의 예외상황 처리부분은 프로그래머에게 프로그램의 예외적인 동작을 쉽게 기술할 수 있게 해주는 편리한 도구이다. 예외상황 처리부분의 도움으로 프로그래머는 자신의 프로그램을 더 간단하고 읽기 쉽게 작성할 수 있다. ML의 예외상황 처리부분은 새로 추가된 두개의 언어 구문으로 이루어져 있다: 예외상황이 발생했을 때 프로그램이 어떻게 행동해야 하는지를 나타내기 위해서 handle구문을 사용한다. 그리고 예외적인 상황이 발생했다는 것을 나타내기 위해서 raise 구문을 사용한다. handle구문과 raise구문의 쌍에 의해서 예외적인 동작에 대한 적절한 처리를 하는 것이 가능하다.

그러나 예외상황이 발생하면 handle구문이 실행할 때

† 비 회 원 : 한국과학기술원 전자전산학과
jtkim@pllab.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전자전산학과 교수
kwang@cs.kaist.ac.kr
논문접수 : 1999년 7월 12일
심사완료 : 1999년 12월 9일

의 상황으로 되돌아가야 하기 때문에 ML에서의 예외상황은 비용이 많이 드는 부분이다. handle구문이 재귀 호출 함수 안에 있는 경우와 같이 프로그램의 실행 상황을 자주 바꾸어야 하는 경우가 발생하면 프로그램의 성능에 큰 영향을 미칠 수 있다.

1.2 CPS 변환

예외상황 처리부분은 CPS(나중에 할 일을 넘겨주는 방식, Continuation Passing Style) 변환을 이용해서도 나타낼 수 있다. 두 개의 남은 할 일(continuation)들 - 정상적인 수행과정을 위한 것과 예외적인 수행과정을 위한 것 - 을 이용하도록 CPS 변환을 확장하면 예외상황 처리기들은 두 번째의 남은 할 일로 나타낼 수 있다.

변환을 한 후에는 프로그램에서 모든 handle구문과 raise구문이 모두 사라지게 된다. 그러므로 변환된 프로그램은 예외상황을 사용하지 않는 방법으로 수행 할 수 있게 된다.

이렇게 예외상황을 처리하기 위해서 두개의 남은 할 일을 사용하는 것은 새로운 생각은 아니다. 예를 들면 Appel은 "compiling with continuations" [1]라는 자신의 책에서 언급하고 있다. 하지만, Appel이 만든 컴파일러가 CPS 변환을 사용함에도 불구하고 raise구문과 handle구문을 없애는 방법을 사용하지 않고 두 구문을 위한 낮은 수준의 연산자인 sethdlr과 gethdlr를 사용하여 구현하였다. 이는 Biagioni등 [2, Figure 2]이 한 것과 같이 모든 함수가 두 개의 남은 할 일을 넘겨주도록 하는 것은 비용 면에서 효율적이지 않기 때문인 것으로 추측된다.

1.3 예외상황 분석기

예외상황 분석기 [13]는 프로그램이 예외상황과 관련된 수행에 대한 정보를 준다. 이 분석기를 이용하면 각 코드 표현이 발생시킬 수 있는 예외상황에 대한 정보를 얻을 수 있다. 또한 함수의 흐름에 대한 정보도 얻을 수 있다.

이러한 두 가지 정보의 도움으로 프로그램의 임의의 코드 표현에 대해서 다음과 같은 질문에 답을 얻을 수 있다 : 코드 표현이 예외상황 처리기에 대한 정보를 필요로 하는가?

6 절에 있는 정보 표시 규칙으로 어떻게 하면 위의 질문에 답을 얻을 수 있는 지를 나타내었다.

2. 해결 방법

이 논문에서 제안하는 것은 두 개의 남은 할 일을 사용하는 CPS 변환을 비용 면에서 효율적으로 구현하는 방법이다. 예외상황 분석에서 얻을 수 있는 정적 분석 정보를 가지고 남은 할 일을 넘겨주는 일을 실제로 필요할 때만 하도록 한다. 이 논문에서는 이러한 작업을 수행하기 위해서 예외상황 분석에서 얻어지는 정적 분석 정보를 기반으로 하여 남은 할 일이 실제로 필요할 때만 생성되도록 하는 선택적 CPS 변환을 제안한다. 이 논문의 선택적 CPS 변환은 두 개의 남은 할 일을 사용하는 CPS 변환(예외상황을 사용하는 코드 표현을 위한)과 아무 처리를 하지 않는 변환(예외상황을 사용하지 않는 코드 표현을 위한)을 모두 일반화시킨 것으로 생각할 수 있다.

이 새로운 CPS 변환을 이용하면 이전의 CPS 변환에 예외상황 분석기의 정보를 적용하여 예외상황이 포함된 프로그램을 컴파일 하는 새로운 방법이 가능하다. 이러한 새로운 변환은 다음과 같은 여러 가지 분야에 유용하다 :

- 특정 ML 컴파일러에서 이 논문의 변환은 몇 가지 프로그램을 더 빠르게 바꾸어 준다. 예를 들어, SML/NJ 컴파일러에서 몇 개의 예외상황을 많이 사용하는 프로그램의 실행 시간을 줄일 수 있다.
- CPS 변환을 이용하여 컴파일된 프로그램 구성 부분들을 다른 기술을 사용하여 컴파일된 프로그램 구성 부분들과 연결할 수 있는 방법을 제공한다. 이 논문의 선택적인 CPS 변환은 원래 프로그램의 특정 부분만을 변환하기 때문에 CPS 변환으로 컴파일 되지 않은 다른 프로그램 구성 부분들을 추가하는 것이 쉽다. 따라서 CPS 변환 기법을 사용하기 위해서 프로그램 전체를 변환해야 할 필요가 없어 졌다.

마지막으로 이 논문은 제안한 선택적 CPS 변환이 원래의 의미를 손상시키지 않는 다는 것을 증명하고 있다.

3. 관련 연구와 개괄

3.1 관련 연구

정적 분석 정보를 이용하여 선택적 변환을 하는 연구는 세 가지를 찾을 수 있었다. 엄격성 분석(strictness analysis)[1] [5]과 전체성 분석(totality analysis)[2] [6]과 바인딩 당시 분석(binding-time analysis)[3] [3]이 있다. 하지만 이 선택적 CPS 변환들은 실제의 컴파일러

1) 함수 입력의 수행 종결여부와 함수의 수행 종결여부의 관계에 대한 분석
2) 각 함수가 모든 입력에 대해서 수행을 종료하고 결과를 출력하는지에 대한 분석
3) 실제의 값이 결정되고 나서 하는 분석

에 장착되지 못했다. 마지막 변환의 경우만이 부분 수행기(partial evaluator)[4]에 장착되었을 뿐이다 [7]. Steckler와 Wand는 "선택적 계산과정 준비(selective thunkification)"에 대한 연구에서 엄격성 분석(strictness analysis)을 이용하여 값호출방식의 프로그램을 이름호출방식의 프로그램으로 바꾸는 과정에서 계산과정 준비(thunk)의 개수를 줄이는 비슷한 방식을 사용하였다 [9, 12].

3.2 개괄

이 논문의 나머지 부분은 다음과 같이 구성되어 있다. 4 절에서는 이 논문에서 대상으로 하는 언어의 구문구조와 의미 구조를 정의한다. 대상으로 하는 언어는 ML의 핵심 부분의 일부분이다. 5 절에서는 무조건 모든 코드 표현에 남은 할 일들을 넘겨주는 단순한 CPS 변환을 설명한다. 6 절에서는 이 논문에서 제시하는 선택적 CPS 변환에서 변환할 대상이 되는 코드 표현들을 골라내어 표시하는 정보 표시 과정을 정의한다. 7 절에서는 정보 표시의 정보를 기반으로 하여 코드 표현에 대한 CPS 변환을 선택적으로 적용하는 선택적 CPS 변환을 제안하고 설명한다. 8 절에서는 제안한 선택적 변환 방법 안전함을 증명한다. 선택적 변환 방법이 프로그램의 의미를 바꾸지 않는 다는 것을 증명함으로서 안전함을 증명할 것이다. 9 절에서는 프로그램을 구성 부분으로 나누어서 프로그램할 때에 발생하는 문제와 그 밖의 다른 문제들에 대해서 다룬다.

4. 사용되는 언어

4.1 개략적인 구문구조

이 논문에서 다루는 언어는 ML의 일부로, ML의 핵심부분에 예외상황 처리부분을 추가한 것이다. 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.)

$e ::=$	1	기본 상수
	x	변수
	$\lambda x.e$	함수
	fix $f\ \lambda x.e$	재귀 호출 함수
	$e_1\ e_2$	함수 호출
	con $\kappa\ e$	상수 생성
	decon e	상수의 인자 추출
	case $e_1\ \kappa\ e_2\ e_3$	선택적 실행
	handle $e_1\ \kappa\ \lambda x.e_2$	예외상황 처리기
	raise e	예외상황 발생

"$\lambda x.e$"는 함수의 내용은 e이고 매개변수는 x인 함수를 나타낸다. "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 할 때 상수 값인 $\kappa \cdot v$는 "con $\kappa\ e$"를 계산하면 얻을 수 있다. 이와는 대칭적으로 상수 값인 $\kappa \cdot v$가 e를 계산하여 얻어질 때 "decon e"를 이용해서 v값을 얻을 수 있다.

"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 case문의 값을 e_2의 값으로 한다. 그렇지 않은 경우에는 e_3의 값을 계산하여 전체 case문의 값을 e_3의 값으로 한다.

예외상황 발생시키는 "raise e"를 계산할 때는 e를 먼저 계산하여 그 값이 $\kappa \cdot v$가 된다고 하면 $\underline{\kappa \cdot v}$라는 예외상황이 발생하게 된다.

예외상황을 처리하는 "handle $e_1\ \kappa\ \lambda x.e_2$"를 계산할 때는 먼저 e_1을 계산하고 예외상황의 발생여부에 따라서 실행을 달리한다. e_1을 계산하는 도중에 $\underline{\kappa \cdot v}$라는 예외상황이 발생하면, 이 예외상황을 나타내는 값인 $\kappa \cdot v$를 인자로 하여 $\lambda x.e_2$의 값을 계산하고 그 값을 전체 handle문의 값으로 한다. 그렇지 않은 경우에는 e_1을 계산한 값이 전체 handle문의 값이 된다.

증명을 간단하게 하기 위해서 새로운 데이타타입을 선언하는 것과 문자열과 메모리 연산을 제외하였다. 실제로는 ML의 완전한 핵심 언어에 대해서 작업을 하였다. 그리고 아직은 위의 언어의 타입에 대해서는 고려하고 있지 않다.

4.2 동작으로 기술되는 프로그램의 의미

앞의 언어의 의미구조를 나타내기 위해서 구조적 동작의미구조(structural operational semantics [11])[5]를 사용한다. 그 중에서도 Felleisen의 계산문맥방법(evaluation context [8])[6]을 사용하였다. 우선, 계산이 모두 끝난 것을 의미하는 것으로 값을 나타내는 v와 발생한 예외상황을 나타내는 p를 정의한다.

$v ::=$	1	기본 상수
	$\lambda x.e$	함수
	fix $\lambda x.e$	재귀 호출 함수
	$\kappa \cdot v$	v를 인자로 가지는 상수
$p ::=$	$\underline{\kappa \cdot v}$	발생한 예외상황

그리고, 코드 표현을 위의 값과 발생한 예외상황을

4) 실행하지 않고도 값이 결정되는 부분을 알아내는 부분 수행기

5) 구문 구조에 맞추어 동작을 기술하는 의미구조

6) 계산할 부분의 프로그램 문맥을 이용한 방법

이용하여 확장한다.

계산할 부분의 프로그램 문맥을 나타내는 C는 다음과 나타낼 수 있다.

$$
\begin{array}{lll}
C ::= & [\,] & \text{구멍} \\
| & \mathrm{con}\ x\ C & \\
| & \mathrm{decon}\ C & \\
| & C\ e & \\
| & v\ C & \\
| & \mathrm{case}\ C\ x\ e_1\ e_2 & \\
| & \mathrm{handle}\ C\ x\ \lambda x.e & \\
| & \mathrm{raise}\ C &
\end{array}
$$

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.

$$\frac{e \rightarrow e'}{C[e] \rightarrow C[e']}$$

정상적인 계산 단계들의 정의:

$$
\begin{array}{rcl}
\mathtt{con}\,\kappa\,v & \rightarrow & \kappa \cdot v \\
\mathtt{decon}\,\kappa \cdot v & \rightarrow & v \\
(\lambda x.\,e)\,v & \rightarrow & [v/x]e \\
(\mathtt{fix}\,f\,\lambda x.\,e)\,v & \rightarrow & [v/x][\mathtt{fix}\,f\,\lambda x.\,e/f]e \\
\mathtt{case}\,\kappa \cdot v\,\kappa\,e_1\,e_2 & \rightarrow & e_1 \\
\mathtt{case}\,\kappa \cdot v\,\kappa'\,e_1\,e_2 & \rightarrow & e_2 \quad (\kappa' \neq \kappa) \\
\mathtt{handle}\,v\,\kappa\,\lambda x.\,e & \rightarrow & v
\end{array}
$$

예외적인 계산 단계들의 정의:

$$
\begin{array}{rcl}
\mathtt{raise}\,\kappa \cdot v & \rightarrow & \underline{\kappa \cdot v} \\
\mathtt{raise}\,\underline{\kappa \cdot v} & \rightarrow & \underline{\kappa \cdot v} \\
\mathtt{handle}\,\underline{\kappa \cdot v}\,\kappa\,\lambda x.\,e & \rightarrow & (\lambda x.\,e)\,\kappa \cdot v \\
\mathtt{handle}\,\underline{\kappa \cdot v}\,\kappa'\,\lambda x.\,e & \rightarrow & \underline{\kappa \cdot v} \quad (\kappa' \neq \kappa) \\
\mathtt{con}\,\kappa\,\underline{\kappa \cdot v} & \rightarrow & \underline{\kappa \cdot v} \\
\mathtt{decon}\,\underline{\kappa \cdot v} & \rightarrow & \underline{\kappa \cdot v} \\
\mathtt{case}\,\underline{\kappa \cdot v}\,\kappa'\,e_1\,e_2 & \rightarrow & \underline{\kappa \cdot v} \\
\underline{\kappa \cdot v}\,e & \rightarrow & \underline{\kappa \cdot v} \\
(\lambda x.\,e)\,\underline{\kappa \cdot v} & \rightarrow & \underline{\kappa \cdot v} \\
(\mathtt{fix}\,f\,\lambda x.\,e)\,\underline{\kappa \cdot v} & \rightarrow & \underline{\kappa \cdot v}
\end{array}
$$

그림 1 계산 단계들의 정의

계산할 부분인 e에 대한 한 단계의 계산인 $e \rightarrow e'$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. 정상적인 계산 단계들은 예외상황과 관련되지 않은 값의 계산을 나타낸다. 예외적인 계산 단계들은 예외상황이 발생하는 것과 발생된 예외상황이 전파되는 것과 발생된 예외상황이 처리되는 것들을 나타낸다. 여기서 사용한 의미구조에서는 예외상황을 발생시키는 코드 표현들은 발생한 예외상황의 값으로 계산되는 것으로 나타난다.

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

그림 2 단순한 CPS 변화 함수 T

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.*

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots.$$

위의 연속된 나열이 더이상 계산되지 않는 v라는 값 (또는 처리되지 않은 예외상황)으로 끝나는 경우 다음과 같이 나타낸다.

$$e \xrightarrow{*} v \quad (\text{또는}, e \xrightarrow{*} \underline{\kappa \cdot v})$$

5. 단순한 CPS 변환

예외상황 처리기에 대한 정보는 또 하나의 남은 할일(continuation)을 이용하면 나타낼 수 있다. 일반적인 CPS 변환은 프로그램의 남은 동작들을 나타내기 위해서 하나의 남은 할 일을 사용한다. 하지만 사용하는 언어에서 예외상황 처리 부분을 지원하게 되면 하나의 남은 할 일만을 가지고는 남은 동작을 쉽게 나타낼 수 없다. 정상적인 수행과정에 대한 하나의 남은 할 일과 예외적인 수행과정에 대한 또 하나의 남은 할 일이 필요하다.

예를 들어, 다음과 같은 handle코드 표현을 생각해 보자.

$$\mathrm{handle}\ e_1\ x\ \lambda x.e_2.$$

예외상황 처리기인 $\lambda x.e_2$는 e_1의 값을 계산하기 전에 먼저 설치되어야 한다. 그렇게 함으로서 e_1의 값을 계산

하는 중에 $\underline{x \cdot v}$이라는 예외상황이 발생하는 경우에 이 예외상황이 처리기에 의해서 처리되고 $x \cdot v$값이 처리하는 함수인 $\lambda x.e_2$에 인자로 적용하여 계산한 값이 handle 코드 표현의 최종 결과 값이 된다. 그리고 예외상황이 발생하지 않는 경우는 e_1의 값이 handle코드 표현의 값이 되고 프로그램의 나머지 부분을 수행할 수 있게 된다.

여기에 나타낸 CPS 변환은 발생한 예외상황을 어떻게 처리하고 처리 후에는 계속해서 어떻게 하는지를 *예외적인 수행과정에 대한 남은 할 일*으로 나타낸다. 이 예외적인 수행과정에 대한 남은 할 일은 하위 코드 표현인 e_1에 넘겨준다. 따라서 raise코드 표현은 현재의 정상적인 수행과정에 대한 남은 할일 대신에 예외적인 수행과정에 대한 남은 할 일을 사용하면 된다. 주의해야 할 것은, 예외적인 수행과정에 대한 남은 할 일을 나타내기 위해서는 예외상황을 처리하고 나서 무엇을 해야 하는지도 알고 있어야 하므로 정상적인 수행과정에 대한 남은 할 일을 알고 있어야만 한다는 것이다. 이 때문에 모든 표현에 두개의 남은 할일(정상적인 수행과정에 대한 남은 할 일과 예외적인 수행과정에 대한 남은 할 일)들을 넘겨주도록 해야 한다.

위의 내용을 만족하는 CPS 변환 함수인 T를 그림 2에 나타내었다.

정리 1 *(T의 정확함) 임의 프로그램 s 에 대해서 다음이 만족함을 보이면 된다.*

$$s \rightarrow v \Rightarrow T(s)\langle K, H\rangle \rightarrow K(\Psi(v))$$

이 때 새로이 정의된 보조 함수 Ψ는 원래의 프로그램에서의 값을 CPS에서의 값으로 바꾸어주는 함수로 다음과 같이 정의된다.

$$\begin{aligned} \Psi(1) &= 1 \\ \Psi(\lambda x.e) &= \lambda x.T(e) \\ \Psi(\text{fix } f\ \lambda x.e) &= \text{fix } f\ \lambda x.T(e) \\ \Psi(x \cdot v) &= x \cdot \Psi(v) \end{aligned}$$

증명. *이 정리는 Plotkin의 시뮬레이션 정리의 증명[10]과 비슷하게 증명할 수 있다. 이를 위해서는 변환 함수 T를 다음과 같이 계산 값과 발생한 예외상황에 대해서 확장해야 한다.*

$$\begin{aligned} T(v) &= \lambda\langle K, H\rangle.K(\Psi(v)) \\ T(\underline{x \cdot v}) &= \lambda\langle K, H\rangle.H(\Psi(x \cdot v)) \end{aligned}$$ □

그러나 T에서 이루어지는 것처럼 모든 코드 표현들을 CPS 형태로 변환하는 것은 raise와 handle을 없애서 얻을 수 있는 이익을 초과하는 비용이 든다. 실제로 모든 코드 표현들이 두 개의 함수(남은 할 일들)를 받는 고차원의 함수를 사용하도록 바뀐다.

이러한 상황에서 다음과 같은 두 가지의 서로 독립적인 개선이 가능하다:

- CPS 변환을 할 때 변환을 위해 부가적으로 추가되는 계산(administrative reduction) [4]을 처리할 수 있다. 우리의 변환을 두 단계로 나타나는 변환으로 바꾸어서 모든 부가적으로 추가되는 계산들을 정적 처리 단계에서 수행할 수 있다. 이에 대한 내용은 9 절에서 다루고 있다.
- 이광근과 류석영의 예외상황 분석기[13]의 결과를 이용하여 CPS 변환을 선택적으로 적용할 수 있다. 예외상황 분석기는 어떠한 코드 표현들이 계산할 때 예외상황을 발생시킬 수 있고, 어떠한 함수들이 사용될 때 예외상황을 발생시킬 수 있는지에 대한 신중하고 안전한 근사치를 알려준다. 따라서 분석기의 결과로부터 어떤 코드 표현과 함수들이 확실히 예외상황을 발생시키지 않는지 알 수 있다. 그리고 그러한 코드 표현들과 함수들은 CPS 변환을 할 필요가 없다.

6. 정보 표시

코드 표현 e의 값을 계산하는 중에 처리되지 않은 예외상황이 발생하고 이 예외상황이 e의 수행 전에 설치된 처리기에 의해서 처리되는 경우를 위해서 e에게 두 개의 남은 할 일을 전달한다. raise코드 표현은 외부의 프로그램 문맥에서 전달된 예외적인 수행과정에 대한 남은 할 일을 호출함으로서 예외상황을 발생시킨다. 따라서 어떤 코드 표현의 값을 계산하는 중에 처리되지 않은 예외상황이 발생하지 않는다는 것을 정적 분석을 통해 알 수 있다면, 넘겨줘야 하는 남은 할 일이 실제로는 필요가 없다는 것을 알 수 있으므로 아예 남은 할 일을 넘겨주지 않도록 하면 된다. 이 때 다음과 같은 세 가지 경우가 가능하다 :

- e의 값을 계산하는 중에 처리되지 않은 예외상황이 절대 발생하지 않는 경우(즉, 정상적인 값으로만 계산되는 경우)가 가능하고 이 경우에는 남은 할 일들이 필요 없다.
- e가 처리되지 않은 예외상황을 발생시킬 수 있지만, 그 예외상황이 프로그램의 어느 부분에서도 처리되지 않는 경우가 가능하고 이 경우에도 남은 할 일들이 필요 없다. 왜냐하면 그러한 예외상황이 발생하는 경우에는 그 때 프로그램을 종료하면 되기 때문이다.
- e가 처리되지 않은 예외상황을 발생시킬 수 있고, 그 예외상황이 프로그램의 다른 부분에서 처리되는 것

이 가능한 경우가 가능하고 이 때에는 남은 할 일을 넘겨줄 필요가 있다.

세 번째 경우에 해당하는 코드 표현들이 이 논문의 선택적 CPS 변환이 변환해야할 것들이다. 다른 경우에 해당하는 것들은 될 수 있으면 그대로 두면 된다.

그러므로 이 논문의 선택적 CPS 변환은 각각의 코드 표현에 대해서 두 가지의 정보가 필요하다.

첫 번째 조건인 $\lceil e \rceil^{\kappa}$는 코드 표현이 실행될 때 이미 설치되어 있을 수 있는 예외상황 처리기가 있는지를 검사한다.

두 번째 조건인 $\lfloor e \rfloor^{\kappa}$는 코드 표현 e의 값을 결정할 때 어떠한 처리되지 않은 예외상황을 발생시킬 수 있는지 검사한다.

코드 표현 e가 있을 때 만약 두 가지 조건 $\lceil e \rceil^{\kappa}$과 $\lfloor e \rfloor^{\kappa}$이 특정 예외상황 x에 대해서 모두 만족한다면 그 코드 표현 e는 *예외상황 가능*이라고 하고 $\check{e}$라고 표기한다. 다른 경우에는 e를 *일반적*이라고 하고 $\dot{e}$라고 표기한다. $\check{e}$라고 표기된 것들이 이 논문의 선택적 CPS 변환의 대상이 된다.

그림 3은 프로그램 $\mathcal{S}$에 있는 하위 코드 표현들 중에서 어떤 것들이 예외상황 가능한지를 결정하는 다음과 같은 규칙을 표시하고 있다 :

$$\boldsymbol{R}(Exn\_analysis_{\mathcal{S}},\ Closure\_analysis_{\mathcal{S}})$$

정보 표시 규칙들은 세 가지 부분으로 나뉜다. 첫 번째 부분에서는 코드 표현이 발생시킬 수 있는 처리되지 않은 예외상황을 결정한다. 이 부분은 예외상황 분석기의 결과를 이용한다 [13] (이광근과 류석영의 분석기는 함수에 대한 정보만을 제공하기 때문에 이 정보를 모든 코드 표현에 대한 것으로 확장해야 한다). 두 번째 부분에서는 코드 표현이 예외상황 처리기에 의해서 둘러싸일 수 있는지를 결정한다. 이 때 다음과 같은 두 가지 경우가 가능하다 :

- 어떤 코드 표현이 예외상황 처리기에 의해서 싸여 있으면 그 코드 표현의 하위 코드 표현들도 같은 예외상황 처리기에 의해서 둘러 싸여 있는 것이다.
- 어떤 함수 호출 코드 표현이 예외상황 처리기에 의해서 싸여 있으면 그 코드 표현에서 호출될 수 있는 모든 함수의 내부도 같은 예외상황 처리기에 의해서 둘러 싸여 있는 것이다.

함수의 호출 관계 분석기 [13]를 사용하여 우리는 함수 호출 코드 표현이 있을 때 실제로 어떤 함수들이 호출 될 수 있는지 알 수 있다. 정보 표시 규칙의 두 번째 부분에 위의 두 가지 경우가 나타나 있다. 세 번째 부분에서는 위의 두 부분의 정보를 통합한다. 어떤 코드 표현이 처리되지 않은 예외상황을 발생시킬 수 있고 그 예외상황을 처리할 수 있는 처리기가 둘러싸고 있을 수 있으면 그 코드 표현은 예외상황 처리기 정보가 필요한 것이다. 만약 그렇지 않은 경우에는 그 코드 표현은 예외상황 처리기에 대한 정보가 필요 없는 것이다.

따라서 우리는 어떤 코드 표현들이 예외상황 처리기의 정보가 필요한지를 알 수 있다.

이러한 분석을 위해서는 두 가지의 보조 분석이 필요하다.

$$\frac{\kappa \in Exn\_analysis_{\mathcal{P}}(e)}{\lfloor e \rfloor_{\kappa}}$$

$$\frac{\texttt{handle}\ e_1\ \kappa\ \lambda x.\ e_2}{\lceil e_1 \rceil^{\kappa}}$$

$$\frac{\lceil \texttt{con}\ \kappa'\ e \rceil^{\kappa}}{\lceil e \rceil^{\kappa}} \qquad \frac{\lceil \texttt{decon}\ e \rceil^{\kappa}}{\lceil e \rceil^{\kappa}} \qquad \frac{\lceil \texttt{raise}\ e \rceil^{\kappa}}{\lceil e \rceil^{\kappa}}$$

$$\frac{\lceil e_1\ e_2 \rceil^{\kappa} \quad \lambda x.\ e \in Closure\_analysis_{\mathcal{P}}(e_1)}{\lceil e_1 \rceil^{\kappa} \quad \lceil e_2 \rceil^{\kappa} \quad \lceil e \rceil^{\kappa}}$$

$$\frac{\lceil \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \rceil^{\kappa}}{\lceil e_1 \rceil^{\kappa} \quad \lceil e_2 \rceil^{\kappa} \quad \lceil e_3 \rceil^{\kappa}}$$

$$\frac{\lceil e \rceil^{\kappa} \quad \lfloor e \rfloor_{\kappa}}{\check{e}}$$

$\lfloor e \rfloor_{\kappa}$ 는 "e의 값을 계산할 때 κ-예외상황이 발생할 수 있음"을 의미한다.
$\lceil e \rceil^{\kappa}$ 는 "κ-예외상황 처리기가 e를 처리할 수 있음" 을 의미한다.
분석이 모두 끝나고 난 후에 $\check{e}$라고 표시되지 않은 모든
코드 표현 e는 $\dot{e}$으로, 즉 일반적이라고 표시한다.

그림 3 프로그램 $\mathcal{S}$에서 코드 표현 e가 예외상황 가능($\check{e}$)인지 일반적($\dot{e}$)인지 결정하는 $\boldsymbol{R}(Exn\_analysis_{\mathcal{S}},\ Closure\_analysis_{\mathcal{S}})$

$Exn_analysis_{\mathcal{S}}$로 나타내는 첫 번째 분석은 프로그램 $\mathcal{S}$의 각각의 코드 표현에 대해서 발생 가능한 처리되지 않은 예외상황을 알려주는 분석이다.

$Closure_analysis_{\mathcal{S}}$로 나타내는 두 번째 분석은 각각의 함수 타입의 코드 표현에 대해서 그 코드 표현이 가질 수 있는 실제 함수들을 알려주는 분석이다. 두 가지의 분석은 옳다고 가정한다. 실제로 구현을 할 때에는 이광근과 류석영에 의해서 구현된 분석기 [13]들을 사용하였다.

정의 2 *프로그램 $\mathcal{S}$에 대한 정보가 표시된 프로그램인 Annotate($\mathcal{S}$)은 그림 3에 나타나 있는 $\boldsymbol{R}(Exn\_analysis_{\mathcal{S}},\ Closure\_analysis_{\mathcal{S}})$ 규칙을 사용하여 각*

각의 하위 코드 표현들에 예외상황 가능 ($\dot{e}$)인지, 일반적 ($\dot{e}$)인지 표시 해 놓은 것을 말한다.

정리 2 (정보 표시의 안전성) *프로그램의 어떤 코드 표현 e가 $\dot{e}$로, 즉 일반적이라고 표시되었다면 그 프로그램의 값을 계산하면 정상적인 값으로 결정되던지 아니면 전체 프로그램의 수행을 종료하는 처리되지 않는 예외상황을 발생시킨다.*

정보가 표시된 코드 표현들 : e ::= $\dot{e} \mid \check{e}$

일반적인 코드 표현들 : $\dot{e}$ ::=
$1 \mid x$
$\mid \lambda x.\dot{e} \mid \lambda x.\check{e}$
$\mid \texttt{fix}\, f\, \lambda x.\, e$
$\mid \dot{e}_1\, \dot{e}_2$
$\mid \texttt{con}\, \kappa\, \dot{e} \mid \texttt{decon}\, \dot{e}$
$\mid \texttt{case}\, \dot{e}_1\, \kappa\, \dot{e}_2\, \dot{e}_3$
$\mid \texttt{handle}\, e_1\, \kappa\, \lambda x.\, \dot{e}_2$
$\mid \texttt{handle}\, \dot{e}_1\, \kappa\, \lambda x.\, \check{e}_2$
$\mid \texttt{raise}\, \dot{e}$

예외상황 가능한 코드 표현들 : $\check{e}$ ::=
$e_1\, e_2$
$\mid \texttt{con}\, \kappa\, \check{e}$
$\mid \texttt{decon}\, \check{e}$
$\mid \texttt{case}\, \check{e}_1\, \kappa\, e_2\, e_3$
$\mid \texttt{case}\, e_1\, \kappa\, \check{e}_2\, e_3$
$\mid \texttt{case}\, e_1\, \kappa\, e_2\, \check{e}_3$
$\mid \texttt{handle}\, \check{e}_1\, \kappa\, \lambda x.\, e_2$
$\mid \texttt{raise}\, e$

그림 4 두 가지 종류의 코드 표현들 : 일반적 또는 예외상황 가능

증명. *반례로 코드 표현 e가 처리되지 않은 예외상황을 발생시키고 이 예외상황이 프로그램 내의 처리기에 의해서 처리된다고 하자. 그러면 안전한 Exn_analysis와 Closure_analysis를 사용하는 $\lfloor e \rfloor^k$과 $\lceil e \rceil^k$에 의해서 e는 $\dot{e}$로 표시되었어야 하고 따라서 모순이 발생한다. 그러므로 위의 정리는 옳다.*□

그림 4는 잘 정의된 일반적인 코드 표현들인 $\dot{e}$의 가능한 구문들과 예외상황 가능한 코드 표현들인 $\check{e}$의 구문들을 나타내고 있다.

7. 선택적 CPS 변환

선택적 CPS 변환은 예외상황 가능한 코드 표현들은 단순한 CPS 변환과 같이 바꾸고 일반적인 코드 표현들은 가능한 한 바꾸지 않는 것이다.

이 선택적 CPS 변환은 두 개의 변환 함수 $\dot{T}$과 $\check{T}$으로 정의되는데 각각 일반적인 코드 표현과 예외상황 가능한 코드 표현을 변환하는데 쓰인다. 프로그램 $\mathcal{S}$을 정보가 표시된 프로그램인 $Annotate(\mathcal{S})$으로 바꾼 후에 $\dot{T}$을 이용하여 변환을 시작한다 :

$$\dot{T}(Annotate(\mathcal{S})).$$

$Annotate(\mathcal{S})$의 각각의 하위 코드 표현들은 각각의 정보 표시에 따라 $\dot{T}$이나 $\check{T}$에 의해서 변환된다. 그림 5는 두 함수의 정의를 나타내고 있다.

$\bot_K = \lambda x.x$과 $\bot_H = \lambda x.\texttt{raise}\, x$를 정의한다.
일반적인 코드 표현인 $\dot{e}$을 위한 변환 $\dot{T}(\dot{e})$.

$$\begin{aligned}
\dot{T}(1) &= 1\\
\dot{T}(x) &= x\\
\dot{T}(\lambda x.\, e) &= \lambda x.\, \check{T}(e)\\
\dot{T}(\texttt{fix}\, f\, \lambda x.\, e) &= \texttt{fix}\, f\, \lambda x.\, \check{T}(e)\\
\dot{T}(\dot{e}_1\, \dot{e}_2) &= \dot{T}(\dot{e}_1)\, \dot{T}(\dot{e}_2)\, \langle \bot_K, \bot_H \rangle\\
\dot{T}(\texttt{con}\, \kappa\, \dot{e}) &= \texttt{con}\, \kappa\, \dot{T}(\dot{e})\\
\dot{T}(\texttt{decon}\, \dot{e}) &= \texttt{decon}\, \dot{T}(\dot{e})\\
\dot{T}(\texttt{case}\, \dot{e}_1\, \kappa\, \dot{e}_2\, \dot{e}_3) &= \texttt{case}\, \dot{T}(\dot{e}_1)\, \kappa\, \dot{T}(\dot{e}_2)\, \dot{T}(\dot{e}_3)\\
\dot{T}(\texttt{handle}\, e_1\, \kappa\, \lambda x.\, e_2) &=\\
&\quad \check{T}(e_1)\, \langle \bot_K, \lambda v.\, \texttt{case}\, v\, \kappa\, ((\lambda x.\, \dot{T}(e_2))\, v)\, (\bot_H\, v) \rangle\\
\dot{T}(\texttt{raise}\, e) &= \bot_H\, \dot{T}(e)
\end{aligned}$$

예외상황 가능한 코드 표현인 $\check{e}$에 대한 변환 $\check{T}(\check{e})$.

$$\begin{aligned}
\check{T}(e_1\, e_2) &= \lambda \langle K, H \rangle.\, \check{T}(e_1)\\
&\quad \langle \lambda f.\, \check{T}(e_2)\, \langle \lambda v.\, f\, v\, \langle K, H \rangle, H \rangle, H \rangle\\
\check{T}(\texttt{con}\, \kappa\, e) &= \lambda \langle K, H \rangle.\, \check{T}(e) \langle \lambda v.\, K\, (\texttt{con}\, \kappa\, v), H \rangle\\
\check{T}(\texttt{decon}\, e) &= \lambda \langle K, H \rangle.\, \check{T}(e) \langle \lambda v.\, K\, (\texttt{decon}\, v), H \rangle\\
\check{T}(\texttt{case}\, e_1\, \kappa\, e_2\, e_3) &= \lambda \langle K, H \rangle.\, \check{T}(e_1)\\
&\quad \langle \lambda v.\, \texttt{case}\, v\, \kappa\, (\check{T}(e_2)\, \langle K, H \rangle)\, (\check{T}(e_3)\, \langle K, H \rangle), H \rangle\\
\check{T}(\texttt{handle}\, e_1\, \kappa\, \lambda x.\, e_2) &= \lambda \langle K, H \rangle.\, \check{T}(e_1)\\
&\quad \langle K, \lambda v.\, \texttt{case}\, v\, \kappa\, ((\lambda x.\, \check{T}(e_2))\, v\, \langle K, H \rangle)\, (H\, v) \rangle\\
\check{T}(\texttt{raise}\, e) &= \lambda \langle K, H \rangle.\, \check{T}(e)\, \langle H, H \rangle
\end{aligned}$$

일반적인 코드 표현인 $\dot{e}$을 위한 변환 $\check{T}(\dot{e})$.

$$\check{T}(\dot{e}) = \lambda \langle K, H \rangle.\, K\, \dot{T}(e)$$

그림 5 선택적 CPS변환

$\check{T}$는 예외상황 가능한 코드 표현들이 예외상황 처리기에 대한 정보를 받아야 하므로 예외상황 가능한 코드 표현들을 두 개의 남은 할 일들을 받도록 만든다.

$\dot{T}$는 일반적인 코드 표현들이 예외상황 처리기에 대한 정보를 필요로 하지 않으므로 일반적인 코드 표현들을 가능한 한 바꾸지 않는다. 이 때 한 가지 바꾸어 줘야 하는 것은 함수의 내부가 일반적인 코드 표현으로 되어 있는 *일반적인 함수*들을 두 개의 남은 할 일들을 받는 형태로 형태 변환을 해준다.

이러한 형태 변환이 필요한 이유는 예외상황 가능한 함수 호출 코드 표현이 일반적인 함수를 호출하는 것이 가능하기 때문인데, 예를 들면 다음과 같은 경우에 발생한다. 함수 호출 코드 표현 $\dot{e}_1\, \dot{e}_2$에 대해서 두 개의 함수 $\lambda x.\check{e}'$와 $\lambda y.\dot{e}''$가 모두 호출이 가능할 수 있다. 이 때

예외상황 가능한 함수의 내부인 $\breve{e}$때문에 원래의 함수 호출 코드 표현이 예외상황 가능으로 표시될 수 있다. 그런 경우에는 Υ에 의해 다음과 같이 변환된다 :

$$\lambda\langle K,H\rangle.\ \Upsilon(\dot{e}_1)\ \Upsilon(\dot{e}_2)\ \langle K,H\rangle.$$

이렇게 되면 호출이 가능한 다른 함수인 $\lambda y.\dot{e}'$를 두 개의 남은 할 일을 받을 수는 있도록 다음과 같이 형태 변환을 해 주어야 한다 :

$$\lambda y.\lambda.\langle K,H\rangle.K(\Upsilon(\dot{e}')).$$

이러한 형태 변환 때문에, 함수 호출 코드 표현인 $e_1\ e_2$은 항상 두 개의 남은 할 일을 넘겨주도록 바꾸어 주어야 한다. 따라서 일반적인 코드 표현에 대한 변환인 $\Upsilon(e_1\ e_2)$의 경우에는 가상의 기본적인 남은 할 일들인 $\langle\perp_K,\perp_H\rangle$를 넘겨주도록 변환한다. 그리고 예외상황 가능한 코드 표현에 대한 변환인 $\breve{\Upsilon}(e_1\ e_2)$의 경우에는 프로그램 문맥으로부터 받은 남은 할 일들인 $\langle K,H\rangle$을 넘겨주도록 변환한다. 이러한 해결방법은 간단한 해결방법이므로 개선할 수 있는 가능성이 있다. 9 절에서는 어떻게 개선할 수 있는지에 대해서 다루고 있다.

주의할 것은 $\perp_H$함수인 $(\lambda x.\ \text{raise}\ x)$때문에 변환된 프로그램에서도 여전히 raise코드 표현이 남아 있을 수 있지만, 이는 주어진 프로그램이 처리되지 않고 프로그램을 종료 시켜 버리는 예외상황을 발생시킬 때에 사용되므로 실제로는 의미 있게 사용되지 않는다.

8. 선택적 CPS 변환의 안전성

이 논문의 변환의 안전성을 증명하기 위해서는 변환과 정보 표시를 계산 중간과정에 대해서 확장을 해야 한다.

첫 번째로 계산 과정에서 나타날 수 있는 모든 중간 표현들에 대해서 변환을 확장해야 한다. 일반적인 코드 표현이 con코드 표현으로부터 만들어지는 상수 값으로 계산될 수 있으므로 일반적인 경우의 변환을 상수 값에 대해서 확장해야 한다. 예외상황 가능한 코드 표현은 어떠한 중간 표현으로 계산될 수 있으므로 예외상황 가능한 경우의 변환은 모든 가능한 중간 표현들에 대해서 확장해야 한다. 다음에 이러한 확장을 나타내고 있다.

$$\begin{aligned}
\Upsilon(e_1\ e_2) &= \Upsilon(e_1)\ \Upsilon(e_2)\ \langle\perp_K,\perp_H\rangle\\
\Upsilon(\text{con}\ x\ e) &= \text{con}\ x\ \Upsilon(e)\\
\Upsilon(\text{decon}\ e) &= \text{decon}\ \Upsilon(e)\\
\Upsilon(\text{case}\ e_1\ x\ e_2\ e_3) &= \text{case}\ \Upsilon(e_1)\ x\ \Upsilon(e_2)\ \Upsilon(e_3)\\
\Upsilon(\text{handle}\ e_1\ x\ \lambda x.e_2) &=\\
&\Upsilon(e_1)\ \langle\perp_K,\lambda v.\ \text{case}\ v\ x\ ((\lambda x.\ \Upsilon(e_2))\ v)\ (\perp_H\ v)\rangle\\
\Upsilon(\breve{e}) &= \breve{\Upsilon}(\breve{e})\ \langle\perp_K,\perp_H\rangle
\end{aligned}$$

$$\begin{aligned}
\Upsilon(x\cdot v) &= x\cdot\Upsilon(v)\\
\breve{\Upsilon}(x\cdot v) &= \Upsilon(x\cdot v)\\
\dot{\Upsilon}(x\cdot v) &= \lambda\langle K,H\rangle.H\ \Upsilon(x\cdot v)\\
\Upsilon(v) &= \lambda\langle K,H\rangle.K\ \Upsilon(v)
\end{aligned}$$

두 번째로 정보 표시의 정의를 계산 과정에서 나타날 수 있는 모든 중간 표현들에 대해서 정의를 해야 한다. 계산 과정에서 나타나는 중간 표현들에 대해서 정의하는 방법은 다음과 같다. 함수 호출 코드 표현을 계산한다면 실제로 불리는 함수의 내부가 가지고 있는 정보 표시를 계산 결과 중간 표현의 정보 표시로 정한다. 그 외의 경우에는 계산이 일어나기 전의 정보 표시를 계산이 일어난 후에도 계속 유지한다.

함수 호출 코드 표현들에 대해서는 다음 규칙에 의해서 정보 표시를 설정한다.

$$\frac{\lambda x.\dot{e}_1\ v\rightarrow[v/x]e_1}{[v/x]\dot{e}_1}$$

$$\frac{(\text{fix}\ f\ \lambda x.\dot{e}_1)\ v\rightarrow[v/x][\text{fix}\ f\ \lambda x.\dot{e}_1/f]e_1}{[v/x][\text{fix}\ f\ \lambda x.\dot{e}_1/f]\dot{e}_1}$$

$$\frac{\breve{e}=(\lambda x.\breve{e}_1)\ v,\quad e\rightarrow[v/x]e_1}{[v/x]\breve{e}_1}$$

$$\frac{\breve{e}=(\text{fix}\ f\ \lambda x.\breve{e}_1)\ v,\quad e\rightarrow[v/x][\text{fix}\ f\ \lambda x.\breve{e}_1/f]e_1}{[v/x][\text{fix}\ f\ \lambda x.\breve{e}_1/f]\breve{e}_1}$$

그 외의 코드 표현들에 대해서는 다음 규칙에 의해서 원래의 코드 표현의 정보 표시를 유지한다.

$$\frac{\dot{e}_1\quad e_1\rightarrow e_2}{\dot{e}_2}\qquad\frac{\breve{e}_1\quad e_1\rightarrow e_2}{\breve{e}_1}$$

정리 3은 이 논문의 선택적 CPS 변환 방법의 안전성을 보장해 준다.

정리 3 *(Υ와 Υ의 정확성) 임의의 프로그램 s에 대해서 다음이 만족한다.*

$$s\xrightarrow{*}v\ \Leftrightarrow\ \Upsilon(Annotate(s))\xrightarrow{*}\Upsilon(v).$$

정리 3이 나타내는 것은 이 논문의 변환으로 변환된 프로그램을 수행해서 얻은 결과를 원래의 프로그램을 수행하여 얻을 수 있는 결과로 쉽게 변환할 수 있다는 것이다. 원래의 프로그램이 함수 값이 포함되지 않은 결과를 낸다면 변환된 프로그램도 같은 결과를 낸다. 그리고 원래의 프로그램이 함수 값을 결과로 낸다면 변환된 프로그램은 원래 프로그램의 결과를 Υ으로 바꾸어준 결과를 낸다.

정리 3을 증명하기 위해서는 우리는 다음의 몇 가지 보조정리와 정의와 정의에 대한 사실들이 필요하다.

보조정리 1은 변수를 특정 값으로 치환하는 것을 $\dot{T}$과 $\grave{T}$함수의 안으로 집어넣을 수 있다는 것을 나타낸다.

보조 정리 1 *임의의 값 v와 프로그램 표현 e에 대해서 다음이 만족한다.*

$$[\dot{T}(v)/x]\,\dot{T}(e) = \dot{T}([v/x]e)$$
$$[\dot{T}(v)/x]\,\grave{T}(e) = \grave{T}([v/x]e)$$

증명. *e의 크기에 대한 귀납법을 사용한다.*

- $\dot{T}(1)$

$$\begin{array}{lll}[\dot{T}(v)/x]\,\dot{T}(1) & & \\ = [\dot{T}(v)/x]\,1 & \dot{T}\text{의 정의} & \\ = 1 & [\cdot/\cdot]\text{의 정의} & \\ = \dot{T}(1) & \dot{T}\text{의 정의} & \\ = \dot{T}([v/x]\,1) & [\cdot/\cdot]\text{의 정의} & \end{array}$$

- $\dot{T}(\text{con } x\ e)$

$$\begin{array}{lll}[\dot{T}(v)/x]\,\dot{T}(\text{con } x\ e) & & \\ = [\dot{T}(v)/x](\text{con } x\ \dot{T}(e)) & \dot{T}\text{의 정의} & \\ = \text{con } x\ ([\dot{T}(v)/x]\,\dot{T}(e)) & [\cdot/\cdot]\text{의 정의} & \\ = \text{con } x\ \dot{T}([v/x]e) & \text{귀납법의 가정} & \\ = \dot{T}(\text{con } x\ [v/x]e) & \dot{T}\text{의 정의} & \\ = \dot{T}([v/x]\,\text{con } x\ e) & [\cdot/\cdot]\text{의 정의} & \end{array}$$

- $\grave{T}(e_1\ e_2)$

$$\begin{array}{lll}[\dot{T}(v)/x]\,\grave{T}(e_1\ e_2) & & \\ = [\dot{T}(v)/x](\lambda\langle K,H\rangle.\,\grave{T}(e_1) & & \\ \quad \langle\lambda f.\,\grave{T}(e_2)\langle\lambda v.f\,v\langle K,H\rangle,H\rangle,H\rangle) & \grave{T}\text{의 정의} & \\ = \lambda\langle K,H\rangle.[\dot{T}(v)/x]\,\grave{T}(e_1) & & \\ \quad \langle\lambda f.[\dot{T}(v)/x]\,\grave{T}(e_2) & & \\ \quad\quad \langle\lambda v.f\,v\,\langle K,H\rangle,H\rangle,H\rangle & [\cdot/\cdot]\text{의 정의} & \\ = \lambda\langle K,H\rangle.\,\grave{T}([v/x]e_1) & & \\ \quad \langle\lambda f.\,\grave{T}([v/x]e_2) & & \\ \quad\quad \langle\lambda v.f\,v\,\langle K,H\rangle,H\rangle,H\rangle & \text{귀납법의 가정} & \\ = \grave{T}(([v/x]e_1)\ ([v/x]e_2)) & \grave{T}\text{의 정의} & \\ = \grave{T}([v/x](e_1\ e_2)) & [\cdot/\cdot]\text{의 정의} & \end{array}$$

- $\grave{T}(\underline{x\cdot v'})$

$$\begin{array}{lll}[\dot{T}(v)/x]\,\grave{T}(\underline{x\cdot v'}) & & \\ = [\dot{T}(v)/x](\lambda\langle K,H\rangle.H\ \dot{T}(x\cdot v)) & \grave{T}\text{의 정의} & \\ = [\dot{T}(v)/x](\lambda\langle K,H\rangle.H\,x\cdot\dot{T}(v)) & \grave{T}\text{의 정의} & \\ = (\lambda\langle K,H\rangle.H\,x\cdot[\dot{T}(v)/x]\,\dot{T}(v)) & \grave{T}\text{의 정의} & \\ = (\lambda\langle K,H\rangle.H\,x\cdot\dot{T}(v)) & [\cdot/\cdot]\text{의정의} & \\ = \grave{T}(\underline{x\cdot v'}) & \grave{T}\text{의 정의} & \\ = \grave{T}([v/x](\underline{x\cdot v'})) & [\cdot/\cdot]\text{의정의} & \end{array}$$

- $\grave{T}(\dot{e})$

$$\begin{array}{lll}[\dot{T}(v)/x]\,\grave{T}(\dot{e}) & & \\ = [\dot{T}(v)/x](\lambda\langle K,H\rangle.K\ \dot{T}(\dot{e})) & \grave{T}\text{의 정의} & \\ = \lambda\langle K,H\rangle.K\,[\dot{T}(v)/x]\,\dot{T}(\dot{e}) & \grave{T}\text{의 정의} & \end{array}$$

이제 $[\dot{T}(v)/x]\,\dot{T}(\dot{e})$이 $\dot{T}([v/x]\dot{e})$과 같다는 것을 보여야 한다. 지금까지 증명 한 것들은 $\dot{T}(\dot{e})$을 포함하므로 앞의 두 표현은 같다. (1)

$$\begin{array}{lll} = \lambda\langle K,H\rangle.K\,[\dot{T}(v)/x]\,\dot{T}(\dot{e}) & & \\ = \lambda\langle K,H\rangle.K\ \dot{T}([v/x]\dot{e}) & By & (1) \\ = \dot{T}([v/x]\,\dot{e}) & \dot{T}\text{의 정의} & \end{array}$$

다음의 경우들은 위의 경우 중의 하나와 비슷하게 증명할 수 있다.

- 경우와 비슷한 경우

$\dot{T}(x)$, $\dot{T}(y)$, $\dot{T}(\lambda x.e)$, $\dot{T}(\text{fix } f.\lambda x.e)$

- 경우와 비슷한 경우

$\dot{T}(\lambda y.e)$ $(y\neq x)$, $\dot{T}(\text{fix } f.\lambda y.e)$ $(y\neq x)$,
$\dot{T}(e_1\ e_2)$, $\dot{T}(\text{decon } e)$,
$\dot{T}(\text{case } e_1\ x\ e_2\ e_3)$, $\dot{T}(\text{handle } e_1\ x\ \lambda x.e_2)$,
$\dot{T}(\text{handle } e_1\ x\ \lambda y.e_2)$ $(y\neq x)$,
$\dot{T}(\text{raise } e)$, $\dot{T}(x\cdot v)$,
$\dot{T}(\underline{x\cdot v})$

- 경우와 비슷한 경우

$\grave{T}(\text{con } x\ e)$, $\grave{T}(\text{decon } e)$,
$\grave{T}(\text{case } e_1\ x\ e_2\ e_3)$, $\grave{T}(\text{handle } e_1\ x\ \lambda x.e_2)$,
$\grave{T}(\text{handle } e_1\ x\ \lambda y.e_2)$ $(y\neq x)$,
$\grave{T}(\text{raise } e)$

- 경우와 비슷한 경우

$\grave{T}(\dot{e})$, $\grave{T}(v)$□

정의 3은 두 개의 닫혀진 중간 표현들의 동치관계를 정의한다.

정의 3 ($\approx$) *두 개의 닫혀진 중간 표현들 e과 e'에 대해서,*

$e \approx e' \Leftrightarrow$ 두 중간표현의 다음과 같이 같은 결과를 내는 경우 ($e \xrightarrow{*} \gamma$ 이고 $e' \xrightarrow{*} \gamma$) 이거나 둘 다 끝나지 않는다.

다음으로 프로그램 표현 e의 계산 과정의 길이에 대한 정의를 한다.

정의 4 (계산 과정의 길이) *두개의 닫혀진 중간 표현들 e에 대해서,*

$$\|e\| = \begin{cases} n & \text{어떤 결과값 } \gamma\text{에 대해서 } e \xrightarrow{n} \gamma \text{ 를만족하는 경우.} \\ \infty & \text{다른 경우} \end{cases}$$

위의 동치 관계 $\approx$와 계산 과정의 길이 $\|\cdot\|$에 대한 몇 가지 사실을 나타내면 다음과 같다.

사실 1 *두 개의 닫혀진 중간 표현들 e와 e'에 대해서, 만약 $e \to e'$이 성립하면 $e \approx e'$과 $\|e\| - 1 = \|e'\|$이 만족한다.*

사실 2 *두 개의 닫혀진 중간 표현들 e에 대해서, $(\lambda x.x)\ e \approx e$와 $\|(\lambda x.x)\ e\| - 1 = \|e\|$이 만족한다.*

사실 3 *두 개의 닫혀진 중간 표현들 e와 e'과 e''에 대해서, 만약 $e \approx e'$이고 $e' \approx e''$이면 $e \approx e''$이 성립한다.*

보조 정리 2 *임의의 계산 문맥 $C[]$와 임의의 두 개의 닫혀진 중간 표현들 e와 e'에 대해서, 만약 $e \approx e'$을 만족하면 $C[e] \approx C[e']$도 만족한다.*

증명. *$C[]$가 계산 문맥이므로 e와 e'는 $C[e]$와 $C[e']$에서 각각 바로 다음에 수행해야 할 부분들이다.*

그리고 e와 e'는 닫혀 있으므로 e와 e'의 수행은 계산 문맥 $C[]$의 영향을 받지 않는다.□

보조 정리 3 *$e \approx e'$이 만족한다고 하자. 그러면 임의의 계산 문맥 $C[]$에 대해서, $\|e\| - k = \|e'\|$ $(0<k<\infty)$이 만족하면 $\|C[e]\| - k = \|C[e']\|$도 만족한다.*

증명. *e와 e'가 같은 값 γ로 계산되는 경우에는 $C[e]$와 $C[e']$가 같은 중간표현인 $C[\gamma]$로 계산된다. e와 e'가 모두 계산이 끝나지 않는 경우에는, $C[e]$와 $C[e']$도 역시 계산이 끝나지 않는다.□*

보조 정리 4 는 만약 e가 e'로 계산된다면, 두 표현의 선택적으로 CPS 변환된 표현들 사이에 동치관계가 있음을 나타낸다.

보조 정리 4 *임의의 닫혀진 중간 표현 $\dot{e}$와 $\breve{e}$에 대해서,*

$$\dot{e} \to e' \Rightarrow (\dot{\mathcal{T}}(\dot{e}) \approx \dot{\mathcal{T}}(e') \text{ and } \exists k(0<k<\infty).\ \|\dot{\mathcal{T}}(\dot{e})\| - k = \|\dot{\mathcal{T}}(e')\|)$$

과

$$\breve{e} \to e' \Rightarrow (\breve{\mathcal{T}}(\breve{e})\langle K,H\rangle \approx \breve{\mathcal{T}}(e')\langle K,H\rangle \text{ and } \exists k(0<k<\infty).\ \|\breve{\mathcal{T}}(\breve{e})\langle K,H\rangle\| - k = \|\breve{\mathcal{T}}(e')\langle K,H\rangle\|).$$

이 만족한다.

증명. e의 크기에 대한 귀납법을 사용한다.

우리의 정보 표시 규칙들은 안전하고 입력 프로그램이 프로그램 어디서도 처리되지 않는 예외상황을 발생시키지 않는다고 가정하고 있으므로 예외상황 가능한 함수들이 일반적인 함수 적용 표현에서 사용될 수 없고 또한 일반적인 함수가 예외상황 가능한 표현이나 처리되지 않은 예외상황으로 계산될 수 없다.

따라서 우리는 가능한 계산과정과 정보 표시들에 대해서만 증명하면 된다.

- $\dot{e} = \dot{e}_1\ \dot{e}_2 \to \dot{e}_1'\ \dot{e}_2 \quad (e_1 \to e_1')$

귀납법의 가정에 의해서,

$\dot{\mathcal{T}}(\dot{e}_1) \approx \dot{\mathcal{T}}(\dot{e}_1')$이고 (1)

$\|\dot{\mathcal{T}}(\dot{e}_1)\| - k = \|\dot{\mathcal{T}}(\dot{e}_1')\|$을 만족하는 k $(0<k<\infty)$가 존재한다. (2)

$$\begin{aligned}
&\dot{\mathcal{T}}(\dot{e}_1\ \dot{e}_2)\\
&= \dot{\mathcal{T}}(\dot{e}_1)\ \dot{\mathcal{T}}(\dot{e}_2)\langle\perp_K,\perp_H\rangle && (\dot{\mathcal{T}}\text{의 정의})\\
&\approx \dot{\mathcal{T}}(\dot{e}_1')\ \dot{\mathcal{T}}(\dot{e}_2)\langle\perp_K,\perp_H\rangle && ((1)\text{과 보조정리 2})\\
&= \dot{\mathcal{T}}(\dot{e}_1'\ \dot{e}_2) && (\dot{\mathcal{T}}\text{의 정의})
\end{aligned}$$

(2)와 보조정리 3에 의해서,

$\|\dot{\mathcal{T}}(\dot{e}_1\ \dot{e}_2)\| - k = \|\dot{\mathcal{T}}(\dot{e}_1'\ \dot{e}_2)\|$.

- $\dot{e} = (\lambda x.\dot{e}_1)\ v \to [v/x]\dot{e}_1$

$$\begin{aligned}
&\dot{\mathcal{T}}((\lambda x.\dot{e}_1)\ v)\\
&= \dot{\mathcal{T}}(\lambda x.\dot{e}_1)\ \dot{\mathcal{T}}(v)\langle\perp_K,\perp_H\rangle && (\dot{\mathcal{T}}\text{의 정의})\\
&= (\lambda x.\dot{\mathcal{T}}(\dot{e}_1))\ \dot{\mathcal{T}}(v)\langle\perp_K,\perp_H\rangle && (\dot{\mathcal{T}}\text{의 정의})\\
&= (\lambda x.\lambda\langle K,H\rangle.K\ \dot{\mathcal{T}}(\dot{e}_1))\ \dot{\mathcal{T}}(v)\langle\perp_K,\perp_H\rangle && (\dot{\mathcal{T}}\text{의 정의})\\
&\to ([\dot{\mathcal{T}}(v)/x]\lambda\langle K,H\rangle.K\ \dot{\mathcal{T}}(\dot{e}_1))\langle\perp_K,\perp_H\rangle && (\to\text{의 정의}) \quad (1)\\
&= (\lambda\langle K,H\rangle.K\ [\dot{\mathcal{T}}(v)/x]\ \dot{\mathcal{T}}(\dot{e}_1))\langle\perp_K,\perp_H\rangle && ([\cdot/\cdot]\text{의정의}) \quad (2)\\
&\to \perp_K\ [\dot{\mathcal{T}}(v)/x]\ \dot{\mathcal{T}}(\dot{e}_1) && (\to\text{의 정의})\\
&= (\lambda x.x)\ [\dot{\mathcal{T}}(v)/x]\ \dot{\mathcal{T}}(\dot{e}_1) && (\text{그림 5에 있는 } \perp_K\text{의 정의}) \quad (3)\\
&\approx [\dot{\mathcal{T}}(v)/x]\ \dot{\mathcal{T}}(\dot{e}_1) && (\text{사실 2})\\
&= \dot{\mathcal{T}}([v/x]\dot{e}_1) && (\text{보조정리 1})
\end{aligned}$$

(1)과 (2)와 (3)과 사실 2에 의해서,

$\|\dot{\mathcal{T}}((\lambda x.\dot{e}_1)\ v)\| - 3 = \|\dot{\mathcal{T}}([v/x]\dot{e}_1)\|$.

- $\dot{e} = \text{handle}\ \underline{x \cdot v}\ x\ \lambda x.\dot{e}_1 \to [x \cdot v/x]\dot{e}_1$

$$\begin{aligned}
&\dot{\mathcal{T}}(\text{handle}\ \underline{x \cdot v}\ x\ \lambda x.\dot{e}_1)\\
&= \dot{\mathcal{T}}(\underline{x \cdot v})\langle\perp_K,\\
&\quad \lambda v.\text{case}\ v\ x\ ((\lambda x.\dot{\mathcal{T}}(\dot{e}_1))\ v)\ (\perp_H v)\rangle && (\dot{\mathcal{T}}\text{의 정의})\\
&= (\lambda\langle K,H\rangle.H\ \dot{\mathcal{T}}(x \cdot v))\langle\perp_K,\\
&\quad \lambda v.\text{case}\ v\ x\ ((\lambda x.\dot{\mathcal{T}}(\dot{e}_1))\ v)\ (\perp_H v)\rangle && (\dot{\mathcal{T}}\text{의 정의})\\
&\xrightarrow{2} \text{case}\ \dot{\mathcal{T}}(x \cdot v)\ x\\
&\quad ((\lambda x.\dot{\mathcal{T}}(\dot{e}_1))\ \dot{\mathcal{T}}(x \cdot v))\\
&\quad (\perp_H\ \dot{\mathcal{T}}(x \cdot v)) && (\to\text{의 정의}) \quad (1)\\
&= \text{case}\ x \cdot \dot{\mathcal{T}}(v)\ x\\
&\quad ((\lambda x.\dot{\mathcal{T}}(\dot{e}_1))\ \dot{\mathcal{T}}(x \cdot v))\\
&\quad (\perp_H\ \dot{\mathcal{T}}(x \cdot v)) && (\dot{\mathcal{T}}\text{의 정의}) \quad (2)\\
&\xrightarrow{2} [\dot{\mathcal{T}}(x \cdot v)/x]\ \dot{\mathcal{T}}(\dot{e}_1) && (\to\text{의 정의})\\
&= \dot{\mathcal{T}}([x \cdot v/x]\dot{e}_1) && (\text{보조정리1})
\end{aligned}$$

(1)과 (2)에 의해서,

$\|\dot{\mathcal{T}}(\text{handle}\ \underline{x \cdot v}\ x\ \lambda x.\dot{e}_1)\| - 4 = \|\dot{\mathcal{T}}([x \cdot v/x]\dot{e}_1)\|$.

- $\breve{e} = \breve{e}_1\ e_2 \to e_1'\ e_2 \quad (\breve{e}_1 \to e_1')$

귀납법의 가정에 의해서

$\breve{\mathcal{T}}(\breve{e}_1)\langle K',H'\rangle \approx \breve{\mathcal{T}}(e_1')\langle K',H'\rangle$이고 (1)

$\|\breve{\mathcal{T}}(\breve{e}_1)\langle K',H'\rangle\| - k = \|\breve{\mathcal{T}}(e_1')\langle K',H'\rangle\|$을 만족하는 $k(0<k<\infty)$가 존재한다. (2)

$$\begin{aligned}
&\breve{\mathcal{T}}(\breve{e}_1\ e_2)\langle K,H\rangle\\
&= (\lambda\langle K,H\rangle.\breve{\mathcal{T}}(\breve{e}_1)\langle\lambda f.\dot{\mathcal{T}}(e_2)\\
&\quad \langle\lambda v.f\ v\langle K,H\rangle,H\rangle,H\rangle\langle K,H\rangle && (\breve{\mathcal{T}}\text{의 정의})\\
&\to \breve{\mathcal{T}}(\breve{e}_1)\langle\lambda f.\dot{\mathcal{T}}(e_2)\\
&\quad \langle\lambda v.f\ v\langle K,H\rangle,H\rangle,H\rangle && (\to\text{의 정의}) \quad (3)\\
&\approx \breve{\mathcal{T}}(e_1')\langle\lambda f.\dot{\mathcal{T}}(e_2)\\
&\quad \langle\lambda v.f\ v\langle K,H\rangle,H\rangle,H\rangle && ((1)\text{과 보조정리 2})\\
&\leftarrow (\lambda\langle K,H\rangle.\breve{\mathcal{T}}(e_1')\langle\lambda f.\dot{\mathcal{T}}(e_2)\\
&\quad \langle\lambda v.f\ v\langle K,H\rangle,H\rangle,H\rangle\langle K,H\rangle && (\to\text{의 정의}) \quad (4)\\
&= \breve{\mathcal{T}}(e_1'\ e_2)\langle K,H\rangle && (\breve{\mathcal{T}}\text{의 정의})
\end{aligned}$$

(2)와 (3)과 (4)에 의해서,

$\|\breve{\mathcal{T}}(\breve{e}_1\ e_2)\langle K,H\rangle\| - k = \|\breve{\mathcal{T}}(e_1'\ e_2)\langle K,H\rangle\|$

- $\breve{e} = (\lambda x.e_1)\ v \to [v/x]e_1$

$\dot{T}((\lambda x.e_1)\ v)\ \langle K,H\rangle$ (1)

$= (\lambda\langle K,H\rangle.\ \dot{T}(\lambda x.e_1)\ \langle\lambda f.\ \dot{T}(v)$
$\quad \langle\lambda v'.f\ v'\ \langle K,H\rangle,H\rangle,H\rangle\ \langle K,H\rangle$ ($\dot{T}$의 정의)
$\rightarrow \dot{T}(\lambda x.e_1)\ \langle\lambda f.\ \dot{T}(v)$
$\quad \langle\lambda v'.f\ v'\ \langle K,H\rangle,H\rangle,H\rangle$ (→의 정의)
$= (\lambda\langle K,H\rangle.K\ \dot{T}(\lambda x.e_1))\ \langle\lambda f.\ \dot{T}(v)$
$\quad \langle\lambda v'.f\ v'\ \langle K,H\rangle,H\rangle,H\rangle$ ($\dot{T}$의 정의) (2)
$\rightarrow (\lambda f.\ \dot{T}(v)\ \langle\lambda v'.f\ v'\ \langle K,H\rangle,H\rangle$
$\quad \dot{T}(\lambda x.e_1)$ (→의 정의)

$= (\lambda f.\ \dot{T}(v)\ \langle\lambda v'.f\ v'\ \langle K,H\rangle,H\rangle$
$\quad \lambda x.\ \dot{T}(e_1)$ ($\dot{T}$의 정의) (3)
$\rightarrow \dot{T}(v)\ \langle\lambda v'.(\lambda x.\ \dot{T}(e_1))\ v'\ \langle K,H\rangle,H\rangle$ (→의 정의)
$= (\lambda\langle K,H\rangle.K\ \dot{T}(v))$
$\quad \langle\lambda v'.(\lambda x.\ \dot{T}(e_1))\ v'\ \langle K,H\rangle,H\rangle$ ($\dot{T}$의 정의)
$\xrightarrow{3} ([\ \dot{T}(v)/x]\ \dot{T}(e_1))\ \langle K,H\rangle$ (→의 정의) (4)
$= \dot{T}([v/x]e_1)\ \langle K,H\rangle$ (보조정리1)

(1)과 (2)와 (3)과 (4)에 의해서,

$\|\dot{T}((\lambda x.e_1)\ v)\ \langle K,H\rangle\| - 6 = \|\dot{T}([v/x]e_1)\ \langle K,H\rangle\|.$

- $\check{e} = \text{con}\ x\ v \rightarrow x \cdot v$

$\dot{T}(\text{con}\ x\ v)\ \langle K,H\rangle$
$= (\lambda\langle K,H\rangle.\ \dot{T}(v)$
$\quad \langle\lambda v.K\ (\text{con}\ x\ v),H\rangle\ \langle K,H\rangle$ ($\dot{T}$의 정의) (1)
$\rightarrow \dot{T}(v)\ \langle\lambda v.K\ (\text{con}\ x\ v),H\rangle$ (→의 정의)

$= (\lambda\langle K,H\rangle.K\ \dot{T}(v))$
$\quad \langle\lambda v.K\ (\text{con}\ x\ v),H\rangle$ ($\dot{T}$의 정의) (2)
$\xrightarrow{3} K\ x \cdot\ \dot{T}(v)$ (→의 정의)

$= K\ \dot{T}(x \cdot v)$ ($\dot{T}$의정의) (3)
$\leftarrow (\lambda\langle K,H\rangle.K\ \dot{T}(x \cdot v))\ \langle K,H\rangle$ (→의 정의)
$= \dot{T}(x \cdot v)\ \langle K,H\rangle$ ($\dot{T}$의 정의)

(1)과 (2)와 (3)에 의해서,

$\|\dot{T}(\text{con}\ x\ v)\ \langle K,H\rangle\| - 3 = \|\dot{T}(x \cdot v)\ \langle K,H\rangle\|.$

다음의 경우들은 위의 경우 중의 하나와 비슷하게 증명할 수 있다.

- *경우와 비슷한 경우.*

$\dot{e}_1 \rightarrow \dot{e}_1'$ *라고 하자.*

$\dot{e} = v\ \dot{e}_1 \rightarrow v\ \dot{e}_1'$

$\dot{e} = \text{con}\ x\ \dot{e}_1 \rightarrow \text{con}\ x\ \dot{e}_1'$

$\dot{e} = \text{decon}\ \dot{e}_1 \rightarrow \text{decon}\ \dot{e}_1'$

$\dot{e} = \text{case}\ \dot{e}_1\ x\ \dot{e}_2\ \dot{e}_3 \rightarrow \text{case}\ \dot{e}_1'\ x\ \dot{e}_2\ \dot{e}_3$

$\dot{e} = \text{raise}\ \dot{e}_1 \rightarrow \text{raise}\ \dot{e}_1'$

$\dot{e} = \text{handle}\ \dot{e}_1\ x\ \lambda x.\dot{e}_2 \rightarrow \text{handle}\ \dot{e}_1'\ x\ \lambda x.\dot{e}_2$

- *경우와 비슷한 경우.*

$\dot{e} = (\text{fix}\ f\ \lambda x.\dot{e}_1)\ v \rightarrow [v/x][\text{fix}\ f\ \lambda x.\dot{e}_1/f]\dot{e}_1$

- *경우와 비슷한 경우.*

$\dot{e} = \text{con}\ x\ v \rightarrow x \cdot v$

$\dot{e} = \text{decon}\ x \cdot v \rightarrow v$

$\dot{e} = \text{case}\ x \cdot v\ x\ e_1\ e_2 \rightarrow e_1$

$\dot{e} = \text{case}\ x \cdot v\ x'\ e_1\ e_2 \rightarrow e_2 \qquad (x' \neq x)$

$\dot{e} = \text{handle}\ v\ x\ \lambda x.e_1 \rightarrow v$

- *경우와 비슷한 경우.*

$\dot{e}_1 \rightarrow \dot{e}_1'$ *라고 하자.*

$\check{e} = \dot{e}_1\ e_2 \rightarrow \dot{e}_1'\ e_2$

$\check{e} = v\ \dot{e}_1 \rightarrow v\ \dot{e}_1'$

$\check{e} = \text{con}\ x\ \dot{e}_1 \rightarrow \text{con}\ x\ \dot{e}_1'$

$\check{e} = \text{decon}\ x\ \dot{e}_1 \rightarrow \text{decon}\ x\ \dot{e}_1'$

$\check{e} = \text{case}\ \dot{e}_1\ x\ e_2\ e_3 \rightarrow \text{case}\ \dot{e}_1'\ x\ e_2\ e_3$

$\check{e} = \text{raise}\ \dot{e}_1 \rightarrow \text{raise}\ \dot{e}_1'$

$\check{e} = \text{handle}\ \dot{e}_1\ x\ \lambda x.e_3 \rightarrow \text{handle}\ \dot{e}_1'\ x\ \lambda x.e_2$

$\check{e}_1 \rightarrow e_1'$ *라고 하자.*

$\check{e} = v\ \check{e}_1 \rightarrow v\ e_1'$

$\check{e} = \text{con}\ x\ \check{e}_1 \rightarrow \text{con}\ x\ e_1'$

$\check{e} = \text{decon}\ x\ \check{e}_1 \rightarrow \text{decon}\ x\ e_1'$

$\check{e} = \text{case}\ \check{e}_1\ x\ e_2\ e_3 \rightarrow \text{case}\ e_1'\ x\ e_2\ e_3$

$\check{e} = \text{raise}\ \check{e}_1 \rightarrow \text{raise}\ e_1'$

$\check{e} = \text{handle}\ \check{e}_1\ x\ \lambda x.e_3 \rightarrow \text{handle}\ e_1'\ x\ \lambda x.e_2$

- *경우와 비슷한 경우.*

$\check{e} = (\text{fix}\ f\ \lambda x.e_1)\ v \rightarrow [v/x][\text{fix}\ f\ \lambda x.e_1/f]e_1$

$\check{e} = \text{case}\ x \cdot v\ x\ e_1\ e_2 \rightarrow e_1$

$\check{e} = \text{case}\ x \cdot v\ x'\ e_1\ e_2 \rightarrow e_2 \qquad (x' \neq x)$

$\check{e} = \text{handle}\ \underline{x \cdot v}\ x\ \lambda x.e_1 \rightarrow [x \cdot v/x]e_1$

- *경우와 비슷한 경우.*

$\check{e} = \text{decon}\ x \cdot v \rightarrow v$

$\check{e} = \text{handle}\ v\ x\ \lambda x.e_1 \rightarrow v$

$\check{e} = \text{raise}\ x \cdot v \rightarrow \underline{x \cdot v}$

$\check{e} = \underline{x \cdot v}\ e_1 \rightarrow \underline{x \cdot v}$

$\check{e} = (\lambda x.e_1)\ \underline{x \cdot v} \rightarrow \underline{x \cdot v}$

$\check{e} = (\text{fix}\ f\ \lambda x.e_1)\ \underline{x \cdot v} \rightarrow \underline{x \cdot v}$

$\check{e} = \text{con}\ x\ \underline{x \cdot v} \rightarrow \underline{x \cdot v}$

$\check{e} = \text{decon}\ \underline{x \cdot v} \rightarrow \underline{x \cdot v}$

$\check{e} = \text{case}\ \underline{x \cdot v}\ x'\ e_1\ e_2 \rightarrow \underline{x \cdot v}$

$\check{e} = \text{handle}\ \underline{x \cdot v}\ x'\ \lambda x.e_1 \rightarrow \underline{x \cdot v} \qquad (x' \neq x)$

$\check{e} = \text{raise}\ \underline{x \cdot v} \rightarrow \underline{x \cdot v}$□

지금까지의 $\approx$와 $\|\cdot\|$에 대한 보조정리들과 사실들을 가지고 정리 3을 증명할 수 있다.

증명 (정리 3) *모든 프로그램들은 일반적인 코드 표현들이다. 따라서 $\mathcal{S}$는 $\mathcal{S}$로 정보가 표시된다.*

(→)

$$\mathcal{S} \xrightarrow{*} v \Rightarrow \dot{T}(Annotate(\mathcal{S})) \approx \dot{T}(v)$$
(보조정리 4과 사실 3)
$$\Rightarrow \dot{T}(Annotate(\mathcal{S})) \xrightarrow{*} \dot{T}(v).$$
(≈의 정의와 함께 $\dot{T}(v)$이 값이므로)

(⇐) 증명해야 하는 사실은 다음과 같다.

$$\neg(\mathcal{S} \xrightarrow{*} v) \Rightarrow \neg(\dot{T}(Annotate(\mathcal{S})) \xrightarrow{*} \dot{T}(v))$$

이는 다음과 동치이다.

$$\|\mathcal{S}\| = \infty \Rightarrow \|\dot{T}(Annotate(\mathcal{S}))\| = \infty.$$

결과를 부정하면 다음과 같이 된다.

$$\|\mathcal{S}\| = \infty \wedge \|\dot{T}(Annotate(\mathcal{S}))\| = m \quad (m<\infty).$$

임의의 정수 $n<\infty$에 대해서, 계산 과정의 길이가 n인 $\mathcal{S}$이 존재한다.

$$\mathcal{S} = \mathcal{S}_0 \rightarrow \mathcal{S}_1 \rightarrow \ldots \rightarrow \mathcal{S}_n.$$

보조정리 4에 의해서,

$$\dot{T}(\mathcal{S}) = \dot{T}(\mathcal{S}_0) \approx \dot{T}(\mathcal{S}_1) \approx \ldots \approx \dot{T}(\mathcal{S}_n).$$

이고

$$\|\dot{T}(\mathcal{S}_i)\| - k_i = \|\dot{T}(\mathcal{S}_{i+1})\|$$

을 만족하는 $0<k_i<\infty$ $(0\le i<n)$가 존재한다. 따라서 다음이 만족한다.

$$\|\dot{T}(\mathcal{S}_0)\| - \sum_{i=1}^{n-1} k_i = \|\dot{T}(\mathcal{S}_n)\| \ge 0. \quad (1)$$

(1) 이 모든 n에 대해서 성립해야 하므로 가능한 경우는 $\|\dot{T}(\mathcal{S}_0)\| = \infty$인 경우뿐이므로 모순이다.

(⇒)와 (⇐)에 의해서 정리가 증명된다. □

9. 개선할 점

9.1 CPS변환의 두 단계 표현

우리의 변환을 실제로 구현하기 위해서는 결과 프로그램을 좀더 간단히 하는 작업을 수행하여야 한다. 이렇게 간단하게 만드는 작업은 변환을 위해 부가적으로 추가되는 계산(administrative reduction)들이라고 불린다 [10].

우리는 Danvy와 Filinski의 방법을 우리의 선택적인 CPS 변환에 적용하였다 [4]. 그리고 우리는 더 이상 변환을 위해 부가적으로 추가되는 계산이 존재하지 않는 간단화된 결과 프로그램을 생성할 수 있었다.

그림 6은 위의 방법을 적용하기 위해서 우리의 CPS 변환을 두 단계 표현으로 바꾼 것이다 [4].

9.2 외부 함수들과의 연결

프로그램을 부분들로 나누어 작성한다면, 일부분은 CPS의 형태로 변환되지 않을 수도 있다. 표준 함수들의 라이브러리와 같은 외부 부분들은 여전히 raise와 handle구문을 사용하고 있을 것이다. 그리고 이 논문의 변환에 의해서 변환된 부분들은 CPS 형태로 바뀌었을 것이고 예외상황 처리기에 대한 정보는 두 번째의 남은 할 일을 이용해서 나타날 것이다.

$\bot_K = \overline{\lambda}x.\, x$과 $\bot_H = \overline{\lambda}x.\, \underline{\texttt{raise}}\, x$를 정의하고 @로 함수 적용을 나타낸다.
일반적인 코드 표현인 $\dot{e}$을 위한 두 단계 변환 $\dot{T}(\dot{e})$.

$$\begin{aligned}
\dot{T}(1) &= 1 \\
\dot{T}(x) &= x \\
\dot{T}(\lambda x.\, e) &= \underline{\lambda} x.\, \underline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e) \\
&\qquad \langle \overline{\lambda} m.\, \underline{@}\, K\, m, \overline{\lambda} m.\, \underline{@}\, H\, m\rangle \\
\dot{T}(\texttt{fix}\, f\, \lambda x.\, e) &= \underline{\texttt{fix}}\, f\, \underline{\lambda} x.\, \underline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e) \\
&\qquad \langle \overline{\lambda} m.\, \underline{@}\, K\, m, \overline{\lambda} m.\, \underline{@}\, H\, m\rangle \\
\dot{T}(@\, \dot{e}_1\, \dot{e}_2) &= \underline{@}\, (\underline{@}\, \dot{T}(\dot{e}_1)\, \dot{T}(\dot{e}_2)) \\
&\qquad \langle \underline{\lambda} a.\, \overline{@}\, \bot_K\, a, \underline{\lambda} a.\, \overline{@}\, \bot_H\, a\rangle \\
\dot{T}(\texttt{con}\, \kappa\, \dot{e}) &= \underline{\texttt{con}}\, \kappa\, \dot{T}(\dot{e}) \\
\dot{T}(\texttt{decon}\, \dot{e}) &= \underline{\texttt{decon}}\, \dot{T}(\dot{e}) \\
\dot{T}(\texttt{case}\, \dot{e}_1\, \kappa\, \dot{e}_2\, \dot{e}_3) &= \underline{\texttt{case}}\, \dot{T}(\dot{e}_1)\, \kappa\, \dot{T}(\dot{e}_2)\, \dot{T}(\dot{e}_3) \\
\dot{T}(\texttt{handle}\, \dot{e}_1\, \kappa\, \lambda x.\, e_2) &= \dot{T}(\dot{e}_1) \\
\dot{T}(\texttt{handle}\, \check{e}_1\, \kappa\, \lambda x.\, \dot{e}_2) &= \overline{@}\, \check{T}(\check{e}_1) \\
&\qquad \langle \bot_K, \overline{\lambda} v.\, \underline{\texttt{case}}\, v\, \kappa\, (\underline{@}\, \dot{T}(\lambda x.\, \dot{e}_2)\, v)\, (\overline{@}\, \bot_H\, v)\rangle
\end{aligned}$$

예외상황 가능한 코드 표현인 $\check{e}$을 위한 두 단계 변환 $\check{T}(\check{e})$.

$$\begin{aligned}
&\check{T}(@\, e_1\, e_2) = \\
&\quad \overline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e_1) \\
&\quad \langle \overline{\lambda} f.\, \overline{@}\, \check{T}(e_2)\, \langle \overline{\lambda} v.\, (\underline{@}\, (\underline{@}\, f\, v)\, \langle \underline{\lambda} a.\, \overline{@}\, K\, a, \underline{\lambda} a.\, \overline{@}\, H\, a\rangle), H\rangle, H\rangle \\
&\check{T}(\texttt{con}\, \kappa\, e) = \\
&\quad \overline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e)\, \langle \overline{\lambda} v.\, \overline{@}\, K\, (\underline{\texttt{con}}\, \kappa\, v), H\rangle \\
&\check{T}(\texttt{decon}\, e) = \\
&\quad \overline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e)\, \langle \overline{\lambda} v.\, \overline{@}\, K\, (\underline{\texttt{decon}}\, v), H\rangle \\
&\check{T}(\texttt{case}\, e_1\, \kappa\, e_2\, e_3) = \\
&\quad \overline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e_1) \\
&\qquad \langle \overline{\lambda} v.\, \underline{\texttt{case}}\, v\, \kappa\, (\overline{@}\, \check{T}(e_2)\, \langle K, H\rangle)\, (\overline{@}\, \check{T}(e_3)\, \langle K, H\rangle), H\rangle \\
&\check{T}(\texttt{handle}\, e_1\, \kappa\, \lambda x.\, e_2) = \\
&\quad \overline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e_1) \\
&\quad \langle K, \overline{\lambda} v.\, \underline{\texttt{case}}\, v\, \kappa\, (\underline{@}\, (\underline{@}\, \dot{T}(\lambda x.\, e_2)\, v)\, \langle \underline{\lambda} a.\, \overline{@}\, K\, a, \underline{\lambda} a.\, \overline{@}\, H\, a\rangle) \\
&\qquad\qquad (\overline{@}\, H\, v)\rangle \\
&\check{T}(\texttt{raise}\, e) = \\
&\quad \overline{\lambda}\langle K, H\rangle.\, \overline{@}\, \check{T}(e)\, \langle H, H\rangle
\end{aligned}$$

그림 6 선택적 CPS 변환의 두 단계 표현

따라서 우리는 이러한 외부의 부분과 논문의 변환에 의해서 변환된 부분사이의 연결을 제공해야 한다. 이러한 연결에는 다음과 같은 두 가지 경우가 가능하다.

- 첫째로 변환된 프로그램 부분을 외부에 사용할 수 있도록 해주어야 한다. 외부에서 사용할 수 있도록 시그너쳐에 명시되어 있는 값들에 대해서는 외부에서 사용할 수 있는 형태로 만들어서 넘겨주어야 한다.
- 둘째로 외부의 프로그램 부분을 변환된 부분에서 사용할 수 있어야 한다. 외부의 값을 사용할 때는 변환된 부분에서 사용할 수 있는 형태로 바꾸어서 사용해야

한다. 그리고 외부에서 함수의 인자의 형태로 넘겨주는 것도 변환된 부분에서 쓸 수 있는 형태로 바꾸어서 사용해야 한다.

우리는 위의 두 가지 경우를 상호 재귀적인 형태의 함수인 *Export*와 *Import*로 나타내었다.

$$Export(e, \iota) = e$$
$$Export(e, \tau_1 \rightarrow \tau_2) = \lambda x. Export(e\ (Import(x, \tau_1))\langle K, H\rangle, \tau_2)$$

$$Import(e, \iota) = e$$
$$Import(e, \tau_1 \rightarrow \tau_2) = \lambda x. \lambda\langle K, H\rangle.\ \text{handle}\ (k\ Import(e\ Export(x, \tau_1), \tau_2))\ \_\ \lambda x. h\ x$$

위에서 쓰인 새로운 패턴인 '_'은 모든 예외상황에 대응하는 패턴이다.

9.3 더 엄격하게 선택하는 CPS변환 (선택적 함수 변환)

이 논문의 CPS 변환은 여러 가지의 개선이 가능하다. 지금까지의 변환은 모든 함수와 함수 호출 부분을 두 개의 남은 할 일을 주고받도록 바꾸고 있다. 이러한 무조건적인 변환은 함수 호출을 할 때 일반적인 함수와 예외상황 가능한 함수 모두 호출될 수 있다는 상황에 대한 간단한 해결책이다. 이러한 문제 때문에 우리는 일반적인 함수들도 모두 남은 할 일을 받도록 바꾸어 주어야 한다. 또한 이러한 일반적인 함수들이 일반적인 함수 호출 코드 표현에서도 호출되기 때문에, 일반적인 함수 호출 코드 표현들도 모두 두 개의 기본 남은 할 일들을 넘겨주도록 바꾸어야 한다.

만약 함수와 함수 호출 코드 표현들을 좀더 세분화해서 나눈다면, 이러한 불필요한 남은 할 일들의 사용을 줄일 수 있다. 우리는 일반적인 함수를 다음과 같은 두 가지 경우로 나눌 수 있다 :

- 순수한 일반적인 함수들 : 확실한 일반적인 함수 호출들에서만 사용되는 일반적인 함수들을 의미한다.
- 순수하지 않은 일반적인 함수들 : 예외상황 가능한 함수 호출이나 순수하지 않은 일반적인 함수 호출에서 호출 될 수 있는 함수들이다.

일반적인 함수들과 대칭적으로, 함수 호출들도 두 가지로 나눌 수 있다.

- 순수한 일반적인 함수 호출들 : 순수한 일반적인 함수들만을 호출하는 함수 호출들을 의미한다.
- 순수하지 않은 일반적인 함수 호출들 : 예외상황 가능한 함수나 순수하지 않은 일반적인 함수 호출을 호출할 수 있는 함수 호출들을 의미한다.

위의 것들 중에서 순수하지 않은 일반적인 함수들만을 남은 할 일들을 받도록 바꾸고, 순수하지 않은 일반적인 함수 호출들만을 기본 남은 할 일을 넘겨주도록 바꾸면 된다.

위의 세분화된 분류를 적용하려면 그림 3에 있는 정보 표시 규칙에 다음의 두 개의 정보 표시 규칙을 더해주면 된다 :

아래의 규칙에서 $\breve{e}$는 순수하지 않은 일반적인 코드 표현들을 의미한다.

$$\frac{\lambda x.\dot{e} \in Closure\_analysis_\tau(e_1) \quad \breve{e} = e_1\ e_2}{\widetilde{\lambda x.\dot{e}}}$$

$$\frac{\overline{\lambda x.\dot{e}} \in Closure\_analysis_\tau(e_1) \quad \dot{e} = \dot{e}_1\ \dot{e}_2}{\dot{e}_1\ \dot{e}_2}$$

위의 세분화된 정보 표시를 사용하는 새로운 변환 함수는 쉽게 정의할 수 있다.

9.4 현재의 제한점

첫째, 이 논문의 선택적 변환에 의해서 변환된 프로그램에 대한 타입 결정 시스템이 없다. 현재는 이 논문의 변환에 의해서 변환된 프로그램은 ML의 타입 시스템을 통과하지 못하는 경우가 있다.

예를 들면, 다음과 같은 프로그램은 타입 시스템을 통과하는 적절한 ML 프로그램이다.

```
exception K
(fn f => if handle ((f 10) > 10) _ => true
         then handle ((f 10) + 10) _ => 0
         else 0)
(fn x => raise K)
```

그리고 위의 프로그램을 이 논문의 변환으로 변환하면 다음과 같다.

```
datatype K
(fn f => if (f 10) ((fn v => v > 10), (fn e => true))
         then (f 10) ((fn v => v + 10), (fn e => 0))
         else 10)
(fn x => fn (k,h) => h K)
```

위의 프로그램에서 변수 f는 ML의 타입 시스템에서 타입이 결정되지 않는다. 만약 위의 프로그램이 모두 CPS의 형태로 변환될 경우에는 변환된 프로그램의 타입을 결정하는 것이 가능하다.

우리의 변환은 SML/NJ 컴파일러에서 몇몇의 예외상황을 많이 사용하는 프로그램의 실행 시간을 줄이는 것이 가능하다. 그러나, 예외상황이 자주 사용되지 않는 일반적인 프로그램에 대해서는 변환하지 않는 경우 보다 더 느려지는 경우가 발생하기 때문에 아직 더 연구가 필요하다. 이는 현재도 불필요한 남은 할 일을 주고받는 것이 많다는 것으로 이를 줄일 방법을 아직 찾고

있다.

CAML 컴파일러에서는 우리의 변환이 프로그램의 수행시간을 줄이지 못했다. 예외상황을 많이 사용하는 프로그램의 경우에도 변환된 프로그램이 더 느려졌다.

10. 결 론

이 논문에서, 우리는 선택적인 CPS 변환을 기반으로 하는 예외상황 처리의 또 다른 구현에 대해서 살펴보았다. 예외상황 처리기의 전역 스택을 사용하는 대신에, 함수에 정상적인 수행과정에 대한 남은 할 일과 예외적인 수행과정에 대한 남은 할 일을 넘겨주는 방법을 사용하였다.

남은 할 일을 넘겨주는 추가비용을 줄이기 위해서 이광근과 류석영의 예외상황 분석기 [13]에서 제공되는 정보를 이용하여 필요 없이 남은 할 일을 주고받는 경우를 줄였다. 그리고 우리의 선택적인 CPS 변환이 프로그램의 의미를 그대로 유지한다는 것을 엄밀한 방법으로 증명하였다. 그러므로 우리는 우리의 변환을 임의의 프로그램에 안전하게 적용할 수 있다.

우리의 선택적인 CPS 변환은 예외상황을 많이 사용하는 프로그램의 실행시간을 줄일 수 있음을 알 수 있었다. 그러나 일반적인 프로그램에 대해서는 우리의 변환에 의해서 추가되는 비용이 SML/NJ의 예외상황을 처리하는 비용보다 더 크게 나타났다.

프로그램을 부분으로 나누어서 프로그래밍 하는 환경에서, 우리는 CPS 변환에 의해서 변환된 부분과 변환되지 않은 외부 부분들 사이의 연결을 제시하였다. 이로서 예외상황을 많이 사용하는 부분만을 우리의 방법으로 변환하고 다른 부분은 변환하지 않은 뒤에 서로 연결하는 방법이 가능하다.

참 고 문 헌

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[2] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation (Née Lisp and Symbolic Computation)*, 11(2), 1998.

[3] Olivier Danvy and Dirk Dussart. CPS transformation after binding-time analysis. Unpublished note, Department of Computer Science, University of Aarhus, April 1995.

[4] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361-391, December 1992.

[5] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195-212, 1993.

[6] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627-648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[7] Dirk Dussart. *Topics in program specialization and analysis for statically typed functional languages.* PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, May 1997.

[8] 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.

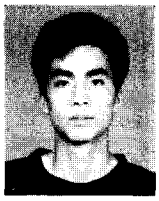
[9] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(2):303-319, 1997.

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

[11] 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.

[12] Paul Steckler and Mitchell Wand. Selective thunkification. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 162-178, Namur, Belgium, September 1994. Springer-Verlag.

[13] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of unaught exceptions in SML programs. In Pascal Van Hentenryck, editor, *Static Analysis*, number 1302 in Lecture Notes in Computer Science, pages 98-113, Paris, France, September 1997. Springer-Verlag.



김 정 택

1997년 한국과학기술원 전산학과 학사(B.S.). 1999년 한국과학기술원 전산학과 석사(M.S.). 1999년 ~ 현재 한국과학기술원 전산학과 박사과정.

이 광 근

1987년 B.S. 계산통계학, 서울대학교 자연과학대학. 1990년 M.S. Computer Science, University of Illinois at Urbana-Chanpaign. 1993년 Ph.D. Computer Science, University of Illinois at Urbana-Champaign. 1993년 ~ 1995년 Member of Technical Staff, Software principles Research Dept., At & T Bell laboratories. 1995년 ~ 현재 한국과학기술원 전산학과 조교수.