Standard ML 프로그램의 실행 중 처리되지 않는 예외 상황을 예측하는 방법

류석영*○ 이광근*
* 한국과학기술원 전산학과

A Practical Method for Estimating Uncaught Exceptions in Standard ML Programs

Sukyoung Ryu* Kwangkeun Yi*
* Dept. of Computer Science
Korea Advanced Institute of Science and Technology

요 약

본 논문에서는 Standard ML(SML)[7, 6] 프로그램의 실행 중 처리되지 않는 예외 상황을 실행 전에 미리 예측하는 분석 방법을 제안한다.

이 분석 방법은 SML/NJ[2, 5] 컴파일러가 제공하는 타잎 정보를 이용하여 프로그램의 함수 흐름도를 먼저 예측하고, 이를 이용하여 예외 상황의 흐름을 방정식으로 표현한 후, 반복 부동점 방법으로 방정식을 풀어서 처리되지 않을 수 있는 예외 상황들을 예측한다. 이 방법의 속도는 현재 25 ~ 700 SML-lines/sec이고, 적절한 비용 대 성능비를 가지고 있는 것으로 밝혀졌다. 예를 들어, ML-Lex 프로그램을 분석하는데 약 23초가 걸렸고, 10개의 예외 상황 중에서 실제로 처리되지 않는 3개를 포함해서 4개의 예외 상황이 처리되지 않을 것으로 분석되었다.

이 분석으로 SML 프로그램의 갑작스런 중단을 미리 예측할 수 있으므로, SML 프로그램의 안전성을 한층 더 증가시킬 수 있다.

1. 서론

SML은 예외 상황의 발생과 소멸을 프로그래머가 자유롭게 표현할 수 있다. 예외 상황은 함수의 인수나 결과 값으로 전달될 수도 있고, 메모리에 저장되었다가 발생될 수도 있다.

예외 상황 관리 방법은 단순히 프로그램 수행 중에 생기는 잘못을 처리하는 데에만 사용되지는 않는다. 여러 겹으로 겹쳐져 있는 프로그램의 제어 구조(control structure)를 한 번에 빠져 나오는 데에도 사용될 수 있고, 같은 연산 결과를 경우에 따라 다르게 처리할 때에도 사용될 수 있다.

하지만 예외 상황을 자유롭게 사용하는 것이 프로그램의 안전도를 해치는 상황으로 발전할 수도 있다. 만일 발생된 예외 상황이 그 관리 함수(exception handler)의 허술로 소멸될 수 없을 경우에는 프로그램이 갑작스럽게 그 실행을 중단해버리기 때문이다. 그러므로 발생하는 예외 상황들이 빠짐없이 소멸될 수 있도록 예외 상황 관리 함수를 적절한 위치에 설치해주어야 한다.

본 연구의 목표는 SML 프로그램 실행 중 발생할 수 있는 예외 상황 관리의 허점을 예측하는 분석기를 개발하는 것이다. SML 프로그램이 가질 수 있는 안전도의 유일한 허점은 해제되지 못한 예외 상황이므로, 컴파일러가 이를 예측할 수 있다면 프로그램의 안전도를 증진시킬 것이다.

본 연구에서는 SML/NJ 컴파일러가 제공하는 타잎을 이용해서 프로그램의 흐름을 예측하고, 그 정보로 예외 상황 관리의 허점을 예측하는 방법을 제안한다. 이 방법을 구현하여 실제 SML 프로그램을 분석했고, 구현된 방법은 기존의 방법보다 매우 향상된 비용 대 성능비를 가지고 있는 것으로 나타났다.

2. SML의 예외 상황 관리 방법(exception mechanism)

SML에서의 예외 상황은 각각 이름을 갖고 있고, 인수를 가질 수도 있다. 예를 들어, 다음 표현은

```
Error("at line 10")
```

문자열을 인수로 갖는 예외 상황 `Error`를 만든다. 예외 상황 이름인 `Error`는 사용되기 전에 미리 선언되어야 한다.

```
exception Error of string
```

예외 상황은 다음과 같은 표현에 의해 발생된다.

$$\texttt{raise}\ e$$

e의 결과는 예외 상황이어야 한다. 예를 들어, `raise !x`는 메모리 `x`에 저장된 예외 상황을 발생시키는 것이 된다.

예외 상황이 발생되면, 그 예외 상황을 처리할 수 있는 예외 상황 관리 함수를 찾아서 현재까지 계산된 순서를 거슬러 올라간다.

SML에서 예외 상황 관리 함수의 문법은 다음과 같다.

$$e\ \texttt{handle}\ p_1\ \texttt{=>}\ e_1\ |\cdots|\ p_n\ \texttt{=>}\ e_n$$

e의 결과로 발생한 예외 상황 이름이 p_k와 같으면 e_k가 계산되고, 같은 이름이 없으면 다른 예외 상황 관리 함수를 만날 때까지 계속해서 계산 과정을 거슬러 올라간다. 발생한 예외 상황을 처리해주는 예외 상황 관리 함수가 없을 경우에는 프로그램이 멈추게 된다.

SML/NJ 1.08[a]	ML-LEX[b]	ML-YACC[c]
91.93%[d] (262/285)	100% (33/33)	96.97% (32/33)

[a]SML 컴파일러

[b]어소 분석기(lexical analyzer) 자동 생성 도구

[c]어법 분석기(parser) 자동 생성 도구

[d]퍼센티지(%)는 프로그램에 나타난 예외 상황 관리 표현식의 총 갯수 중에서 간단한 방법으로 표현된 갯수들의 비율이다. "간단한 방법"이란, 예외 상황의 발생 표현식에 직접 예외 상황의 이름을 표현한 경우를 말한다. SML에서는 예외 상황이 다른 여타 값과 구분없이 사용될 수 있기 때문에, "raise f()"와 같이 어떤 예외 상황을 발생시키는지를 프로그램의 텍스트만을 보고는 알 수 없는 복잡한 경우가 있다.

그림 1: 예외 상황 발생 표현식의 간단한 정도

3. 문제 정의

SML의 예외 상황은 함수의 인수로 전달되거나 결과로 전달받을 수 있으므로, 프로그램을 봐서는 발생되는 예외 상황이 소멸되는지 알 수 없다. 예를 들어, 아래 함수 f에서

$$\texttt{f(x) = } \cdots \texttt{raise x} \cdots$$

어떤 예외 상황이 발생되는지를 알려면 함수 f가 어떤 예외 상황을 인수로 전달받는지 알아야 한다. 또, 발생되는 예외 상황이 소멸될 수 있는지를 알려면 함수 f를 호출할 때 어떤 예외 상황 관리 함수가 제공되는지도 알아야 한다.

또 다른 예로, 함수를 인수로 받는 함수 f의 경우,

$$\texttt{f(g) = } \cdots \texttt{g(x) handle E =>} \cdots$$

g(x)에서 어떤 예외 상황이 발생하는지 알려면 함수 f가 어떤 함수를 인수로 받는지 알아야 한다. 또, f의 계산 도중에 발생하는 예외 상황을 없애기 위해서 f를 호출할 때 어떤 예외 상황 관리 함수를 제공하는지도 알아야 한다.

SML에서는 예외 상황이 함수 사이에서 자유롭게 전달될 수 있으므로, 요약 해석(abstract interpretation)[3]을 사용하는 것이 자연스럽다. 이 방법으로 예외 상황 분석을 하면 함수 흐름의 분석이 자연스럽게 표현되기 때문이다.

그러나 요약 해석을 이용한 분석은 그 비용이 너무 크다. 함수 흐름을 파악하면서 예외 상황 흐름을 분석하므로, 두 가지 흐름을 동시에 처리해서 비용이 커지는 것이다.

하지만 그림 1에서 볼 수 있듯이, SML 프로그램에서 예외 상황을 사용하는 예가 대부분 간단하고,예외 상황에 연계된 함수 흐름도 간단한 경우가 대부분이다. 그러므로 적은 비용의 간단한 방법으로 함수 흐름을 분석해도 예외 상황 분석에는 효과적일 수 있다.

본 논문에서는 예외 상황 분석과 함수 흐름 예측을 분리하여 비용을 줄이는 방법을 제안한다. 프로그램의 함수 흐름도는 SML/NJ 컴파일러의 타잎 유추를 통해 거의 비용을 들이지 않고 예측하고, 그것을 이용하여 예외 상황 흐름을 표현하는 연립 방정식을 만든다. 만들어진 방정식의 해는 전통적인 반복 부동점 방법을 이용하여 계산한다.

4. SML 프로그램의 함수 흐름도 예측

함수 호출식 "$e_1\ e_2$"에 대해서, e_1이 어떤 함수인지 아는 것이 프로그램의 흐름을 알아내는 것이다. 만일, e_1이 함수 이름도 아니고, "fn x => ⋯"와 같이 곧바로 정의된 함수식도 아닐 경우에는 e_1의 타잎과 같아질 수 있는(unifiable[8]) 타잎의 함수가 모두 호출된다고 가정한다.

이 방법은 정확도가 문제될 수 있지만, SML 프로그램 함수가 다양한 타잎을 갖으므로 유용하게 사용될 수 있다. 그림 2는 함수들이 매우 다양한 타잎을 갖고 있고, 그 타잎과 같아질 수 있는 타잎이 많지 않음을 보이고 있다.

프로그램	#fn-tys[a] (함수 갯수)	#same ty-fns 평균	#same ty-fns σ[c]	#unif-tys[b] 평균	#unif-tys σ
KB.sml[d]	129 (177)	1.30	1.03	4.68	7.92
ml-lex.sml[e]	105 (173)	1.56	1.69	2.78	10.27
pathname.sml[f]	25 (48)	1.80	1.23	0	0
string-cvt.sml[g]	24 (77)	2.92	3.11	0	0
compiler.sml[h]	156 (186)	1.14	0.75	10.68	16.70

[a]서로 다른 함수 타잎의 갯수

[b]타잎이 같아질 수 있는(unifiable) 함수 타잎의 갯수

[c]표준 편차

[d]Knuth-Bendix.sml

[e]SML/NJ 108.13의 ML-Lex 프로그램

[f]SML/NJ 108.13 라이브러리 프로그램

[g]SMl/NJ 108.13 라이브러리 프로그램

[h]4학년 컴파일러 과목의 프로젝트 프로그램

그림 2: SML 프로그램의 함수와 그 타잎

5. 전처리(Preprocessing)

예외 상황의 흐름이 프로그램 텍스트에 최대한으로 표현되게 하기 위해서, SML 프로그램에 대해 다음과 같은 처리를 한 후에 분석이 진행된다.

5.1 예외 상황 관리 함수

다음과 같은 예외 상황 관리 표현에서,

$$e \ \texttt{handle}\ p_1 \ \texttt{=>}\ e_1 \mid \cdots \mid p_n \ \texttt{=>}\ e_n$$

handle의 오른쪽 부분을 다음과 같이 함수의 모습으로 바꾼다. (이를 예외 상황 함수라고 부른다.)

$$\lambda \texttt{x.case x of}\ p_1 \ \texttt{=>}\ e_1 \mid \cdots \mid p_n \ \texttt{=>}\ e_n$$
$$\mid \_ \ \texttt{=> -raise x}\ p_1 \cdots p_n$$

즉, e에서 발생한 예외 상황을 x로 받아서 case문을 통해서 처리되는 것으로 한다. 이 때, 항상 -raise를 덧붙여서 예외 상황 함수가 처리하지 못하는 예외 상황을 다시 발생시키도록 한다.

5.2 -raise

-raise는 발생시킬 예외 상황에 포함되지 않는 예외 상황들을 알고 있을 때 사용된다. 주로, 예외 상황 관리 함수에서 처리하지 못하고 빠져나오는 예외 상황을 다시 발생시키는데 쓰인다. 5.1절의 예에서 처리하지 못하는 예외 상황은 $p_1,\cdots,p_n$을 제외한 것이므로,

$$\texttt{-raise x}\ p_1 \cdots p_n$$

을 사용하여 예외 상황을 다시 발생시키면서, 다시 발생하는 예외 상황이 $p_1,\cdots,p_n$은 아니라는 것을 나타낸다.

5.3 +raise

+raise는 발생시킬 예외 상황이 알려져 있을 때 사용된다. 주로, 예외 상황 관리 함수가 예외 상황의 인수를 모두 처리하지 못해서 다시 발생시키는 데에 쓰인다. 예를 들어,

```
exception E of int list
...
e handle E nil => 1
```

의 경우에, 이 예외 상황 관리 함수는 예외 상황의 인수가 nil이 아닐 때에는 처리하지 못한다. 이 경우에는 다음과 같이 +raise를 사용하여 예외 상황을 다시 발생시키면서, 다시 발생한 예외 상황이 E뿐임을 나타낸다.

```
e handle λx.case x of E nil => 1
                    | E _ => +raise x E
                    | _ => -raise x E
```

6. 예외 상황 분석 방정식의 도출

4.절에서 구한 함수 흐름도를 이용해서, 발생할 수 있는 예외 상황들의 집합을 표현하는 방정식을 만든다. 첫번째 방정식은 함수 f를 실행하는 동안 발생해서 소멸하지 않을 수 있는 예외 상황들의 집합인데, P_f로 표시한다. 두번째 방정식은 함수 f를 실행하는 동안 발생할 수 있는 예외 상황들의 집합이고, X_f로 표시한다.

6.1 소멸 불가능한 예외 상황에 대한 방정식

각 함수에 대해서 궁극적으로 알고 싶은 것은, 그 함수를 실행하는 동안 처리되지 않을 수 있는 예외 상황들의 집합이다. 함수 f에 대한 이러한 집합을 P_f로 나타내는데, 다음과 같은 세 가지 종류의 예외 상황들로 이루어진다.

- `raise` 표현에 의해서 발생되는 예외 상황들
  다음의 예와 같이 함수 f 안에서 `raise` 표현에 의해서 발생하는 예외 상황들이다.

  $$\texttt{fun f() = } \cdots \texttt{(raise } e\texttt{)} \cdots$$

  집합식으로 표현하면,

  $$\bigcup_{e \in R(f)} Trim(e, X_f)$$

  $\mathrm{R}(f) = \{e |\ e$는 f안에 있는 raise 표현식$\}$

  $$Trim(e, X_f) = \begin{cases} \{\kappa\} & \text{if } e = \text{“}\texttt{raise } \kappa\text{”} \\ X_f & \text{if } e = \text{“}\texttt{raise } e'\text{”} \\ X_f \cap_{e'} \{\kappa\} & \text{if } e = \text{“}\texttt{+raise } e'\ \kappa\text{”} \\ X_f \setminus_{e'} \{\kappa_i\} & \text{if } e = \text{“}\texttt{-raise } e'\ \kappa_i\text{”} \\ & \quad 1 \le i \le n \end{cases}$$

  $$X_f \cap_{e'} \{\kappa\} = \begin{cases} X_f & e'\text{이 예외 상황을 인수로 갖는 예외 상황일 경우} \\ X_f \cap \{\kappa\} & \text{아닌 경우} \end{cases}$$

  $$X_f \setminus_{e'} \{\kappa_i\} = \begin{cases} X_f & e'\text{이 예외 상황을 인수로 갖는 예외 상황일 경우} \\ X_f \setminus \{\kappa_i\} & \text{아닌 경우} \end{cases}$$

- 함수 호출에 의해서 발생되는 예외 상황들
  다음의 예와 같이, 함수 f 안에서 호출되는 함수에게 예외 상황 관리 함수가 주어지지 않을 경우이다.

  $$\texttt{fun f() = } \cdots \texttt{(h x)} \cdots$$

  집합식으로 표현하면,

  $$\bigcup_{g \in N(f)} P_g$$

  $\mathrm{N}(f) = \{g |$ 예외 상황 관리 함수없이 f가 g를 호출 $\}$

- 예외 상황 관리 중에 발생되는 예외 상황들
  다음의 예와 같이, 함수 f 안에 있는 예외 상황 관리 함수가 소멸시키지 못하고 재발생시키는 경우이다.

  $$\texttt{fun f() = } \cdots (e \texttt{ handle } \lambda\texttt{x}.e') \cdots$$

  집합식으로 표현하면,

  $$\bigcup_{h \in H(f)} P_h$$

  $\mathrm{H}(f) = \{g |\ g$는 f 안에 있는 예외 상황 함수$\}$

  예외 상황 관리 함수가 소멸시키지 못하는 예외 상황은 다시 발생되어서 계산 순서를 거슬러서 전달된다. 이러한 예외 상황들은 예외 상황 관리 함수 안에서 `-raise`와 `+raise`에 의해 전달된다.

그러므로, P_f에 대한 방정식은 지금까지 알아본 세 가지 경우를 다 합한 것이 된다.

$$P_f = \bigcup_{e \in R(f)} Trim(e, X_f) \cup \bigcup_{g \in N(f)} P_g \cup \bigcup_{h \in H(f)} P_h. \quad (1)$$

6.2 발생 가능한 예외 상황에 대한 방정식

P_f에 대한 방정식을 구하려면 X_f를 알아야 한다는 것을 알 수 있다. X_f는 함수 f를 수행하는 동안에 발생 가능한 예외 상황들의 집합인데, 다음의 네 가지 종류의 예외 상황들로 이루어진다.

- 직접 나타나는 예외 상황들

  ```
  exception Error
  ```

  $K_f = \{\kappa |\ \kappa$는 f 안에 있는 예외 상황$\}$

- 호출한 함수로부터 전달되는 예외 상황들

  $$\texttt{f(g)} = \cdots \texttt{g(x)} \cdots$$

  집합식으로 표현하면,

  $$\bigcup_{g \in \mathrm{From}(f)} X_g$$

  $\mathrm{From}(f) = \{g | f$가 g를 호출하고 g의 결과로 예외 상황을 전달할 수 있는 경우$\}$. *

- 호출하는 함수에서 전달되는 예외 상황들
  첫째로 함수 f가 예외 상황 관리 함수가 아닌 보통 함수일 경우에는, 함수 f를 호출하는 함수 g로부터 전달받는 예외 상황들이다. 집합식으로 표현하면,

  $$\bigcup_{g \in \mathrm{To}(f)} X_g$$

  $\mathrm{To}(f) = \{g | g$가 f를 호출하고 f의 인수로 예외 상황을 가질 수 있는 경우$\}$.

  둘째로 함수 f가 예외 상황 관리 함수인 경우에는, 다음의 예와 같이 e로부터 발생한 예외 상황들이 예외 상황 관리 함수로 전달된다.

  $$\texttt{h(g) = } \cdots e \texttt{ handle } \underbrace{\lambda\texttt{x}.\cdots}_{f}$$

  P_f는 함수에 대해서만 정의되었지만, 이 경우에는 e를 내용으로 하는 가상의 함수 (`fn () =>` e) 가 있다고 가정하고 다룬다. 이때 이 함수를 $hd(f)$로 표현하면, X_f는 $P_{hd(f)}$가 된다.

- 자유 변수(free variable)를 통해 전달되는 예외 상황들

  $$\texttt{fun h(x) = } \cdots \underbrace{\texttt{(fn y => } \cdots \texttt{ x} \cdots \texttt{)}}_{f} \cdots$$

  집합식으로 표현하면,

  $$\bigcup_{g \in Free(f)} X_g$$

  $Free(f) = \cup\{To(f') | f' \in Owner(FV(f)) \cap Scope(f)$ 이고, f'의 인수로 예외 상황을 가질 수 있는 경우 $\}$.
  $FV(f) = \{v | v$가 f 안에서 자유 변수$\}$
  $Owner(x) = \{f | x$는 f의 인수$\}$,
  $Owner(V) = \cup_{v \in V} Owner(v)$
  $Scope(f) = \{g | g$는 f를 둘러싼 함수$\}$.

그러므로, X_f에 대한 방정식은 지금까지 알아본 네 가지 경우를 다 합한 것이 된다.

$$X_f = K_f \cup \bigcup_{g \in From(f)} X_g \cup \bigcup_{g \in To(f)} X_g \cup P_{hd(f)} \cup \bigcup_{g \in Free(f)} X_g. \quad (2)$$

*타입 정보를 이용해서 4.절에서 구한 함수 흐름도를 통해 함수 사이의 호출 관계를 예측한다.

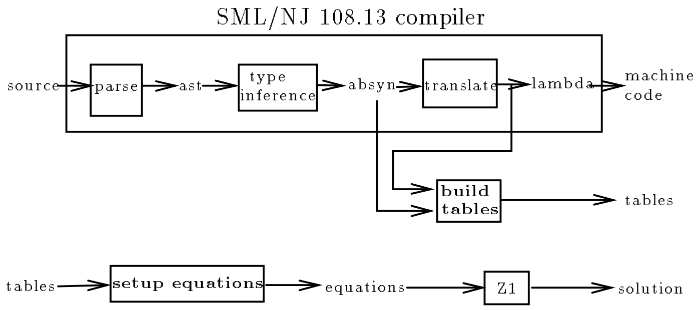

그림 3: 예외 상황 분석기

프로그램	통계				
	#lns[a]	#cls[b]	#exs[c]	#rss[d]	#hds[e]
KB.sml	519	750	1	1	10
ml-lex.sml	1204	1463	10	19	10
pathname.sml	426	267	4	6	3
string-cvt.sml	454	285	1	10	4
compiler.sml	3511	3650	11	34	4

프로그램	분석 비용 (초)		분석 결과
	cfa + setup[f]	solve[g]	
KB.sml	19.98[h]	0.11[i]	1 (1x,1r,10h)[j]
ml-lex.sml	22.9	0.05	4 (10x,19r,10h)
pathname.sml	1.09	0.01	4 (4x,6r,3h)
string-cvt.sml	0.65	0.00[k]	1 (1x,10r,4h)
compiler.sml	116.04	0.05	3 (11x,34r,4h)

[a]SML 프로그램 길이
[b]함수 호출식의 갯수
[c]예외 상황 갯수
[d]예외 상황 발생식의 갯수
[e]예외 상황 관리 함수의 갯수
[f]함수 흐름 분석과 방정식 만들기 : SML/NJ 108.13으로 컴파일, DEC Alpha Server1000(4/200)에서 실행
[g]방정식 풀기 : GCC로 컴파일, Sparc 20에서 실행
[h]SML user+system+gc 시간
[i]C user+system 시간
[j]최상위 함수에서 처리되지 않을 예외 상황의 갯수/ 예외 상황(x), 예외 상황 발생식(r), 예외 상황 관리 함수(h)의 갯수
[k]$\leq$ 10 밀리 초

그림 4: 분석 성능

7. 구현 결과

본 논문에서는 SML/NJ 108.13[1]의 컴파일러 소스를 이용하여 예외 상황 분석기를 구현하였다. SML/NJ 108.13 컴파일러의 내부 표현인 `absyn`을 입력 받아 함수 흐름을 분석한다. 즉, 앞에서 설명한 $From(f)$, $Free(f)$등의 집합들을 테이블로 구현하고 그 테이블을 이용해서 각 함수에 해당하는 방정식(P_f와 X_f)을 만든 후, Z1[10]이라는 도구를 이용해서 푼다. 이 과정은 그림 3과 같다.

각 함수에 대한 P_f와 X_f의 방정식을 만든 후에, 그 방정식의 해는 전통적인 반복 부동점 방법(iterative fixpoint method)에 의해서 구해진다. 본 논문에서는 예외 상황 분석으로 만들어진 방정식을 해결하는 데 Z1을 사용하였다. Z1은 재귀적으로 만들어진 방정식의 집합을 입력받아서, 그 방정식의 가장 작은 해를 구하는 프로그램을 출력한다.

본 논문에서 제안한 방법의 속도는 프로그램에 따라 약 26 SML-lines/sec에서부터 약 700 SML-lines/sec에까지 차이가 많지만, 요약 해석을 이용한 예전 방법(0.3 SML-lines/sec[9])보다는 월등한 속도로 분석을 마칠 수 있다. 분석 결과도 매우 정확한 정보를 제공한다. 예로, Knuth-Bendix.sml, pathname.sml, string-cvt.sml, compiler.sml은, 실제로 발생해서 소멸되지 않는 예외 상황을 찾아내고, ML-lex.sml는 실제로는 3개인데 4개를 찾아낸다.

참고로, 본 논문의 내용과 같은 문제를 또 다른 방법(set-constraint-based type inference[4])으로 진행하고 있는 Berkeley 대학의 Alexander Aiken 교수팀의 현재(1996년 5월) 분석 속도는 약 10 SML-lines/sec(HP 715/60, 256 MB)이고, 분석 정확도도 본 논문의 결과와 비슷한 수준이다.

8. 결론

본 논문에서는 프로그램 실행 중에 발생해서 처리되지 않는 예외 상황이 있는지를 실제로 프로그램을 실행하기 전에 예측할 수 있는 분석 방법을 제안한다. 이 방법은 SML/NJ 컴파일러가 타잎 유추를 통해 제공하는 타잎 정보를 이용해서, 프로그램의 함수 흐름을 분석하여 함수 흐름도를 만들어낸다. 이 함수 흐름도를 이용하여, 발생된 후에 소멸되지 않아서 프로그램을 비정상적으로 끝낼 수 있는 예외 상황들에 대한 방정식을 만든다. 마지막으로, 이 방정식의 가장 작은 해를 구하면 된다.

이 분석을 구현한 결과, 적절한 비용 대 성능비를 가지고 있는 것으로 밝혀졌다. 예를 들어 ML-Lex 프로그램은 예전 방법으로 약 53분이 걸린데 비해, 본 논문에서 제안한 방법으로 약 23초가 걸렸다.

이 분석을 SML/NJ 컴파일러에 장착하기 위해서 다음의 일들이 필요하다. 첫째로, 현재는 SML core만을 분석할 수 있는데 모듈도 분석 가능하게 하여, 전체 프로그램이 없어도 프로그램의 일부를 분석할 수 있게 하는 것이다. 둘째로, 구현 방법을 개선하여 비용 대 성능비를 더욱 향상시키는 것이다.

참고 문헌

[1] ftp://research.att.com/dist/ml.

[2] Andrew W. Appel and David B. MacQueen. Standard ML compiler. In *Lecture Notes in Computer Science*, volume 274. Springer-Verlag, 1987.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[4] Manuel Fahndrich and Alex Aiken, May 1996. (personal communication).

[5] David B. MacQueen and Andrew W. Appel. Standard ML of New Jersey. Technical memorandum, AT&T Bell Labs., 1993.

[6] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[7] Robin Milner, Mads Tofte, and Robert Haper. *The Definition of Standard ML.* MIT Press, 1990.

[8] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[9] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science*, volume 864. Proceedings of the first international static analysis symposium edition, 1994. `http://compiler.kaist.ac.kr/~kwang/paper/94-sas-yi.ps`.

[10] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Proceedings of the Annual ACM Symposium on Principles of Program ming Languages*, pages 246–259, January 1993. `http://compiler.kaist.ac.kr/~kwang/paper/popl93.ps`.