타입환경상의 편리한 프로그램 작성을 위한 도구 *

이욱세, 편기현, 이광근
한국과학기술원 전산학과

Tools for Typeful Programming

Ookseh Lee, Ki-Hyun Pyun and Kwangkeun Yi
Dep. of Computer Science, KAIST

1996년 11월 23일

요 약

타입유추 시스템을 갖춘 언어로 프로그램을 작성하다 보면 컴파일러가 넓은 범위의 오류 정보를 주어 오류 수정이 어려울 때가 있다. 이를 해결하기 위해 두 가지 접근 방법을 취해 보았다. 하나는, 오류 정보를 주는 타입유추 알고리즘 자체에 문제가 있음을 제기하고 더 좁은 범위의 오류 정보를 주는 새로운 타입유추 알고리즘을 제시하였다. 또 하나는 쉽게 각각의 프로그램 부분의 타입 정보를 보여주는 도구를 개발하여 사용자가 타입 정보를 따라 가면서 오류를 수정할 수 있도록 하였다.

1 서론

정적 타입 시스템은 많은 실행시간 오류를 컴파일시간에 미리 알아냄으로써 결과 프로그램의 안정성을 보장해 준다. 타입유추(type inference) 알고리즘은 프로그램의 각각의 구성요소들의 타입을 유추해 냄으로써 사용자의 프로그램 작성에 큰 도움을 줄 뿐만 아니라 프로그램의 타입을 모두 검증한다. 그러나, 프로그램에 오류가 있으면 타입유추가 실패하여 오류 정보를 사용자에게 주는데, 실제 사용자가 오류를 범한 장소에서 먼 장소나 광범위한 장소를 지적하여 오류 수정이 어려운 경우가 많다. 그러므로, 사용자가 오류 장소를 쉽게 찾아 낼 수 있는 프로그램 개발 환경이 필요하다.

기존의 접근 방법은 대부분 타입유추 정보를 설명하는데 중점을 두고 있다.[7, 3] 이러한 방법은 정제된 정보와 편리한 사용자 환경이 제공되지 않으면 사용자는 불필요한 많은 정보를 해석해야 하고, 컴파일러는 그러한 정보를 유지하기 위해 쓸데없는 계산을 많이 하게 된다.

본 연구에서는 두 가지 방법을 제시하고 있다. 하나는, 광범위한 타입오류 정보생성이 기존의 상향성(bottom-up) 타입유추 알고리즘에 기인한다고 보고, 새로운 하향성(top-down) 알고리즘을 소개한다. 또 하나는, 각각의 구성요소의 타입을 모른체 프로그래밍을 하다가 타입오류를 발생시키는 경우가 있어, 사용자가 쉽게 각각의 구성요소에 대한 타입 정보를 알아볼 수 있는 도구를

*이 연구는 정보통신부의 "Standard ML(SML)의 프로그래밍 환경 개발"의 지원을 받았음을 밝힙니다.

개발하여, 이를 이용하면 타입오류가 없는 프로그램 작성에 도움을 줄 뿐만 아니라 오류발생시에 타입정보를 보고 쉽게 오류를 수정할 수 있도록 한다.

2 좁은 범위의 오류 정보를 주는 타입유추 알고리즘

2.1 동기

오류 장소를 지적하는 메시지가 광범위해지는 이유가 기존의 타입유추 알고리즘, 즉, $\mathcal{W}$ 알고리즘[4]이 상향성 알고리즘이기 때문이라 생각한다. 상향성 알고리즘은 문법트리의 아래쪽부터 타입정보가 구체화(instantiation)되어 올라오다가 동일화(unification) 과정이 있는 함수 적용(application)에서 오류가 발생한다. 다음 예는 $\mathcal{W}$ 알고리즘을 기반으로 타입유추를 하는 SML/NJ[6]로 작성된 프로그램과 그에 따라 생성된 오류 정보이다. 연산자와 피연산자의 함수 적용 부분, 즉, 전체 프로그램에서 오류가 발생했다. 사용자가 피연산자가 틀렸다고 생각하면 사용자의 머리속에서 타입유추를 하여 왜 피연산자가 `string`이 되었는지 따라 내려가 봐야 하고 결국 `"a"`와 `"b"`가 틀렸음을 알게 된다. 만약 피연산자가 아주 크다면 사용자가 타입오류의 원인을 알기위해 타입을 유추하는 것이 매우 어려운 일이 될 것이다.

```
(fn x => x+1) ((fn y => if y then "a" else "b") false)

Error: operator and operand don't agree (tycon mismatch)
  operator domain: int
  operand:         string
  in expression:
    ((fn x => <exp> <exp>)) (((fn <rule>)) false)
```

경험적으로 대부분의 오류 수정은 변수나 상수같은 문법트리의 잎 노드(leaf-node)에서 이루어 진다. 위의 예제에서 사용자가 결국은 오류 정보로부터 타입유추를 해서 오류가 잎 노드인 `"a"`와 `"b"`에 있음을 알게 되고 그를 수정하게 된다. 그러면 사용자는 왜 컴파일러가 잎 노드에서 직접 오류 정보를 주지 않을까 불평을 할 수 있다. 앞서 말했듯이 상향성 알고리즘은 잎 노드에서 오류가 발생하지 않는다. 만약 하향성 타입유추 알고리즘이 있다면 오류가 잎 노드에서 발생할 것이다.

2.2 $\mathcal{M}$ 알고리즘

하향성 타입유추 알고리즘은 각각의 표현식(expression)에 타입을 강요하는 개념을 생각하면 자연스럽게 만들어 질 수 있다. 예를 들어, $e_1\ e_2$를 어떤 타입 ρ가 되도록 강요한다면, e_1은 타입 $\beta \to \rho$가 되도록 강요하면 된다. 여기서 β는 사용되지 않은 새로운 타입변수이고 이러한 타입변수들은 나중에 특정 타입으로 구체화될 수 있다. 이러한 구체화는 기존의 알고리즘과 마찬가지로 동일화 과정에 의해 이루어지며, 동일화는 잎 노드, 즉, 변수, 상수와 함수선언에서 이루어진다.

그림 1은 Damas와 Milner가 사용한 언어[4]에 대한 강요 개념을 이용한 타입유추 알고리즘이다. 알고리즘의 결과값이 타입치환함수(substitution)인 이유는 결과 타입이 강요한 타입의 용례(instance)이어야 하므로 결과 타입을 결과값으로 하지 않아도 타입치환함수과 강요한 타입으로 결과값을 만들어 낼 수 있기 때문이다.

다음은 앞의 예제 프로그램을 $\mathcal{M}$ 알고리즘으로 타입유추했을 때의 결과이다. 잎 노드에서 오류 정보를 생성하고 있음을 알 수 있다.

$$\mathcal{M} : TypEnv \times Expr \times Type \rightarrow Subst$$

$$
\begin{array}{l}
\mathcal{M}(\Gamma, x, \rho) = \\
\qquad \mathcal{U}(\rho, [\beta_i/\alpha_i]\tau), \\
\qquad\qquad \text{where } \Gamma(x) = \forall\vec{\alpha}.\tau,\ \text{new } \vec{\beta} \\
\mathcal{M}(\Gamma, \lambda x.e, \rho) = \\
\quad \text{let} \quad S_1 = \mathcal{U}(\rho, \beta_1 \rightarrow \beta_2),\ \text{new } \beta_1, \beta_2 \\
\qquad\quad\ S_2 = \mathcal{M}(S_1\Gamma + x : S_1\beta_1, e, S_1\beta_2) \\
\quad \text{in} \quad S_2 S_1 \\
\mathcal{M}(\Gamma, e_1\ e_2, \rho) = \\
\quad \text{let} \quad S_1 = \mathcal{M}(\Gamma, e_1, \beta \rightarrow \rho),\ \text{new } \beta \\
\qquad\quad\ S_2 = \mathcal{M}(S_1\Gamma, e_2, S_1\beta) \\
\quad \text{in} \quad S_2 S_1 \\
\mathcal{M}(\Gamma, \texttt{let } x\texttt{=}e_1 \texttt{ in } e_2, \rho) = \\
\quad \text{let} \quad S_1 = \mathcal{M}(\Gamma, e_1, \beta),\ \text{new } \beta \\
\qquad\quad\ S_2 = \mathcal{M}(S_1\Gamma + x : Clos_{S_1\Gamma}(S_1\beta), e_2, S_1\rho) \\
\quad \text{in} \quad S_2 S_1
\end{array}
$$

그림 1: $\mathcal{M}$ 알고리즘

```
Error: expected type mismatch in "a".
  expected: int
  inferred: string
```

2.3 안전성과 완전성

새로 제시된 알고리즘으로 기존의 알고리즘을 대체하기 위해서는 제시된 알고리즘이 주어진 언어의 안전성(soundness)과 완전성(completeness)이 증명되어야 한다. 다음의 두 정리가 각각 안전성과 완전성을 말하고 있고, 증명되었다.[1]

정리 1 ($\mathcal{M}$ 알고리즘의 안전성) *표현식 e, 타입유추환경 Γ에 대하여 $\mathcal{M}(\Gamma, e, \rho) = S$인 타입 ρ가 존재하면, $S\Gamma \vdash e : S\rho$가 존재한다.*

정리 2 ($\mathcal{M}$ 알고리즘의 완전성) *표현식 e, 타입유추환경 Γ에 대하여 $P\Gamma \vdash e : P\rho$를 만족하는 타입 ρ와 타입치환함수 P가 존재하면, $\mathcal{M}(\Gamma, e, \rho)$가 S를 결과값으로 성공하며, 도중에 생성된 타입변수들 밖에서 $P = RS$를 만족하는 타입치환함수 R이 존재한다.*

2.4 좁은 오류 범위

$\mathcal{M}$ 알고리즘은 오류가 있는 입력 프로그램에 대해 잎 노드에서 오류가 발생하도록 설계되었다. 즉, $\mathcal{W}$ 알고리즘이 오류를 발생하는 범위보다 좁은곳에서 오류가 발생하도록 설계되었다. 그러나, 항상 그렇지는 않다. 하나 이상의 오류를 가진 프로그램이 그 반례이다. 하나 이상의 오류를 가졌다는 말은 프로그램을 $\mathcal{W}$로 유추하다가 발생되는 오류장소를 어떻게 수정하던 간에

수정된 프로그램을 $\mathcal{W}$로 유추하면 또 오류가 발생하는 것을 말한다. 그러나, 하나의 오류를 가진 프로그램에서는 $\mathcal{M}$의 오류 발생 장소가 $\mathcal{W}$의 오류 발생 장소의 하위표현식(sub-expression)이다.

하나의 오류를 가졌다라는 말을 수식적으로 설명하기 위해 오류가 있어도 중단하지 않는 (error tolerant) $\mathcal{W}$의 변형 알고리즘을 서술할 필요가 있다. 이 알고리즘을 $\mathcal{W}'$ 알고리즘[1]이라 하겠다. 변형은 간단하다. 동일화 오류가 발생하면 단지 빈 정보, 즉 타입치환함수으로 id를 , 결과타입으로 새 타입변수를 결과값으로 주도록 하면 된다. 여기에 오류 발생 장소 정보를 누적하여 결과값으로 주도록 하면 된다.

다음 정의는 앞에 이야기한 표현식이 하나의 오류만을 갖는 다는 것을 수식적으로 정의한 것이고, 다음 정리는 $\mathcal{W}$ 알고리즘과 $\mathcal{M}$ 알고리즘의 오류 발생장소가 하나의 오류만을 가질때 $\mathcal{M}$의 오류 발생장소가 $\mathcal{W}$의 오류 발생장소의 하위표현식이라는 것을 말하고 있고 증명되었다.[1]

정의 1 *표현식 p, $\mathcal{W}'(\emptyset, p)$의 결과값 (S, τ, F)에 대해서, $|F| = 1$이면, p는 $\mathcal{W}'$의 관점에서 오직 하나의 오류만을 갖고 있다라고 한다.*

정리 3 *표현식 p, 새 타입 변수 β에 대하여 $\mathcal{W}(\emptyset, p)$가 e에서 실패하고, $\mathcal{M}(\emptyset, p, \beta)$가 e'에서 실패하고, p가 $\mathcal{W}'$의 관점에서 오직 하나만의 오류를 가졌다면, e'는 e의 하위표현식이다.*

어떤 경우에는 $\mathcal{M}$이 불공정해 보일 소지가 있다. 정리 3에 의하면 하나의 오류만을 가진 프로그램에서 $\mathcal{M}$ 알고리즘의 오류 발생 장소는 $\mathcal{W}$ 알고리즘의 오류 발생 장소의 하위표현식이라 하였다. 그럼, $\mathcal{W}$ 알고리즘의 오류 발생 장소 아래에 속하면서 $\mathcal{M}$ 알고리즘의 오류 발생 장소 아래에는 속하지 않는 표현식에는 잘못이 없을까? 예를 들어, $e_1\ e_2$ 프로그램에 대해 e_1이 `int`→`int` 타입을 가져야 하고 e_2가 `int` 타입을 가져야 할 때, 사용자가 실수하여 e_1이 `string`→`int` 타입을 가지도록 했다면 $\mathcal{M}$은 e_2에게 `string`을 강요하여 e_2내에서 오류가 발생하게 된다.

$\mathcal{M}$ 알고리즘을 불공정한 오류 메시지를 내는 알고리즘이라고 한다면 대부분의 오류 메시지를 내는 알고리즘들도 불공정하다고 할 수 있다. 예를 들어, C로 쓰여진 프로그램이 변수 `x1`이 `int x1;`에 의해 선언되고 나중에 뒤에서 쓰였다고 생각해 보자. 그런데 사용자가 타이핑 실수로 `int x2;`라고 쓰면 C 컴파일러는 선언문에서 오류를 발생하지 않고 `x1`이 처음 사용된 문장에서 선언되지 않은 변수를 사용했다고 오류 메시지를 내게 된다.

2.5 오류 허용 $\mathcal{M}$ 알고리즘

오류 허용(error tolerant) 타입유추 알고리즘은 구현에서 매우 필요하다. 다음에 제시될 타입 정보를 보여주는 도구에 적용할 때, 만약 타입유추 알고리즘이 오류가 있을때 정지한다면 전체 타입 정보를 보여 줄 수 없고 또한 오류 지점도 한 장소밖에 보여 줄 수 없다. 단순히 ML 인터프리터(interpreter)에서 작업을 하고 있는 상황을 생각해도 만약 프로그램이 많은 작은 오류를 가지고 있다고 하면 사용자는 오류의 개수만큼 타입유추와 수정작업을 반복해야 한다.

$\mathcal{M}$ 알고리즘을 변형시켜 오류 허용 알고리즘으로 변환하는 것은 단순하다. 동일화 과정 중 실패할 때 빈 정보, 즉 결과 타입치환함수으로 id를 주면 결과 타입이 강요된 타입이 된다. 이렇게 만들어진 알고리즘을 $\mathcal{M}'$ 알고리즘[1]이라 하겠다.

$\mathcal{M}'$ 알고리즘은 $\mathcal{W}'$ 알고리즘보다 적은 정보를 무시하게 된다. 앞의 예제 프로그램을 $\mathcal{W}'$ 알고리즘에 돌려보면 전체 프로그램에 오류가 있으므로 전체 프로그램을 무시한다. 그러나, $\mathcal{M}'$ 알고리즘은 다음과 같은 오류 메시지를 내면서 `"a"`와 `"b"`를 `int`로 간주하고, 결과 타입으로 `int`를 준다.

```
Error: expected type mismatch in "a".
  expected: int
  inferred: string
Error: expected type mismatch in "b".
  expected: int
  inferred: string
```

3 타입유추 정보를 보여주는 도구

Standard ML(SML)[6, 5]에서는 함수나 변수의 타입을 프로그래머가 직접 기술할 필요가 없이 컴파일러가 타입유추를 통해 알아 낸다. 그러나, 정의한 함수나 변수를 사용하기 위해서는 반드시 그 타입을 알고 있어야 하고, 또 타입 오류(type error)가 발생시 프로그래머가 직접 타입유추를 해서 타입 오류의 진원지(source of type error)를 찾아야 한다.

본 연구에서는 프로그래머가 프로그램을 작성하거나 작성된 프로그램을 분석할 때 도움을 주는 도구를 개발하였다. 프로그래머는 emacs 편집기를 통해 프로그램내의 변수나 함수의 타입을 볼 수 있다. 타입 오류가 발생한 프로그램의 경우 프로그래머가 유추된 타입을 보고 타입 오류의 원인을 직접 쉽게 찾을 수 있다.

본 연구의 명확한 설명을 위해 몇가지 용어를 앞서 정의 한다.

영역(region) SML 프로그램내의 언어구조(language construct)에 대한 시작과 끝의 위치

기본문(atomic construct) 더 이상 분리될 수 없는 언어구조

복합문(composite construct) 둘 이상의 기본문혹은 복합문으로 이루어진 언어구조

타입노드(type node) 타입을 가지는 언어구조

영역을 가진 타입(type with region) 영역, 기본문인지 복합문인지를 나타내는 표시, 그리고 타입으로 이루어진 튜플(tuple)

3.1 도구의 전체 구조

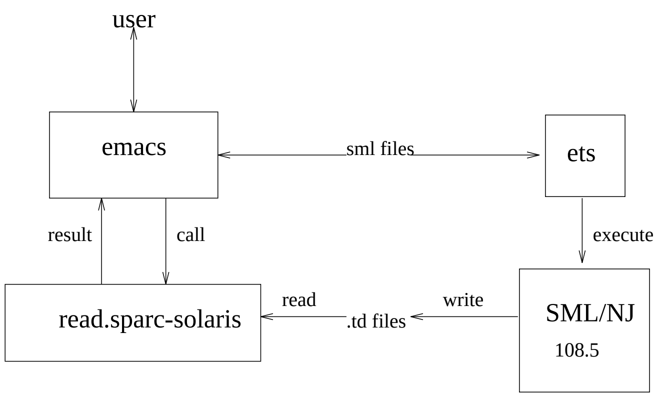

그림 2: 전체 도구 구조

그림 2는 전체 도구의 구조이다. 사용자가 분석을 원하는 SML 프로그램들을 ets의 입력으로 준다. ets는 변형된 SML/NJ 컴파일러에서 그 프로그램들을 실행시키고, 그 결과 각 프로그램의

이름에 확장자로 .td가 붙은 파일들을 출력한다. 그 후 사용자는 그 프로그램들을 emacs 편집기로 읽은 뒤, 각 프로그램내의 타입노드를 지정한 후 원하는 emacs 함수를 호출한다. 그 함수는 read.sparc-solaris를 실행시킨 결과를 emacs 편집기에 보여 준다. read.sparc-solaris는 그 프로그램의 이름 및 그 영역 정보를 받아서 그 대응하는 .td 파일로부터 SML/NJ 컴파일러에서 유추된 타입을 출력 한다.

사용자는 emacs 편집기에서 프로그램내의 모든 타입노드에 대해 다음의 2가지 emacs 함수를 호출 할 수 있다.

- find-tight-type 함수
  편집기내의 커서위치를 포함하는 기본문의 타입을 보여줌
- find-max-type 함수
  편집기내의 지정된 영역내의 가장 큰 타입노드의 타입을 보여줌

3.2 변형된 SML/NJ 108.5의 구조

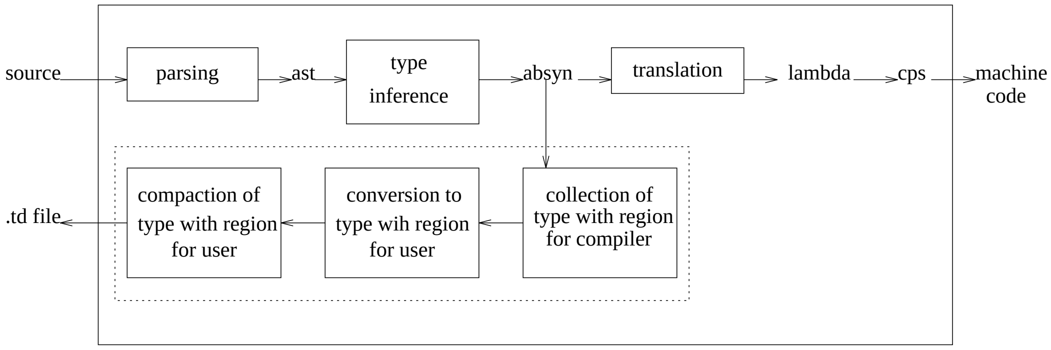

그림 3: 변형된 SML/NJ 108.5의 구조

그림 3은 변형된 SML/NJ 108.5 컴파일러의 구조이다. 본래의 이 컴파일러는 SML 프로그램을 입력으로 받으면 컴파일러내의 프로그램 표현형식인 ast로 변형한다. ast는 타입유추를 통해 타입 정보가 붙은 absyn으로 바뀐다. absyn은 함수 언어의 기본 언어인 lambda 언어로 바뀐 뒤, 폰 노이만 기계에 가깝게 설계된 언어인 cps로 바뀌어진 후 기계어로 번역 된다.

본 연구에서는 이 컴파일러의 absyn에서 필요한 모든 정보를 얻도록 하였다. 구문 분석기(parser)가 프로그램내의 모든 타입노드의 영역을 알 수 있도록 자료 구조를 첨가 하고, 타입 유추시에 그 자료 구조를 계속 유지하도록 변경하였다. absyn에서 필요한 모든 정보를 얻은 뒤에서 첨가된 모든 자료 구조를 제거함으로써 컴파일러의 뒷단계(back-end)를 수정할 필요가 없게 하였다.

absyn에서 각 타입노드에 대한 영역을 가진 타입을 모은 뒤 그 모은 정보를 효율적으로 .td 파일에 쓰기위해 다음의 방법을 사용하였다.

- 타입 표현시 문자열 및 타입의 공유
- 영역을 가진 타입의 원소들을 비트단위로 표현

그림 4는 타입 표현시 문자열 및 타입이 어떻게 공유되는 지를 보여준다. S는 문자열을, T는 튜플 타입(tuple type), F는 함수 타입(function type), 그리고, R은 레코드 타입(record type)을

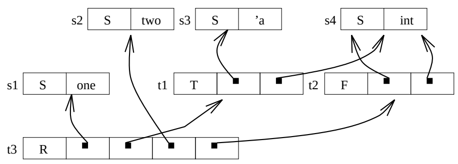

그림 4: 타입 표현의 예

의미하는데 S는 타입을 표현하는 최소 단위이다. t1, t2, t3는 각각 `'a→int`, `int * int`, {`one : 'a → int, two : int * int`}를 나타낸다. 그림 4에서 모든 문자열 및 타입은 한번만 쓰였고, 두번 이상쓰일때는 포인터로 공유하였다. 실제로 본 도구는 프로그램내의 모든 타입노드에 대한 서로 다른 문자열 및 타입은 오직 하나씩만 존재하도록 한다[2].

또, 프로그램내의 가장 큰 영역을 나타낼 수 있는 최소 비트수로 모든 영역들을 표현하며, 그림 4의 t1, t2, t3에서 포인터는 s1,s2드을 저장하는 문자열 테이블과 t1,t2등을 저장하는 타입 테이블의 크기가 결정된 후 log_2를 적용한 결과의 비트수로 표현한다. 기본문인지 복합문인지는 1비트로 표현한다.

영역을 가진 타입들로부터 효과적으로 원하는 곳의 타입을 찾기 위해 영역에 대한 부분 순서를 정의한다. 두 영역r_1, r_2에 대해 각각을 (x_1, x_2), (y_1, y_2)라고 했을 경우 그들간에 부분 순서는 다음과 같이 정의 한다.

- $x_2 < y_1$ 이면, $r_1 < r_2$
- $x_1 \geq y_1$ 이고 $x_2 \leq y_2$ 이면, $r_1 < r_2$
- 위 두 조건에 속하지 않으면 정의되지 않음

프로그램내의 모든 타입노드에 대한 영역을 가진 타입은 영역에 대해 서로 순서를 갖게 되므로[2], 오름차순으로 정렬을 한 후 .td 파일에 쓴다.

3.3 영역찾기

다음은 emacs 편집기에서 find-tight-type 함수를 호출했을 때, read.sparc-solaris가 그 편집기로부터 받은 프로그램이름의 .td 파일로부터 현재 커서를 포함하는 기본인 타입노드의 타입을 출력하는 알고리즘을 기술한 것이다.

- 찾는 영역을 가진 타입을 t라 하고, no type으로 초기화 시킨다.
- 이진 탐색으로 주어진 지점 p을 포함하는 영역을 가진 타입 t'을 찾는다.
- 그러한 t'을 찾지 못하면, 입력된 점 p를 포함하며 기본문인 영역을 가진 타입은 없음을 출력하고 끝난다.
- t에 t'을 할당한다.
- 파일의 위쪽에 있는 영역을 가진 타입 t',((x_1, x_2),tag,type)에 대해서 $x_2 \geq$ p 인동안 이 단계를 반복하면서, $x_1 \leq$ p 이고 tag가 기본문이면 t에 t'을 할당한다.

find-max-type 함수를 호출했을 때 read.sparc-solaris가 수행하는 알고리즘은 다음과 같다.

- 찾는 영역을 가진 타입을 t라 하고, no type으로 초기화 시킨다.
- 이진 탐색으로 주어진 영역 r, (x_1, x_2)을 포함하는 영역을 가진 타입 t'을 찾는다.
- 그러한 t'을 찾지 못하면, r내에는 영역을 가진 타입은 없음을 출력하고 끝난다.
- t에 t'을 할당한다.
- 파일의 아래쪽에 있는 영역을 가진 타입 t', $((y_1, y_2)$,tag,type)에 대해서 $x_1 \leq y_2$이고 $x_2 \geq y_1$ 인동안 이 단계를 반복하면서, t에 t'을 할당한다.

3.4 실험 결과

본 연구에서 개발한 도구를 SML/NJ 108.5 컴파일러 원시 코드의 200개가 넘는 파일들에 적용하였다. 실험은 256 MB의 메모리를 갖는 Ultra Sparc 1에서 하였다. .td 파일의 크기에 대해 살펴보면, 평균 31 바이트/라인이었고, 평균 6 바이트/타입노드였다. 또, 프로그래머가 타입노드의 타입을 요구시 실시간으로 emacs 편집기에 보여 주었다.

4 결론

타입 오류 수정을 용이하게 하기 위해서 두가지 방법으로 접근하여 보았다.

첫번째 방법은 넓은 범위의 오류 정보 생성을 막기 위해서 하향성 타입유추 알고리즘을 제시하고, 이 알고리즘의 안전성과 완전성, 그리고, 좁은 범위의 오류 정보를 생성함을 증명하였다. 그러나, 사용자가 이 알고리즘이 생성하는 오류 정보를 보고 오류 수정이 용이한지는 알 수 없다. 이를 통계적으로 증명하기 위해, SML/NJ 컴파일러에 구현하여 사용하도록 할 것이다.

두번째 방법은 타입 정보를 사용자가 직접 쉽게 알아 볼 수 있는 사용자 도구를 효율적으로 구현하였다. SML 프로그램에 타입 오류가 발생했을 경우, 프로그래머는 실제 컴파일러가 유추한 타입을 봄으로써 그 오류를 직접 용이하게 수정할 수 있도록 하였다.

참고 서적

[1] 이욱세. 정확한 타입오류 정보를 위한 타입유추 알고리즘. 석사논문, 한국과학기술원, 1997 예정.

[2] 편기현. Sml 프로그램에 대해 최소타입을 보여주는 도구. 석사논문, 한국과학기술원, 1997 예정.

[3] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages and Systems*, volume 2, pages 17–30, March-December 1993.

[4] Luis Damas and Robin Milner. Principal type-scheme for functional programs. In *Proceedings of the Annual ACM Symposium on Principles of Program ming Languages*, 1982.

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

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

[7] Mitchell Wand. Finding the source of type errors. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, 1986.