$\mathcal{M}$: 타입 오류를 일찍 감지하는 타입 유추 알고리즘*

이욱세 이광근
한국과학기술원 전산학과

$\mathcal{M}$: a Let-polymorphic Type Inference Algorithm that Stops Early

Ooksesh Lee and Kwangkeun Yi
Dept. of Computer Science, KAIST

요 약

SML[9]과 같은 타입 시스템을 갖춘 언어로 프로그램을 작성하다 보면 컴파일러가 넓은 범위의 타입 오류 정보를 주는 경우가 많다. 넓은 범위의 오류 정보는 오류 수정에 전혀 도움을 주지 않을 수도 있다. 이러한 오류 정보는, 표준 타입 유추 알고리즘인, $\mathcal{W}$ 알고리즘[8, 4]이 너무 늦게 오류를 감지하기 때문이라 판단된다.

본 논문에서는 일찍 오류를 감지하는 새로운 타입유추 알고리즘 $\mathcal{M}$을 소개한다. $\mathcal{M}$ 알고리즘은 각각의 표현식(expression)이 가질 수 있는 타입 제한 조건을 만들어, 조건을 벗어나는 타입이 유추될 경우 멈춤으로써, 일찍 오류를 감지할 수 있다. $\mathcal{M}$ 알고리즘은 타입 유추 시스템을 충실히 구현하고 있고, $\mathcal{W}$ 알고리즘보다 일찍 오류를 감지한다.

1. 서론

$\mathcal{W}$ 알고리즘[8, 4]은 표준 타입유추(type inference) 알고리즘이다. Standard ML[9]과 같은 언어의 타입유추 과정에서, $\mathcal{W}$ 알고리즘을 기반으로 하는 유추 알고리즘을 사용하고 있다.

$\mathcal{W}$ 알고리즘은 너무 늦게 타입 오류를 감지하는 경향이 있다. 함수 적용문(application)에서 함수와 인자가 서로 충돌되는 타입이 유추되었을 때 실패한다. 즉, 함수와 인자가 따로 유추가 되고 나서야 오류가 감지되는 것이다. 그러므로, 함수 적용문 전체에 대한 오류 정보를 생성하게 된다. 그림 1의 예제 프로그램을 보면, 전체 프로그램(1–8 줄)을 지적하는 오류 정보가 주어진다. 이러한 넓은 범위의 오류 정보는 때로 프로그램 작성자가 오류를 수정하는데 도움이 되지 못하는 경우가 많다.

본 논문에서는 일찍 오류를 감지하는 타입 유추 알고리즘을 소개한다. 예를 들어 그림 1 프로그램의 경우, 4번째 줄 "`if x then false`"에서 `false`가 `real` 타입이어야 한다고 지적하는 오류 정보가 생성된다.

새로운 알고리즘의 기본 생각은, 각각의 표현식이 가져야 할 타입의 조건을 만들어, 이를 만족시키도록 타입 유추를 하는 것이다. 처음 유추를 시작할 때는 제한 조건이 없으므로, 새 타입 변수 β를 조건으로 유추한다. 함수 적용문 "$e_1\,e_2$"가 `int` 조건으로 유추되어야 하는 경우, e_1은 $\alpha \to$ `int` 조건내에 유추되어야 하고, e_2는 α가 e_1을 유추하여 변경된 어떤 타입 조건으로 유추되어야 한다. 변수와 상수는 각각이 가지고 있는 타입이 타입 조건을 만족하여야 유추가 성공한다. 이런 식으로 프로그램 각 하위 표현식(sub-expression)들에 대해 각각의 타입 조건을 만들어 가며 그 조건을 강요하는 것이다.

```
(fn x =>
  x+1.0)
((fn x =>
    if x then false
        else true)
 (true
   andalso
  false))
>a.sml:1.1-8.9 Error: operator and operand don't agree ...
>  operator domain: real
>  operand:         bool
>  in expression: ...
```

그림 1: SML/NJ 109.8의 타입 오류 정보(>로 표시된 부분)

이러한 타입 조건은, 바깥쪽 표현식에서 결정된 타입 정보가 안쪽 표현식의 타입 유추시 사용되도록 하여, 오류를 일찍 감지토록 하는 역할을 한다. (정확하게는, 바깥쪽뿐만 아니라 유추 알고리즘이 방문하는 순서에 따라, 먼저 유추된 타입 정보가 뒤에 유추할 때 사용된다.) $\mathcal{W}$의 경우, 이런 정보의 사용없이 각각의 하위 표현식을 독립적으로 유추한 후, 그 결과로부터 오류를 알아 내므로, 늦게 감지하게 된다.

2. $\mathcal{M}$ 알고리즘

2.1 개요

$\mathcal{M}$ 알고리즘은 타입 조건을 받아, 주어진 표현식이 조건을 만족하지 않으면 실패한다. 예를 들어, 다음과 같은 표현식을 생각해

*본 연구는 정보통신부, 한국과학재단(과제번호 95-0100-54-3), 삼성전자의 지원을 받았음을 밝힙니다.

보자.

$$\underbrace{\texttt{(fn x=> x+1)}}_{e_1}\ \underbrace{\texttt{(}\overbrace{\texttt{(fn y=> if y then "a" else "b")}}^{e_3}\ \overbrace{\texttt{false}}^{e_4}\texttt{)}}_{e_2}$$

e_1은 함수 타입을 가져야 하므로, $\alpha \to \beta$라는 조건으로 유추한다. e_1에 대한 타입 유추는 성공하여 타입 치환함수(substitution) $\{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}\}$을 결과로 준다. 이것은, 즉 함수의 인자 타입이 `int`이어야 함을 뜻한다. 그러므로, e_2를 `int`라는 조건으로 유추한다. 순차적으로 e_3를 $\gamma \to \texttt{int}$ 조건으로 유추하고, `if`-문을 `int` 조건으로 유추하고, 결국, `"a"`를 `int` 조건으로 유추한다. `"a"`는 `string` 타입을 가지므로 주어진 조건을 만족할 수 없다. 그러므로, $\mathcal{M}$ 알고리즘은 `"a"`에서 실패한다.

위의 예제를 $\mathcal{W}$ 알고리즘으로 유추하면 전체 프로그램에서 오류가 발생한다. e_1의 타입이 $\texttt{int}\to\texttt{int}$ 타입으로 유추되고, e_2의 타입이 `string` 타입으로 유추된다. 그리하여, e_2를 e_1에 적용할 수 없게 된다. 그러므로, 전체 프로그램인 함수 적용문에서 실패한다.

추상 구문(abstract syntax)

$$\begin{array}{llcll}
\textit{Expr} & e & := & () & \text{상수} \\
 & & | & x & \text{변수} \\
 & & | & \lambda x.e & \text{함수 정의문} \\
 & & | & e\ e & \text{함수 적용문} \\
 & & | & \texttt{let}\ x\texttt{=}e\ \texttt{in}\ e & \texttt{let}\text{-문} \\
 & & | & \texttt{fix}\ f\ \lambda x.e & \texttt{fix}\text{-문} \\
\textit{Type} & \tau & := & \iota & \text{상수 타입} \\
 & & | & \alpha & \text{타입 변수} \\
 & & | & \tau \to \tau & \text{함수 타입} \\
\textit{TypeScheme} & \sigma & := & \tau \mid \forall\vec{\alpha}.\sigma & \\
\textit{TypeEnv} & \Gamma & \in & \textit{Var} \xrightarrow{\text{fin}} \textit{TypeScheme} & \text{타입 환경}
\end{array}$$

$$\begin{array}{cc}
\text{(CON)} & \Gamma \vdash () : \iota \\[6pt]
\text{(VAR)} & \dfrac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau} \\[10pt]
\text{(FN)} & \dfrac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \\[10pt]
\text{(APP)} & \dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_2} \\[10pt]
\text{(LET)} & \dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \textit{Clos}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : \tau_2} \\[10pt]
\text{(FIX)} & \dfrac{\Gamma + f : \tau \vdash \lambda x.e : \tau}{\Gamma \vdash \texttt{fix}\ f\ \lambda x.e : \tau}
\end{array}$$

$\mathcal{W}$: *TypEnv* $\times$ *Expr* $\to$ *Subst* $\times$ *Type*

$$\begin{array}{lll}
\mathcal{W}(\Gamma, ()) = & (id, \iota) & \\
\mathcal{W}(\Gamma, x) = & (id, [\beta_i/\alpha_i]\tau) \text{ where } \Gamma(x) = \forall\vec{\alpha}.\tau,\ \text{new } \vec{\beta} & \\
\mathcal{W}(\Gamma, \lambda x.e) = & \text{let} & (S_1, \tau_1) = \mathcal{W}(\Gamma + x : \beta, e),\ \text{new } \beta \\
 & \text{in} & (S_1, S_1\beta \to \tau_1) \\
\mathcal{W}(\Gamma, e_1\ e_2) = & \text{let} & (S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \\
 & & (S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2) \\
 & & S_3 = \mathcal{U}(S_2\tau_1, \tau_2 \to \beta),\ \text{new } \beta \\
 & \text{in} & (S_3S_2S_1, S_3\beta) \\
\multicolumn{3}{l}{\mathcal{W}(\Gamma, \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2) =} \\
 & \text{let} & (S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \\
 & & (S_2, \tau_2) = \mathcal{W}(S_1\Gamma + x : \textit{Clos}_{S_1\Gamma}(\tau_1), e_2) \\
 & \text{in} & (S_2S_1, \tau_2) \\
\multicolumn{3}{l}{\mathcal{W}(\Gamma, \texttt{fix}\ f\ \lambda x.e) =} \\
 & \text{let} & (S_1, \tau_1) = \mathcal{W}(\Gamma + f : \beta, \lambda x.e),\ \text{new } \beta \\
 & & S_2 = \mathcal{U}(S_1\beta, \tau_1) \\
 & \text{in} & (S_2S_1, S_2\tau_1)
\end{array}$$

그림 2: 기반 언어, 타입유추 규칙, $\mathcal{W}$ 알고리즘

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

$$\begin{array}{llr}
\multicolumn{2}{l}{\mathcal{M}(\Gamma, (), \rho) = \mathcal{U}(\rho, \iota)} & (\mathcal{M}.1) \\
\multicolumn{2}{l}{\mathcal{M}(\Gamma, x, \rho) = \mathcal{U}(\rho, [\beta_i/\alpha_i]\tau) \text{ where } \Gamma(x) = \forall\vec{\alpha}.\tau,\ \text{new } \vec{\beta}} & (\mathcal{M}.2) \\
\multicolumn{2}{l}{\mathcal{M}(\Gamma, \lambda x.e, \rho) =} & \\
\quad\text{let} & S_1 = \mathcal{U}(\rho, \beta_1 \to \beta_2),\ \text{new } \beta_1, \beta_2 & (\mathcal{M}.3) \\
 & S_2 = \mathcal{M}(S_1\Gamma + x : S_1\beta_1, e, S_1\beta_2) & (\mathcal{M}.4) \\
\quad\text{in} & S_2S_1 & \\
\multicolumn{2}{l}{\mathcal{M}(\Gamma, e_1\ e_2, \rho) =} & \\
\quad\text{let} & S_1 = \mathcal{M}(\Gamma, e_1, \beta \to \rho),\ \text{new } \beta & (\mathcal{M}.5) \\
 & S_2 = \mathcal{M}(S_1\Gamma, e_2, S_1\beta) & (\mathcal{M}.6) \\
\quad\text{in} & S_2S_1 & \\
\multicolumn{2}{l}{\mathcal{M}(\Gamma, \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2, \rho) =} & \\
\quad\text{let} & S_1 = \mathcal{M}(\Gamma, e_1, \beta),\ \text{new } \beta & (\mathcal{M}.7) \\
 & S_2 = \mathcal{M}(S_1\Gamma + x : \textit{Clos}_{S_1\Gamma}(S_1\beta), e_2, S_1\rho) & (\mathcal{M}.8) \\
\quad\text{in} & S_2S_1 & \\
\multicolumn{2}{l}{\mathcal{M}(\Gamma, \texttt{fix}\ f\ \lambda x.e, \rho) = \mathcal{M}(\Gamma + f : \rho, \lambda x.e, \rho)} & (\mathcal{M}.9)
\end{array}$$

그림 3: $\mathcal{M}$ 알고리즘

2.2 알고리즘

그림 3이 $\mathcal{M}$ 알고리즘을 기술한 것이다. 기반 언어와 타입 시스템, 그리고, 기반 언어에 대한 $\mathcal{W}$ 알고리즘은 그림 2에 있다. 사용된 기호들은 다음과 같다. "$\forall\vec{\alpha}.\tau' \succ \tau$"는, $S\tau' = \tau$와 $dom(S) \subseteq \vec{\alpha}$을 만족하는 타입 치환함수 S가 존재함을 뜻한다. 벡터로 표시된 $\vec{\alpha}$와 같은 기호는 $\alpha_1 \cdots \alpha_n$을 줄여 표기한 것이다. "$\Gamma + x : \sigma$"는, $\{y \mapsto \sigma' \mid x \neq y, y \mapsto \sigma' \in \Gamma\} \cup \{x \mapsto \sigma\}$를 뜻한다. "$\textit{Clos}_\Gamma(\tau)$"("Gen($\Gamma,\tau$)"[4])는 $\vec{\alpha} = ftv(\tau) - ftv(\Gamma)$라 할 때, $\forall\vec{\alpha}.\tau$을 뜻한다. "$[\beta_i/\alpha_i]\tau$"는 τ내의 모든 $\alpha_i \in \vec{\alpha}$를 이에 상응하는 $\beta_i \in \vec{\beta}$로 치환한 결과를 뜻한다.

$\mathcal{M}$ 알고리즘은 표현식, 타입환경(type environment), 타입 조건을 입력으로 받아, 타입 치환함수를 결과로 준다. 타입 조건은 단순히 타입으로 나타낼 수 있다. 표현식의 유추된 결과 타입은 타입 조건에 타입 치환함수을 적용하여 얻을 수 있다.

$\mathcal{M}$ 알고리즘은 동일화(unification) 알고리즘을 사용하는 곳이 함수 적용문이 아니라, 상수, 변수, 함수 정의문이다. 변수의 경우($\mathcal{M}$.2), 주어진 타입 조건 ρ와 변수 x의 타입 $\Gamma(x)$가 같은 타입이 될 수 있어야 한다: $\mathcal{U}(\rho, [\beta_i/\alpha_i]\tau)$. 함수 정의문의 경우 ($\mathcal{M}$.3), 우선 주어진 타입 조건 ρ가 함수 타입인지 검사하여야 한다. 동일화 과정인

$$\mathcal{U}(\rho, \beta_1 \to \beta_2), \quad \text{new } \beta_1, \beta_2 \qquad (\mathcal{M}.3)$$

이 검사하는 작업이다. 함수 적용문의 경우($\mathcal{M}$.5), 주어진 타입 조건 ρ가 e_1의 타입 조건의 치역이 되어야 한다.

$$\mathcal{M}(\Gamma, e_1, \beta \to \rho), \quad \text{new } \beta \qquad (\mathcal{M}.5)$$

그리고, β가 e_1의 유추를 통해서 보다 구체적인 타입이 된 $S_1\beta$가 e_2의 타입 조건이 되어야 한다. `let`-문의 경우($\mathcal{M}$.7), e_1은 아무 조건 없이 유추된다.

$$\mathcal{M}(\Gamma, e_1, \beta), \quad \text{new } \beta \qquad (\mathcal{M}.7)$$

왜냐하면 어떠한 조건도 만들어 낼 수 없기 때문이다. e_2의 타입 조건은 ρ가 e_1의 유추를 통해서 보다 구체적인 타입이 된 $S_1\rho$가 된다. `fix`-문의 경우($\mathcal{M}$.9), $\lambda x.e$의 타입 조건은 주어진 타입 조건 ρ와 같다.

2.3 안전성(soundness)과 완전성(completeness)

$\mathcal{M}$ 알고리즘은 주어진 타입유추 규칙(type inference rule)을 충실히 구현(sound and complete)하고 있다. 타입유추 알고리즘이 타입유추 규칙을 충실히 구현한다는 것은, 안전성(soundness)과 완전성(completeness)을 만족한다는 것을 의미한다. 안전성은 타입유추 알고리즘을 통해 얻어진 결과 타입이 타입유추 규칙을 만족하는 성질을 말한다. 완전성은 타입유추 규칙을 만족하는 타입을, 타입유추 알고리즘을 통해서도 얻을 수 있는 성질을 말한다.

$\mathcal{M}$ 알고리즘의 안전성은 다음과 같이 나타낼 수 있다.

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

$\mathcal{M}$ 알고리즘의 완전성은 다음과 같이 나타낼 수 있다.

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

즉, e의 유추된 타입 τ가 어떤 타입 조건 ρ를 만족하면 ($\exists P, \tau = P\rho$), $\mathcal{M}$ 알고리즘은 S를 결과로 성공하며, 결과 타입 $S\rho$는 τ를 대표하고 있음($\exists R, \tau = R(S\rho)$)을 뜻한다. "도중에 생성된 타입 변수들 밖에서 $P = RS$"는, V가 생성된 타입 변수들의 집합이고, $P|_V \stackrel{\text{def}}{=} \{a \mapsto b \in P \mid a \notin V\}$를 나타내면, $P|_V = (RS)|_V$임을 뜻한다.

$\mathcal{M}$ 알고리즘의 안전성과 완전성의 증명은 생략한다.[1]

2.4 오류의 빠른 감지

타입 유추 알고리즘의 실행과정을 호출 문자열(call string)로 나타낼 수 있다. 어떤 표현식 e, 타입환경 Γ에 대하여, $\mathcal{W}(\Gamma, e)$의 호출 문자열을 만드는 과정은 다음과 같다. 처음에 빈 문자열(ϵ)으로 시작하여, 알고리즘이 실행되는 과정에서 $\mathcal{W}$를 호출할 때의 인자를 순서대로 추가한다. 호출 뿐만 아니라 귀환할 때도 호출할 때의 인자를 순서대로 추가한다. 호출과 귀환을 구별하기 위해 호출할 때는 d를 귀환할 때는 u를 위에 붙인다. 알고리즘이 실행하거나 성공할 때까지 계속 추가하고, 더 이상 추가할 수 없을 때, 만들어진 문자열을 호출 문자열이라 한다. 예를 들어 표현식

$$\overbrace{(\underbrace{(\texttt{fn x=>x})}_{e_1}\ \texttt{2})}^{e},$$

에 대하여 $\mathcal{W}(\Gamma, e)$의 호출 문자열은

$$(\Gamma, e)^d (\Gamma, e_1)^d \cdots (\Gamma, e_1)^u (\Gamma, \texttt{2})^d (\Gamma, \texttt{2})^u (\Gamma, e)^u$$

가 된다. $\mathcal{M}(\Gamma, e, \beta)$의 호출 문자열도 비슷한 방법으로 정의한다.

정의 1 $[\![\mathcal{W}(\Gamma, e)]\!]$은 $\mathcal{W}(\Gamma, e)$의 호출 문자열을 나타낸다. $\mathcal{M}$에 대해서도 비슷하게 정의한다.

$[\![\mathcal{W}(\Gamma, e)]\!]$와 $[\![\mathcal{M}(\Gamma, e, \rho)]\!]$는 항상 유한함을 주목하라. 왜냐하면 각각의 하위 표현식을 $\mathcal{W}$와 $\mathcal{M}$은 한 번씩만 호출하기 때문이다.

$\mathcal{W}$의 동일화 과정이 실패하였을 때의 인자가 $(_, e)$인 경우, "$\mathcal{W}$ 알고리즘이 e에서 실패한다"라고 한다. $\mathcal{M}$ 알고리즘에 대해서도 비슷하게 정의한다.

정의 2 타입 오류를 가진 표현식 e, 타입환경 Γ, 새 타입 변수 β에 대하여, $[\![\mathcal{W}(\Gamma, e)]\!]$에서 $(\Gamma', e')^d(\Gamma', e')^u$ 형태의 문자열을 지울 수 없을 때까지 지워서 남은 문자열, $Fail[\![\mathcal{W}(\Gamma, e)]\!]$이 $(\Gamma', e')^d$로 끝나면, 즉,

$$Fail[\![\mathcal{W}(\Gamma, e)]\!] = \cdots (\Gamma', e')^d$$

이면, "$\mathcal{W}(\Gamma, e)$가 e'에서 실패한다"라고 한다. 마찬가지로

$$Fail[\![\mathcal{M}(\Gamma, e, \beta)]\!] = \cdots (\Gamma', e', \rho')^d$$

이면, "$\mathcal{M}(\Gamma, e, \beta)$이 e'에서 실패한다"라고 한다.

단순히 "$\mathcal{W}(\Gamma, e)$가 실패한다"는 $Fail[\![\mathcal{W}(\Gamma, e)]\!] = \epsilon$일 때 사용한다.

호출 문자열을 이용하여, $\mathcal{M}$ 알고리즘이 $\mathcal{W}$ 알고리즘보다 일찍 오류를 감지함을 다음과 같이 나타낼 수 있다.

정리 3 (빠른 오류 감지) 타입환경 Γ, 표현식 e, 새 타입 변수 β에 대하여,

$$|[\![\mathcal{M}(\Gamma, e, \beta)]\!]| \leq |[\![\mathcal{W}(\Gamma, e)]\!]|$$

여기서 $|s|$ 는 호출 문자열 s의 튜플의 수를 의미한다.

호출 문자열이 더 짧다는 것은 주어진 입력이 타입 오류를 가지는 경우, 일찍 실패한다는 것을 의미한다. 즉, 일찍 오류를 감지함을 의미한다.

예제 1 표현식 `((`λ`x.x^"a") true)`의 경우를 보자. $\mathcal{W}$ 알고리즘은 `((`λ`x.x^"a") true)`의 하위 표현식들을 유추하고 나서 실패한다. 이에 비해 $\mathcal{M}$ 알고리즘은 `true`에서 실패한다.

예제 2 표현식 `(1 (2 3))`의 경우를 보자. $\mathcal{W}$ 알고리즘은 `(2 3)`에서 실패한다. $\mathcal{M}$ 알고리즘은 일찍 `1`에서 실패한다. 함수 적용문에서 `1`이 유추되고 나서 `(2 3)`이 유추됨을 주목하라.

예제 3 표현식 `((`λ`x.x+1)` λ`y.(1 1))`의 경우를 보자. $\mathcal{W}$ 알고리즘은 `(1 1)`에서 실패한다. $\mathcal{M}$ 알고리즘은 `(`λ`y.(1 1))`에서 실패하는데, 이것이 호출되고 나서 `(1 1)`이 호출된다.

정리 3의 증명에서 보조정리 1-3이 사용된다. 정의 3은 수식을 간단하게 하기 위하여 사용된다.

정의 3 두 타입환경 Γ, Γ'에 대해 $dom(\Gamma) = dom(\Gamma')$이고, 모든 $dom(\Gamma)$ 안의 x에 대해 $\Gamma'(x) \succ \Gamma(x)$이면, Γ'가 Γ보다 일반적이라 하고, 역도 성립한다. 이런 경우 $\Gamma' \succ \Gamma$로 표기한다.

보조정리 1 [7] $\Gamma' \succ \Gamma$을 만족하는 두 타입환경 Γ, Γ'에 대해 $\Gamma \vdash e : \tau$가 존재하면 $\Gamma' \vdash e : \tau$도 존재한다.

보조정리 2 [1] 표현식 e, 타입환경 Γ, 새 타입 변수 β에 대하여, $[\![\mathcal{W}(\Gamma, e)]\!]$에 $(\Gamma^{\mathcal{W}}, e')^d$가 있고, $[\![\mathcal{M}(\Gamma, e, \beta)]\!]$에 $(\Gamma^{\mathcal{W}}, e', \rho)^d$가 있으면, $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$을 만족하는 타입 치환함수 R이 존재한다.

보조정리 3 표현식 e, 타입환경, 새 타입 변수 β에 대하여, $[\![\mathcal{W}(\Gamma, e)]\!]$에 $(\Gamma^{\mathcal{W}}, e')^d$가 있고, $[\![\mathcal{M}(\Gamma, e, \beta)]\!]$에 $(\Gamma^{\mathcal{M}}, e', \rho)^d$가 있을 때, $\mathcal{W}(\Gamma^{\mathcal{W}}, e')$가 실패하면, $\mathcal{M}(\Gamma^{\mathcal{M}}, e', \rho)$도 실패한다.

증명. $\mathcal{M}(\Gamma^{\mathcal{M}}, e', \rho)$가 어떤 타입 치환함수 S를 결과로 성공한다고 가정하자. 그러면, $\mathcal{M}$ 알고리즘의 안전성에 의해 다음 규칙이 성립한다.

$$S\Gamma^{\mathcal{M}} \vdash e' : S\rho$$

보조정리 2에 의해, $\Gamma^{\mathcal{M}} \succ R\Gamma^{\mathcal{W}}$을 만족하는 타입 치환함수 R이 존재한다. 즉, $S\Gamma^{\mathcal{M}} \succ SR\Gamma^{\mathcal{W}}$이다. 보조정리 1에 의해,

$$SR\Gamma^{\mathcal{W}} \vdash e' : S\rho$$

이 성립하므로, $\mathcal{W}$ 알고리즘의 완전성에 의해, $\mathcal{W}(\Gamma^{\mathcal{W}}, e')$가 성공한다. 이는 주어진 조건에 모순된다. 그러므로, $\mathcal{M}(\Gamma^{\mathcal{M}}, e', \rho)$는 실패한다. □

이제 정리 3을 증명할 차례이다.

정리 3의 증명. e에 타입 오류가 없을 때는 쉽게 증명된다. $|[\![\mathcal{W}(\Gamma, e)]\!]| = |[\![\mathcal{M}(\Gamma, e, \beta)]\!]|$임이 명백하기 때문이다.

e에 타입 오류가 있을 경우를 보자. $\mathcal{W}(\Gamma, e)$가 $(e_1\ e_2)$에서 실패한다고 하자. 그러면, $\mathcal{W}(\Gamma, e)$의 호출 문자열은 다음과 같은 형태이다.

$$[\![\mathcal{W}(\Gamma, e)]\!] = \cdots (\Gamma_1^{\mathcal{W}}, e_1\ e_2)^d \cdots (\Gamma_2^{\mathcal{W}}, e_2)^u$$

왜냐하면 $\mathcal{W}$ 알고리즘은 함수 적용문의 피연산자로부터 귀환하자마자 실패하기 때문이다.

다음을 가정하자.

$$|[\![\mathcal{W}(\Gamma, e)]\!]| < |[\![\mathcal{M}(\Gamma, e, \beta)]\!]|.$$

그러면,

$$[\![\mathcal{M}(\Gamma, e, \beta)]\!] = \cdots (\Gamma_1^{\mathcal{M}}, e_1\ e_2, \rho_1)^d \cdots (\Gamma_2^{\mathcal{M}}, e_2, \rho_2)^u \cdot \kappa\ (\text{단}, \kappa \neq \epsilon)$$

와 같은 형태를 갖는다. 왜냐하면 하위 표현식을 방문하는 순서는 $\mathcal{W}$나 $\mathcal{M}$이나 같고, e_2로부터 귀환한 다음 순서는 $e_1\ e_2$로부터 귀환하는 것이기 때문이다. 그러므로, κ는 $(\Gamma_1^{\mathcal{M}}, e_1\ e_2, \rho_1)^u \cdot \kappa'$의 형태이다. 이것은, 즉 $\mathcal{M}(\Gamma_1^{\mathcal{M}}, e_1\ e_2, \rho_1)$이 성공함을 뜻한다. 그러나, $\mathcal{W}(\Gamma_1^{\mathcal{W}}, e_1\ e_2)$가 실패하기 때문에, 보조정리 3에 의해, $\mathcal{M}(\Gamma^{\mathcal{M}}, e_1\ e_2, \rho_1)$은 실패한다. 그러므로, 가정은 사실이 아니다. □

3. 관련 연구

대부분의 이 분야에 관한 연구들은, 타입 유추 과정을 사용자에게 설명하는 것에 집중되어 있다. Wand[10]는 동일화 알고리즘을 수정하여, 매번 타입변수가 특정 타입으로 결정될 때마다, 장소 정보를 각 타입변수에 누적하여 기록하도록 하였다. 오류가 발생할 때, 오류의 원인인 타입변수에 기록된 장소 정보를 보여주도록 하였다. Beaven과 Stansifer[2]는 중복되는 정보를 제거함으로써 Wand의 방법을 향상시켰다. Duggan과 Bent[5]는 타입 유추 과정을 그래프에 기록함으로써 쉽게 유추 과정을 보여줄 수 있도록 하였다. 이러한 타입 유추 과정을 설명하는 방법은, 컴파일러는 필요없이 많은 정보를 가지고 계산을 하게 되고, 사용자는 복잡한 타입유추 정보 설명에 혼란을 받게 될 수 있다.

Johnson과 Walz[6]는 타입 정보를 보여주는 ML 전용 편집기(ML-oriented editor)를 구현하기 위해, 오류 허용(error tolerant) 동일화 알고리즘을 연구하였다. 하나의 타입 변수가 동일화 불가능한 여러 타입들로 결정되어야 할 때, 수가 많은 것을 옳은 것으로 간주하였다.

Bernstein과 Stark[3]은 타입유추 알고리즘을 수정하여, 정의되지 않은 변수에 대한 기대 타입을 유추하도록 하였다. 사용자가 프로그램의 의심스러운 곳에 정의되지 않은 변수를 삽입하여, 그 타입 정보를 얻어 프로그램을 수정할 수 있도록 하였다. 이 방법은 $\mathcal{W}$ 알고리즘은 물론, $\mathcal{M}$ 알고리즘에도 적용하여 사용될 수 있다.

4. 토의 및 결론

본 논문에서는 Hindley/Milner의 타입 유추 시스템을 충실히 구현하는 타입 유추 알고리즘 $\mathcal{M}$을 소개하고, $\mathcal{M}$ 알고리즘이 $\mathcal{W}$ 알고리즘보다 일찍 타입 오류를 감지함을 증명하였다. 일찍 오류를 감지한다는 것은, 타입 오류의 원인이 된 부분을 직접적으로 지적함으로써, 프로그램 작성자가 왜 유추 알고리즘이 성공하지 못했는지를 쉽게 이해하도록 해 준다.

정확한 타입 오류 장소를 지적한다는 것은 불가능하다. 정확한 타입 오류 장소의 정의가 불분명하기 때문이다. 타입 유추 알고리즘은 단지 프로그램 작성자가 무엇이 틀렸는지 찾을 수 있도록 힌트를 줄 수 밖에 없다. 여러 예제들과 일찍 오류를 감지한다는 사실을 증명한 것으로 볼 때, $\mathcal{M}$ 알고리즘은 $\mathcal{W}$ 알고리즘을 대체할 수 있는 새로운 알고리즘임은 분명하다.

참고문헌

[1] 이욱세. 좁은 범위의 타입 오류 정보를 위한 타입유추 알고리즘. 석사 논문, 한국과학기술원, 1997.

[2] 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.

[3] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.

[4] Luis Damas and Robin Milner. Principal type-scheme for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] Dominic Duggan and Frederick Bent. Explaining type inference. Technical Report UW TR CS-94-14, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, 1994.

[6] Gregory F. Johnson and Janet A. Walz. A maximal-flow approach to anomaly isolation in unification-based incremental type inference. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 44–57, 1986.

[7] Xavier Leroy. *Polymorphic Typing of An Algorithmic Language.* PhD thesis, Institut National de Recherche en Informatique et en Automatique, 1992.

[8] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Messachusetts, London, England, 1990.

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