프로그래밍 언어 메타이론의 정형화 및 변수 묶기

(Formalization of the Meta-Theory of a Programming Language with Binders)

이 계 식 †

(Gyesik Lee)

요 약 프로그래밍 언어의 구문 구조(syntax)와 메타이론을 정형화(formalization)하고 관련된 명세(specification)의 증명을 자동화(automatization)하는 과정에서 일어날 수 있는 모든 종류의 변수 묶기(variable binding)를 정형적(formal)으로 구현, 해결하는 방식을 개략적으로 소개한다. 또한 함수언어(functional language)의 기본으로 사용되는 Lambda calculus와 연계해서 POPLmark Challenge와 관련된 시도들의 공통점, 차이점 및 각각의 특성을 증명보조 툴인 Coq에서 구현된 간단한 예제들을 통해 보여준다.

키워드 : 프로그래밍 언어, 정형화, 변수 묶기, 증명보조 툴, Coq

Abstract We introduce some well-known approaches to formalization and automatization of the meta-theory of a programming language with binders. They represent the trends in POPLmark Challenge. We demonstrate some characteristics of each approach by showing how to formalize some basic notations and concepts of Lambda-calculus using the proof assistant Coq.

Key words : Programming Languages, Formalization, Variable Binding, Proof Assistants, Coq

1. 서 론

프로그래밍 언어와 관련된 증명들이 점점 길어지고 복잡해짐에 따라 프로그램 전체에 대한 조망은 보다 어려워질 수밖에 없다. 또한 작은 실수 하나가 프로그램 전체에 미치는 영향도 함께 증가한다. 따라서 정의와 정리 및 증명을 보다 체계적이고 일관되게 유지하고, 상호 유기적인 관계를 보다 효과적으로 입증해야 할 필요성이 점점 커져 왔다.

이에 대한 하나의 해결책으로 기계화된, 자동화된 증명, 즉 컴퓨터를 이용한 엄밀한 증명이 제시되었다. 1960년대 후반, 드 브로인(de Bruijn)이 수학적 증명을 컴퓨터로 구현하기 위해 고안한 Automath[1] 이후로 관련된 연구가 집중적으로 실행되어 왔고 현재는 자동화된 증명보조(automated proof assistant) 툴들이 그 전통을 이어 발전해왔다. ACL2[2], Coq[3], HOL[4], Isabelle[5], Lego[6], NuPRL[7] 등이 안정성을 인정받은 대표적인 툴들이다.

이러한 툴들의 발전을 제대로 이용하려면 이론적인 배경뿐만 아니라 기술적인 문제도 함께 다루어야 한다. 예를 들어, 보다 간결하고 명료한 표현방식이 요구되는데 이것은 다루어야 하는 내용이 길어지거나 복잡해질수록 그 의미가 더욱 커진다.

당장에 메타이론의 구문 구조(syntax), 프로그램 분석, 정형적인 의미구조(formal semantics), 실행적 의미구조(operational semantics) 등의 정의 및 증명 등을 정형적으로 구현하려면 그것들을 주어진 툴 내에서 간결하고 명쾌하게 표현해 낼 수 있어야 한다.

이렇듯 두 분야, 즉 프로그래밍 언어와 자동화된 증명보조 툴 분야가 상호 보완, 발전되어야 할 필요가 대두

· 이 논문은 2008 한국컴퓨터종합학술대회에서 '프로그래밍 언어의 메타이론의 정형화에 대한 locally nameless 방식과 두 종류의 변수를 사용하는 nominal 방식의 비교 분석'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : 산업기술총합연구소(AIST, Japan) 정보보안연구센터 (RCIS) PostDoc
gyesik.lee@aist.go.jp

논문접수 : 2008년 8월 25일
심사완료 : 2008년 10월 20일

정보과학회논문지: 소프트웨어 및 응용 제35권 제12호(2008.12)

되었으며 대표적으로 POPLmark Challenge[8]를 언급할 수 있다. 즉, 프로그래밍 언어 메타이론의 기계화 정도를 평가하려는 시도가 타입이론(type theory) 분야와 프로그래밍 언어 분야 자체에서 이루어지고 있다. 프로그래밍 언어의 정형화(formalization)를 평가하는 기준으로는

- 복잡하지 않은 인프라 구조
- 일상의 프로그래밍 언어 구문 구조와의 유사성
- 쉬운 접근성

등이 제시되어 있다. 이 기준들을 만족시키는 메타 언어와 그 특성에 대한 많은 연구가 이루어지고 있으며 변수 묶기(variable binding)의 구현이 중요 과제 중의 하나로 부상했다.

많은 프로그래밍 언어의 구문 구조(syntax)에는 특정 형태의 변수 묶기가 존재한다. Haskell, OCaml 등의 함수형 프로그래밍 언어(functional programming language)의 경우가 그러하며 그러한 구문 구조 내에서의 작업을 올바르게 정형화하기 위해서는 묶인 변수(bound variable 또는 binder)의 α-전환(α-conversion)이 요구되기도 한다. 예를 들어, 함수 $f(x)=x$를 함수 $g(y)=y$ 등으로 대체해야 하는 경우처럼 사용된 묶인 변수(bound variable)의 치환(substitution)이 종종 요구된다.

함수를 정의할 때 사용되는 변수의 이름은 전혀 중요하지 않음을 의미하는 α-전환의 기본 원리는 매우 간단하지만 그것을 컴퓨터에서 구현하는 일은 원리만큼 간단하지 않다. 일반적으로 수학자, 논리학자들은 이런 α-전환이 언제나 가능하며, 필요시 자동적으로 행해진다고 관습적으로 가정한다. 반면에 프로그래밍 언어와 그것의 메타이론의 정형화 또는 자동화 과정에서 구체적인 작업을 하다보면 α-전환은 하나의 해결되어야 할 과제로 주어진다. 이에 대한 해결책들이 프로그래밍 언어의 발전과 함께 다각도로 시도되어 왔으며, 드 브로인의 nameless representation[9](일명 de Bruijn indices), locally nameless[10], locally named[11,12], higher-order abstract syntax(HOAS)[13] 등을 대표적으로 언급할 수 있다.

본 논문에서는 α-전환의 구현과 관련된 시도들을 Lambda calculus의 간단한 메타 이론을 정형화(formalization)하는 방식을 이용하여 소개하고 각 시도의 장단점을 비교, 분석한다.

2. Lambda Calculus와 Coq

λ-calculus는 함수, 함수의 적용(application) 및 회귀(recursion)를 연구하기 위해 고안된 정형적 체계(formal system)이다. 1930년대에 처치(A. Church)와 클리니(S. C. Kleene)에 의해 처음으로 소개된 이후 수학 및 수리논리학에서 폭 넓게 사용되었을 뿐만 아니라 Lisp, Scheme, ML, Haskell, Ocaml 등의 함수형 프로그래밍 언어(functional programming languages)의 기초를 제공하였다. λ-calculus에 대한 자세한 내용은 바렌드레흐트(Barendregt)[14]를 참조할 수 있다.

반면에 이런 함수형 프로그래밍 언어들은 타입이론(type theory)에 기반하고 있다. 처치(A. Curch)와 커리(H. B. Curry)의 Simple Type Theory를 시작으로 해서 많은 타입 시스템 등이 소개되었으며 대표적으로 마틴-뢰프(P. Martin-Löf)의 (intuitionistic) type theory [15,16], 지라(J.-Y. Girard)의 system F[17], 꼬깡(Th. Coquand)과 위에(G. Huet)의 Calculus of Constructions (CC) [18] 등이 대표적이다.

언급된 타입 시스템들은 모두 Curry-Howard correspondence를 구현하였으며 앞서 언급된 증명보조 툴들의 기초를 제공하였다. 특히 꼬깡(Th. Coquand)과 위에(G. Huet)의 CC는 마틴-뢰프(P. Martin-Löf)의 메타언어(meta-language)를 하나의 구체적인 대상언어로 구현함으로 해서 새로운 프로그래밍 언어의 기저로 쉽게 사용될 수 있으며 실제로 증명보조 툴인 Coq에 사용되었다.

Coq은 고차 논리(higher-order logic)를 구현하는 CC를 기반으로 해서 모든 프로그래밍 언어의 기본인 귀납적 정의(inductive definition) 및 귀납법을 이용한 증명(proof by induction)을 지원한다. 또한 회귀함수(recursive function)의 정의 및 연산을 수행한다. 보다 자세한 사항은 [3]을 참조한다.

λ-calculus 및 모든 타입이론의 기본은 함수를 대상으로 한다. 주어진 어떤 값 x에 대해 값 t를 대응시키는 함수를 $\lambda x.t$로 나타내면서 그 함수가 어떤 값을 계산하는 것을 체계적으로 구현한다.

기본 구문 구조는 아래와 같다. 먼저 셀 수 있게 무한히 많은 변수 x, y, z 등을 원소로 갖는 집합 var의 존재를 가정한다. 그러면 λ-표식(λ-term)은 다음과 같이 귀납적으로 정의된다:

- 임의의 변수 x는 λ-표식이다.
- t와 s 모두 λ-표식이면 (ts)도 λ-표식이다.
- t가 λ-표식이면 모든 변수 x에 대해 $\lambda x.t$ 도 λ-표식이다.

이와 같이 한 종류의 변수를 이용해 자유 변수와 묶인 변수 모두를 구현하는 방식을 nominal 방식이라 하며 전통적으로 사용되어 왔다. 아래는 Coq 증명보조 툴에서의 정의 코드이다.

```
Inductive term : Set :=
| Var : var -> term
| App : term -> term -> term
```

| Lambda : var -> term -> term.

위 Coq 코드를 간단하게 설명하면 다음과 같다. Inductive 명령어는 이어서 정의되는, term이라 임의로 명명된 타입, 즉 λ-표식을 원소로 갖는 집합이 귀납적으로 정의됨을 암시하는 Coq 시스템 명령어이다. 집합 term의 원소를 생성하는 방식은 앞서 설명된 대로 세 가지가 있으며 각각의 방식은 Var, App, Lambda 라는 이름을 갖는 constructor를 통해 이뤄진다.

이와 같이 Inductive 명령어를 이용해 정의된 집합의 원소는 명시된 constructor에 의해서, 그리고 오직 그것들만을 이용해서 귀납적으로 만들어진다. 즉, 집합 term의 어떤 원소도 Var, App, 또는 Lambda로 시작되는 표식(term)으로 표현된다.

3. 묶인 변수, 즉 자리지킴이의 역할

주어진 λ-표식의 자유 변수(free variable) 또는 묶인 변수(bound variable 또는 binder)는 주어진 변수가 λ에 의해 묶여있는가의 여부에 따라 결정된다. 예를 들어 λ-표식($\lambda xy.x$)에서 y는 자유 변수이고, x는 묶인 변수이다. 묶인 변수의 이름만 다른 두 표식을 동일하다고 간주하는 것을 α-동치(α-equivalence)라 한다.

사실 묶인 변수는 일종의 자리지킴이(place holder)에 불과하다. 즉 어떤 집합의 전부를 대상으로 하지 어떤 특정 대상을 지칭하지 않는다. 따라서 묶인 변수를 무엇으로 선택하는지는 그리 중요하지 않다. 반면에 자유 변수는 임의지만 어떤 집합의 특정한 하나의 원소를 대변한다. 따라서 x, y 등의 서로 다른 이름은 일반적으로 서로 다른 값을 내포한다. 하지만 자유 변수와 묶인 변수를 혼합해서 사용할 경우 주의해야 할 요소가 하나 있다.

예를 들어보자. ($\lambda xy.x$)와 ($\lambda zy.z$)는 동일한 표식으로 간주되지만 ($\lambda xy.y$)와는 α-동치하지 않다. 실제로 후자는 전혀 다른 함수를 구현하고 있다. 이와 같이 하나의 변수를 다른 이름의 변수로 바꾸는 변수 치환의 경우 변수 조건(variable condition)을 고려해야 한다. 실제로 ($\lambda xy.y$)는 ($\lambda xy.x$)에서 두 번째 x를 y로 대체해서 만들어진 표식이지만 전혀 다른 함수를 구현한다. 즉, 전자는 두 개의 값이 주어졌을 때 두 번째 값을 취하지만, 후자의 경우에는 첫 번째 값을 취한다.

보다 일반적으로는 주어진 변수에 대해 임의의 λ-표식 t를 대입할 경우 위의 경우와 같은 변수의 충돌이 언제나 일어날 수 있으며 그런 변수의 충돌은 의도했던 함수의 구현을 방해하곤 한다. 다음은 변수대입의 정의이다. 이하에서 $[t/x]s$는 표식 s에 나타나는 자유 변수 x에 표식 t를 대입함을 나타내며 x와 y는 서로 다른 변수를 나타낸다.

표 1 변수 대입

$$
\begin{aligned}
[t/x]x &:= t \\
[t/x]y &:= y \\
[t/x](s_1 s_2) &:= ([t/x]s_1)([t/x]s_2) \\
[t/x](\lambda x.s) &:= (\lambda x.s) \\
[t/x](\lambda y.s) &:= \lambda y.([t/x]s)
\end{aligned}
$$

(단, 마지막 경우에서 t의 어떤 자유 변수도 $\lambda x.s$에서 묶여 있지 않아야 한다.)

이와 같이, 어떤 자유 변수에 대입되는 λ-표식이 이미 묶인 변수를 자유 변수로 포함하지 않아야 한다는 조건을 변수조건(variable condition)이라 한다. 이 변수조건이 위배되면 변수의 치환으로 얻어진 표식이 원래의 표식과 동치하다고 장담할 수 없게 된다. 다음의 대입정리가 변수조건의 또 다른 필요성을 보여준다.

대입정리(Substitution Lemma) x, y는 서로 다른 변수이고, t, s, u는 임의의 λ-표식이다. 만약에 x가 u의 자유 변수가 아니라면 다음의 등식이 성립한다.

$$[u/y]([s/x]t) \equiv [(s[u/y])/x]([u/y]t)$$

대입정리는 λ-calculus에서 사용되는 가장 기본적인 정리 중의 하나이며 타입이론 등의 관련된 모든 분야에서 동일한 역할과 의미를 갖는다. t에 대한 구조적 귀납법(induction on the construction of t)을 이용하여 증명할 수 있다. 변수조건이 충족되지 않을 경우 다음의 예에서 볼 수 있듯이 대입정리의 등식이 성립하지 않을 수도 있다. $t=(xy)$, $s=y$, $u=(xx)$라 하면

$$
\begin{aligned}
[u/y]([s/x]t) &\equiv uu \\
&\neq u(uu) \\
&\equiv [(s[u/y])x]([u/y]t)
\end{aligned}
$$

4. 묶인 변수(binder) 구현 방식

앞서 표 1에서 보았듯이 변수 대입을 하면서 대입되는 표식의 자유 변수와 기존의 표식의 묶인 변수간의 충돌이 없을 경우에만 변수 대입이 가능하다. 따라서 변수 충돌이 발생하지 않는지를 먼저 확인해 줄 필요가 있으며 그럴 경우 앞서 언급한 묶인 변수의 치환을 이용해 변수 충돌이 발생하지 않는 α-동치의 표식을 먼저 찾아줘야 한다.

x	**Var** x
(ts)	**App** t s
$\lambda x.t$	**Lambda** x t

앞서 언급한 nominal 방식에서 발생하는 α-동치의 표식을 찾거나 변수 충돌을 애초부터 피하거나 아니면

최소한 보다 효과적으로 다루고자 하는 목적으로 여러 시도들이 이루어져 왔으며 크게 두 부류로 나뉜다. 첫째, 변수와 변수 묶기를 구체적으로 다루는 대상으로 하는 부류가 있고, 둘째, 변수 묶기를 메타언어의 함수로 보는 방식이 있다. 둘째 부류에 속하는 방식으로는 대표적으로 HOAS(Higher-Order Abstract Syntax)[13]이 있다. 하지만 이 논문에서는 첫째 부류에 속하는 방식 중에서 중요 관심을 받는 방식 세 가지만을 다룬다.

4.1 드 브로인 인덱스(de Bruijn indices)

변수 조건과 관련된 변수 치환에서의 사실상의 해결책은 변수 대신에 숫자를 사용하는 **드 브로인 인덱스** [9]에 의해 제시되었다. 이 방식을 사용하면 α-동치인 표식들은 동일한, 즉 유일한 방식으로 표현된다. 따라서 변수치환의 필요성이 아예 발생하지 않는다. 드 브로인 인덱스 구현 방식을 이 자리에서 엄밀하게 정의하는 대신에 간단한 예를 들어 설명하겠다.

드 브로인 인덱스를 사용하는 방식은 상당히 기계적인 구문 구조(syntax)를 제공한다. 표현하고자 하는 표식의 오른쪽에서 시작하여 묶임 수준(binding level)을, 즉 변수의 왼편에 자신의 묶임을 제외하고 몇 번의 변수 묶임이 이루어졌는가를 숫자로 $0, 1, 2, 3$등으로 나타낸다. 예를 들어 λ-표식 $\lambda x.(yx)$을 드 브로인 인덱스 방식으로 구현하면 $\lambda.10$이 된다. $\lambda.10$ 에서 숫자 0은 변수 x가 최초로 묶였음을 나타내고, 숫자 1은 y 자신을 기준으로해서 왼편에 하나의 다른 변수, 여기서는 변수 x가 묶여 있음을 나타낸다. 즉, 숫자는 주어진 변수의 일종의 묶임 수준(binding level)을 나타낸다.

이와 같은 기계성은 드 브로인 인덱스 방식을 따르는 표현들이 실용상의 프로그래밍 언어의 그것들과는 매우 이질적이라는 치명적인 한계를 갖고 있다. 즉, 이 방식으로 구현된 프로그램을 인간이 이해하는 일이 그리 간단하지 않다. 더욱이 새로운 변수 묶임이 이루어질 때마다 숫자이동(shifting)이 이루어지기에 더더욱 그러하다. (숫자이동은 아래에서 보다 자세히 설명되어진다.) 기계적으로 처리되기에는 훌륭하지만 인간에게는 너무 기계적이라는 한계를 갖고 있다.

4.2 Locally nameless식 접근

앞에서 언급된 드 브로인 인덱스 방식의 한계를 극복하려는 시도가 바로 locally nameless 접근이다. 이 접근에서 변수 묶기는 드 브로인 인덱스 형식을 따르지만 자유 변수는 nominal 형식, 즉, 일상에서처럼 x, y, z 등의 구체적인 이름을 사용한다. 아래의 표 2에 드 브로인 인덱스 방식과 locally nameless 방식을 이용한 예제를 들었다.

λ-calculus의 구문 구조를 locally nameless식으로 정의하면 아래와 같다. 먼저 셀 수 있게 무한히 많은 변수 x, y, z 등을 원소로 갖는 집합 var의 존재를 가정한다. 하지만 λ-표식을 정의하기 전에 먼저 예비표식(pseudo-term)을 정의할 필요가 있다.

표 2 형식적 바인딩

방식	$\lambda x.(yx)$
nominal	$\lambda x.(yx)$
de Bruijn	$\lambda.10$
locally nameless	$\lambda.y0$

- 임의의 변수 x는 예비표식이다.
- 임의의 자연수 n은 예비표식이다.
- t와 s 모두 예비표식이면 (ts)도 예비표식이다.
- t가 예비표식이면 $\lambda.t$도 예비표식이다.

아래는 Coq 코드이다. (nat은 자연수들의 집합이다.)

```
Inductive pre_lterm : Set :=
| LocVar : var -> pre_lterm
| LocNat : nat -> pre_lterm
| LocApp : pre_lterm -> pre_lterm ->
pre_lterm
| LocLambda : pre_lterm -> pre_lterm.
```

앞서의 nominal 방식에서의 Coq 코드와의 근본적인 차이점은 마지막 항목에서 볼 수 있다. 여기에서 보면 묶인 변수를 나타내는 항목이 없다. 즉, 변수라는 이름 대신에 자연수에 의해 묶인 변수의 위치 및 묶인 정도를 동시에 구현하고 있다.

예비표식에는 자연수, 즉 묶인 변수를 자유로이 사용할 수 있다. 하지만 묶인 변수는 원래 어떤 의미를 갖기보다는 앞서 설명하였듯이 자리지킴이(place holder) 역할을 한다. 따라서 묶이지 않은 숫자는 어떤 의미도 갖지 않는다. 이런 의미에서 잘 정의된 표식은 모든 숫자가 '특정한 규칙'에 따라 묶여있는 예비표식으로 정의된다. 여기에서 '특정한 규칙'을 일일이 나열하는 대신에 아래의 설명으로 대신한다.

Locally nameless 접근의 장점은 nominal 접근과 드 브로인 인덱스의 장점을 조합한 데에 있다. 실용상의 언어로부터 상대적으로 덜 이질적이면서 변수치환의 문제를 해결하였기 때문이다. 즉, 변수 충돌이 전혀 발생하지 않는다. 이런 이유로 일부에서는 최선의 방식이라고 주장하기도 한다. 하지만 숫자를 변수로 사용함으로써 발생하는 가독성(readability)의 한계는 여전히 남아 있다. 특히나 아래에서 설명할 숫자이동의 문제는 드 브로인 인덱스를 사용하는 모든 스타일의 방식에의 접근을 어렵게 만든다.

숫자이동(shifting)을 가능하면 일반적으로 설명하기 위해 앞의 예보다 좀 더 복잡한 표식을 살필 필요가 있다. locally nameless 방식을 따를 때 다음과 같은 표식

을 얻는다.

$$\lambda x.(x(\lambda y.yxz)) \equiv \lambda.0(\lambda.01z)$$

이렇게 완성된 표식을 보면서 숫자의 사용을 이해하는 건 별로 어렵지 않다. 처음의 0은 변수 x가 처음으로 묶였음을, 두 번째의 0은 변수 y가 처음으로 묶였음을, 그리고 숫자 1은 같은 곳에 위치한 변수 x가 묶이기 전에 이미 y가 묶였음을 의미한다. 끝으로 z는 자유 변수이다.

반면에, 위 과정을 구문적(syntactic)으로 완성시키는 과정을 설명하려면 일이 좀 더 복잡해진다. 위의 표식이 구문적으로 잘 정의되려면 먼저 그것의 부분표식(sub-term)인 $(0(\lambda.01z))$ 또한 잘 정의 되어 있어야 하는데 그렇지가 않다. 처음의 0과 숫자 1이 묶여 있지 않았는데 이것은 묶인 변수만 숫자로 표현한다는 문법에 위배된다.

이점을 피하기 위해 숫자에서 이름으로, 그리고 다시 이름에서 숫자로 치환하는 다음과 같은 과정이 요구된다.

$$0(\lambda.01z) \Rightarrow x(\lambda.0xz) \Rightarrow \lambda.0(\lambda.01z)$$

즉, 예비표식 $(x(\lambda.0xz))$가 구문적으로 잘 정의된 표식임을 먼저 증명해야 하며, 여기서 변수 x는 기존에 사용되지 않은 임의의 자유 변수이다. 이와 같이 숫자에서 이름으로, 이름에서 숫자로의 연속된 치환을 통해서야만 주어진 예비표식이 구문적으로 잘 정의된 표식인지를 확인할 수 있다.

문제는 어떻게 숫자와 이름 사이의 치환을 해야 하는가이다. 동일한 숫자라면 일괄되게 치환을 해주면 되지만 다른 숫자가 같은 변수를 표현하기 때문에 숫자이동을 염두 해야만 한다. 즉, 표식의 왼쪽에서부터 0으로 출발해서 변수 묶음(variable binding)이 발생할 때마다 1씩 더해가는 과정을 거쳐서 0, 1, 2 등으로 순서대로 진행되는 숫자를 아직 사용되지 않은 새로운 이름, 예를 들어 x로 치환한다. 이런 과정을 통해 $(x(\lambda.0xz))$라는 잘 정의된 표식을 먼저 구한다. 이 과정을 반복해서 적용하면서 각각의 부분표식들이 모두 잘 정의되었는지를 확인해야 한다. 물론 모든 확인 과정에서 숫자이동을 계산해야 한다. 보다 자세한 설명은 [10]을 참조한다.

4.3 Locally named식 접근

묶인 변수의 α-전환(α-conversion)의 필요성은 앞서 설명하였듯이 자유 변수를 다른 표식으로 치환할 경우 발생할 수 있는 변수들의 충돌을 피하기 위해서이다. 드브로인 인덱스 스타일과 그것의 변형인 locally nameless 시도는 그런 충돌을 애초부터 피하도록 고안되었고 기계적으로 뛰어난 효율성을 보이지만 일상 프로그래밍 언어에서 사용되기에는 가독성이 떨어진다는 한계를 갖고 있다. 따라서 변수의 충돌을 피하려면 숫자를 사용해야 하고 더불어 숫자이동(shifting)을 감수해야 하는가? 라는 질문이 던져진다. 반면에 자유 변수와 묶인 변수의 구분은 필연적으로 보인다.

위의 질문에 대한 답으로 McKinna-Pollack[11,12]은 숫자 대신에 새로운 종류의 이름을 사용하는 방식을 Lego에서 구현하였으며, Herbelin-Lee[19]는 변형된 방식으로 locally named 방식을 Coq에서 구현하였다. 즉 자유 변수와 묶인 변수에 각기 다른 종류(different sort)의 이름을 부여한다.

다음은 locally named 방식을 따른 λ-calculus의 구문 구조이다. 먼저 셀 수 있게 무한히 많은 자유 변수 a, b, c 등을 원소로 갖는 집합 fvar와 역시 셀 수 있게 무한히 많은 묶인 변수 x, y, z 등을 원소로 갖는 집합 bvar이 존재한다고 가정한다. 여기에서도 예비표식을 먼저 정의해야 한다.

- 임의의 자유 변수 a는 예비표식이다.
- 임의의 묶인 변수 x는 예비표식이다.
- t와 s 모두 예비표식이면 (ts)도 예비표식이다.
- t가 예비표식이면 모든 묶인 변수 x에 대해 $\lambda x.t$ 도 예비표식이다.

아래는 Coq 코드이다.

```
Inductive pre_nterm : Set :=
| NameFvar : fvar -> pre_nterm
| NameBvar : bvar -> pre_nterm
| NameApp : pre_nterm -> pre_nterm ->
            pre_nterm
| NameLambda : bvar -> pre_nterm ->
               pre_nterm.
```

위 Coq 코드에서 주의 할 부분은 역시나 마지막 항목이다. bvar의 원소만이 묶인 변수(binder)로 사용될 수 있다.

여기에서 예비표식을 먼저 정의한 이유는 locally nameless의 접근과 같은 이유에서이다. 예비표식 중에서 모든 묶인 변수가 정말로 묶여 있으면 구문적으로 잘 정의된 표식이라고 한다. 차이점으로 변수 이동(variable shifting) 등이 전혀 요구되지 않을 뿐더러 서로 다른 종류의 변수를 사용함으로써 자유 변수에 대한 치환을 실행할 경우 발생할 수 있는 변수들의 충돌 문제 또한 자연스럽게 해결되었다. 아래의 표 3은 locally named 방식에서의 자유 변수에 대한 변수 대입의 귀납적 정의를 나타낸다. 변수에 대한 어떠한 형태의 조건이 없다는 데에 주목할 필요가 있다.

반면에 locally nameless식 접근에서 얻을 수 있는 표식의 유일성은 가질 수가 없다. 예를 들어, $(\lambda x.(yx))$을 표현함에 있어서 locally nameless식으로는 $(\lambda.(y0))$이라는 유일한 표현이 있지만 두 종류의 변수를 사용하는 nominal 식에서는

표 3 locally named 방식의 변수 대입

$$
\begin{aligned}
[t/a]a &:= t \\
[t/a]b &:= b \ (\text{단, } a \neq b) \\
[t/a]x &:= x \\
[t/a](s_1\, s_2) &:= ([t/a]s_1)([t/a]s_2) \\
[t/a](\lambda x.s) &:= \lambda x.([t/a]s)
\end{aligned}
$$

$$\lambda x.(yx) \equiv \lambda z.(yz) \equiv \cdots$$

에서 볼 수 있듯이 무한히 많은 α-동치의 표식이 존재한다.

하지만 이것은 의미가 별로 크지 않아 보인다. 실제로 Aydemir et al.[10]과 Herbelin-Lee[19]의 두 논문에서 시도된 구체적인 활용 예들에 있어서 표식 표현의 유일성이 전혀 문제시 되지 않았다. 더불어 [19]의 경우 어떠한 형태의 α-전환이 요구되지 않았다.

하지만 때로는 α-전환이 요구되는 경우가 발생할 수 있다. 예를 들어 [11]에서 Pure Type System 일반을 다루다보면 발생하는 경우에서처럼 예비표식의 β-변환(β-reduction)을 정의할 때가 그렇다.

$$(\lambda x.t)s \rightarrow [s/x]t$$

하지만 잘 정의된(well-defined) 표식의 β-변환의 경우 치환되는 표식 s 또한 잘 정의되어 있기 때문에 α-전환이 전혀 요구되지 않는다. (s에 묶이지 않은 묶인 변수가 나타나지 않기 때문이다.) 따라서 이런 한계가 얼마나 크게 작용하는지는 좀 더 분석이 필요하다.

Locally named 방식의 또 하나의 단점은 자연수가 아닌 이름을 사용하는 것에서 기인한다. 같은 동전의 이면이라 할 수 있다. Coq을 포함해서 대부분의 증명보조 툴들은 자연수 및 여러 기본 함수와 연산을 내재하고 있다. 즉, 숫자이동 등을 정의할 때 관련된 많은 성질들을 따로 증명할 필요가 없다. 반면에 locally named 방식에서는 보다 많은 구문론적(syntactical), 의미론적(semantical)인 작업이 요구된다. 많은 보조정리를 추가로 증명해야 한다.

반면에 가독성 및 일상의 프로그래밍 언어와의 유사성은 자유 변수와 묶인 변수를 구분하는 관습을 따르면서 자연스럽게 보장되었다. 사실 변수의 구분은 메타 논리적으로도 의미가 있다. 앞서 설명하였듯이 묶인 변수는 일종의 자리지킴이(place holder)에 불과하다. 즉 어떤 집합의 전부를 대상으로 한다. 반면에 자유 변수는 임의지만 어떤 집합의 특정한 하나의 원소를 대변한다. 따라서 x, y 등의 서로 다른 이름은 일반적으로 서로 다른 값을 내포한다.

4.4 Nominal 방식과 메타 이론의 조화

끝으로 묶인 변수(binder)의 구현 방식과 관련된 또 하나의 시도를 소개하고자 한다. 지금까지 살펴보았듯이 논의의 핵심은 어떻게 하면 변수 충돌을 피하는가이다. 이에 대한 해결 방안으로 그것이 자연수이든, 다른 종류(sort)의 변수이든 변수의 집합을 공통 원소를 공유하지 않는 두 집합으로 나누었다. 즉, 전통적인 nominal 방식을 피하면서 해결책을 모색하였다.

반면에 Herbelin-Lee[19]에서는 nominal 방식에 한 가지 새로운 시도를 접합하였다. 즉, 변수 조건(variable condition)이라는 메타 이론적 성질을 구문 구조(syntax)에 포함시켰다. 아래는 Coq에서 표 1에 정의된 변수 대입을 정의하는 코드이다.

```
Fixpoint subst (t:term)(x:var)(u:term){struct t}
  : var_cond u t -> term :=
  match
    t return var_cond u t -> term
  with
  | Var y => fun b =>
      if (var_eq_dec x y) then u else (Var y)
  | App (s1 s2) => fun b =>
      App (subst s1 x u b) (subst s2 x u b)
  | Lambda y s => fun b =>
      if (var_eq_dec x y) then
      (Lambda y s)
      else
      Lambda y (subst s x u (vc_inv s x u b))
  end.
```

위의 Coq 코드를 설명하면 다음과 같다. 먼저 Fixpoint는 subst라 불리게 될 함수가 recursive하게 정의됨을 의미하는 명령어이고, 함수 subst는 네 개의 변수를 가지며 처음 세 개의 매개 변수는 각각 term, var, term의 원소를, 마지막 네 번째 매개 변수는 (var_cond u t)에 대한 증명, 즉 u와 t 사이에 변수 조건의 성립을 보이는 임의의 증명을 취한다. 그리고 {struct t}는 t의 모양에 따른 pattern-matching 형식으로 subst 함수가 정의됨을 나타내며, 따라서 이후의 정의는 Var, App, Lambda의 세 경우로 이루어진다. 정의에서 (var_eq_dec x y)은 변수 x와 y의 구분 가능성, 즉 {x = y} + {x ≠y} 을 나타내며, 변수 집합을 정의할 때 구분 가능성(equality decidability)이 증명될 수 있도록 집합 var를 귀납적으로 정의할 수 있다. (자연수 집합을 정의하는 방식과 동일하다.)

위 정의의 주요 특징은 (var_cond u t)로 구현되는 변수 조건이 정의 자체의 하나의 표식으로 사용된다. 전통적인 nominal 방식에서는 이것을 하나의 표식으로 사용하지 않고 변수 조건이 성립할 경우와 아닐 경우로 나누어 정의하였다. 즉, (if ... then ... else ...)를 한 번 더 사용한다. 마지막 항목에서의 (vc_inv s x u b)는

(var_cond u (Lambda y s))이 성립할 때 (var_cond u s) 또한 성립함을 증명하는 함수를 의미한다.

이와 같이 변수 조건의 구문 구조(syntax)의 일부로 사용할 경우 (if ... then ... else ...)를 사용할 경우보다 다루어야 할 경우가 줄어들게 되는데 이는 변수 조건이 만족되는 경우만 다루기 때문이다. 반면에 vc_inv와 같은 변수 조건이 부분 표식(sub-term)으로 세습되어 성립함을 보여야 하지만 그런 증명은 일반적으로 매우 간단하다.

변수 조건이 만족되는 경우만을 다루기에 충분히 일반적이지 않다 라고 볼 수 있지만 그렇지 않다. 실제로 α-동치인 표식을 필요할 경우 만들어 내는 건 간단하다. 또 위와 같은 정의를 이용해서 논리적으로 충분히 복잡한 내용을 정형화 하는데 전혀 문제가 없었다. 실제로 [19]에서 수리 논리학에서의 괴델(K. Gödel)의 완전성 증명(completeness theorem)을 정형적으로 증명하였다.

5. 활용성 및 결론

묶인 변수의 구현으로 제안된 locally nameless와 locally named 방식 모두 일상적으로 훌륭하게 활용될 수 있다. 전자의 경우[10]를 비롯하여 이미 다양한 형식으로 그 응용성을 인정받았다. 후자의 경우는 [11,12]를 통해 가능성을 보였다.

경험에 의하면 두 접근방식을 응용성 면에서 구분지을 수는 없어 보인다. 다만 경우에 따라 서로의 장단점이 다르게 드러날 수는 있다. locally nameless의 경우 기계적인 해석이 가장 우수하고, 더불어 많은 경우 보다 짧은 코드를 이용한 증명이 가능하다. 이것은 정형화된 증명(formalized proof)의 보다 빠르고 단순한 자동화(automatization)에 적합하다고 할 수 있다.

반면에 가독성의 문제는 일상의 프로그래밍 언어로 사용되는 데에 어느 정도의 한계를 가질 수밖에 없다. 게다가 기술적으로도 몇 가지 문제를 내포하고 있다. 앞서 언급된 숫자이동(shifting)의 경우를 보면 표식의 복잡성이 증가할 때마다 - 실제로 임의로 큰 복잡성을 가진 표식을 만들어 낼 수 있다 - 그것의 복잡성도 함께 증가한다.

사실 가독성의 문제는 경우에 따라 그 비중이 변한다고 할 수 있다. 실제로 프로그램을 만들다 보면 가독성이 그리 크게 문제되지 않는다고 말하는 경우도 많다. 하지만 단순히 프로그램을 짜는 것에 더해 다른 목적, 예를 들어 교육 등에 사용하려는 목적이 있다면 가독성은 중요한 변수로 작용할 수 있다.

Locally named 방식의 가장 큰 장점은 이런 점에서 보았을 때 바로 뛰어난 가독성과 일상에서 사용되는 언어와의 동질성이다. 반면에 앞서 언급한 단점을 갖고 있다. 자연수를 사용하면 사용 툴에 이미 내재되어 있는 정리와 증명을 이용하기만 하면 되지만, 변수를 사용하면 어찌 보면 자명한 것들까지도 일일이 따로 증명해야 하는 경우가 많다. 이는 짜야 할 코드의 양을 증가시킬 수 있으며 보다 많은 시간과 에너지를 요할 수 있다.

Locally nameless와의 또 다른 경험을 통해 확인된 사실은, 동시 치환(simultaneous substitution)의 경우 두 종류의 묶인 변수, 즉 두 종류의 숫자가 요구된다는 것이다. 즉, $0,1,2$ 등뿐만 아니라 $0',1',2'$ 등 또한 필요하다. 이것은 같은 묶임 수준(binding level)을 갖는 서로 다른 두 개의 묶인 변수를 동시에 치환하는 것을 구현해내기 위해서 필요하다. 예를 들어 '모든 쌍 (x,y)에 대해서 ...'와 같은 표현에서처럼 쌍(pair)으로 이루어진 변수묶음을 구현하고자 할 때 그와 같은 문제가 발생하며, 이런 종류의 변수묶음은 type theory에서 언제나 나타날 수 있다. 즉, 변수 x와 y를 차례대로가 아닌 동시에 치환할 경우가 있는데 그러려면 두 변수의 묶임 수준을 따로따로 서로 독립적으로 표현할 수 있어야 한다. 즉, 두 종류의 묶인 변수가 요구된다. 따라서 앞서 언급된 바와 같이 새로운 숫자에 대한 여러 성질을 역시나 새롭게 증명해야 한다. 반면에 locally named 방식에서는 묶음수준을 고려할 필요가 없고 언제나 쉬운 표현이 가능하다.

두 접근방식의 공통된 장점은 (Locally named 방식의 경우 부분적으로 이긴 하지만) α-전환(α-conversion)이 불필요하다는 점이며 이는 자유 변수와 묶인 변수를 구문론적으로 구분을 하기 때문이다. 따라서 변수 대입(substitution)도 구조적 귀납법(structural induction)을 이용하여 쉽게 정의될 수 있다. (두 종류의 이름을 사용하는 nominal 시도의 경우에는 잘 정의된 표식들의 경우에 한정된다.)

결론적으로 변수충돌의 해결책은 본질적으로 자유 변수와 묶인 변수의 구문론적 구분에 있다고 말할 수 있다. 하지만 어떤 방식이 보다 우수하다고 결론내리기는 어려워 보인다. 두 시도 모두 장단점을 갖고 있으며 어디에 어떤 목적으로 활용되는가에 따라 무엇을 사용할 것인가를 결정할 필요가 있다. 저자의 경험에 의하면 구문론적인(syntactic) 정형화를 시도하는 경우에는 locally nameless가, 의미론적(semantic)인 정형화의 경우에는 nominal 방식과 locally named 방식이 좀 더 원래의 의도에 부합하였다.

참 고 문 헌

[1] N. G. de Bruijn, AUTOMATH, a language for mathematics. Technical Report 68-WSK-05, T.H.-

Reports, Einhoven University of Technology, 1968.
[2] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers. 2000.
[3] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development (Coq'Art: The Calculus of Inductive Constructions)*, volume XXV of *EATCS Texts in Theoretical Computer Science.* Springer-Verlag, 2004.
[4] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press. 1993.
[5] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A Proof Assistant For Higher-Order Logic, volume 2283 of Lecture Notes in Computer Science. Springer-Verlag. 2002.
[6] Z. Luo and R. Pollack. *The LEGO proof development system: A user's manual.* Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.
[7] R. L. Constable, S. F. Allen, M. Bromley, R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall. 1986.
[8] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. 2005. http://www.cis.upenn.edu/~plclub/wiki-static/poplmark.pdf
[9] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae, 34(5):381-392, 1972.
[10] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3-15. ACM Press, 2008.
[11] J. McKinna and R. Pollack. Pure Type Systems formalized. In *Typed Lambda Calculi and Applications* (TLCA '93), volume 664 of LNCS, pages 289 - 305. 1993.
[12] J. McKinna and R. Pollack. Some lambda calculus and type theory formalized. Journal of Automated Reasoning, 23(3-4):373-409, 1999.
[13] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199-208. ACM Press, 1988.
[14] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1984.
[15] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.
[16] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.
[17] J.-Y. Girard. The System F of Variable Types, Fifteen Years Later. Theor. Comput. Sci. 45(2): 159-192. 1986.
[18] Th. Coquand and G. Huet. The Calculus of Constructions. Information and Computation. 76(2/3). 1988.
[19] H. Herbelin and G. Lee. Semantical normalisation using Kripke models for full predicate logic (a case study on the representation of binders). In preparation, 2008.

이 계 식

1992년 서울대학교 수학과 졸업(학사) 1996년 독일 Univ. of Muenster 수학 및 전산학과 졸업(석사). 2005년 독일 Univ. of Muenster 수학 및 전산학과 졸업(박사). 2005년~2007년 프랑스 INRIA Postdoc. 2007년~현재 일본 AIST Postdoc. 관심분야는 type theory, theorem proving, language-based security