Calculus of Constructionsのステージ化

鈴木　歩　　浅井　健一
お茶の水女子大学
ayumi@pllab.is.ocha.ac.jp
asai@is.ocha.ac.jp

概 要

依存型とは「式の型が項に依存して決定する型」である。依存型を使うと型に豊富な情報を入れることができ、実際に Dependent ML における array bound check などで有効に利用されている。しかし、型に書くことができる項は算術計算などの簡単なものに制限されている。一方、Calculus of Constructions（CoC）では任意の式を型として書くことができる。そこで、本稿では表現力の大きい CoC をプログラミング言語として用いるために CoC のステージ化を提案する。CoC の式を複数のステージに分解し、各ステージで型チェック・実行を行うことで、型の表現力の豊かさを失うことなく CoC の式を通常の関数型言語に近い効率で実行できるようになる。本稿ではそのようなステージ化を行うための型規則を提示し、種々の例についてステージ化できることを示す。現在のところ型規則の健全性等の証明はできていないが、プロトタイプを実装しており、多くの CoC の式を実行することができている。

1　はじめに

現在広く利用されている型付き言語は、コンパイルを行う段階で型情報からプログラム全体の整合性のチェックを行っている。プログラムはこの「型チェック」を行うことによって、実行時に起こり得るエラーの多くを事前に検出するとともに、実行時の動的なチェックを行わずにすむために、高速な実行をすることができる [11, 12]。 このような利点をさらに有効に利用するため、近年では型の表現力をより豊富にし、様々なプログラムの性質を表そうという試みが活発に行われている。リストや配列の型にその長さの情報を付け加えることで実現できる array bound check [13] はそういった例のひとつとして挙げられる。また、Cayenne [2] のように型にさらに複雑な情報を付け加えることができるシステムも存在する。

しかし、型により強力なことが書けるようになるにつれ、その型チェックは難しくなる一方だ。型内に複雑なものを書くと、それだけチェックは困難となる。従来のシステムでは型チェックを実行前に一度だけ行うことを前提としているので、型に書けるものは算術計算などの容易にチェックが可能なものに制限されてきた。そこで、本稿では逆のアプローチを展開する。それは即ち、「型チェックが実行前に一度だけ行われる」という前提を崩し、代わりに型の部分にも任意の式を書くことのできる Calculus of Constructions (CoC) [6] の式を書く事を許す。これにより、型のレベルにおいても他の項と全く変わらないプログラミングをすることが可能になり、任意に強力な性質を表現することが実現できる。

ところが、型レベルにおいても任意の CoC の項を書く事を許可すると、型の同値性判定のところで問題になってくる。一般に型チェックにおいては型の同値性判定が必要である。単純型付き λ 計算ではこの同値性は型の構造の同値性により簡単に判定できるが、型レベルに CoC の式が許されるとどうしても実行をしなければ型の比較を行うことができない、あるいは比較が難しい部分が登場する。例えば (`tuple` (`n` + `1`)) と (`tuple` (`1` + `n`)) を比較することを考える。自由変数 `n` の値が未知の状態でこのふたつの式を比較しようと思えば `n` + `1` という式のレベルでの同値性を判定しなければならない。これは簡単な定理証明系が必要となるなど重い処理となる。また、定理証明系で示せる範囲に書ける式が限定されたりする。

これは通常の型付き関数型言語の実行とは対照的である。通常の型付き関数型言語の実行では自由変数は全て環境に保持され、値の比較や計算をする際には値が全て確定している。これにより定理証明系など

の重い処理を排除し、効率的な実行が可能になっている。たとえユーザが `if n+1 = 1+n then ...` というプログラムを書いたとしても、この時点で `n` の値はわかっているはずであり、その値を使って `n+1` と `1+n` を計算し、比較すればよい。

そこで本稿では、このような効率のよさを犠牲にせず、なおかつ型付き言語における「式を実行する前には必ず事前に型チェックが行われている」— 即ち、信頼性と高速実行の原則を保ったまま、任意の CoC の式を実行する方法を提案する。具体的には「型チェック」と「その際に必要な実行」を元の式よりひとつ上のステージであると考え、全ての式を複数ステージに分割するための型規則を示し、それに基づいて種々の例がステージ化できることを示す。ステージ化により、自由変数の値は全て事前確定するという性質が保障され、従って通常の型付き関数型言語のような効率的な実行が可能となる。このような複数ステージによる実行体系を考えることは、CoC に基づくプログラミング言語や実行時コード生成をするシステムを作る際の礎石となると考えられる。そしてさらには、従来の依存型を含む体系では不明確であった Phase Distinction [3] に対するひとつの解を与えるものになると期待される。

本研究は CoC に基づいたプログラミング言語を設計するとどのようなことになるのかを探ることが第 1 の目的であるため、型規則に対する性質の証明などはまだ行っていない。そういった意味では発展途上の研究ではある。しかし、ここに示すようなステージ化は CoC の式の実行には有益なものであると考えられ、将来的には、型レベルに任意の式を許すプログラミング言語の設計の基礎を与えるものと期待される。

以下、2 章は関連研究として型を強力にしたシステムである Dependent ML と Cayenne を紹介し、3 章で CoC の構文・型規則を紹介する。 4 章ではプログラムをステージ化するために作成した実行順序付きの型規則を導入し、種々の例についてステージ化を行う。5 章では 4 章で説明したシステムを実装する方法を述べ、6 章でまとめる。

2 関連研究

プログラミング言語中に現れる型を強力にする研究は種々なされている。ここではその中から代表的なものとして Dependent ML(DML) [13] と Cayenne [2] について述べる。

2.1 Dependent ML

DML は、依存型をプログラミング言語に導入することで、例えばリストの長さの情報を型に書けるようにした言語である。これにより array bound check をすることができる。DML は型に書ける情報を算術計算などに制限することで、静的に型チェックをできるようにしている。この算術計算は型チェック時に定理証明系を使って行われる。この研究をさらに進め、CPS 変換における対象言語の情報を型に入れ込むことにより、変換後の式が対象言語の native な型を持つようにした研究も存在する [4]。

本研究が DML と異なる点は、型チェック、実行のレベルを複数回行うようにしたことである。これによって自由変数が型チェック前に全て確定し、型チェック時に定理証明系を使う必要がなくなる。また、型に書ける情報が制限されることもない。これは、DML を発展させた $\mathcal{ATS}$ のアプローチに近い。$\mathcal{ATS}$ では、定理証明系を使う代わりに証明をユーザが書くようになっている [5] 。本研究はそのような状況でさらに型チェック、実行のレベルを複数回行うようにしていると捉えることができる。

2.2 Cayenne

型により多くの情報を書けるようにしたプログラミング言語として Cayenne があげられる。Cayenne では、型に（制限はあるものの）多くの式を書くことができる。Cayenne の型チェックは決定不能だが、実用上は多くの場合、問題ないと報告されている。Cayenne を使うと、universal type を使うことなく、型

のついた言語に対するインタプリタを書くことができるようになる [1]。このインタプリタを使うと、通常の特化を行うだけで型の特化 [8] が行われると考えられている。

本研究のアプローチは、静的な型チェックをあきらめ型になるべく多くのことを書けるようにするという意味で Cayenne のアプローチと近い。一方、Cayenne は型チェックが実行前に一度のみ行われることを仮定している。そのため Cayenne の型チェックは式の実行を含むと思われるが、そこの部分の処理がどのようになっているかは明らかではない。本研究は、型チェック中に行われる実行を別ステージとして実行することで、各ステージの実行がすべて型チェック後に行われることを保証するものとなっている。

また先に挙げた Cayenne で作成されたインタプリタと同様の手法を用い、そのインタプリタをステージ化してコンパイルする手法を示すような研究も存在する [7]。しかし、ここで言うステージ化とは部分評価 [9] における束縛時に対応するもので、本稿で考えている型と項の依存関係を基にしたステージ化とは異なるものである。

3 Calculus of Constructions

依存型（dependent-type）とは「型の値が項に依存して決定するような型」である。通常の型付き λ 計算にこの依存型を導入し、型の部分に全ての式を書ける様に拡張したシステムのことを Calculus of Constructions (CoC) という。この章では本稿で扱う CoC の構文・型規則を紹介する。

3.1 CoC の構文

CoC の構文は以下の様になっている。

$$\begin{aligned}
&\mathtt{E} \in \textit{Expression} \quad \mathtt{N} \in \textit{Integer} \\
&\mathtt{E} ::= \quad \mathtt{int} \mid \mathtt{N} \mid \mathtt{x} \\
&\qquad \mid (\lambda \mathtt{x} : \mathtt{E}_1.\mathtt{E}_2) \mid (\mathtt{E}_1 @ \mathtt{E}_2) \mid (\Pi \mathtt{x} : \mathtt{E}_1.\mathtt{E}_2) \mid * \mid \Box \\
&\qquad \mid (\mathtt{tuple\ E}) \mid (\mathtt{init\ E}_1\ \mathtt{E}_2) \mid (\mathtt{lookup\ E}_1\ \mathtt{E}_2)
\end{aligned}$$

依存型はその性質から型にも項が含まれるために、型も式と全く同じ Syntax を持つ。int は整数を表す型の定数、N は整数の項を表し、x は変数を表している。$(\lambda \mathtt{x} : \mathtt{E}_1.\mathtt{E}_2)$ は関数抽象、$(\mathtt{E}_1 @ \mathtt{E}_2)$ は関数適用である。$(\Pi \mathtt{x} : \mathtt{E}_1.\mathtt{E}_2)$ は型レベルの関数で、「型が $\mathtt{E}_1$ である値 x を受け取り、型が $\mathtt{E}_2$ の値を返す」ものである。この結果の型 $\mathtt{E}_2$ は、引数 x の値に依存することがある。 $\mathtt{x} \notin \mathtt{E}_2$ なら $\mathtt{E}_1 \to \mathtt{E}_2$ と同意である。

$*$ は一般に kind と呼ばれている。$*$ は「型が属する集合」を表している。例えば $\mathtt{int} : *$ や $\alpha : *$ のように用いられるが、特に後者は「α は型」、即ち「α は型変数」という意味である。$\Box$ はさらに「$*$ の属する集合」を表す。その例としては $* : \Box$ や $(\Pi \mathtt{x} : *.*) : \Box$ などが挙げられる。

通常の CoC の構文はここまでであるが、これだけではあまり依存型を用いた意味のある例が作成できないので、本稿では [12] で使われているリストに関する構文をみっつ追加する。(tuple E) は長さが E のリストの型であり、対して $(\mathtt{init}\ \mathtt{E}_1\ \mathtt{E}_2)$ は $(\mathtt{tuple}\ \mathtt{E}_1)$ 型を持つ長さ $\mathtt{E}_1$、全ての要素が $\mathtt{E}_2$ であるリストを生成するものである。そして $(\mathtt{lookup}\ \mathtt{E}_1\ \mathtt{E}_2)$ はリスト $\mathtt{E}_2$ の $\mathtt{E}_1$ 番目を参照する。

3.2 CoC の型規則

CoC の型規則は図 1 の通りである。これらの型規則は「上が成り立てば下も成り立つ」と読む。各規則は一般的に $\Gamma \vdash \mathtt{E}_1 : \mathtt{E}_2$ という格好をしており、 A、B、E は式（便宜的に型レベルの式は、A, B を使っているが、意味的には E と同じである）、Γ は型環境である。

(STAR) は、定数 $*$ がより上位の kind である $\Box$ を持つことを表している。$\Box$ はこの型規則のヒエラルキーの中で一番上位のものである。しかし、プログラム中には現れないので $\Box$ に型を付ける規則は存在し

$$\frac{}{\vdash * \ : \ \Box}\ (\text{STAR}) \qquad \frac{}{\vdash \mathtt{N} \ : \ \mathtt{int}}\ (\text{INT}) \qquad \frac{}{\vdash \mathtt{int} \ : \ *}\ (\text{T-INT}) \qquad \frac{\Gamma \vdash \mathtt{A} \ : \ \mathtt{s}}{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{x} \ : \ \mathtt{A}}\ (\text{VAR})$$

$$\frac{\Gamma \vdash \mathtt{E}_1 : (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}) \quad \Gamma \vdash \mathtt{E}_2 : \mathtt{A}}{\Gamma \vdash (\mathtt{E}_1 \ @ \ \mathtt{E}_2) : \mathtt{B}\{\mathtt{x} = \mathtt{E}_2\}}\ (\text{APP}) \qquad \frac{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{E} : \mathtt{B} \quad \Gamma \vdash (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}) : \mathtt{t}}{\Gamma \vdash (\lambda \mathtt{x} : \mathtt{A}.\mathtt{E}) : (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B})}\ (\text{LAM})$$

$$\frac{\Gamma \vdash \mathtt{A} : \mathtt{s} \quad \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{B} : \mathtt{t}}{\Gamma \vdash (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}) : \mathtt{t}}\ (\text{PI})$$

$$\frac{\Gamma \vdash \mathtt{E} : \mathtt{A} \quad \Gamma \vdash \mathtt{B} : \mathtt{s} \quad \mathtt{A} =_\beta \mathtt{B}}{\Gamma \vdash \mathtt{E} : \mathtt{B}}\ (\text{CONV})$$

図 1：CoC の型規則 ($\mathtt{s}, \mathtt{t} \in \{*, \Box\}$)

ない。(INT) は整数が int 型を持つ事を定義している。(VAR) は A が well-typed かつ、環境に x : A が含まれているならば、x の型は A であると読む。

(APP) は関数適用の規則である。この特徴を、以下の依存型を含まない λ 計算と比較して説明する。

$$\frac{\Gamma \vdash \mathtt{E}_1 : \mathtt{A} \to \mathtt{B} \quad \Gamma \vdash \mathtt{E}_2 : \mathtt{A}}{\Gamma \vdash (\mathtt{E}_1 \ @ \ \mathtt{E}_2) : \mathtt{B}}$$

依存型を含まない λ 計算では、型 B の中に項は現れない。よって $\mathtt{E}_1$ の型は A → B （A 型を持つ引数を受け取り、返す値の型が B であるような関数の型）となる。しかし、依存型では渡される引数の項 x が返り値の型である B の中に現れ得る。その為 ($\mathtt{E}_1$@$\mathtt{E}_2$) の型を決定する際には、まず $\mathtt{E}_1$ の型を (Πx : A.B) （受け取る引数 x の型が A であり、返り値の型が B であるような関数の型）であることをチェックする。その上で $\mathtt{E}_2$ の型が $\mathtt{E}_1$ の型と等しいかどうかチェックをし、最終的に結果の型は B の中の x を $\mathtt{E}_2$ に置換する必要がある。

次の (LAM) は関数抽象の規則だが、一方で (Πx : A.B) がどのように生成されるのかを示している。まず (λx : A.E) という式において、 x が型 A を持ち、そこで E の型が B であると仮定する。この時、先ほど挙げた通常の型付 λ 計算ならば、この関数抽象の型は A → B となるが、CoC では x が B の中に現れ得るので (Πx : A.B) と表す。しかし、ここで新たに生成された (Πx : A.B) が型として正当であるためには、A と B が正当な型でなければならない。

そこで次に (PI) の規則が導入される。まず初めに (Πx : A.B) 中の A にきちんと型が付いているかをチェックし、その結果を s とする。次に x : A を環境 Γ に追加し、同じように B のチェックを行う。そしてここで求まった t と s が kind、即ち ∗ または □ になっていれば (Πx : A.B) は B と同じ kind を持つとする。

最後の (CONV) は式 E の型が A かつ、型 B が A と $=_\beta$ ならば E の型を B にできるという規則である。これは依存型が「型の部分に全ての式を書くことができる」という特徴のために必要となる。例えば (APP) の式において $\mathtt{E}_1$ の型が (Πx : ((λy : ∗.y)@int).B) かつ、$\mathtt{E}_2$ の型が int である場合を考える。規則によれば ((λy : ∗.y)@int) と int は等しくなければならない。しかし、この場合前者を実行しない限りふたつの型が等しいことは解らない。(CONV) はこうした型チェック時に必要な実行を行うための規則である。今、先ほどの ((λy : ∗.y)@int) をこの規則の A とする。すると、その実行結果 B は int である。ここで B が $\mathtt{E}_2$ の型 A と等しいことを初めて確認することができる。

通常の CoC の型規則は以上の様になっている。次に、本稿で追加したリストに関する構文に対しての規則をみっつ追加する（図 2)。 (TUPLE) は E が int ならば (tuple E) は型として正しいことを表す。(INIT) は元々項であった $\mathtt{E}_1$ が (tuple $\mathtt{E}_1$) と型の中に入っており、項に依存した tuple 型の生成過程が表されている。(init 2 3) と書けば、長さが 2 で全ての要素の値が 3 、型は (tuple 2) のリストを生成することができる。 (LOOKUP) では (TUPLE) から $\mathtt{E}_3$ が int であることがわかるので、$\mathtt{E}_1$ が int かつ

$$\frac{\vdash \mathtt{E} : \mathtt{int}}{\vdash (\mathtt{tuple}\ \mathtt{E}) : *}\ (\mathrm{TUPLE}) \qquad \frac{\vdash \mathtt{E}_1 : \mathtt{int} \quad \vdash \mathtt{E}_2 : \mathtt{int}}{\vdash (\mathtt{init}\ \mathtt{E}_1\ \mathtt{E}_2) : (\mathtt{tuple}\ \mathtt{E}_1)}\ (\mathrm{INIT})$$

$$\frac{\vdash \mathtt{E}_1 : \mathtt{int} \quad \vdash \mathtt{E}_2 : (\mathtt{tuple}\ \mathtt{E}_3)}{\vdash (\mathtt{lookup}\ \mathtt{E}_1\ \mathtt{E}_2) : \mathtt{int}}\ \text{if } \mathtt{E}_3 \geq \mathtt{E}_1\ \ (\mathrm{LOOKUP})$$

図 2：リスト系構文に対する型規則

$\mathtt{E}_3 \geq \mathtt{E}_1$ を条件とすることで、参照する要素がリスト中に存在するかどうかのチェックが可能である (array bound check)。リストの長さに関してはほとんどの言語においては実行時に動的なチェックをする方法がとられているが、このように依存型を導入することで型にその情報を付加できるようになり、型チェック時に実行時の安全を保障することができる。

3.3 Syntax directed rule

前節では型規則の際に必要な実行があることを述べた。しかし、図 1 及び図 2 の型規則のままでは (CONV) の規則を使って必要な場所で式を実行することはできるが、一見しただけでは一体どこでその実行が必要となるのかがわからない。そこでまずは型チェック中に必要な実行を抽出するために Syntax directed rules を導入する。これは図 1 中の (CONV) を適用すべき部分を型規則の中で明示的に示したもので、以下のように定義される。

$$\frac{}{\vdash * : \Box}\ (\mathrm{STAR}) \qquad \frac{}{\vdash \mathtt{N} : \mathtt{int}}\ (\mathrm{INT}) \qquad \frac{}{\vdash \mathtt{int} : *}\ (\mathrm{T-INT}) \qquad \frac{\Gamma \vdash \mathtt{A} : \mathtt{s}}{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{x} : \mathtt{A}}\ (\mathrm{VAR})$$

$$\frac{\Gamma \vdash \mathtt{E}_1 :\twoheadrightarrow (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}) \quad \Gamma \vdash \mathtt{E}_2 : \mathtt{A}' \quad \mathtt{A} =_\beta \mathtt{A}'}{\Gamma \vdash (\mathtt{E}_1\ @\ \mathtt{E}_2) : \mathtt{B}\ \{\mathtt{x} = \mathtt{E}_2\}}\ (\mathrm{APP}) \qquad \frac{\Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{E} : \mathtt{B} \quad \Gamma \vdash (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}) : \mathtt{t}}{\Gamma \vdash (\lambda \mathtt{x} : \mathtt{A}.\mathtt{E}) : (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B})}\ (\mathrm{LAM})$$

$$\frac{\Gamma \vdash \mathtt{A} :\twoheadrightarrow \mathtt{s} \quad \Gamma, \mathtt{x} : \mathtt{A} \vdash \mathtt{B} :\twoheadrightarrow \mathtt{t}}{\Gamma \vdash (\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}) : \mathtt{t}}\ (\mathrm{PI})$$

$$\frac{\vdash \mathtt{E} :\twoheadrightarrow \mathtt{A} \quad \mathtt{A} =_\beta \mathtt{int}}{\vdash (\mathtt{tuple}\ \mathtt{E}) : *}\ (\mathrm{TUPLE}) \qquad \frac{\vdash \mathtt{E}_1 :\twoheadrightarrow \mathtt{A} \quad \vdash \mathtt{E}_2 :\twoheadrightarrow \mathtt{A}' \quad \mathtt{A} =_\beta \mathtt{A}' =_\beta \mathtt{int}}{\vdash (\mathtt{init}\ \mathtt{E}_1\ \mathtt{E}_2) : (\mathtt{tuple}\ \mathtt{E}_1)}\ (\mathrm{INIT})$$

$$\frac{\vdash \mathtt{E}_1 :\twoheadrightarrow \mathtt{A} \quad \vdash \mathtt{E}_2 :\twoheadrightarrow \mathtt{A}' \quad \mathtt{A} =_\beta \mathtt{int} \quad \mathtt{A} =_\beta (\mathtt{tuple}\ \mathtt{E}_3)}{\vdash (\mathtt{lookup}\ \mathtt{E}_1\ \mathtt{E}_2) : \mathtt{int}}\ \text{if } \mathtt{E}_3 \geq \mathtt{E}_1\ \ (\mathrm{LOOKUP})$$

$$\frac{\lambda \vdash \mathtt{a} : \mathtt{A} \quad \mathtt{A} \twoheadrightarrow^{\mathtt{wh}} \mathtt{B}}{\lambda \vdash \mathtt{a} :\twoheadrightarrow \mathtt{B}}\ (\mathrm{RED})$$

図 3：Syntax directed rule ($\mathtt{s}, \mathtt{t} \in \{*, \Box\}$)

図 1 と図 3 は同じ CoC の型規則を表現したものであり、どちらの規則を使っても同じ結果が得ることができるが [10]、両者の一番の違いは Syntax directed rule の方には規則のあちこちに $\twoheadrightarrow$ が存在することである。(RED) 内の $\mathtt{A} \twoheadrightarrow^{\mathtt{wh}} \mathtt{B}$ とは「$\mathtt{A}$ をその weak head normal form である $\mathtt{B}$ へと簡約する」という意味で、これは規則中の $\Gamma \vdash \mathtt{a} :\twoheadrightarrow \mathtt{B}$ から呼び出され、$\mathtt{a}$ の型を求めた後に行われる。これを踏まえて規則全体を眺めると、型チェック中に実行が必要となるのは (APP) と (PI)、そして本稿で付け加えた (TUPLE)、(INIT)、(LOOKUP) における型のマッチを行う部分であることがわかる。

(APP) は $\mathtt{E}_1$ の型を実行した結果 $(\Pi \mathtt{x} : \mathtt{A}.\mathtt{B})$ の形となり、その上で $\mathtt{A} =_\beta \mathtt{A}'$ ならば $(\mathtt{E}_1 @ \mathtt{E}_2)$ の型は $\mathtt{B}$ とすることを示している。他の規則もほぼ同様に読むことができ、この規則を用いると、型チェック中のどこ

で実行が必要なのかが明示される。

3.4 型チェック中の実行とその問題点

型チェック中の実行は、依存型を含む体系の特徴である。本節では具体的な例を用いていつ実行が起きるのかを見るとともに、実行に関する従来のアプローチの問題点を指摘する。

$$((\lambda \mathtt{x} : (\mathtt{tuple}\ 4).(\mathtt{lookup}\ 2\ \mathtt{x}))@(\mathtt{init}\ ((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@4)\ 7))$$

この式は、「長さ 4 のリストを受け取り、その 2 番目の要素を参照するような関数抽象」に「長さが $((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@4)$ で要素が全て 7 であるようなリスト」を適用するものだ。この式を型チェックしようと思うと、関数部分の引数 x の型は (tuple 4) なので、引数である (init ...) がこれと同じ型を持つ必要がある。ところがこの引数の型は (tuple $((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@4)$) と、tuple 中の式が関数呼び出しになっているので、この式の型チェックをするためには、関数適用を実行する必要がある。

DML のような言語では、型部分に書ける式を一定の形のものに制限し、定理証明系を使って型の等価性を判定している。例えば上の例の場合、$((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@\mathtt{n}) = \mathtt{n}$ という等式を定理証明系があらかじめ知っているならば、その等価性を判定することができる。しかし、このようなアプローチでは、型部分に書くことのできる式は大きく制限せざるを得ない。また、等価性の判定には定理証明系という重い処理を行わなくてはならないばかりか、型部分の実行と通常の実行というふたつの処理を別々に実装してやる必要がある。

しかし CoC における型部分の実行と通常の実行は、全く同じことだ。両者はいずれも単に CoC の式の実行をすることである。そこで本稿では、通常の実行と型部分での実行とを統一的にひとつの仕組みで取り扱う。つまり、今まで「型チェックの一部」として捉えられていたものを「実行」として捉えなおし、型部分の実行は通常の実行がひとつ上のレベルで起きているものと解釈する。

上の例で、式全体の実行をレベル 0 で行っているとする。すると、$((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@4)$ は型レベルに属する式なのでレベル 1 で実行を行うべきである。レベル 1 の実行も通常の実行と同じとみなされるので、実行の前には型チェックが行われる。ここでは $(\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})$ に渡される 4 が int であることが確認される。厳密には、レベル 2 で int が正しい型であることが確認される。この場合 $4 : \mathtt{int}$ であるので、型チェックを通り、レベル 1 での実行が行われる。その結果、4 をレベル 1 で得ることができ、引数 (init ...) は (tuple 4) 型であることが判明する。よってここで初めてレベル 0 の式の型チェックを行うことができるようになる。この場合は、きちんと引数が (tuple 4) であることが解ったので型チェックを通り、晴れてレベル 0 の実行を行うことができる。

このようにすると、定理証明系を使うことなく任意の式を型部分に書くことができる。さらに、型部分で実行される式も自然に型チェックがなされるため、全ての式が実行前に型チェックされるという性質が保たれたまま実行が可能となる。

本研究ではこのような実行体系を実現するために、プログラム中の全ての式、及び型チェック中に構成される証明木の要素に、それが属するレベルを割り当てるための型規則を導入する。そして分解した各ステージごとに、その式に正しい型がついているかどうかを確認してから実行を行う方法を提案する。

4 型チェックと実行のステージ化

この章では、全ての式に自身が属するステージを決定するために作成した実行順序付き型規則を説明する。そしてステージ化されたプログラムが多段的に実行される様子を例を挙げながら示していく。

4.1 実行順序付き型規則

まず、CoC の式をステージ化する為に、図 3 の各規則に実行順序を表す添え字を添加した結果を図 4 に示す。ここで、各式の添字はその式が実行可能となるレベルの番号を表しており、数字を大きければ大きいほど先に実行可能であるとする。

$$\frac{}{\Gamma \vdash *^{n} : \Box^{n+1}}\ (\text{STAR}) \qquad \frac{}{\Gamma \vdash \mathtt{N}^{n} : \mathtt{int}^{n+1}}\ (\text{INT})$$

$$\frac{}{\Gamma \vdash \mathtt{int}^{n} : *^{n+1}}\ (\text{T-INT}) \qquad \frac{\Gamma \vdash \mathtt{A}^{n+1} : \mathtt{s}^{n+2}}{\Gamma, \mathtt{x}^{m} : \mathtt{A}^{m+1} \vdash \mathtt{x}^{n} : \mathtt{A}^{n+1}}\ (\mathtt{m} \geq \mathtt{n})\ (\text{VAR})$$

$$\frac{\Gamma \vdash \mathtt{E}_1^{n} :\twoheadrightarrow^{n+1} (\Pi \mathtt{x}^{n} : \mathtt{A}^{n+1}.\mathtt{B}^{n+d})^{n+1} \quad \Gamma \vdash \mathtt{E}_2^{n} :\twoheadrightarrow^{n+1} \mathtt{A}'^{n+1} \quad (\mathtt{A} = \mathtt{A}')^{n+1}}{\Gamma \vdash (\mathtt{E}_1^{n}\ @^{n}\ \mathtt{E}_2^{n})^{n+d-1} : \mathtt{B}^{n+d}\{\mathtt{x} = \mathtt{E}_2\}^{n+1}}\ (\text{APP})$$

$$\frac{\Gamma, \mathtt{x}^{n} : \mathtt{A}^{n+1} \vdash \mathtt{E}^{n+d-1} : \mathtt{B}^{n+d} \quad \Gamma \vdash (\Pi \mathtt{x}^{n} : \mathtt{A}^{n+1}.\mathtt{B}^{n+d})^{n+1} : \mathtt{s}^{n+2}}{\Gamma \vdash (\lambda \mathtt{x}^{n} : \mathtt{A}^{n+1}.\mathtt{E}^{n+d-1})^{n} : (\Pi \mathtt{x}^{n} : \mathtt{A}^{n+1}.\mathtt{B}^{n+d})^{n+1}}\ (\text{LAM})$$

$$\frac{\Gamma \vdash \mathtt{A}^{n+1} :\twoheadrightarrow^{n+2} \mathtt{s}^{n+2} \quad \Gamma, \mathtt{x}^{n} : \mathtt{A}^{n+1} \vdash \mathtt{B}^{n+d} :\twoheadrightarrow^{n+d+1} \mathtt{t}^{n+d+1}}{\Gamma \vdash (\Pi \mathtt{x}^{n} : \mathtt{A}^{n+1}.\mathtt{B}^{n+d})^{n+1} : \mathtt{t}^{n+2}}\ (\text{PI})$$

$$\frac{\Gamma \vdash \mathtt{E}^{n} :\twoheadrightarrow^{n+1} \mathtt{A}^{n+1} \quad (\mathtt{A}=\mathtt{int})^{n+1}}{\Gamma \vdash (\mathtt{tuple}\ \mathtt{E}^{n})^{n} : *^{n+1}}\ (\text{TUPLE}) \qquad \frac{\Gamma \vdash \mathtt{E}_1^{n+1} :\twoheadrightarrow \mathtt{A}^{n+2} \quad \Gamma \vdash \mathtt{E}_2^{n} :\twoheadrightarrow \mathtt{A}'^{n+1} \quad (\mathtt{A}=\mathtt{A}'=\mathtt{int})^{n+1}}{\Gamma \vdash (\mathtt{init}\ \mathtt{E}_1^{n+1}\ \mathtt{E}_2^{n})^{n} : (\mathtt{tuple}\ \mathtt{E}_1^{n+1})^{n+1}}\ (\text{INIT})$$

$$\frac{\Gamma \vdash \mathtt{E}_1^{n+1} :\twoheadrightarrow^{n+2} \mathtt{A}^{n+2} \quad \Gamma \vdash \mathtt{E}_2^{n} :\twoheadrightarrow^{n+1} \mathtt{A}'^{n+1} \quad (\mathtt{A}=\mathtt{int})^{n+2} \quad (\mathtt{A}'=(\mathtt{tuple}\ \mathtt{E}_3))^{n+1} \quad (\mathtt{E}_3 \geq \mathtt{E}_1)^{n+1}}{\Gamma \vdash (\mathtt{lookup}\ \mathtt{E}_1^{n+1}\ \mathtt{E}_2^{n})^{n} : \mathtt{int}^{n+1}}\ (\text{LOOKUP})$$

$$\frac{\Gamma \vdash \mathtt{a}^{n} : \mathtt{A}^{n+1} \quad (\mathtt{A} \twoheadrightarrow^{\mathtt{wh}} \mathtt{B})^{n+1}}{\Gamma \vdash \mathtt{a}^{n} :\twoheadrightarrow^{n+1} \mathtt{B}^{n+1}}\ (\text{RED})$$

図 4：実行順序付き型付け規則 ($\mathtt{d} = 0 \mid 1$ 、 $\mathtt{s}, \mathtt{t} \in \{*, \Box\}$)

全ての規則は基本的に、項に n が付いているならば、その型にはそれよりひとつ上のレベルを表す $\mathtt{n}+1$ が付いている。これはレベル n でその項を実行するためには、その型にあたる式はひとつ前の $\mathtt{n}+1$ レベルまでに型チェック及び実行が終わっていなければならないことを意味している。もっと端的に言うならば、$\mathtt{n}+1$ の操作は n の項の型チェックとなっている。(STAR)、(T-INT)、(INT) は全てその通りに規則が作られている。しかし (VAR) は少し特殊で、不等式 ($\mathtt{m} \geq \mathtt{n}$) が 「x がレベル m で手元にあるならば、m より後のレベル n で使ってもよい」ことを示している。これは部分評価の lift 操作 [9] に相当する。なお、通常 n の初期値は 0 とする。

しかし、全ての式に対してこの方式でレベル付け、及び実行を行うと「関数抽象の body のレベルが落ちる場合（4.2 節)」、「実行を行わなければ型マッチができない場合（4.3 節)」、「関数抽象の引数の型が、実行後でなければ ($\Pi \mathtt{x} : \mathtt{A}.\mathtt{B}$) だとわからない場合（4.4 節)」、「自由変数を含む式を実行しなければならない場合（4.5 節)」の 4 つの問題が発生する。以下、各規則の説明と並行してそれぞれの問題に該当する例とその解消法について説明する。

4.2 関数抽象のレベル付け

通常、関数抽象の body 部分は関数抽象自身のステージと同じステージで実行できる。例として ($\lambda \mathtt{x} : \mathtt{int}.\mathtt{x}$) にレベルを付ける事を考える。まずはじめにこの式全体をレベル n で実行すると仮定する。すると、x は

やはり同じレベル $\mathtt{n}$ で値が渡されるものと考えられる。するとこの関数の body 部分である変数 $\mathtt{x}$ も同じレベル内で実行することができるのでそのレベルは $\mathtt{n}$ となる。よって、この関数抽象自体もレベル $\mathtt{n}$ に属することが解る。従って、この式全体には直感的に $(\lambda \mathtt{x}^{\mathtt{n}} : \mathtt{int}^{\mathtt{n}+1}.\mathtt{x}^{\mathtt{n}})^{\mathtt{n}}$ とレベルを付けることができるが、これは (LAM) の規則内の $\mathtt{d}$ に 1 を入れた規則を用いてつけた結果と同じものになっている。

ところが、関数抽象の body 部分のレベルがひとつ下がる場合が存在する。例として System F における polymorphic identity を考えてみよう。

$$(\lambda\alpha : *.(\lambda \mathtt{x} : \alpha.\mathtt{x}))$$

この式にレベルを付ける時、まずはじめに式全体のレベルを $\mathtt{n}$ と定義すると、先ほどと同様に α はレベル $\mathtt{n}$、 α の型の $*$ は $\mathtt{n}+1$ である。次に内側のλ式 $(\lambda \mathtt{x} : \alpha.\mathtt{x})$ を見ると、今度はレベル $\mathtt{n}$ で来るはずの α が $\mathtt{x}$ の型として使われている。よって $\mathtt{x}$ にはそれよりひとつ後のレベルが付けられるので、$\mathtt{x}^{\mathtt{n}-1} : \alpha^{\mathtt{n}}$ となる。するとこのλ式全体は、$(\lambda \mathtt{x}^{\mathtt{n}-1} : \alpha^{\mathtt{n}}.\mathtt{x}^{\mathtt{n}-1})^{\mathtt{n}-1}$ となり、外側のλ式が返すもの（つまり内側のλ式）は式全体のレベル $\mathtt{n}$ よりひとつ低いレベルとなる。このように body 部分のレベルがひとつ下がることがあるのに対応するために、図 4 の型規則では $\mathtt{d}$ を導入している。この $\mathtt{d}$ は通常は 1 だが、body 部分のレベルが下がる場合に 0 となる。例えば、先の式を図 4 の規則に適用すると以下のようになる。

$$\dfrac{\dfrac{\alpha^{\mathtt{n}}:*^{\mathtt{n}+1},\mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}} \vdash \underline{\mathtt{x}^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-2}:\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1}}}{\alpha^{\mathtt{n}}:*^{\mathtt{n}+1} \vdash (\Pi \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1})^{\mathtt{n}+\mathtt{d}}:*^{\mathtt{n}+\mathtt{d}+1}} \quad \dfrac{\alpha^{\mathtt{n}}:*^{\mathtt{n}+1} \vdash \underline{\alpha^{\mathtt{n}+\mathtt{d}}:*^{\mathtt{n}+\mathtt{d}+1}} \quad \alpha^{\mathtt{n}}:*^{\mathtt{n}+1},\mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}} \vdash \underline{\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1}:*^{\mathtt{n}+\mathtt{d}+\mathtt{d}'}}}{\alpha^{\mathtt{n}}:*^{\mathtt{n}+1} \vdash (\Pi \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1})^{\mathtt{n}+\mathtt{d}}:*^{\mathtt{n}+\mathtt{d}+1}}}{\dfrac{\alpha^{\mathtt{n}}:*^{\mathtt{n}+1} \vdash (\lambda \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\mathtt{x}^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-2})^{\mathtt{n}+\mathtt{d}-1}:(\Pi \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1})^{\mathtt{n}+\mathtt{d}} \quad \vdash (\Pi\alpha^{\mathtt{n}}:*^{\mathtt{n}+1}.(\Pi \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1})^{\mathtt{n}+\mathtt{d}})^{\mathtt{n}+1}:*^{\mathtt{n}+2}}{\vdash ((\lambda\alpha^{\mathtt{n}}:*^{\mathtt{n}+1}.(\lambda \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\mathtt{x}^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-2})^{\mathtt{n}+\mathtt{d}-1})^{\mathtt{n}}:(\Pi\alpha^{\mathtt{n}}:*^{\mathtt{n}+1}.(\Pi \mathtt{x}^{\mathtt{n}+\mathtt{d}-1}:\alpha^{\mathtt{n}+\mathtt{d}}.\alpha^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-1})^{\mathtt{n}+\mathtt{d}})^{\mathtt{n}+1}}$$

ここで先ほどレベルが落ちていた内側のλ式を見ると、$(\lambda \mathtt{x}^{\mathtt{n}+\mathtt{d}-1} : \alpha^{\mathtt{n}+\mathtt{d}}.\mathtt{x}^{\mathtt{n}+\mathtt{d}+\mathtt{d}'-2})^{\mathtt{n}+\mathtt{d}-1}$ となっている。ここで $\mathtt{d}$ と $\mathtt{d}'$ の最適解を決定するために、(VAR) の規則の不等式を使用する。この例では下線を引いた箇所で (VAR) を適用しており、その各部分で規則通りに方程式を作ると次のようになる。

$$\begin{aligned} \mathtt{n}+\mathtt{d}-1 &\geq \mathtt{n}+\mathtt{d}+\mathtt{d}'-2 \\ \mathtt{n} &\geq \mathtt{n}+\mathtt{d} \\ \mathtt{n} &\geq \mathtt{n}+\mathtt{d}+\mathtt{d}'-1 \end{aligned}$$

これらの不等式全てを満たし、さらに式がなるべく早い段階で実行できるように（$\mathtt{d}$ 及び $\mathtt{d}'$ が大きくなるように）$\mathtt{d}$ の値を求めると、$\mathtt{d}=0,\ \mathtt{d}'=1$ となる。これを元の式に適用すると $(\lambda\alpha^{\mathtt{n}} : *^{\mathtt{n}+1}.(\lambda \mathtt{x}^{\mathtt{n}-1} : \alpha^{\mathtt{n}}.\mathtt{x}^{\mathtt{n}-1})^{\mathtt{n}-1})^{\mathtt{n}}$ となる。内側の α のレベルを $\mathtt{n}$ に抑えたことで、$\mathtt{x}$ も連動して $\mathtt{n}-1$ となり、確かに内側の関数抽象全体のレベルが抑えることができる。

4.3 (APP) と実行

依存型の入った体系では型チェックの際に式の実行が必要になってくる。その実行のレベル付けを行うのが (APP) など $\twoheadrightarrow$ の入った規則である。ここでは、3.4 節でマッチングができなかった式を例にとって実行のレベル付け及びそれを含む型付け規則について説明する。

3.4 節にて挙げた $((\lambda \mathtt{x} : (\mathtt{tuple}\ 4).(\mathtt{lookup}\ 2\ \mathtt{x}))@(\mathtt{init}\ ((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@4)\ 7))$ は、$((\lambda \mathtt{x} : \mathtt{int}.\mathtt{x})@4)$ を実行しなければ型のマッチングができなかった。よって元の式のレベルを 0 とすると、この部分はその型チェックと同じレベル 1 で実行しなければならない。また、この式を 1 で実行するためにはこの式の型チェックにはレベル 2 以上がついている必要がある。よってこの型は直感的には $((\lambda \mathtt{x}^{1} : \mathtt{int}^{2}.\mathtt{x}^{1})^{1}@4^{1})^{1}$ とレベルがつけられるはずである。

次に、この予想をふまえて図 4 の (APP) の規則をこの式に適用した結果を考える。以下は $\mathtt{d}$ の値は全て決定済みかつ $\mathtt{n}=0$ として規則を適用した結果である。

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{x}^1{:}\mathtt{int}^2\vdash\mathtt{x}^1{:}\mathtt{int}^2}\quad\dfrac{\dfrac{}{\vdash\mathtt{int}^2{:}*^3}\quad\dfrac{}{\mathtt{x}^1{:}\mathtt{int}^2\vdash\mathtt{int}^2{:}*^3}}{(\Pi\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{int}^2){:}*^3}}{\vdash(\lambda\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{x}^1)^1{:}\twoheadrightarrow^2(\Pi\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{int}^2)^2}\quad\dfrac{}{\vdash 4^1{:}\twoheadrightarrow^2\mathtt{int}^2}}{\vdash((\lambda\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{x}^1)^1@^1 4^1)^1{:}\twoheadrightarrow^2\mathtt{int}^2}\quad\dfrac{}{\vdash 7^0{:}\twoheadrightarrow^1\mathtt{int}^1}}{\vdash(\mathtt{init}\ ((\lambda\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{x}^1)^1@^1 4^1)^1\ 7^0)^0{:}(\mathtt{tuple}\ ((\lambda\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{x}^1)^1@^1 4^1)^1)^1\quad((\mathtt{tuple}(\lambda\mathtt{x}{:}\mathtt{int}.\mathtt{x})@4))\twoheadrightarrow\mathtt{A}'){}^1}}{\vdash((\lambda\mathtt{x}^0{:}(\mathtt{tuple}\ 4)^1.(\mathtt{lookup}\ 2^1\ \mathtt{x}^0)^0)^0{:}\twoheadrightarrow^1(\Pi\mathtt{x}^0{:}(\mathtt{tuple}\ 4)^1.\mathtt{int}^1)^1\qquad\vdash(\mathtt{init}\ ((\lambda\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{x}^1)^1@^1 4^1)^1\ 7^0)^0{:}\twoheadrightarrow^1\mathtt{A}'^1\qquad((\mathtt{tuple}\ 4){=}\mathtt{A}')^1}$$
$$\vdash((\lambda\mathtt{x}^0{:}(\mathtt{tuple}\ 4)^1.(\mathtt{lookup}\ 2^1\ \mathtt{x}^0)^0)^0@^0(\mathtt{init}\ ((\lambda\mathtt{x}^1{:}\mathtt{int}^2.\mathtt{x}^1)^1@4^1)^1\ 7^0)^0)^0{:}\mathtt{int}^1$$

(APP) の規則では、Syntax directed rule にあった $\mathtt{E}_1 :\twoheadrightarrow \mathtt{A}$ に加えて、$\mathtt{E}_2$ にも $\twoheadrightarrow$ を加えた。これによって $\mathtt{E}_1$ と $\mathtt{E}_2$ の型は両方とも β リダクションされるので、型マッチは $=_\beta$ から $=$ と変更した。そしてこの $\twoheadrightarrow$ にも式と同じように添字を付けた。この添字がその式が実行を行わなければならないレベルを表している。

ここで先ほどの予想と規則を適用した結果を比較する。すると、$(\mathtt{init}\ ...) :\twoheadrightarrow^1 \mathtt{A}'^1$ となり、確かに $(\mathtt{tuple}\ ((\lambda\mathtt{x} : \mathtt{int}.\mathtt{x})@4))$ の実行はレベル 1 となる。ここで実行に関する規則 (RED) を適用すると、さらにその式の型チェックにもレベルが割り当てられる。ここでは下から 4 段目以上が (RED) の適用結果となっているが、$((\lambda\mathtt{x} : \mathtt{int}.\mathtt{x})@4)$ の型チェックには必ず 2 以上のレベルが割りふられているのが解る。よって、レベル 2 までにこの式の型チェックは完了するので、レベル 1 では実行時チェックなしで式を実行することができる。そしてその結果、$\mathtt{A}' = (\mathtt{tuple}\ 4)$ となり、やはりレベル 1 において下から 2 段目右端で型マッチを行うことができる。さらにこれが通ると、0 の実行に対する型チェックが終了し、最終的に元の式も実行時チェックをせずに実行できる。

また、(TUPLE) や (INIT) の規則も実行を含んでいるが、これらも基本的に (APP) の場合と同じである。(TUPLE) は E の型をレベル $\mathtt{n}+1$ で実行し、その結果が int であることをやはり $\mathtt{n}+1$ で確認する。(INIT) もほぼ同様だが、式中の $\mathtt{E}_1$ はその型の中でも使われるために全体と $\mathtt{E}_2$ より 1 つ大きい数字が付けることにする。(LOOKUP) の場合は $\mathtt{E}_2$ の型と大小関係の比較をするために、やはり $\mathtt{E}_1$ には式全体と $\mathtt{E}_2$ より 1 つ高いレベルをつけている。

以上が、型チェック時に必要な実行とそれに対する型チェックを検出し、それぞれが実行可能な番号をつける手順である。この例では関数抽象の引数の型がはじめから $(\Pi\mathtt{x} : \mathtt{A}.\mathtt{B})$ の形になっているので、型マッチ部分以外の証明木は全てレベル付けの時に作成できる。あとはレベルが高い方から証明木中に現れる実行を順に行い、その結果がわかり次第残っていた型マッチを行っていけばよい。そして最終的にエラーを起こすことなく証明木が完成すれば、元々の式はチェックなしでの実行をすることができる。

4.4 部分的に木を作らなければならない場合

今までは初めからほぼ証明木が構成できる例を扱ってきたが、次に関数抽象の引数の型が関数適用であるために、実行しなければ $(\Pi\mathtt{x} : \mathtt{A}.\mathtt{B})$ だとわからない場合の例を挙げる。このような場合には後の実行中に部分的に証明木を作成しなければならない。

具体的には以下の例を考える。

$$(\lambda\mathtt{f} : ((\lambda\mathtt{x} : *.\mathtt{x})@(\Pi\mathtt{y} : \mathtt{int}.\mathtt{int})).(\mathtt{f}@2))$$

これは引数の型が $((\lambda\mathtt{x} : *.\mathtt{x})@(\Pi\mathtt{y} : \mathtt{int}.\mathtt{int}))$ という関数適用であるような f を受け取り、それに 2 を適用するような関数抽象である。この式に図 4 の型規則を適用すると以下の通りになる。

$$\dfrac{\dfrac{\dfrac{}{\Gamma\vdash\mathtt{f}^0{:}\twoheadrightarrow^1(\Pi?^0{:}\mathtt{A}^1.\mathtt{B}^1)^1}\quad\dfrac{}{\Gamma\vdash 2^0{:}\twoheadrightarrow^1\mathtt{int}^1}\quad(\mathtt{A}{=}\mathtt{int})^1}{\Gamma\vdash(\mathtt{f}^0@2^0)^0{:}\mathtt{B}^1\{?{=}2\}^1}\qquad\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash *^2{:}\twoheadrightarrow^3\square^3}\quad\dfrac{}{\mathtt{x}{:}*\vdash *^2{:}\twoheadrightarrow^3\square^3}}{\mathtt{x}^1{:}*^2\vdash\mathtt{x}^1{:}*^2\qquad\vdash(\Pi\mathtt{x}^1{:}*^2.*^2)^2{:}\square^3}}{\vdash(\lambda\mathtt{x}^1{:}*^2.\mathtt{x}^1)^1{:}\twoheadrightarrow^2(\Pi\mathtt{x}^1{:}*^2.*^2)^2}\qquad\dfrac{\vdash\mathtt{int}^1{:}\twoheadrightarrow^2*^2\quad\vdash\mathtt{int}^1{:}\twoheadrightarrow^2*^2}{(\Pi\mathtt{y}^0{:}\mathtt{int}^1.\mathtt{int}^1)^1{:}\twoheadrightarrow^2*^2}}{\vdash((\lambda\mathtt{x}^1{:}*^2.\mathtt{x}^1)^1@(\Pi\mathtt{y}^1{:}\mathtt{int}^2.\mathtt{int}^1)^1)^1{:}\twoheadrightarrow^2*^2\qquad\Gamma\vdash\mathtt{B}^1\{?{=}2\}^1{:}\twoheadrightarrow^2\mathtt{B}'^2}}{\vdash(\Pi\mathtt{f}^0{:}((\lambda\mathtt{x}^1{:}*^2.\mathtt{x}^1)^1@(\Pi\mathtt{y}^1{:}\mathtt{int}^2.\mathtt{int}^1)^1)^1.\mathtt{B}^1\{?{=}2\}^1)^2{:}\mathtt{B}'^2}$$
$$\vdash(\lambda\mathtt{f}^0{:}((\lambda\mathtt{x}{:}*.\mathtt{x})^1@(\Pi\mathtt{y}{:}\mathtt{int}.\mathtt{int})^1)^1.(\mathtt{f}^0@2^0)^0){:}(\Pi\mathtt{f}^0{:}((\lambda\mathtt{x}^1{:}*^2.\mathtt{x}^1)^1@(\Pi\mathtt{y}^1{:}\mathtt{int}^2.\mathtt{int}^1)^1)^1.\mathtt{B}^1\{?{=}2\}^1)^1$$

まず、この式全体は関数抽象なので、body 部分の木を作成するために $\mathtt{f}^0 : ((\lambda \mathtt{x} : *.\mathtt{x})@(\Pi \mathtt{y} : \mathtt{int}.\mathtt{int}))^1$ を環境に入れようと試みる。しかし、このままではこの関数適用全体に対するレベルは付けられるものの、内側の式のレベルが全く不明のままである。そこで、関数抽象の body 部分を根とする証明木を作成する前に、$(\Pi \mathtt{f} : ((\lambda \mathtt{x} : *.\mathtt{x})@(\Pi \mathtt{y} : \mathtt{int}.\mathtt{int}))^1 \ ...)$ を根とする証明木の左側、即ち、$((\lambda \mathtt{x} : *.\mathtt{x})@(\Pi \mathtt{y} : \mathtt{int}.\mathtt{int}))^1$ に (APP) を用いてレベル付けを行う。その結果求まったレベルの付いた式を環境に入れ（これを Γ とする）、次に (f@2) へと進む。

(f@2) は関数適用なので図 4 の (APP) を用いて型とレベルを付けるが、f の型は先ほど Γ に入れた関数適用の式である。これはもちろん実行すれば $(\Pi \mathtt{y} : \mathtt{int}.\mathtt{int})$ であるが、証明木を生成している時点では関数適用のままであり、$(\Pi \mathtt{x} : \mathtt{A}.\mathtt{B})$ の形にはなっていない。つまり、今の時点では規則中の A も (f@2) の型である B も、それがどんな式なのか全くわからない。よって A と int の型マッチができないと同時に、下から二段目、右側の $(\Pi \mathtt{f} : (...).\mathtt{B}\{? = 2\})$ において $\mathtt{B}\{? = 2\}$ から先の証明木を作ることも、f の型を実行し、B がどんな型であるのか判明するまでは行うことができない。実際に B にレベル、及び型をつける為の証明木は、レベル 1 で $((\lambda \mathtt{x} : *.\mathtt{x})@(\Pi \mathtt{y} : \mathtt{int}.\mathtt{int}))$ が実行され、B に int が入ることがわかった後にはじめて作ることができる。従って、この様な場合にはレベル 1 での実行後、f の型が $(\Pi \mathtt{y} : \mathtt{int}.\mathtt{int})$ だと判明した後に、B の型チェックを行う証明木、—つまり、

$$\Gamma \vdash \mathtt{int}^1 : *^2$$

を根とする証明木を独立して生成する。そしてできあがった後に、この部分木を元の証明木の $\mathtt{B}^1 : \mathtt{B}'^2$ と置き換えれば、証明木の空白部分が完全に埋まる。後はこれまでと同じように元の証明木についてレベル 0 の実行→チェックを行えば、元の式の型チェックは全て完了し、安全に実行することができる。

4.5 実行と型マッチの遅延

この節ではレベル内で出来ないために実行と型マッチを遅延させる必要があるケースについて説明する。図 4 の型規則の、(APP)、(TUPLE)、(INIT)、(LOOKUP) は規則内に型マッチングを含んでいる。ここで比較すべき式は型レベルでの値、即ち int、$*$、$\Box$、(tuple E)、$(\Pi \mathtt{x} : \mathtt{A}.\mathtt{B})$ のいずれかだ。関数適用は直前までの実行で既に他の式に簡約済みであり、関数抽象を式の型とすることは型規則によって弾かれる。はじめのみっつ同士は、普通に比較を行って問題ない。しかし、残りのふたつは型の中に別の式を含んでいるために簡単に型マッチを行うことはできない。以下にその例のひとつを示す。加算、乗算は定義されているものと仮定する。

$$((\lambda \mathtt{p}^0 : (\Pi \mathtt{n}^0 : \mathtt{int}^1.((\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0@(\mathtt{tuple}\ (\mathtt{n}+3)^0)^0)^0)^1.(\mathtt{p}^0@3^0)^{-1})^0@(\lambda \mathtt{n}^0 : \mathtt{int}^1.(\mathtt{init}\ (\mathtt{n}*2)^0\ 7^{-1})^{-1})^0)^0$$

これは型が $(\Pi \mathtt{n} : \mathtt{int}.((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple}\ (\mathtt{n}+3))))$ である引数 p を受け取って、(p@3) を返す関数に、int 型の引数 n を受け取り、全ての要素の値が 7 かつ長さが $(\mathtt{n}*2)$ のリストを返すような関数を渡す、全体としては関数適用の式である。よって全体の関数適用を行うひとつ上のレベルで、引数 p の型 $(\Pi \mathtt{n} : \mathtt{int}.(...))$ と $(\lambda \mathtt{n} : \mathtt{int}.(...))$ の型である $(\Pi \mathtt{n} : \mathtt{int}.(\mathtt{tuple}\ (\mathtt{n}*2)))$ との比較を行わなければならない。この式は 4.2 節で扱った、関数抽象の body 部分のレベルが全体のレベルよりひとつ落ちているものの型である。ここで、n の型である int 同士が等しいことはすぐにわかる。ところが Π の body 部分である $((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple}\ (\mathtt{n}+3)))$ と $(\mathtt{tuple}\ (\mathtt{n}*2))$ は、実際に body 部分で適用された関数抽象に n として 3 が渡されない限り、両者の比較を行うことができない。この現象は型チェック中に実行を行わなければならない式の中に自由変数（ここでは n）が存在し、その値が実際にわかるまでは $(\mathtt{n}+3)$ と $(\mathtt{n}*2)$ の計算を行うことができず、型を比較することができないからである。

では、いつならできるのか。それは n の具体的な値がわかった後である。つまりこの型を持つ関数抽象の式に n として実際に何かが適用された瞬間、$((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple}\ (\mathtt{n}+3)))$ と $(\mathtt{tuple}\ (\mathtt{n}*2))$ の中の n をその値に置換した後ならば、このふたつの式の未知数の値がわかるので実際に実行し、比較することが

可能である。このような場合には、本来実行と型マッチを行わなければならないレベルにおいてはこの部分の型マッチは通るものと仮定して先に進み、代わりに以下のような注釈を n の値が判明するレベルに追加する。

$$\lambda \mathtt{n} : \mathtt{int}.\mathit{if}\ (((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple}\ (\mathtt{n}+3))) = (\mathtt{tuple}\ (\mathtt{n}*2)))\ \mathit{then}\ \mathtt{ok}\ \mathit{else}\ \mathtt{type\ error}$$

これは整数型の n を受け取った後に、もし $((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple}\ (\mathtt{n}+3)))$ と $(\mathtt{tuple}\ (\mathtt{n}*2))$ の実行結果が等しければ型が正しく付いており、そうでなければ型エラーであると読む。これを証明木の外部に持ち、実際に n の値が解った時に行えば、チェックを遅延することができる。つまりこの例の場合、左の関数抽象の body 部分、(p@3) が実行される直前にこの注釈のチェックを行う。$\mathtt{n} = 3$ ならば、未実行の式はどちらも (tuple 6) となるので、この式には正しく型がついていることがわかる。そうでないなら型エラーである。定理証明系を使ったアプローチでは、一般には $\mathtt{n}+1$ と $\mathtt{n}*2$ は等しくないので、常に型エラーとせざるをえない。

型チェックの遅延は、実行時チェックになってしまっていると思うかもしれないが、そうではない。(tuple 6) の型チェックはレベル 0 に遅延されたが、それは body 部分である (p@3) のレベルが式全体よりひとつ下がった -1 だからである。即ち、(p@3) の型チェックは、正しく実行よりひとつ手前のレベルで行われている。これは、λ式の body 部分のレベルが下がっているときにのみ起こる現象であり、型チェックを遅延させることで、正しいレベルで型チェックを行うことができるようになっている。

また、(APP) の他に (TUPLE)、(INIT)、(LOOKUP) にも型の実行とマッチングが存在するので似たような問題が発生する。例として以下の式を考える。

$$(\lambda \mathtt{n} : \mathtt{int}.(\lambda \mathtt{x} : (\mathtt{tuple\ n}).(\mathtt{lookup\ 2\ x})))$$

これは関数抽象の式だが、はじめに int 型の n を受け取り、(tuple n) 型の引数 x を受け取って、その 2 番目の値を返すような関数抽象を返す。問題は (lookup 2 x) の型チェックで発生する。ここで、2 の型は int、x の型は (tuple n) である。従って次に array bound check として $(\mathtt{n} \geq 2)$ をしたいが、n の実際の値がわからないため、やはり先ほどの場合と同じく証明木だけではこの式の型チェックを完全に行うことはできない。そこでやはり先ほどと同様に、以下のような注釈を生成する。

$$\lambda \mathtt{n} : \mathtt{int}.\mathit{if}\ (\mathtt{n} \geq 2)\ \mathit{then}\ \mathtt{ok}\ \mathit{else}\ \mathtt{type\ error}$$

ここでは実際に n の値がくることはないので、この注釈は結果として最後までチェックされることはない。しかし、注釈としてこの情報を残しておけば、後からこの式全体に n として 4 を適用するときに、関数適用が終わった瞬間にその注釈を参照することで、残っていたチェック — $(4 \geq 2)$ を行うことができる。結果、(LOOKUP) の規則に埋め込まれた array bound check を (lookup 2 4) を実行する前に機能させることができる。よって、全ての式は型チェックを行ってから実行をするという原則をきちんと保つことができている。

5 実装

4 章では CoC の式に図 4 で示した実行順序付きの型規則を適用すると、その実行を複数のステージに分解できることを示した。そこで本章ではこの規則を実装する。特に、その際にやはり問題となる「部分的に証明木を作らなければならない場合（5.2 節）」と「実行と型チェックを遅延させなければならない場合（5.3 節）」を解決するにはどのように実装すればよいのかを詳しく示していく。

5.1 型テーブルの作成

プログラムは入力されると、字句解析・構文解析を行って構文木に変換される。そして、実行順序を付けながら証明木を構成していく。この時構文木に直接型を書き込むのではなく、証明木に登場する全ての

型には index をふる。そして、証明木の型とレベル情報をまとめたテーブルを作成し、実際の型はそこに記述する。以下、この方法で実際にプログラムの解析を行ってテーブルを作成した例を示す。

$$(\lambda \mathtt{n} : \mathtt{int}.(\lambda \mathtt{f} : ((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple\ n})).(\mathtt{lookup\ 2\ f})))$$

この式は関数抽象で、int 型の引数 n を受け取り、その上で $((\lambda \mathtt{x} : *.\mathtt{x})@(\mathtt{tuple\ n}))$ 型の引数 f を受け取って、その 2 番目の値を参照した値を返すような関数抽象を返す。この式に対して、型に index $\mathtt{T_i}$ をふった証明木と型参照テーブルは以下の様になる。

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash 2^0 : \twoheadrightarrow^1 \mathtt{T}^1_{13} \quad \Gamma \vdash \mathtt{f}^{-1} : \twoheadrightarrow^0 \mathtt{T}^0_{14}}{\Gamma \vdash (\mathtt{lookup\ 2^0\ f^{-1}})^{-1} : \mathtt{T}^0_{15}} \quad \dfrac{\dfrac{\dfrac{\vdash *^1 : \twoheadrightarrow^2 \mathtt{T}^2_3 \quad \mathtt{x} : \mathtt{T}^1_4 \vdash \mathtt{T}^1_5 : \twoheadrightarrow^2 \mathtt{T}^2_6}{\mathtt{x}^0 : \mathtt{T}^1_4 \vdash \mathtt{x}^0 : \mathtt{T}^1_5 \quad \vdash (\Pi \mathtt{x}^0 : \mathtt{T}^1_4.\mathtt{T}^1_5)^1 : \mathtt{T}^2_7}}{\vdash (\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0 : \twoheadrightarrow^1 \mathtt{T}^1_8} \quad \dfrac{\Gamma \vdash \mathtt{n}^0 : \twoheadrightarrow^1 \mathtt{T}^1_9}{\Gamma \vdash (\mathtt{tuple\ n}^0)^0 : \twoheadrightarrow^1 \mathtt{T}^1_{10}}}{\vdash ((\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0 @ (\mathtt{tuple\ n})^0)^0 : \twoheadrightarrow^1 \mathtt{T}^1_{11} \quad \Gamma \vdash \mathtt{T}^0_{15} : \twoheadrightarrow^1 \mathtt{T}^1_{16}}}{\vdash (\Pi \mathtt{f}^{-1} : \mathtt{T}^0_{12}.\mathtt{T}^0_{15})^0 : \mathtt{T}^1_{17}} \quad \dfrac{\vdash \mathtt{int}^1 : \twoheadrightarrow^2 \mathtt{T}^2_1 \quad \mathtt{n}^0 : \mathtt{T}^1_2 \vdash \mathtt{T}^0_{18} \twoheadrightarrow^1 \mathtt{T}^1_{19}}{\vdash (\Pi \mathtt{n}^0 : \mathtt{T}^1_2.\mathtt{T}^0_{18})^1 : \mathtt{T}^2_{20}}}{\mathtt{n}^0 : \mathtt{T}^1_2 \vdash (\lambda \mathtt{f}^{-1} : ((\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0 @ (\mathtt{tuple\ n}^0)^0)^0.(\mathtt{lookup\ 2^0\ f^{-1}})^{-1})^{-1} : \mathtt{T}^0_{18}}}{\vdash (\lambda \mathtt{n}^0 : \mathtt{int}^1.(\lambda \mathtt{f}^{-1} : ((\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0 @ (\mathtt{tuple\ n}^0)^0)^0.(\mathtt{lookup\ 2^0\ f^{-1}})^{-1})^{-1})^0 : \mathtt{T}^1_{21}}$$

index	n	type
$\mathtt{T}_1$	2	$*^2$
$\mathtt{T}_2$	1	$\mathtt{int}^1$
$\mathtt{T}_3$	2	$\Box^2$
$\mathtt{T}_4$	1	$*^1$
$\mathtt{T}_5$	2	$= \mathtt{T}_4$
$\mathtt{T}_6$	2	$\Box^2$
$\mathtt{T}_7$	2	$= \mathtt{T}_6$
$\mathtt{T}_8$	1	$(\Pi \mathtt{x}^0 : \mathtt{T}^1_4.\mathtt{T}^1_5)^1$
$\mathtt{T}_9$	1	$= \mathtt{T}_2$
$\mathtt{T}_{10}$	1	None $(\twoheadrightarrow^1 *)$
$\mathtt{T}_{11}$	1	None $(\twoheadrightarrow^1 = \mathtt{T}_5)$

index	n	type
$\mathtt{T}_{12}$	0	$((\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0 @ (\mathtt{tuple\ n}^0)^0)^0$
$\mathtt{T}_{13}$	1	$\mathtt{int}^1$
$\mathtt{T}_{14}$	0	$= \mathtt{T}_{12}$
$\mathtt{T}_{15}$	0	None $(\twoheadrightarrow^0 \mathtt{int})$
$\mathtt{T}_{16}$	1	type_check $\mathtt{T}_{15}$ Γ $(\twoheadrightarrow^0 *)$
$\mathtt{T}_{17}$	1	$= \mathtt{T}_{16}$
$\mathtt{T}_{18}$	0	$(\Pi \mathtt{f}^{-1} : \mathtt{T}^0_{12}.\mathtt{T}^0_{15})^0$
$\mathtt{T}_{19}$	1	type_check $\mathtt{T}_{18}$ $(\mathtt{n}^0 : \mathtt{T}^1_2)$ $(\twoheadrightarrow^0 *)$
$\mathtt{T}_{20}$	2	$= \mathtt{T}_{19}$
$\mathtt{T}_{21}$	1	$(\Pi \mathtt{n}^0.\mathtt{T}^1_2.\mathtt{T}^0_{18})^1$
($\mathtt{T}_{22}$	0	$((\lambda \mathtt{x}^0 : *^1.\mathtt{x}^0)^0 @ (\mathtt{tuple\ 2^0})^0)^0$)

表 1: 型参照テーブル

n	実行内容
2	exe_pi $\mathtt{T}_3$ $\mathtt{T}_6$
1	exe_tuple $\mathtt{T}_9$ $\mathtt{T}_{10}$
1	exe_app $\mathtt{T}_8$ $\mathtt{T}_{10}$ $\mathtt{T}_{11}$ (tuple n)

n	実行内容
0	exe_lookup $\mathtt{T}_{13}$ $\mathtt{T}_{14}$ $\mathtt{T}_{15}$ 2
1	exe_pi $\mathtt{T}_{11}$ $\mathtt{T}_{16}$
1	exe_pi $\mathtt{T}_1$ $\mathtt{T}_{19}$

表 2: 各レベルでの実行

表 1 は、上の構文木中に現れる各式の依存関係を表現している。各 index に対してそれに対応する式と、式に付けられたレベルを保存している。また、$\mathtt{T}_{16}$ や $\mathtt{T}_{19}$ のようにレベル付けの段階では型が解らない型（後に実行を行わなければわからないもの）は None とし、後にこの None の型チェックを行わなければならない場所（4.5 節参照）には、type_check $\mathtt{T}_{15}$ Γ と書くことで、ここで後ほど部分的に証明木を構成しなければならない旨を記しておく。ここで Γ とはその部分木を生成する時に使用する型環境である。

次に表 2 だが、これは証明木中に現れた全ての実行をまとめたものである。exe_pi $\mathtt{T}_3$ $\mathtt{T}_6$ は図 4 中の (PI) に現れる実行と型マッチを行うもので、$\mathtt{T}_3$ と $\mathtt{T}_6$ を実行し、その両方の結果が $*$ または $\Box$ とマッチするならば、それぞれのテーブルの値を実行結果に書き換える。exe_tuple $\mathtt{T}_9$ $\mathtt{T}_{10}$ は同じく (TUPLE) の実行と

型マッチを表し、T_9 を実行し、結果の型が int とマッチするならば、 T_{10} の値を $*$ に書き換える。同じく (APP) の実行は exe_app T_8 T_{10} T_{11} (tuple n) で実現され、はじめに T_8 と T_{10} を実行する。次に前者の結果が ($\Pi x : A.B$) の形になっているならば、A と T_{10} の実行結果を比べ、それがマッチするならば T_{11} の値を B の中の x を (tuple n) に置換した式に書き換える。(LOOKUP) については、exe_lookup T_{13} T_{14} T_{15} 2 が T_{13} と T_{14} を実行し、$T_{13} =$ int の型マッチと、 T_{14} が、(tuple E) の形をしているかどうかのチェックをする。そして最後に ($E \geq 2$) を確認できたなら、T_{15} の値を (tuple E) に書き換える。

2 つのテーブル作成後は、付けたレベルの順番に実行と型のチェックを行っていく。はじめに型参照テーブルで使われている n の中で最も大きい数字（ここでは 2 ）の実行から処理を始める。この場合は実行テーブルの 1 番先頭の exe_pi T_3 T_6 が該当するので、前節の定義どおりに実行をする。ここでは $T_3 = *$、$T_6 = \Box$ なので実行後もテーブルに変化はない。次に型テーブルの n=2 を順にチェックする。このチェックは該当エントリの type が None ではないことを確認するものである（None があったら型エラー）。表 1 中のレベル 2 のエントリの中で None が入っているものはない。よって、次にレベル 1 の実行に移る。レベル 1 の実行は、まず exe_tuple T_9 T_{10} を行う。すると T_{10} が $*$ に書き換わる。次の exe_app T_8 T_{10} T_{11} (tuple n) も同じように実行すると、T_{11} が $= T_5$ に書き換わる。

5.2 部分木の生成

次に、exe_pi T_{11} T_{16} において、T_{11} と T_{16} を実行しようと試みる。T_{11} は先ほどの結果を使えばよい。T_{16} は type_check T_{15} Γ なので T_{15} を根とした証明木を作ろうと試みる。しかし T_{15} のレベルは 0 なので、0 の実行が終わるまでこの木を生成することはできない。そこで、この exe_pi のエントリの n を 0 に書き換えて、この実行のレベルを 0 に落とす事にする。するとこの部分木は 0 の実行で生成される。exe_pi T_1 T_{19} でも全く同じことが起こり、これもレベル 0 まで実行を見送ることにする。

これは 4.4 節で扱った「部分的に木を作らなければならない場合」に対応しており、この証明木は元の証明木の部分木ではあるが、式に割り当てるレベルはその実行・型チェックとは独立させる必要がある。よってテーブルもこの部分木用に全く新しものを用意し、それをもとに実行・型チェックを行う。そして T_{16} を得られた結果である $*$ に書き換え、定義どおりに実行・型チェックを行えば、この実行は終了する。

ここでひとつだけ注意しなければならないのは、部分木を構成する途中で、元々構成していた表 1、表 2 を参照する必要がある場合があることだ。今、型環境 Γ には $n : T_2$ と $f : T_{11}$ が入っている。この状況で新たに n の証明木を構成する必要がある場合、n の型 T_2 は元の型テーブル（表 1）を参照しなければわからない。これに対応するため、現在の実装では、もうひとつグローバルに参照可能なテーブルを用意し、表 1 のエントリをコピーする。この新しいテーブルはこの式の型チェック・実行中ならばいつでも参照することができる。また、部分木を構成した際に作成したテーブルも全ての値が決定した段階でグローバルテーブルのエントリに追加する。このテーブルは新しく部分木を作るたびに追加・更新され、最終的に全ての index と型の関係を保存することになる。

5.3 実行と型チェックを遅延させなければならない場合

ここではレベルごとの実行・チェック中に発生する「実行と型チェックを遅延させなければならない場合 (4.5 節)」に、注釈を生成してチェックを遅延させる方法を説明する。

レベル 0 の実行は、まず exe_lookup T_{13} T_{14} T_{15} 2 から開始する。T_{13} は int なので特に問題は無く実行できる。が、$T_{14} = T_{12} = ((\lambda x : *.x)@(\text{tuple n}))$ の場合 n が自由変数になってしまうので、n の値が実際に来るまでチェックを遅らせる。これは 4.5 節で扱った例と対応しており、ここでは exe_lookup が本来するべきチェックを行う代わりに 4.5 節で説明した注釈を生成し、この部分のチェックを遅らせることにする。

この注釈を実装するために、まず n の値が来たら行わなければならない処理を以下にまとめた。

- n を実際に来た値に置換して、$T_{14}(= T_{12})$ を実行
- その結果が (tuple E) の形かどうかをチェック
- $(E \geq 2)$ をチェック

その上で、T_{12} の中の n がなんらかの式に置換された時に、その結果を新しいエントリとして表に追加する。例えばこの式全体に外側から 2 を適用する。すると関数抽象の式の型は $T_{21} = (\Pi n : T_2(= *).T_{18}(= (\Pi f : T_{12}.T_{15})))$ なので、この式に 2 が適用された際に行う実行 exe_app 内で T_{21} が実行された後、T_{18} 中の全ての n が 2 に置換されるはずである。この時、T_{18} には T_{12} も含まれているので、T_{12} 中の n を 2 に置換する。そして、この置換の結果である $((\lambda x : *.x)@(\text{tuple } 2))$ を、新しいエントリ T_{22} として型テーブルに追加する。そして T_{12} と、その n を 2 で置換した結果が入っている新しく作ったエントリ T_{22} に関係を持たせるために、以下のようなメモを表とは別に作成する。

$$(T_{12},\ (n,\ 2,\ T_{22}))$$

これは「T_{12} の n を 2 で置換した結果は T_{22} 」と読む。そしてこのメモが作られる前、—exe_lookup 内で n が自由変数で値がわからない時点では、型チェックをする代わりに以下のようなチェックをする注釈を作成する。

- メモから T_{12} の各置換先を参照する
- n が置換された結果を格納したエントリ、即ち T_{22} を実行
- その結果が (tuple E) の形かどうかをチェック
- $E \geq 2$ をチェック
- 全て通れば型は正しくついているのでそのまま処理を続行。そうでなければ型エラー

生成した注釈は、f の型チェックが終わるレベル、— つまり、f よりひとつ上のレベル 0 の最後の実行として、exe_lookup が本来行うチェックの代わりに実行テーブルに追加する。最後に T_{14} の type を注釈によるチェックを通った結果 (lookup ...) の型として付くべき int に書き換えて、exe_lookup による実行は終了する。この例の場合は T_{12} に対して置換は一切行われないので置換情報は空のままである。従って注釈の内容は実行されずに最後まで残ってしまう。しかし、実際に (lookup 2 f) を実行しなければならない場合は f, n は両方とも値が解っているはずなので、レベル 0 の実行の最後に注釈による実行と型マッチが行われ、安全な実行を行うことができる。

以上のように、テーブルにある全ての実行が終了するまで型チェック・実行を繰り返すことで、最終的には注釈として残るべきもの以外は全ての型チェックが終了し、元々の式は安全が保障された状態で実行することができる。

6 現状と今後の課題

以上の方針に従って、実際に OCaml を使ってシステムを実装している。そして、それを 4 章で扱った各例について適用すると、これまで述べてきたような結果を得ることができている。今後はより多くの例について実験するとともに、型システムの各種の性質に対する証明を行っていく予定である。

謝辞 筑波大学の亀山幸義氏、及び査読者の方々から多くの有益なコメントを頂きました。ここに記して感謝いたします。

参考文献

[1] Augustsson, L., and M. Carlsson "An Exercise in Dependent Types: A Well-typed Interpreter," available from `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`, 13 pages.

[2] Augustsson, L. "Cayenne — A Language with Dependent Types," *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pp. 239–250 (September 1998).

[3] Cardelli, L. "Phase Distinctions in Type Theory," Research report, DEC SRC, 15 pages (January 1988).

[4] Chen, C., and H. Xi "Implementing Typeful Program Transformations," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '03)*, pp. 260–268 (June 2003).

[5] Chen, C., and H. Xi "Combining programming with theorem proving," *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pp. 66–77 (September 2005).

[6] Coquand, T., and G. Huet "The Calculus of Constructions," *Information and Computation*, Vol. 76, Nos. 2/3, pp. 95–120 (February/March 1988).

[7] Pašalić, E., W. Taha, and T. Sheard "Tagless Staged Interpreters for Typed Languages," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pp. 218–229 (October 2002).

[8] Hughes, J. "Type Specialisation for the lambda-Calculus; or, A New Paradigm for Partial Evaluation Based on Type Inference," In Danvy, Glück, and Thiemann editors, *Partial Evaluation (LNCS 1110)*, pp. 183–215 (February 1996).

[9] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[10] Peyton Jones, S., and E. Meijer "Henk: A Typed Intermediate Language," *Proceedings of the International Workshop on Types in Compilation (TIC 1997)*, Technical Report BCCS-97-03, Boston College Computer Science Department, 13 pages (June 1997).

[11] Pierce, B. C. *Types and Programming Languages* Cambridge: MIT Press (2002).

[12] Schmidt, D. A. *The Structure of Typed Programming Languages* Cambridge: MIT Press (1994).

[13] Xi, H., and F. Pfenning "Dependent Types in Practical Programming," *Conference Record of the 26th ACM Symposium on Principles of Programming Languages*, pp. 214–227 (January 1999).