Applied Type System

(Extended Abstract)

Hongwei Xi - Boston University

概要：Pure Type System フレームワーク (*PTS*) は型システムをデザイン/形式化する単純で一般的なアプローチを提供します。けれども依存型の存在を認めると、一般帰納、再帰型、作用 (例: 例外、参照、入出力)、などのような多くの実際のプログラミングの機能に *PTS* を適用させることが難しくなります。この論文では、実際のプログラミングの機能をサポートする型システムをたやすくデザイン/形式化できる、新しい Applied Type System (*ATS*) フレームワークを提案します。*ATS* の鍵となる突出した機能は、コンストラクトされて評価されるプログラムを含む動的な部分から、形作られて根拠となる型を含む静的な部分を完全に分離することにあります。この分離を用いると、*PTS* では許可されていましたが、プログラムが型の中に現われることは不可能になります。*ATS* の形式的な開発だけでなく、実用的なプログラミングのための型システムを作るフレームワークとして *ATS* を使ったいくつかの例も紹介します。
この翻訳の元論文は http://www.ats-lang.org/PAPER/ATS-types03.pdf です。

1. はじめに

Pure Type System フレームワーク (*PTS*) [Bar92] は型システムをデザイン/形式化する単純で一般的なアプローチを提供します。けれども依存型の存在を認めると、多くの実際のプログラミングの機能に *PTS* を適用させることが難しくなります。とりわけ、一般帰納 [CS87]、再帰型 [Men87]、作用 [HMST95]、例外 [HN88]、入出力が存在するとき *PTS* を使って純粋性を担保するためには、多大な努力が必要になることを私達は学びました。このような *PTS* の限界に対処するために、実際のプログラミングの機能をサポートする型システムをたやすくデザイン/形式化できる、新しい Applied Type System (*ATS*) フレームワークを提案します。*ATS* の鍵となる突出した機能は、コンストラクトされて評価されるプログラムを含む動的な部分から、形作られて根拠となる型を含む静的な部分を完全に分離することにあります。この分離は Dependent ML (DML) [XP99,Xi98] で開発された制限された依存型の成果に由来していて、参照や例外のような作用の存在下でも依存型を柔軟にサポートします。また、ガード型 (*guarded types*) と アサート型 (*asserting types*) という 2 つの新しい (馴染みのない) 型を導入することで、*ATS* では *PTS* よりも柔軟に効果的にプログラムの不変条件を捕捉することができることを示します。

ATS のデザインと形式化がこの論文の主な主張で、[Zen97,XP99,XCC03] と似たアイデアを使った研究成果です。*ATS* を使うと、*PTS* のある種生来の欠陥を乗り越えて、依存型の存在下で多くの一般的なプログラミングの機能をサポートする型システムをたやすく設計できます。私達は現在 *ATS* に基づいた型システムを持ち、(DML で開発されたような) 依存型のみではなく guarded recursive datatypes [XCC03] をもサポートする、型付き関数型プログラミング言語を設計/実装している最中です。Scheme が導入したようなアプローチ、つまり既存の言語に新しい言語構造を実装するような手法で、*ATS* を用いて多様な言語拡張をサポートできないか私達は探求しています。とりわけ、オブジェクト指向プログラミング [XCC03]、メタプログラミング [XCC03,CX03]、型クラス [XCC02] のようないくつかのプログラミングの機能をこの手法で扱えることを、私達は既に示しています。

論文の残りは次のような構成になっています。2 章では、*ATS* フレームワークの詳細な開発成果を示します。*ATS* で構成された汎用的な Applied Type System である ATS を形式化し、それから subject reduction と progress 定理を定義します。3 章では、*ATS* を拡張して、一般帰納、パターンマッチ、作用のような一般的で現実的なプログラミングの機能のいく

つかに順応させます。4 章では、applied type system の興味深い例をいくつか示します。最後に、関連研究と開発の将来の可能性について紹介した後、結論を述べます。この論文はオンライン [Xi03] から入手できます。

2. Applied Type System

この章では Applied Type System (*ATS*) フレームワークの形式化を示します。ここでは *ATS* で形式化された型システムを表わすのに *applied type system* という用語を使います。この後の説明では、ATS を静的な要素 (statics) と動的な要素 (dynamics) から成る一般的な applied type system として定義します。直感的に、statics と dynamics はそれぞれ型とプログラムを扱っています。単純化のために、statics は簡単な純粋型付き言語であると仮定します。そしてこの言語の型を 種 (*sort*) という名前で呼ぶことにします。statics の項を 静的な項 (*static term*) と呼びます。また dynamics の項を 動的な項 (*dynamic term*) と呼びます。そして、特別な種 *type* の静的な項は dynamics の型として機能します。

$$\frac{}{\vdash \mathcal{S}_\emptyset\ [sig]} \qquad \frac{\vdash \mathcal{S}\ [sig]}{\vdash \mathcal{S}, sc : [\sigma_1, \ldots, \sigma_n] \Rightarrow b\ [sig]}$$

$$\frac{\Sigma(a) = \sigma}{\Sigma \vdash_{\mathcal{S}} a : \sigma} \qquad \frac{\mathcal{S}(sc) = [\sigma_1, \ldots, \sigma_n] \Rightarrow b \quad \Sigma \vdash_{\mathcal{S}} s_i : \sigma_i \text{ for } i = 1, \ldots, n}{\Sigma \vdash_{\mathcal{S}} sc[s_1, \ldots, s_n] : b}$$

$$\frac{\Sigma, a : \sigma_1 \vdash_{\mathcal{S}} s : \sigma_2}{\Sigma \vdash_{\mathcal{S}} \lambda a : \sigma_1.s : \sigma_1 \to \sigma_2} \qquad \frac{\Sigma \vdash_{\mathcal{S}} s_1 : \sigma_1 \to \sigma_2 \quad \Sigma \vdash_{\mathcal{S}} s_2 : \sigma_1}{\Sigma \vdash_{\mathcal{S}} s_1(s_2) : \sigma_2}$$

図 1 statics を表わすシグニチャフォーマット規則と分類規則

2.1 Statics

静的な要素の形式的な表現を示します。基礎種を表わすのに b と書きます。2 つの特別な基礎種 *type* と *bool* が存在すると仮定します。

$$\begin{array}{ll}
\text{sorts} & \sigma ::= b \mid \sigma_1 \to \sigma_2 \\
\text{static terms} & s ::= a \mid sc[s_1, \ldots, s_n] \mid \lambda a : \sigma.s \mid s_1(s_2) \\
\text{static var. ctx.} & \Sigma ::= \emptyset \mid \Sigma, a : \sigma \\
\text{signatures} & \mathcal{S} ::= \mathcal{S}_\emptyset \mid \mathcal{S}, sc : [\sigma_1, \ldots, \sigma_n] \Rightarrow b \\
\text{static subst.} & \Theta_S ::= [] \mid \Theta_S[a \mapsto s]
\end{array}$$

静的な項の変数は α を使って表わし、静的な項の集合を s で表わします。静的な定数 sc を宣言することもできます。この定数は、静的な定数コンストラクタ scc もしくは静的な定数関数 scf のどちらかです。sc 種 (sc-sorts) を表わすのに $[\sigma_1, \ldots, \sigma_n] \Rightarrow b$ を使い、これは静的な定数に割り当てられます。静的な定数 sc が与えられたとき、もし sc になんらかの種 $\sigma_1, \ldots, \sigma_n$ について sc 種 $[\sigma_1, \ldots, \sigma_n] \Rightarrow b$ が割り当てられていて、$i = 1, \ldots, n$ について s_i に種 σ_i を割り当てることができるなら、種 b の項 $sc[s_1, \ldots, s_n]$ を作ることができます。誤解を生じない場合は、$sc[]$ を sc と書くことができます。sc 種は (標準の) 種とは見なされていないことに注意してください。

静的な変数を静的な項に写像するような静的な置換を表わすに Θ_S を使います。また Θ_S のドメインを $\mathbf{dom}(\Theta_S)$ で表わします。空の写像を表わすのに [] と書きます。$a \notin \mathbf{dom}(\Theta_S)$ を仮定したとき、a から s へのリンクで Θ_S を拡張した写像を $\Theta_S[a \mapsto s]$ と表わします。また、Θ_S を構文 $\bullet$ に適用した結果を表わすのに、$[\Theta_S]$ と書きます。構文 $\bullet$ は、静的な項、静的な項の列、この後で定義する動的な可変のコンテキスト、のいずれかを表わします。

シグニチャは宣言された静的な定数 sc に割り当てた sc 種を表わし、そのシグニチャの形成ルールは図 1 で与えられます。初期シグニチャ S_0 が次の宣言を含むことを仮定しています。

$$\mathbf{1} : [] \Rightarrow type \qquad \top : [] \Rightarrow bool \qquad \bot : [] \Rightarrow bool$$

$$\to_{tp} : [type, type] \Rightarrow type \qquad \supset : [bool, type] \Rightarrow type$$

$$\wedge : [bool, type] \Rightarrow type \qquad \leq_{tp} : [type, type] \Rightarrow bool$$

これは、その左側は静的な定数で: その右側には一致する sc 種が割り当てられています。また、それぞれの種 σ について、S_0

は 2 つの静的コンストラクタ $\forall_\sigma$ と $\exists_\sigma$ を sc 種 $[\sigma \to_{tp} type] \Rightarrow type$ に割り当てることを仮定しています。静的な定数を表わすのに中置表記を使うこともできます。例えば、$\to_{tp}\ [s_1, s_2]$ を表わすのに $s_1 \to_{tp} s_2$ のように、$\leq_{tp}\ [s_1, s_2]$ を表わすのに $s_1 \leq_{tp} s_2$ のように書けます。さらに、$\forall_a[\lambda a : \sigma.s]$ と $\exists_a[\lambda a : \sigma.s]$ をそれぞれ $\forall_a : \sigma.s$ と $\exists_a : \sigma.s$ のように書くこともできます。statics の分類規則は図 1 で与えられますが、大部分は標準的なものです。例えば、$\emptyset \vdash_{S_0} \forall_{type}[\lambda a : type.a \to_{tp} a] : type$ が導けるので、$\forall a : type.a \to_{tp} a$ は種 type を割り当てることができる静的な項です。なんらかの種 $\sigma_1, \ldots, \sigma_n$ について sc 種 $[\sigma_1, \ldots, \sigma_n] \Rightarrow type$ を割り当てられるなら、静的なコンストラクタ sc は型コンストラクタです。例えば $\mathbf{1}$, $\to_{tp}$, $\supset$, $\wedge$, $\forall_\sigma$, $\exists_\sigma$ は全て型コンストラクタですが、$\leq_{tp}$ は型コンストラクタではありません。直感的には、$\mathbf{1}$ は通常のユニット型を表わし、$\to_{tp}$ は関数型を作り、$\leq_{tp}$ は型におけるサブタイピング関係を表わします。静的なコンストラクタ $\supset$ と $\wedge$ はそれぞれガード型 (guarded types) とアサート型 (asserting types) を作り、これらは後で解説します。

静的な変数群に種を割り当てるような、静的な可変のコンテキストを表わすのに Σ を使います; $\mathbf{dom}(\Sigma)$ は Σ で宣言された静的な変数の組です; もし $a : \sigma$ が Σ で宣言されていたら、$\Sigma(a) = \sigma$ です。例によって、Σ において静的な変数 a は一回しか宣言できません。もし $\Sigma \vdash s : bool$ が導けるなら、静的な項 s は Σ の下の 命題 (*proposition*) と呼ばれます。(なんらかの静的な可変のコンテキストの下で) 命題を表わすのに P を使います。$P \supset s$ という形の型を表わすのに ガード型 (*guarded type*)、$P \wedge s$ という形の型を表わすのに アサート型 (*asserting type*) という名前を使います。これらは両方とも次の例で使います。

Example 1. int を整数 [*1] を表わす種に、**list** を sc 種 $[type, int] \Rightarrow type$ の型コンストラクタとすると、次の静的な項は型になります:

$$\forall a : type.\forall n : int.\ n \geq 0 \supset (\mathbf{list}[a, n] \to_{tp} \mathbf{list}[a, n])$$

直感的には、もし **list**[s,n] がそれぞれの要素の型が s で長さ n のリストを表わす型だとすると、リストの長さを変えないようなリストからリストへの関数を上記の型は意図していると言えます。また次の型は、もし与えられたリストが空でなければそのリストの tail を返し、そうでなければ単に例外を発生させるような関数に割り当てることを意図しています。

$$\forall a : type.\forall n : int.\ n \geq 0 \supset (\mathbf{list}[a, n] \to_{tp} n > 0 \wedge \mathbf{list}[a, n-1])$$

アサート型 $n > 0 \wedge \mathbf{list}[a, n-1]$ は、長さ n のリストにこの関数を適用した後もしこの関数が返るなら、 $n > 0$ であり返値が長さ $n-1$ のリストであるという不変条件を捕捉しています。これはいくぶん興味深い機能で、この Example 2 で詳しく解説します。Dependent ML [XP99,Xi98] における研究でも既にアサート型がありましたが、アサート型の正確な概念はこれまで形式化ざれたことがありませんでした: DML では、この論文でアサート型と呼んでいるものをシミュレートするのにサブセット種を使わねばなりませんでした。

PTS でのデザインと同様に、*ATS* でのデザインにおいても型の等価性は難解な問題です。けれどもさらなる研究によって、*ATS* における型の等価性はサブタイピング関係 $\leq_{tp}$ によって定義できることが明らかになりました。2 つの型 s_1 と s_2 が与えられたとき、もし命題 $s_1 \leq_{tp} s_2$ と命題 $s_2 \leq_{tp} s_1$ の両方が成立するなら、s_1 と s_2 は等しいと言えるのです。一般に、(ある前提の元に) 与えられた命題が成立するかどうか決定する必要があります。そこで、次ような制約関係の概念を導入します。

Definition 1. S, Σ, $\vec{P}$, P_0 をそれぞれ 静的なシグニチャ, 静的な可変のコンテキスト, Σ の下での命題の組, Σ の下での 1 つの命題であるとします。もし次の規則条件が満たされるなら、関係 $\Sigma; \vec{P} \models_S P_0$ は正規の制約関係であると言います:

(1) 図 2 の規則ルール全てが有効である; すなわちそれぞれの規則ルールについて、もしルールの根拠が成立するならルールの結論が成立し、なおかつ

(2) $\Sigma; \vec{P} \models_S s_1 \to_{tp} s_2 \leq_{tp} s_1' \to_{tp} s_2'$ は $\Sigma; \vec{P} \models_S s_1' \leq_{tp} s_1$ と $\Sigma; \vec{P} \models_S s_2 \leq_{tp} s_2'$ を意味し、なおかつ

(3) $\Sigma; \vec{P} \models_S P \supset s \leq_{tp} P' \supset s'$ は $\Sigma; \vec{P}, P' \models_S P$ と $\Sigma; \vec{P}, P' \models_S s \leq_{tp} s'$ を意味し、なおかつ

(4) $\Sigma; \vec{P} \models_S P \wedge s \leq_{tp} P' \wedge s'$ は $\Sigma; \vec{P}, P \models_S P'$ と $\Sigma; \vec{P}, P \models_S s \leq_{tp} s'$ を意味し、なおかつ

(5) $\Sigma; \vec{P} \models_S \forall a : \sigma.s \leq_{tp} \forall a : \sigma.s'$ は $\Sigma, a : \sigma; \vec{P} \models_S s \leq_{tp} s'$ を意味し、なおかつ

(6) $\Sigma; \vec{P} \models_S \exists a : \sigma.s \leq_{tp} \exists a : \sigma.s'$ は $\Sigma, a : \sigma; \vec{P} \models_S s \leq_{tp} s'$ を意味し、なおかつ

(7) $\emptyset; \emptyset \models_S scc[s_1, \ldots, s_n] \leq_{tp} scc'[s_1', \ldots, s_{n'}']$ は $scc = scc'$ を意味します

[*1] 形式的には、それぞれの整数 n について、sc 種 $[] \Rightarrow int$ の静的なコンストラクタ $\underline{n}$ が存在し、$\underline{n}[]$ は n に相当する種 int の静的な項であることを言う必要があります。

$$\frac{}{\Sigma;\vec{P}\models_{\mathcal{S}}\top}\ \textbf{(reg-true)} \qquad \frac{}{\Sigma;\vec{P},\bot\models_{\mathcal{S}}P}\ \textbf{(reg-false)}$$

$$\frac{\Sigma;\vec{P}\models_{\mathcal{S}}P_0}{\Sigma,a:\sigma;\vec{P}\models_{\mathcal{S}}P_0}\ \textbf{(reg-var-thin)} \qquad \frac{\Sigma\vdash_{\mathcal{S}}P:\mathit{bool}\quad\Sigma;\vec{P}\models_{\mathcal{S}}P_0}{\Sigma;\vec{P},P\models_{\mathcal{S}}P_0}\ \textbf{(reg-prop-thin)}$$

$$\frac{\Sigma,a:\sigma;\vec{P}\models_{\mathcal{S}}P\quad\Sigma\vdash_{\mathcal{S}}s:\sigma}{\Sigma;\vec{P}[a\mapsto s]\models_{\mathcal{S}}P[a\mapsto s]}\ \textbf{(reg-subst)} \qquad \frac{\Sigma;\vec{P}\models_{\mathcal{S}}P_0\quad\Sigma;\vec{P},P_0\models_{\mathcal{S}}P}{\Sigma;\vec{P}\models_{\mathcal{S}}P}\ \textbf{(reg-cut)}$$

$$\frac{\Sigma\vdash_{\mathcal{S}}s:\mathit{type}}{\Sigma;\vec{P}\models_{\mathcal{S}}s\leq_{tp}s}\ \textbf{(reg-refl)} \qquad \frac{\Sigma;\vec{P}\models_{\mathcal{S}}s_1\leq_{tp}s_2\quad\Sigma;\vec{P}\models_{\mathcal{S}}s_2\leq_{tp}s_3}{\Sigma;\vec{P}\models_{\mathcal{S}}s_1\leq_{tp}s_3}\ \textbf{(reg-tran)}$$

図 2 規則ルール

$\Sigma;\vec{P}\models_S P_0$ と書くときはいつでも $P\in\vec{P},P_0$ について $\Sigma\vdash_S P:\mathit{bool}$ を導出できることを仮定していることに注意してください。

ATS の dynamics を作るとき、制約関係を必要とします。全ての単一の規則条件と同様に、全ての単一の規則ルールは subject reduction 定理 (定理 1) と progress 定理 (定理 2) を規定するために後で使うことになります。一般に、*ATS* フレームワークは制約関係の上にパラメータ化されています。この時点では、制約関係の決定可能性を気にする必要がありません。それぞれの制約関係 $\models_S$ について、どのような $\Sigma,\vec{P}$ と P_0 が与えられても $\Sigma;\vec{P}\models_S P_0$ が満たされるかどうか、決定してくれるオラクルが存在することを単純に仮定することができます。後で、制約関係を決定する実用的なアルゴリズムを持つ applied type system の例をいくつか紹介します。

非叙述性 (impredicativity) のために、与えられたシグニチャ S について制約関係 $\models_S$ を厳密に定義できる方法が一般により難しい問題であることは、強調されるべきです。[Xi03] では、モデル理論的なアプローチで私達はこの問題を解決しています。

2.2 Dynamics

ATS の dynamics は型付き言語で、種 *type* の静的な項は dynamics における型です。動的な定数をいくつか宣言することができ、引数 n の動的な定数 dc それぞれに次の形の dc 型を割り当てることができます。

$$\forall a_1:\sigma_1\ldots\forall a_k:\sigma_k.P_1\supset(\ldots(P_m\supset([s_1,\ldots,s_n]\Rightarrow_{tp}s))\ldots)$$

このとき $s_1,\ldots,s_n,s$ は型であると仮定しています。dc が動的なコンストラクタ dcc である場合、なんらかの型コンストラクタ scc について 型 s は $scc[\vec{s}]$ の形を取れなければなりません。すると dcc は scc と関連があると言えるのです。静的な項の (空である可能性のある) 列を表わすのに $\vec{s}$ を使っていることに注意してください。例えば、次のように dc 型を割り当てることで、2 つの動的なコンストラクタ $\underline{nil}$ と $\underline{cons}$ を型コンストラクタ **list** と関連付けることができます。

$$\begin{array}{l}\underline{nil}:\forall a:\mathit{type}.\mathbf{list}[a,0]\\ \underline{cons}:\forall a:\mathit{type}.\forall n:\mathit{int}.n\geq 0\supset([a,\mathbf{list}[a,n]]\Rightarrow_{tp}\mathbf{list}[a,n+1])\end{array}$$

このとき、要素の型が a で長さが n のリストを表わす型として $\mathbf{list}[a,n]$ を使っています。

動的な値を動的な項に写像する動的な置換を Θ_D で表わします。また Θ_D のドメインを $\mathbf{dom}(\Theta_D)$ で表わします。静的な置換と同じように、動的な置換を形成して適用するような構文を示すことを省きます。$\mathbf{dom}(\Theta_D^1)\cap\mathbf{dom}(\Theta_D^2)=\emptyset$ となるような Θ_D^1 と Θ_D^2 が与えられたとき、Θ_D^1 と Θ_D^2 の和集合を $\Theta_D^1\cup\Theta_D^2$ で表わします。

任意の構文を $\bullet$ で表わすと、$\Sigma=a_1:\sigma_1,\ldots,a_k:\sigma_k$ において、$\forall a_1:\sigma_1\ldots\forall a_k:\sigma_k.\bullet$ を $\forall\Sigma.\bullet$ と書くことができます。同様に、$\vec{P}=P_1,\ldots,P_m$ において $P_1\supset(\ldots(P_m\supset\bullet)\ldots)$ を $\vec{P}\supset\bullet$ と書くことができます。例えば、dc 型は常に $\forall\Sigma.\vec{P}\supset([s_1,\ldots,s_n]\Rightarrow_{tp}s)$ の形になります。動的な定数の宣言を許可するために、シグニチャの定義を次のように拡張する必要があります。

$$\text{signatures }\mathcal{S}::=\ldots\mid\mathcal{S},dc:\forall\Sigma.\vec{P}\supset([s_1,\ldots,s_n]\Rightarrow_{tp}s)$$

さらに、シグニチャを作るために次のような追加のルールが必要になります。

$$\frac{\vdash \mathcal{S}\ [sig] \quad \Sigma \vdash_{\mathcal{S}} P : bool \ \text{ for each } P \text{ in } \vec{P} \quad \Sigma \vdash_{\mathcal{S}} s_i : type \ \text{ for each } 1 \le i \le n \quad \Sigma \vdash_{\mathcal{S}} s : type}{\vdash \mathcal{S}, dc : \forall \Sigma.\vec{P} \supset ([s_1, \ldots, s_n] \Rightarrow_{tp} s)\ [sig]}$$

動的な項の変数として x を、動的な項として d を用いるとき、dynamics の構文を図 3 に示します。引数の個数が n の動的な定数 dc が与えられたとき、引数 $d_1, \ldots, d_n$ への dc の適用を $dc[d_1, \ldots, d_n]$ と書きます。$n = 0$ の場合には $dc[]$ の代わりに dc と書くこともできます。

$$\begin{array}{lll}
\text{dyn. terms} & d ::= & x \mid dc[d_1, \ldots, d_n] \mid \mathbf{lam}\ x.d \mid \mathbf{app}(d_1, d_2) \mid \\
& & \supset^+(v) \mid \supset^-(d) \mid \wedge(d) \mid \mathbf{let}\ \wedge(x) = d_1\ \mathbf{in}\ d_2 \mid \\
& & \forall^+(v) \mid \forall^-(d) \mid \exists(d) \mid \mathbf{let}\ \exists(x) = d_1\ \mathbf{in}\ d_2 \\
\text{values} & v ::= & x \mid dcc\,[v_1, \ldots, v_n] \mid \mathbf{lam}\ x.d \mid \supset^+(v) \mid \wedge(v) \mid \forall^+(v) \mid \exists(v) \\
\text{dyn. var. ctx.} & \Delta ::= & \emptyset \mid \Delta, x : s \\
\text{dyn. subst.} & \Theta_D ::= & [] \mid \Theta_D[x \mapsto d]
\end{array}$$

図 3 dynamics の構文

$$\frac{\vdash \mathcal{S}\ [sig]}{\Sigma \vdash_{\mathcal{S}} \emptyset\ [dctx]} \qquad \frac{\Sigma \vdash_{\mathcal{S}} \Delta\ [dctx] \quad \Sigma \vdash_{\mathcal{S}} s : type}{\Sigma \vdash_{\mathcal{S}} \Delta, x : s\ [dctx]}$$

図 4 動的な可変のコンテキストを表わす形成ルール

型の導出において帰納的な意味付けに必要な Lemma 3 を証明するために、標識 $\supset^+(\cdot)$, $\supset^-(\cdot)$, $\wedge(\cdot)$, $\forall^+(\cdot)$, $\forall^-(\cdot)$, $\exists(\cdot)$ を導入します。これらの標識がないと、型の導出において帰納的証明を行なうことが、著しく困難になります。Lemma 3 を証明することも困難になってしまうでしょう。

$\Sigma \vdash_S \Delta[dctx]$ の形の判定は、Σ と S の下で Δ が well-formed な動的な可変のコンテキストであることを示しています。このような判定を導出するルールを図 4 に示します。型付けされたコンテキトを $\Sigma; \vec{P}; \Delta$ で表わします。次のルールは $\vdash_S \Sigma; \vec{P}; \Delta$ の形の判定を導出しています。

$$\frac{\Sigma \vdash_{\mathcal{S}} P : bool \ \text{ for each } P \text{ in } \vec{P} \quad \Sigma \vdash \Delta\ [dctx]}{\vdash_{\mathcal{S}} \Sigma; \vec{P}; \Delta}$$

これは $\Sigma; \vec{P}; \Delta$ が well-formed であることを示しています。

$\Sigma; \vec{P}; \Delta$ が well-formed な型付けされたコンテキストで、かつ $\Sigma \vdash_S s : type$ 導出できると仮定したとき、型付け判定は $\Sigma; \vec{P}; \Delta \vdash_S d : s$ の形を取ります。制約関係 $\models_S$ が正則であると仮定したとき、このような判定を導くための型付けルールを図 5 に示します。$\Sigma \vdash_S \Theta_S : \Sigma_0$ と書くとき、それぞれの $a \in \mathbf{dom}(\Theta_S) = \mathbf{dom}(\Sigma)$ について $\Sigma \vdash_S \Theta_S(a) : \Sigma(a)$ が導出できることを意味します。型付けルールに関連する明らかな条件のいくつかを省略していることに注意してください。例えば、ルール **(ty-∀-intro)** を適用するとき、$\vec{P}$, Δ もしくは s において値 a は自由に出現 (free occurrences) できません。また値の形に、型付けルール **(ty-gua-intro)** と **(ty-∀-intro)** の制約を付けています。これは後で ATS に作用を導入するための準備です。[*2] 技術的な理由で、ルール **(ty-var)** を次のようなルールで置き換えます。

$$\frac{\vdash_{\mathcal{S}} \Sigma; \vec{P}; \Delta \quad \Delta(x) = s \quad \Sigma; \vec{P} \models_{\mathcal{S}} s \le_{tp} s'}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} x : s'}\ \textbf{(ty-var')}$$

これは **(ty-var)** と **(ty-sub)** を結合しています。この置換は Lemma 2 成立させるために必要です。

ここで、動的な項を評価するルールの表現に進む前に、ガード型とアサート型がセキュリティを強制する興味深い役割を演じるようなシナリオをスケッチしてみましょう。これらの型を理解をさらに容易にしてくれるはずです。

Example 2. Secret は命題定数で、password と action が次のような dc 型が割り当てられた 2 つの関数で宣言されている、

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s \quad \Sigma; \vec{P} \models_{\mathcal{S}} s \leq_{tp} s'}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s'} \textbf{(ty-sub)}$$

$$\frac{\begin{array}{c}\vdash_{\mathcal{S}} \Sigma; \vec{P}; \Delta \qquad \mathcal{S}(dc) = \forall \Sigma_0.\vec{P}_0 \supset [s_1, \ldots, s_n] \Rightarrow_{tp} s \\ \Sigma \vdash_{\mathcal{S}} \Theta_S : \Sigma_0 \qquad \Sigma; \vec{P} \models_{\mathcal{S}} P[\Theta_S] \text{ for each } P \in \vec{P}_0 \\ \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_i : s_i[\Theta_S] \text{ for } i = 1, \ldots, n \qquad \Sigma; \vec{P} \models_{\mathcal{S}} s[\Theta_S] \leq_{tp} s'\end{array}}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} dc[d_1, \ldots, d_n] : s'} \textbf{(ty-dc)}$$

$$\frac{\vdash_{\mathcal{S}} \Sigma; \vec{P}; \Delta \quad \Delta(x) = s \quad \Sigma; \vec{P} \models_{\mathcal{S}} s \leq_{tp} s'}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} x : s'} \textbf{(ty-var)}$$

$$\frac{\Sigma; \vec{P}; \Delta, x : s_1 \vdash_{\mathcal{S}} d : s_2}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{lam}\ x.d : s_1 \to_{tp} s_2} \textbf{(ty-fun-intro)}$$

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_1 : s_1 \to_{tp} s_2 \quad \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_2 : s_1}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{app}(d_1, d_2) : s_2} \textbf{(ty-fun-elim)}$$

$$\frac{\Sigma; \vec{P}, P; \Delta \vdash_{\mathcal{S}} d : s}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \supset^{+}(d) : P \supset s} \textbf{(ty-gua-intro)}$$

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : P \supset s \quad \Sigma; \vec{P} \models_{\mathcal{S}} P}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \supset^{-}(d) : s} \textbf{(ty-gua-elim)}$$

$$\frac{\Sigma; \vec{P} \models_{\mathcal{S}} P \quad \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \wedge(d) : P \wedge s} \textbf{(ty-ass-intro)}$$

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_1 : P \wedge s_1 \quad \Sigma; \vec{P}, P; \Delta, x : s_1 \vdash_{\mathcal{S}} d_2 : s_2}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{let}\ \wedge(x) = d_1\ \mathbf{in}\ d_2 : s_2} \textbf{(ty-ass-elim)}$$

$$\frac{\Sigma, a : \sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} v : s}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \forall^{+}(v) : \forall a : \sigma.s} \textbf{(ty-}\forall\textbf{-intro)}$$

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : \forall a : \sigma.s \quad \Sigma \vdash_{\mathcal{S}} s_0 : \sigma}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \forall^{-}(d) : s[a \mapsto s_0]} \textbf{(ty-}\forall\textbf{-elim)}$$

$$\frac{\Sigma \vdash_{\mathcal{S}} s_0 : \sigma \quad \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s[a \mapsto s_0]}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \exists(d) : \exists a : \sigma.s} \textbf{(ty-}\exists\textbf{-intro)}$$

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_1 : \exists a : \sigma.s_1 \quad \Sigma, a : \sigma; \vec{P}; \Delta, x : s_1 \vdash_{\mathcal{S}} d_2 : s_2}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{let}\ \exists(x) = d_1\ \mathbf{in}\ d_2 : s_2} \textbf{(ty-}\exists\textbf{-elim)}$$

図 5 dynamics の型付けルール

$$\underline{action} : \underline{Secret} \supset [\mathbf{1}] \Rightarrow_{tp} \mathbf{1} \qquad \underline{password} : [\mathbf{1}] \Rightarrow_{tp} \underline{Secret} \wedge \mathbf{1}$$

と仮定します。

password を呼び出しが返る前になんらかの secret 情報を検証しなければならない、というような方法で関数 *password* を実装できます。一方では、関数呼び出し $\underline{action}[\langle\rangle]$ をする前に、命題 *Secret* を成立させなければなりません。このとき、$\langle\rangle$ はユニット型 $\mathbf{1}$ の値を意味します。もう一方、関数呼び出し $\underline{password}[\langle\rangle]$ が返った後では、命題 *Secret* は成立しています。従って、*action* 呼び出しは次のプログラムパターンを意味していることになります:

$$\mathbf{let}\ \wedge(x) = \underline{password}[\langle\rangle]\ \mathbf{in}\ \ldots \underline{action}[\langle\rangle] \ldots$$

特に x のスコープ外における *action* 呼び出しは ill-typed です。なぜなら命題 *Secret* を成立させることができないからです。

値渡し (call-by-value) の動的な構文を動的な項に割り当てるために、次に定義する評価コンテキストを利用します:

*2 実際には Theorem 2 を成立させるために、型付けルール **(ty-gua-intro)** の値に対する制約は、この時点で必要です。

$$\begin{aligned}\text{eval. ctx. } E ::= {} & [] \mid dc[v_1,\ldots,v_{i-1},E,d_{i+1},\ldots,d_n] \mid \\ & \mathbf{app}(E,d) \mid \mathbf{app}(v,E) \mid \supset^-(E) \mid \forall^-(E) \mid \\ & \wedge(E) \mid \mathbf{let}\ \wedge(x) = E\ \mathbf{in}\ d \mid \exists(E) \mid \mathbf{let}\ \exists(x) = E\ \mathbf{in}\ d\end{aligned}$$

Definition 2. 簡約可能式 (redex) と簡約 (reducation) を次のように定義します。

- $\mathbf{app}(\mathbf{lam}x.d, v)$ は簡約可能式で、その簡約は $d[x \mapsto v]$ です。
- $\supset^-(\supset^+(v))$ は簡約可能式で、その簡約は v です。
- $\mathbf{let} \wedge (x) = \wedge(v) \mathbf{in} d$ は簡約可能式で、その簡約は $d[x \mapsto v]$ です。
- $\forall^-(\forall^+(v))$ は簡約可能式で、その簡約は v です。
- $\mathbf{let}\exists(x) = \exists(v)\mathbf{in}d$ は簡約可能式で、その簡約は $d[x \mapsto v]$ です。
- なんらかの値 v に等しく定義された $dcf[v_1,\ldots,v_n]$ は簡約可能式で、その簡約は v です。

なんらかの簡約可能式 d とその簡約 d' について $d_1 = E[d]$ かつ $d_2 = E[d']$ のような 2 つの動的な項 d_1 と d_2 が与えられたとき、$d_1 \hookrightarrow d_2$ は 1 ステップで d_1 を d_2 に簡約することを意味します。$\hookrightarrow$ の再帰的で推移的なクロージャを $\hookrightarrow^*$ で表わします。

動的な定数の関数 dcf それぞれに割り当てられた型が妥当であると仮定します。すなわち、もし $\emptyset;\emptyset;\emptyset \vdash_S dcf[v_1,\ldots,v_n] : s$ が導出でき、かつ $dcf[v_1,\ldots,v_n] \hookrightarrow v$ が成立するなら、$\emptyset;\emptyset;\emptyset \vdash_S v : s$ が導出できるとします。

判定 J が与えられたとき、D が J の導出であることを示すために $D :: J$ と書きます。つまり D が J を結論とする導出であることを意味しています。

Lemma 1 (Substitution). 次が成立します。

(1) $D :: \Sigma, a : \sigma; \vec{P}; \Delta \vdash_S d : s$ と $D_0 :: \Sigma \vdash_S s_0 : \sigma$ を仮定します。すると $\Sigma; \vec{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash_S d : s[a \mapsto s_0]$ を導出できます。

(2) $D :: \Sigma; \vec{P}, P; \Delta \vdash_S d : s$ と $\Sigma; \vec{P} \models_S P$ を仮定します。すると $\Sigma; \vec{P}; \Delta \vdash_S d : s$ を導出できます。

(3) $D :: \Sigma; \vec{P}; \Delta, x : s_1 \vdash_S d_2 : s_2$ と $\Sigma; \vec{P}; \Delta \vdash_S d_1 : s_1$ を仮定します。すると $\Sigma; \vec{P}; \Delta \vdash_S d_2[x \mapsto d_1] : s_2$ を導出できます。

Proof. D に関する構造帰納法を使って (1),(2),(3) を簡単に証明できます。(1) と (2) を証明する際、規則ルール **(reg-subst)** と **(reg-cut)** をそれぞれ利用する必要があります。

導出 D が与えられたとき、D の高さを $\mathbf{h}(D)$ で表わします。これは一般的な方法で定義できます。

Lemma 2. $D :: \Sigma; \vec{P}; \Delta, x : s_1 \vdash_S d : s_2$ と $\Sigma; \vec{P} \models_S s_1' \leq_{tp} s_1$ を仮定します。すると $\mathbf{h}(D') = \mathbf{h}(D)$ となるような導出 $D' :: \Sigma; \vec{P}; \Delta, x : s_1' \vdash_S d : s_2$ が存在します。

Proof. 証明は D に対する構造帰納法を使ってすぐに得られます。D に最後に適用されたルールが **(ty-var')** であるような場合を扱うために、規則ルール **(reg-trans)** を使います。

ルール **(tyrule-eq)** が存在するために、次の反転は一般的なものと少し異なります。

Lemma 3 (Inversion). $D :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ を仮定します。

(1) もし $d = \mathbf{lam}x.d_1$ かつ $s = s_1 \to_{tp} s_2$ ならば $\mathbf{h}(D') \leq \mathbf{h}(D)$ であるような導出 $D' :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ が存在します。なおかつ D' に適用された最後のルールは **(ty-sub)** ではありません。

(2) もし $d = \supset^+(d_1)$ かつ $s = P \supset s_1$ ならば $\mathbf{h}(D') \leq \mathbf{h}(D)$ であるような導出 $D' :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ が存在します。なおかつ D' に適用された最後のルールは **(ty-sub)** ではありません。

(3) もし $d = \wedge(d_1)$ かつ $s = P \wedge s_1$ ならば $\mathbf{h}(D') \leq \mathbf{h}(D)$ であるような導出 $D' :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ が存在します。なおかつ D' に適用された最後のルールは **(ty-sub)** ではありません。

(4) もし $d = \forall^+(d_1)$ かつ $s = \forall a : \sigma.s_1$ ならば $\mathbf{h}(D') \leq \mathbf{h}(D)$ であるような導出 $D' :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ が存在します。なおかつ D' に適用された最後のルールは **(ty-sub)** ではありません。

(5) もし $d = \exists(d_1)$ かつ $s = \exists a : \sigma.s_1$ ならば $\mathbf{h}(D') \leq \mathbf{h}(D)$ であるような導出 $D' :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ が存在します。なおかつ D' に適用された最後のルールは **(ty-sub)** ではありません。

Proof. $\mathbf{h}(D)$ に関する帰納法を使います。特に、(1) を成立するために Lemma 2 が必要になります。

ATS の型の健全性は次に示す 2 つの定理に基づいています。これらの証明は一般的であるため、ここでは省略します。

Theorem 1 (Subject Reduction). $D :: \Sigma; \vec{P}; \Delta \vdash_S d : s$ と $d \hookrightarrow d'$ の両方を仮定します。すると $\Sigma; \vec{P}; \Delta \vdash_S d : s$ を導出できます。

Theorem 2 (Progress). $D :: \emptyset;\emptyset;\emptyset \vdash_S d : s$ を仮定します。すると、d は値となるか、もしくはなんらかの動的な項 d' につ

いて $d \hookrightarrow d'$ が成立するか、もしくはなんらかの簡約可能式でない動的な項 $dcf(v_1, \ldots, v_n)$ について $d = E[dcf(v_1, \ldots, v_n)]$ が成立します。

2.3 型消去済の動的な項

動的な項から、意味論を維持した型無しのラムダ式に変換する関数を示します。次のように形式的に定義された型消去済の動的な項を e で表わします:

$$\begin{array}{ll} \text{erasures} & e ::= x \mid dc[e_1, \ldots, e_n] \mid \mathbf{lam}\ x.e \mid \mathbf{app}(e_1, e_2) \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\ \text{erasure values} & w ::= x \mid dcc\,[w_1, \ldots, w_n] \mid \mathbf{lam}\ x.e \end{array}$$

すると、動的な項から型消去済の動的な項への変換する、次のような関数 $|\cdot|$ を定義できます。

$$\begin{array}{rlrl} |x| &= x & |dc[d_1, \ldots, d_n]| &= dc[|d_1|, \ldots, |d_n|] \\ |\mathbf{lam}\ x.d| &= \mathbf{lam}\ x.|d| & |\mathbf{app}(d_1, d_2)| &= \mathbf{app}(|d_1|, |d_2|) \\ |\supset^+(d)| &= |d| & |\supset^-(d)| &= |d| \\ |\wedge(d)| &= |d| & |\mathbf{let}\ \wedge(x) = d_1\ \mathbf{in}\ d_2| &= \mathbf{let}\ x = |d_1|\ \mathbf{in}\ |d_2| \\ |\forall^+(d)| &= |d| & |\forall^-(d)| &= |d| \\ |\exists(d)| &= |d| & |\mathbf{let}\ \exists(x) = d_1\ \mathbf{in}\ d_2| &= \mathbf{let}\ x = |d_1|\ \mathbf{in}\ |d_2| \end{array}$$

Theorem 3. $D :: \emptyset; \emptyset; \emptyset \vdash_S d : s$ を仮定したとき
(1) もし $d \hookrightarrow^* v$ ならば $|d| \hookrightarrow^* |v|$ です。
(2) もし $|d| \hookrightarrow^* w$ ならば $d \hookrightarrow^* v$ かつ $|v| = w$ であるような値 v が存在します。
Proof. (1) は自明です。(2) は D に対する構造帰納法から得られます。
Theorem 3 を用いると、型消去済の動的な項 d を単純に評価することで、動的な項 d を評価することができます。

3. 拡張

この章では、 *ATS* を拡張して一般的で現実的なプログラミングの機能のいくつかをサポートします。

General Recursion *ATS* で一般帰納 (general recursion) をサポートするために、不動点演算子 **fix** を導入します。ここでは、変数 x を **lam** 変数と呼び、**fix** 変数 f を導入します。**lam** 変数か **fix** 変数のいずれかである変数を xf で表わします。

$$\begin{array}{ll} \text{dyn. terms} & d ::= \ldots \mid f \mid \mathbf{fix}\ f.d \\ \text{dyn. var. ctx.} & \Delta ::= \ldots \mid \Delta, f : s \\ \text{dyn. subst.} & \Theta_D ::= \ldots \mid \Theta_D[f \mapsto d] \end{array}$$

ルール **(ty-var)** が変更を受けるために必要で、ルール **(ty-fix)** が不動点演算子の扱いを追加するために必要です:

$$\frac{\vdash_{\mathcal{S}} \Sigma; \vec{P}; \Delta \quad \Delta(xf) = s}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} xf : s'}\ \textbf{(ty-var)} \qquad \frac{\Sigma; \vec{P}; \Delta, f : s \vdash_{\mathcal{S}} d : s}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{fix}\ f.d : s}\ \textbf{(ty-fix)}$$

fix $f.d$ の動的な項は簡約可能式で、その簡約は $d[f \mapsto \mathbf{fix} f.d]$ になります。この拡張について、subject reduction 定理 (Theorem 1) と progress 定理 (Theorem 2) の両方が素直に成立します。

Datatypes and Pattern Matching *ATS* を拡張してデータ型とパターンマッチをサポートするアプローチを示します。それからいくつかの簡単な例を紹介します。次のような追加の構文が必要になります。

$$\begin{array}{ll} \text{patterns} & p ::= x \mid dcc\,[p_1, \ldots, p_n] \\ \text{dyn. terms} & d ::= \ldots \mid \mathbf{case}\ d_0\ \mathbf{of}\ p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n \\ \text{eval. ctx.} & E ::= \ldots \mid \mathbf{case}\ E\ \mathbf{of}\ p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n \end{array}$$

いつものように、どのような変数 x もパターン中に一度だけ登場できます。値 v とパターン p が与えられたとき、$v = p[\Theta_D]$

$$\frac{}{v \Downarrow x \Rightarrow [x \mapsto v]}\ \textbf{(vp-var)} \qquad \frac{v_i \Downarrow p_i \Rightarrow \Theta_D^i \quad \text{for } 1 \le i \le n}{dcc\,[v_1, \ldots, v_n] \Downarrow dcc\,[p_1, \ldots, p_n] \Rightarrow \Theta_D^1 \cup \ldots \cup \Theta_D^n}\ \textbf{(vp-dcc)}$$

を示すのに $v \Downarrow p \Rightarrow \Theta_D$ の形の判定を使います。そのような判定を導出するためのルールは次のように与えられます。
もし、なんらかの動的な代入 Θ_D について $v \Downarrow p \Rightarrow \Theta_D$ が導出できるなら、v は p にマッチすると言えます。ルール **(vp-dcc)** において、$n = 0$ のとき空の動的な代入 $[]$ になるようなユニオン $\Theta_D^1 \cup \ldots \cup \Theta_D^n$ は well-defined であることに注意してください。なぜなら、1 つのパターン中にどのような変数も最大一度まで登場できるからです。

$1 \le i \le n$ について $v \Downarrow p_i \Rightarrow \Theta_D$ が成立するなら、**case** v **of** $p_1 \Rightarrow d_1 | \cdots | p_n \Rightarrow d_n$ の形の動的な項は簡約可能式で、その簡約は $d_i[\Theta_D]$ になります。もし v が複数のパターン p_i にマッチするなら、このような簡約可能式の簡約は非決定論を引き起こすかもしれないことに注意してください。

$$\frac{\Sigma \vdash_{\mathcal{S}} s : \mathit{type}}{\Sigma \vdash x \Downarrow s \Rightarrow \emptyset; \emptyset; \emptyset, x : s}\ \textbf{(pat-var)}$$

$$\frac{\begin{array}{c}\mathcal{S}(dcc) = \forall \Sigma_0.\vec{P}_0 \supset ([s_1, \ldots, s_n] \Rightarrow_{tp} scc\,[\vec{s}_0]) \\ \Sigma, \Sigma_0 \vdash p_i \Downarrow s_i \Rightarrow \Sigma_i; \vec{P}_i; \Delta_i \quad \text{for } 1 \le i \le n \\ \Sigma' = \Sigma_1, \ldots, \Sigma_n \quad \vec{P}' = \vec{P}_1, \ldots, \vec{P}_n \quad \Delta' = \Delta_1, \ldots, \Delta_n\end{array}}{\Sigma \vdash dcc\,[p_1, \ldots, p_n] \Downarrow scc\,[\vec{s}] \Rightarrow \Sigma_0, \Sigma'; \vec{P}_0, scc\,[\vec{s}_0] \le_{tp} scc\,[\vec{s}], \vec{P}'; \Delta'}\ \textbf{(pat-dc)}$$

$$\frac{\Sigma \vdash p \Downarrow s_1 \Rightarrow \Sigma'; \vec{P}'; \Delta' \quad \Sigma, \Sigma'; \vec{P}, \vec{P}'; \Delta, \Delta' \vdash_{\mathcal{S}} d : s_2}{\Sigma; \vec{P}; \Delta \vdash p \Rightarrow d \Downarrow s_1 \Rightarrow s_2}\ \textbf{(ty-cla)}$$

$$\frac{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_0 : s_1 \quad \Sigma; \vec{P}; \Delta \vdash p_i \Downarrow d_i : s_1 \Rightarrow s_2 \quad \text{for } 1 \le i \le n}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} (\textbf{case}\ d_0\ \textbf{of}\ p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n) : s_2}\ \textbf{(ty-cas)}$$

図 6 パターンマッチの型付けルール

パターンマッチの型付けルールを図 6 に示します。$\Sigma \vdash p \Downarrow s \Rightarrow \Sigma'; P'; \Delta'$ の判定の意味は、次の補題で形式的に捕捉されます。

Lemma 4. $D :: \emptyset; \emptyset; \emptyset \vdash_S v : s, \varepsilon_1 :: \emptyset \vdash p \Downarrow s \vdash \Sigma; \vec{P}; \Delta$ と $\varepsilon_2 :: v \Downarrow p \Rightarrow \Theta_D$ を仮定します。すると、$\vec{P}$ のそれぞれの P について $\emptyset; \emptyset \models_S P[\Theta_S]$ であり、かつ $\emptyset; \emptyset; \emptyset \vdash_S \Theta_D : \Delta$ ような $\Theta_S : \Sigma$ が存在します。

Proof. この補題は ε_1 における構造帰納法から得られます。

例として、次の判定は導出可能です。

$$a' : \mathit{type}, n' : \mathit{int} \vdash \underline{\mathit{cons}}[x_1, x_2] \Downarrow \mathbf{list}[a', n'] \Rightarrow \Sigma; \vec{P}; \Delta$$

ここで $\underline{\mathit{cons}}$ には次の dc 型が割り当てられます。

$$\forall a : \mathit{type}.\forall n : \mathit{int}.n \ge 0 \supset ([a, \mathbf{list}[a, n]] \Rightarrow_{tp} \mathbf{list}[a, n + 1])$$

なおかつ $\Sigma = (a : \mathit{type}, n : \mathit{int})$、 $\vec{P} = (n \ge 0, \mathbf{list}[a, n + 1] \le_{tp} \mathbf{list}[a', n'])$、かつ $\Delta = (x_1 : a, x_2 : \mathbf{list}[a, n])$ です。

この拡張において subject reduction 定理 (Theorem 1) はすぐに証明できます: 簡約されたインデックスが次の形になる場合を扱うために、Lemma 4 が必要になります:

$$\textbf{case}\ d_0\ \textbf{of}\ p_1 \Rightarrow d_1 \mid \ldots \mid p_n \Rightarrow d_n$$

また、この拡張において progress 定理 (Theorem 2) を成立させることができます。このとき well-typed なプログラム d が次の形を取る可能性を考慮するために、定理のわずかな修正が必要です。

$$E[\textbf{case}\ v_0\ \textbf{of}\ p_1 \Rightarrow d_1 \mid \ldots \mid p_n \Rightarrow d_n]$$

ここで、もし d が値でもなく簡約することもできないなら、$1 \leq i \leq n$ について v_0 はどのような p_i にもマッチしません。

Effects *PTS* と異なり、*ATS* は参照や例外のような作用をサポートする際も素直な方法で拡張できます。例えば、参照を *ATS* に導入するために、sc 種 $[type] \Rightarrow type$ の型コンストラクタ ref と、次のように対応する dc 型が割り当てられた動的な関数を単純に宣言できます。

$$\begin{aligned} mkref &: \forall a : type.[a] \Rightarrow_{tp} ref(a) \\ deref &: \forall a : type.[ref(a)] \Rightarrow_{tp} a \\ assign &: \forall a : type.[ref(a), a] \Rightarrow_{tp} \mathbf{1} \end{aligned}$$

これらの関数の意図する意味は明らかです。また、ref が不変の型コンストラクタになってしまう問題に対処するため、Definition 1 に次の規則条件を追加する必要があります。

- $\Sigma; \vec{P} \models_S ref(s) \leq_{tp} ref(s')$ は $\Sigma; \vec{P} \models_S s \leq_{tp} s'$ と $\Sigma; \vec{P} \models_S s' \leq_{tp} s$ を意味します。

この拡張に動的な意味論を割り当てる手法は一般的なもので、subject reduction 定理と progress 定理の両方を成立できます。この手法の詳細については [Har94] を参照してください。

同様に *ATS* に例外を導入することも素直な方法で可能です。そのため詳細は省略します。

4. Applied Type System の例

驚くことではありませんが、λ-cube [Bar92] における System λ_2 と λ_ω が applied type system であることは簡単に示すことができます。また guarded recursive datatypes で λ_2 を拡張した $\lambda_{G\mu}$ 言語 [XCC03] と Dependent ML [XP99] も applied type system です。より詳細な説明は [Xi03] を参照してください。

5. 関連研究と結論

ATS フレームワークは、依存データ型を使って ML の型システムを改良した Dependent ML [XP99,Xi98] と、最新の研究である guarded recursive datatypes [XCC03] の成果に基づいています。これら 2 つの型 [*3] の間に類似点を見つけた私達は、自然にこれらの統一された表現の探求に導かれました。

Haskell の型クラスの根底にある qualified types [Jon94] に精通した人に対して、qualified type はガード型と見なせないことを指摘しておきます。その単純な理由は、applied type system におけるガードの証明が計算上の意味を持っていないことです。つまり、プログラムの実行時の挙動に影響を及ぼすことができないのです。しかし、qualified types の環境における型の述語の証明である dictionary はプログラムの実行時の挙動に影響を与えることができ、また好んで使われます。

別方向の関連研究として certified binaries [SSTP02] を用いた型システムの形式化があります。これもまた型とプログラムを完全に分離するアイデアです。帰納的な定義 (CiC) [PPM89,PM93] を拡張した calculus of constructions にこの型言語は基づいているとはいえ、基本的にこの型システムにおける型言語と計算言語の概念は、それぞれ ATS の statics と dynamics の概念に相当します。けれども、ATS における制約関係の概念は [SSTP02] のものに相当するわけではありません。その代わりに、型の正規形を比較することで、2 つの型の等価性を決定できます。私達が applied type system と関連した制約関係の証明を効果的に表現/検証するアプローチ持っている以上、[SSTP02] の意味での certify binaries に対して applied type system を構築することも困難ではありません。

要約すると、実用的なプログラミングをサポートするための型システムのデザインと形式化を促進するために、*ATS* フレームワークを示しました。statics と dynamics を完全に分離することで、*ATS* は作用の存在下における依存型のサポートに特に威力を発揮します。また、*ATS* のガード型とアサート型を使うことで、より柔軟により効果的にプログラムの不変条件を捕捉できますこれまで研究されてきた依存型 [XP99,Xi98] と guarded recursive datatypes [XCC03] の統一化と一般化であると *ATS* をとらえることもできます。

現時点では *ATS* の静的な要素は単純なラムダ計算に基づいています。従って、静的な要素が多相性と依存型をサポートする型付きラムダ計算の上に構築できるか研究することができるでしょう。また、*ATS* に基づく型システムを使って関数型プログラミング言語をデザイン/実装することに、私達は特に興味を持っています。それは既存の言語において新しい言語構造を実装ような方法で、言語拡張を行なう手段を提供できるでしょう。

謝辞 Assaf Kfoury から本論文のドラフトに対するコメントをもらいました。また、Chiyan Chen とは本論文の主題について議論しました。感謝します。

[*3] 実際には guarded recursive datatypes を、型が型をインデックスするような "依存型" として考えることができます。