Applied Type System[⋆]

(Extended Abstract)

Hongwei Xi

Boston University

Abstract. The framework Pure Type System ($\mathcal{PTS}$) offers a simple and general approach to designing and formalizing type systems. However, in the presence of dependent types, there often exist some acute problems that make it difficult for $\mathcal{PTS}$ to accommodate many common realistic programming features such as general recursion, recursive types, effects (e.g., exceptions, references, input/output), etc. In this paper, we propose a new framework Applied Type System ($\mathcal{ATS}$) to allow for designing and formalizing type systems that can readily support common realistic programming features. The key salient feature of $\mathcal{ATS}$ lies in a complete separation between statics, in which types are formed and reasoned about, and dynamics, in which programs are constructed and evaluated. With this separation, it is no longer possible for a program to occur in a type as is otherwise allowed in $\mathcal{PTS}$. We present not only a formal development of $\mathcal{ATS}$ but also mention some examples in support of using $\mathcal{ATS}$ as a framework to form type systems for practical programming.

1 Introduction

There is already a framework Pure Type System ($\mathcal{PTS}$) [Bar92] that offers a simple and general approach to designing and formalizing type systems. However, in the presence of dependent types, there often exist some acute problems that make it difficult for $\mathcal{PTS}$ to accommodate many common realistic programming features. In particular, we have learned that some great efforts are required in order to maintain a style of pure reasoning as is advocated in $\mathcal{PTS}$ when programming features such as general recursion [CS87], recursive types [Men87], effects [HMST95], exceptions [HN88] and input/output are present. To address such limitations of $\mathcal{PTS}$, we propose a new framework Applied Type System ($\mathcal{ATS}$) to allow for designing and formalizing type systems that can readily support common realistic programming features. The key salient feature of $\mathcal{ATS}$ lies in a complete separation between statics, in which types are formed and reasoned about, and dynamics, in which programs are constructed and evaluated. This separation, with its origin in a previous study on a restricted form of dependent types developed in Dependent ML (DML) [XP99,Xi98], makes it feasible to support dependent types in the presence of effects such as references

⋆ Partially supported by NSF grants no. CCR-0224244 and no. CCR-0229480

and exceptions. Also, with the introduction of two new (and thus unfamiliar) forms of types: *guarded types* and *asserting types*, we argue that $\mathcal{ATS}$ is able to capture program invariants in a more flexible and more effective manner than $\mathcal{PTS}$.

The design and formalization of $\mathcal{ATS}$ constitutes the primary contribution of the paper, which aims at setting a reference point for future work that makes use of similar ideas presented in [Zen97,XP99,XCC03]. With $\mathcal{ATS}$, we can readily form type systems to support many common programming features in the presence of dependent types, overcoming certain inherent deficiencies of $\mathcal{PTS}$. We are currently in the process of designing and implementing a typed functional programming language with its type system based on $\mathcal{ATS}$ that can support not only dependent types (like those developed DML) but also guarded recursive datatypes [XCC03]. With such a design, we seek to support a variety of language extensions by mostly implementing new language constructs in terms of existing ones, following an approach like the one adopted by Scheme. In particular, we have already shown that various programming features such as object-oriented programming [XCC03], meta-programming [XCC03,CX03] and type classes [XCC02] can be handled in such a manner.

We organize the rest of the extended abstract as follows. In Section 2, we present a detailed development of the framework $\mathcal{ATS}$, formalizing a generic applied type system ATS constructed in $\mathcal{ATS}$ and then establishing both subject reduction and progress theorems for ATS. We extend $\mathcal{ATS}$ in Section 3 to accommodate some common realistic programming features such as general recursion, pattern matching and effects, and present some interesting examples of applied type systems in Section 4. Lastly, we mention some related work as well as certain potential development for the future, and then conclude. A completed full paper is availabe on-line [Xi03] in which the missing details in this extended abstract can be found.

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

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

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

Fig. 1. The signature formating rules and the sorting rules for statics

2 Applied Type System

We present a formalization of the framework Applied Type System ($\mathcal{ATS}$) in this section. We use the name *applied type system* for a type system formed in the $\mathcal{ATS}$ framework. In the following presentation, let ATS be a generic applied type system, which consists of a static component (statics) and a dynamic component (dynamics). Intuitively, the statics and dynamics are each for handling types and programs, respectively. To simplify the presentation, we assume that the statics is a pure simply typed language and we use the name *sort* to refer to a type in this language. A term in the statics is called a *static term* while a term in the dynamics is called a *dynamic term*, and a static term of a special sort *type* serves as a type in the dynamics.

2.1 Statics

We present a formal description of a static component. We write b for a base sort and assume the existence of two special base sorts *type* and *bool*.

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

We use a for static term variables and s for static terms. There may also be some declared static constants sc, which are either static constant constructors scc or static constant functions scf. We use $[\sigma_1, \ldots, \sigma_n] \Rightarrow b$ for sc-sorts, which are assigned to static constants. Given a static constant sc, we can form a term $sc[s_1, \ldots, s_n]$ of sort b if sc is assigned a sc-sort $[\sigma_1, \ldots, \sigma_n] \Rightarrow b$ for some sorts $\sigma_1, \ldots, \sigma_n$ and s_i can be assigned the sorts σ_i for $i = 1, \ldots, n$. We may write sc for $sc[]$ if there is no risk of confusion. Note that a sc-sort is not regarded as a (regular) sort.

We use Θ_S for a static substitution that maps static variables to static terms and $\mathbf{dom}(\Theta_S)$ for the domain of Θ_S. We write $[]$ for the empty mapping and $\Theta_S[a \mapsto s]$, where we assume $a \notin \mathbf{dom}(\Theta_S)$, for the mapping that extends Θ_S with a link from a to s. Also, we write $\bullet[\Theta_S]$ for the result of applying Θ_S to some syntax $\bullet$, which may represent a static term, a sequence of static terms, or a dynamic variable context as is defined later.

A signature is for assigning sc-sorts to declared static constants sc, and the rules for forming signatures are in given Figure 1. We assume that the initial signature $\mathcal{S}_\emptyset$ contains the following declarations,

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

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

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

that is, the static constants on the left-hand side of : are assigned the corresponding sc-sorts on the right-hand side. Also, for each sort σ, we assume that $\mathcal{S}_\emptyset$ assigns the two static constructors $\forall_\sigma$ and $\exists_\sigma$ the sc-sort $[\sigma \rightarrow_{tp} type] \Rightarrow type$. We may use infix notation for some static constants. For instance, we write $s_1 \rightarrow_{tp} s_2$ for $\rightarrow_{tp} [s_1, s_2]$ and $s_1 \leq_{tp} s_2$ for $\leq_{tp} [s_1, s_2]$. In addition, we may write $\forall a : \sigma.s$ and $\exists a : \sigma.s$ for $\forall_\sigma[\lambda a : \sigma.s]$ and $\exists_\sigma[\lambda a : \sigma.s]$, respectively. The sorting rules for the statics are given in Figure 1, which are mostly standard. For instance, $\forall a : \mathit{type}.a \rightarrow_{tp} a$ is a static term that can be assigned the sort *type* since $\emptyset \vdash_{\mathcal{S}_\emptyset} \forall_{\mathit{type}}[\lambda a : \mathit{type}.a \rightarrow_{tp} a] : \mathit{type}$ is derivable. A static constructor *sc* is a type constructor if it is assigned a sc-sort $[\sigma_1, \ldots, \sigma_n] \Rightarrow \mathit{type}$ for some sorts $\sigma_1, \ldots, \sigma_n$. For instance, $\mathbf{1}$, $\rightarrow_{tp}$, $\supset$, $\wedge$, $\forall_\sigma$ and $\exists_\sigma$ are all type constructors, but $\leq_{tp}$ is not. Intuitively, $\mathbf{1}$ represents the usual unit type and $\rightarrow_{tp}$ forms function types, and $\leq_{tp}$ stands for a subtyping relation on types. The static constructors $\supset$ and $\wedge$ form guarded types and asserting types, respectively, which are to be explained later.

We use Σ for a static variable context that assigns sorts to static variables; $\mathbf{dom}(\Sigma)$ is the set of static variables declared in Σ; $\Sigma(a) = \sigma$ if $a : \sigma$ is declared in Σ. As usual, a static variable a may be declared at most once in Σ. A static term s is called a *proposition* under Σ if $\Sigma \vdash s : \mathit{bool}$ is derivable. We use P for propositions (under some static variable contexts). We use the name *guarded type* for a type of the form $P \supset s$ and the name *asserting type* for a type of the form $P \wedge s$, both of which are involved in the following example.

Example 1. Let *int* be the sort for integers[1] and **list** be a type constructor of the sc-sort $[\mathit{type}, \mathit{int}] \Rightarrow \mathit{type}$. Then the following static term is a type:

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

Intuitively, if $\mathbf{list}[s, n]$ is the type for lists of length n in which each element is of type s, then the above type is intended for a function from lists to lists that preserves list length. Also, the following type is intended to be assigned to a function that returns the tail of a given list if the list is not empty or simply raises an exception otherwise.

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

The asserting type $n > 0 \wedge \mathbf{list}[a, n-1]$ captures the invariant that $n > 0$ holds and the returned value is a list of length $n - 1$ *if the function returns* after it is applied to a list of length n. This is rather interesting feature and will be further explained later in Example 2. While there are already some traces of asserting types in the studies on Dependent ML [XP99,Xi98], the precise notion of asserting types has not be previously formalized: In DML, one must use subset sorts to simulate what we call asserting types here.

As in the design of $\mathcal{PTS}$, the issue of type equality plays a profound rôle in the design of $\mathcal{ATS}$. However, further study reveals that type equality in $\mathcal{ATS}$ can be defined in terms of a subtyping relation $\leq_{tp}$. Given two types s_1 and s_2, we say that s_1 equals s_2 if both the proposition $s_1 \leq_{tp} s_2$ and the proposition $s_2 \leq_{tp} s_1$ hold. In general, we need to determine whether a given proposition holds (under certain assumptions), and we introduce the following notion of constraint relation for this purpose.

[1] Formally speaking, we need to say that for each integer n, there is a static constructor $\underline{n}$ of the sc-sort $[] \Rightarrow \mathit{int}$ and $\underline{n}[]$ is the static term of the sort *int* that corresponds to n.

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

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

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

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

Fig. 2. Regularity Rules

Definition 1. *Let $\mathcal{S}, \Sigma, \vec{P}, P_0$ be a static signature, a static variable context, a set of propositions under Σ and a proposition under Σ, respectively. We say a relation $\Sigma; \vec{P} \models_{\mathcal{S}} P_0$ is a regular constraint relation if the following regularity conditions are satisfied:*

1. *all the regularity rules in Figure 2 are valid; that is, for each regularity rule, the conclusion of the rule holds if the premises of the rule hold, and*
2. *$\Sigma; \vec{P} \models_{\mathcal{S}} s_1 \rightarrow_{tp} s_2 \leq_{tp} s'_1 \rightarrow_{tp} s'_2$ implies $\Sigma; \vec{P} \models_{\mathcal{S}} s'_1 \leq_{tp} s_1$ and $\Sigma; \vec{P} \models_{\mathcal{S}} s_2 \leq_{tp} s'_2$, and*
3. *$\Sigma; \vec{P} \models_{\mathcal{S}} P \supset s \leq_{tp} P' \supset s'$ implies $\Sigma; \vec{P}, P' \models_{\mathcal{S}} P$ and $\Sigma; \vec{P}, P' \models_{\mathcal{S}} s \leq_{tp} s'$, and*
4. *$\Sigma; \vec{P} \models_{\mathcal{S}} P \wedge s \leq_{tp} P' \wedge s'$ implies $\Sigma; \vec{P}, P \models_{\mathcal{S}} P'$ and $\Sigma; \vec{P}, P \models_{\mathcal{S}} s \leq_{tp} s'$, and*
5. *$\Sigma; \vec{P} \models_{\mathcal{S}} \forall a : \sigma.s \leq_{tp} \forall a : \sigma.s'$ implies $\Sigma, a : \sigma; \vec{P} \models_{\mathcal{S}} s \leq_{tp} s'$, and*
6. *$\Sigma; \vec{P} \models_{\mathcal{S}} \exists a : \sigma.s \leq_{tp} \exists a : \sigma.s'$ implies $\Sigma, a : \sigma; \vec{P} \models_{\mathcal{S}} s \leq_{tp} s'$, and*
7. *$\emptyset; \emptyset \models_{\mathcal{S}} \mathit{scc}[s_1, \ldots, s_n] \leq_{tp} \mathit{scc}'[s'_1, \ldots, s'_{n'}]$ implies $\mathit{scc} = \mathit{scc}'$.*

Note that we assume $\Sigma \vdash_{\mathcal{S}} P : \mathit{bool}$ is derivable for each $P \in \vec{P}, P_0$ whenever we write $\Sigma; \vec{P} \models_{\mathcal{S}} P_0$.

We are in need of a regular constraint relation when forming the dynamics of ATS. Every single regularity rule as well as every single regularity condition is used later for establishing the subject reduction theorem (Theorem 1) and and the progress theorem (Theorem 2). In general, the framework $\mathcal{ATS}$ is parameterized over regular constraint relations. We need not be concerned with the decidability of a regular constraint relation at this point. For each regular constraint relation $\models_{\mathcal{S}}$, we may simply assume that an oracle is available to determine whether $\Sigma; \vec{P} \models_{\mathcal{S}} P_0$ holds whenever appropriate $\Sigma, \vec{P}$ and P_0 are given. Later, we will present some examples of applied type systems where there are practical algorithms for determining the regular constraint relations involved.

It should be emphasized that because of impredicativity, it is in general a rather delicate issue as to how a regular constraint relation $\models_{\mathcal{S}}$ can be properly defined for a given signature $\mathcal{S}$. In [Xi03], we have presented a model-theoretical approach to address this important issue.

2.2 Dynamics

The dynamics of ATS is a typed language and a static term of the sort *type* is a type in the dynamics. There may be some declared dynamic constants, and we are to assign a dc-type of the following form to each dynamic constant dc of arity n,

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

where $s_1, \ldots, s_n, s$ are assumed to be types. In the case where dc is a dynamic constructor dcc, the type s needs to be of the form $scc[\vec{s}]$ for some type constructor scc, and we say that dcc is associated with scc. Note that we use $\vec{s}$ for a (possibly empty) sequence of static terms. For instance, we can associate two dynamic constructors $\underline{nil}$ and $\underline{cons}$ with the type constructor **list** as follows by assigning them the following dc-types,

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

where we use $\mathbf{list}[a, n]$ as the type for lists of length n in which each element is of type a.

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

Fig. 3. The syntax for dynamics

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

Fig. 4. The formation rules for dynamic variable contexts

We use Θ_D for a dynamic substitution that maps dynamic variables to dynamic terms and $\mathbf{dom}(\Theta_D)$ for the domain of Θ_D. We omit presenting the syntax for forming and applying dynamic substitutions, which is similar to that for static substitutions. Given Θ_D^1 and Θ_D^2 such that $\mathbf{dom}(\Theta_D^1) \cap \mathbf{dom}(\Theta_D^2) = \emptyset$, we use $\Theta_D^1 \cup \Theta_D^2$ for the union of Θ_D^1 and Θ_D^2.

For $\Sigma = a_1 : \sigma_1, \ldots, a_k : \sigma_k$, we may write $\forall \Sigma.\bullet$ for $\forall a_1 : \sigma_1 \ldots \forall a_k : \sigma_k.\bullet$, where we simply use $\bullet$ for arbitrary syntax. Similarly, For $\vec{P} = P_1, \ldots, P_m$, we may use $\vec{P} \supset \bullet$ for $P_1 \supset (\ldots (P_m \supset \bullet) \ldots)$. For instance, a dc-type is always of the form $\forall \Sigma.\vec{P} \supset ([s_1, \ldots, s_n] \Rightarrow_{tp} s)$. The definition of signatures needs to be extended as follows to allow that dynamic constants be declared,

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

and the following additional rule is needed to form signatures.

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

The syntax for the dynamics is given in Figure 3, where we use x for dynamic term variables and d for dynamic terms. Given a dynamic constant dc of arity n, we write $dc[d_1, \ldots, d_n]$ for the application of dc to the arguments $d_1, \ldots, d_n$. In the case where $n = 0$, we may write dc for $dc[]$.

The markers $\supset^+(\cdot), \supset^-(\cdot), \wedge(\cdot), \forall^+(\cdot), \forall^-(\cdot), \exists(\cdot)$ are introduced to establish Lemma 3, which is needed for conducting inductive reasoning on typing derivations. Without these markers, it would be significantly more involved to establish proofs by induction on typing derivations as Lemma 3 can no longer be established as it is stated now.

A judgment of the form $\Sigma \vdash_{\mathcal{S}} \Delta\ [dctx]$ indicates that Δ is a well-formed dynamic variable context under Σ and $\mathcal{S}$. The rules for deriving such judgments are given in Figure 4. We use $\Sigma; \vec{P}; \Delta$ for a typing context. The following rule is for deriving a judgment of the form $\vdash_{\mathcal{S}} \Sigma; \vec{P}; \Delta$,

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

which indicates that $\Sigma; \vec{P}; \Delta$ is well-formed.

A typing judgment is of the form $\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$, where we assume that $\Sigma; \vec{P}; \Delta$ is a well-formed typing context and $\Sigma \vdash_{\mathcal{S}} s : type$ is derivable. The typing rules for deriving such judgments are presented in Figure 5, where we assume that the constraint relation $\models_{\mathcal{S}}$ is regular. We write $\Sigma \vdash_{\mathcal{S}} \Theta_S : \Sigma_0$ to mean that $\Sigma \vdash_{\mathcal{S}} \Theta_S(a) : \Sigma(a)$ is derivable for each $a \in \mathbf{dom}(\Theta_S) = \mathbf{dom}(\Sigma)$. Note that we have omitted some obvious side conditions associated with some of the typing rules. For instance, the variable a is not allowed to have free occurrences in $\vec{P}$, Δ, or s when the rule **(ty-∀-intro)** is applied. Also, we have imposed a form of value restriction on the typing rules **(ty-gua-intro)** and **(ty-∀-intro)**, preparing for introducing effects into ATS later.[2] For a technical reason, we are to replace the rule **(ty-var)** with the following rule,

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

which combines **(ty-var)** with **(ty-sub)**. This replacement is needed for establishing Lemma 2.

Before proceeding to the presentation of the rules for evaluating dynamic terms, we now sketch a scenario in which a guarded type and an asserting type

[2] Actually, it is already necessary to impose this form of value restriction on the typing rule **(ty-gua-intro)** in order to establish Theorem 2.

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 5. The typing rules for the dynamics

play an interesting role in enforcing security, facilitating further understanding of such types.

Example 2. Assume that $\underline{\mathit{Secret}}$ is a proposition constant and $\underline{\mathit{password}}$ and $\underline{\mathit{action}}$ are two declared functions, which are assigned the following dc-types.

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

The function $\underline{\mathit{password}}$ can be implemented in a manner so that some secret information must be verified before a call to $\underline{\mathit{password}}$ returns. On one hand, the proposition $\underline{\mathit{Secret}}$ needs to be established before the function call $\underline{\mathit{action}}[\langle\rangle]$ can be made, where $\langle\rangle$ denotes the value of the unit type $\mathbf{1}$. On the other hand, the proposition $\underline{\mathit{Secret}}$ is established after the function call $\underline{\mathit{password}}[\langle\rangle]$ returns. Therefore, a proper means to calling $\underline{\mathit{action}}$ is through the following program pattern:

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

In particular, a call to $\underline{action}$ outside the scope of x is ill-typed since the proposition $\underline{Secret}$ cannot be established.

In order to assign a call-by-value dynamic semantics to dynamic terms, we make use of evaluation contexts, which are defined below:

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

Definition 2. *We define redexes and their reductions as follows.*

- $\textbf{app}(\textbf{lam } x.d, v)$ *is a redex, and its reduction is* $d[x \mapsto v]$.
- $\supset^-(\supset^+(v))$ *is a redex, and its reduction is* v.
- $\textbf{let } \wedge(x) = \wedge(v) \textbf{ in } d$ *is a redex, and its reduction is* $d[x \mapsto v]$.
- $\forall^-(\forall^+(v))$ *is a redex, and its reduction is* v.
- $\textbf{let } \exists(x) = \exists(v) \textbf{ in } d$ *is a redex, and its reduction is* $d[x \mapsto v]$.
- $dcf[v_1, \ldots, v_n]$ *is a redex if* $dcf[v_1, \ldots, v_n]$ *is defined to equal some value* v*, and its reduction is* v.

Given two dynamic terms d_1 *and* d_2 *such that* $d_1 = E[d]$ *and* $d_2 = E[d']$ *for some redex* d *and its reduction* d'*, we write* $d_1 \hookrightarrow d_2$ *and say that* d_1 *reduces to* d_2 *in one step. We use* $\hookrightarrow^*$ *for the reflexive and transitive closure of* $\hookrightarrow$.

We assume that the type assgined to each dynamic constant function dcf is appropriate, that is, $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} v : s$ is derivable if $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} dcf[v_1, \ldots, v_n] : s$ is derivable and $dcf[v_1, \ldots, v_n] \hookrightarrow v$ holds.

Given a judgment J, we write $\mathcal{D} :: J$ to indicate that $\mathcal{D}$ is a derivation of J, that is, $\mathcal{D}$ is a derivation whose conclusion is J.

Lemma 1 (Substitution). *We have the following.*

1. *Assume* $\mathcal{D} :: \Sigma, a : \sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ *and* $\mathcal{D}_0 :: \Sigma \vdash_{\mathcal{S}} s_0 : \sigma$*. Then* $\Sigma; \vec{P}[a \mapsto s_0]; \Delta[a \mapsto s_0] \vdash_{\mathcal{S}} d : s[a \mapsto s_0]$ *is derivable.*
2. *Assume* $\mathcal{D} :: \Sigma; \vec{P}, P; \Delta \vdash_{\mathcal{S}} d : s$ *and* $\Sigma; \vec{P} \models_{\mathcal{S}} P$*. Then* $\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ *is derivable.*
3. *Assume* $\mathcal{D} :: \Sigma; \vec{P}; \Delta, x : s_1 \vdash_{\mathcal{S}} d_2 : s_2$ *and* $\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_1 : s_1$*. Then* $\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d_2[x \mapsto d_1] : s_2$ *is derivable.*

Proof. We can readily prove (1), (2) and (3) by structural induction on $\mathcal{D}$. When proving (1) and (2), we need to make use of the regularity rules **(reg-subst)** and **(reg-cut)**, respectively.

Given a derivation $\mathcal{D}$, we use $\mathbf{h}(\mathcal{D})$ for the height of $\mathcal{D}$, which can be defined in a standard manner.

Lemma 2. *Assume* $\mathcal{D} :: \Sigma; \vec{P}; \Delta, x : s_1 \vdash_{\mathcal{S}} d : s_2$ *and* $\Sigma; \vec{P} \models_{\mathcal{S}} s_1' \leq_{tp} s_1$*. Then there is a derivation* $\mathcal{D}' :: \Sigma; \vec{P}; \Delta, x : s_1' \vdash_{\mathcal{S}} d : s_2$ *such that* $\mathbf{h}(\mathcal{D}') = \mathbf{h}(\mathcal{D})$.

Proof. The proof follows from structural induction on $\mathcal{D}$ immediately. The regularity rule **(reg-trans)** is used to handle the case where the last applied rule in $\mathcal{D}$ is **(ty-var')**.

The following inversion is slightly different from a standard one because of the existence of the rule **(tyrule-eq)**.

Lemma 3 (Inversion). *Assume $\mathcal{D} :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$.*

1. *If $d = \mathbf{lam}\ x.d_1$ and $s = s_1 \rightarrow_{tp} s_2$, then there is a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ such that $\mathbf{h}(\mathcal{D}') \leq \mathbf{h}(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not* **(ty-sub)**.
2. *If $d = \supset^+(d_1)$ and $s = P \supset s_1$, then there is a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ such that $\mathbf{h}(\mathcal{D}') \leq \mathbf{h}(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not* **(ty-sub)**.
3. *If $d = \wedge(d_1)$ and $s = P \wedge s_1$, then there is a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ such that $\mathbf{h}(\mathcal{D}') \leq \mathbf{h}(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not* **(ty-sub)**.
4. *If $d = \forall^+(d_1)$ and $s = \forall a : \sigma.s_1$, then there is a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ such that $\mathbf{h}(\mathcal{D}') \leq \mathbf{h}(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not* **(ty-sub)**.
5. *If $d = \exists(d_1)$ and $s = \exists a : \sigma.s_1$, then there is a derivation $\mathcal{D}' :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ such that $\mathbf{h}(\mathcal{D}') \leq \mathbf{h}(\mathcal{D})$ and the last rule applied in $\mathcal{D}'$ is not* **(ty-sub)**.

Proof. By induction by $\mathbf{h}(\mathcal{D})$. In particular, Lemma 2 is needed to establish (1).

The type soundess of ATS rests upon the following two theorems, who proofs are largely standard and thus omitted here.

Theorem 1 (Subject Reduction). *Assume both $\mathcal{D} :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ and $d \hookrightarrow d'$. Then $\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} d : s$ is derivable.*

Theorem 2 (Progress). *Assume $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} d : s$. Then d is a value, or $d \hookrightarrow d'$ holds for some dynamic term d', or $d = E[dcf(v_1, \ldots, v_n)]$ for some dynamic term $dcf(v_1, \ldots, v_n)$ that is not a redex.*

2.3 Erasure

We present a function from dynamic terms to untyped λ-expressions that preserves semantics. We use e for the erasures of dynamic terms, which are formally defined as follows:

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

We can then define a function $|\cdot|$ as follows that translates dynamic terms into erasures.

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

Similar to assigning dynamic semantics to the dynamic terms, we can readily assign dynamic semantics to the erasures, which are just untyped λ-expressions. We write $e_1 \hookrightarrow e_2$ to mean that e_1 reduces to e_2 in one step, and use $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$.

Theorem 3. *Assume $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} d : s$.*

1. *If $d \hookrightarrow^* v$, then $|d| \hookrightarrow^* |v|$.*
2. *If $|d| \hookrightarrow^* w$, then there is a value v such that $d \hookrightarrow^* v$ and $|v| = w$.*

Proof. (1) is straightforward and (2) follows from structural induction on $\mathcal{D}$.

With Theorem 3, we can evaluate a dynamic term d by simply evaluating the erasure of d.

3 Extensions

We extend $\mathcal{ATS}$ to accommodate some common realistic programming features in this section.

General Recursion We introduce a fixed-point operator **fix** to support general recursion in $\mathcal{ATS}$. We now call variables x **lam**-variables and introduce **fix**-variables f. We use xf for a variable that is either a **lam**-variable or a **fix**-variable.

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

The rule **(ty-var)** needs to be modified and the rule **(tyrule-fix)** needs to be added to handle the fixed-point operator:

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

A dynamic term of the form $\mathbf{fix}\, f.d$ is a redex and its reduction is $d[f \mapsto \mathbf{fix}\, f.d]$. It is straightforward to establish both the subject reduction theorem (Theorem 1) and the progress theorem (Theorem 2) for this extension.

Datatypes and Pattern Matching We present an approach to extending $\mathcal{ATS}$ with support for datatypes and pattern matching and then provide with some simple examples. The following is some additional syntax we need.

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

As usual, we require that any variable x can occur at most once in a pattern. Given a value v and a pattern p, we use a judgment of the form $v \Downarrow p \Rightarrow \Theta_D$ to indicate $v = p[\Theta_D]$. The rules for deriving such judgments are given as follows,

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

and we say that v matches p if $v \Downarrow p \Rightarrow \Theta_D$ is derivable for some dynamic substitution Θ_D. Note that in the rule **(vp-dcc)**, the union $\Theta_D^1 \cup \ldots \cup \Theta_D^n$, which becomes the empty dynamic substitution [] when $n = 0$, is well-defined since any variable can occur at most once in a pattern.

A dynamic term of the form **case** v **of** $p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n$ is a redex if $v \Downarrow p_i \Rightarrow \Theta_D$ holds for some $1 \leq i \leq n$, and its reduction is $d_i[\Theta_D]$. Note that reducing such a redex may involve nondeterminism if v matches several patterns p_i.

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

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

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

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

Fig. 6. The typing rules for pattern matching

The typing rules for pattern matching is given in Figure 6. The meaning of a judgment of the form $\Sigma \vdash p \Downarrow s \Rightarrow \Sigma'; \vec{P}'; \Delta'$ is formally captured in the following lemma.

Lemma 4. *Assume $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} v : s$, $\mathcal{E}_1 :: \emptyset \vdash p \Downarrow s \vdash \Sigma; \vec{P}; \Delta$ and $\mathcal{E}_2 :: v \Downarrow p \Rightarrow \Theta_D$. Then there exists $\Theta_S : \Sigma$ such that $\emptyset; \emptyset \models_{\mathcal{S}} P[\Theta_S]$ for each P in $\vec{P}$ and $\emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} \Theta_D : \Delta$.*

Proof. The lemma follows from structural induction on $\mathcal{E}_1$.

As an example, the judgment below is derivable,

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

where $\underline{cons}$ is assigned the following dc-type,

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

and $\Sigma = (a : type, n : int)$, $\vec{P} = (n \geq 0, \textbf{list}[a, n+1] \leq_{tp} \textbf{list}[a', n'])$ and $\Delta = (x_1 : a, x_2 : \textbf{list}[a, n])$.

We can readily prove the subject reduction theorem (Theorem 1) for this extension: Lemma 4 is needed to handle the case where the reduced index is of the following form:

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

Also, we can establish the progress theorem (Theorem 2) for this extension after slightly modifying it to include the possibility that a well-type program d may be of the following form,

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

where v_0 does not match any p_i for $1 \leq i \leq n$ if d is neither a value nor can be further reduced.

Effects Unlike $\mathcal{PTS}$, $\mathcal{ATS}$ can be extended in a straightforward manner to accommodate effects such as references and exceptions. For instance, to introduce references into $\mathcal{ATS}$, we can simply declare a type constructor *ref* of the sc-sort $[type] \Rightarrow type$ and then the following dynamic functions of the corresponding assigned dc-types.

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

The intended meaning of these functions should be obvious. We also need to add into Definition 1 the following regularity condition to address the issue of *ref* being an invariant type constructor.

– $\Sigma; \vec{P} \models_{\mathcal{S}} ref(s) \leq_{tp} ref(s')$ implies $\Sigma; \vec{P} \models_{\mathcal{S}} s \leq_{tp} s'$ and $\Sigma; \vec{P} \models_{\mathcal{S}} s' \leq_{tp} s$.

It is a standard procedure to assign dynamic semantics to this extension and then establish both the subject reduction theorem and the progress theorem. Please see [Har94] for some details on such a procedure.

It is straightforward as well to introduce exceptions into $\mathcal{ATS}$, and we omit further details.

4 Examples of Applied Type Systems

Unsurprizingly, it can be readily shown that the systems λ_2 and λ_ω in λ-cube [Bar92] are applied type systems. Also, the language $\lambda_{G\mu}$ [XCC03], which extends λ_2 with guarded recursive datatypes, and Dependent ML [XP99] are applied type systems. Please see [Xi03] for more detailed explanation.

5 Related Work and Conclusion

The framework $\mathcal{ATS}$ is rooted in the work on Dependent ML [XP99,Xi98], where the type system of ML is enriched with a restricted form of dependent datatypes, and the recent work on guarded recursive datatypes [XCC03]. Given the similarity between these two forms of types[3], we are naturally led to seeking a unified presentation for them.

[3] Actually, guarded recursive datatypes may be thought of as "dependent types" in which the type indexes are also types.

For those who are familiar with qualified types [Jon94], which underlies the type class mechanism in Haskell, we point out that a qualified type can *not* be regarded as a guarded type. The simple reason is that the proof of a guard in an applied type system bears no computational meaning, that is, it cannot affect the run-time behavior of a program, while a dictionary, which is really the proof of some predicate on types in the setting of qualified types, can and is mostly likely to affect the run-time behaviour of a program.

Another line of closely related work is the formation of a type system in support of certified binaries [SSTP02], in which the idea of a complete separation between types and programs is also employed. Basically, the notions of type language and computational language in the type system correspond to the notions of statics and dynamics in $\mathcal{ATS}$, respectively, though the type language is based on the calculus of constructions extended with inductive definitions (CiC) [PPM89,PM93]. However, the notion of a constraint relation in $\mathcal{ATS}$ does not have a counterpart in [SSTP02]. Instead, the equality between two types is determined by comparing the normal forms of these types. It is not difficult to see that an applied type system can also be constructed to certify binaries in the sense of [SSTP02] as long as we have an approach to effectively representing and verifying proofs of the constraint relation associated with the applied type system.

In summary, we have presented a framework $\mathcal{ATS}$ for facilitating the design and formalization of type systems to support practical programming. With a complete separation between statics and dynamics, $\mathcal{ATS}$ works particularly well on supporting dependent types in the presence of effects. Also, the availability of guarded types and asserting types in $\mathcal{ATS}$ makes it both more flexible and more effective to capture program invariants. We also see $\mathcal{ATS}$ as a unification as well as a generalization of the previous work on a restricted form of dependent types [XP99,Xi98] and guarded recursive datatypes [XCC03].

A static component in $\mathcal{ATS}$ is currently based on a simply typed λ-calculus. Therefore, it is natural to study how a static component can be built upon a typed λ-calculus supporting polymorphism and/or dependent types. Also, we are particularly interested in designing and implementing a functional programming language with a type system based on $\mathcal{ATS}$, which can then offer a means to language extension by mostly implementing new language constructs in terms of some existing ones.

Acknowledgments The author thanks Assaf Kfoury for his comments on a preliminary draft of the paper and also acknowledges some discussions with Chiyan Chen on the subject of the paper.

References

[Bar92] Hendrik Pieter Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–441. Clarendon Press, Oxford, 1992.

[CS87] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Proceedings of Symposium on Logic in Computer Science*, pages 183–193. Ithaca, New York, June 1987.

[CX03] Chiyan Chen and Hongwei Xi. Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180. Uppsala, Sweden, August 2003.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[HMST95] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 15 May 1995.

[HN88] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. The MIT Press, 1988.

[Jon94] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, November 1994.

[Men87] N.P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of Symposium on Logic in Computer Science*, pages 30–36. The Computer Society of the IEEE, Ithaca, New York, June 1987.

[PM93] Christine Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. In M. Bezem and J.F. de Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Utrecht, The Netherlands, 1993.

[PPM89] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of fifth International Conference on Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228, 1989.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A Type System for Certified Binaries. In *Proceedings of 29th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '02)*, pages 217–232. Portland, OR, January 2002.

[XCC02] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors, 2002. Available at `http://www.cs.bu.edu/~hwxi/GRecTypecon/`.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235. New Orleans, January 2003.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. viii+181 pp. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[Xi03] Hongwei Xi. Applied Type System, July 2003. Available at: `http://www.cs.bu.edu/~hwxi/ATS/ATS.ps`.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. San Antonio, Texas, January 1999.

[Zen97] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.