Applied Type System*

Hongwei Xi

Computer Science Department
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 some examples in support of using $\mathcal{ATS}$ as a framework to form type systems for practical programming.

Keywords

Applied Type System, ATS, guarded types, asserting types

1 Introduction

There is often a serious contention between the type system of a programming language and the need to extend the programming language. For instance, when the issue of language extension is concerned, we can readily observe a striking difference between Scheme, an untyped functional programming language, and ML (Milner et al., 1997), a typed functional programming language. Assume that we are interested in supporting object-oriented programming (OOP). In Scheme, we can achieve this by implementing a package like Common Lisp Object System (CLOS) to directly support OOP constructs, which, to a large extent, act like syntactic sugar as programs that make use of them can be systematically translated into programs that do not. On the other hand, it does not seem likely to support OOP in ML in a similar manner as the type system of ML is too restrictive to allow for a typed implementation of a package like CLOS. Instead, we need to add directly into ML some OOP constructs, which are then treated as primitives. This is rather unpleasant since the newly added constructs not only complicate the semantics of ML significantly but may also result in unexpected interaction with some existing features in ML. This situation of programming language extension stays more or less the same when many other programming features need to be supported. For instance, Standard ML (SML), which extends ML with a module system, and MetaML, which extends ML with some meta-programming constructs, are both greatly more involved than ML. Evidently such an unpleasant situation is primarily caused by the limited expressiveness of a type system like that of ML. Therefore, we are naturally led to seeking more expressive type systems that can better accommodate the issue of programming language extension.

There is already a framework Pure Type System ($\mathcal{PTS}$) (Barendregt, 1992) 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 (Constable and Smith, 1987), recursive types (Mendler, 1987), effects (Honsell et al., 1995), exceptions (Hayashi and Nakano, 1988) 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) (Xi and Pfenning, 1999; Xi, 1998), makes it feasible to support dependent types in the presence of effects such as references 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 (Zenger, 1997; Xi and Pfenning, 1999; Xi et al., 2003). 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 (Xi et al., 2003). With such a design, we seek to support a variety of language extensions by mostly implementing new language con-

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

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

Figure 1. The formation rules for signatures

$$\frac{\mathcal{S}(sc) = [\sigma_1, \ldots, \sigma_n] \Rightarrow b \quad \Sigma \vdash_{\mathcal{S}} s_i : \sigma_i \text{ for } i = 1, \ldots, n}{\Sigma \vdash_{\mathcal{S}} sc[s_1, \ldots, s_n] : b} \textbf{ (so-sc)}$$
$$\frac{\Sigma(a) = \sigma}{\Sigma \vdash_{\mathcal{S}} a : \sigma} \textbf{ (so-var)}$$
$$\frac{\Sigma, a : \sigma_1 \vdash_{\mathcal{S}} s : \sigma_2}{\Sigma \vdash_{\mathcal{S}} \lambda a : \sigma_1 . s : \sigma_1 \to \sigma_2} \textbf{ (so-lam)}$$
$$\frac{\Sigma \vdash_{\mathcal{S}} s_1 : \sigma_1 \to \sigma_2 \quad \Sigma \vdash_{\mathcal{S}} s_2 : \sigma_1}{\Sigma \vdash_{\mathcal{S}} s_1(s_2) : \sigma_2} \textbf{ (so-app)}$$

Figure 2. The sorting rules for statics

structs 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 (Xi et al., 2003), meta-programming (Xi et al., 2003; Chen and Xi, 2003) and type classes (Xi et al., 2002) can be handled in such a manner.

We organize the rest of the paper 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.

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

$$\begin{array}{rcl} \mathbf{1} & : & [] \Rightarrow type \\ \top & : & [] \Rightarrow bool \\ \bot & : & [] \Rightarrow bool \\ \to_{tp} & : & [type, type] \Rightarrow type \\ \supset & : & [bool, type] \Rightarrow type \\ \wedge & : & [bool, type] \Rightarrow type \\ \leq_{tp} & : & [type, type] \Rightarrow bool \end{array}$$

that is, the static constants on the left are assigned the corresponding sc-sorts on the right. Also, for each sort σ, we assume that $\mathcal{S}_\emptyset$ assigns the two static constructors $\forall_\sigma$ and $\exists_\sigma$ the sc-sort $[\sigma \to type] \Rightarrow type$. We may use infix notation for some static constants. For instance, we write $s_1 \to_{tp} s_2$ for $\to_{tp} [s_1, s_2]$ 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 2, which are mostly standard. For instance, $\forall a : type . a \to_{tp} a$ is a static term that can be assigned the sort *type* since $\emptyset \vdash_{\mathcal{S}_\emptyset} \forall_{type}[\lambda a : type . a \to_{tp} a] : type$ is derivable. A static constructor *sc* is a type constructor if it is assigned a sc-sort $[\sigma_1, \ldots, \sigma_n] \Rightarrow type$ for some sorts $\sigma_1, \ldots, \sigma_n$. For instance, $\mathbf{1}$, $\to_{tp}$, $\supset$, $\wedge$, $\forall_\sigma$ and $\exists_\sigma$ are all type constructors, but $\leq_{tp}$ is not. Intuitively, $\mathbf{1}$ represents the usual unit type and $\to_{tp}$ 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 $\Sigma \vdash s : bool$ 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.

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

$$\frac{\Sigma;\vec{P}\models\bot}{\Sigma;\vec{P}\models_{\mathcal{S}}P}\ \textbf{(reg-false)}$$

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

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

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

$$\frac{\Sigma;\vec{P}\models_{\mathcal{S}}P_0\quad\Sigma;\vec{P},P_0\models_{\mathcal{S}}P}{\Sigma;\vec{P}\models_{\mathcal{S}}P}\ \textbf{(reg-cut)}$$

$$\frac{\Sigma\vdash_{\mathcal{S}}s:type}{\Sigma;\vec{P}\models_{\mathcal{S}}s\leq_{tp}s}\ \textbf{(reg-refl)}$$

$$\frac{\Sigma;\vec{P}\models_{\mathcal{S}}s_1\leq_{tp}s_2\quad\Sigma;\vec{P}\models_{\mathcal{S}}s_2\leq_{tp}s_3}{\Sigma;\vec{P}\models_{\mathcal{S}}s_1\leq_{tp}s_3}\ \textbf{(reg-tran)}$$

Figure 3. Regularity Rules

EXAMPLE 2.1. *Let int be the sort for integers*[1] *and* **list** *be a type constructor of the sc-sort* $[type,int]\Rightarrow type$. *Then the following static term is a type:*

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

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.*

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

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 2.5.*

As is 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.

DEFINITION 2.2. *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 3 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. $\emptyset;\emptyset\models_{\mathcal{S}}scc[s_1,\ldots,s_n]\leq_{tp}scc'[s_1',\ldots,s_{n'}']$ *implies* $scc=scc'$.

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

Every single regularity rule as well as every single regularity condition is used later for establishing the subject reduction theorem 2.10 and and the progress theorem 2.12.

EXAMPLE 2.3. *Let* **int** *be a type constructor of the sc-sort* $[int]\Rightarrow type$ *such that for each integer n,* $\mathbf{int}[n]$ *is the singleton type containing only n. Intuitively, the types* $\mathbf{Int}=\exists n:int.\top\wedge\mathbf{int}[n]$ *and* $\mathbf{Nat}=\exists n:int.n\geq 0\wedge\mathbf{int}[n]$ *are for integers and natural numbers, respectively. Note that given a regular constraint relation* $\models_{\mathcal{S}}$ *(for some properly chosen* $\mathcal{S}$*), it is not necessary true that* $\emptyset;\emptyset\vdash\mathbf{Nat}\leq_{tp}\mathbf{Int}$ *holds. On the other hand, the regularity condition (6) can rule out a case like* $\emptyset;\emptyset\vdash\mathbf{Int}\leq_{tp}\mathbf{Nat}$ *(if* $\geq$ *is given the standard interpretation).*

We are in need of a regular constraint relation when forming the dynamics of ATS. 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 $\mathcal{S}$, and we are to address this issue with a model-theoretical approach in Section 2.2.

In the following presentation, we use STATICS for a static component and STATICS$(\mathcal{B};\mathcal{S})$ for a static component in which the set of base sorts is $\mathcal{B}$ and the signature is $\mathcal{S}$.

2.2 Models for Statics

We present a means to defining regular constraint relations through the use of models for statics.

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

Let STATICS be a static component and SORT be the set of sorts σ in STATICS. A frame is a collection $\{\mathbf{D}_\sigma\}_{\sigma\in \text{Sort}}$ of nonempty domains (sets) $\mathbf{D}_\sigma$, one for each sort σ such that $\mathbf{D}_{bool} = \{\mathbf{tt}, \mathbf{ff}\}$ and $\mathbf{D}_{\sigma_1 \to_{tp} \sigma_2}$ is some collection of functions that map $\mathbf{D}_{\sigma_1}$ into $\mathbf{D}_{\sigma_2}$. An interpretation $\langle \{\mathbf{D}_\sigma\}_{\sigma\in \text{sort}}, \mathbf{I}\rangle$ of a given signature $\mathcal{S}$ consists of a frame and a function $\mathbf{I}$ that maps each static constant *sc* of sc-sort $[\sigma_1, \ldots, \sigma_n] \Rightarrow b$, where *b* stands for a base sort, to a function $\mathbf{I}(sc)$ from $\mathbf{D}_{\sigma_1} \times \ldots \times \mathbf{D}_{\sigma_n}$ into $\mathbf{D}_b$ (or an element in $\mathbf{D}_b$ if $n = 0$). In addition, we require

1. $\mathbf{I}(\top) = \mathbf{tt}$ and $\mathbf{I}(\bot) = \mathbf{ff}$, and
2. $\mathbf{I}(scc)$ is a one-to-one function for each static constructor *scc*, and
3. the intersection of the domains of $\mathbf{I}(scc_1)$ and $\mathbf{I}(scc_2)$ is empty if scc_1 and scc_2 are two distinct type constructors.

Note that the approach we use here is adopted from Chapter 5 (Andrews, 1986), where models for (simple) type theory (Church, 1940) are studied.

Clearly, if $\mathbf{D}_\sigma$ contain more than one elements, then $\mathbf{D}_{type \to_{tp} type}$ cannot contain *all* functions from $\mathbf{D}_{type}$ to $\mathbf{D}_{type}$ since otherwise neither $\mathbf{I}(\forall_{type})$ nor $\mathbf{I}(\exists_{type})$ can be an injection from $\mathbf{D}_{type \to_{tp} type}$ into $\mathbf{D}_{type}$.

An assignment ϕ is a finite mapping from static variables to $\mathbf{D} = \cup_{\sigma\in \text{sort}} \mathbf{D}_\sigma$, and we use $\mathbf{dom}(\phi)$ for the domain of ϕ. As usual, we use $[]$ for the empty mapping and $\phi[a \mapsto \mathbf{a}]$ for the mapping that extends ϕ with a link from the variable *a* to the element $\mathbf{a}$ in $\mathbf{D}$, where we assume $a \notin \mathbf{dom}(\phi)$. We write $\phi : \Sigma$ if $\phi(a) \in \mathbf{D}_\sigma$ for each $a \in \mathbf{dom}(\phi) = \mathbf{dom}(\Sigma)$, where $\sigma = \Sigma(a)$, that is, $a : \sigma$ is declared in Σ.

Let $\mathcal{S}$ be the signature for the static constants in STATICS. An interpretation $\mathcal{M} = \langle \{\mathbf{D}_\sigma\}_{\sigma\in \text{sort}}, \mathbf{I}\rangle$ of $\mathcal{S}$ is a model for STATICS if there exists a (partial) binary function $\mathbf{V}^{\mathcal{M}}$ such that for each assignment $\phi : \Sigma$, $\mathbf{V}^{\mathcal{M}}(\phi, s)$ is defined for a static term *s* and $\mathbf{V}^{\mathcal{M}}(s) \in \mathbf{D}_\sigma$ holds whenever $\Sigma \vdash_{\mathcal{S}} s : \sigma$ is derivable for some σ, and the following conditions are also satisfied

1. $\mathbf{V}^{\mathcal{M}}(\phi, a) = \phi(a)$ for each $a \in \mathbf{dom}(\phi)$, and
2. $\mathbf{V}^{\mathcal{M}}(sc[s_1, \ldots, s_n]) = I(sc)(\mathbf{V}^{\mathcal{M}}(s_1), \ldots, \mathbf{V}^{\mathcal{M}}(s_n))$, and
3. $\mathbf{V}^{\mathcal{M}}(\phi, s_1(s_2)) = \mathbf{V}^{\mathcal{M}}(\phi, s_1)(\mathbf{V}^{\mathcal{M}}(\phi, s_2))$ whenever $\Sigma \vdash_{\mathcal{S}} s_1(s_2) : \sigma$ is derivable for some σ, and
4. $\mathbf{V}^{\mathcal{M}}(\phi, \lambda a : \sigma_1.s)$ is the function that maps each element $\mathbf{a} \in \mathbf{D}_{\sigma_1}$ to $\mathbf{V}^{\mathcal{M}}(\phi[a \mapsto \mathbf{a}], s)$ whenever $\Sigma \vdash \lambda a : \sigma_1.s : \sigma_1 \to_{tp} \sigma_2$ is derivable for some σ_2.

Note that not all frames belong to interpretations, and not all interpretations are models (Andrews, 1972). Given a model $\mathcal{M}$ for STATICS, we can define a constraint relation $\models^{\mathcal{M}}_{\mathcal{S}}$ as follows: $\Sigma; \vec{P} \models^{\mathcal{M}}_{\mathcal{S}} P_0$ holds if and only if for each assignment $\phi : \Sigma$, $\mathbf{V}^{\mathcal{M}}(\phi, P_0) = \mathbf{tt}$ or $\mathbf{V}^{\mathcal{M}}(\phi, P) = \mathbf{ff}$ for some $P \in \vec{P}$.

THEOREM 2.4. *Let a $\mathcal{M} = \langle \{\mathbf{D}_\sigma\}_{\sigma\in \text{sort}}, \mathbf{I}\rangle$ be a model for* STATICS. *If $\leq_{tp}$ is interpreted as the equality function on $\mathbf{D}_{type} \times \mathbf{D}_{type}$, that is, for all $\mathbf{a}, \mathbf{a}' \in \mathbf{D}_{type}$, $\mathbf{I}(\leq_{tp})(\mathbf{a}, \mathbf{a}') = \mathbf{tt}$ if and only if $\mathbf{a} = \mathbf{a}'$, then $\models^{\mathcal{M}}_{\mathcal{S}}$ is a regular constraint relation.*

PROOF. It is a simple routine to verify that $\models^{\mathcal{M}}_{\mathcal{S}}$ satisfies all the regularity conditions in Definition 2.2. □

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

Figure 4. The syntax for dynamics

$$\frac{\vdash \mathcal{S}\ [sig]}{\Sigma \vdash_{\mathcal{S}} \emptyset\ [dctx]}$$

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

Figure 5. The formation rules for dynamic variable contexts

2.3 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 *nil* and *cons* with the type constructor **list** as follows by assigning them the following dc-types,

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

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

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^1_D and Θ^2_D such that $\mathbf{dom}(\Theta^1_D) \cap \mathbf{dom}(\Theta^2_D) = \emptyset$, we use $\Theta^1_D \cup \Theta^2_D$ for the union of Θ^1_D and Θ^2_D.

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,

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.

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

The syntax for the dynamics is given in Figure 4, 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 2.9, 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 2.9 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 5. 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 6, 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-$\forall$-intro)** is applied. Also, we have imposed a form of value restriction on the typing rules **(ty-gua-intro)** and **(ty-$\forall$-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,

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

which combines **(ty-var)** with **(ty-sub)**. This replacement is needed for establishing Lemma 2.8.

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

EXAMPLE 2.5. *Assume that* Secret *is a proposition constant and* password *and* action *are two declared functions, which are assigned the following dc-types.*

$$\begin{array}{rcl} \underline{action} & : & \underline{Secret} \supset [\mathbf{1}] \Rightarrow_{tp} \mathbf{1} \\ \underline{password} & : & [\mathbf{1}] \Rightarrow_{tp} \underline{Secret} \wedge \mathbf{1} \end{array}$$

[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 2.12.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6. The typing rules for the dynamics

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

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

In particular, a call to action *outside the scope of* x *is ill-typed since the proposition* 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:

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

DEFINITION 2.6. *We define redexes and their reductions as follows.*

- $\mathbf{app}(\mathbf{lam}\ x.d, v)$ *is a redex, and its reduction is* $d[x \mapsto v]$.
- $\supset^-(\supset^+(v))$ *is a redex, and its reduction is* v.
- $\mathbf{let}\ \wedge(x) = \wedge(v)\ \mathbf{in}\ d$ *is a redex, and its reduction is* $d[x \mapsto v]$.
- $\forall^-(\forall^+(v))$ *is a redex, and its reduction is* v.
- $\mathbf{let}\ \exists(x) = \exists(v)\ \mathbf{in}\ d$ *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 2.7 (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 2.8. *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 needed to handle the case where the last applied rule in $\mathcal{D}$ is **(ty-var')**. □

LEMMA 2.9 (INVERSION). *Assume* $\mathcal{D} :: \Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d : s$.

1. *If* $d = \mathbf{lam}\ x.d_1$ *and* $s = s_1 \to_{tp} s_2$*, 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. We prove (1) by induction on $\mathbf{h}(\mathcal{D})$. Let $\mathcal{D}'$ be $\mathcal{D}$ if $\mathcal{D}$ does not end with an application of the rule **(ty-sub)**. Hence, in the rest of the proof, we assume that the last applied rule in $\mathcal{D}$ is **(ty-sub)**, that is, $\mathcal{D}$ is of the following form:

$$\frac{\mathcal{D}_1 :: \Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d : s' \quad \Sigma;\vec{P} \models_{\mathcal{S}} s' \leq_{tp} s}{\Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d : s}\ \textbf{(ty-sub)}$$

By induction hypothesis, we have a derivation $\mathcal{D}_1' :: \Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d : s'$ such that $\mathbf{h}(\mathcal{D}_1') \leq \mathbf{h}(\mathcal{D}_1)$ and the last applied rule in $\mathcal{D}_1'$ is not **(ty-sub)**. Hence, $\mathcal{D}_1'$ is of the following form:

$$\frac{\mathcal{D}_2' :: \Sigma;\vec{P};\Delta, x : s_1' \vdash_{\mathcal{S}} d_1 : s_2'}{\Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} \mathbf{lam}\ x.d_1 : s_1' \to_{tp} s_2'}\ \textbf{(ty-lam)}$$

where $s' = s_1' \to_{tp} s_2'$ and $d = \mathbf{lam}\ x.d_1$. Since $\Sigma;\vec{P} \models_{\mathcal{S}} s_1' \to_{tp} s_2' \leq_{tp} s_1 \to_{tp} s_2$, we have $\Sigma;\vec{P} \models_{\mathcal{S}} s_1 \leq_{tp} s_1'$ and $\Sigma;\vec{P} \models_{\mathcal{S}} s_2' \leq_{tp} s_2$ by Definition 2.2. Hence, by Lemma 2.8, there is a derivation $\mathcal{D}_2'' :: \Sigma;\vec{P};\Delta, x : s_1 \vdash_{\mathcal{S}} d_1 : s_2'$ such that $\mathbf{h}(\mathcal{D}_2'') = \mathbf{h}(\mathcal{D}_2')$. Let $\mathcal{D}'$ be the following derivation,

$$\frac{\dfrac{\mathcal{D}_2'' :: \Sigma;\vec{P};\Delta, x : s_1 \vdash_{\mathcal{S}} d_1 : s_2' \quad \Sigma;\vec{P} \models_{\mathcal{S}} s_2' \leq_{tp} s_2}{\Sigma;\vec{P};\Delta, x : s_1 \vdash_{\mathcal{S}} d_1 : s_2}\ \textbf{(ty-sub)}}{\Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} \mathbf{lam}\ x.d_1 : s_1 \to_{tp} s_2}\ \textbf{(ty-lam)}$$

and we are done since $\mathbf{h}(\mathcal{D}') = 1+1+\mathbf{h}(\mathcal{D}_2'') = 1+\mathbf{h}(\mathcal{D}_2') \leq 1+\mathbf{h}(\mathcal{D}_1) = \mathbf{h}(\mathcal{D})$. We can prove (2), (3), (4) and (5) similarly. □

THEOREM 2.10 (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.*

PROOF. Assume $d = E[d_0]$ and $d' = E[d_0']$, where d_0 is a redex and d_0' is the reduction of the redex. The proof proceeds by structural induction on $\mathbf{h}(\mathcal{D})$. Assume that the last applied rule in $\mathcal{D}$ is **(ty-sub)**, that is, $\mathcal{D}$ is of the following form:

$$\frac{\mathcal{D}_1 :: \Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d : s_1 \quad \Sigma;\vec{P} \models_{\mathcal{S}} s_1 \leq_{tp} s}{\mathcal{D}_1 :: \Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d : s}\ \textbf{(ty-sub)}$$

By induction hypothesis on $\mathcal{D}_1$, $\Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d' : s_1$ is derivable and therefore $\Sigma;\vec{P};\Delta \vdash_{\mathcal{S}} d' : s$ is derivable. From now on, we assume that the last applied rule in $\mathcal{D}$ is not **(ty-sub)**, and proceed by structural induction on E. We present the most interesting case where $E = []$. In this case, $d = d_0$ is a redex and $d' = d_0'$.

- $d = \mathbf{app}(\mathbf{lam}\ x.d_1, v_2)$ and $d' = d_1[x \mapsto v_2]$. Hence, by Lemma 2.9, we may assume that the derivation $\mathcal{D}$ is of the following form,

$$\dfrac{\dfrac{\mathcal{D}_1 :: \Sigma; \vec{P}; \Delta, x : s_1 \vdash_{\mathcal{S}} d_1 : s_2}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{lam}\ x.d_1 : s_1 \to_{tp} s_2} \quad \mathcal{D}_2 :: \Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} v_2 : s_1}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \mathbf{app}(\mathbf{lam}\ x.d_1, v_2) : s_2}$$

  where $s = s_2$. By Lemma 2.7 (3), we know $\Sigma; \vec{P}; \Delta \vdash d_1[x \mapsto v_2] : s_2$ is derivable.

- $d = \supset^-(\supset^+(v))$ and $d' = v$. Hence, by Lemma 2.9, we may assume that the derivation $\mathcal{D}$ is of the following form:

$$\dfrac{\dfrac{\Sigma; \vec{P}, P; \Delta \vdash_{\mathcal{S}} v : s}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \supset^+(v) : P \supset s} \quad \Sigma; \vec{P} \models_{\mathcal{S}} P}{\Sigma; \vec{P}; \Delta \vdash \supset^-(\supset^+(v)) : s}$$

  By Lemma 2.7 (2), we know $\Sigma; \vec{P} \vdash_{\mathcal{S}} v : s$ is derivable.

- $d = \forall^-(\forall^+(v))$ and $d' = v$. Hence, by Lemma 2.9, we may assume that the derivation $\mathcal{D}$ is of the following form:

$$\dfrac{\dfrac{\Sigma, a : \sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} v : s_1}{\Sigma; \vec{P}; \Delta \vdash_{\mathcal{S}} \forall^+(v) : \forall a : \sigma.s_1} \quad \Sigma \vdash_{\mathcal{S}} s_0 : \sigma}{\Sigma; \vec{P}; \Delta \vdash \forall^-(\forall^+(v)) : s_1[a \mapsto s_0]}$$

  where $s = s_1[a \mapsto s_0]$. By Lemma 2.7 (1), we know $\Sigma; \vec{P}; \Delta \vdash v : s_1[a \mapsto s_0]$ is derivable.

All other cases can be handled similarly. □

LEMMA 2.11 (CANONICAL FORMS). *Assume $\mathcal{D} :: \emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} v : s$. Then we have the following.*

1. *If $s = scc[\vec{s}]$, then v is of the form $dcc[\vec{v}]$ for some dynamic constructor dcc associated with the type constructor scc.*
2. *If $s = s_1 \to_{tp} s_2$, then v is of the form $\mathbf{lam}\ x.d$.*
3. *If $s = P \supset s_0$, then v is of the form $\supset^+(v_0)$.*
4. *If $s = \forall a : \sigma.s_0$, then v is of the form $\forall^+(v_0)$.*
5. *If $s = P \wedge s_0$, then v is of the form $\wedge(v_0)$.*
6. *If $s = \exists a : \sigma.s_0$, then v is of the form $\exists(v_0)$.*

PROOF. With Definition 2.2 (7), the lemma follows from structural induction on $\mathcal{D}$ immediately. □

THEOREM 2.12 (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.*

PROOF. The theorem follows from structural induction on $\mathcal{D}$. We present the case where the last applied rule in $\mathcal{D}$ is **(ty-∃-elim)**. Hence, $\mathcal{D}$ is of the following form,

$$\dfrac{\emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} d_1 : \exists a : \sigma.s_1 \quad \emptyset, a : \sigma; \emptyset; \emptyset, x : s_1 \vdash_{\mathcal{S}} d_2 : s_2}{\emptyset; \emptyset; \emptyset \vdash_{\mathcal{S}} \mathbf{let}\ \exists(x) = d_1\ \mathbf{in}\ d_2 : s_2}$$

where $d = \mathbf{let}\ \exists(x) = d_1\ \mathbf{in}\ d_2$ and $s = s_2$. We now have the following three subcases.

- $d_1 \hookrightarrow d_1'$ holds. Then we can readily verify that $d \hookrightarrow d' = \mathbf{let}\ \exists(x) = d_1'\ \mathbf{in}\ d_2$.
- $d_1 = E_1[dcf(\vec{v})]$ for some dynamic term $dcf(\vec{v})$ that is not a redex. Then let $E = \mathbf{let}\ \exists(x) = E_1\ \mathbf{in}\ d_2$ and we have $d = E[dcf(\vec{v})]$.
- d_1 is a value. Then d_1 is of the form $\exists(v_1)$ by Lemma 2.11. Therefore, $d \hookrightarrow d_2[x \mapsto v_1]$ holds.

Hence, this case is finished. The other cases can be handled similarly. □

2.4 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:

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

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

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

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 2.13. *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. With Lemma 2.11, (2) follows from structural induction on $\mathcal{D}$. □

With Theorem 2.13, 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.

3.1 General Recursion

We introduce a fixed-point operator **fix** to support general recursion in $\mathcal{ATS}$. We now call variables x **lam**-variables and intro-

duce **fix**-variables f. We use xf for a variable that is either a **lam**-variable or a **fix**-variable.

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

The typing rule **(ty-var)** is modified as follows,

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

and the following rule is added to handle the fixed-point operator.

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

A dynamic term of the form **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 2.10) and the progress theorem (Theorem 2.12) for this extension.

3.2 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.

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

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,

$$\frac{}{v \Downarrow x \Rightarrow [x \mapsto v]} \textbf{ (vp-var)}$$

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

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 $1 \le i \le n$, and its reduction is $d_i[\Theta_D]$. Note that reducing such a redex may involve nondeterminism if v matches several patterns p_i.

The typing rules for pattern matching is given in Figure 7. The meaning of a judgment of the form $\Sigma \vdash p \Downarrow s \Rightarrow \Sigma';\vec{P}';\Delta'$ is formally captured by the following lemma.

LEMMA 3.1. *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 $\emptyset;\emptyset;\emptyset \vdash_{\mathcal{S}} \Theta_D : \Delta[\Theta_S]$.*

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

As an example, the judgment below is derivable,

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

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

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

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

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

Figure 7. The typing rules for pattern matching

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

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

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

We can readily prove the subject reduction theorem (Theorem 2.10) for this extension: Lemma 3.1 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 2.12) 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 $1 \le i \le n$ if d is neither a value nor can be further reduced.

3.2.1 Products

Let $*$ be a binary static constructor of the sc-sort $[type, type] \Rightarrow type$. That is, $*$ is a type constructor. Also, Let $\underline{pair}$ be the only dynamic constructor associated with $*$, and the following dc-type is assigned to $\underline{pair}$,

$$\forall a_1 : type.\forall a_2 : type.[a_1,a_2] \Rightarrow_{tp} a_1 * a_2$$

where we use $a_1 * a_2$ for $*[a_1,a_2]$. Then the pairing function with the type $\forall a_1 : type.\forall a_2 : type.a_1 \rightarrow_{tp} a_2 \rightarrow_{tp} a_1 * a_2$ can be defined as follows,

$$\forall^+(\forall^+(\mathbf{lam}\ x_1.\mathbf{lam}\ x_2.\underline{pair}[x_1,x_2]))$$

and the first projection with the type $\forall a_1 : type.\forall a_2 : type.a_1 * a_2 \rightarrow_{tp} a_1$ can be defined as follows,

$$\forall^+(\forall^+(\mathbf{lam}\ x.\mathbf{case}\ x\ \mathbf{of}\ \underline{pair}[x_1,x_2] \Rightarrow x_1))$$

and the second projection can be defined similarly.

3.2.2 Sums

Let $+$ be a static constructor of the sc-sort $[type, type] \Rightarrow type$, and $\underline{inl}$ and $\underline{inr}$ be the two dynamic constructors associated with $+$ which are assigned the following dc-types respectively:

$$\forall a_1 : type.\forall a_2 : type.[a_1] \Rightarrow_{tp} a_1 + a_2$$
$$\forall a_1 : type.\forall a_2 : type.[a_2] \Rightarrow_{tp} a_1 + a_2$$

For examples, we can form a function of the type $s_1 + s_2 \rightarrow_{tp} s$ as follows,

$$\textbf{lam}\ x.(\textbf{case}\ x\ \textbf{of}\ \underline{inl}[x_1] \Rightarrow f_1(x_1) \mid \underline{inr}[x_2] \Rightarrow f_2(x_2))$$

where the two given functions f_1 and f_2 are of the types $s_1 \rightarrow_{tp} s$ and $s_2 \rightarrow_{tp} s$, respectively,

3.3 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.

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

The intended meaning of these functions should be obvious. We also need to add into Definition 2.2 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 (Harper, 1994) 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

We present some examples of applied type systems in this section.

4.1 λ_ω and $\lambda_{2,G\mu}$

Let $\text{STATICS}_0 = \text{STATICS}(\{bool, type\}; \mathcal{S}_0)$, that is, STATICS_0 is a static component in which the only base sorts are *bool* and *type*, and the only static constants are those declared in the initial signature $\mathcal{S}_\emptyset$. Let $\mathcal{M}_0$ be a model for STATICS_0 as is defined in Section 2.2. It is clear that β-conversion is valid in $\mathcal{M}_0$, that is,

$$\mathbf{V}^{\mathcal{M}_0}(\phi, (\lambda a : \sigma_1 : s_2)(s_1)) = \mathbf{V}^{\mathcal{M}_0}(\phi, s_2[a \mapsto s_1])$$

for every $\phi : \Sigma$ such that $\Sigma \vdash (\lambda a : \sigma_1 : s_2)(s_1) : \sigma_2$ is derivable.

One of the typed λ-calculi in λ-cube (Barendregt, 1992) is λ_ω, which is also known as System F_ω. We present the syntax of λ_ω as follows:

kinds	κ	$::= type \mid \kappa_1 \rightarrow_{tp} \kappa_2$
constructors	τ	$::= \alpha \mid \lambda\alpha : \kappa.\tau \mid \tau_1(\tau_2) \mid \tau_1 \rightarrow_{tp} \tau_2 \mid \forall\alpha : \kappa.\tau$
kinding ctx.	Σ	$::= \emptyset \mid \Sigma, \alpha : \kappa$
typing ctx.	Δ	$::= \emptyset \mid \Delta, \alpha : \kappa$
terms	t	$::= x \mid \lambda x : \tau.t \mid t_1(t_2) \mid \lambda\alpha : \kappa.t \mid t(\tau)$

Note that a kind κ and a constructor τ in λ_ω are just a sort and a static term in STATICS_0, respectively. In addition, a kinding context and a typing context in λ_ω are just a static context and a dynamic context in STATICS_0, respectively. A typing judgment in λ_ω is of the form $\Sigma; \Delta \vdash t : \tau$, for which the typing rules are standard. For instance, the following rule addresses the issue of type conversion,

$$\frac{\Sigma; \Delta \vdash t : \tau_1 \quad \Sigma \vdash \tau_1 =_\beta \tau_2 : type}{\Sigma; \Delta \vdash t : \tau_2}\ \textbf{(ty-}\beta\textbf{-conv)}$$

where $\Sigma \vdash \tau_1 =_\beta \tau_2 : type$ means that τ_1 and τ_2 are two well-formed types under Σ that are β-equivalent.

We define as follows a translation $\|\cdot\|$ that maps terms in λ_ω into dynamic terms in some applied type system ATS_0 whose statics is STATICS_0 and constraint relation is $\models^{\mathcal{M}_0}_{\mathcal{S}_\emptyset}$:

$$\begin{array}{rcl} \|x\| & = & x \\ \|\lambda x : \tau.t\| & = & \textbf{lam}\ x.\|t\| \\ \|t_1(t_2)\| & = & \textbf{app}(\|t_1\|, \|t_2\|) \\ \|\lambda\alpha : \kappa.t\| & = & \forall^+(\|t\|) \\ \|t(\tau)\| & = & \forall^-(\|t\|) \end{array}$$

PROPOSITION 4.1. *Assume* $\mathcal{D} :: \Sigma; \Delta \vdash t : \tau$ *in* λ_ω. *Then* $\Sigma; \emptyset; \Delta \vdash_{\mathcal{S}_\emptyset} \|t\| : \tau$ *is derivable in* ATS_0.

PROOF. By structural induction on $\mathcal{D}$. The only interesting point is to notice that $\Sigma \vdash \tau_1 =_\beta \tau_2$ implies $\Sigma; \emptyset \models^{\mathcal{M}_0}_{\mathcal{S}_\emptyset} \tau_1 \leq_{tp} \tau_2$ as $\mathcal{M}_0$ honors β-conversion. □

Like λ_ω, the language $\lambda_{2,G\mu}$ (Xi et al., 2003), which extends the second-order polymorphic λ-calculus λ_2 with guarded recursive datatypes, can also be embedded into ATS_0.

4.2 Dependent ML

Let $\text{STATICS}_1 = \text{STATICS}(\{bool, type, int\}; \mathcal{S}_1)$, where $\mathcal{S}_1$ extends the initial static signature $\mathcal{S}_\emptyset$ with declarations such as

$$\begin{array}{rcl} \underline{add} & : & [int, int] \Rightarrow_{tp} int \\ \underline{sub} & : & [int, int] \Rightarrow_{tp} int \\ \underline{mul} & : & [int, int] \Rightarrow_{tp} int \\ \underline{div} & : & [int, int] \Rightarrow_{tp} int \\ \underline{geq} & : & [int, int] \Rightarrow_{tp} bool \\ \underline{leq} & : & [int, int] \Rightarrow_{tp} bool \\ \underline{and} & : & [bool, bool] \Rightarrow_{tp} bool \\ \underline{or} & : & [bool, bool] \Rightarrow_{tp} bool \\ \underline{not} & : & [bool] \Rightarrow_{tp} bool \\ \ldots & : & \ldots \end{array}$$

Let $\mathcal{M}_1$ be a model for STATICS_1 such that the domain $\mathbf{D}_{int}$ in $\mathcal{M}_1$ is the set of integers and the above static functions are all given the standard interpretation. For instance, $\underline{add}$ is interpreted as the addition function on integers, $\underline{leq}$ is interpreted as the less-than-or-equal-to relation on integers, $\underline{or}$ is interpreted as the disjunction on booleans, etc.

Let ATS_1 be some applied type system such that its statics is and its constraint relation is $\models^{\mathcal{M}_1}_{\mathcal{S}_1}$. Then the version of Dependent ML as is presented in (Xi, 2002) can be easily embedded into ATS_1, provided that the dynamics of ATS_1 support general recursion, pattern matching, references and exceptions.

5 Related Work and Conclusion

The framework $\mathcal{ATS}$ is rooted in the work on Dependent ML (Xi and Pfenning, 1999; Xi, 1998), where the type system of ML is enriched with a restricted form of dependent datatypes, and the recent work on guarded recursive datatypes (Xi et al., 2003). Given the similarity between these two forms of types[3], we are naturally led to seeking a unified presentation for them.

For those who are familiar with qualified types (Jones, 1994), which underlies the type class mechanism in Haskell (Peyton Jones et al., 1999), 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 (Shao et al., 2002), 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) (Pfenning and Paulin-Mohring, 1989; Paulin-Mohring, 1993). However, the notion of a constraint relation in $\mathcal{ATS}$ does not have a counterpart in (Shao et al., 2002). 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 (Shao et al., 2002) 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 (Xi and Pfenning, 1999; Xi, 1998) and guarded recursive datatypes (Xi et al., 2003).

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.

6 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.

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

References

Andrews, P. B. (1972). General Models, Descriptions and Choice in Type Theory. *Journal of Symbolic Logic*, 37:385–394.

Andrews, P. B. (1986). *An Introduction to Mathematical Logic: To Truth through Proof*. Academic Press, Inc., Orlando, Florida.

Barendregt, H. P. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D. M., and Maibaum, T., editors, *Handbook of Logic in Computer Science*, volume II, pages 117–441. Clarendon Press, Oxford.

Chen, C. and Xi, H. (2003). Meta-Programming through Typeful Code Representation. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Uppsala, Sweden.

Church, A. (1940). A formulation of the simple type theory of types. *Journal of Symbolic Logic*, 5:56–68.

Constable, R. L. and Smith, S. F. (1987). Partial objects in constructive type theory. In *Proceedings of Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York.

Harper, R. (1994). A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206.

Hayashi, S. and Nakano, H. (1988). *PX: A Computational Logic*. The MIT Press.

Honsell, F., Mason, I. A., Smith, S., and Talcott, C. (1995). A variable typed logic of effects. *Information and Computation*, 119(1):55–90.

Jones, M. P. (1994). *Qualified Types: Theory and Practice*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK.

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

Milner, R., Tofte, M., Harper, R. W., and MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts.

Paulin-Mohring, C. (1993). Inductive Definitions in the System Coq: Rules and Properties. In Bezem, M. and de Groote, J., 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.

Peyton Jones, S. et al. (1999). Haskell 98 – A non-strict, purely functional language. Available at `http://www.haskell.org/onlinereport/`.

Pfenning, F. and Paulin-Mohring, C. (1989). 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.

Shao, Z., Saha, B., Trifonov, V., and Papaspyrou, N. (2002). 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.

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

Xi, H. (2002). Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Compu-*

tation, 15(1):91–132.

Xi, H., Chen, C., and Chen, G. (2002). Guarded Recursive Datatype Constructors. Available at `http://www.cs.bu.edu/~hwxi/GRecTypecon/`.

Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans.

Xi, H. and Pfenning, F. (1999). Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas.

Zenger, C. (1997). Indexed types. *Theoretical Computer Science*, 187:147–165.