Mem. Fac. Sci. Eng. Shimane Univ.
Series A
34, pp. 95-103 (2000)

Safe Dynamics and Distributed Programming

Ken-etsu FUJITA

Shimane University
fujiken@cis.shimane-u.ac.jp

Abstract

For distributed programming, we introduce the notion of safe dynamics with dynamic types into ML-like functional programming language. The notion of dynamic types is here parameterized with a set of monotypes, called a kind. The kind means an inductively defined set of monotypes. Hence, type case (dynamic type dispatch) mechanism can be naturally obtained as a recursive program over types, à la Martin-Löf, via an elimination rule for the inductively defined kinds. From a viewpoint of client-server programming, dynamic types under a constraint of the kinds provide a well-connected condition between a server and a client, with respect to types. Moreover, this point makes it possible to statically check dynamic types, such as ML-programs. A prototype of the system, called SDS (Safe Dynamics for Statically typed functional programming language), has been implemented by using C-language.

1. Introduction

Network programming requires a mechanism of sending and receiving not only the so-called data but also types. A pair of a term and its type is known as dynamics [ACPP91], and network programming should support the dynamics as a first class as well as type of dynamics, i.e., dynamic types in a typed language. We say that a dynamic type is safe if it can be statically type-checked, and dynamics with safe dynamic types is called a safe dynamics. The notion of safe dynamics is originally introduced by Duggan [Dug99]. In this paper, for distributed programming we introduce safe dynamics with dynamic types into ML-like functional programming. The dynamic type **Dynamic** is here parameterized with a set of monotypes, called a kind κ, which is denoted by **Dynamic**(κ). The dynamic types are dependent types upon kinds, and the kinds are inductively defined from monotypes. This natural definition leads to a useful mechanism such that type case programs are generic and recursive functions over the kinds. The key difference from Duggan [Dug99] is as follows:

(i) Kinds and subkind relations are simultaneously and inductively defined, and hence inhabitation of types and subkind relations are theoretically clear and simpler. Here, the subkind relation plays a role of a well-connected condition between distributed sites.

(ii) Our underlying theory is the core-ML [Mil78], and the type inference algorithm $\mathcal{W}$ can be naturally extended to our case together with a theory of inductive kinds.

The second statement implies that our language is implicitly typed, i.e., in the style of Curry rather than Church. Type theory [Bar92] usually has a hierarchy of expressions such as a level of kinds, type constructors, and objects in this order. However, the first statement above means that kinds are no guarantee of the existence of type constructors, but that type constructors are the basis of our kinds. Hence, a type case (dynamic type dispatch) mechanism can be naturally obtained as a recursive program over types, à la Martin-Löf [ML84, NPS90, DS99], from an elimination rule for the inductively defined kinds. This recursive function over types can be regarded as a generic function with a generic type as the domain. From a viewpoint of client-server programming, dynamic types under a constraint of inductively defined kinds provide a well-connected condition between a server and a client, with respect to types. The well-connected condition on kinds gives flexibility such that a server and a client can be independently updated without type errors. A prototype of the system, called SDS (Safe Dynamics for Statically typed functional programming language), has been implemented by using C-language [FO00].

2. Safe dynamics for statically typed language

We give the definition of typed language with safe dynamics, called $\mathbf{ML}_{\mathbf{sds}}$. The syntax of monomorphic types, polymorphic types, generic types, kinds, and terms are defined below.

Monotypes $\tau ::= \alpha \mid \mathbf{t}(\tau_1, \ldots, \tau_n) \mid \mathbf{Dynamic}(\kappa)$

We use α for a metavariable of type variables, and $\mathbf{t}$ denotes a type constructor. We will introduce pattern variables β latter, and pattern variables are type variables. For type constructors, in the case of nil arity, we have base types such as **int**, **string**, **bool**, **unit**, and in the case of non-nil arities, $\mathbf{list}(\alpha)$, $\alpha_1 \to \alpha_2$, $\alpha_1 \times \alpha_2$, etc. A type of dynamics is denoted by $\mathbf{Dynamic}(\kappa)$ parameterized with a kind κ defined latter.

We next define a syntax of patterns p for monotypes, consisting of monotypes and type constructors.

Patterns $p ::= \mathbf{t} \mid \tau$

Arities of patterns are defined as follows:

(1) $*$ is an arity of monotypes (saturated patterns).

(2) If $a_1, \ldots, a_n$ $(n \geq 1)$ are arities, then $(a_1, \ldots, a_n)$ is an arity of combined patterns.

(3) If a_1 and a_2 are arities, then $a_1 \twoheadrightarrow a_2$ is an arity of unsaturated patterns.

The arity $(a_1, \ldots, a_n)$ where $n=1$ is simply written by α_1.

We assume given type constructors have arities such that $\mathbf{list} : * \twoheadrightarrow *$, $\times : (*, *) \twoheadrightarrow *$, $\to : (*, *) \twoheadrightarrow *$, etc. Argument positions in patterns may be explicitly represented by using $[\ ]$, such as

$$p[\ ]_1 \cdots [\ ]_n : (\underbrace{*, \cdots, *}_{n}) \twoheadrightarrow *.$$

We consider only well-formed patterns with respect to arities, which are simply called patterns. As usual, well-formed monotypes are constructed from base types or **Dynamic** by the use of type constructors.

$$\frac{\overline{\mathbf{t}_{\mathbf{base}}:*} \quad \overline{\mathbf{Dynamic}(\kappa):*} \quad \overline{\mathbf{t}:(*,\cdots,*)\twoheadrightarrow *} \quad p[\ ]_1\cdots[\ ]_n:(*,\cdots,*)\twoheadrightarrow * \quad p_i:*(1\leq i\leq n)}{p[p_1]_1\cdots[p_n]_n:*}$$

The saturated pattern $p_i[\tau_1]\cdots[\tau_{n_i}]$ may be written by $p_i(\tau_1, \ldots, \tau_{n_i})$.

Polymorphic types σ are defined exactly as in ML. On the other hand, a generic type ς can be considered as a generalization of monotypes, and be used for a type case program via an elimination rule for a kind.

Ploytypes	Generic Types
$\sigma ::= \tau \mid \forall\alpha.\sigma$	$\varsigma ::= \forall\alpha::\kappa.\tau$

The syntax of kinds κ is defined from a set of patterns or **Dynamic**(κ) by using $\sqcup$.

Kinds $\kappa ::= \{p:(a_1, \ldots, a_n)\twoheadrightarrow *\} \mid \{\mathbf{Dynamic}(\kappa):*\} \mid \kappa\sqcup\kappa$

Well-formed kinds and a subkind relation on well-formed kinds are simultaneously defined as follows:

Inductive Definition of Kinds (*Well-Formed Kinds*)

(1-1) If $\tau_1, \ldots, \tau_n$ $(n>0)$ are monotypes excluding **Dynamic**, then the non-empty set $\{\tau_1:*, \ldots, \tau_n:*\}$ is a well-formed kind. The introduction rule is given as follows:

$$\frac{}{\tau_i::\{\tau_1:*,\cdots,\tau_n:*\}}\ (\text{mono})$$

(1-2) The next rules are introduction rules for kinds including **Dynamic**.

$$\frac{\tau::\kappa}{\tau::\{\mathbf{Dynamic}(\kappa):*\}}\ (Dy_1) \qquad \frac{\kappa \text{ is well-formed}}{\mathbf{Dynamic}(\kappa')::\{\mathbf{Dynamic}(\kappa):*\}}\ (Dy_2) \text{ for } \forall\kappa'\sqsubseteq\kappa.$$

(1-3) The following rules are inductive definition of kinds. The step case is derived from a set of unsaturated patterns (i.e., with arity $\alpha\twoheadrightarrow *$).

$$\frac{\tau::\kappa_b}{\tau::\kappa_b\sqcup\{p_1:a_1\twoheadrightarrow *,\cdots,p_m:a_m\twoheadrightarrow *\}}\ (base)$$

$$\frac{\tau_j::\kappa_b\sqcup\{p_1:a_1\twoheadrightarrow *,\cdots,p_m:a_m\twoheadrightarrow *\} \quad p_i[\tau_1]\cdots[\tau_{n_i}]:*}{p_i[\tau_1]\cdots[\tau_{n_i}]::\kappa_b\sqcup\{p_1:a_1\twoheadrightarrow *,\cdots,p_m:a_m\twoheadrightarrow *\}}\ (step) \text{ where } 1\leq j\leq n_i.$$

For well-formed κ_1 and κ_2, a subkind relation $\kappa_1\sqsubseteq\kappa_2$ is defined as follows:

(2-1) For any $\tau\neq\mathbf{Dynamic}(\kappa)$, if we have $\tau::\kappa_1$ then $\tau::\kappa_2$.

(2-2) For any **Dynamic**(κ), if we have **Dynamic**$(\kappa)::\kappa_1$ then **Dynamic**$(\kappa')::\kappa_2$ for any well-formed $\kappa'\sqsubseteq\kappa$.

From the definition, we have the following facts for well-formed kinds:

(i) $\kappa\sqsubseteq\kappa$

(ii) If $\kappa_1\sqsubseteq\kappa_2$ and $\kappa_2\sqsubseteq\kappa_3$, then $\kappa_1\sqsubseteq\kappa_3$.

(iii) If we have $\tau::\kappa_1$ and $\kappa_1\sqsubseteq\kappa_2$, then $\tau::\kappa_2$.

(iv) $\{\mathbf{Dynamic}(\kappa_1):*\}\sqsubseteq\{\mathbf{Dynamic}(\kappa_2):*\}$ if and only if $\kappa_1\sqsubseteq\kappa_2$

Lemma 1 (1) *Given κ, then it can be checked whether κ is well-formed.*

(2) *Given a monotype τ and a well-formed kind κ, then it can be checked whether $\tau::\kappa$.*

(3) *Given well-formed κ_1 and κ_2, then we can check whether $\kappa_1 \sqsubseteq \kappa_2$ or not.*

Proof. (1) Following the inductive definition.

(2) By induction on the construction of κ.

(3) By induction on κ_1 and κ_2. We consider only the case where both κ_1 and κ_2 are derived from (*base*−*step*). This case can be proved from the proposition below. Let $\kappa_i = \kappa_{bi} \sqcup P_i$ where P_i denotes a set of patterns with arities ($i=1, 2$). Then one can verify the following proposition:

Proposition 1 *$\kappa_1 \sqsubseteq \kappa_2$ if and only if either $\kappa_1 \sqsubseteq \kappa_{b2}$ or $\kappa_{b1} \sqsubseteq \kappa_2$ and $P_1 \subseteq P_2$.* □

A well-formed kind $\kappa \sqcup \{p_1:a_1 \twoheadrightarrow *, \ldots, p_m:a_m \twoheadrightarrow *\}$ may be simply written in the form of a set-like notation; $\kappa \cup \{p_1:a_1 \twoheadrightarrow *, \ldots, p_m:a_m \twoheadrightarrow *\}$. $\vec{\beta}::\kappa$ denotes $\beta_i::\kappa$ where $1 \leq i \leq n$ and $\vec{\beta} = (\beta_1, \ldots, \beta_n)$ called pattern variables. For instance, let κ_1 be {**int**:*, **string**:*, **list**:* $\twoheadrightarrow$ *, $\times$:(*, *) $\twoheadrightarrow$ *}. Then κ_1 denotes a set of monotypes, which is inductively constructed from **int** and **string** by the use of **list** and $\times$. One has **list**(**int**$\times$**string**)$::\kappa_1$ and **int**$\times$**list**(**string**)$::\kappa_1$. One also has {**int**:*, **string**:*} $\sqsubseteq \kappa_1$ and {**int**:*, **list**:* $\twoheadrightarrow$ *} $\sqsubseteq \kappa_1$. Let κ be {**Dynamic**(κ_1):*}. Then we have **Dynamic**(κ_1)$::\kappa$ and **list**(**int**)$::\kappa$, but not **list**(**Dynamic**(κ_1))$::\kappa$.

A monotype τ contains no greater kinds than those of τ', denoted by $\tau \preceq \tau'$, is defined as follows:

(0) $\tau \preceq \tau$;

(1) If we have $\kappa_1 \sqsubseteq \kappa_2$, then **Dynamic**($\kappa_1$) $\preceq$ **Dynamic**(κ_2);

(2) If we have $\tau_i \preceq \tau_i'$ $(1 \leq i \leq n)$ and $p \not\equiv \rightarrow$is an unsaturated pattern such that $p(\tau_1, \ldots, \tau_n)$:* and $p(\tau_1', \ldots, \tau_n')$:*, then $p(\alpha_1, \ldots, \tau_n) \preceq p(\tau_1', \ldots, \tau_n')$;

(3) If we have $\tau_i \preceq \tau_i'$ $(1 \leq i \leq 2)$, then $\tau_1' \rightarrow \tau_2 \preceq \tau_1 \rightarrow \tau_2'$.

The binary relation on monotypes is called a subtype relation, and from the definition if $\tau_1 \preceq \tau_2$ then both τ_1 and τ_2 have the same structure except for the occurrences of kinds appeared in **Dynamic**.

The syntax of terms (programs) M is defined as follows:

Terms

$M ::= x \mid \mathbf{c} \mid \lambda x.M \mid MM \mid \langle M, M \rangle \mid \mathbf{fst}(M) \mid \mathbf{snd}(M) \mid \mathbf{if}\ M\ \mathbf{then}\ M\ \mathbf{else}\ M$
$\mid \mathbf{let}\ x=M\ \mathbf{in}\ M \mid \mathbf{gen}\ x=Tycase\ \mathbf{in}\ M \mid \mathbf{rec}\ x.M \mid M[\tau:\kappa]M$
$\mid \mathbf{dynamic}(M, \tau:\kappa) \mid \mathbf{dyapp}(M, M:\kappa)$

$Tycase ::= \{(x_1, p_1:a_1) \Rightarrow M$" | "…" | "$(x_m, p_m:a_m) \Rightarrow M\}$" | "…
…" | "$\{(x_1, p_1:a_1) \Rightarrow M$" | "…" | "$(x_n, p_n:a_n) \Rightarrow M\}$

Coercion of a dynamics, i.e., a type case (dynamic type dispatch) subprogram is written in the form of $\{(x_1, p_1:a_1) \Rightarrow M_1\} \mid \{(x_2, p_2:a_2) \Rightarrow M_2 \mid (x_3, p_3:a_3) \Rightarrow M_3\}$. The type case subprogram as a generic function is used locally in the in-part of **gen** like let-polymorphism. A dynamics consisting of a pair of a type τ and a term M is denoted by **dynamic**(M, $\tau:\kappa$) with kind, and **dyapp** is an open-operator of the dynamics and then applies a generic function to the components of the dynamics, i.e., invoking coercion of a dynamics.

Basis for kinds and for types are respectively denoted by $\mathscr{K}$ and Γ.

Kind Basis $\mathscr{K} ::= < > \mid \beta ::\kappa, \mathscr{K}$ *Type Basis* $\Gamma ::= < > \mid x:\sigma, \Gamma \mid x: \forall \alpha ::\kappa.\alpha \to \tau, \Gamma$

where each kind in $\mathscr{K}$ and Γ is well-formed, and $\alpha \notin FV(\tau)$ in the generic type $\forall \alpha ::\kappa.\alpha \to \tau$.

$\Gamma_1 \preceq \Gamma_2$ is defined as $\Gamma_1(x) \preceq \Gamma_2(x)$ for each $x \in Dom(\Gamma_2) = Dom(\Gamma_1)$.

A judgement $\mathscr{K}; \Gamma \vdash \tau ::\kappa$ states that type τ has kind κ under $\mathscr{K}$ and Γ. We represent a conjunction of judgements by $\otimes$; we often write

$$\frac{\otimes_{1 \le i \le n} \mathscr{K}_i; \Gamma_i \vdash M_i : \tau_i}{\mathscr{K}; \Gamma \vdash M : \tau} \text{ for } \frac{\mathscr{K}_1; \Gamma_1 \vdash M_1 : \tau_1 \cdots \mathscr{K}_n; \Gamma_n \vdash M_n : \tau_n}{\mathscr{K}; \Gamma \vdash M : \tau}$$

Kind assignment rules are defined as follows:

Kind Assignment Rules

$$\frac{\mathscr{K}(\beta) \sqsubseteq \kappa}{\mathscr{K}; \Gamma \vdash \beta ::\kappa} \ (varkd) \qquad \frac{\tau ::\kappa}{\mathscr{K}; \Gamma \vdash \tau ::\kappa} \ (conkd)$$

$$\frac{\otimes_{1 \le i \le n} \mathscr{K}; \Gamma \vdash \beta_i ::\kappa \qquad \begin{matrix} [\beta_1 ::\kappa] \cdots [\beta_n ::\kappa] \\ \vdots \\ \tau ::\kappa \end{matrix}}{\mathscr{K}; \Gamma \vdash \tau ::\kappa} \ (kind)$$

A binary relation on types, $\tau \le \forall \alpha_1 \cdots \alpha_n.\tau'$ is defined such that $\tau = \tau'\ \theta$ under a simultaneous substitution $\theta = [\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]$ for type variables excluding pattern variables. A judgement $\mathscr{K}; \Gamma \vdash M : \tau$ states that term M has type τ under $\mathscr{K}$ and Γ. We write $\mathscr{K}; \Gamma \vdash M : \tau :: \kappa$ for both $\mathscr{K}; \Gamma \vdash M : \tau$ and $\mathscr{K}; \Gamma \vdash \tau :: \kappa$. The definition of type assignment rules is given as follows:

Type Assignment Rules

$$\frac{\tau \le \Gamma(x)}{\mathscr{K}; \Gamma \vdash x : \tau} \ (var) \qquad \frac{\Gamma_2 \preceq \Gamma_1 \quad \mathscr{K}; \Gamma_1 \vdash M : \tau_1 \quad \tau_1 \preceq \tau_2}{\mathscr{K}; \Gamma_2 \vdash M : \tau_2} \ (subty)$$

$$\frac{\mathscr{K}; \Gamma, x : \tau_1 \vdash M : \tau_2}{\mathscr{K}; \Gamma \vdash \lambda x.M : \tau_1 \to \tau_2} \ (abst) \qquad \frac{\mathscr{K}; \Gamma \vdash M_1 : \tau_1 \to \tau_2 \quad \mathscr{K}; \Gamma \vdash M_2 : \tau_1}{\mathscr{K}; \Gamma \vdash M_1 M_2 : \tau_2} \ (app)$$

$$\frac{\mathscr{K}; \Gamma \vdash M_1 : \tau_1 \quad \mathscr{K}; \Gamma \vdash M_2 : \tau_2}{\mathscr{K}; \Gamma \vdash \langle M_1, M_2 \rangle : \tau_1 \times \tau_2} \ (pair)$$

$$\frac{\mathscr{K}; \Gamma \vdash M : \tau_1 \times \tau_2}{\mathscr{K}; \Gamma \vdash \mathbf{fst}(M) : \tau_1} \ (fst) \qquad \frac{\mathscr{K}; \Gamma \vdash M : \tau_1 \times \tau_2}{\mathscr{K}; \Gamma \vdash \mathbf{snd}(M) : \tau_2} \ (snd)$$

$$\frac{\mathscr{K}; \Gamma \vdash M_1 : \mathbf{bool} \quad \mathscr{K}; \Gamma \vdash M_2 : \tau \quad \mathscr{K}; \Gamma \vdash M_3 : \tau}{\mathscr{K}; \Gamma \vdash \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : \tau} \ (if)$$

$$\frac{\mathscr{K}; \Gamma \vdash M_1 : \tau_1 \quad \mathscr{K}; \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash M_2 : \tau_2}{\mathscr{K}; \Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2} \ (let) \text{ where } \vec{\alpha} \notin FV(\Gamma).$$

$$\frac{\mathscr{K}; \Gamma, x : \tau \vdash M : \tau}{\mathscr{K}; \Gamma \vdash \mathbf{rec}\ x.M : \tau} \ (rec) \qquad \frac{\mathscr{K}; \Gamma \vdash M : \tau :: \kappa}{\mathscr{K}; \Gamma \vdash \mathbf{dynamic}\ (M, \tau : \kappa) : \mathbf{Dynamic}(\kappa)} \ (pack)$$

Let κ be a well-formed kind. Then we define a conjunction of judgements $Judge(\mathscr{K}; \Gamma; \kappa; \tau)$

and type case program $Tycase(\kappa)$ simultaneously by induction on the construction of κ.

(1) $\kappa=\{\tau_1{:}*, \cdots, \tau_n{:}*\}$ is derived by $(mono)$:

$$Judge(\mathscr{K}; \Gamma; \kappa; \tau) \stackrel{\text{def}}{=} \otimes_{1\leq i\leq n}\mathscr{K}; \Gamma, x_i{:}\tau_i \vdash M_i{:}\tau \text{ and } Tycase(\kappa) \stackrel{\text{def}}{=} \begin{array}{l} \{(x_1, \tau_1{:}*) \Rightarrow M_1 \\ \quad\vdots \\ \mid (x_n, \tau_n{:}*) \Rightarrow M_n\} \end{array}$$

for some M_i $(1\leq i\leq n)$.

(2) $\kappa=\{\mathbf{Dynamic}(\kappa_1){:}*\}$ is derived by (Dy):

$$Judge(\mathscr{K}; \Gamma; \kappa; \tau) \stackrel{\text{def}}{=} \mathscr{K}; \Gamma, x{:}\mathbf{Dynamic}\ (\kappa_1) \vdash M{:}\tau \otimes Judge(\mathscr{K}; \Gamma; \kappa_1; \tau)$$

and

$$Tycase(\kappa) \stackrel{\text{def}}{=} \begin{array}{l} \{(x, \mathbf{Dynamic}\ (\kappa_1){:}*) \Rightarrow M\} \\ \mid Tycase(\kappa_1) \end{array} \text{ for some } M.$$

(3) $\kappa=\kappa_1 \sqsubseteq \{p_1{:}a_1 \twoheadrightarrow *, \cdots, p_n{:}a_n \twoheadrightarrow *\}$ is derived by $(base-step)$:

$$Judge(\mathscr{K}; \Gamma; \kappa; \tau) \stackrel{\text{def}}{=} (\otimes_{1\leq i\leq n}\mathscr{K}, \vec{\beta}_i{::}\kappa; \Gamma, x_i{:}p_i(\vec{\beta}_i) \vdash M_i{:}\tau) \otimes Judge(\mathscr{K}; \Gamma; \kappa_1; \tau)$$

and

$$Tycase(\kappa) \stackrel{\text{def}}{=} \begin{array}{l} \{(x_1, p_1{:}a_1 \twoheadrightarrow *) \Rightarrow M_1 \\ \quad\vdots \\ \mid (x_n, p_n{:}a_n \twoheadrightarrow *) \Rightarrow M_n\} \\ \mid Tycase(\kappa_1) \end{array} \text{ for some } M_i(1\leq i\leq n).$$

$$\frac{Judga(\mathscr{K}; \Gamma, x{:}\forall\alpha{::}\kappa.\alpha\to\tau_1; \kappa; \tau_1) \quad \mathscr{K}; \Gamma, x{:}\forall\alpha{::}\kappa.\alpha\to\tau_1 \vdash M{:}\tau_2}{\mathscr{K}; \Gamma \vdash \mathbf{gen}\ x=Tycase(\kappa)\ \mathbf{in}\ M{:}\tau_2}\ (gen)$$

$$\frac{\Gamma(x)=\forall\alpha{::}\kappa_s\cdot\alpha\to\tau \quad \mathscr{K}; \Gamma \vdash M{:}\mathbf{Dynamic}(\kappa_c) \quad \kappa_c \sqsubseteq \kappa_s}{\mathscr{K}; \Gamma \vdash \mathbf{dyapp}(x, M{:}\kappa_c){:}\tau}\ (dyapp)$$

$$\frac{\Gamma(x)=\forall\alpha{::}\kappa_s.\alpha\to\tau_1 \quad \mathscr{K}; \Gamma \vdash M{:}\tau_2{::}\kappa_c \quad \kappa_c \sqsubseteq \kappa_s}{\mathscr{K}; \Gamma \vdash x[\tau_2{:}\kappa_c]M{:}\tau_1}\ (tyapp)$$

where the side condition $\kappa_c \sqsubseteq \kappa_s$ is called a well-connected condition.

Noted that (gen) involves an elimination rule for the inductively defined κ. For example, let $\kappa=\{\mathbf{string}\times\mathbf{int}{:}*, \mathbf{list}{:}*\to*\to*\}$, and $\Gamma' = \Gamma, f{:}\forall\alpha{::}\kappa.\alpha\to\tau_1$ where $\alpha\notin FV(\tau_1)$. Then the application of (gen) gives

$$\frac{\mathscr{K}; \Gamma', x_1{:}\mathbf{string}\times\mathbf{int} \vdash M_1{:}\tau_1 \quad \mathscr{K}, \beta_2{::}\kappa; \Gamma', x_2{:}\mathbf{list}(\beta_2) \vdash M_2{:}\tau_1 \quad \mathscr{K}; \Gamma' \vdash M{:}\tau_2}{\mathscr{K}; \Gamma \vdash \mathbf{gen}\ f=Tycase\ \mathbf{in}\ M{:}\tau_2}$$

where $Tycase$ is $\{(x_1, \mathbf{string}\times\mathbf{int}{:}*) \Rightarrow M_1\} \mid \{(x_2, \mathbf{list}{:}*\to*\to*) \Rightarrow M_2\}$ which plays a role of a recursive program over types, i.e., a type case program with a generic type.

The use of $(subty)$ and the subkinds relation $\sqsubseteq$ enable us to write flexible programs. Our intention is that a generic function with type $\Gamma(x)=\forall\alpha{::}\kappa_s.\alpha\to\tau$ is waiting for programs to be analyzed at a server site, and terms with a type whose kind is κ_c are sending from a client. The side condition $\kappa_c \sqsubseteq \kappa_s$ can be checked statically before exporting terms to the server, and hence

the condition guarantees the type safety of network connections, as observed in [Dug99]. See also examples in the next section.

3. Applying Safe Dynamics to Network Programming

We give two simple examples of network programming by the use of $\mathbf{ML}_{\mathbf{sds}}$. The one is a program of network printer based on the client-server paradigm. Another is an example of sending functions to access databases in a distributed site.

(i) Network printer

Let κ_c be $\{\mathbf{t}_1{:}*, \ldots, \mathbf{t}_n{:}*\} \sqcup \{\mathbf{s}_1{:}a_1 \twoheadrightarrow *, \ldots, \mathbf{s}_m{:}a_m \twoheadrightarrow *\}$, and **makedynamics** be the following subprogram where arities are omitted:

$$\begin{array}{rl} & \{(y_1, \mathbf{s}_1) \Rightarrow \mathbf{dynamic}\ (y_1, \mathbf{s}_1(\vec{\beta}_1){:}\kappa) \\ & \quad \vdots \\ \mathbf{makedynamics} \stackrel{\text{def}}{=} & (y_m, \mathbf{s}_m) \Rightarrow \mathbf{dynamic}\ (y_m, \mathbf{s}_m(\vec{\beta}_m){:}\kappa)\} \\ & \{(x_1, \mathbf{t}_1) \Rightarrow \mathbf{dynamic}\ (x_1, \mathbf{t}_1{:}\kappa) \\ & \quad \vdots \\ & (x_n, \mathbf{t}_n) \Rightarrow \mathbf{dynamic}\ (x_n, \mathbf{t}_n{:}\kappa)\} \end{array}$$

Then **makedynamics** packs dynamics with a term and a type as follows:

$$\mathbf{gen}\ f = \mathbf{makedynamics\ in}\ [f[\tau_1{:}\kappa_c]M_1, \ldots, f[\tau_n{:}\kappa_c]M_n] : \mathbf{list}(\mathbf{Dynamic}(\kappa_c))$$

To print it out, dynamics is sent to the server, and then at the server site, the dynamics is unpacked and next analyzed by a type case program. Let κ_s be $\{\mathbf{int}{:}*, \mathbf{string}{:}*, \mathbf{list}{:}* \twoheadrightarrow *, \times{:}(*, *) \twoheadrightarrow *\}$. We define the following recursive subprogram **opendynamics** to unpack and analyze dynamics, where $f{:}\forall\alpha{::}\kappa.\alpha \to \mathbf{unit}$ with $\kappa = \{\mathbf{Dynamic}(\kappa_s){:}*\}$:

$$\begin{array}{rl} & \{(x_1, \mathbf{int}) \Rightarrow \mathbf{output}(\mathbf{int\ TO\ string}\ x_1) \\ & (x_2, \mathbf{string}) \Rightarrow \mathbf{output}\ x_2\} \\ & \{(x_3, \mathbf{list}(\beta_3)) \Rightarrow \mathbf{if}\ x_3 = \mathbf{nil} \\ & \qquad \mathbf{then\ output}\ \text{``nil''} \\ & \qquad \mathbf{else}\ (\mathbf{output}\ (f[\beta_3{:}\kappa_s](\mathbf{car}\ x_3))) \\ \mathrm{opendynamics} \stackrel{\text{def}}{=} & \qquad\quad \mathbf{concatenate} \\ & \qquad\quad (f[\mathbf{list}(\beta_3){:}\kappa_s](\mathbf{cdr}\ x_3)) \\ & (x_4, \beta_4 \times \beta_5) \Rightarrow (\mathbf{output}\text{``(''}; f[\beta_4{:}\kappa_s](\mathbf{fst}\ x_4); \\ & \qquad \mathbf{output}\ \text{``,''}; \\ & \qquad f[\beta_5{:}\kappa_s](\mathbf{snd}\ x_4); \mathbf{output}\ \text{``)''})\} \\ & \{(x, \mathbf{Dynamic}\ (\kappa_s)) \Rightarrow \mathbf{dyapp}\ (f, x{:}\kappa_s)\} \end{array}$$

Then the program **Printer** can be obtained as follows:

$\lambda l.$ **gen** $f =$ **opendynamics in**
  let $g = (\mathbf{rec}\ z.\lambda y.\ \mathbf{if}\ y = \mathbf{nil\ then}\ (\ )\ \mathbf{else}\ (f[\mathbf{Dynamic}\ (\kappa_s)](\mathbf{car}\ y); z(\mathbf{cdr}\ y)))$
  in gl : **list** (**Dynamic** $(\kappa_s)) \to$ **unit**

Here, **Printer** is implemented as a function to print each element of **list**(**Dynamic**(κ_s)). If we have $\kappa_c \sqsubseteq \kappa_s$, then every term with type **list**(**Dynamics**(κ_c)) *gives no type error (type safe) and hence can be sent to the server for printing.* Moreover, **Printer** and the data to be printed can be independently updated without type errors if the well-connected condition holds true.

(ii) Database access

Let κ be {**list**(**int**) → **int**:*, **list**(**string**×**int**) → **string**:*}. For simplicity, assume that we have d_1:**list**(**int**) and d_2:**list**(**string***int**) at the server site. Let r_1:**ref**(**int**) and r_2:**ref**(**string**). We define the following function **access** to apply functions sent out to the database:

$$\textbf{access} \stackrel{\text{def}}{=} \begin{array}{l} | \ \{(x_1, \textbf{list}\ (\textbf{int}) \to \textbf{int}) \Rightarrow r_1 := (x_1 d_1) \\ | \ (x_2, \textbf{list}\ (\textbf{string} \times \textbf{int}) \to \textbf{string}) \Rightarrow r_2 := (x_2 d_2)\} \end{array}$$

Here, any term with the type contained in κ can be sent to the server, and then at the server site, the functions sent out can access d_1 and d_2 by the program below:

gen f=**access in rec** $g.\lambda l$. **if** l=**nil then** ( ) **else** (**dyapp** (f, **car** l:κ); g(**cdr** l))
:**list** (**Dynamic** (κ)) → **unit**

4. Concluding remarks

For distributed programming, we have provided the statically typed functional programming language with safe dynamics $\mathbf{ML}_{\mathbf{sds}}$. In this paper, we described only outline of the idea and simple examples. In network programming, dynamic types as dependent types upon kinds can provide the type safety of network connection. A prototype of the system has been experimentally implemented by using C-language [FO00].

In a general approach of type theory, Dybjer and Setzer [DS99] have introduced the principles for induction-recursion which allows simultaneous definitions of a function by structural recursion on a special type of codes for inductive-recursive definitions. The type dispatch programs can be explained as a special case of the general approach and can be regarded as a kind of an elimination rule for the universe (the set of small sets) [NPS90]. We come to a conclusion that the use of the universe practically has an interesting and important application area of network programming with type safety.

Forthcoming papers are going to be devoted to type inference, dynamic semantics, process language, and relations to the explicitly typed system of Duggan [Dug99], existential types [MP85] and ML plus dynamics [ACPP91, LM93] without kinds.

Acknowledgement I am grateful to Peter Dybjer for valuable and helpful comments on induction-recursion and polytypic programming, which gave a better perspective of this work.

Reference

[ACPP91] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G.: Dynamic typing in a statically-typed language, *ACM Transactions on Programming Languages and Systems*, 13(2), pp. 237–268, 1991.

[Bar92] Barendregt, H. P.: Lambda Calculi with Types, Handbook of Logic in Computer Science Vol. II, Oxford University Press, pp. 1–189, 1992.

[DM82] Damas, L. and R. Milner: Principal type-schemes for functional programs, *Proc. ACM Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[Dug99] Duggan, D.: Dynamic Typing for Distributed Programming in Polymorphic Languages, *ACM Transactions on Programming Languages and Systems*, 21(1), pp. 11–45, 1999.

[DS99] Dybjer, P. and Setzer, A.: A Finite Axiomatization of Inductive-Recursive Definitions, Springer Lecture Notes in Computer Science Vol. 1581 (*Typed Lambda Calculi and Applications*), Jean-Yves Girard (Ed.), pp. 129–146, 1999.

[LM93] Leroy, X. and Nauny, M.: Dynamics in ML, *J. Functional Programming*, 3(4), pp. 431–463, 1993.

[ML84] Martin-Löf, P.: Intuitionistic Type Theory, Bibliopolis, 1984.

[Mil78] Milner, R.: A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences*, 17, pp. 348–375, 1978.

[MP85] Mitchell, J. C. and G. D. Plotkin: Abstract types have existential type, *Proc. 12th ACM Symposium on Principles of Programming Languages*, pp. 37–51, 1985.

[NPS90] Nordström, B., Petersson, K., and J. M. Smith: Programming in Martin-Löf's Type Theory An Introduction, Oxford University Press, 1990.

[FO00] Fujita, K. and Obara, S.: Safe Dynamics for Distributed Programming, Technical Report in Computer Science and Systems Engineering CSSE-8, Kyushu Institute of Technology, March 1, 2000.