PROCEEDINGS OF THE YEREVAN STATE UNIVERSITY

Physical and Mathematical Sciences 2009, № 3, p. 42–51

Informatics

ON THE TYPE CORRECTNESS OF POLYMORPHIC λ-TERMS. 1

A. H. ARAKELYAN*

Chair of Programming and Information Thechnologies, YSU

In this paper polymorphic lambda terms are considered, where no type information is provided for the variables. The aim of this work is to extend the algorithm of typification [1] of such terms introducing type constants and term constants.

***Keywords*:** type, term, constraint, skeleton, expansion, typing.

1. Introduction. Types are used in programming languages to analyze programs without executing them, for purposes such as detecting programming errors earlier, for doing optimizations etc. In some programming languages no explicit type information is provided by the programmer, hence some system of type inference is required to recover the lost information and do compile time type checking. One of such type inference systems is the well known Hindley/Milner system [2], used in languages such as Haskell, SML, OCaml etc. Important property of type systems is the property of *principal typings* [3, 4], which allows the compiler to do *compositional analysis*, i.e. analysis of modules in absence of information about other modules [3, 4]. Unfortunately Hindley/Milner system doesn't support the property of *principal typings* [3]. In this paper we consider a type inference system, called *System E* [1, 5]. In section 2 an extended version of *System E* is presented, which adds type constants and term constants to the original *System E*.

2. Definitions Used and Previous Results.

2.1. Definitions used. Let *TypeVariable* be a countable set of type variables, *TypeConstant* be a finite set of built-in types, *TermVariable* be a countable set of term variables, *Constant* be a countable set of constants and $(ExpansionVariable, \preceq)$ be a countable totally ordered set of expansion variables.

Definition 2.1. The set of types *Type* is defined as follows:

1. $\omega \in Type$; 2. If $\alpha \in TypeVariable$, then $\alpha \in Type$; 3. If $s \in TypeConstant$, then $s \in Type$; 4. If $e \in ExpansionVariable$ and $\tau \in Type$, then $e\tau \in Type$; 5. If $\tau_1, \tau_2 \in Type$, then $(\tau_1 \to \tau_2) \in Type$ and $(\tau_1 \cap \tau_2) \in Type$.

The set of expansions *Expansion* is defined as follows:

* E-mail: ara_arakelyan@yahoo.com

1. $\omega \in Expansion$; 2. If σ is a substitution (we will define substitutions later), then $\sigma \in Expansion$; 3. If $e \in ExpansionVariable$ and $E \in Expansion$, then $eE \in Expansion$; 4. If $E_1, E_2 \in Expansion$, then $(E_1 \cap E_2) \in Expansion$.

The set of terms $Term$ is defined as follows:

1. If $x \in TermVariable$, then $x \in Term$; 2. If $c \in Constant$, then $c \in Term$; 3. If $x \in TermVariable$ and $M \in Term$, then $(\lambda x.M) \in Term$; 4. If $(\lambda x.M) \in Term$, then $(M_1 M_2) \in Term$.

The set of constraints $Constraint$ is defined as follows:

1. $\omega \in Constraint$; 2. If $\tau_1, \tau_2 \in Type$, then $(\tau_1 \doteq \tau_2) \in Constraint$; 3. If $e \in ExpansionVariable$ and $\Delta \in Constraint$, then $e\Delta \in Constraint$; 4. If $\Delta_1, \Delta_2 \in Constraint$, then $(\Delta_1 \cap \Delta_2) \in Constraint$.

The set of skeletons $Skeleton$ is defined as follows:

1. If $M \in Term$, then $\omega^M \in Skeleton$; 2. If $c \in Constant$ and $\tau \in Type$, then $c^{:\tau} \in Skeleton$; 3. If $x \in TermVariable$ and $\tau \in Type$, then $x^{:\tau} \in Skeleton$; 4. If $e \in ExpansionVariable$ and $Q \in Skeleton$, then $eQ \in Skeleton$; 5. If $x \in TermVariable$ and $Q \in Skeleton$, then $(\lambda x.Q) \in Skeleton$; 6. If $Q_1, Q_2 \in Skeleton$, then $(Q_1 \cap Q_2) \in Skeleton$; 7. If $Q_1, Q_2 \in Skeleton$ and $\tau \in Type$, then $(Q_1 Q_2)^{:\tau} \in Skeleton$.

We assume that:

1. $((T_1 \cap T_2) \cap T_3) = (T_1 \cap (T_2 \cap T_3))$; 2. $(T_1 \cap T_2) = (T_2 \cap T_1)$; 3. $(\omega \cap T) = T$; 4. $e(T_1 \cap T_2) = (eT_1 \cap eT_2)$; 5. $e\omega = e$, where $T_1, T_2, T_3 \in Type$ or $T_1, T_2, T_3 \in Constraint$ and $e \in ExpansionVariable$.

Definition 2.2. Let $\alpha_1, \ldots, \alpha_n \in TypeVariable$, $e_1, \ldots, e_m \in ExpansionVariable$, $\tau_1, \ldots, \tau_n \in Type$, $E_1, \ldots, E_m \in Expansion$, $n \geq 0$, $m \geq 0$. The set of pairs $\{\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n, e_1 := E_1, \ldots, e_m := E_m\}$ is called a substitution, if the following conditions are satisfied:

1. $i \neq j \Rightarrow \alpha_i \neq \alpha_j$, where $i, j = 1, \ldots, n$; 2. $i \neq j \Rightarrow e_i \neq e_j$, where $i, j = 1, \ldots, m$.

We denote by ε the empty substitution.

Definition 2.3. Let $e_1, \ldots, e_n \in ExpansionVariable$, $n \geq 0$. Then $e_1 e_2 \ldots e_n$ is called E-path and is denoted by $\vec{e}$. In case of $n=0$ E-path will be denoted by $\vec{\varepsilon}$.

Definition 2.4. Let $\vec{e}_1 = e_1 e_2 \ldots e_n$ and $\vec{e}_2 = e'_1 e'_2 \ldots e'_m$, where $n, m \geq 0$. If $\exists i$ s.t. $1 \leq i \leq \min(n, m)$ and $e_j = e'_j$ $\forall j = 1, \ldots, i-1$ and $e_i \prec e'_i$ ($e_i \succ e'_i$), then $\vec{e}_1 \preceq \vec{e}_2$ ($\vec{e}_1 \succeq \vec{e}_2$). Else if $n \leq m$ ($n \geq m$), then $\vec{e}_1 \preceq \vec{e}_2$ ($\vec{e}_1 \succeq \vec{e}_2$).

It is easy to see that the set of E-paths with order $\preceq$ is a totally ordered set.

Definition 2.5. A constraint Δ is singular, if it was constructed without using operation $\cap$.

Remark 2.1. Taking into account the definition of constraints, it is easy to see that each constraint has one of the following forms: $\Delta = \vec{e}_1(\tau_1^1 \doteq \tau_2^1) \cap \ldots \cap \vec{e}_n(\tau_1^n \doteq \tau_2^n)$, $n \geq 1$, or $\Delta = \omega$, i.e. each constraint is an intersection of zero or more singular constraints.

Let us introduce the following notation: $E-Path(\vec{e}(\tau_1 \doteq \tau_2)) = \vec{e}$.

Definition 2.6. A constraint Δ is solved, iff it is of the form $\Delta=\vec{e}_1(\tau_1 \doteq \tau_1) \cap \ldots \cap \vec{e}_n(\tau_n \doteq \tau_n)$ $n \geq 1$ or $\Delta=\omega$. The unsolved part of a constraint Δ, written $unsolved(\Delta)$, is the smallest part of Δ such that $\Delta=unsolved(\Delta) \cap \Delta'$ and Δ' is solved. Consequently Δ' is the greatest solved part of a Δ, it is called solved part of a constraint Δ and is written $solved(\Delta)$. So each constraint is an intersection of its solved and unsolved parts: $\Delta=unsolved(\Delta) \cap solved(\Delta)$.

As we will see later, the Skeleton is an object, that contains all information about the type inference tree of some term. We can calculate the term corresponding to the Skeleton, using the following function.

Definition 2.7. The function $term:Skeleton \to Term$ is defined as follows:

1. $term(\omega^M) = M$; 2. $term(c^{:\tau}) = c$; 3. $term(x^{:\tau}) = x$; 4. $term(eQ) = term(Q)$; 5. $term((\lambda x.Q)) = (\lambda x.term(Q))$; 6. $term((Q_1Q_2)^{:\tau}) = (term(Q_1)term(Q_2))$; 7. If $term(Q_1)=term(Q_2)$, then $term((Q_1 \cap Q_2))=term(Q_1)$, else $term((Q_1 \cap Q_2))$ is undefined.

Definition 2.8. The skeleton Q is well formed, iff $term(Q)$ is defined, i.e. the corresponding term of skeleton exists.

Convention 2.1. Henceforth only well formed skeletons are considered.

The following two definitions define the application of expansions to types, constraints, expansions and skeletons.

Definition 2.9. Let $X \in Type \cup Constraint \cup Expansion \cup Skeleton$ and σ be a substitution. Then the application of σ to X is denoted by $[\sigma]X$ and is obtained from σ and X by the following rules:

1. If $\alpha := \tau \in \sigma$, then $[\sigma]\alpha=\tau$; 2. If $\alpha := \tau \notin \sigma$ $\forall \tau \in Type$, then $[\sigma]\alpha=\alpha$; 3. $[\sigma]s=s$; 4. $[\sigma]\omega=\omega$; 5. If $e := E \in \sigma$, then $[\sigma]eY=[E]Y$; 6. If $e := E \notin \sigma$ $\forall E \in Expansion$, then $[\sigma]eY=eY$; 7. $[\sigma](\tau_1 \to \tau_2) = ([\sigma]\tau_1 \to [\sigma]\tau_2)$; 8. $[\sigma](X_1 \cap X_2) = =([\sigma]X_1 \cap [\sigma]X_2)$; 9. $[\sigma]\{\alpha_1 := \tau_1,\ldots,\alpha_n := \tau_n, e_1 := E_1,\ldots,e_m := E_m\} = \{\alpha_1 := [\sigma]\tau_1,\ldots,\alpha_n := =[\sigma]\tau_n, e_1 := [\sigma]E_1,\ldots, e_m := [\sigma]E_m\} \cup \{\alpha := \tau | \alpha \notin \{\alpha_1,\ldots,\alpha_n\}$ and $\alpha := \tau \in \sigma\} \cup \cup\{e := E | e \notin \{e_1,\ldots,e_m\}, e := E \in \sigma\}$; 10. $[\sigma](\tau_1 \doteq \tau_2) = ([\sigma]\tau_1 \doteq [\sigma]\tau_2)$; 11. $[\sigma]\omega^M = \omega^M$; 12. $[\sigma]x^{:\tau} = x^{:[\sigma]\tau}$; 13. $[\sigma]c^{:\tau} = c^{:[\sigma]\tau}$; 14. $[\sigma](\lambda x.Q) = (\lambda x.[\sigma]Q)$; 15. $[\sigma](Q_1Q_2)^{:\tau} = =([\sigma]Q_1[\sigma]Q_2)^{:[\sigma]\tau}$, where $\alpha_1,\ldots,\alpha_n,\alpha \in TypeVariable, e_1,\ldots,e_m,$ $e \in ExpansionVariable,$ $c \in Constant,$ $s \in TypeConstant,$ $E_1,\ldots,E_m,E \in Expansion,$ $\tau_1,\ldots,\tau_n,\tau,\tau_1,\tau_2 \in Type,$ $Y,X_1,X_2 \in Type \cup Constraint \cup Expansion \cup Skeleton,$ $M \in Term,$ $n,m \geq 0,$ $Q,Q_1,Q_2 \in Skeleton,$ and $[E]Y$ will be defined now.

Definition 2.10. Let $X \in Type \cup Constraint \cup Expansion \cup Skeleton$ and $E \in Expansion$. Then the application of E to X is denoted by $[E]X$ and is obtained from E and X by the following rules:

1. If $E=\omega$, then $[E]Y=\omega$, where $Y \in Type \cup Constraint \cup Expansion$; 2. If $E=\omega$, then $[E]Q=\omega^{term(Q)}$, where $Q \in Skeleton$; 3. If $E=\sigma$, then $[E]X=[\sigma]X$, where σ is a substitution; 4. If $E=eE'$, then $[E]X=e[E']X$, where

$e \in ExpansionVariable$ and $E' \in Expansion$; 5. If $E=(E_1 \cap E_2)$, then $[E]X=([E_1]X \cap [E_2]X)$, where $E_1, E_2 \in Expansion$.

Let us introduce the following notations:

1. $e/\sigma=\{e := e\sigma\}$; 2. If $\vec{e}=e_1 e_2 \ldots e_n$, then $\vec{e}/\sigma=e_1/e_2/\ldots/e_n/\sigma$, $n \geq 0$, where $e_1,\ldots,e_n, e \in ExpansionVariable$ and σ is a substitution. It is easy to see that $[e/\sigma]eX=e[\sigma]X$, $[\vec{e}/\sigma]\vec{e}X=\vec{e}[\sigma]X$, where

$X \in Type \cup Constraint \cup Expansion \cup Skeleton.$

Lemma 2.1 (property of expansions). Let $E_1, E_2, E \in Expansion$ and $X \in Type \cup Constraint \cup Expansion \cup Skeleton$, then $[[E_1]E_2]X=[E_1][E_2]X$ and $[E]\varepsilon=E$.

Definition 2.11. A total function $A: TermVariable \to Type$ is called environment, if the following set is finite: $\{x | x \in TermVariable$ and $A(x) \neq \omega\}$. Environment A can be written also as a set of pairs $A=\{(x, A(x)) \mid x \in TermVariable\}$.

Let us introduce the following notations:

1. $A[x \to \tau] = \{(y, A(y)) \mid y \in TermVariable$ and $y \neq x\} \cup \{(x,\tau)\}$; 2. $A \cap B= =\{(x,(A(x) \cap B(x))) \mid x \in TermVariable\}$; 3. $eA=\{(x, eA(x)) \mid x \in TermVariable\}$; 4. $[E]A=\{(x,[E]A(x)) \mid x \in TermVariable\}$; 5. $env_\omega=\{(x,\omega) | x \in TermVariable\}$, where A, B are environments, $E \in Expansion$, $e \in ExpansionVariable$, $x \in TermVariable$ and $\tau \in Type$.

Definition 2.12. The set $CType \subset Type$ is the set, satisfying the following conditions:

1. If $s \in TypeConstant$, then $s \in CType$; 2. If $s \in TypeConstant$ and $\tau \in CType$, then $(s \to \tau) \in CType$.

Definition 2.13. The mapping $\Sigma: Constant \to CType$ is called a constant table.

Convention 2.2. In order not to mention the constant table later, let us suppose that henceforth we are using some fixed constant table.

2.2. Type inference rules.

Definition 2.14. The quintuple of term, skeleton, environment, type and constraint, written $(M \triangleright Q) : (A \vdash \tau) / \Delta$, is called a judgement. The intended meaning of judgement is that Q is a proof that M has typing $(A \vdash \tau)$, provided that the constraint Δ is solved.

Now let us introduce type inference rules, that are used to derive judgements. Type inference rules are the following:

[VAR] $\dfrac{}{(x \triangleright x^{:\tau}) : (env_\omega[x \to \tau] \vdash \tau) / \omega}$, [CONST] $\dfrac{}{(c \triangleright c^{:\tau}) : (env_\omega \vdash \tau) / \omega}$,

where $\tau = \Sigma(c)$,

[E-VAR] $\dfrac{(M \triangleright Q) : (A \vdash \tau) / \Delta}{(M \triangleright eQ) : (eA \vdash e\tau) / e\Delta}$, [OMEGA] $\dfrac{}{(M \triangleright \omega^M) : (env_\omega \vdash \omega) / \omega}$,

[ABS] $\dfrac{(M \triangleright Q) : (A \vdash \tau) / \Delta}{((\lambda x.M) \triangleright (\lambda x.Q)) : (A[x \to \omega] \vdash (A(x) \to \tau)) / \Delta}$,

[APP] $\dfrac{(M_1 \triangleright Q_1):(A_1 \vdash \tau_1)/\Delta_1 \text{ and } (M_2 \triangleright Q_2):(A_2 \vdash \tau_2)/\Delta_2}{((M_1M_2) \triangleright (Q_1Q_2)^{:\tau}):(A_1 \cap A_2 \vdash \tau)/\Delta_1 \cap \Delta_2 \cap (\tau_1 \doteq (\tau_2 \to \tau))}$,

[INT] $\dfrac{(M \triangleright Q_1):(A_1 \vdash \tau_1)/\Delta_1 \text{ and } (M \triangleright Q_2):(A_2 \vdash \tau_2)/\Delta_2}{(M \triangleright (Q_1 \cap Q_2)):(A_1 \cap A_2 \vdash (\tau_1 \cap \tau_2))/\Delta_1 \cap \Delta_2}$.

Definition 2.15. The pair $(A \vdash \tau)$ of an environment and a type is called typing of the term M, if $\exists Q \in Skeleton$ and $\exists \Delta \in Constraint$ s.t. judgement $(M \triangleright Q):(A \vdash \tau)/\Delta$ is inferable and Δ is solved.

Definition 2.16. The pair $(A \vdash \tau)$ of an environment and a type is called a principal typing of the term M, if $(A \vdash \tau)$ is a typing of M, and if $(A' \vdash \tau')$ is a typing of M, then $\exists E \in Expansion$ s.t. $A'=[E]A$ and $\tau'=[E]\tau$. In other words, all typings of the term are obtained from principal typing through expansion.

Lemma 2.2. If the judgement $(M \triangleright Q):(A \vdash \tau)/\Delta$ is inferable, then the judgement $(M \triangleright [E]Q):([E]A \vdash [E]\tau)/[E]\Delta$ is also inferable for any expansion E.

The next lemma shows that each skeleton contains information about one and only one inferable judgement, i.e. represents one and only one derivation tree of some judgement.

Lemma 2.3. Let $Q \in Skeleton$. Then there exist one and only one term M, an environment A, a type τ and a constraint Δ s.t. the judgement $(M \triangleright Q):(A \vdash \tau)/\Delta$ is inferable and $M=term(Q)$.

This Lemma allows us to introduce the following functions: $env(Q)=A$, $type(Q)=\tau$, $constraint(Q)=\Delta$, $typing(Q)=(A \vdash \tau)$. It is easy to present algorithms of calculating functions *env*, *type*, *constraint* and *typing*.

2.3. Initial skeleton. Type inference algorithm, that will be introduced in section 2.6, starts term typification by constructing initial skeleton of that term.

Definition 2.17. Let us fix a type variable a_0 and expansion variables e_0, e_1, e_2 such that $e_0 \prec e_1 \prec e_2$. The function $initial: Term \to Skeleton$ maps terms to skeletons as follows: $initial(x)=x^{:a_0}$, $initial(c)=c^{:\Sigma(c)}$, $initial((\lambda x.M))=(\lambda x.e_0 initial(M))$, $initial((M_1M_2))=(e_1 initial(M_1)\ \ e_2 initial(M_2))^{:a_0}$, where $x \in TermVariable$, $c \in Constant$ and $M, M_1, M_2 \in Term$. The range of *initial* is denoted by *InitialSkeleton* and elements of *InitialSkeleton* are called initial skeletons.

Lemma 2.4. Let $P \in InitialSkeleton$. Then $solved(constraint(P))=\omega$, and $\forall x \in TermVariable$ $\exists n \geq 0$ and $\exists \vec{e}_1, \ldots, \vec{e}_n$ E-paths s.t. $env(P)(x)=\vec{e}_1 a_0 \cap \ldots \cap \vec{e}_n a_0$, and no $\vec{e}_i$ is a proper prefix of $\vec{e}_j$, $i, j \in \{1, \ldots, n\}$.

From the first part of Lemma 2.4 it is easy to see that all singular constraints, which are a part of $constraint(P)$, are unsolved, where P is an initial skeleton of some term. In section 2.6 we will see, that the type inference algorithm tries to solve some singular constraints by applying substitutions on them, and it starts solving the process from singular constraints, which are a part of $constraint(P)$. Unification rules, introduced in the next section, are used to produce substitutions for solving singular constraints.

2.4. Unification rules. In this section three unification rules will be presented.

Definition 2.18. The set $Type' \subset Type$ is the set of types that are constructed without using type constants and operation $\rightarrow$.

Definition 2.19. The function $Extract_E : Type' \rightarrow Expansion$ maps types from the set $Type'$ to expansions as follows: $Extract_E(\omega)=\omega$, $Extract_E(\alpha)=\varepsilon$, $Extract_E(e\tau)=eExtract_E(\tau)$, $Extract_E((\tau_1 \cap \tau_2))=(Extract_E(\tau_1) \cap Extract_E(\tau_2))$, where $\alpha \in TypeVariable$, $e \in ExpansionVariable$ and $\tau, \tau_1, \tau_2 \in Type'$.

Definition 2.20. The function $Extract_S : Type' \times Type \rightarrow Substitution$ maps pairs of type from $Type'$, and type to substitutions as follows: $Extract_S(\omega, \tau')=\varepsilon$, $Extract_S(\alpha, \tau')=\{\alpha := \tau'\}$, $Extract_S(e\tau, \tau') = e/Extract_S(\tau, \tau')$, $Extract_S((\tau_1 \cap \tau_2), \tau') = = [Extract_S(\tau_2, \tau')]Extract_S(\tau_1, \tau')$, where $\tau' \in Type$, $e \in ExpansionVariable$, $\alpha \in TypeVariable$ and $\tau, \tau_1, \tau_2 \in Type'$.

Definition 2.21 (unify$_\beta$ rule). Let $\overline{\Delta}=\vec{e}(e_1(e_0\tau_0 \rightarrow e_0\tau_1) \doteq (e_2\tau_2 \rightarrow a_0))$ be a singular constraint, where $\tau_0 \in Type'$ and $\tau_1, \tau_2 \in Type$. Then the rule $unify_\beta$ is applicable to $\overline{\Delta}$, and the result of this application is the following substitution: $\sigma=\vec{e}/\{a_0 := [\sigma']\tau_1, e_1 := \{e_0 := \sigma'\}, e_2 := E'\}$, where $E' = Extract_E(\tau_0)$ and $\sigma'=Extract_S(\tau_0, \tau_2)$.

The application of the rule $unify_\beta$ is written as $\overline{\Delta} \xrightarrow{unify_\beta} \sigma$.

Now let us explain the meaning of the rule $unify_\beta$. Let $((\lambda x.M_1(M_2))$ be a subterm of some term M, where $x \in TermVariable$ and $M_1, M_2 \in Term$. The initial skeleton of that subterm will be $(e_1(\lambda x.e_0P_1)e_2P_2)^{:a_0}$, where $P_1=initial(M_1)$ and $P_2=initial(M_2)$. The part of $constraint(initial(M))$ that corresponds the initial skeleton of subterm mentioned above will be $\vec{e}(e_1(e_0\tau_0 \rightarrow e_0\tau_1) \doteq (e_2\tau_2 \rightarrow a_0))$, where τ_1 corresponds to the type of M_1 $(\tau_1=type(P_1))$, τ_2 corresponds to the type of M_2 $(\tau_2=type(P_2))$ and τ_0 corresponds to the type of x in term M_1 $(\tau_0=env(P_1)(x))$. Before applying substitution created by the rule $unify_\beta$, the type a_0 is associated with any free occurrence of variable x in the term M_1. After applying substitution, all that a_0 type variables will be replaced with the type τ_2 (this replacement is done by the substitution created by the function $Extract_S$), and the type of x in term M_1 will be changed. The same type is obtained, when applying substitution created by the function $Extract_E$ to the type τ_2 (it makes as many copies of type τ_2 as there are free occurrences of variable x in term M_1). It is easy to see that the process described above is very similar to the one step β-reduction. Next two lemmas show the exact correspondence of the rule $unify_\beta$ with β-reduction.

Lemma 2.5 (correspondence with β-reduction). Let $M \in Term$ and

P=$initial(M)$. If $constraint(P) = \overline{\Delta} \cap \Delta'$, where $\overline{\Delta}$ is a singular constraint, to which the rule $unify_\beta$ is applicable and $\overline{\Delta} \xrightarrow{unify_\beta} \sigma$, then $\exists M' \in Term$ s.t. $constraint(P')$=$[\sigma]\Delta'$, $env(P') = [\sigma]env(P)$, $type(P') = [\sigma]type(P)$ and $M \to_\beta M'$, where $P' = initial(M')$.

Lemma 2.6 (correspondence with β-reduction). Let $M, M' \in Term$, P=$initial(M)$ and $P' = initial(M')$. If $M \to_\beta M'$, then $\exists\overline{\Delta}$ singular constraint s.t. $constraint(P) = \overline{\Delta} \cap \Delta'$ the rule $unify_\beta$ is applicable to $\overline{\Delta}$ and $\overline{\Delta} \xrightarrow{unify_\beta} \sigma$, $constraint(P') = [\sigma]\Delta'$, $env(P') = [\sigma]env(P)$ and $type(P') = [\sigma]type(P)$.

Definition 2.22 ($unify_x$ rule). Let $\overline{\Delta}$=$\vec{e}(e_1 a_0 \doteq (e_2\tau \to a_0))$ be a singular constraint, where $\tau \in Type$. Then the rule $unify_x$ is applicable to $\overline{\Delta}$, and the result of application is the following substitution: σ=$\vec{e}/\{e_1 := \{a_0 := (e_2\tau \to a_0), e_1 := e_1e_1\varepsilon, e_2 := e_1e_2\varepsilon\}\}$.

The application of the rule $unify_x$ is written as $\overline{\Delta} \xrightarrow{unify_x} \sigma$.

Now let us explain the meaning of the rule $unify_x$. Let (M_1M_2) be a subterm of some term M, where $M_1, M_2 \in Term$. During the work of the type inference algorithm the corresponding skeleton of that subterm can be $(e_1P_1e_2P_2)^{:a_0}$, where $P_1, P_2 \in Skeleton$ and $type(P_1) = a_0$. The singular constraint corresponding to the skeleton mentioned above is $\vec{e}(e_1a_0 \doteq (e_2\tau \to a_0))$, where τ corresponds to the type of M_2, and a_0 corresponds to the type of M_1 in the current stage of the work of type inference algorithm. After applying substitution σ the type of M_1 will be replaced with $(e_2\tau \to a_0)$, and the skeleton mentioned above will have the following form: $(P_1'e_2P_2)^{:a_0}$, where $type(P_1')$=$(e_2\tau \to a_0)$.

Definition 2.23 ($unify_c$ rule). Let $\overline{\Delta}$=$\vec{e}(e_1\tau_0 \doteq (e_2\tau \to a_0))$ be a singular constraint, where $\tau_0 \in CType$ and $\tau \in Type$. Then the rule $unify_c$ is applicable to $\overline{\Delta}$:

1. If τ_0=s or $\tau \neq s$ and $\tau \neq a_0$, where $s \in TypeConstant$, then application of the rule $unify_c$ fails;

2. If τ_0=$(s \to \tau')$ and τ=s, where $s \in TypeConstant$ and $\tau' \in CType$, then the result of applying the rule $unify_c$ is
σ=$\vec{e}/\{a_0 := \tau', e_1 := \{a_0 := e_1a_0, e_1 := e_1e_1\varepsilon, e_2 := e_1e_2\varepsilon\}, e_2 := \{a_0 := e_2a_0, e_1 := e_2e_1\varepsilon, e_2 := e_2e_2\varepsilon\}\}$;

3. If τ_0=$(s \to \tau')$ and τ=a_0, where $s \in TypeConstant$ and $\tau' \in CType$, then the result of applying the rule $unify_c$ is
σ=$\vec{e}/\{a_0 := \tau', e_1 := \{a_0 := e_1a_0, e_1 := e_1e_1\varepsilon, e_2 := e_1e_2\varepsilon\}, e_2 := \{a_0 := s, e_1 := e_2e_1\varepsilon, e_2 := e_2e_2\varepsilon\}\}$.

In cases 2 and 3 application of the rule $unify_c$ is written as $\overline{\Delta} \xrightarrow{unify_c} \sigma$.

Now let us explain the meaning of the rule $unify_c$. Let (M_1M_2) be a subterm of some term M, where $M_1, M_2 \in Term$. During the work of the type inference algorithm the corresponding skeleton of that subterm can be $(e_1P_1e_2P_2)^{:a_0}$,

where $P_1, P_2 \in Skeleton$ and $type(P_1) = \tau_0 \in CType$. The singular constraint corresponding the skeleton mentioned above will be $\vec{e}(e_1\tau_0 \doteq (e_2\tau \to a_0))$, where τ corresponds to the type of M_2, and τ_0 corresponds to the type of M_1 in the current stage of work of the type inference algorithm. After applying substitution σ the type of (M_1M_2) will be replaced with τ', and the type of M_2 will be replaced with s, if needed, and skeleton mentioned above will have the following form: $(P_1'P_2')^{:\tau'}$, where $type(P_1') = (s \to \tau')$ and $type(P_2') = s$.

Next lemma shows, that the substitution created by the rule $unify_\beta$, $unify_x$ or $unify_c$ solves the corresponding singular constraint.

Lemma 2.7. Let $\overline{\Delta}$ be a singular constraint, to which the rule $unify_y$ is applicable and $\overline{\Delta} \xrightarrow{unify_y} \sigma$, where $y \in \{\beta, x, c\}$. Then $[\sigma]\overline{\Delta}$ is solved.

Let us now consider singular constraints that are a part of constraint corresponding to some initial skeleton. Next lemma shows, that one and only one unificitation rule is applicable to each of that singular constraints.

Lemma 2.8. Let $M \in Term$, $P=initial(M)$ and $constraint(P) = \overline{\Delta}_1 \cap \ldots \cap \overline{\Delta}_n$, $n \geq 1$, where $\overline{\Delta}_i$ is a singular constraint, i=1,…,n. Then one and only one rule from rules $unify_\beta, unify_x$ and $unify_c$ is applicable to each $\overline{\Delta}_i$, $i = 1,\ldots,n$.

2.5. Unification algorithm. The unification algorithm tries to solve the given constraint that initially corresponds to some initial skeleton. It is called from the type inference algorithm and in fact is does the main work of type inference.

Definition 2.24. Let $\Delta=\overline{\Delta}_1 \cap \ldots \cap \overline{\Delta}_n \in Constraint$, $n \geq 1$, where $\overline{\Delta}_1,\ldots,\overline{\Delta}_n$ are singular constraints and $E-Path(\overline{\Delta}_i) \neq E-Path(\overline{\Delta}_j)$, i, j=1,…,n. Then leftmost/outermost constraint of Δ, written as $LO(\Delta)$, is a singular constraint from $\overline{\Delta}_1,\ldots,\overline{\Delta}_n$ that has the least E-path, i.e. $LO(\Delta) = \overline{\Delta}_k$, where $k \in \{1,\ldots,n\}$ and $E-Path(\overline{\Delta}_k) \prec E-Path(\overline{\Delta}_i)\ \forall i \in \{1,\ldots,n\} \setminus \{k\}$.

Definition 2.25. Let $\Delta=\overline{\Delta}_1 \cap \ldots \cap \overline{\Delta}_n \in Constraint$, $n \geq 1$, where $\overline{\Delta}_1,\ldots,\overline{\Delta}_n$ are singular constraints and $E-Path(\overline{\Delta}_i) \neq E-Path(\overline{\Delta}_j)$, i, j=1,…,n. Then rightmost/innermost constraint of Δ, written as $RI(\Delta)$, is a singular constraint from $\overline{\Delta}_1,\ldots,\overline{\Delta}_n$ that has the greatest E-path, i.e. $RI(\Delta) = \overline{\Delta}_k$, where $k \in \{1,\ldots,n\}$ and $E-Path(\overline{\Delta}_i) \prec E-Path(\overline{\Delta}_k)\ \forall i \in \{1,\ldots,n\} \setminus \{k\}$.

Let us explain the meaning of $LO(\Delta)$ and $RI(\Delta)$. Looking at the type inference rules, we can say that a new singular constraint is added to the constraint part of skeleton only after applying rule [APP]. Hence each singular constraint corresponds to one subterm of the form (M_1M_2), where $M_1, M_2 \in Term$. Let us mention without proving that $LO(\Delta)$ corresponds to the leftmost, outermost subterm of the form (M_1M_2) and $RI(\Delta)$ corresponds to the rightmost, innermost subterm of the form (M_1M_2).

Definition 2.26. Let $\Delta=\overline{\Delta}_1 \cap \ldots \cap \overline{\Delta}_n \in Constraint$, $n \geq 1$, where $\overline{\Delta}_1,\ldots,\overline{\Delta}_n$ are singular constraints, $I=\{i|, 1 \leq i \leq n$, and rule $unify_\beta$ is applicable to $\overline{\Delta}_i\}$. Then $filter_\beta(\Delta)=\bigcap_{i\in I}\overline{\Delta}_i$ (we suppose that $filter_\beta(\Delta)=\omega$ for $I=\varnothing$).

Algorithm of unification(Unify).

Input: constraint Δ such that $solved(\Delta)=\omega$.

Output: either returns the substitution that solves constraint or fails, or never returns.

1. If $\Delta=\omega$, then return ε.

2. If $filter_\beta(\Delta)\neq\omega$, then $LO(filter_\beta(\Delta)) \xrightarrow{unify_\beta} \sigma$ and returns $[Unify(unsolved([\sigma]\Delta))]\sigma$.

3. If the rule $unify_x$ is applicable to $RI(\Delta)$, then $RI(\Delta) \xrightarrow{unify_\beta} \sigma$ and returns $[Unify(unsolved([\sigma]\Delta))]\sigma$.

4. If the rule $unify_c$ is applicable to $RI(\Delta)$ and this application doesn't fail, then $RI(\Delta) \xrightarrow{unify_\beta} \sigma$ and returns $[Unify(unsolved([\sigma]\Delta))]\sigma$, else fail.

Lemma 2.9 (correctness of the unification algorithm). Let $M \in Term$ and $\Delta=constraint(initial(M))$. Then if $Unify(\Delta)=\sigma$, $[\sigma]\Delta$ is solved.

It is easy to see that the unification algorithm first tries to solve singular constraints, to which the rule $unify_\beta$ is applicable. It means that during his work the unification algorithm does implicit β-reductions in initial term until reducing the initial term to β-normal form, which happens when the unification algorithm first time arrives in point 3 or ends his work at point 1.

Lemma 2.10. Let $M \in Term$, $P=initial(M)$, $M \twoheadrightarrow_\beta M' \in \beta-NF$ and $\Delta=constraint(P)$. Then for input Δ the unification algorithm does a finite number of recursive calles from point 2 and tries to return the following: $[Unify(\Delta')][\sigma_m]\ldots[\sigma_2]\sigma_1$, where $\sigma_1,\ldots,\sigma_m$ are created by the rule $unify_\beta$ during the work of the algorithm and $\Delta'=constraint(initial(M'))$.

Remark 2.2. It is very important that in point 2 the unification algorithm applies the rule $unify_\beta$ to $LO(filter_\beta(\Delta))$. This choice ensures that in each step of the implicit β-reduction the unification algorithm will treat the leftmost, outermost β-redex. It is known that in this case β-normal form is reachable, if it exists.

Lemma 2.11. Let $M \in Term$, $P=initial(M)$, $\Delta=constraint(P)$ and M hasn't a β-normal form. Then for input Δ the unification algorithm will infinitely call himself recursively in point 2 and will never return.

Now let us consider the situation, when the term has a β-normal form. In this case we have no singular constraint, to which the rule $unify_\beta$ is applicable, and the unification algorithm tries to solve singular constraints, to which the rule $unify_x$ or rule $unify_c$ is applicable. It is important that in each step of work the unification algorithm tries to solve the singular constraint $RI(\Delta)$. This choice ensures that

during the next recursive calls of unification algorithm the rule $unify_x$ or the rule $unify_c$ will be applicable to each singular constraint.

Lemma 2.12. Let $M \in Term$, $P=initial(M)$, $\Delta=constraint(P)$ and $M \in \beta-$ $-NF$. Then for input Δ the unification algorithm does a finite number of recursive calls from point 3 or 4, and succeeds in point 1 or failed in point 4, because the application of the rule $unify_c$ was failed.

2.6. Type inference algorithm. Now let us present the type inference algorithm.

Type inference algorithm(Typify).

Input: term M.

Output: either returns the typing of M or fails, or never returns.

1. $P=initial(M)$. 2. $\sigma=Unify(constraint(P))$. 3. Return $([\sigma]env(P) \vdash [\sigma]type(P))$.

Theorem 2.1 (correctness of the Typify algorithm).

Let $M \in Term$. Then if $Typify(M) = (A \vdash \tau)$, then $(A \vdash \tau)$ is the typing of term M.

Received 06.04.2009

REFERENCES

1. **Carlier S., Wells J.B.** Type Inference with Expansion Variables and Intersection Types in System E and an Exact Correspondence with β-eduction. In Proc. 6th Int'l Conf. Principles & Practice Declarative Programming, 2004.
2. **Milner R.** Journal of Computer and System Sciences, 1978, № 17, p. 348–375.
3. **Wells J.B.** The Essence of Principal Typings. In Proc. 29th Int'l Coll. Automata, Languages and Programming. Springer-Verlag, 2002, v. 2380 of LNCS.
4. **Jim T.** What are Principal Typings and What are They Good for? In Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs., 1996.
5. **Carlier S., Polakow J., Wells J.B., Kfoury A.J.** System E: Expansion Variables for Flexible Typing with Linear and Non-linear Types and Intersection Types. In Programming Languages & Systems, 13th European Symp. Programming. Springer-Verlag, 2004, v. 2986 of LNCS.
6. **Barendregt H.P.** The Lambda Calculus: Its Syntax and Semantics. Amsterdam, North Holand, 1981.

Proc. of the Yerevan State Univ. Phys. and Mathem. Sci., 2009, № 3, p. 42–51.

λ-թերմերի տիպային կոռեկտության մասին: 1

Աշխատանքում դիտարկվում են պոլիմորֆ λ-թերմերը, որոնցում չկա ինֆորմացիա փոփոխականների տիպերի մասին: Աշխատանքի նպատակն է ընդլայնել այդպիսի թերմերի տիպայնացման ալգորիթմը [1] տիպերի հաստատունների և թերմերի հաստատունների գաղափարների ներմուծմամբ:

О типовой корректности полиморфных λ-термов. 1

В работе рассматриваются полиморфные λ-термы, в которых отсутствует информация о типах переменных. Цель даной работы – расширить алгоритм типизации таких термов [1] введением понятий констант типов и констант термов.