Type Inference System of Polymorphic λ-Terms

Ara, Arakelyan

Yerevan State University
Yerevan, Armenia

e-mail: ara_arakelyan@yahoo.com

ABSTRACT

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 and to prove that this algorithm typifies such terms in most common way.

Keywords

Type, expansion, skeleton, constraint, term.

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 well known Hindley/Milner system [2], used in languages such as Haskell, SML, OCaml etc. Important property of type systems is property of *principal typings* [3, 4], which allows compiler to do *compositional analysis*, i.e. analysis of modules without knowledge about other modules [3, 4]. Unfortunately Hindley/Milner system doesn't support property of *principal typings* [3]. In this paper we consider type inference system called $System\ E$ [1, 5]. In section 2 extended version of $System\ E$ is presented, which adds type constants and term constants to the original $System\ E$. In section 3 type inference algorithm and the main theorem of this paper is presented.

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, $Constant$ be a countable set of constants, $TermVariable$ be a countable set of term variables 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 $e \in ExpansionVariable$, $\tau \in Type$, then $e\tau \in Type$;
5. If $\tau_1, \tau_2 \in Type$, then $(\tau_1 \to \tau_2) \in Type$, $(\tau_1 \cap \tau_2) \in Type$;
The set of expansions $Expansion$ is defined as follows:
1. $\omega \in Expansion$;
2. If σ is a substitution (we will define substitutions later), then $\sigma \in Expansion$;
3. If $e \in ExpansionVariable, E \in Expansion$, 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 $M \in Term, x \in TermVariable$, then $(\lambda x.M) \in Term$;
4. If $M_1, M_2 \in Term$, then $(M_1 M_2) \in Term$;
The set of constraints $Constraint$ is defined as follows:
1. $\omega \in Constraint$;
2. If $\tau_1, \tau_2 \in Constraint$, then $(\tau_1 \doteq \tau_2) \in Constraint$;
3. If $e \in ExpansionVariable, \Delta \in Constraint$, 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 $c \in Constant, \tau \in Type$, then $c^{:\tau} \in Skeleton$;
3. If $x \in TermVariable, \tau \in Type$, then $x^{:\tau} \in Skeleton$;
4. If $x \in TermVariable, Q \in Skeleton$, then $(\lambda x.Q) \in Skeleton$;
5. If $e \in ExpansionVariable, Q \in Skeleton$, then $eQ \in Skeleton$;
6. If $Q_1, Q_2 \in Skeleton$, then $(Q_1 \cap Q_2) \in Skeleton$;
7. If $Q_1, Q_2 \in Skeleton, \tau \in Type$, 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 $\tau_1, \ldots, \tau_n \in Type,\ \alpha_1, \ldots, \alpha_n \in TypeVariable,\ E_1, \ldots, E_m \in Expansion,\ e_1, \ldots, e_m \in ExpansionVariable,\ n \geq 0,\ m \geq 0$. 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. $i \neq j \Rightarrow \alpha_i \neq \alpha_j,\ i, j = 1, \ldots, n$;
2. $i \neq j \Rightarrow e_i \neq e_j,\ i, j = 1, \ldots, m$.
By ε we will denote 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 $\vec{e}$.

Now let us define order on the set of E-paths.

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 $e_1 \preceq e_2 (e_1 \succeq e_2)$. Else if $n \leq m$, then $e_1 \preceq e_2$, else $e_1 \succeq e_2$. 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 is constructed without using operation $\cap$.

Remark 2.1. Taking into account definition of constraints, it is easy to see that each constraint is 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: $E\text{-}Path(\vec{e}(\tau_1 \doteq \tau_2)) = \vec{e}$.

Definition 2.6. The 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 Δ, written $unsolved(\Delta)$, is the smallest part of a Δ such that $\Delta = unsolved(\Delta) \cap \Delta'$ and Δ' is solved. Consequently Δ' will be the greatest solved part of a Δ, which 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, which contains all information about 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 the 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 is 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. If $e := E \in \sigma$, then $[\sigma]eY = [E]Y$;
5. If $e := E \notin \sigma\ \forall E \in Expansion$, then $[\sigma]eY = eY$;
6. $[\sigma]\omega = \omega$; 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. $[\sigma]\{\alpha_1 := \tau_1, \dots, \alpha_n := \tau_n, e_1 := E_1, \dots, e_m := E_m\} = \{\alpha_1 := [\sigma]\tau_1, \dots, \alpha_n := [\sigma]\tau_n, e_1 := [\sigma]E_1, \dots, e_m := [\sigma]E_m\} \cup \{\alpha := \tau | \alpha \notin \{\alpha_1, \dots, \alpha_n\}$ and $\alpha := \tau \in \sigma\} \cup \{e := E | e \notin \{e_1, \dots, e_m\}$ and $e := E \in \sigma\}$;
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. $[\sigma](\lambda x.Q) = (\lambda x.[\sigma]Q)$;
14. $[\sigma]c^{:\tau} = c^{:[\sigma]\tau}$; 15. $[\sigma](Q_1Q_2)^{:\tau} = ([\sigma]Q_1[\sigma]Q_2)^{:[\sigma]\tau}$,
where $\alpha_1, \dots, \alpha_n, \alpha \in TypeVariable, c \in Constant$, $e_1, \dots, e_m, e \in ExpansionVariable, s \in TypeConstant$, $E_1, \dots, E_m, E \in Expansion, \tau_1, \dots, \tau_n, \tau, \tau_1, \tau_2 \in Type$, $Y, X_1, X_2 \in Type \cup Constraint \cup Expansion \cup Skeleton$, $Q, Q_1, Q_2 \in Skeleton, M \in Term, n, m \geq 0$ and $[E]Y$ will be defined by the next definition.

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 $\vec{e} = e_1e_2 \dots e_n$, then $\vec{e}/\sigma = e_1/e_2/\dots/e_n/\sigma\ n \geq 0$, where $e_1, \dots, e_n, e \in ExpansionVariable$ 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 $X \in Type \cup Constraint \cup Expansion \cup Skeleton$.

Definition 2.11. The total function $A : TermVariable \to Type$ is called environment, if the following set is finite: $\{x | x \in TermVariable$ and $A(x) \notin \omega\}$.

Environment A also can be written as a set of pairs: $A = \{(x, A(x)) | x \in TermVariable\}$.

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

Definition 2.12. The set $CType \subset Type$ is the least 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 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 a term, a skeleton, an environment, a type and a constraint, written $(M \triangleright Q) : (A \vdash \tau)/\Delta$, is called a judgement. The intended meaning of a judgement is that Q is a proof that M has typing $(A \vdash \tau)$, provided the constraint Δ is solved.

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

$$[\text{VAR}]\ \frac{}{(x \triangleright x^{:\tau}) : (env_\omega[x \to \tau] \vdash \tau)/\omega}$$

$$[\text{CONST}]\ \frac{}{(c \triangleright c^{:\tau}) : (env_\omega \vdash \tau)/\omega}, \text{ where } \tau = \Sigma(c)$$

$$[\text{OMEGA}]\ \frac{}{(M \triangleright \omega^M) : (env_\omega \vdash \omega)/\omega}$$

$$[\text{E-VAR}]\ \frac{(M \triangleright Q) : (A \vdash \tau)/\Delta}{(M \triangleright eQ) : (eA \vdash e\tau)/e\Delta}$$

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

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

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

Definition 2.15. The pair $(A \vdash \tau)$ of an environment and a type is called a typing of a term M if $\exists Q \in Skeleton$ and $\exists \Delta \in Constraint$ s.t. the 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 a term M if
1. $(A \vdash \tau)$ is a typing of M;
2. 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 a term are obtained from principal typing by means of expansion application.

Next lemma shows that each skeleton contains information about one and only one inferable judgement.

Lemma 2.1. Let $Q \in Skeleton$. Then there exist one and only one term M, environment A, type τ and constraint Δ such that the judgement $(M \triangleright Q) : (A \vdash \tau)/\Delta$ is inferable and $M = term(Q)$.
This lemma let us 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, which will be introduced in section 3.2, starts term typification by constructing initial skeleton of that term.

Definition 2.17. Let fix 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:
1. $initial(x) = x^{:a_0}$, where $x \in TermVariable$;
2. $initial(c) = c^{:\Sigma(c)}$, where $c \in Constant$;
3. $initial((\lambda x.M)) = (\lambda x.e_0 initial(M))$, where $x \in TermVariable$ and $M \in Term$;
4. $initial((M_1 M_2)) = (e_1 initial(M_1) e_2 initial(M_2))^{:a_0}$, where $M_1, M_2 \in Term$.

Lemma 2.2. Let $P = initial(M)$, where $M \in Term$. Then $solved(constraint(P)) = \omega$;

From lemma 2.2 it is easy to see that all singular constraints, which are part of $constraint(P)$, are unsolved, where P is an initial skeleton of some term. In section 3.2 we will see, that type inference algorithm tries to solve some singular constraints by applying substitutions on them and it starts solving process from singular constraints, which are part of $constraint(P)$. Unification rules, introduced in the next section, are used to produce substitutions for solving singular constraints.

2.4 Unification rules

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

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

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

Definition 2.21 ($unify_\beta$ rule). Let $\bar{\Delta} = \vec{e}(e_1(e_0\tau_0 \to e_0\tau_1) \doteq (e_2\tau_2 \to a_0))$ be a singular constraint, where $\tau_0 \in Type'$ and $\tau_1, \tau_2 \in Type$. Then rule $unify_\beta$ is applicable to $\bar{\Delta}$ and the result of 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 rule $unify_\beta$ is written as $\bar{\Delta} \xrightarrow{unify_\beta} \sigma$.

Now let us explain the meaning of rule $unify_\beta$.
Let $((\lambda x.M_1)M_2)$ be a subterm of some term M, where $x \in TermVariable$ and $M_1, M_2 \in Term$. Initial skeleton of that subterm will be $(e_1(\lambda x.e_0 P_1)e_2 P_2)^{:a_0}$, where $P_1 = initial(M_1)$ and $P_2 = initial(M_2)$. The part of $constraint(initial(M))$, that corresponds to the initial skeleton of subterm mentioned above will be $\vec{e}(e_1(e_0\tau_0 \to e_0\tau_1) \doteq (e_2\tau_2 \to a_0))$, where τ_1 corresponds to the type of M_1 ($\tau_1 = type(P_1)$) and 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 rule $unify_\beta$, type a_0 is associated with each free occurrence of variable x in term M_1. After applying substitution, all that a_0 type variables will be replaced with type τ_2 (this replacement is done by substitution created by function $Extract_S$) and type of x in term M_1 will be changed. The same type is obtained when applying substitution created by 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 of β-reduction. Next two lemmas show exact correspondence of rule $unify_\beta$ with β-reduction.

Lemma 2.3 (correspondence with β-reduction). Let $M \in Term$ and $P = initial(M)$. If $constraint(P) = \bar{\Delta} \cap \Delta'$, where $\bar{\Delta}$ is a singular constraint to which rule $unify_\beta$ is applicable and $\bar{\Delta} \xrightarrow{unify_\beta} \sigma$, then $\exists M' \in Term$ s.t. $constraint(P') = [\sigma]\Delta'$ and $env(P') = [\sigma]env(P)$ and $type(P') = [\sigma]type(P)$ and $M \to_\beta M'$, where $P' = initial(M')$.

Lemma 2.4 (correspondence with β-reduction). Let $M, M' \in Term$ and $P = initial(M)$ and $P' = initial(M')$. If $M \to_\beta M'$, then $\exists \bar{\Delta}$ singular constraint such that $constraint(P) = \bar{\Delta} \cap \Delta'$ and rule $unify_\beta$ is applicable to $\bar{\Delta}$ and $\bar{\Delta} \xrightarrow{unify_\beta} \sigma$ and $constraint(P') = [\sigma]\Delta'$ and $env(P') = [\sigma]env(P)$ and $type(P') = [\sigma]type(P)$.

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

Now let us explain the meaning of rule $unify_x$. Let $(M_1 M_2)$ be a subterm of some term M, where $M_1, M_2 \in Term$. During work of type inference algorithm corresponding skeleton of that subterm can be $(e_1 P_1 e_2 P_2)^{:a_0}$, where $P_1, P_2 \in Skeleton$ and $type(P_1) = a_0$. Singular constraint corresponding to the skeleton mentioned above will be $\vec{e}(e_1 a_0 \doteq (e_2\tau \to a_0))$, where τ corresponds to the type of M_2 and a_0 corresponds to the type of M_1 in the current stage of work of type inference algorithm. After applying substitution created by rule $unify_x$ type of M_1 will be replaced with $(e_2\tau \to a_0)$ and the skeleton mentioned above will have the following form: $(P'_1 e_2 P_2)^{:a_0}$, where $type(P'_1) = (e_2\tau \to a_0)$.
The third unification rule is called $unify_c$.

Definition 2.23 ($unify_c$ rule). Let $\bar{\Delta} = \vec{e}(e_1\tau_0 \doteq (e_2\tau \to a_0))$ be a singular constraint, where $\tau_0 \in CType$ and $\tau \in Type$. Then rule $unify_c$ is applicable to $\bar{\Delta}$:
1. If $\tau_0 = s$, where $s \in TypeConstant$, then application of rule $unify_c$ is failed;
2. If $\tau \neq s$ and $\tau \neq a_0$, where $s \in TypeConstant$, then application of rule $unify_c$ is failed;
3. If $\tau_0 = (s \to \tau')$ and $\tau = s$, where $s \in TypeConstant$ and $\tau' \in CType$, then the result of application of rule $unify_c$ is $\sigma = \vec{e}/\{a_0 := \tau', e_1 := \{a_0 := e_1a_0, e_1 := e_1e_1\varepsilon, e_2 := e_1e_2\varepsilon\}, e_2 := \{a_0 := e_2a_0, e_1 := e_2e_1\varepsilon, e_2 := e_2e_2\varepsilon\}\}$;
4. If $\tau_0 = (s \to \tau')$ and $\tau = a_0$, where $s \in TypeConstant$ and $\tau' \in CType$, then the result of application of rule $unify_c$ is $\sigma = \vec{e}/\{a_0 := \tau', e_1 := \{a_0 := e_1a_0, e_1 := e_1e_1\varepsilon, e_2 := e_1e_2\varepsilon\}, e_2 := \{a_0 := s, e_1 := e_2e_1\varepsilon, e_2 := e_2e_2\varepsilon\}\}$.
In cases 3 and 4 application of rule $unify_c$ is written as $\bar{\Delta} \xrightarrow{unify_c} \sigma$.

Now let us explain the meaning of rule $unify_c$. Let $(M_1 M_2)$ be a subterm of some term M, where $M_1, M_2 \in Term$. During work of type inference algorithm corresponding skeleton of that subterm can be $(e_1 P_1 e_2 P_2)^{:a_0}$, where $P_1, P_2 \in$ *Skeleton* and $type(P_1) = \tau_0 \in CType$. Singular constraint corresponding to the skeleton mentioned above will be $\vec{e}(e_1\tau_0 \doteq (e_2\tau \rightarrow a_0))$, 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 type inference algorithm. After applying substitution created by rule $unify_c$ type of (M_1M_2) will be replaced with τ' and type of M_2 will be replaced with s if necessary and the skeleton mentioned above will have the following form: $(P_1' P_2')^{:\tau'}$, where $type(P_1') = (s \rightarrow \tau')$ and $type(P_2') = s$. Next lemma shows, that the substitution created by rule $unify_\beta$, $unify_x$ or $unify_c$ solves the corresponding singular constraint.

Lemma 2.5. Let $\bar{\Delta}$ be a singular constraint to which rule $unify_y$ is applicable and $\bar{\Delta} \xrightarrow{unify_y} \sigma$, where $y \in \{\beta, x, c\}$. Then $[\sigma]\bar{\Delta}$ is solved.

3. TYPE INFERENCE ALGORITHM

3.1 Unification algorithm

Unification algorithm tries to solve given constraint that initially corresponds to some initial skeleton. Unification algorithm is called from type inference algorithm and in fact is doing the main work of type inference. Before presenting unification algorithm let us give some definitions.

Definition 3.1. Let $\Delta = \bar{\Delta}_1 \cap \ldots \cap \bar{\Delta}_n \in Constraint$ $n \geq 1$, where $\bar{\Delta}_1, \ldots, \bar{\Delta}_n$ are singular constraints and $E\text{-}Path(\bar{\Delta}_i) \neq E\text{-}Path(\bar{\Delta}_j)$ $i, j = 1, \ldots, n$. Then the leftmost/ outermost constraint of Δ, written $LO(\Delta)$, is a singular constraint from $\bar{\Delta}_1, \ldots, \bar{\Delta}_n$ that has the least E-path, i.e. $LO(\Delta) = \bar{\Delta}_k$, where $k \in \{1, \ldots, n\}$ and $E\text{-}Path(\bar{\Delta}_k) \prec E\text{-}Path(\bar{\Delta}_i)$ $\forall i \in \{1, \ldots, n\} \setminus \{k\}$.

Definition 3.2. Let $\Delta = \bar{\Delta}_1 \cap \ldots \cap \bar{\Delta}_n \in Constraint$ $n \geq 1$, where $\bar{\Delta}_1, \ldots, \bar{\Delta}_n$ are singular constraints and $E\text{-}Path(\bar{\Delta}_i) \neq E\text{-}Path(\bar{\Delta}_j)$ $i, j = 1, \ldots, n$. Then the rightmost/innermost constraint of Δ, written $RI(\Delta)$, is a singular constraint from $\bar{\Delta}_1, \ldots, \bar{\Delta}_n$ that has the greatest E-path, i.e. $RI(\Delta) = \bar{\Delta}_k$, where $k \in \{1, \ldots, n\}$ and $E\text{-}Path(\bar{\Delta}_i) \prec E\text{-}Path(\bar{\Delta}_k)$ $\forall i \in \{1, \ldots, n\} \setminus \{k\}$.

Let us explain the meaning of $LO(\Delta)$ and $RI(\Delta)$. Looking at type inference rules we can say that new singular constraint is added to the constraint part of the skeleton only after applying rule [APP]. Hence each singular constraint corresponds to the one subterm of the form (M_1M_2), where $M_1, M_2 \in Term$. Without proof let us mention 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 3.3. Let $\Delta = \bar{\Delta}_1 \cap \ldots \cap \bar{\Delta}_n \in Constraint$ $n \geq 1$, where $\bar{\Delta}_1, \ldots, \bar{\Delta}_n$ are singular constraints and $I = \{i | 1 \leq i \leq n$ and rule $unify_\beta$ is applicable to $\bar{\Delta}_i\}$. Then $filter_\beta(\Delta) = \bigcap_{i \in I} \bar{\Delta}_i$ (we suppose that $filter_\beta(\Delta) = \omega$ in that case when $I = \emptyset$).

Algorithm of unification(Unify).
Input: constraint Δ such that $solved(\Delta) = \omega$.
Output: returns substitution that solves constraint Δ 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 return $[Unify(unsolved([\sigma]\Delta))]\sigma$.
3. If rule $unify_x$ is applicable to $RI(\Delta)$, then $RI(\Delta) \xrightarrow{unify_x} \sigma$ and return $[Unify(unsolved([\sigma]\Delta))]\sigma$.
4. If rule $unify_c$ is applicable to $RI(\Delta)$ and this application is not failed, then $RI(\Delta) \xrightarrow{unify_c} \sigma$ and return $[Unify(unsolved([\sigma]\Delta))]\sigma$, else fail.

Lemma 3.1 (correctness of unification algorithm). Let $M \in Term$ and $\Delta = constraint(initial(M))$. Then if $Unify(\Delta) = \sigma$, then $[\sigma]\Delta$ is solved.

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

Remark 3.1. It is very important that in point 2 unification algorithm applies rule $unify_\beta$ to the $LO(filter_\beta(\Delta))$. This choice ensures that in each step of implicit *β-reduction* unification algorithm will treat the leftmost, outermost β-redex. It is known that in this case β-normal form is reachable if it is exists.

3.2 Type inference algorithm

Type inference algorithm(Typify).
Input: term M.
Output: returns 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 3.1 (correctness of Typify algorithm). Let $M \in Term$. Then if $Typify(M) = (A \vdash \tau)$, then $(A \vdash \tau)$ is a typing of a term M.

Now let us present the main theorem of this paper, which shows that typing returned by the type inference algorithm is a principal typing of a given term.

Theorem 3.2. Let $M \in Term$ and $\exists M' \in Term$ s.t. $M' \in \beta\text{-}NF$ and $M \twoheadrightarrow_\beta M'$. Then:
1. If exists typing of a term M' such that during inference of corresponding judgement rule [OMEGA] is not used, then $Typify(M)$ succeeds.
2. If $(A \vdash \tau) = Typify(M)$, then $(A \vdash \tau)$ is a principal typing of a term M.

Remark 3.2. Type inference algorithm returns principal typing of a term that has β-normal form, except that situations when it is not possible to type β-normal form of a given term without using rule [OMEGA]. Type inference algorithm never returns for terms that haven't β-normal form.

REFERENCES

[1] S. Carlier, J. B.Wells, "Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction.", *In Proc. 6th Int'l Conf. Principles & Practice Declarative Programming*, 2004.

[2] R. Milner, "Theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences, No. 17*, pp. 348-375, 1978.

[3] J. B. Wells, "The essence of principal typings.", *In Proc. 29th Int'l Coll. Automata, Languages, and Programming, vol. 2380 of LNCS. Springer-Verlag*, 2002.

[4] T. Jim, "What are principal typings and what are they good for?", *In Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[5] S. Carlier, J. Polakow, J. B. Wells, A. J. Kfoury, "System E: Expansion variables for flexible typing with linear and non-linear types and intersection types.", *In Programming Languages & Systems, 13th European Symp. Programming, vol. 2986 of LNCS. Springer-Verlag*, 2004.

[6] H. P. Barendregt, "The Lambda Calculus: Its Syntax and Semantics.", *Amsterdam, North Holand*, 1981.