一种带约束的多态类型系统[1]

郑红军[2]　张乃孝[3]

（北京大学）

摘要　本文讨论了一种带约束的多态类型系统，引入了约束类型。约束与全称量化的结合使得参数化多态函数的应用更安全，同时也为重载的表示和实现提供了一个新的途径，提高了类型表示的抽象度。本文讨论的类型系统具有两个不同层次的类型结构，约束的引入与消去是不同层次上的操作。最后，本文给出了类型检查算法 W_τ，并证明了此算法中约束的可满足性是可判定的。

关键词　多态　　约束类型　　类型系统　　类型检查

1　引言

多态是有力且灵活的描述机制，许多程序设计语言(如 ML, Ada)都提供了定义多态函数的能力。多态函数具有一定的开放性，是一种抽象的过程，可以应用于不同类型的数据。考虑用于比较整数和字符的的‘≤’函数，它们类型是

$$\leq : \mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool} \quad ; \quad \leq : \mathrm{Char} \times \mathrm{Char} \to \mathrm{Bool} \qquad (\bigstar 1)$$

现欲将‘≤’定义为比较两个同类型序列的多态函数，在现有的 ML 类型系统中，其类型一般表示为：

$$\leq : \forall a \,.\, \mathrm{seq}[a] \times \mathrm{seq}[a] \to \mathrm{Bool} \qquad (\bigstar 2)$$

(★2)中的‘≤’是一种参数化多态(parametric polymorphism)函数[1]，seq[a]表示元素类型为 a 的序列，类型变量 a 的全称量化可能会使此函数被应用于无法比较的序列，从而导致程序运行错误。问题在于，对类型变量 a 无约束的全称量化使得此函数可应用的类型范围过宽，这样就降低了程序的安全性。因此，有必要标识出这类参数化多态函数可能应用的类型范围，以达到安全性与灵活性的统一[11]。那么，哪些序列可以用‘≤’来比较呢？确切地说，是那些序列中的元素可用‘≤’比较的序列：

$$\begin{aligned} &\leq : \mathrm{seq}[\mathrm{Char}] \times \mathrm{seq}[\mathrm{Char}] \to \mathrm{Bool} \quad , \\ &\leq : \mathrm{seq}[\mathrm{Int}] \times \mathrm{seq}[\mathrm{Int}] \to \mathrm{Bool} \quad , \qquad (\bigstar 3) \\ &\leq : \mathrm{seq}[\mathrm{seq}[\mathrm{Char}]] \times \mathrm{seq}[\mathrm{seq}[\mathrm{Char}]] \to \mathrm{Bool} \quad , \\ &\qquad \ldots \quad \ldots \end{aligned}$$

(★1)显示出‘≤’在 Int 和 Char 上的重载，重载是一类特殊的多态(*ad-hoc* polymorphism)形式。重载函数对于不同类型的操作数具有不同的函数体，执行

---

[1]本项研究得到国家自然科学基金的资助.郑红军，1969 年生，1997 年在北大计算机系获博士学位，主要研究领域为软件方法学和程序设计语言.e-mail:zhj@iist.unu.edu. 张乃孝，1942 年生，教授，博士生导师，主要研究领域为软件方法学和程序设计语言.联系地址：北京大学数学科学院信息科学系(北京 100871) .
e-mail:naixiao@pku.edu.cn.

不同的代码；而参数化多态函数对于不同类型的操作数只有一个函数体，执行同一段的代码[1]。Int 和 Char 上‘≤’的重载使得(★2)所示的参数化多态函数变为(★3)中的重载函数，以至于不能以一般的参数化多态形式统一地给出这些函数的类型，如果象(★3)中那样将所有这些重载函数的类型都显式地表示出来又过于繁琐，也不可能。

Jones 在[12]中给出的资格类型(qualified types)可用于标识参数化多态函数的应用类型范围和表示类似于(★3)的重载函数。资格类型的形式为：$\forall a \,.\, \pi(a) \Rightarrow T(a)$意为满足谓词$\pi(a)$的所有类型 $T(a)$。本文所讨论的约束类型是资格类型的一种具体形式：$\pi(a)$为 $T(a)$的约束。通过约束与全称量化的结合，(★3)中所有‘≤’重载函数的类型可以统一地表示为：

$$\leq : \forall a \,.\, (\leq : a \times a \rightarrow \text{Bool}) \,.\, \text{seq}[a] \times \text{seq}[a] \rightarrow \text{Bool} \qquad (\bigstar 4)$$

意为满足约束$\leq : a \times a \rightarrow \text{Bool}$的所有函数$\leq : \text{seq}[a] \times \text{seq}[a] \rightarrow \text{Bool}$，换句话说，(★2)所示的序列比较函数的应用条件为：必须在a类型上提供‘≤’运算，否则不可应用。约束与全称量化的结合弥补了参数化多态表示的类型范围过宽，而简单的重载表示的类型范围又可能过窄的不足。

关于约束类型方面的研究，可查的文献不多，但也有一些有益的研究成果。Jones[12]在研究 Haskell 语言的基础上，讨论了使多态 l-演算和 ML 类型系统支持资格类型的一般方法；Smith[7]将这种带约束的多态机制称为受限的多态(bounded polymorphism)，重点讨论了在 Damas/Milner 类型系统[6]中扩充入约束类型后，如何保持原有类型系统的性质；Palsberg 在[13]中以 l-演算为工具，在无全称量化的条件下，研究了面向对象语言中具有子类型关系的约束类型及其表达能力。

我们在研究语言的抽象与封装机制 **Garment**[9][10]中根据需要引入了约束类型，讨论了约束在语言和程序不同层次上的引入和消去，设计了带约束类型的类型检查算法，并且解决了约束的可满足性判定问题。为清晰起见，本文以 *Exp* 语言[5]为对象，抽象地介绍了上述工作。

2 *Exp* 语言

考虑 l-提升[8]后的 *Exp* 程序，它由一组函数定义和程序体(一个待求值的表达式)组成。*Exp* 程序的结构可以抽象地表示为：

$$\begin{aligned}
&f_1 \quad x_1 \quad \ldots \quad x_{n1} \quad = \quad e_1 \quad ; \\
&f_2 \quad x_1 \quad \ldots \quad x_{n2} \quad = \quad e_2 \quad ; \\
&\qquad\qquad \ldots \quad \ldots \\
&f_m \quad x_1 \quad \ldots \quad x_{nm} \quad = \quad e_m \quad ; \\
&e_0 \qquad\qquad\qquad (\text{程序体})
\end{aligned}$$

表达式 $e_i\ (0 \leq i \leq m)$不含 l-抽象，也就是没有局部函数定义；函数的定义部分的各函数允许相互递归，也允许函数的重载定义。给定一组标识符$(f\,,\, y\,,\, a\,,\, \leq\,,\, 1\,,\, \ldots)$，表达式 e 由如下的语法生成：

e ::= x | $x(e_1, \ldots, e_n)$ | **let** $x=e_1$ **in** e_2 | **if** e_1 **then** e_2 **else** e_3

在此，x 为标识符(包含常量)；$x(e_1, \ldots, e_n)$表示函数 x 的应用；let-表达式表示局部标识符 x 在其作用域 e_2 内的值为 e_1；if -表达式即为一般的条件表达式。

上述程序结构显示出函数的定义与函数的使用是分离的。我们约定，函数定义部分允许定义多态函数，函数体 $e_1, \ldots, e_m$ 中表达式的类型可以带类型变量，故函数 $f_1, \ldots, f_m$ 的类型亦可象(★4)中那样含类型变量；程序体 e_0 中的每个表达式具有唯一确定的类型(不含类型变量)，因此 e_0 中表达式的类型是单态的。ML 中典型的多态 *let* 表达式[3][6]，如：

let $f = lx\,.\,x$ **in** $(\ldots f(1) \ldots f(true) \ldots)$

以这种程序结构可描述为

$f\,x = x$; $(\ldots f(1) \ldots f(true) \ldots)$

Exp 中的多态是函数定义级别上的多态，与 ML 中的多态具有同样的表达能力。下面将首先讨论函数定义部分的类型结构，然后给出程序体部分的类型结构。这两部分构成了 *Exp* 的类型系统。

3 函数定义部分的类型结构

函数定义部分的类型分为三类：

① 数据类型 g ::= a | Tid | $c\,[g_1, \ldots, g_j]$ | $g_1 \times \ldots \times g_n \to g$

其中，a 为类型变量；Tid 为不含类型变量的类型常量，包括语言中的初等类型(Int，Real，Bool，Char 等)及为某类型引入的新名字；c 为类型符，其中的 $g_1, \ldots, g_j$ 为 c 的类型参数，如(★2)中的 seq[a]。

② 约束类型 r ::= $(x : g)\,.\,r$ | g $(x : g)$称为一个约束。

③ 量化类型 s ::= $\forall a\,.\,s$ | r (a 为 s 中的全称量化类型变量)

不失一般性，我们有时将 s 写成 $\forall \overline{a}\,.\,C\,.\,g$ 的形式，其中，$\overline{a}$ 为 s 中全称量化类型变量 $a_1, \ldots, a_n$ 的简写，C 表示 s 的约束部分 $(x_1 : g_1, \ldots, x_n : g_n)$，$g$ 称为 s 的体。若没有量化的类型变量或约束，$\forall \overline{a}$ 和 C 这两部分均可缺省。

类型系统的作用不仅在于所能表示出的类型，也在于它所能表示的类型间的关系。上述三类类型中的函数构造符'→'、笛卡尔乘积'×'、全称量词'∀'以及约束(constraint)表示了类型间的关系，这种能力蕴涵着某种在类型上的计算能力。一个类型系统可以用于证明表达式 $e : s$，表示 e 的类型为 s。一般来说，$e : s$ 依赖于 e 中标识符的类型，标识符的类型收集于一个类型假设集合中。一个类型假设集合 A 是由形为 $x : s$ 的类型假设构成的有限集合，有时也称该集合为类型环境[4]。

如果 A 中存在一个关于 x 的类型假设，或 A 中某个类型假设的一个约束为 $x : g$，则说 x 出现于 A。假设 $s = \forall a_1, \ldots, a_n\,.\,r$，$b$ 是 r 中的类型变量，如果 $b \notin \{a_1, \ldots, a_n\}$，则说 b 在 s 中是自由的。$free(s)$ 表示由出现于 s 中的自由类型变量构成的集合，$A \setminus x$ 表示除去 A 中所有关于 x 的类型假设。类型规则中 $A \vdash e : s$

表示由类型假设集合 A 可以得出 e 具有类型 s。一种简单的情况是：

$$[\text{Ide}] \qquad A \vdash x : s \qquad \text{if} \quad x : s \in A$$

函数定义部分所定义的函数由函数构造符'→'表示其类型，对于函数定义

$$f \quad x_1, \ldots, x_n \quad = \quad e$$

'→'的引入规则为：

$$[\rightarrow\text{Intro}] \qquad \frac{A \cup \{x_1 : g_1, \ldots, x_n : g_n\} \vdash e : \forall \overline{a} .C. g}{A \vdash f : \forall \overline{a} .C. (g_1 \times \ldots \times g_n \rightarrow g)}$$

其中，$x_1, \ldots, x_n$ 不出现于 A，$A \cup \{x_1 : g_1, \ldots, x_n : g_n\} \vdash e : \forall \overline{a} . C . g$ 隐含着对函数体 e 的类型推理过程，具体的类型推理规则及算法参见文献[7][12]。在此，我们仅考虑与约束类型相关的类型规则。

[→Intro]中类型变量 $\overline{a}$ 及全称量词'∀'的引入体现了 Exp 类型。系统的参数化多态性，'∀'的引入规则为：

$$[\forall\text{-Intro}] \qquad \frac{A \vdash e : s}{A \vdash e : \forall a .s}$$

[→Intro]中约束 C 限定了函数 f 的应用类型范围，如 (★4)。约束的引入规则是：

$$[\text{Cons-Intro}] \quad \frac{A \cup \{x : g\} \vdash e : r}{A \vdash e : (x : g). r} \qquad (x :_o s \in A)$$

类型环境 A 中可能有多个关于同一标识符 x 的类型假设，如(★1)、(★3)中的'≤'，这种情况即为 x 在 A 中重载，表示为 $x :_o s$。实际上，$x :_o s$ 中的 s 反映了 x 之重载的一般结构。如果 $x :_o s \in A$，则 A 中就可能含有多个关于 x 的类型假设 $(x : s_1, \ldots, x : s_n)$，这些类型假设满足：

(a) $s \geq_A s_i$ ，$1 \leq i \leq n$ (b) s_i 两两非重叠(nonoverlapping)

条件(a)中的 $s \geq_A s_i$ 表示在类型环境 A 中 s_i 是 s 的一个实例，$\geq_A$ 关系包含了[6]中的实例关系(instance relation)；条件(b)可以确保由类型 s_i 确定(重载的) x 是唯一的，这是实现重载的必要条件。$\geq_A$ 关系和类型重叠的具体定义如下：

定义($s \geq_A g$) 设 $s = \forall a_1, \ldots, a_n .C.g'$，且 $a_1, \ldots, a_n$ 中无重名,如果存在一个替换 $[g_1/a_1, \ldots, g_n/a_n]$ 使得

(a) $A \vdash C[g_1/a_1, \ldots, g_n/a_n]$，并且 (b) $g'[g_1/a_1, \ldots, g_n/a_n] = g$

则说 $s \geq_A g$。条件(a)表示 $C[g_1/a_1, \ldots, g_n/a_n]$ 中的所有约束在 A 中是可满足的。

定义($r \geq_A s'$) 如果对于任意 g 都满足：若 $s' \geq_A g$,则 $s \geq_A g$ ，则说 $s \geq_A s'$。

定义(约束的可满足性) 一个约束 $(x : g)$ 在 A 中是可满足的，当且仅当

(a) $(x : g) \in A$， 或者 (b) $\exists s'. (x : s') \in A$ 且 $s' \geq_A g$.

定义(重叠) 设 $s = \forall \overline{a} . C . g$ 与 $s' = \forall \overline{b} . C' . g'$，且 $free(g) \cap free(g') = \phi$. 如果 g 和 g' 可合一，则说 s 与 s' 是重叠的。其中的 $free(g) \cap free(g') = \phi$ 可通过

具有等价关系性质的α-变换[4]实现。

下面引入文献[5]中关于合一算法的定理，此算法将是类型检查算法的核心部分，此定理的证明参见文献[5]。

定理（合一算法） 存在一个以类型表达式g_1和g_2为参数的算法$\hat{A}$，若$\hat{A}$成功，就可以得到的g_1和g_2的最一般合一U，满足：

(a) $g_1 U = g_2 U$，即U合一g_1和g_2.

(b) 如果H是g_1和g_2的另一个合一替换，则存在一个替换S，满足$H=SU$.

例 1：对于A中$x :_o \forall\alpha.\alpha \times \mathrm{Int} \to \alpha$的两个类型假设：

$$x : \forall\alpha.(* : \alpha \times \alpha \to \alpha).\ \alpha \times \mathrm{Int} \to \alpha \qquad x : \mathrm{Real} \times \mathrm{Int} \to \mathrm{Real}$$

$\alpha \times \mathrm{Int} \to \alpha$与$\mathrm{Real} \times \mathrm{Int} \to \mathrm{Real}$在替换$[\mathrm{Real}/\alpha]$下可合一，那么$x$的这两个类型就是重叠的。我们称这种情况为重载函数类型假设的。

例 2：对于类型环境A_1中'$\le$'的如下重载：

$$\le : \mathrm{Char} \times \mathrm{Char} \to \mathrm{Bool} \quad , \quad \le : \mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}$$

$$\le : \forall\alpha.(\le : \alpha \times \alpha \to \mathrm{Bool}).\ \mathrm{seq}[\alpha] \times \mathrm{seq}[\alpha] \to \mathrm{Bool}$$

$$\le : \forall\alpha, \beta.(\le : \alpha \times \alpha \to \mathrm{Bool}, \le : \beta \times \beta \to \mathrm{Bool}).\ (\alpha \times \beta) \times (\alpha \times \beta) \to \mathrm{Bool}$$

即有$\le :_o \forall\alpha.\alpha \times \alpha \to \mathrm{Bool} \in A$，表示对其所有的实例都接受两个同一类型的参数，返回一个Bool类型的值。

例 3：如果在例 2 的A_1中定义函数

$$f\ x = x \le x$$

那么，可由下面对函数体$x \le x$的类型推理过程得到函数f的类型：

$A_1 \cup \{\le : \alpha \times \alpha \to \mathrm{Bool}, x : \alpha\} \vdash \le : \alpha \times \alpha \to \mathrm{Bool}$ [Ide]

$A_1 \cup \{\le : \alpha \times \alpha \to \mathrm{Bool}, x : \alpha\} \vdash x : \alpha$ [Ide]

$A_1 \cup \{\le : \alpha \times \alpha \to \mathrm{Bool}, x : \alpha\} \vdash x \le x : \mathrm{Bool}$ [$\to$E]

$A_1 \cup \{\le : \alpha \times \alpha \to \mathrm{Bool}\} \vdash f : \alpha \to \mathrm{Bool}$ [$\to$Intro]

$\le :_o \forall\alpha.\alpha \times \alpha \to \mathrm{Bool} \in A_1$ [Ide]

$A_1 \vdash f : (\le : \alpha \times \alpha \to \mathrm{Bool}).\ \alpha \to \mathrm{Bool}$ [Cons-Intro]

$A_1 \vdash f : \forall\alpha.(\le : \alpha \times \alpha \to \mathrm{Bool}).\ \alpha \to \mathrm{Bool}$ [$\forall$-Intro]

推理过程中用到的[$\to$E]规则为

$$[\to\mathrm{E}] \qquad \frac{A \vdash x : (g_1 \times \ldots \times g_n \to g) \quad A \vdash e_1 : g_1, \ldots, A \vdash e_n : g_n}{A \vdash x(e_1, \ldots, e_n) : g}$$

上面的推理过程显示出了约束在类型推理过程中引入的方法。在*Exp*程序的函数定义部分说明函数的类型时，可以不通过类型推理引入约束，一般是根据[Cons-Intro]规则从语用角度引入的，用于标识函数应用的类型范围。例 3 中的约束表明：函数$f : \alpha \to \mathrm{Bool}$只能应用于具有$\le : \alpha \times \alpha \to \mathrm{Bool}$运算的那些类型$\alpha$上。

4 对约束类型和重载的限制

Exp 程序的函数定义部分允许定义重载函数，这些重载函数定义所产生的类型假设要加入类型环境 A 中，由于约束的存在及重载实现上的限制，我们要求加入到 A 中的类型假设必须满足

(1) 加入此假设于 A 中不会产生重载函数类型假设的重叠。

(2) 对于类型假设 $x:\forall\,\overline{a}\,.C.\,g$，要求 C 中的每个约束 $h:g\,'$至少含有一个类型变量，并且至少有一个出现于g中。如果不满足，就可以通过检查 A 中关于 h 的所有类型假设来判断此约束是否可满足。例如，类型假设

$$x:\forall a\,.\,(*:a\times a\to a)\,.\,\mathrm{Real}\times\mathrm{Int}\to\mathrm{Real}$$

中，a只在约束$*:a\times a\to a$中出现。如果 A 中关于‘$*$’有如下形式的重载:

$$*:\mathrm{Int}\times\mathrm{Int}\to\mathrm{Int}\quad,\quad *:\mathrm{Real}\times\mathrm{Real}\to\mathrm{Real}$$

就可以通过将a分别替换为 Int 和 Real 消去此约束，于是，只需将类型假设 $x:\mathrm{Real}\times\mathrm{Int}\to\mathrm{Real}$加入 A 中；如果 A 中关于‘$*$’的一个类型假设与 x 的约束$*:a\times a\to a$不可合一或 A 中没有关于‘$*$’的类型假设，那么，上面 x 的类型假设就不能加入 A 中。显然，约束$*:a\times a\to a$对于类型环境 A 是多余的。

(3) 此假设的加入不会产生类型假设的循环序列。类型假设的一个循环序列具有如下形式：

$$x_0:\forall\,\overline{a}\,.(x_1:g_0).\ r_0\ \ ,x_1:\forall\,\overline{a}\,.(x_2:g_1).\ r_1\,,\ldots,x_{n-1}:\forall\,\overline{a}\,.(x_0:g_{n-1}).\ r_{n-1}$$

其中，n>1, x_i可以重名。n=1 的情况是下面的条件(4)。

(4) 如果待加入的是递归约束类型假设 $x:\forall\,\overline{a}\,.\,(x:g\,'\,,\ldots)\,.\,g$，如(★4)式，要求$g\,'$不是$g$的替换实例，即：不存在替换 S，使得$g\,S=g\,'$。不满足这个条件的递归约束类型假设在 A 中是没有意义的，因为无法判定约束 $x:g'$ 在 A 中是否可满足，具体证明见下面的命题。

命题 类型环境 A 满足上面的条件（1），若 $x:\forall\,\overline{a}\,.(x:g',\ldots).g\in$ A, 且 g'是g的一个替换实例，则无法判定约束 $x:g'$在 A 中的可满足性。

证明： 不失一般性，首先对 g'施行a－变换[4], 使得 $free(g)\cap free(g\,')=\Phi$。根据约束可满足性的定义，只能通过下面的三个途径来判定约束 $x:g'$ 在 A 中的可满足性：

<1> $x:g'\in$A: 由于g'是g的一个替换实例，即存在一个替换 S，使得gS＝g'。因 $free(g)\cap free(g\,')=\Phi$，所以，替换 S 可作为$g'$和$g$的合一替换，满足$g$S＝$g'$S，即：$g'$与$g$可合一。根据重叠的定义可得，类型假设 $x:\forall\,\overline{a}\,.(x:g',\ldots).g$与 $x:g'$重叠。因 $x:\forall\,\overline{a}\,.(x:g',\ldots).g\in$ A, 且 A 满足上面的条件（1），所以 $x:g'$不可能在 A 中，故 $x:g'\notin$A。

<2> $x:\forall\,\overline{b}\,.\mathrm{C}.g''\in$ A,且$\forall\,\overline{b}\,.\mathrm{C}.g''\geq_{\mathrm{A}}g'$: 由$\geq_{\mathrm{A}}$关系的定义及$a$－变换可得，$g'$与$g''$可合一。因$g'$是$g$的一个替换实例，故$g$与$g''$也可合一。于是，由类型假设的重叠和此命题的条件可得，类型假设 $x:\forall\,\overline{b}\,.\mathrm{C}.g''$不可能在 A 中。

<3> 由<1>和<2>可以看出，A 中能够用于判定约束 $x:g'$可满足性的类型假设只有此递归约束类型假设 $x:\forall\,\overline{a}.(x:g',\ldots).g$ 本身：判定 $\forall\,\overline{a}.(x:g',\ldots).g$ 与g' 间是否有$\geq_A$关系。这样就又回到了判定约束 $x:g'$ 的可满足性问题。

<1>，<2>和<3>表明：若g'是g的一个替换实例，则无法判定类型假设 $x:\forall\,\overline{a}.(x:g',\ldots).g$ 中的约束 $x:g'$在 A 中的可满足性。

为叙述方便，我们称上面的四个条件为 *TA*(Type Assumptions)条件。只有满足 *TA* 条件的类型假设才允许加入类型环境 *A* 中，称这种类型环境满足 *TA* 条件。*TA* 条件是对函数类型中引入约束和重载函数定义的限制，条件(3)和(4)是判定约束可满足性的必要条件。约束类型的引入提高了类型表示的抽象度和语言的简明性，但增加了类型检查的负担，因为要判定 *TA* 条件。形式地，可将上述 *TA* 条件定义为函数 *TA*（A，$x:s$）：Bool,并可作为 *TA* 条件中的判定算法。其中：A 为类型环境，$x:s$为欲加入 A 的一个类型假设。

TA（A，$x:s$）＝nonoverlapping（A，$x:s$） **and**

nonredundant （A，$x:s$） **and**

nonrecursive （A，$x:s$）

nonoverlapping（A，$x:s$）$=\forall\,x:s'\in A.\mathrm{R}(s',s)=$ **fail**

nonredundant （A，$x:s$）= **let** $\forall\,\overline{a}.\mathrm{C}.g=s$ **in**

$\forall$ h :$g'\in$ C,tvar(g') $\cap$ tvar{g } $\neq$ {}

其中 tvar(g)为类型g 中类型变量构成的集合，由第 3 节中g 的定义可得:

tvar(a)={a}

tvar(Tid)={}

tvar($x[g_1,\ldots,g_j]$)=tvar(g_1) $\cup\ldots\cup$ tvar(g_j)

tvar($g_1\times\ldots\times g_n\rightarrow g$)=tvar($g_1$) $\cup\ldots\cup$ tvar(g_n)

nonrecursive(A, x:s) =

let $\forall\,\overline{a}.\mathrm{C}.g=s$ **in**

$\forall$y:$g'\in$ C**. if** y=x **then** noninstance(A, x:s)

else $x\notin$ dependant(A,y)

dependant(A, y) =

let{y: $\forall\,\overline{a}.\mathrm{C}.g$} $\cup$ B=extract(A,y) **in**

dependantc (C, A) $\cup$ dependanta (B, A)

extract(A,y)={x:s | x:$s\in$ A **and** x=y}

dependantc(C,A)= **if** c={} **then** {} **else**

let {*x*:*g*} ∪ C'=C **in** {*x*} dependant(A,*x*) ∪ dependantc(C',A)

dependanta(B,A)=**if** B={} **then** {} **else**

let {z: $\overline{a}$.C.*g*} ∪ B'=B **in**

dependantc(C,A) ∪ dependanta(B',A)

noninstance(A,*x*:*s*)=**let** ∀$\overline{a}$.(*x*:*g*′).C.*g* =*s* **in**

R(*g*,*g*′)=**fail**.

5 程序体部分的类型结构

程序体部分表达式的类型均为基类型，即为不含类型变量的类型，我们以*t*来表示，其具体形式如下：

$$t \quad ::= \quad Tid \quad | \quad c\,[t_1,\ldots,t_n] \quad | \quad t_1\times\ldots\times t_n \to t$$

其中的 *Tid* 和*c*与在*g*中的 *Tid* 和*c*含义相同。程序体部分的类型规则如下：

[Var] $A \vdash x : t \quad (x : t \in A)$

[If] $$\frac{A \vdash e_1 : Bool, \quad A \vdash e_2 : t, \quad A \vdash e_3 : t}{A \vdash if\ e_1\ then\ e_2\ else\ e_3 : t}$$

[Let] $$\frac{A \vdash e_1 : t_1, \quad A \setminus x \cup \{x : t_1\} \vdash e_2 : t_2}{A \vdash let\ x = e_1\ in\ e_2 : t_2}$$

提升后 *Exp* 程序的结构使其程序体内无函数定义，只有函数应用，对应的类型规则为：

[→Elim] $$\frac{\begin{array}{l} A \vdash x : \forall \overline{a}.C.(g_1\times\ldots\times g_n \to g) \\ A \vdash e_1 : t_1, \ldots, A \vdash e_n : t_n \\ AU \vdash CU \end{array}}{A \vdash x(e_1, \ldots, e_n) : gU}$$

其中，$U = \hat{A}\,(g_1\times\ldots\times g_n\ ,\ t_1\times\ldots\times t_n)$ 。由[→Elim]规则可以看出，由于程序体中表达式的类型均为基类型，在程序体中必须消去全称量词‘∀’并以基类型实例化类型变量，其规则为：

[∀-Elim] $$\frac{A \vdash e : \forall a.s}{A \vdash e : s[t/a]}$$

函数定义中引入的约束在程序体中必须是可满足的，约束的可满足就意味着约束的可消去，在类型环境 *A* 中无法消去不可满足的约束。[→Elim]规则中的 *AU* ⊢*CU* 表明在程序体中不能应用带有不可消去约束的函数，在应用函数前要检查约束的可满足性以消去这些约束，约束的消去规则是：

[Cons-Elim] $$\frac{A \vdash e : (x : t)\,.\,r \quad , \quad A \vdash x : t}{A \vdash e : r}$$

例 4：以例 2 的类型环境 A_1 为基础，假设类型环境

$$B = A_1 \cup \{ f : \forall a\,.\,(\le : a\times a \to \mathrm{Bool})\,.\,a \to \mathrm{Bool} \quad , \quad p : \mathrm{seq[Int]}\ \}$$

那么，对于程序体中 *f* 的应用 *f* (*p*)有：

$B \vdash f : \forall a\ .\ (\leq : a \times a \to \text{Bool})\ .\ a \to \text{Bool}$ [Ide]

$B \vdash f : (\leq : \text{seq[Int]} \times \text{seq[Int]} \to \text{Bool})\ .\ \text{seq[Int]} \to \text{Bool}$ [∀-Elim]

$B \vdash \leq : \forall a\ .\ (\leq : a \times a \to \text{Bool})\ .\ \text{seq}[a] \times \text{seq}[a] \to \text{Bool}$ [Ide]

$B \vdash \leq : (\leq : \text{Int} \times \text{Int} \to \text{Bool})\ .\ \text{seq[Int]} \times \text{seq[Int]} \to \text{Bool}$ [∀-Elim]

$B \vdash \leq : \text{Int} \times \text{Int} \to \text{Bool}$ [Var]

$B \vdash \leq : \text{seq[Int]} \times \text{seq[Int]} \to \text{Bool}$ [Cons-Elim]

$B \vdash f : \text{seq[Int]} \to \text{Bool}$ [Cons-Elim]

$B \vdash p : \text{seq[Int]}$ [Var]

$B \vdash f(p) : \text{Bool}$ [→Elim]

6 类型检查算法 W_t

设 $Rule(P)$是与程序 P 中表达式相关的类型规则构成的集合，它限定了 P 中各表达式类型间的关系。在此关系下，类型检查与类型推理的含义是：

类型检查：给定一个类型 s 和表达式 e，判定是否有 $e{:}s \mid\!=_A Rule(P)$

类型推理：给定一个表达式 e，证明 $\exists s\ .\ e{:}s \mid\!=_A Rule(P)$

类型检查与类型推理的本质差别在于前者是判定已知的类型 s 是否满足 $Rule(P)$，后者是根据现有的类型环境计算出满足 $Rule(P)$ 的一个类型 s。在类型推理与类型检查之间由 s 中已知信息从少到多形成一个连续谱，其中间状态可以视之为部分类型推理，若 s 中的信息全部已知(s 是基类型)，则类型推理就转化为类型检查。

下面给出对 Exp 程序体的类型检查算法 $W_\tau(A, e)$，W_τ 以类型环境 A 和程序体中的表达式 e 为输入，返回 e 的类型 t 。由第 5 节的类型规则，以[2]中的方式可得 W_τ 的具体算法 $W_\tau(A, e)$ 如下：

1. e is "x"

if x is overloaded in A **then** *fail*
else if $x : t \in A$ **then** t.

2. e is "$x(e_1, \ldots, e_n)$"

let $t_1 = W_\tau(A, e_1)$, $t_2 = W_\tau(A, e_2)$, . . . , $t_n = W_\tau(A, e_n)$ **in**
  if $x : \forall a_1, \ldots, a_m . C.\ g_1 \times \ldots \times g_n \to g \in A$ **then**
  let $U = \hat{A}\ (t_1 \times \ldots \times t_n, g_1 \times \ldots \times g_n)$ **in**
  if $satisfiable(CU, AU)$ **then** gU
  else *fail*
 else *fail.*

3. e is "**if** e_1 **then** e_2 **else** e_3"

let $t = W_\tau(A, e_1), t_1 = W_\tau(A, e_2)$, $t_2 = W_\tau(A, e_3)$ **in**
  if t = Bool and $t_1 = t_2$ **then** t_1
 else *fail* .

4. e is "**let** $x = e_1$ **in** e_2"

let $t_1 = W_\tau(A\ ,\ e_1)$ **in** $W_\tau(A \setminus x \cup \{x : t_1\}\ ,\ e_2)$.

W_τ**中，若算法** $\hat{A}$ **失败，则** W_τ**失败；在此，等价采用名等价原则；**W_τ**中的函数** *satisfiable* **以约束集合** *Con* **和类型环境** *Ass* **为输入，用于判断约束的可满足性。由约束的可满足性定义，** *satisfiable* **的算法如下：**

satisfiable (*Con* , *Ass*) =

for all $x : t \in Con$ **do**
if $x : t \notin Ass$ **then**
if $x : \forall\, \overline{a}\ .Con'.\ g \in Ass$ **and** $U = \hat{A}\ (t\ ,\ g\ )$ **then**
return (*satisfiable* (*Con'U* , *AssU*))
else *return* (*false*)
else *return* (*true*) .

文献[7][12]**给出的类型推理算法中，约束的可满足性是不可判定的。可以证明，算法** W_τ**中约束的可满足性是可判定的。**

命题 (**约束可满足性的判定**)

给定一个类型假设集合 *Ass* **和约束集合** *Con*，**若** *Ass* **满足** *TA* **条件，则算法** *satisfiable* **一定能停机。即：约束集合** *Con* **中的约束在类型环境** *Ass* **下的可满足性是可判定的。**

证明：分别讨论调用 *satisfiable* **的二种情形：**

<1> **算法** W_τ**中的情形** 3 **以** *satisfiable*(*CSU* , *ASU*)**形式调用** *satisfiable* **函数，其中的替换** *S* **和** *U* **使得** *Con* **中的类型均为基类型，** *A* **中与** *C* **相关的类型在** *Ass* **中也都是基类型。考虑** *Con* **中的一个约束** $x : t$ **，由于** *Ass* **和** *Con* **都是有限集合，且** *t***为基类型，所以，只需通过** $x : t$**与** *Ass***中的类型假设有限次的匹配即可判定** $x : t$**是否在** *Ass* **中，如果** $x : t \in Ass$**，则说明** *Ass* **满足此约束，返回结果** *true***；如果** $x : t \notin Ass$**，则是下面的情况**(2)**。**

<2>**假设** *Ass* **中有** $x : \forall\, \overline{a}\ .Con'.\ g$**，且** $U=\hat{A}\ (t\ ,\ g\ )$**成功，算法** *satisfiable* **中将以** *satisfiable* (*Con'U* , *AssU*)**递归调用** *satisfiable* **函数，否则返回结果** *false***。下面的四个事实保证了上述递归调用不会形成死递归。**

- ✧ *Con'***和** *Ass* **是有限集合；**
- ✧ *Con'U* **中的类型均为基类型；**
- ✧ *Ass* **中与** *Con'***相关的类型在** *AssU* **中也都是基类型；**
- ✧ *AssU* **满足** *TA* **条件；**

<1>、<2>**表明：算法** *satisfiable* **一定能停机。即：约束集合** *Con* **中的约束在 类型环境** *Ass* **下的可满足性是可判定的。**

定理（算法 W_τ**的可靠性） 若** W_τ（A，e）$= t$ **,则有** A ⊢e：t 。

证明：对 e **的语法结构归纳：**

(1) e is “x”

算法 W_τ **若成功，即为：**

if(x**:** t)∈ A then t

由类型规则[Var]**:**

$\mathbf{A} \vdash x : t \quad (x : t \in \mathrm{A})$

显然，W_τ **成功的条件与类型规则**[Var]**的施用条件一致，故有：**

若 W_τ（A，x）＝ t **，则有** $\mathbf{A} \vdash e : t$

(2) e is "x(e_1,. . . , e_n)"

算法 W_τ**中,** $t_1 = W_\tau(\mathrm{A}, e_1), \ldots, t_n = W_\tau(\mathrm{A}, e_n)$

由归纳假设即有 $\mathbf{A} \vdash e_1 : t_1$ ， . . . ， $\mathbf{A} \vdash e_n : t_n$ （PRE1）

算法 W_τ **中若有**

$x : \forall a_1, \ldots, a_m.\mathrm{C}.g_1 \times \ldots \times g_n \to g \in A$

成立，则由归纳假设

$\mathbf{A} \vdash x : \forall a_1, \ldots, a_m.\mathrm{C}.g_1 \times \ldots \times g_n \to g$ （PRE2）

算法 W_τ **中，令** U=R(t_1,. . . , t_n ,$g_1 \times \ldots \times g_n$), **若** R **成功，则** U **即为关于类**

型变量a_1,. . . , a_m **的替换，于是即可籍此及** PRE2**，并由类型规则**[∀-Elim]

可得

$\mathbf{A} \vdash$ e : CU.($g_1 \times \ldots \times g_n$)U→ gU

算法 W_τ **中,satisfiable(CU,AU)用于判断约束** CU **是否可满足，若** satisfiable **成功，由归纳假设即有：**

AU ⊢ CU （PRE3）

由（PRE1**），（**PRE2**），（**PRE3**）及**[→Elim] **规则即可得：**

$\mathbf{A} \vdash$x(e_1,. . . , e_n) : gU

(3) e **is "if** e_1 **then** e_2 **else** e_3 "

算法 W_τ **中令** $t = W_\tau$（A, e_1），$t_1 = W_\tau$（A, e_2） $t_2 = W_\tau$（A,e_3） ， **故由归纳假设即有：**

$\mathbf{A} \vdash e_1 : t$, $\mathbf{A} \vdash e_2 : t_1$, $\mathbf{A} \vdash e_3 : t_2$

若 W_τ **成功，则必有** t=*Bool*, $t_1 = t_2$ **由此及类型规则**[If] ， **即有**

$\mathbf{A} \vdash$ **if** e_1 **then** e_2 **else** $e_3 : t_1$

(4) e **is "let** $x = e_1$ **in** e_2 "

算法 W_τ**若成功，则将返回** $W_\tau(\mathrm{A}\backslash x \cup \{x : t_1\}, e_2)$ **，其中** $t_1 = W_\tau(\mathrm{A}, e_1)$ **于**

是即有：

$\mathbf{A} \vdash e_1 : t_1$

$A\backslash x \cup \{x: t_1\} \vdash e_2: t_2$

由类型规则[Let]

$A \vdash \textbf{let}\ x: e_1\ \textbf{in}\ e_2: t_2$.

证毕。

显然，算法 W_τ**不具有完备性。如：由类型规则**[∀-Intro]**和**[Cons-Intro]**引入的关于表达式的新的类型假设，即无法由** W_τ **算法得出，其不具有完备性的根本原因在于** W_τ**是关于程序体部分（层次）的类型检查算法而非函数定义层次上的类型检查算法，而这两个层次的类型结构构成了** Exp **语言的类型系统，因此说算法** W_τ **对于** Exp **类型系统来说是不完备的，若能使** W_τ**亦具有函数定义层次上的类型推理能力，将可能会使** W_τ**具有完备性。这也是需进一步研究的问题。**

7 结语

本文以 *Exp* **语言为例，引入了约束类型,讨论了一种带约束的多态类型系统。约束与全称量化的结合使得参数化多态函数的应用更安全，同时也为重载的表示和实现提供了一个新的途径 — 多态重载**[12]**，提高了类型表示的抽象度。函数定义部分的类型符**c**在本文中只提供了一个序列类型符** seq**，实际上，容易在** *Exp* **语言中引入数据抽象机制(如**[1]**中的** *Fun*)**，这样就可以根据程序的需要定义类型符**[3]**。函数定义与函数应用分离的** *Exp* **程序结构以及** *Exp* **类型系统的两个不同层次为分离语言的定义与语言的应用提供了研究模型，也就是将语言的开发与程序的开发分离，旨在提高语言本身的简明性，降低程序设计的复杂度，这种两个不同层次的类型结构正是语言与程序不同层次的体现。我们将数据抽象思想与上述分离的思想相结合，提出了语言的一种抽象与封装机制 Garment**[9][10]**。本文讨论的类型系统可望对** Garment **的设计与实现有一定的指导意义。关于子类型约束的引入以及本文给出的多态类型系统在** Garment **中的进一步应用是需进一步研究的问题。**

参考文献

[1] Luca Cardelli & Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, Vol.17(4), p 471-525, 1985.

[2] Luca Cardelli. Basic Polymorphism Type Checking. *Science of Computer Programming*, Vol.8, p 147-172, 1987.

[3] R. Harper & J.C. Mitchell. On the Type Structure of Standard ML. *ACM Trans. on Programming Languages and Systems*, Vol.15(2), p 211-252, 1993.

[4] J.C. Mitchell. Type Systems for Programming Languages. *Handbook of Theoretical Computer Science*, p 364-458, 1990.

[5] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, Vol.17, p 348-375, 1978.

[6] L. Damas & R. Milner. Principal Type Schemes for Functional Programs. *Proc. of 19th ACM Symposium on Principles of Prog. Lang.*, p207-212, 1982.

[7] G. Smith. Polymorphism Type Inference with Overloading and Subtyping. LNCS668，Colloquium on TAPSOFT'93,p671-685,1993.

[8] T. Johnsson. Lambda Lifting - Transforming Programs to Recursive Equations. *LNCS* 201, *Functional Prog. Lang. and Computer Architecture*, p190-203, 1985.

[9] 张乃孝，许卓群，屈婉玲 面向模型的变换型软件开发方法研究，《*理论计算机科学*》(吴文俊主编) Vol.2, p 54-64, 上海科技文献出版社 1994.

[10] Zhang Naixiao, Zheng Hongjun and Qiu Zongyan. GARMENT——A Mechanism For Abstraction And Encapsulation Of Languages, *ACM SIGPLAN Notices*, p52-58, 1997.6.

[11] C. Ponder & B. Bush. Polymorphism Considered Harmful. *ACM SIGSOFT Software Engineering Notes*, Vol.19(2), p 35-37, 1994.

[12] M.P. Jones. A Theory of Qualified Types. *LNCS* 582, *Proc. of European Symposium on Programming*, p 287-306, 1992.

[13] J. Palsberg & S. Smith. Constrained Types and Their Expressiveness. *ACM Trans. on Programming Languages and Systems*, Vol.18(5), p 519-527, 1996.

A Polymorphic Type System with Constraints

Zheng Hongjun Zhang Naixiao

(Peking University)

Abstract This paper concentrates on a polymorphic type system with constraints based on constrained types. By incorporating constraints into unviversal quantification, the system can make applications of parametrically polymorphic function more safe. Constrained types provide a new way for expressing and implementing overloading. The incoporating can improve the expressiveness of types. There are two layers with different type structures in the type system given in this paper. Introduction and elimination of constraints are in the different levels. It is proved that the satisfiability of constraints in W_τ, which is a type checking algorithm proposed in the paper, is decidable.

Keywords Polymorphism Constrained type Type system Type checking