ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.7, July 2010, pp.1481−1490
doi: 10.3724/SP.J.1001.2010.03806

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

采用了剪枝优化的子类型关系判定算法*

戴晓君+, 陈海明

(中国科学院 软件研究所 计算机科学国家重点实验室,北京 100190)

Subtyping Algorithm with Pruning Optimization

DAI Xiao-Jun+, CHEN Hai-Ming

(State Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: E-mail: dai.xiaojun@gmail.com

Dai XJ, Chen HM. Subtyping algorithm with pruning optimization. *Journal of Software*, 2010,21(7): 1481−1490. http://www.jos.org.cn/1000-9825/3806.htm

Abstract: Statically typed XML processing languages show new ways of processing XML data. However, current languages are not efficient enough. This paper studies the decision problem of subtyping relation which is an important issue of the languages, and optimizes XDuce's subtyping algorithm with a pruning strategy. Experimental data show the efficiency of the algorithm increased 20% averagely. This optimization strategy can be applied to other languages which use similar subtyping algorithm.

Key words: XML; statically typed language; type checking; subtyping; algorithm optimization

摘 要: 静态类型化 XML 处理语言为处理 XML 数据提供了新的途径,但现有的此类语言大多数效率较低.研究此类语言的一个重要问题——子类型关系的判定,并使用剪枝优化策略对 XDuce 的子类型关系判定算法进行优化.实验数据显示,优化后算法的执行效率平均提高 20%.该策略具有普遍性,对所有使用类似算法的静态类型化 XML 处理语言都有效.

关键词: XML;静态类型化语言;类型检查;子类型关系判定;算法优化
中图法分类号: TP301 **文献标识码**: A

类型检查是编译器的重要组成部分.在编译时刻对本地类型进行类型检查,可以避免在运行时出现类型相关的错误,从而提高程序的鲁棒性.

XML 作为不同应用系统之间交换数据问题的开放解决方案,已被广泛用于 Web 服务、银行事务、电子商务等不同领域.XML 的流行,很大程度上得益于可以使用模式限制 XML 文档的结构[1].静态类型化 XML 处理语言是近期研究 XML 处理方法的热点领域.该种语言使用模式提供的静态类型化能力增加了 XML 数据交换和处理的鲁棒性,弥补现有 XML 处理方法只能在运行时刻检查输入/输出 XML 文档是否满足模式约束,但不能保证操作这些文档的程序始终产生满足模式约束的文档这一缺陷.当进行静态类型检查时,静态类型化 XML 处理语言把模式作为类似 Int,Float 等本地类型来对待,对整个程序进行类型检查.静态类型检查使用模式提供的类型信息确保程序不会在运行时刻发生与类型相关的错误.静态类型化 XML 处理语言的代表语言有

* Supported by the National Natural Science Foundation of China under Grant Nos.60573013, 60721061 (国家自然科学基金)
Received 2008-12-24; Revised 2009-06-09; Accepted 2009-11-26

Xduce[2−5],Cduce[6−8]和 Xtatic[9−12]等.

由于 XML 的类型与树自动机[13]对应,XML 类型检查大多通过树自动机实现,但是类型检查的复杂度是指数时间.因此,一个重要的研究问题就是如何提高实际的效率.

类型检查过程中最重要的问题是判定两个类型是否满足子类型关系.类型检查的实现很大部分依赖于子类型关系判定.子类型关系判定算法(Subtyping 算法)是类型检查的核心算法,其运行占用整个静态类型检查的大部分时间.设计一种高效的 Subtyping 算法可以大大减少类型检查的时间,加快程序的运行速度,使静态类型化 XML 处理语言更加实用.

本文研究如何提高 XML 的 Subtyping 算法的效率.在静态类型化 XML 处理语言中,许多语言都基于 XDuce 的类型系统.因此,对于 XML 的 Subtyping 算法,XDuce 具有代表性.本文以 XDuce 的 Subtyping 算法为基础,研究与 XML 相关的子类型关系定义和 Subtyping 算法,提出剪枝策略,对算法进一步优化.实验数据显示,使用剪枝优化策略后,算法的执行效率平均可以提高 20%.

本文第 1 节定义子类型关系.第 2 节描述 XDuce 的 Subtyping 算法.第 3 节介绍剪枝优化策略.第 4 节给出实验结果.第 5 节讨论其他 Subtyping 算法和优化策略.第 6 节总结全文.

1 子类型关系

通常,子类型关系使用语法方式定义(下称语法子类型),即定义一个由公理和规则构成的形式系统,通过在形式系统上进行推理得到类型之间的子类型关系.

除此之外,定义子类型关系还可以使用语义方法(下称语义子类型),即定义一个语言模型,将类型映射为该语言模型的子集.子类型关系定义为对应子集之间的包含关系.若此包含关系是可判定的,则可得到子类型关系的判定算法.

语义子类型的形式定义如下:

假设有类型 t,s 和值 v,$<:$为子类型关系,$\subseteq$为集合上的包含关系:

$$[\![t]\!]=\{v\mid\vdash v:t\},t<:s\Leftrightarrow[\![t]\!]\subseteq[\![s]\!].$$

相比较于语法子类型,语义子类型具有如下优点:

(1) 假设有类型 t 和 s.当 t 不是 s 的子类型时,语义子类型可以得到一个属于 t 但不属于 s 的反例,用于提供详细的出错信息;而语法子类型并不能得到这些详细的信息;

(2) 语法子类型关系的判定算法与其定义的形式系统密切相关,导致算法的修改和优化都很困难;而语义子类型通过选择合适的语言模型,使算法仅与语言模型相关,令算法实现和优化变得简单;

(3) 得到语法子类型的性质需要通过复杂的证明;而语义子类型的许多性质可以直接从对应的语言模型得到,如传递性、分配律等.

虽然语义子类型的以上优点非常吸引人,但是,为类型找到合适的语言模型的子集非常困难.在一些语言中,类型可以很自然地用该类型的数据值集合来表示,这时,语义子类的定义可以变得较为简单,如 XDuce,LFC 语言[2].

下文中的所有子类型关系,若无特殊指明,均指语义子类型关系.

2 子类型关系判定算法

XDuce 是静态类型化 XML 处理语言的代表,它是一种类似于 ML 的函数式语言.XDuce 专为处理 XML 设计一种简洁而强大的类型系统.该类型系统直接基于树自动机理论,其基本数据值为 XML 文档,基本数据类型称为正规表达式类型(regular expression type).

与普通有穷状态自动机接受字符串不同,树自动机[9]是接受树的有穷状态自动机.树自动机可以看作是类型,该自动机所能接受的所有树的集合成为该类型所对应的值.非确定性树自动机可以用于表示并类型(union type),带有环的迁移关系(transition)可以用于表示递归类型(recursive type).

正规表达式类型可以与树自动机有效地进行一一对应的转换[2].因此,子类型关系判定问题归结为树自动机的包含问题.此问题是可判定的,算法理论复杂度为 EXPTIME-complete[14].

判断两个树自动机包含关系理论算法的基本思想来源于以下等价关系:

假设有两个自动机 M 和 M',$L(M)$是 M 接受的语言,$\overline{L}$ 是 L 的补,

$$L(M)\subseteq L(M')\Leftrightarrow L(M)\cap\overline{L(M')}=\varnothing.$$

理论算法步骤简单、直观,仅适合在进行理论证明时使用,不适合在实际使用的语言中作为核心算法.

XDuce 提出一种更为有效的 Subtyping 算法.该算法是一种自顶向下的方法,其基本思想来源于 Aiken 和 Murphy 的算法[15]:从一对类型开始,每一步仅检查最顶层的类型,然后递归地检查类型的每个分量,直至到达叶子节点.叶子节点只需进行简单检查.

自顶向下的算法可以得到一些简单的优化,例如,在检查过程中可以使用自反性($t<:t$)避免深入检查整个类型,也可以使用缓存保存子类型关系检查中的所有中间结果以供以后使用.

完整的 Subtyping 算法描述如下:

Subtyping 算法定义为两种形式的判断:$\Gamma\vdash A<:B\Rightarrow\Gamma'$和$\Gamma\vdash^{*}A<:B\Rightarrow\Gamma'$,称为假设.其中,$\Gamma,\Gamma'$是由形如 $C<:D$ 的类型对构成的集合,A,B,C,D 为类型,$A<:B$ 表示 A 是 B 的子类型.这两种判断都读作“若输入集合Γ中所有形如 $C<:D$ 的子关系成立,则 $A<:B$ 成立,且在输出集合Γ'加入在此过程中已被证明的所有类型对 $E<:F$”.这两种判断的区别在下文中将给出解释.

实际上,同一对类型之间的子类型关系总是在多处被使用.输出集合Γ'用于保存所有已被检查的类型对,以减少重复冗余的检查.它是所有已被验证的子类型关系的缓存,可用于子类型检查和整个程序的其他编译过程.

Subtyping 算法定义如下规则:

$$\frac{A<:B\in\Gamma}{\Gamma\vdash A<:B\Rightarrow\Gamma} \tag{2.1}$$

$$\frac{A<:B\notin\Gamma\qquad \Gamma;A<:B\vdash^{*}A<:B\Rightarrow\Gamma'}{\Gamma\vdash A<:B\Rightarrow\Gamma'} \tag{2.2}$$

规则(2.1)表明,如果类型对 $A<:B$ 已在假设Γ中,则成功返回;否则,如规则(2.2)所示,将类型对加入假设Γ中.这两条规则保证不会重复验证相同的类型对.在规则(2.2)中,我们将判断从$\Gamma\vdash A<:B\Rightarrow\Gamma'$转换为$\Gamma\vdash^{*}A<:B\Rightarrow\Gamma'$,这样可以防止在规则(2.2)后立刻使用规则(2.1)所导致的不正确结果.在下面的规则中,我们继续使用判断$\Gamma\vdash^{*}A<:B\Rightarrow\Gamma'$,直到最后一条规则(2.8)再转换回$\Gamma\vdash A<:B\Rightarrow\Gamma'$.

余下的规则取决于输入的形式:

$$\frac{}{\Gamma\vdash^{*}\varnothing<:A\Rightarrow\Gamma} \tag{2.3}$$

$$\frac{size(A)<size(A\mid A')\qquad size(A')<size(A\mid A')\qquad \Gamma\vdash^{*}A<:B\Rightarrow\Gamma'\qquad \Gamma'\vdash^{*}A<:B\Rightarrow\Gamma''}{\Gamma\vdash^{*}A\mid A'<:B\Rightarrow\Gamma''} \tag{2.4}$$

$$\frac{}{\Gamma\vdash^{*}\varepsilon<:\varepsilon\mid A\Rightarrow\Gamma} \tag{2.5}$$

规则(2.3)表明,若左端为空集,则子类型关系显然成立.规则(2.4)表明,若左端能被拆分为集合 A 和 A'的并,则为每个集合生成一个子目标.因为基于集合的性质:$A\cup A'\subseteq B$ 成立当且仅当 $A\subseteq B$ 和 $A'\subseteq B$ 同时成立.条件$size(A)<size(A|A')$和$size(A')<size(A|A')$保证每次算法处理更小的集合.规则(2.5)表明,若左端仅含有叶子节点ε,即空序列,则检查右端是否同样含有叶子节点.算法仅在此规则可能出现失败.

以下规则中左端仅包含一个分枝.

$$\frac{size(C)<size(l'(B,B')\mid C)\qquad l\equiv l'\qquad \Gamma\vdash^{*}l(A,A')<:C\Rightarrow\Gamma'}{\Gamma\vdash^{*}l(A,A')<:l'(B,B')\mid C\Rightarrow\Gamma'} \tag{2.6}$$

$$\frac{size(C) < size(\varepsilon \mid C)}{\Gamma \vdash^{*} l(A,A') <: C \Rightarrow \Gamma'} \quad (2.7)$$
$$\Gamma \vdash^{*} l(A,A') <: \varepsilon \mid C \Rightarrow \Gamma'$$

$$\frac{\begin{array}{c}\text{for all } 1 \le j \le n, l \prec l_j, \\ \text{for all } 1 \le i \le 2^n, \text{ either} \\ \Gamma_{i-1} \vdash A <:|_{j \in I_i^n} B_j \Rightarrow \Gamma_i \text{ or } \Gamma_{i-1} \vdash A' <:|_{j \in \overline{I_i^n}} B'_j \Rightarrow \Gamma_i\end{array}}{\Gamma_0 \vdash^{*} l(A,A') < : l_1(B_1,B'_1) \mid ... \mid l_n(B_n,B'_n) \Rightarrow \Gamma_{2^n}} \quad (2.8)$$

规则(2.6)和规则(2.7)除去右端的叶子节点和所带标注 l'不大于 l 的分枝,规则(2.8)处理其他情况:右端是类型 $l_j(B_j,B'_j)$ 的并(可能为空),且所有的标注 l_j 都大于 l.在此规则中,以任意顺序枚举集合 1,…,n 的所有子集,记为 I_1^n 到 $I_{2^n}^n$,共 2^n 个.记 $\overline{I_i^n}$ 为 I_i^n 的补集.对于每个 i,都需"证明 A 是并$|_{j \in I_i^n} B_j$ 的子类型或 A'是并$|_{j \in \overline{I_i^n}} B'_j$ 的子类型"两者成立其一.为了解释规则(2.8),参考如下例子.假定要证明 $l(T,U)<:l(R_1,S_1)|l(R_2,S_2)|l(R_3,S_3)$,一种方法是判断左端与右端子项之一满足子类型关系.但是这种方法太弱,例如,如果 $T=R_1|R_2,U=S_1=S_2=S$,则无法做出判断.在 XDuce 中给出了如下解决方法.首先右端可以重写为

$$(l(R_1,T) \cap l(T,S_1)) | (l(R_2,T) \cap l(T,S_2)) | (l(R_3,T) \cap l(T,S_3)),$$

其中,T 表示所有树的集合.利用交对并的分配律,可得

$$(l(R_1,T) | l(R_2,T) | l(R_3,T)) \cap$$
$$(l(T,S_1) | l(R_2,T) | l(R_3,T)) \cap$$
$$(l(R_1,T) | l(T,S_2) | l(R_3,T)) \cap$$
$$\cdots$$

由此,原来的子类型关系归约为对每个 I,

$$l(T,U) <: (|_{i \in I}\ l(R_i,T)) \mid (|_{i \in \overline{I}}\ l(T,S_i)) \text{ 或 } l(T,U) <: (l(|_{i \in I}\ R_i,T)) \mid (l(T,|_{i \in \overline{I}}\ S_i)),$$

其中,I 是{1,2,3}的子集,$\overline{I}$ 是{1,2,3}\$I$.这等价于 $T<:|_{i \in I} R_i$ 或 $U <:|_{i \in \overline{I}}\ S_i$.

3 剪枝优化策略

在其他静态类型化 XML 处理语言中,许多语言都将 XDuce 的类型系统作为其本身类型系统的一部分.

由于本文只考虑与 XML 相关的 Subtyping 算法,因此 XDuce 是一个理想的实验平台.使用 XDuce,一方面降低了类型系统的复杂度,不需要考虑与 XML 无关的其他类型;另一方面可以使本文的结论适用于所有使用 XDuce 的 Subtyping 算法的其他静态类型化 XML 处理语言.

XDuce 的 Subtyping 算法运行效率可进一步提高.XDuce 已有实现中介绍一些优化策略,其基本思想是,针对实际应用中的一些特殊情况,使用简化规则代替规则(2.8),尽量避免 Subtyping 算法使用此规则,称为高层实现技术(high-level implementation techniques)[2].通过以上高层实现技术,可以在一些情况下减少规则(2.8)的调用次数.

但我们在实验中发现,在许多情况下,规则(2.8)无法避免被调用.针对这种情况,我们提出一种更复杂的优化策略,通过减少规则(2.8)中递归调用的次数,进一步缩短此规则的运行时间,使 Subtyping 算法更高效.

3.1 基本思想

仔细观察 XDuce 的 Subtyping 算法,其复杂度主要来自于对并类型的处理,即规则(2.8).在规则(2.8)中,需要枚举子类型关系判断式右端 n 个分枝的 2^n 种组合,并对子目标进行 2×2^n 次递归调用,因此,理论复杂度为 $O(2^n)$,n 为右端并表达式的分枝数.此规则消耗了整个 Subtyping 算法的大部分计算时间.

图 1 显示证明 $l(\varepsilon,X') <: l_1(\varepsilon,Y'_1) \mid l_2(Y_2,Y'_2)$ 关系成立的递归调用树.该递归调用树共有 $2^2=4$ 个分枝,进行 $2\times2^2=8$ 次递归调用对子目标进行类型检查.

如果对于一些显然成立的分枝直接做出判断,并使用类似于布尔表达式中短路计算的方法,可以明显地减少需要递归调用的次数,其效果类似于剪枝.

同样,对于图 1 中的例子,图 2 显示使用剪枝优化策略后的递归调用树.叉表示对或运算所做的剪枝,圆圈表示 $X'<:Y_1'|Y_2'$ 不成立时对与运算所做的剪枝.第 1 次减枝(如叉所示)减少 3 次递归调用.若 $X'<:Y_1'|Y_2'$ 不成立,则第 2 次减枝(如圆圈所示)可减少 1 次递归调用.

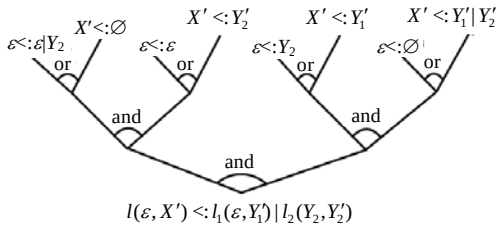

Fig.1 Tree of recursive calls

图 1 递归调用树

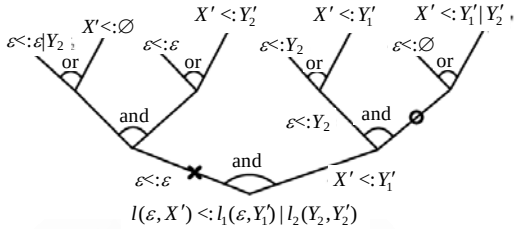

Fig.2 Tree of recursive calls after pruning

图 2 减枝后的递归调用树

3.2 实现算法描述

使用剪枝优化策略后,规则(2.8)的算法实现如下:

算法 1. 函数 $rec(A,S_1,A',S_2,k)$.

输入:l,l_i,B_i,B_i',A 和 A'对应于规则(2.8)中的同名符号;S_1 和 S_2 保存构造集合 $|_{j\in I_i^n} B_j$ 和集合 $|_{j\in \overline{I_i^n}} B_j'$ 过程的中间结果,初始值为空;k 为集合构造过程中,本次需处理的右端并表达式的分量下标,初始值为 1.其中,l,l_i,B_i,B_i' 是外部变量.

输出:若子类型关系 $l(A,A')<:l_1(B_1,B_1')|...|l_n(B_n,B_n')$ 成立,则返回 TRUE;否则,返回 FALSE.

过程:

(1) 若 $1\leqslant k\leqslant n$ 不成立,则转至步骤(7);否则,

(2) 检查剪枝条件 $A<:B_k$ 和递归调用 $rec(A,S_1\cup B_k,A',S_2,k+1)$的返回值;

(3) 若两个返回值都是 FALSE,则返回 FALSE,算法终止;否则,

(4) 检查剪枝条件 $A'<:B_k'$ 和递归调用 $rec(A,S_1,A',S_2\cup B_k',k+1)$ 的返回值;

(5) 若两个返回值都是 FALSE,则返回 FALSE,算法终止;否则,

(6) 返回 TRUE,算法终止.

(7) 若 $A<:S_1$ 成立,则返回 TRUE,算法终止;否则,

(8) 若 $A'<:S_2$ 成立,则返回 TRUE,算法终止;否则,

(9) 返回 FALSE,算法终止.

算法 1 可分为两部分:步骤(1)~步骤(6)为集合 $|_{j\in I_i^n} B_j$ 和集合 $|_{j\in \overline{I_i^n}} B_j'$ 的构造过程,采用递归的方法;步骤(7)~步骤(9)为子目标的子类型关系判定的过程,递归调用 Subtyping 算法.步骤(2)和步骤(4)中对剪枝条件和递归调用返回值的判断,分别对应或运算和与运算的剪枝.算法 1 的理论复杂度为 $O(2^n)$,n 为右端并表达式的分枝数.

3.3 算法正确性证明

求证:函数 $rec(A,S_1,A',S_2,k)$返回 TRUE 当且仅当以下表达式成立:

$$\{A<:S_1\cup B_k\cup|_{j\in I_{i,k+1}^n} B_j \text{ or } A'<:S_2\cup|_{j\in \overline{I_{i,k+1}^n}} B_j'\} \text{ and } \{A<:S_1\cup|_{j\in I_{i,k+1}^n} B_j \text{ or } A'<:S_2\cup B_k'\cup|_{j\in \overline{I_{i,k+1}^n}} B_j'\},$$

其中,$S_1=|_{j\in I_i^{n,k-1}} B_j, S_2=|_{j\in \overline{I_i^{n,k-1}}} B_j'$,$j\in I_i^{n,k-1}$ 表示 $j\in I_i^n$ 的从 1 到 k–1 段,$j\in \overline{I_i^{n,k-1}}$ 表示 $j\in \overline{I_i^n}$ 的从 1 到 k–1 段,$j\in I_{i,k+1}^n$ 表示 $j\in I_i^n$ 的从 k+1 到 n 段,$j\in \overline{I_{i,k+1}^n}$ 表示 $j\in \overline{I_i^n}$ 的从 k+1 到 n 段.

证明:现证明,对于任何整数 $k\geqslant 1$,函数 $rec(A,S_1,A',S_2,k)$返回 TRUE 时,以下表达式成立:

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}.$$

施归纳于 k.

第 1 部分,当 $1\leqslant k\leqslant n$ 时,执行算法 1 的步骤(2).

(2.1) 若剪枝条件 $A<:B_k$ 成立,则显然 $A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j$ 成立.因此表达式:

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$$

成立.执行算法 1 的步骤(4);否则,

(2.2) 若递归调用 $rec(A,S_1\cup B_k,A',S_2,k+1)$返回 TRUE,根据递归假设,表达式:

$$\{A <: S_1 \cup B_k \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\}$$

成立.根据 $S_i(i=1,2)$和$|_{j\in \overline{I^n_{i,k}}} B'_j$ 的定义,我们可以重新组合表达式:

$$\{A <: S_1 \cup B_k \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} =$$

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}.$$

因此,表达式 $\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$ 成立.执行算法 1 的步骤(4);否则,

(3) 函数 $rec(A,S_1\cup B_k,A',S_2,k+1)$返回 FALSE,根据递归假设,表达式:

$$\{A <: S_1 \cup B_k \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\}$$

不成立.根据 $S_i(i=1,2)$和$|_{j\in \overline{I^n_{i,k}}} B'_j$ 的定义,我们可以重新组合表达式:

$$\{A <: S_1 \cup B_k \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} =$$

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}.$$

因此,表达式 $\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$ 不成立,因此表达式:

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$$

不成立.

(4.1) 若剪枝条件 $A' <: B'_k$ 成立,则显然 $A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j$ 成立.因此表达式:

$$\{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$$

成立.执行算法 1 的步骤(6);否则,

(4.2) 若递归调用 $rec(A,S_1,A',S_2\cup B'_k,k+1)$ 返回 TRUE,则根据递归假设,表达式:

$$\{A <: S_1 \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\}$$

成立.根据 $S_i(i=1,2)$和$|_{j\in \overline{I^n_{i,k}}} B'_j$ 的定义,我们可以重新组合表达式:

$$\{A <: S_1 \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} =$$

$$\{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}.$$

因此,表达式 $\{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$ 成立.执行算法 1 的步骤(6);否则,

(5) 函数 $rec(A,S_1,A',S_2\cup B'_k,k+1)$ 返回 FALSE,根据递归假设,表达式:

$$\{A <: S_1 \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\}$$

不成立.根据 $S_i(i=1,2)$和$|_{j\in \overline{I^n_{i,k}}} B'_j$ 的定义,我们可以重新组合表达式:

$\{A <: S_1 \cup B_{k+1} \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\}$ and $\{A <: S_1 \cup |_{j\in I^n_{i,k+2}} B_j \text{ or } A' <: S_2 \cup B'_k \cup B'_{k+1} \cup |_{j\in \overline{I^n_{i,k+2}}} B'_j\} =$ $\{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$.

因此,表达式 $\{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$ 不成立,所以表达式:

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$$

不成立.

(6) 函数 $rec(A,S_1,A',S_2,k)$返回 TRUE.根据上面的证明,此时表达式:

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\} \text{ 与 } \{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$$

同时成立.因此,表达式

$$\{A <: S_1 \cup B_k \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\} \text{ and } \{A <: S_1 \cup |_{j\in I^n_{i,k+1}} B_j \text{ or } A' <: S_2 \cup B'_k \cup |_{j\in \overline{I^n_{i,k+1}}} B'_j\}$$

成立.

第 2 部分,当 $k>n$ 时,执行算法 1 的步骤(7).此时,表达式简化为 $A<:S_1$ or $A<:S_2$.

(7) 若 $A<:S_1$ 成立,则函数 $rec(A,S_1,A',S_2,k)$返回 TRUE.同时,显然,表达式 $A<:S_1$ or $A<:S_2$ 成立;否则,

(8) 若 $A<:S_2$ 成立,则函数 $rec(A,S_1,A',S_2,k)$返回 TRUE.同时,显然,表达式 $A<:S_1$ or $A<:S_2$ 成立;否则,

(9) 函数 $rec(A,S_1,A',S_2,k)$返回 FALSE,同时,表达式 $A<:S_1$ 和 $A<:S_2$ 都不成立.因此,表达式 $A<:S_1$ or $A<:S_2$ 不成立.

反之亦然.证毕. □

当 $k=1,\ldots,n$ 时,分别对应于规则(2.8)的 2^n 组假设成立,因此规则(2.8)的结论成立.

由以上证明可知,加入剪枝优化策略后的算法 1 没有改变规则(2.8)中假设与结论的关系,算法 1 得到的结论与规则(2.8)得到的结论一致.因此,算法 1 是正确的.

3.4 进一步的讨论

为进行剪枝,需要设定剪枝的条件.当此条件满足时,就进行剪枝,终止此次调用;若不满足,则继续进行计算.设定剪枝条件可有两种选择:

(1) 仅对一些简单条件进行判断.简单条件可定义为不需要耗费大量运算时间的子类型关系判定,如使用高层实现技术中所提出的简化规则;

(2) 对所有情况都进行子类型关系判定,其中可能递归调用 Subtyping 算法,耗费大量的时间.但这里所做的所有子类型关系判定最终都需要进行计算,此策略只是将这些计算提前到合适的地方.

通过对图 1 和图 2 中的递归推导树的观察可以发现,使用递归构造集合 $|_{j\in I^n_i} B_j$ 和 $|_{j\in \overline{I^n_i}} B'_j$ 导致原有的 2^n 个子目标构成二叉树状的层次结构.虽然二叉树每一枝上剪枝条件的不同排列顺序构成不同的剪枝结果,但是最终减少的递归调用次数是相同的.因此,子目标的层次结构或剪枝条件的排列顺序并不影响剪枝优化策略的效率,无须对它们作特别的优化.

4 实验结果

为验证剪枝优化策略的有效性,我们修改 XDuce 0.5.0[16]的源代码,实现第 3 节描述的剪枝优化策略,运行若干测试用例,收集并比较使用剪枝优化策略前后程序的运行数据.

我们所关注的测试数据有:在使用剪枝优化策略后,

(1) 满足剪枝条件的表达式或模式个数;

(2) 子类型关系判定函数调用减少的次数;

(3) 静态类型检查减少的时间.

考虑到静态类型化 XML 处理语言在现实中的应用,我们针对性地选择了几个不同场景,包括对简单 XML

实例的类型检测和转换,对不同 XML 类型定义之间的转换和对常用的复杂 HTML 文件的转换.这些测试用例覆盖了现实应用中使用 XML 的主要场景,包括 XML 文件的类型检测、片段抽取和类型转换.

选择的测试用例主要来自 XDuce 提供的例子,其中,对于较复杂的测试用例 html2latex 同时还使用 CDuce 提供的对应例子.addrbook 从地址簿中将有电话者的名字和电话号码提取出来组成电话簿.bookmark 从 Netscape 格式的书签文件中提取 Public 文件夹,构建一张内容目录,并建立目录与其实体的链接.toUpper 将所有食谱条目 title 标签中的内容改为大写.ns2xbel 将 Netscape 格式的书签文件转换为 XBEL 格式的书签文件.rng2xduce 将 Relax NG 格式的文件转换为 XDuce 格式的文件.html2latex1 来源于 CDuce 的测试用例.它通过解释标签命令将 HTML 文件转换为 LATEX 文件.html2latex2 是 XDuce 自带的测试用例,其功能与 html2latex1 相同.polysample 用于测试剪枝优化策略在含有多态的程序中的有效性.

前 3 个测试用例简单地使用递归函数对整棵 XML 树进行遍历.第 4 和第 5 个例子与前 3 个类似,但是 XML 树的规模和复杂度更大.第 6 和第 7 个例子中使用 HTML 作为类型.HTML 类型是 Web 文档中最庞大的类型,因此可以作为测试剪枝优化策略是否有效的理想例子.最后一个例子用于测试剪枝优化策略在 XDuce 的新特性——多态下是否有效.

本测试主要基于 XDuce 0.5.0,默认打开 patopt.基本运行平台为 Ubuntu5.10,Intel Celeron CPU 2.0GHz, 726MB RAM.使用 XDuce Counter 进行计数.使用 XDuce Timer 进行计时,时间单位为 ms.

表 1 列出所有测试用例的结果.Pruned 列显示满足剪枝条件的表达式或模式个数.Subgoals 列显示使用剪枝优化策略前后调用子类型关系判定函数次数减少的百分比.Type check 列显示使用剪枝优化策略前后静态类型检查花费时间减少的百分比.

Table 1 Results of running XDuce test cases

表 1 XDuce 测试用例运行结果

Examples	Pruned	Subgoals			Type check (ms)		
		Before	After	Perc (%)	Before	After	Perc (%)
addrbook	276	1 925	1 091	43.3	6.2	4.7	24.2
bookmarks	361	7 263	6 175	15.0	261.9	259.0	1.1
toUpper	264	1 776	950	46.5	5.6	4.5	19.6
ns2xbel	283	2 104	1 277	39.3	10.3	8.4	18.4
rng2xduce	6 835	33 402	24 317	27.2	268.3	147.5	45.0
html2latex1	1 865	106 243	94 298	11.2	1 119.1	981.0	12.3
html2latex2	45 932	764 429	280 813	63.3	11 173.1	7 275.1	34.9
polysample	129	1 325	789	40.5	9.9	8.7	12.1

实验数据显示,使用剪枝优化策略前、后,XDuce 的执行效率平均可以提高 20%.子类型关系判定函数调用次数平均可以减少 35%.静态类型检查时间的减少除了与子类型关系判定函数的调用次数有关以外,还与子类型关系判定函数本身在 XML 树中所处位置相关.同样,减少一次子类型关系判定函数调用,接近 XML 树根部的子类型关系判定函数调用可以节约更多的静态类型检查时间.因此,Subgoals 的减少比例与 Type check 并不一定成正比关系.这与实际测试用例的设计有关.

5 讨 论

5.1 其他Subtyping算法

许多静态类型化 XML 处理语言,如 Cduce,Xtatic,都使用与 XDuce 类似的 Subtyping 算法.除此以外,XHaskell 使用与 XDuce 不同的 Subtyping 算法[17].

XHaskell 是 Haskell[18]语言的一种扩展.它将 XDuce 的正规表达式类型(regular expression type)、正规表达式模式匹配(regular expression pattern matching)和语义子类型(semantic subtyping)加入到 Haskell 语言中.在 XDuce 和 CDuce 的运行时刻,基本数据值失去原有的结构,被编码成另一种结构.XHaskell 可以保留基本数据值原有的结构,并使用代数证明方法——Antimirov 算法[19]判定正规表达式之间的包含关系,但是尚无 XHaskell 与 XDuce 等基于树自动机的 Subtypign 算法的效率比较.

5.2 其他优化策略

CDuce 提出其他优化 Subtyping 算法的策略[7].与 XDuce 稍有不同,CDuce 的 Subtyping 算法涉及基本类型、乘积类型和函数类型.

较简单的策略保存子类型判定不成功的类型对.在 Subtyping 算法执行之前,若查到该类型对已在以前判定中显示不成立,则直接返回结果;否则,继续执行 Subtyping 算法.

更复杂的策略称为 Split and Distribute 技术.与第 3 节剪枝优化策略的目的相同,此策略改进了规则(2.8).不同之处在于,此策略基于以下集合性质:

假设存在类型$(t,s)\backslash(t_1,s_1)\backslash\ldots\backslash(t_n,s_n)$,$t_i(i=1,\ldots,n)$两两不相交,则

$$(t,s)\backslash(t_1,s_1)\backslash\ldots\backslash(t_n,s_n)=(t\backslash t_1\backslash\ldots\backslash t_n,s)|(t\&t_1,s\backslash s_1)|\ldots|(t\&t_n,s|s_n) \tag{5.1}$$

其中,\为集合差运算,&为集合交运算,|为集合并运算.等式(5.1)右端通过删除空项可以进一步加以简化.

通过分裂(split)原式中不满足两两不相交条件的类型 t_i,总是可以找到满足上述条件$(t,s)\backslash(t_1,s_1)\backslash\ldots\backslash(t_n,s_n)$的类型,从而套用等式(5.1).

此优化策略的困难在于如何分裂原式中不满足两两不相交条件的类型 t_i,具体分裂方法在文献[5]中并没有阐述,需参考 CDuce 源程序[20].注意,分裂后类型数目可能呈指数次增长.

根据等式(5.1),规则(2.8)可变为

$$\frac{\begin{array}{c}B_1,...,B_n\text{不相交, 且}\\ \Gamma_0\vdash A<:B_1|...|B_n\Rightarrow\Gamma_1\\ \text{and for all } 1\leqslant i\leqslant n,\ \text{either}\\ A\cap B_i=\varnothing\ \text{ or }\Gamma_i\vdash A'<:B_i'\Rightarrow\Gamma_{i+1}\end{array}}{\Gamma_0\vdash^{*} l(A,A')<:l_1(B_1,B_1')|...|l_n(B_n,B_n')\Rightarrow\Gamma_{n+1}} \tag{5.2}$$

相比较规则(2.8),规则(5.2)避免回溯.在规则(5.2)中,首先进行 1 次递归调用计算 $A<:B_1|\ldots|B_n$ 是否成立,然后遍历子类型关系判断式右端 n 个分枝,并对每个分枝进行 1 次集合交运算和 1 次递归调用计算判断 $A'<:B_i'$ 是否成立.因此,理论复杂度为 $O(n)$,n 为右端并表达式的分枝数.

假设ε与 Y_2 不相交,图 3 显示使用规则(5.2)证明 $l(\varepsilon,X')<:l_1(\varepsilon,Y_1')|l_2(Y_2,Y_2')$ 关系成立的递归调用树.该递归调用树共有 1+2=3 个分枝,进行 2 次集合交运算和 1+2=3 次递归调用对子目标进行类型检查.

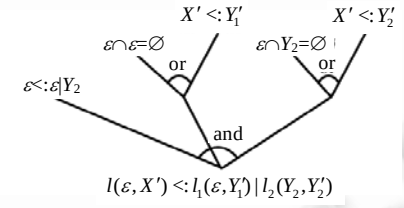

Fig.3 Tree of recursive calls after using CDuce optimization strategies

图 3 使用 CDuce 优化策略后的递归调用树

规则(5.2)可以使用第 3 节的剪枝优化策略.剪枝条件设定为"若 $A\cap B_i=\varnothing$,则剪枝;否则,判定 $A'<:B_i'=\varnothing$ 是否成立".CDuce 的 Split and Distribute 技术利用集合性质避免回溯,降低了 Subtyping 算法的复杂度,是优化的有效策略.

6 结 论

静态类型化 XML 处理语言使用模式提供的静态类型化能力,使 XML 成为和 Int,Float 相同的本地类型,为程序开发人员提供更丰富的数据结构,增加了 XML 数据交换和处理的鲁棒性.但是,现有的大多数静态类型化 XML 处理语言效率较低.

本文尝试通过剪枝优化策略提高静态类型化 XML 处理语言类型系统中的核心算法——子类型关系判定算法的效率.实验数据显示,使用剪枝优化策略前、后,XDuce 的执行效率平均可以提高 20%.此策略具有一定的普遍性,对所有使用 XDuce 的 Subtyping 算法的语言都有效.

References:

[1] Chen HM, Dong YM. A formal specification language supporting specification acquisition. Chinese Journal of Computers, 2002, 25(5):459−466 (in Chinese with English abstract).

[2] Hosoya H. Regular expression types for XML [Ph.D. Thesis]. Tokyo: The University of Tokyo, 2000.

[3] Hosoya H, Pierce BC. Regular expression pattern matching for XML. Journal of Functional Programming, 2002,13(6):961−1004. [doi: 10.1017/S0956796802004410]

[4] Hosoya H, Pierce BC. XDuce: A statically typed XML processing language. ACM Trans. on Internet Technology, 2003,3(2): 117−148. [doi: 10.1145/767193.767195]

[5] Hosoya H, Vouillon J, Pierce BC. Regular expression types for XML. ACM Trans. on Programming Languages and Systems, 2005, 27(1):46−90. [doi: 10.1145/1053468.1053470]

[6] Castagna G, Frisch A. A gentle introduction to semantic subtyping. In: Proc. of the 7th ACM SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming. New York: ACM Press, 2005. 198−199. http://centria.di.fct.unl.pt/conferences/ppdp05/

[7] Benzaken V, Castagna G, Frisch A. CDuce: An XML-centric general-purpose language. In: Proc. of the 8th ACM SIGPLAN Int'l Conf. on Functional Programming. New York: ACM Press, 2003. 51−63. http://www.cc.gatech.edu/icfp03/

[8] Frisch A, Castagna G, Benzaken V. Semantic subtyping. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science. Washington: IEEE Computer Society, 2002. 137−146. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8005

[9] Gapeyev V, Garillot F, Pierce BC. Statically typed document transformation: An Xtatic experience. In: Castagna G, Raghavachari M, eds. Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X). Aarhus: BRICS, 2006. 2−13.

[10] Gapeyev V, Levin M, Pierce BC, Schmitt A. XML goes native: Run-Time representations for Xtatic. In: Bodik R, ed. Proc. of the 14th Int'l Conf. on Compiler Construction. Berlin: Springer-Verlag, 2005. 43−58.

[11] Gapeyev V, Levin M, Pierce BC, Schmitt A. The Xtatic experience. In: Proc. of the Workshop on Programming Language Technologies for XML (PLAN-X). 2005. 16−31. http://www.research.att.com/conf/planx2005/

[12] Gapeyev V, Pierce BC. Regular object types. In: Cardelli L, ed. Proc. of the European Conf. on Object-Oriented Programming (ECOOP). Berlin: Springer-Verlag, 2003. 151−175.

[13] Comon H, Dauchet M, Gilleron R, Jacquemard F, Lugiez D, Tison S, Tommasi M. Tree automata techniques and applications. 2002. http://www.grappa.univ-lille3.fr/tata

[14] Seid H. Deciding equivalence of finite tree automata. SIAM Journal, 1990,19(3):424−437. [doi: 10.1137/0219027]

[15] Aiken A, Murphy BR. Implementing regular tree expressions. In: Hughes J, ed. Proc. of the 5th ACM Conf. on Functional Programming Languages and Computer Architecture. London: Springer-Verlag, 1991. 427−447.

[16] XDuce: A typed XML processing language. 2003. http://xduce.sourceforge.net/

[17] Lu K, Sulzmann M. An implementation of subtyping among regular expression types. In: Chin WN, ed. Proc. of the APLAS 2004. Berlin: Springer-Verlag, 2004. 57−73.

[18] Haskell. http://www.haskell.org/

[19] Antimirov V. Rewriting regular inequalities (extended abstract). In: Reichel H, ed. Proc. of the 10th Int'l Symp. on Fundamentals of Computation Theory. London: Springer-Verlag, 1995. 116−125.

[20] CDuce. http://www.cduce.org/

附中文参考文献:

[1] 陈海明,董韫美.一个支持规约获取的形式规约语言.计算机学报,2002,25(5):459−466.

戴晓君(1982－),男,江苏江阴人,硕士,主要研究领域为形式语言,形式规约.

陈海明(1966－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为软件设计,计算模型,程序语言.