基于上下文无关文法的可逆变换模型

吴阳怿[1,2], 吴逸鸣[1,2], 熊英飞[1,2+]

[1](北京大学 信息科学技术学院软件研究所,北京 100871)

[2](北京大学 高可信软件技术教育部重点实验室,北京 100871)

[+]通讯作者，邮件：xiongyf@pku.edu.cn

摘 要: 可逆变换和双向变换等数据转换问题一直是近年来的研究热点,研究人员针对该问题提出了大量相关的语言和模型.通过使用这些语言,程序员可以只写一个程序而产生正向和逆向两个变换,不仅提高了程序的健壮性,同时也降低了编写程序的工作量.但是,这些实现往往是建立在一种新的计算模型之上,从而导致需要花费较大的学习成本去了解其计算模型.另一方面,上下文无关文法被广泛使用于编译技术,且有着等价于下推自动机的可计算性.作为语法解析的基本工具,上下文无关文法对于绝大多数程序员来说都是不陌生的.本文提出了一种基于上下文无关文法的计算模型,用来构造字符串上的可逆变换,并对其性质和表达能力进行了探讨.

关键词: 可逆变换;上下文无关文法

1 引言

数据变换是计算机应用的一个常见内容,很多软件的开发都涉及到了数据变换.比如汇编程序把汇编代码变换成机器码,压缩算法把原始数据转换成压缩后的数据,加密算法把明文数据变成密文,程序维护时把 ini 配置文件转成 XML 配置文件等等.在很多场景下,我们不仅需要一个单向的数据变换,也需要该变换的逆变换.换句话说,如果我们把数据变换看成是从输入映射到输出的一个函数 f,我们同时也需要这个函数的反函数 f^{-1}.比如,给定一个汇编程序,我们同时会需要一个反汇编,他们互为逆变换.同样的,我们也可能需要压缩的逆变换解压缩,加密的逆变换解密程序,以及把 XML 配置文件转换回 ini 配置文件的工具.

对于这类互逆的变换程序,现有的主流开发方法需要写两个程序,一个做正向变换,另一个做逆向变换.互为反函数的一对变换往往含有大量的重复部分,写两个变换不仅存在很多重复劳动,还会增加出现低级错误的可能性,从而导致更多的调试时间和精力的浪费.

为了解决这个问题,近年来研究人员提出了采用领域特定的语言来构造一对相关的数据变换,其中代表性的工作包括可逆语言(Reversible Languages)[1]和能力更强一点的双向语言(Bidirectional Language)[1][2].通过使用这些语言,程序员只需写一段程序,就可以同时得到正向和逆向两个变换.一方面避免了重复劳动,另一方面保证了所生成的两个变换互逆,避免了出现低级错误的可能.

为了实现这个目标,现有的可逆语言和双向语言往往需要提出新的计算模型.比如,Foster 等人设计的双向变换语言 Boomerang[3]提出了一套基于组合子的计算模型,Glück 等人设计的可逆语言[1]采用了基于可逆图灵机的计算模型.由于这些计算模型和传统模型往往差别较大,程序员使用这些新型语言需要较大的学习成本.虽然可逆语言和双向语言有良好的性质,但目前在实践中的运用还比较少.

在本文中我们提出一种使用上下文无关文法来构造字符串上的可逆变换的方法,我们的方法主要基于如下观察:很多变换的核心内容是数据格式的变换,比如,在配置文件的转换中,是在 XML 的保存格式和 ini 的保存格式之间转换,而数据的内容没有变化.在现有技术中,适合描述数据格式的工具就是上下文无关文法.上下文无关文法是绝大部分程序员所熟知的一种表示方式,采用上下文无关文法来描述可逆变换可以使得学习成本大大降低.

本文的主要贡献如下:

---

[1] 双向变换和可逆变换的主要区别是，双向变换不要求转换的两个数据域包含的信息是对等的，一个数据域可以包含不和另一侧共享的信息。但可逆变换要求两个数据域包含的信息是严格对等的。

- 本文提出了一种新的计算模型来描述字符串上的可逆变换.使用该计算模型,程序员定义一个抽象数据结构并且使用上下文无关文法描述该数据结构和具体的字符串格式之间的对应关系.然后我们的方法可以自动从该描述中导出一对互逆的 parser 和 printer,完成具体字符串和该抽象结构的表示.通过描述抽象结构和两个具体格式的对应关系,我们就可以实现两个不同格式之间的可逆变换.
- 我们将我们的计算模型实现为一个基于 Scheme 的领域特定语言,并且采用该语言完成了大型实例研究:我们用我们的语言实现了 MIPS 指令集上汇编和反汇编.我们的语言能够完整的覆盖 MIPS 指令集从汇编格式到机器格式之间的对应关系,并且以较精简的代码就能完成该实例变换.该实例研究说明,我们的语言已经具有较好的表达能力,可以描述比较大型的可逆变换程序.

本文的剩余部分按如下方式组织:首先,我们用一个例子来展现我们方法的主要内容;其次,我们给出计算模型的精确定义,并证明我们的方法达到了生成可逆计算这一目标;再次,我们描述我们模型的实现和在 MIPS 汇编指令上所进行的验证;最后,我们讨论相关工作并且给出论文结论.

2 方法概览

首先我们抽象地介绍一下我们的总体思路.如图 1(a)所示,给定两个不同的数据域 A,B,我们的目标是要产生一对可逆变换 $f:A \to B$ 和 $g:B \to A$,使得它们互为反函数,即 $f = g^{-1}$.由于 A 和 B 之间可以互相转换,所以它们之间所包含的信息量是相同的.我们可以不考虑具体的形式,把这些信息抽象出来,形成一个新的数据域 O,如图 1(b)所示.由于 O 是一个抽象的逻辑结构,所以 O 到具体形式 A 之间的变换就可以看成是一个解析函数 *a.parse* 和一个输出函数 *a.print*.同理,O 到 B 之间的变换可以看做是 *b.parse* 和 *b.print*.这样,我们就把原来的函数 f 分解成了两个函数 $b.print \circ a.parse$.同样,我们有 $g = a.print \circ b.parse$.如果我们能保证 $a.parse = a.print^{-1}$ 且 $b.parse = b.print^{-1}$,我们就能得到 $f = g^{-1}$.这样,我们就把两个具体格式之间的变换分解成了一个逻辑结构和两个具体格式之间的解析和输出.如果我们能够定义出抽象结构 O 和其到 A,B 的解析和输出函数,我们就能够定义出 f 和 g.

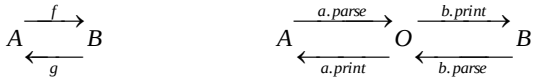

图 1 (a)可逆变换 (b)基于抽象结构的可逆变换

为了定义出 O 和解析及输出函数,我们考虑上下文无关文法.如果我们通过上下文无关文法定义出一个语法分析器,并且用该语法分析器解析得到一棵具体语法树.通过去掉该语法树中的常量部分我们可以得到一棵抽象语法树,而该抽象语法树实际上就表示了该文本的抽象逻辑结构.同时,我们如果对具体语法树做一次先序遍历并输出每个叶节点的值,我们实际就能得到原始的文本.这样,我们就通过一个上下文无关文法定义出了抽象逻辑结构和相关的解析和输出函数.在我们的方法中,为了表示的方便,我们采取让用户定义抽象结构和输出函数的方式,然后从中提取出上下文无关文法来生成语法分析器.

下面我们用一个具体的例子来说明我们的方法.我们考虑前缀表达式与后缀表达式互相转换的问题,例如,对于中缀表达式((9 * 5) + 2),其在前后缀形式之间的转换如下所示:

```
+ * 9 5 2 ⇔ 9 5 * 2 +
```

自然地,它们仅仅是不同的表现形式而已,在语义上并无区别.我们可以抽象出一个保存着整个表达式的结构(称作 *Expr*),但不再有前后缀之分了,因为这个结构只包含其逻辑形式.于是我们可以定义“表达式”这一核心的逻辑结构:一个表达式要么是一个单独的数字(*number*),要么包含一个运算符(*oper*)和两个子表达式:

$$\textbf{\textit{oper}} = [+*]$$

$$\textbf{\textit{number}} = [0-9]+$$

$$\textbf{\textit{Expr}} \to \{\textbf{\textit{oper}}\ op,\ \textbf{\textit{Expr}}\ e_1,\ \textbf{\textit{Expr}}\ e_2\} \mid \{\textbf{\textit{number}}\ n\}$$

在这个定义中,*Expr*,*number* 和 *oper* 表示类型,其中 *Expr* 的首字母大写,表示一个非终结的结构类型,否则表示终结的正则表达式类型.在我们的具体例子中,上述表达式的逻辑结构就是

$$Expr \to \left\{ \begin{array}{l} oper"+" \\ Expr \to \left\{ \begin{array}{l} oper"*" \\ Expr \to \{number"9"\} \\ Expr \to \{number"5"\} \end{array} \right\} \\ Expr \to \{number"2"\} \end{array} \right\}$$

定义出逻辑结构之后,我们可以定义具体的解析和输出函数.如前所述,我们采用定义输出函数的方式,然后从中提取上下文无关文法来产生解析函数.下面的公式给出了从逻辑结构到前缀表达式上的输出函数

$$pre(\boldsymbol{Expr}) = \{op, \text{" "}, e_1, \text{" "}, e_2\} \mid \{n\}$$

该公式对 *Expr* 的两种形式分别定义输出.对于第一种形式,我们先输出其运算符 *op*,然后输出一个空格,再递归输出其第一个子表达式 e_1,再输出一个空格,最后递归输出其第二个子表达式 e_2.对于第二种形式,我们直接输出对应数字 *n*.当对应变量是终结符的时候,我们直接输出对应的值,否则我们就递归调用相应的函数输出.换言之,我们可以认为上面的公式定义了如下函数:

$$pre.print(\boldsymbol{Expr}\{\boldsymbol{oper}\ op, \boldsymbol{Expr}\ e_1, \boldsymbol{Expr}\ e_2\}) = op + \text{" "} + pre.print(e_1) + \text{" "} + post.print(e_2)$$

$$pre.print(\boldsymbol{Expr}\{\boldsymbol{number}\ n\}) = n$$

类似的,我们也可以定义从逻辑结构到后缀表达式的输出函数:

$$post(\boldsymbol{Expr}) = \{e_1, \text{" "}, e_2, \text{" "}, op\} \mid \{n\}$$

最后,我们从输出函数中提取上下文无关文法来构造解析函数.这个过程只需要对应的把输出函数公式的右边部分换成上下文无关文法的推出部分即可.例如,对于 *pre* 的解析函数,我们构造如下文法:

$$\begin{aligned} Expr_{pre} ::= &\ oper\ \text{" "}\ Expr_{pre}\ \text{" "}\ Expr_{pre} \\ \mid &\ number \end{aligned}$$

如果提取出的文法符合 LR 等可以自动推导出语法分析器的文法类别,我们就可以用相应的算法构造语法分析器.其中,对于该语法分析器的一些终端结点(本例中为 *oper* 和 *number*),我们用之前所述的正则表达式类型来对其进行定义,这些定义用于在语法分析器试图获取一个终端结点时,从文本中解析出其具体内容.我们将文本解析成具体语法树,再将语法树对应到抽象逻辑结构,我们就得到了一个解析函数.比如,用上面的文法,我们可以将"+ * 9 5 2"解析出来,得到这样的一个结构

$$Expr_{pre}\text{"+ * 9 5 2"} \to \left\{ \begin{array}{l} oper"+" \\ \text{" "} \\ Expr_{pre}\text{"+ * 9 5"} \to \left\{ \begin{array}{l} oper"*" \\ \text{" "} \\ Expr_{pre}\text{"9"} \to \{number"9"\} \\ \text{" "} \\ Expr_{pre}\text{"5"} \to \{number"5"\} \end{array} \right\} \\ \text{" "} \\ Expr_{pre}\text{"2"} \to \{number"2"\} \end{array} \right\}$$

将这个结构对应回去(即对 op, e_1, e_2 做相应的赋值),我们就得到了之前的逻辑结构了.

最后,将一个形式的解析函数和另一个形式的输出函数复合起来,就能达到转换的目的了.例如,对于前缀转后缀的变换就是 $post.print(pre.parse(s))$,而其逆变换后缀转前缀则是 $pre.print(post.parse(s))$.

这样,通过定义逻辑结构和输出函数,我们就实现了两种不同形式之间的转换.这样做的一个附加的好处是,对于三个或者更多的格式之间的转换,我们也无需写若干个相互转换的函数,我们只需要构造出其等价的逻辑结构,和相对应数量的输出函数即可.

3 定义及证明

本章我们给出计算模型的严格定义.我们用一个形式语言来表示计算模型中需要用户指派的部分,该形式语言的语法如**错误!未找到引用源。**所示.我们的计算模型包括三个主要的部分:逻辑结构的定义和终端结点的

正则表达式定义,逻辑结构的输出和逻辑结构的解析.我们接下来分别介绍这三个部分.

statement	::= *redef* | *strdef* | *prdef*	*prdef*	::= *pname*(*stype*) = *outputs*
type	::= *rtype* | *stype*	*outputs*	::= *output* | *outputs*
redef	::= *rtype* = *regexp*		| *output*
strdef	::= *stype* → *forms*	*output*	::= { *terms* }
forms	::= *form* | *forms*	*terms*	::= *term*, *terms*
	| *form*		| *term*
form	::= { *fields* }	*term*	::= *name* | *const*
fields	::= *field*, *fields*	*name*	::= `[a-z]+`
	| *field*	*rtype*	::= `[a-z]+`
field	::= *type name*	*stype*	::= `[A-Z][a-z]*`
		const	::= 字符串常量

图 2　形式语言的语法

3.1　正则表达式与逻辑结构类型

模型中定义的两种类型(由**错误!未找到引用源。**中的 *type* 表示)包括正则表达式类型 *rtype* 和逻辑结构类型 *stype*,正则表达式类型的定义由 *redef* 所示,其中 *regexp* 表示一个正则表达式.逻辑结构类型的定义由 *strdef* 所示,一个逻辑结构由若干种 *form* 构成,其实际形态即为若干种 *form* 之一.*form* 可以理解为 C 语言中的一个 struct,由若干个 *field* 所构成,其中的 *type* 和 *name* 分别表示 *field* 的类型和名字.

3.2　逻辑结构的输出

从逻辑结构生成其某一表现形式的字符串的方式叫做该逻辑结构的一个输出,其定义由 *prdef* 所示.对于逻辑结构 $stype \rightarrow form_1 \mid form_2 \mid ... \mid form_n$,我们需要将其所有的 *form* 都定义一个相应的输出,即 $pname(stype) = output_1 \mid output_2 \mid ... \mid output_n$,其中 $output_i$ 为 $form_i$ 所对应的输出.为保证可逆性,$output_i$ 中必须不重不漏地出现 $form_i$ 的所有 *name*,即 *field* 的名字,除此之外则可以出现任意多的字符串常量 *const*.从逻辑结构的输出我们可以导出一个函数 *pname.print*:

$$pname.print(stype\{form_i\}) = out(output_i), \quad 1 \le i \le n$$

$$out(\{term_1, term_2, ..., term_k\}) = print(term_1) + print(term_2) + ... + print(term_k)$$

$$print(term) = \begin{cases} pname.print(term) & \text{若}term\text{为结构体类型} \\ term & \text{若}term\text{为正则表达式类型或字符串常量} \end{cases}$$

逻辑结构的输出要包含该逻辑结构的所有 *form* 所涉及到的全部其它逻辑结构,例如说,假设我们需要之前所述的表达式除了支持整数之外还允许使用变量,我们可以对 *Expr* 和 *post* 的定义做如下修改[1]:

$$\textbf{\textit{var}} = \texttt{[a-z]+}$$

$$\textbf{\textit{Expr}} \rightarrow \{\textbf{\textit{oper}}\ op, \textbf{\textit{Expr}}\ e_1, \textbf{\textit{Expr}}\ e_2\} \mid \{\textbf{\textit{Id}}\ i\}$$

$$\textbf{\textit{Id}} \rightarrow \{\textbf{\textit{number}}\ n\} \mid \{\textbf{\textit{var}}\ v\}$$

$$post(\textbf{\textit{Expr}}) = \{e_1, \text{" "}, e_2, \text{" "}, op\} \mid \{i\}$$

那么,为使 *post.print* 是良定义的,则还需要定义对 *Id* 的输出:$post(\textbf{\textit{Id}}) = \{n\} \mid \{v\}$,其对应的 *print* 函数为

$$post.print(\textbf{\textit{Id}}\{\textbf{\textit{number}}\ n\}) = n$$

$$post.print(\textbf{\textit{Id}}\{\textbf{\textit{var}}\ v\}) = v$$

3.3　逆向解析函数的构造

我们通过对逻辑结构的输出稍加修改,就能很容易地得出解析其输出字符串的上下文无关文法,对于一个输出 $pname(stype) = output_1 \mid output_2 \mid ... \mid output_n$,其文法为

$$stype_{pname} ::= production(output_1) \mid production(output_2)$$

[1] 对于这个例子来说,更简单的做法是直接定义正则表达式 id = `[0-9]+|[a-z]+`.

| ...
| $production(output_n)$

其中有 $production(\{term_1, term_2, ..., term_k\}) = node(term_1) + node(term_2) + ... + node(term_k)$,和

$$node(term) = \begin{cases} rtype_{pname} & 若term为结构体类型rtype \\ term & 若term为正则表达式类型或字符串常量 \end{cases}$$

我们的计算模型要求生成出来的文法符合某个可以自动解析的类别,这样我们就能自动导出语法分析器解析相应字符串构造出原逻辑结构.我们目前的实现要求生成出来的文法是 LR(1)的.现给出给定一组符合 LR(1)的输出规则后,*parse* 函数的执行过程:

- 首先通过一个 LR(1)语法分析器解析出语法树
- 对于当前的一颗树,考察其是由哪条产生式产生,用如下步骤构造具有相应 *form* 的一个逻辑结构
  - 对于根结点的所有孩子结点
    - 若其是一个字符串常量,则忽略它
    - 若其是一个正则表达式类型的终端结点,则对逻辑结构中相应变量进行赋值
    - 否则其必然是一个非终端结点,构造其所代表的逻辑结构,递归地对以该结点为根的子树进行转换,然后将转换后的逻辑结构赋值给相应变量

3.4 性质及证明

引理.对于任意的逻辑结构输出 p,若由 p 生成出的文法是 LR(1)的,那么就有 $p.parse = p.print^{-1}$,即解析函数是输出函数的反函数.

证明:设 p 是某一逻辑结构类型 A 的一个输出,a 为 A 的一个实例,$s = p.print(a)$,T 是 s 经过 LR(1)解析之后,忽略掉所有字符串常量结点的语法树.首先,根据语法分析的正确性,对于所有终端结点,其值必然与原结构 a 中相应正则表达式类型的 *field* 的值相等.其次,我们有 T 的高度至少为 2,因为其根结点是非终端结点 A_s 类型,故至少经过了一次规约.设其从产生式 $prod = production(output)$规约而来,以下根据 T 的高度进行归纳证明.

若 T 的高度为 2,则 T 是经过一次规约所产生的语法树,其孩子结点均为终端结点.由于我们要求每一个 *output* 都必须不重不漏地出现其相对应的 *form* 中所有 *field* 的名字,而根据 *production* 函数的定义,*prod* 也不重不漏地出现了所有 *field* 的名字.而终端结点的正确性有 LR 分析保证,故 T 可以正确地对应回原始的 a 结构.

若当 T 的高度小于 k 时结论均成立,则当其高度等于 k 时,由归纳假设可保证其孩子结点均能正确地对应回相应结构或正则表达式类型,同样类似以上分析,将其所有的孩子结点对应回去,即可得到 a 结构.

综上所述,我们有 $p.parse(s) = a$,由 a 取值的任意性,我们有 $p.parse = p.print^{-1}$.

定理.对于任意的逻辑结构类型 A,若其两个输出 p,q 都能正确地生成 *parse* 函数,则我们可以构造这两种形式之间的可逆变换:$f_{pq}:P \to Q, f_{qp}:Q \to P$,且 $f_{pq} = f_{qp}^{-1}$,其中 $P = \{p.print(a) \mid a \in A\}, Q = \{q.print(a) \mid a \in A\}$.

证明:令 $f_{pq} = q.print \circ p.parse, f_{qp} = p.print \circ q.parse$,则有 $f_{pq}:P \to Q, f_{qp}:Q \to P$ 且 $f_{qp}^{-1} = (p.print \circ q.parse)^{-1} = q.parse^{-1} \circ p.print^{-1} = q.print \circ p.parse = f_{pq}$.

4 实例研究

4.1 模型的实现

我们把我们的计算模型实现成为了一个基于 Scheme 的领域特定语言.实现采用了 Scheme 中的宏(macro)系统,这使得该语言的任意程序也是合法的 Scheme 程序,能够直接嵌入任何 Scheme 程序内,并能与 Scheme 中的内部数据结构进行直接交互.

4.2 MIPS32指令集上的汇编和反汇编

MIPS(Microprocessor without Interlocked Pipeline Stages)架构是一种采取精简指令集的处理器架构,现已被广泛使用在各种设备上.MIPS32 架构具有上百条指令,指令通常包括操作码,寄存器号,内存地址,立即数等部分.且汇编指令有两种形式:一种是易于程序员理解和记忆的助记符形式,另一种是实际的存储格式二进制形

式,汇编器将一系列助记符形式的指令转换为二进制的机器指令,反汇编器则将机器指令转换成助记符形式.为了验证我们的计算模型的表达能力和在实践中的可用性,我们采用我们的领域特定语言实现了 MIPS 指令集上的汇编和反汇编.

4.2.1 实例描述

本实例研究的任务就是采用我们的领域特定语言实现 MIPS 指令集上的汇编和反汇编.我们选用 MIPS 指令集是因为其在很多场景中广泛使用,使得该实例研究具有实践意义.同时 MIPS 指令集的规模适中,适合验证我们的语言.我们也考察了其他的一些可选项,如 Intel X86 指令集和 Java 虚拟机字节码.其中 X86 指令集过于庞大,难以在较短时间段内完成验证.而 Java 字节码的助记符形式和二进制形式之间的对应关系过于简单,不足以对语言的表达能力进行完整检验.

4.2.2 实现过程

如前所述,汇编和反汇编的过程就是把各条 MIPS 指令在助记符形式和二进制形式之间对应转换的过程.如果我们能用我们的语言建立起对应转换,就能实现 MIPS 的汇编和反汇编.

需要注意的是,有一些部分的转换天生具备数值计算的特点.例如,助记符形式的寄存器号和立即数等常数通常采用十进制表示,那么我们要将其转换成二进制表示.这类数值变换实现起来并不是十分方便.所以在具体实现中,我们设计了一个助记符形式的标准型,同时实现了一个辅助转换把任意的助记符形式转换成标准型.在该标准型中,所有的常数都采用二进制表示.由于 MIPS 汇编代码中采用了特定格式来标注立即数和内存地址,所以该辅助变换可以不用理解具体 MIPS 指令的含义,采用通用的方式完成转换.此外,该辅助变换同时过滤掉原汇编里面多余的空白,保证汇编代码呈现标准格式.

另一方面,由于我们的实现是完成针对文本的变换,所以我们只生成 ASCII 文本的 01 串,而不是实际的二进制数据.为了解决这个问题,我们设定二进制格式的标准型为 ASCII 文本的 01 串,同时实现两个辅助变换在 ASCII 的 01 文本和二进制编码之间转换.

实际的变换就在两个标准型之间进行,该变换的书写过程主要是用我们的语言建立每条指令中不同组成部分之间的对应,然后建立起多条指令之间的对应.MIPS 指令被划分为三种类型:R 型,I 型和 J 型.其中 R 型指令大多都是寄存器间的数学运算指令;I 型指令主要是运算指令和条件跳转指令,但与 R 型指令不同的是,对于 I 型的运算指令,其运算数带有一个 16 位整数;而 J 型指令,则绝大多数都是无条件跳转指令,带有一个 26 位的表示跳转地址的整数.在此我们从三种类型中各选出一些指令作为代表,对于类型中的其它指令,转换方法与所给例子是类似的.

4.2.2.1 R 型指令举例

add 指令的格式是 `add $rd, $rs, $rt`,其中 *rd,rs,rt* 为寄存器号,在此表示为 5 位 01 串,其二进制机器码为 `000000ssssstttttddddd00000100000`,其中 `s`,`t` 和 `d` 分别与 *rs,rt,rd* 相对应,故我们可以定义出一个逻辑结构 *Add* 以及其上的两个输出 *text,bin*,分别表示助记符和二进制形式:

$$\textbf{\textit{reg}} = \texttt{[01]\{5\}}$$

$$\textbf{\textit{Add}} \rightarrow \{\textbf{\textit{reg}}\ rd, \textbf{\textit{reg}}\ rs, \textbf{\textit{reg}}\ rt\}$$

$$text(\textbf{\textit{Add}}) = \{\texttt{"add \$"}, rd, \texttt{", \$"}, rs, \texttt{", \$"}, rt\}$$

$$bin(\textbf{\textit{Add}}) = \{\texttt{"000000"}, rs, rt, rd, \texttt{"00000100000"}\}$$

与 add 指令类似,subu 指令格式为 `subu $rd, $rs, $rt`,机器码为 `000000sssssttttddddd00000100011`,故其逻辑结构与输出也可仿制 add 指令进行定义.

4.2.2.2 I 型指令举例

ori 指令的格式是 `ori $rt, $rs, i`,其中 *i* 为一个 16 位无符号整数,其机器码为 `000000sssssttttiiiiiiiiiiiiiiii`,我们同样可以定义出其逻辑结构和两个输出:

$$\textbf{\textit{imm}} = \texttt{[01]\{16\}}$$

$$\textbf{\textit{OrI}} \rightarrow \{\textbf{\textit{reg}}\ rs, \textbf{\textit{reg}}\ rt, \textbf{\textit{imm}}\ i\}$$

$$text(\textbf{\textit{OrI}}) = \{\texttt{"ori \$"}, rt, \texttt{", \$"}, rs, \texttt{", "}, i\}$$

$$bin(\textbf{\textit{OrI}}) = \{\texttt{"001101"}, rs, rt, i\}$$

类似的 beq 指令格式为 `beq $rs, $rt, i`,机器码为 `000100ssssstttttiiiiiiiiiiiiiiii`,其定义也相差无几.

4.2.2.3 J 型指令举例

jump 指令的格式是 `j a`,其中 *a* 为一个代表指令地址的 26 位无符号整数,其机器码为 `000010aaaaaaaaaaaaaaaaaaaaaaaaaa`,对此我们有如下定义:

$$\textbf{\textit{addr}} = \texttt{[01]\{26\}}$$

$$\textbf{\textit{Jump}} \rightarrow \{\textbf{\textit{addr}}\ a\}$$

$$text(\textbf{\textit{Jump}}) = \{\texttt{"j "}, a\}$$

$$bin(\textbf{\textit{Jump}}) = \{\texttt{"000010"}, a\}$$

同样,我们也有与其类似的 jal 指令,其格式为 `jal a`,机器码为 `000011aaaaaaaaaaaaaaaaaaaaaaaa`.

4.2.2.4 若干种指令的组合

以上我们描述了对于给定的一种指令,其逻辑结构与两种形式的输出定义,但一条指令可能是多种指令之一,需要把它们进行组合.我们定义 *Instr* 类型为一条指令的逻辑结构(假设仅包含以上 6 条指令):

$$\textbf{\textit{Instr}} \rightarrow \{\textbf{\textit{Add}}\ a\} \mid \{\textbf{\textit{SubU}}\ s\} \mid \{\textbf{\textit{OrI}}\ o\} \mid \{\textbf{\textit{BEq}}\ b\} \mid \{\textbf{\textit{Jump}}\ j\} \mid \{\textbf{\textit{JAL}}\ j\}$$

其输出方式也能以简单直接的方式定义:

$$text(\textbf{\textit{Instr}}) = \{a\} \mid \{s\} \mid \{o\} \mid \{b\} \mid \{j\} \mid \{j\}$$

$$bin(\textbf{\textit{Instr}}) = \{a\} \mid \{s\} \mid \{o\} \mid \{b\} \mid \{j\} \mid \{j\}$$

4.2.3 结论与分析

首先,虽然 MIPS32 是一个中等规模的指令体系结构,采用我们的语言,我们能在较短时间完成汇编和反汇编的编写.总体代码长度也比较短.我们也注意对比了采用传统方式书写一对汇编和反汇编的工作量.为了完成两个变换,我们需要撰写两个语法制导的翻译过程,该翻译过程的两份代码基本是重复的.同时,程序员可能会不小心把两个语法制导翻译并没有写成互逆的形式,但采用传统的方法没有办法静态检查出这个错误,只能留给测试检查,增加了测试和调试的工作量,也增加了最终程序携带缺陷的可能性.另一方面,我们注意到在机器码格式上并不能很容易的构造词法分析器,这是因为用户提供的立即数可能会和操作码重复,我们没有办法在词法级别区分出不同的词法单元.而我们的计算模型中采用了将正则表达式嵌入语法分析器的做法,只有在语法分析请求下一个符号的时候我们才用相应的正则表达式去匹配符号,这样我们实际将语法信息引入了词法分析级别,可以很容易的解决机器码的词法分析构造问题.

其次,我们的模型完整的表示出了汇编标准型和二进制标准型之间的转换,同时在理论上我们也可以表示普通汇编和汇编标准型之间的转换.考虑到 MIPS 已经是一个中等规模的指令体系结构,这个结果说明我们的计算模型具有较强的表达能力,可以表达在实践中具有一定规模的变换.

再次,我们的方法在表达涉及数值计算的变换时并不是特别方便.但本实例研究同时也说明,涉及数值计算的变换往往具有一定的通用性,可以采取预先转换成标准型的方式避免在主要变换中涉及数值变换.在本实例研究中,经过标准型的分离之后,数值变换的处理在整体代码中并不占据太多比例,主要的代码仍然指令之间的对应,而指令之间的对应可以用我们的方法减少编码工作量,避免低级错误.

最后,我们的方法目前只允许固定的字符串常量,导致必须先过滤多余空白才能转换代码.为了解决这个问题,我们可以考虑加上特定的"空白"类型,使得解析函数遇到空白时自动忽略,这是一项未来工作.

5 相关工作

本文设计的语言是一个可逆语言.可逆语言允许从一个程序中导出两个转换函数,这两个函数互为反函数.有代表性的可逆语言为 Gluck 等人定义的命令式可逆语言[1].同我们的语言相比,该语言主要用于做可逆的数值变换而不是格式转换.同时,该语言定义在可逆图灵机上,其计算模型与传统语言有较大差别,需要花费一定

的学习成本.同时,现有的可逆语言强调在两个数据域之间的直接变换.而本文提出的方法引入了中间的逻辑结构,这样在存在多于两种形式的数据需要进行转换时,程序员的工作量仅随形式的数量线性增长,而不需要对多种形式两两编写转换方法.

双向语言主要用于编写双向变换.与可逆变换不同,双向变换允许两个数据域的信息量不完全对等,每个数据域可以包含自己私有的信息.因此,可逆变换可以看做是特殊情况下的双向变换,而双向变换语言通常在其表达能力允许的范围内也可以编写可逆变换.双向变换语言在处理字符串数据上代表语言是 Boomerang[3].同我们的方法相比,Boomerang 采用 lens combinator 作为其计算模型,和传统语言有较大差别,需要花费一定的学习成本.同时,Boomerang 采用正则类型系统,这就导致了 Boomerang 不能处理一些上下文无关文法.而本文的模型可以处理所有 LR(1)的文法.此外,Boomerang 也存在之前所述的处理多份数据之间的转换的问题.

除通用字符串变换外的领域,部分工作已经开始考虑应用上下文无关文法书写成对的变换.在 XML 处理领域,biXid[4]是一个在 XML 文档上的双向变换语言,它采用了关系式编程模型,该模型同使用两个上下文无关文法的形式类似.XSugar[5]是一个对纯文本和 XML 文档进行转换的语言,使用了与本文类似的对偶文法的概念.在自然语言翻译领域,synchronous grammar 采用两个对等书写的上下文无关文法来实现不同语言之间的转换.triple-graph grammar[6]把 synchronous grammar 扩展到图上以便处理更复杂数据结构,但同时由于图结构的复杂性,丢失了一些变换的特性.Grammatical framework[7]将 synchronous grammar 扩充成了一个完整的函数式编程语言,但为保证其计算能力,它也牺牲了部分变换的性质.同这些语言相比,我们的方法是第一个把上下文无关文法应用于通用字符串处理领域,探讨了基于正则表达式的词法解析等字符串解析领域特定的问题.同时,本文也明确提出了区分逻辑结构,解析函数和打印函数,并对可逆的性质进行了探讨与证明.最后,我们的实例研究是我们知识范围内第一个用上下文无关文法相关技术对主流体系结构的汇编/反汇编所进行的同步实现.

6 结语

上下文无关文法是解析字符串的有力工具,本文的计算模型借助其良好的表达能力,将其应用到数据转换上.对于可逆变换来说,变换的输入和输出都具有相同的本质逻辑:它们都是同一个结构的不同表现形式.本文的计算模型有效地利用了这一性质,将数据的逻辑结构抽象出来,将变换问题转化为输出和解析的问题,使得从文法出发构造可逆变换成为可能.经过实例验证,本文所提出的计算模型具有较强的表达能力.

7 致谢

本文受国家重点基础研究发展规划 973 资助项目(2011CB302604);国家高技术研究发展计划 863 资助项目(2013AA01A605);国家自然科学基金(61202071,61121063,U1201252)资助.

References:

[1] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. (2011). Towards a reversible functional language. In Proceedings of the Third international conference on Reversible Computation (RC'11), Alexis Vos and Robert Wille (Eds.). Springer-Verlag, Berlin, Heidelberg, 14-29.

[2] Janis Voigtländer. (2009). Bidirectionalization for free! (Pearl). In Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09). ACM, New York, NY, USA, 165-176.

[3] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., & Schmitt, A. (2008). Boomerang: Resourceful lenses for string data. ACM SIGPLAN Notices, 43(1), 407-419.

[4] Kawanaka, S., & Hosoya, H. (2006). biXid: a bidirectional transformation language for XML. ACM SIGPLAN Notices, 41(9), 201-214.

[5] Brabrand, C., Møller, A., & Schwartzbach, M. I. (2008). Dual syntax for XML languages. Information Systems, 33(4), 385-406.

[6] Schürr, A. (1995, January). Specification of graph translators with triple graph grammars. In Graph-Theoretic Concepts in Computer Science (pp. 151-163). Springer Berlin Heidelberg.

[7] Ranta, A. (2004). Grammatical framework. Journal of Functional Programming, 14(2), 145-189.