ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.10, October 2008, pp.2508−2516
DOI: 10.3724/SP.J.1001.2008.02508

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

传名调用演算的二值传递 CPS 变换*

喻 钢[1,2+], 柳欣欣[1]

[1](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)

[2](中国科学院 研究生院,北京 100049)

Two Values Passing CPS Transformation for Call-by-Name Calculus with Constants

YU Gang[1,2+], LIU Xin-Xin[1]

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

[2](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: E-mail: yug@ios.ac.cn

Yu G, Liu XX. Two values passing CPS transformation for call-by-name calculus with constants. *Journal of Software*, 2008,19(10):2508−2516. http://www.jos.org.cn/1000-9825/19/2508.htm

Abstract: In this paper, a new CPS (continuation-passing-style) transformation for Plotkin's call-by-name λ calculus with constants is proposed. It is based on evaluation contexts transformation and the features that two values, instead of one, are passed to the continuation every time. With encodings, a CPS language is introduced. Then, Plotkin's simulation theorem is proved by establishing 1-to-1 correspondence between the source language and CPS language. Compared with Plotkin's work, a reduction closed CPS language is defined in which all continuations are explicitly expressed as functional encodings and it is simpler to prove both the soundness and completeness directions of simulation theorem.

Key words: programming calculi; formal semantics; call-by-name; CPS (continuation-passing-style); reduction

摘 要: 为 Plotkin 带常数传名调用λ演算定义了一个新的 CPS(continuation-passing-style)变换方法.方法基于求值上下文变换,新颖之处在于,每次传递二值给继续而不是常规的一值.先给出二值 CPS 变换编码,再在此基础上定义 CPS 语言,最后建立源语言和 CPS 语言的一一映射关系并证明 Plotkin 的模拟定理.与 Plotkin 的工作比较,工作特点在于,给出了一个 CPS 归约闭语言,该语言中所有继续都可以用函数形式表达,且模拟定理的可靠性和完备性方向证明更为简单.

关键词: 程序演算;形式语义;传名调用;CPS(continuation-passing-style);归约

中图法分类号: TP301 **文献标识码**: A

继续(continuation)[1]在计算机科学中有许多应用,其中非常重要的一个应用是比较λ演算的不同归约策略.比如,要模拟某个项 M 在某策略ρ_1下的归约,可以首先将 M 变换为基于继续传递风格(continuation-passing-style,简称 CPS)[2]的项 M',然后用另外一种策略ρ_2来归约 M',如果 M 的归约结果和 M'的归约结果存在某个特定的联系,则我们说ρ_2可模拟ρ.Plotkin 在文献[2]中首先证明了传值调用(call-by-value,简称 CBV)策略和传名调用

* Supported by the National Natural Science Foundation of China under Grant Nos.60496321, 60673045 (国家自然科学基金)
Received 2007-01-29; Accepted 2007-08-03

(call-by-name,简称 CBN)策略可互相模拟.在模拟定理的完备性方向证明中,Plotkin 构造了一种特殊的操作符":"来辅助证明,并得到了不错的结果.但在可靠性方向,Plotkin 并没有提出好的解决方案,他归纳定义了一个"停机项"集合,并在此基础上运用无限序列反证了可靠性,Plotkin 并没有规范给出 CPS 项与源语言项之间的规约对应关系.另外,在文献[2]的最后,Plotkin 给出了不区分常数传名调用演算的 CPS 编码.但是,此编码不仅复杂,而且很难理解和证明.另外,此编码得到的 CPS 语言也不是一个严格意义上的 CPS 语言[3,4].

求值上下文(evaluation-contexts)[5]已被证实在函数式程序设计语言领域有着许多重要应用,这其中也包括描述λ演算的归约策略和操作语义[6].近年来,传值调用求值上下文已得到广泛研究,并在传值调用模拟上获得了相关的结果[7,8].但在传名调用研究方面,虽已有不少工作,如部分求值方法[9]、语义[10]、类型系统[11]、演算和逻辑[12]等,但与传名调用求值上下文相关的工作还鲜有报道.

本文针对 Plotkin 的不区分常数传名调用语言提出了一种新的 CPS 变换方法.该变换方法着眼于继续的函数形式表达,运用求值上下文翻译技术实现.与 Plotkin 的方法比较,新方法在下列方面有所改进:

(1) 给出了一种新的 CPS 归约闭语言,并逆向映射到原传名调用语言上.

(2) 新变换在结构上更加清晰,需要的额外(administrative)归约更少.

(3) 模拟定理的可靠性和完备性方向证明都更加容易.

本文第 1 节定义讨论对象:不区分常数传名调用语言,并给出程序的传名调用求值上下文定义以及基于此定义的项唯一分解引理和基于求值上下文归约引理.第 2 节给出二值传递 CPS 变换.第 3 节定义 CPS 规约闭语言,并证明 Plotkin 的模拟定理.第 4 节进行工作比较,首先比较对象语言相同的 Plotkin CPS 变换,然后比较区分常数语言和不区分常数语言的程序表达能力和 CPS 变换.最后总结本文工作并讨论后续研究方向.

1 常数传名调用λ演算

首先给出本文中定义的λ项语法:

$$\begin{aligned} x &\in \text{变元集}\\ a,b,c,d &\in \text{常数集}\\ M &::= V \mid (MN)\\ V &::= x \mid a \mid \lambda x.M \end{aligned}$$

依据常规,假定变元 M,N 表示语法产生的项(term),而 V,W 表示语法产生的值(value).如果项 L 的语法形式为 MN,则称 L 为组合.项为某个值当且仅当项不是一个组合.形为$\lambda x.M$ 的值被称为λ抽象.在本文中,变元的自由出现和受囿出现也按照常规方式定义.如果某个项M没有变元的自由出现,则称M为闭项或程序.本文用 $M[N/x]$ 表示将项 M 中所有的 x 自由出现都用 N 来替换得到的新项.

传名调用归约关系 $\rightarrow_n$ 定义为由下列规则归纳定义的最小关系:

$$(\lambda x.M)N \rightarrow_n M[N/x] \qquad \frac{constapply(a,b) \text{ is defined}}{(ab) \rightarrow_n constapply(a,b)}$$

$$\frac{M \rightarrow_n M'}{MN \rightarrow_n M'N} \qquad \frac{M \rightarrow_n M'}{xM \rightarrow_n xM'} \qquad \frac{M \rightarrow_n M'}{cM \rightarrow_n cM'}$$

Fig.1 Call-by-Name reduction $\rightarrow_n$

图 1 传名调用归约关系 $\rightarrow_n$

其中,偏函数 $constapply : Constant \times Constant \rightarrow^{p} closed\ Values$ 解释常数之间的归约关系.这里,*Constant* 是常数集,*closed values* 是闭项和值的交集.

与图 1 对应,不难得到传名调用求值上下文的定义:

$$E_{n0} ::= [] \mid E_{n0}[x[]] \mid E_{n0}[c[]] \mid E_{n0}[[]M].$$

实际应用中,程序(闭项)是关心的子集.因此,可以从 E_{n0} 获得一个对程序有效的子集 E_n:

$$E_n ::= [] \mid E_n[[]M] \mid E_n[c[]].$$

定义求值上下文填充函数 $\langle\cdot\rangle : E_n \times T \to T$:

$$[\ ]\langle M\rangle = M$$
$$E_n[c[]]\langle M\rangle = E_n\langle cM\rangle$$
$$E_n[[]N]\langle M\rangle = E_n\langle MN\rangle$$

引理 1.1(唯一分解引理). 任给程序 P,如果 P 不是一个值,那么存在唯一的 E,V 和 N,使得 $P = E\langle VN\rangle$,且 V 是某个λ抽象,或 V 和 N 都是值.

证明:施结构归纳于 P. □

引理 1.2(基于求值上下文的归约). 程序 P 的 $\to_n$ 关系可由下列两条规则导出:

$$E_n\langle(\lambda x.M)N\rangle \to_n E_n\langle M[N/x]\rangle,$$
$$E_n\langle ab\rangle \to_n E_n\langle constapply(a,b)\rangle \qquad \text{if } constapply(a,b) \text{ is defined.}$$

证明:结构归纳 E_n. □

2 二值传递 CPS 编码

本节给出常数传名调用演算的语义,即传名调用语言的 CPS 变换编码.在 Plotkin 的传统语言(文献[2]第 3 节)定义中,常数被划分为函数型常数和基本型常数,函数型常数只能和基本型常数按照特定顺序进行归约,其归约特性由其名字静态决定.所以,某常数 c 的 CPS 编码是由编程语言静态决定的.能否为所有常数定义一个公共的、一般的 CPS 编码?常数 c 在归约中是扮演一个函数型常数角色还是一个基本型常数角色将由归约过程动态决定.Plotkin 在文献[2]的最后一段中给出了此问题的一个解决方案,本节将给出一个全新的 CPS 编码.其核心思想在于继续的函数形式表达,由此给出求值上下文的 CPS 变换.本文用单射˜标记从源语言(传名调用语言)映射到 CPS 语言中的变元和常数,其余变元在 CPS 变换过程中产生.传名语言中项的 CPS 编码定义如下:

$$\underline{x} = \tilde{x}$$
$$\underline{V} = \lambda\kappa.\kappa\ \Phi_1(V)\Phi_2(V)$$
$$\underline{MN} = \lambda\kappa.\underline{M}\lambda\alpha.\lambda\beta.\alpha\underline{N}\kappa$$

Φ_1 和 Φ_2 定义在闭值上,其值域是项:

$$\Phi_1(\lambda x.M) = \lambda\tilde{x}.\underline{M}$$
$$\Phi_1(c) = \lambda m.\lambda\kappa.m(\lambda\alpha.\lambda\beta.\tilde{c}\beta\kappa)$$
$$\Phi_2(\lambda x.M) = \lambda\tilde{x}.\underline{M}$$
$$\Phi_2(c) = \tilde{c}$$

求值上下文翻译函数 $[\![\bullet]\!]$ 定义在 E_n 上,其值域是 CPS 项,完成求值上下文到继续的函数形式翻译:

$$[\![[]]\!] = \lambda\alpha.\lambda\beta.\beta$$
$$[\![E_n[[]M]]\!] = \lambda\alpha.\lambda\beta.\alpha\underline{M}[\![E_n]\!]$$
$$[\![E_n[c[]]]\!] = \lambda\alpha.\lambda\beta.\tilde{c}\beta[\![E_n]\!]$$

程序作 CPS 变换后,继续成为显式变元出现在 CPS 程序中,是从值到"当前程序剩余计算"的函数.通常情况下,继续是一元函数,接收一个 CPS 归约值,返回整个计算结果.在我们的编码中,继续表示为二元函数(准确讲是概念上的二元函数,具体表示为 curry 的编码形式),即值 V 的两个 CPS 编码 $\Phi_1(V)$和$\Phi_2(V)$,连续传递给继续,并由继续根据归约状态选择其中某个值进行余下归约.因此,V 的 CPS 变换为 $\underline{V} = \lambda\kappa.\kappa\ \Phi_1(V)\Phi_2(V)$.确定二值传递后,依据传名调用演算不归约参数的特征,我们得到变元 x 和组合 MN 的 CPS 变换项.

再看值和求值上下文的变换:如果 V 是一个λ抽象,则无论$\underline{\lambda x.M}$出现在继续中的什么位置,CPS 编码替换引理(引理 2.1)是唯一需要满足的编码条件,由此可导出λ抽象的两个编码值相同.如果 V 是一个常数 c,则情况稍显复杂.可以将$\Phi_1(c)$称作 c 的"函数"常数编码,因为$\Phi_1(c)$的设计目标是做几步额外归约,让 c 在下一继续中表现为

一个“函数”常数的角色.与之对应地,$\Phi_2(c)$可以成为 c 的“基本”常数编码,因为传递$\Phi_2(V)$给继续等同于传递一个基本常数 c 给当前计算.

由给出的 CPS 编码族,不难得到下列命题:

引理 2.1(CPS 编码替换). $\underline{M[N/x]} = \underline{M}[N/x]$.

命题 2.2(二值传递). $\underline{V}[\![C]\!] \to^2 [\![C]\!]\Phi_1(V)\Phi_2(V)$.

命题 2.3(函数常数传递). $\Phi_1(c)\underline{M}[\![E]\!] \to^* \underline{M}[\![E[c[]]]\!]$.

命题 2.4(求值上下文的额外归约). $\underline{E_1\langle N\rangle}[\![E_2]\!] \to^* \underline{N}[\![E_2[E_1[]]]\!]$.

其中,$\to$表示 CPS 归约,是$\to_n$的子集,其定义将在第 3.2 节中给出.$\to^2$表示长度为 2 的连续$\to$规约,$\to^*$表示有限长度的连续$\to$规约.

3 CPS 归约语言

假定 L 是第 1 节定义的传名调用语言,本节将定义由 L 的 CPS 变换得到的 CPS 归约语言 L'.本节研究 L' 作为 CPS 语言所拥有的一些特殊属性,也研究 L 和 L'之间存在的映射与反映射关系.L'中的变元和常数包括从 L 中由$\tilde{}$映射过来的常数和变元,外加 4 个特殊变元:α,β,m 和κ.

与 Plotkin[2]一样,首先定义 *constapply* 的 CPS 变换:

定义 3.1. $constapply(\tilde{c},\tilde{d}) \triangleq \underline{constapply(c,d)}$.

3.1 语言组件

定义项集由下列规则归纳给出:

$$C ::= \tilde{x} \mid \lambda\kappa.\kappa(\lambda m.\lambda\kappa.m(\lambda\alpha.\lambda\beta.\tilde{c}\beta\kappa))(\tilde{c})$$
$$\lambda\kappa.\kappa(\lambda\tilde{x}.C_1)(\lambda\tilde{x}.C_1) \mid \lambda\kappa.C_1(\lambda\alpha.\lambda\beta.\alpha C_2\kappa)$$

由 C 产生的项集合称为 CPS_Code.显然,CPS_Code 是第 1 节定义的 λ 项的一个子集.定义函数 $\lfloor\_\rfloor: CPS\_code \to Term$:

$$\lfloor\tilde{x}\rfloor = x$$
$$\lfloor\lambda\kappa.\kappa(\lambda m.\lambda\kappa.m(\lambda\alpha.\lambda\beta.\tilde{c}\beta\kappa))(\tilde{c})\rfloor = c$$
$$\lfloor\lambda\kappa.\kappa(\lambda\tilde{x}.C_1)(\lambda\tilde{x}.C_1)\rfloor = \lambda x.\lfloor C_1\rfloor$$
$$\lfloor\lambda\kappa.C_1(\lambda\alpha.\lambda\beta.\alpha C_2\kappa)\rfloor = \lfloor C_1\rfloor\lfloor C_2\rfloor$$

同样地,根据下列规则定义项集 *Cont*:

$$K ::= \lambda\alpha.\lambda\beta.\beta \mid \lambda\alpha.\lambda\beta.\alpha CK \mid \lambda\alpha.\lambda\beta.\tilde{c}\beta K$$

这里,$C \in CPS\_Code$.定义函数$[\![\ ]\!]^{-}: Cont \to E_n$:

$$[\![\lambda\alpha.\lambda\beta.\beta]\!]^{-} = []$$
$$[\![\lambda\alpha.\lambda\beta.\alpha CK]\!]^{-} = [\![K]\!]^{-}[[]\lfloor C\rfloor]$$
$$[\![\lambda\alpha.\lambda\beta.\tilde{c}\beta K]\!]^{-} = [\![K]\!]^{-}[c[]]$$

最后,定义 CPS 语言中的值映射集合.L'用两个集合 $FunV$ 和 $BasicV$ 来处理传名调用中的值映射.其中,集合 $FunV = \{\lambda m.\lambda\kappa.m(\lambda\alpha.\lambda\beta.\tilde{c}\beta\kappa), \lambda\tilde{x}.C\}$ $(C \in CPS\_Code)$, $FunV$ 的元素在 L'中可称为函数值.集合 $BasicV = \{\tilde{c}, \lambda\tilde{x}.C\}$ ($C \in CPS\_Code$),其元素在 L'中称为基本值.

定义函数 $\Psi_1: FunV \to V$ 和 $\Psi_2: BasicV \to V$:

$$\Psi_1(\lambda m.\lambda\kappa.m(\lambda\alpha.\lambda\beta.\tilde{c}\beta\kappa)) = c$$
$$\Psi_1(\lambda\tilde{x}.C) = \lambda x.\lfloor C\rfloor$$
$$\Psi_2(\tilde{c}) = c$$
$$\Psi_2(\lambda\tilde{x}.C) = \lambda x.\lfloor C\rfloor$$

命题 3.2. _ 和 $\lfloor\ \rfloor$,$[\![\ ]\!]$ 和 $[\![\ ]\!]^-$,Φ_1 和 Ψ_1 以及 Φ_2 和 Ψ_2 是 4 组逆函数对.

命题 3.3. *CPS_Code*,*Cont*,*FunV*,*BasicV* 中的元素在 L' 中都是值.

3.2 CPS语言

本节定义 CPS 语言,即定义 L' 的项集 *CPS_Prog* .

定义 3.4.

$$\begin{aligned} CPS\_Prog = BasicV &\cup \{CK \mid C \in CPS\_Code, K \in Cont\} \\ &\cup \{KVW \mid K \in Cont, V \in FunV, W \in BasicV\} \\ &\cup \{\tilde{c}WK \mid W \in BasicV, K \in Cont\} \\ &\cup \{VCK \mid V \in FunV, C \in CPS\_Code, K \in Cont\} \end{aligned}$$

假定 M 是 *CPS_Prog* 中的元素,V_n (n=1,2,…)是 L' 中的值,由命题 3.3 和定义 3.4 不难得出 M 及其后继规约生成项只可能是 V_1,V_1V_2 或者 $V_1V_2V_3$ 三种形式之一.由此,得到了 CPS 语言中项的特殊规约性质:在 CPS 语言中,项的求值上下文可简化为 $E ::= [] \mid E[]V$ .CPS 规约 $\rightarrow$ 是由下列 3 条规则归纳定义产生的最小集:

$$(\lambda x.M)V \rightarrow M[V/x]$$

$$\frac{constapply(a,b) \text{ is defined}}{(ab) \rightarrow constapply(a,b)}$$

$$\frac{M \rightarrow M'}{MV \rightarrow M'V}$$

引理 3.5(CPS 归约闭语言). 若 $M \in CPS\_Prog$, $M \rightarrow N$ 则 $\exists N'.N \rightarrow^* N'$ 且 $N' \in CPS\_Prog$.

证明:对 *CPS_Prog* 集合作情况分析. □

3.3 模拟定理

本节建立 CPS 语言 L' 和源语言 L 之间的映射和反映射关系.

首先,定义函数 $(\ )^- : CPS\_Prog \rightarrow Terms$

$$\begin{aligned} (W)^- &= \Psi_2(W) \\ (CK)^- &= [\![K]\!]^- \langle \lfloor C \rfloor \rangle \\ (VCK)^- &= [\![K]\!]^- \langle \Psi_1(V) \lfloor C \rfloor \rangle \\ (\tilde{c}WK)^- &= [\![K]\!]^- \langle c\Psi_2(W) \rangle \\ ((\lambda\alpha.\lambda\beta.\beta)VW)^- &= \Psi_2(W) \\ ((\lambda\alpha.\lambda\beta.\alpha CK)VW)^- &= \left([\![K]\!]^- [][\lfloor C \rfloor]\right) \langle \Psi_1(V) \rangle \\ ((\lambda\alpha.\lambda\beta.\tilde{c}\beta K)VW)^- &= \left([\![K]\!]^- [c[]]\right) \langle \Psi_2(W) \rangle \end{aligned}$$

完备性方向的证明:

引理 3.6(单步 CPS 归约). 若 $P \in CPS\_Prog$, $P \rightarrow P'$,则 $\exists P''.P'' \in CPS\_Prog$, $P' \rightarrow^* P''$ 且 $P^- \rightarrow_n^* P''^-$.

证明:对 *CPS_Prog* 集合作情况分析,并添加 $P^- \rightarrow_n^* P''^-$ 的检查.需要注意归约 $\tilde{c}WK \rightarrow CK$ 和 $(\lambda x.C_1)C_2K \rightarrow C_3K$,当且仅当在应用这两条归约规则时,有 $P^- \rightarrow_n P''^-$,我们称其为实归约(real reduction).产生其他归约时,都只能得到 $P^- = P''^-$,我们称这些归约为额外归约(administrative reduction).设计好的 CPS 变换语言的一个重要目标是减少额外归约[5]. □

图 2 显示了 *CPS_Prog* 的归约情况,实归约用实线标出,额外归约用虚线标出,归约箭头上的数字表示状态迁移时的归约步数.

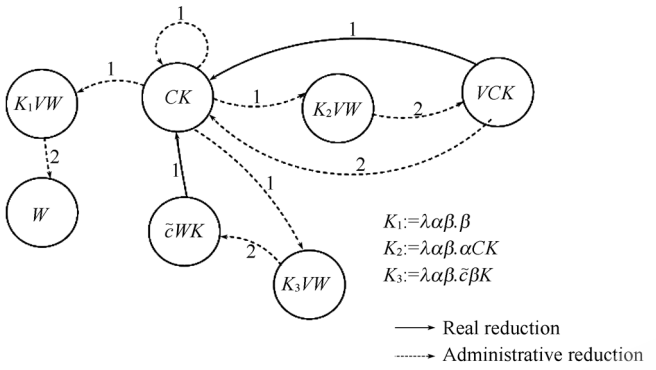

Fig.2 Reduction graph for CPS language

图 2 CPS 语言归约图

引理 3.7(多步 CPS 归约). 若 $P \in CPS\_Prog$, $P \to^* P'$,则 $\exists P''.P'' \in CPS\_Prog$, $P' \to^* P''$ 且 $P^- \to_n^* P''^-$.

证明:施归纳于 $P \to^* P'$.

- 基本情况: $P'' = P' = P$.
- 归纳情况:假定 $P \to^* P_0'$,由归纳假设可得 P_0'',使得 $P_0' \to^* P_0''$ 且 $P^- \to_n^* P_0''^-$.同时,由条件 $P_0' \to P'$ 可得, $P_0' \to^* P_0''$ 或 $P_0'' \to P'$.

  · $P_0' \to^* P_0''$:有 $P'' = P_0''$.

  · $P_0'' \to P'$:由归纳假设可得 P_1'',有 $P' \to^* P_1''$ 且 $P_0''^- \to_n^* P_1''^-$,合并第 1 次归纳假设,可得 $P \to^* P_1''$ 以及 $P^- \to_n^* P_1''^-$,可得 $P'' = P_1''$. □

模拟定理的可靠性方向:

引理 3.8. $C\langle M\rangle = E\langle ab\rangle$,则有 $\underline{M}[\![C]\!] \to^* \tilde{a}\tilde{b}[\![E]\!]$.

引理 3.9. $C\langle M\rangle = E\langle(\lambda x.N_1)N_2\rangle$,则有 $\underline{M}[\![C]\!] \to^* (\lambda\tilde{x}.\underline{N_1})\underline{N_2}[\![E]\!]$.

引理 3.10(单步传名调用归约). $M \to_n N$,对所有的 P,若 $P^- = M$,则存在 Q,使得 $Q^- = N$ 且 $P^- \to^* Q^-$.

证明:由引理 1.2, $M \to_n N$ 有下列两种情况:

- $E_n\langle(\lambda x.N_1)N_2\rangle \to_n E_n\langle N_1[N_2/x]\rangle$:假定 $P^- = E_n\langle(\lambda x.N_1)N_2\rangle$,由引理 3.9 可得 $P \to^* (\lambda\tilde{x}.\underline{N_1})\underline{N_2}[\![E_n]\!]$,而 $(\lambda\tilde{x}.\underline{N_1})\underline{N_2}[\![E_n]\!] \to \underline{(N_1[N_2/x])}[\![E_n]\!]$,且 $\underline{((N_1[N_2/x])}[\![E_n]\!])^- = E_n\langle N_1[N_2/x]\rangle$,故 $Q = \underline{(N_1[N_2/x])}[\![E_n]\!]$.
- $E_n\langle ab\rangle \to_n E_n\langle constapply(a,b)\rangle$:证明方法同上,根据定义 3.1 有 $Q = \underline{(constapply(a,b))}[\![E_n]\!]$. □

引理 3.11(多步传名调用归约). $M \to_n^* N$,对所有的 P,若 $P^- = M$,则存在 Q,使得 $Q^- = N$ 且 $P^- \to^* Q^-$.

证明:结构归纳于 $M \to_n^* N$. □

最后给出模拟定理的证明.

定理 3.12(CPS 模拟定理). 若 M 是一个程序, V 是一个值,则 $M \to_n^* V$ 当且仅当 $\underline{M}[\![[]]\!] \to^* \Phi_2(V)$.

证明:$\Rightarrow$:有 $(\underline{M}[\![[]]\!])^- = M$,根据引理 3.11,可找到 Q 使得 $Q^- = V$ 且有 $M \to^* Q^-$,由 CPS_Prog 的定义, Q 只能是 $(\lambda\alpha.\lambda\beta.\beta)\Phi_2(V)$ 或 $\Phi_2(V)$,两种情况下都成立.

$\Leftarrow$: $\underline{M}[\![[]]\!] \in CPS\_Prog$,由引理 3.7,可找到 W 使得 $(\Phi_2(V))^- = W$,可得 $V = W$,定理得证. □

4 相关工作比较

4.1 与Plotkin的工作比较

首先对照给出 Plotkin[2]以及本文对第 1 节定义的传名调用语言定义的 CPS 变换.

Table 1 Comparison of CPS transformation

表 1 CPS 变换比照

Plotkin encoding	Encoding in this paper
$\underline{x}=\tilde{x}$ $\underline{b}=\lambda\kappa.\kappa(\lambda\pi.\pi K\tilde{b})$ $\underline{\lambda x.M}=\lambda\kappa.\kappa(\lambda\pi.\pi(KI)(\lambda\tilde{x}.\underline{M}))$ $MN=\lambda\kappa.\kappa\underline{M}(\lambda\alpha.(\alpha(KI))((\alpha K)(\lambda\delta.(NI)(KI))(\lambda\delta.\underline{N})I)\kappa))$ $K\triangleq\lambda x.\lambda y.x$ $I\triangleq\lambda x.x$	$\underline{x}=\tilde{x}$ $\underline{b}=\lambda\kappa.\kappa(\lambda m.\lambda\kappa.m(\lambda\alpha.\lambda\beta.\tilde{b}\beta\kappa))\tilde{b}$ $\underline{\lambda x.M}=\lambda\kappa.\kappa(\lambda\tilde{x}.\underline{M})(\lambda\tilde{x}.\underline{M})$ $\underline{MN}=\lambda\kappa.\underline{M}(\lambda\alpha.\lambda\beta.\alpha\underline{N}\kappa)$

显然,从结构上看,本文提出的 CPS 编码比 Plotkin 的编码更清晰、明了.

我们来看动态归约性质:考虑发送函数常数的情况,在 Plotkin 的编码中,项 $\underline{b}\underline{M}\kappa$ 通过 n 步 CPS 额外归约可得中间结果 $\tilde{b}((\underline{M}I)(KI)\kappa$.如果想让归约继续,则需要引入传值调用规则 $\dfrac{M\to_v M'}{VM\to_v VM'}$.此时,CPS 项的继续只能表示为 $\tilde{b}((\_)(KI))\kappa$,它不是一个函数形式,从这里开始,CPS 语言中的调用不全部是尾调用(tail call)[13].因此,Plotkin 的 CPS 语言不能称为一种完全"CPS"风格语言.我们再来看使用本文的编码情况,假定 $\kappa=[\![E]\!]$,有 $\underline{b}\underline{M}\kappa\to^{*}\underline{M}[\![E[b[]]]\!]$.此时,继续表示为 $[\![E[b[]]]\!]$,仍旧是函数形式.由引理 3.5 和定义 3.4 可知,本文提出的 *CPS_Prog* 在所有归约状态都有继续的函数形式表达,是完全 CPS 风格语言.

最后,额外归约的数量也是评价 CPS 编码的重要指标[5].表 2 比较了几种值传递情况下的额外归约.

Table 2 Comparison of CPS administrative reductions (ARs)

表 2 CPS 额外归约比较

	Encoding in this paper			Plotkin encoding		
Argument type	Start	End	ARs	Start	End	ARs
Func. constant	$\underline{b}\underline{M}\kappa$	$\underline{M}(\lambda\alpha.\lambda\beta.\tilde{b}\beta\kappa)$	6	$\underline{b}\underline{M}\kappa$	$\tilde{b}((\underline{M}I)(KI)\kappa$	13
Basic constant	$\underline{b}(\lambda\alpha.\lambda\beta.\tilde{a}\beta\kappa)$	$\tilde{a}\tilde{b}\kappa$	3	$\tilde{a}((\underline{b}I)(KI))\kappa$	$\tilde{a}\tilde{b}\kappa$	6
λ abstraction	$\underline{\lambda x.M}(\lambda\alpha.\lambda\beta.\alpha\underline{N}\kappa)$	$(\lambda\tilde{x}.\underline{M})\underline{N}\kappa$	3	$\underline{\lambda x.MZ}$	$(\lambda\tilde{x}.\underline{M})\underline{N}\kappa$	13

Note: $Z\triangleq(\lambda\alpha.(\alpha(KI))((\alpha K)(\lambda\delta.(\underline{N}I)(KI))(\lambda\delta.\underline{N})I)\kappa)$.

不难得出,在各种情况下,较之 Plotkin 编码,本文提出的 CPS 编码需要的额外归约(ARs)更少.

4.2 与区分常数传名调用语言比较

区分常数的传名调用语言(文献[2]第 3 节)CPS 变换表示如下:

$$\underline{x}=\tilde{x}$$
$$\underline{f}=\lambda\kappa.(\kappa\lambda m.\lambda\kappa.m(\lambda\alpha.\tilde{f}\alpha\kappa))$$
$$\underline{b}=\lambda\kappa.\kappa\tilde{b}$$
$$\underline{\lambda x.M}=\lambda\kappa.\kappa\lambda\tilde{x}.\underline{M}$$
$$\underline{MN}=\lambda\kappa.\underline{M}(\lambda\alpha.\alpha\underline{N}\kappa)$$

不难看出,区分常数语言的语义来得更简单、明了,进行 CPS 模拟时需要的额外归约也更少.

但是,区分常数语言存在下列两个主要问题:

(1) 偏函数 *constapply* 的表达能力问题:区分常数语言的定义指出,常数根据其名字静态划分为函数常数

$FunConst(f,g,...)$ 和基本常数 $BasicConst(b,c,...)$,并定义 $constapply: FunConst \times BasicConst \to^P closed\ Values$.此定义实际上将函数常数限定为一阶常数,只允许函数常数和基本常数之间的规约.不允许函数型常数之间归约,比如,已知谓词≥0,想定义高阶常数 *not*,然后去定义新谓词<0,即需要:

$$<0 \triangleq constapply(not, \geq 0).$$

在区分常数语言中 *not* 和上式的<0 都无法定义,而在本文第 1 节定义的不区分常数语言却是可定义的.换句话说,不区分常数语言允许定义高阶常数,而区分常数语言只允许定义一阶常数,不区分常数语言的表达能力强于区分常数语言.

(2) CPS 模拟步骤不一致问题:假定 $\tilde{b}$ 是区分常数语言的基本常数,CPS 项 M 模拟项 $E_n\langle bN\rangle$.不难得出 $M \to^* \tilde{b}\underline{N}[\![E_n]\!]$,然后停机.再看源语言(传名调用语言)的情况,假定 $N \to_n N'$,则有 $E_n\langle bN\rangle \to_n E_n\langle bN'\rangle$.现在,源语言继续归约,而模拟语言停机,即,模拟语言停止了对源语言的模拟,我们说 CPS 模拟出现了步骤不一致性问题.模拟步骤不一致显然不是 CPS 变换所希望的结果.我们来看不区分常数语言中的情况:

$$\Phi_1(b)\underline{N}[\![E_n]\!] \to^* \underline{N}[\![E_n[b[]]]\!] \to^* \cdots$$

CPS 项并不停机,而是继续模拟源语言项.由引理 3.5 和定理 3.6 可得,CPS 项当且仅当归约至 $\tilde{a}W[\![E_n]\!]$ 且 W 不是常数或 $W=\tilde{b}$ 但 $constapply(a,b)$ 无定义的情况下停机.这与源语言中规约项仅在 $E_n\langle aV\rangle$ 且 V 不是常数或 $V=b$ 但 $constapply(a,b)$无定义的情况下停机等同.所以,不区分常数语言没有模拟步骤不一致问题.

5 结 语

如何形式化地定义λ演算的求值策略是程序设计语言的重要理论基础,也是重要的研究领域.Plotkin 的著名文献[2]奠定了这一方向的研究基础,活性(eager)语言的传值调用求值策略已得到广泛研究.而惰性(lazy)语言的传名调用研究则稍显不足.用继续变换来表达传名调用演算相关研究更是鲜有报道.

本文定义了一种新的常数传名调用演算的继续变换:首先定义传名调用语言及其求值上下文,然后阐明了本文核心思想:求值上下文的 CPS 变换.此变换定义每个源语言归约中产生的值被翻译成 CPS 语言中的二值,然后传递给继续函数,继续函数选择其中一值构造下一求值项和继续函数,使得 CPS 语言中的所有继续都是函数形式.随后根据这种二值传递模式,本文定义了 CPS 语言及其反向映射函数,并证明了 CPS 语言和源语言之间的模拟定理.

与 Plotkin 的工作比较:本文提出的 CPS 编码结构更加清晰、明了,是完全 CPS 风格语言且产生的额外归约更少.同时,本文也比较了区分常数和不区分常数两种传名调用语言,可得出下列结论:不区分常数语言的 CPS 变换复杂,或者说其语义更为复杂.但语言更一般,可表达高阶常数,也不存在 CPS 模拟过程中的步骤不一致问题.

本文中所有的定义和证明已经在定理证明器 Isabelle[14]上检验通过,对细节有兴趣的读者可以通过作者的 E-mail 方式获取本文证明的 Isabelle 源代码.

总结本文的工作可得出:定义求值上下文及其变换,然后给出变换语言与源语言的模拟关系可以看作是求值策略形式定义研究的一般方法.如何将本文的工作拓展到并发演算,以及将本文工作继续一般化并和 Moggi 的计算型(computational)λ演算[15]联系起来将是下一步的研究目标.

References:

[1] Reynolds JC. The discoveries of continuations. Lisp and Symbolic Computation, 1993,6(3-4):233−248.

[2] Plotkin GD. Call-by-Name, call-by-value- and the lambda-calculus. Theoretical Computer Science, 1975,1(2):125−159.

[3] Sabry A, Felleisen M. Reasoning about programs in continuation-passing style. Lisp and Symbolic Computation, 1993,6(3-4): 289−360.

[4] Appel A. Compiling with Continuations. Cambridge: Cambridge University Press, 1992.

[5] Danvy O, Filinski A. Representing control, a study of the CPS transformation. Mathematical Structures in Computer Science, 1992, 2(4):361−391.

[6] Felleisen M, Friedman D. Control operators, the SECD machine and the λ-calculus. In: Wirsing M, ed. Formal Description of Programming Concepts III. Amsterdam: North-Holland, 1986. 193−217.

[7] Sabry A. The Formal relationship between direct and continuation-passing style optimizing compilers: A synthesis of two paradigms [Ph.D. Thesis]. Houston: Rice University, 1994.

[8] Felleisen M, Friedman DP, Kohlbecker E, Duba B. Reasoning with continuations. In: Proc. of the Symp. on Logic in Computer Science (LICS'86). Washington: IEEE Computer Society Press, 1986. 131−141.

[9] Nielsen K, Srensen MH. Call-by-Name cpstranslation as a binding-time improvement. In: Mycroft A, ed. SAS'95: Proc. of the 2nd Int'l Symp. on Static Analysis. London: Springer-Verlag, 1995. 296−313. http://citeseer.ist.psu.edu/nielsen95callbyname.html

[10] Boudol. On the semantics of the call-by-name CPS transform. Theoretical Computer Science, 2000,234(1-2):309−348.

[11] Thielecke H. Answer type polymorphism in call-byname continuation passing. In: Schmidt DA, ed. Programming Languages and Systems, the 13th European Symp. on Programming, ESOP 2004, Held as Part of the Joint European Conf. on Theory and Practice of Software, ETAPS 2004. LNCS 2986, Barcelona: Springer-Verlag, 2004. 279−293.

[12] Wadler P. Call-by-Value is dual to call-by-name. In: Runciman C, Shivers O, eds. Proc. of the 8th ACM SIGPLAN Int'l Conf. on Functional Programming, ICFP 2003. New York: ACM Press, 2003. 189−201. http://doi.acm.org/10.1145/944705.944723

[13] Clinger WD. Proper tail recursion and space efficiency. In: Wiedijk F, ed. Proc. of the ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI-98). New York: ACM Press, 1998. 174−185. http://doi.acm.org/10.1145/277650.277719

[14] Wenzel M, Paulson L. Isabelle/isar. In: Wiedijk F, ed. Proc. of the 17th Provers of the World. LNCS 3600, Springer-Verlag, 2006. 41−49.

[15] Moggi E. Computational lambda-calculus and monads. Technical Report, ECS-LFCS-88-66, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.

喻钢(1977－),男,湖北武汉人,博士,主要研究领域为基于继续的程序设计语言形式语义和实现技术.

柳欣欣(1962－),男,博士,研究员,博士生导师,主要研究领域为形式语义,并发理论,进程演算.