软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(5):1221−1232 [doi: 10.13328/j.cnki.jos.005206]

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

peC 语言的部分求值器及在编译器测试中的应用*

郭德贵[1,2], 王冠成[1], 吕 帅[1,2,3], 刘 磊[1,4]

[1](吉林大学 计算机科学与技术学院,吉林 长春 130012)

[2](符号计算与知识工程教育部重点实验室(吉林大学),吉林 长春 130012)

[3](吉林大学 数学学院,吉林 长春 130012)

[4](吉林大学 软件学院,吉林 长春 130012)

通讯作者: 刘磊, E-mail: liulei@jlu.edu.cn

摘 要: 部分求值技术在程序优化及软件自动生成等方面起着极为重要的作用.将部分求值技术应用到编译器测试中.为此,设计了一种 C 语言的子集 peC 语言,给出了该语言的部分求值策略的形式化描述,实现了 peC 语言的部分求值器,设计了基于部分求值技术的编译器测试框架.通过实验,该方法可以检测出大部分之前其他方法发现的 GCC,LLVM 编译器中的错误,此外还发现了其他方法不能发现的错误,这表明,将部分求值技术应用到编译器测试中是有效的.

关键词: 部分求值;剩余程序;测试用例;编译器测试;抽象语法树

中图法分类号: TP314

中文引用格式: 郭德贵,王冠成,吕帅,刘磊.peC 语言的部分求值器及在编译器测试中的应用.软件学报,2017,28(5):1221−1232. http://www.jos.org.cn/1000-9825/5206.htm

英文引用格式: Guo DG, Wang GC, Lü S, Liu L. Partial evaluator for peC and its application to compiler validation. Ruan Jian Xue Bao/Journal of Software, 2017,28(5):1221−1232 (in Chinese). http://www.jos.org.cn/1000-9825/5206.htm

Partial Evaluator for peC and Its Application to Compiler Validation

GUO De-Gui[1,2], WANG Guan-Cheng[1], LÜ Shuai[1,2,3], LIU Lei[1,4]

[1](College of Computer Science and Technology, Jilin University, Changchun 130012, China)

[2](Key Laboratory of Symbolic Computation and Knowledge Engineering, Ministry of Education (Jilin University), Changchun 130012, China)

[3](College of Mathematics, Jilin University, Changchun 130012, China)

[4](College of Software, Jilin University, Changchun 130012, China)

Abstract: Partial evaluation plays an important role in many areas such as program optimization and automatic software generation. This paper applies partial evaluation to validating compilers. The design of peC, which is a subset of C, and the formalized description of partial evaluation strategy for peC are presented. Furthermore, the implementation of a partial evaluator for peC, and a compiler testing framework based on partial evaluation are provided. Experiments show that this new approach can not only detect errors which can be detected by other methods in GCC and LLVM but also found some errors which are not detected by the other methods. In summary, the work by this paper demonstrates applying partial evaluation in testing compilers is effective.

* 基金项目: 吉林省科技发展计划(20150101054JC, 20140520069JH, 20150520060JH); 国家自然科学基金(61300049); 教育部高等学校博士学科点专项科研基金(20120061120059); 吉林大学研究生创新基金(2016181)

Foundation item: Key Program for Science and Technology Development of Jilin Province of China (20150101054JC, 20140520069JH, 20150520060JH); National Natural Science Foundation of China (61300049); Specialized Research Fund for the Doctoral Program of Higher Education of Ministry of Education of China (20120061120059); Graduate Innovation Fund of Jilin University (201681)

收稿时间: 2016-06-02; 修改时间: 2016-09-25; 采用时间: 2016-12-07; jos 在线出版时间: 2017-01-20

CNKI 网络优先出版: 2017-01-20 16:06:40, http://www.cnki.net/kcms/detail/11.2560.TP.20170120.1606.016.html

Key words: partial evaluation; redundant program; test case; compiler testing; abstract syntax tree

部分求值思想[1]自 1976 年被提出来以后,经过不断补充和完善,Jones 等人于 1996 年完整地描述了部分求值理论,并提出了较为完善的程序例化思想[2].部分求值技术为研究程序设计语言的性质提供了更加便捷的途径,在科学计算、逻辑推理、专家系统等领域中扮演着非常重要的角色.近年来,部分求值技术在程序优化[3,4]、程序自动生成[5]、程序分析[6]、软件开发[7]等领域得到了广泛应用.

编译器在软件开发中发挥了重要的作用.如果编译器本身存在问题,那么它编译生成的可执行文件可能会出现严重的错误.由于编译器测试工作具有重大意义,它已经成为计算机研究中较为热门的研究领域[8].蜕变测试,即使用一对具有某种关系(称为蜕变关系)的测试用例对待测程序进行测试,通过检查测试用例输出结果是否满足相应的关系得出测试结果[9].Tao 等人提出了等价的程序转换策略,由源程序生成与之等价的另一个源程序,利用等价程序对编译器等开源软件进行测试[10].他们利用等价程序转换策略的编译器测试方法能够发现大部分人工植入到 GCC,LLVM 等编译器中的错误,并检测出两个分别属于 GCC-4.4.3 和 UCC-1.6 的错误[10].Le 等人提出了 EMI(equivalence modulo inputs)程序变体方法[8],利用代码覆盖检测工具 gcov[11]获得源程序在某一个特定输入下的执行路径,通过对 LLVM 编译器生成的抽象语法树进行剪枝操作,消除源程序中未执行的"死代码"的方法,生成与源程序语义上等价的程序,并利用它们组成的等价程序对进行编译器测试.他们利用 EMI 方法发现 GCC,LLVM 编译器对一些源程序和相应的 EMI 变体程序编译会生成错误的代码,并导致编译器崩溃或运行结果错误[8],发现的大部分错误已经被其他开发者提交或修复,提交了对 GCC,LLVM 编译器少数开发者版本(非正式发行版本)检测出的错误,并得到了官方开发人员的确认.Tao 等人提出的利用等价的程序转换策略[10]和 Le 等人提出的 EMI 程序变体方法[8]构造测试用例对开源编译器进行测试,本质上都是通过构造等价的蜕变关系,利用蜕变测试对开源编译器进行测试的方法,并且能够发现 GCC,LLVM 等编译器中的错误.部分求值技术本质上是一种程序例化技术.在给定的部分输入下输入程序与部分求值后,得到的剩余程序是等价的.本文利用这种思想生成与输入程序互为等价的剩余程序,并将部分求值技术应用到编译器测试中.现有 C 语言的部分求值器 C-mix[12]和 Tempo[13]在程序例化等领域具有较为广泛的运用,为了便于将部分求值技术应用到编译器测试中,本文设计了 peC 语言,实现了 peC 的部分求值器,设计了基于部分求值技术的编译器测试框架,对 GCC,LLVM 编译器进行检测.

本文的主要贡献如下:(1) 针对 C 语言子集 peC,形式化地描述了其部分求值策略.(2) 基于部分求值策略的形式化描述,利用 LLVM 编译器的前端 Clang,实现了 peC 语言的部分求值器.(3) 设计了基于部分求值技术的编译器测试框架,对编译器进行测试,检测出了大部分之前其他方法发现的 GCC,LLVM 编译器中的错误;此外,还发现了其他方法不能发现的错误.

1 peC 语言的部分求值技术

1.1 部分求值的基本定义

部分求值技术是一种源程序到源程序的转换技术,它重点强调的是程序的完全自动化的转换,转换后的源程序是优化的源程序,所以说它是一种程序优化技术,或者更确切地叫做程序例化技术[1].

为了给出剩余程序和部分求值器的定义,首先对程序功能函数⟦_⟧进行如下描述:

$$output=⟦p⟧[x_0,\ldots,x_{n-1}],$$

其中,*output* 是 p 在输入 $x_0,\ldots,x_{n-1}(n\geq 1)$下的结果.

定义 1. 设源程序 p 和 r,对于 p 来说,若给定其部分输入 $in1=\{d_0,\ldots,d_{n-1}\}$,对任意剩余输入 $in2=\{d_n,\ldots,d_m\}$,有:

$$⟦p⟧[in1,in2]=⟦r⟧[in2],$$

则称 r 为 p 关于 $in1=\{d_0,\ldots,d_{n-1}\}$的剩余程序.

定义 2. 设源程序 p 和 r,若对于 p 的部分输入 $in1=\{d_0,\ldots,d_{n-1}\}$和任意的剩余输入 $in2=\{d_n,\ldots,d_m\}$,有:

$$[\![p]\!][in1,in2]=[\![[\![mix]\!][p,in1]]\!][in2]=[\![r]\!][in2],$$

则称 *mix* 是部分求值器.

图 1 为部分求值器 *mix* 的示意图.例化过程就是执行程序 *p* 仅依赖于部分输入 *in*1 的计算部分,并生成代码 p_{in1},而 p_{in1} 是仅依赖于那些仍未知的输入 *in*2 的计算.

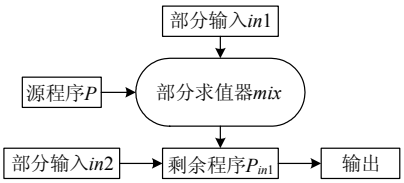

Fig.1 Partial evaluator mix

图 1 部分求值器 mix

1.2 C语言子集peC

现有的 C 语言部分求值器开发时间较早且没有开源,找不到可以使用的版本.由于 C 语言的数据类型和语句结构等非常丰富,为了便于将部分求值技术应用到编译器测试中,本文设计了 C 语言子集 peC 语言.peC 语言保留了过程式语言的基本特征,其文法描述如图 2 所示.

$P\rightarrow B$
$B\rightarrow\{SL\}|S;$
$SL\rightarrow S;|S;SL$
$S\rightarrow SKIP|v=E$ | if (E) B| if (E) B_1 else B_2
| while (E) B | do B while (E) | Dec
$Dec\rightarrow$int v | int $v=E$
$E\rightarrow N$ | v | E_1 ω E_2
$\omega\rightarrow$+ | − | * | / | < | >

Fig.2 Grammar description of peC

图 2 peC 语言的文法描述

1.3 部分求值策略

由于过程式程序中的变量在程序执行的过程中,其值是在不断地动态变化的,采用动态的部分求值技术对过程式语言进行部分求值更为实际和方便[14].我们定义了转换域、预定义函数和转换函数,形式化地描述了 peC 语言的部分求值策略.

1) 转换域

Prog=P

RProg=P

Stm=S

RStm=S

DStm=Dec

RDStm=Dec

StmL=SL

RStmL=SL

Block=B

RBlock=B

Exp=E

RExp=E

$\rho \in State=IDE \to VALUE$

$\sigma \in Stack=State^*$

$VALUE$=INT+BOOL+$\bot$

$\pi \in \Pi=\bot+\top$

其中,*Stack* 是变量状态栈,即由各个程序块的变量状态形成的栈,随着部分求值的过程动态变化.变量的状态包含未知($\bot$)和已知的值.表达式状态包含已知($\top$)和未知($\bot$).

2) 预定义函数

update: $Stack \to Value \to IDE \to Stack$

update[] _ _=*error*

update $\rho:\sigma$ *val* $v=(v \in dom(\rho) \to \rho[val/v]:\sigma, \rho:update\ \sigma\ val\ v)$

comb: $State \to State \to State$

comb[][]=[]

comb $x \to v_1: \rho_1\ x \to v_2: \rho_2=(v_1=v_2 \to (x \to v_1: comb\ \rho_1\ \rho_2), (x \to \bot: comb\ \rho_1\ \rho_2))$

$\oplus$: $Stack \to Stack \to Stack$

$[\cdot]\oplus[\cdot]=[\cdot]$

$\rho_1:\sigma_1 \oplus \rho_2:\sigma_2=comb(\rho_1\ \rho_2):(\sigma_1 \oplus \sigma_2)$

$\dot{+}$: $State \to Stack \to Stack$

$\rho \dot{+} []=[\rho]$

$\rho \dot{+} \rho_1:\sigma=(\rho_1 \wedge \rho):\sigma$

find: $Stack \to IDE \to VALUE$

find[] _=*error*

find ρ: $\sigma\ v=(v \in dom(\rho) \to \rho(v), \rho:find\ \sigma\ v)$

部分求值策略中的转换函数中包含了 5 个辅助操作:*update* 函数用 *val* 更新某个变量 v 在变量状态栈中的状态;*comb* 函数对经过不同例化过程的同一变量对应的变量状态表进行合并,即一旦该变量的状态在两个表中不同,就将该变量的状态置为未知;$\oplus$函数合并程序块经过不同例化过程后产生的变量状态栈;当部分求值分析过程在某一作用域中遇到变量的声明时,$\dot{+}$函数将声明的变量对应的映射添加到变量状态栈中该变量所在作用域对应的变量状态表的表尾,声明的变量的初始状态可以在全局变量状态表中查到;*find* 函数在变量状态栈中从栈顶到栈底查找变量 v 的状态.

3) 转换函数

K_P: $Prog \to [State \to RProg]$

K_B: $Block \to [State \to Stack \to RBlock \times State \times Stack]$

K_{SL}: $StmL \to [State \to Stack \to RStmL \times State \times Stack]$

K_S: $Stm \to [State \to Stack \to RStm \times State \times Stack]$

K_D: $DStm \to [State \to Stack \to RDStm \times State \times Stack]$

K_E: $Exp \to [Stack \to Rexp \times \Pi]$

$K_p[\![B]\!]=\lambda\rho.let(\text{"}rb\text{"}, \rho_1, \sigma_1)=K_B[\![B]\!]\rho[]$ in "*rb*"

$K_B[\![S]\!]=\lambda\rho\sigma.let(\text{"}rs\text{"}, \rho_1, \rho_2:\sigma_1)=K_S[\![S]\!]\rho[]\ \sigma$ in ("*rs*", ρ_1, σ_1)

$K_B[\![\{SL\}]\!]=\lambda\rho\sigma.let(\text{"}rsl\text{"}, \rho_1, \rho_2:\sigma_1)=K_{SL}[\![SL]\!]\rho[]\ \sigma$ in ("*rsl*", ρ_1, σ_1)

部分求值过程从函数 K_P 开始,通过调用 $K_P[\![B]\!]\rho_0$ 计算 peC 程序 B 的剩余程序 *rb*.其中,ρ_0 是全局变量状态,即程序中变量的已知的值或未知($\bot$)状态.函数 K_B 对程序块 B 进行部分求值.当某一程序块开始其部分求值过程时,需要新建一个变量状态表维护当前程序块中的局部变量和其状态的映射关系.某一程序块的部分求值过程结束后,该作用域的局部变量被释放,从变量状态栈中删除该程序块对应的变量状态表.

$K_{SL}[\![S]\!]=\lambda\rho\sigma.let("rs",\rho_1,\sigma_1)=K_S[\![S]\!]\rho[\,]\ \rho\sigma \text{ in } ("rs",\rho_1,\sigma_1)$

$K_{SL}[\![S;SL]\!]=\lambda\rho\sigma.let("rs",\rho_1,\sigma_1)=K_S[\![S]\!]\rho\sigma \text{ in } let("rsl",\rho_2,\sigma_2)=K_{SL}[\![SL]\!]\rho_1\sigma_1 \text{ in } ("rs;rsl",\rho_2,\sigma_2)$

$K_S[\![SKIP]\!]=\lambda\rho\sigma.(\varepsilon,\rho,\sigma)$

$$K_S[\![v=E]\!]=\lambda\rho\sigma.let("re",\pi)=K_E[\![E]\!]\sigma \text{ in}$$
$$\begin{pmatrix}\pi=\top\to let\ val=\varphi(re) \text{ in } ("v=val",\rho,update(\sigma,val,v)),\\ \pi=\bot\to("v=re",\rho,update(\sigma,\bot,v))\end{pmatrix}$$

$$K_S[\![\text{if}(E)\ B]\!]=\lambda\rho\sigma.let("re",\pi)=K_E[\![E]\!]\sigma \text{ in } let("rb",\rho_1,\sigma_1)=K_B[\![B]\!]\rho\sigma \text{ in}$$
$$\begin{pmatrix}\pi=\top\ \&\varphi(re)\to("rb",\rho_1,\sigma_1),\pi=\top\ \&!\varphi(re)\to(\varepsilon,\rho_1,\sigma),\\ \pi=\bot\to("\text{if }(re)\ rb",\rho_1,\sigma\oplus\sigma_1)\end{pmatrix}$$

$$K_S[\![\text{if}(E)\ B_1 \text{ else } B_2]\!]=\lambda\rho\sigma.let("re",\pi)=K_E[\![E]\!]\sigma \text{ in}$$
$$let("rb_1",\rho_1,\sigma_1)=K_B[\![B_1]\!]\rho\sigma \text{ in } let("rb_2",\rho_2,\sigma_2)=K_B[\![B_2]\!]\rho_1\sigma \text{ in}$$
$$\begin{pmatrix}\pi=\top\ \&\varphi(re)\to("rb_1",\rho_2,\sigma_1),\pi=\top\ \&!\varphi(re)\to("rb_2",\rho_2,\sigma_2),\\ \pi=\bot\to("\text{if }(re)\ rb_1 \text{ else } rb_2",\rho_2,\sigma_1\oplus\sigma_2)\end{pmatrix}$$

其中,φ函数将剩余表达式转换成数值,ε表示空串.

$$K_S[\![\text{while }(E)\ B]\!]=\lambda\rho\sigma.let("re",\pi)=K_E[\![E]\!]\sigma \text{ in}$$
$$let("rb",\rho_1,\sigma_1)=K_B[\![B]\!]\rho\sigma \text{ in}$$
$$\begin{pmatrix}\pi=\top\ \&!\varphi(re)\to(\varepsilon,\rho_1,\sigma),\\ \pi=\top\ \&\varphi(re)\to let("rs_1",\rho_2,\sigma_2)=K_S[\![\text{while }(E)\ B]\!]\rho\sigma_1 \text{ in } ("rb;rs_1",\rho_2,\sigma_2),\\ \pi=\bot\to let("rs_2",\rho_3,\sigma_3)=K_S[\![\text{while }(E)\ B]\!]\rho\sigma_1 \text{ in } ("\text{if }(re)\ \{rb;rs_2\}",\rho_3,\sigma_3)\end{pmatrix}$$

$$K_S[\![\text{do } B \text{ while }(E)]\!]=\lambda\rho\sigma.\ let("rb",\rho_1,\sigma_1)=K_B[\![B]\!]\rho\sigma \text{ in}$$
$$let("re",\pi)=K_E[\![E]\!]\sigma_1 \text{ in}$$
$$\begin{pmatrix}\pi=\top\ \&\varphi(re)\to let("rs",\rho_2,\sigma_2)=K_S[\![\text{do } B \text{ while }(E)]\!]\rho\sigma_1 \text{ in } ("rb;rs",\rho_2,\sigma_2),\\ \pi=\top\ \&!\varphi(re)\to("rb",\rho_1,\sigma_1),\\ \pi=\bot\to let("rs_1",\rho_3,\sigma_3)=K_S[\![\text{while }(E)\ B]\!]\rho\sigma_1 \text{ in } ("rb;rs_1",\rho_3,\sigma_3)\end{pmatrix}$$

当循环语句的条件表达式为真时,由于循环体可能被执行多次,而其中的局部变量仅在全局变量状态ρ中定义了 1 次,所以循环体中的局部变量至多会根据ρ被初始化 1 次.

$K_S[\![Dec]\!]=\lambda\rho\sigma.let("rds",\rho_1,\sigma_1)=K_D[\![Dec]\!]\rho\sigma \text{ in } ("rds",\rho_1,\sigma_1)$

$K_D[\![\text{int } v]\!]=\lambda\rho\sigma.let\ val=\rho(v) \text{ in } let\ \sigma_1=[v\to val]\dotplus\sigma \text{ in } ("\text{int } v",\rho/v,\sigma_1)$

$$K_D[\![\text{int } v=E]\!]=\lambda\rho\sigma.let("re",\pi)=K_E[\![E]\!]\sigma \text{ in}$$
$$\begin{pmatrix}\pi=\top\to("\text{int } v=re",\rho/v,[v\to\varphi(re)]\dotplus\sigma),\\ \pi=\bot\to("\text{int } v=re",\rho/v,[v\to\bot]\dotplus\sigma)\end{pmatrix}$$

声明语句声明新的变量,需要从全局变量状态中查找该变量对应的状态进行初始化.若声明语句中没有对变量进行初始化,则函数 $K_D[\![\text{int } v]\!]$从全局变量状态中删除变量 v;若声明语句中对变量进行初始化,则函数$K_D[\![\text{int } v=E]\!]$从全局变量状态中删除变量 v,并在变量状态栈中更新 v 的状态.

$K_E[\![N]\!]=\lambda\sigma.(n,\top)$

$K_E[\![v]\!]=\lambda\sigma.let\ val=find(\sigma,v) \text{ in } (val=\bot\to("v",\bot),("val",\top))$

$$K_E[\![E_1\ \omega\ E_2]\!]=\lambda\sigma.let("re_1",\pi_1)=K_E[\![E_1]\!]\sigma \text{ in}$$
$$let("re_2",\pi_2)=K_E[\![E_2]\!]\sigma \text{ in}$$
$$(\pi_1=\top\ \&\pi_2=\top\to("\varphi(re_1)\ \vec{\omega}\ \varphi(re_2)",\top),("re_1\ \omega\ re_2",\bot))$$

其中,$\vec{\omega}$表示对两个整数进行加、减、乘、除等操作.

2 部分求值器的实现

本文实现的部分求值器利用LLVM的LibTooling库[15]将peC语言源程序通过语法分析转换为抽象语法树(abstract syntax tree,简称AST),基于部分求值策略修改AST,最终生成剩余程序.

图3为本文实现的peC语言部分求值器的基本结构.peC语言部分求值器主要由3个部分构成.

1) partialeval是部分求值器的基本环境.
2) 部分求值过程是基于部分求值策略对peC程序的AST结点进行删除、替换等操作的过程,在这个过程中,需要建立动态环境维护中间信息.动态环境由全局变量状态ρ_0、变量状态栈σ和语句分析栈StmtEnv构成,其中,ρ_0包含了程序p的部分输入$in1$.
3) 部分求值过程结束后,输入程序p,经过剩余程序代码生成阶段,最终得到剩余程序.

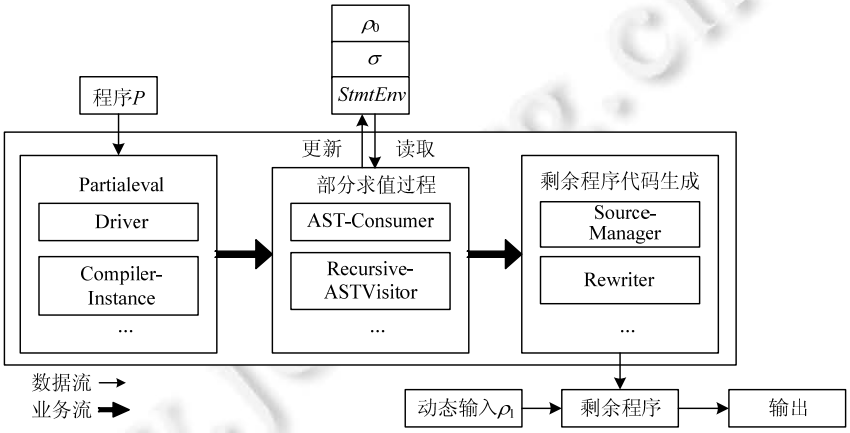

Fig.3 Partial evaluator of peC

图3 peC语言部分求值器

2.1 LLVM基本环境设置

为了实现部分求值过程,在LLVM编译器的前端Clang中添加了命令选项"-partialeval",并添加相应的入口.当使用该选项时,触发部分求值任务,对输入源程序进行基于部分求值策略的程序转换,最终得到剩余程序.

2.2 peC程序的中间表示

LLVM编译器通过语法分析生成peC程序的中间表示AST.通过访问AST的结点,可以得到与源程序相关的信息.

例1:如图4所示的源程序包含了声明语句、if条件语句等,其对应的AST如图5所示.为了便于理解,在图5的AST中保留了AST中的一些特征信息,如操作符类型、变量类型、变量名等.

```
int main(void){
    int a=3;
    if (a<5)
        a=a+1;
    else
        a=a−2;
    return 0
}
```

Fig.4 Simple source code

图4 简单的源程序

其中,DeclStmt是变量声明语句对应的AST结点类型,访问该结点可以获取到声明的变量名、初始化情况等信息;IfStmt是if条件语句对应的AST结点类型,访问该结点可以获取到与条件表达式、then分支、else分

支对应的子树等信息;DeclRefExpr 是函数调用、变量引用等操作对应的 AST 结点类型,变量引用的结点类型中可以获取到变量类型、变量名等信息.

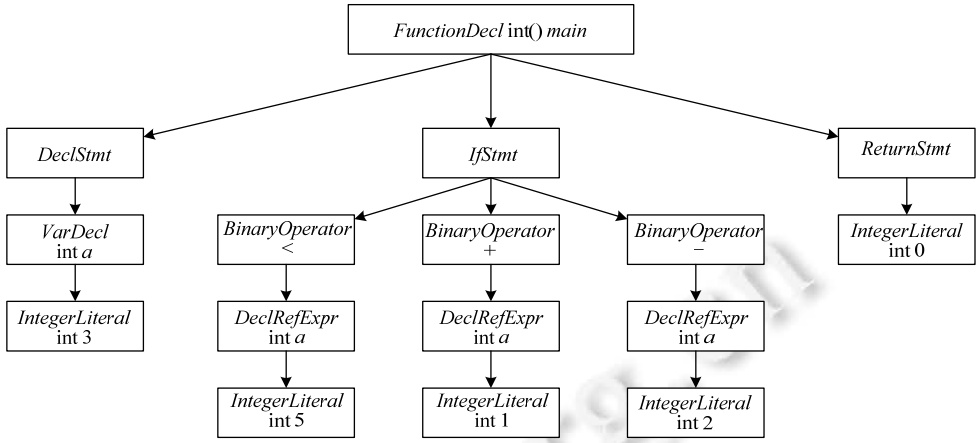

Fig.5 Example of Clang AST

图 5 Clang AST 样例

2.3 动态环境维护

部分求值过程的动态环境是指语句分析栈 StmtEnv、全局变量状态ρ_0和变量状态栈σ.根据部分求值策略,对它们的维护过程如下.

1) 语句分析栈 *StmtEnv* 的每个栈元素是一个语句分析表,保存了一个作用域中的待分析语句.当部分求值过程分析到一个新的程序块 *Block* 时,将一个空的语句分析表压入 *StmtEnv*,并将该程序块中的语句依次保存到表中.在对某条语句部分求值后,将该语句从其对应的语句分析表中删除.
2) 全局变量状态ρ_0 根据程序的部分输入集合收集了输入程序中所有的变量及其初始状态的映射,变量在表中出现的次序和它们在输入程序中出现的顺序是一致的.在部分求值过程中,每当遇到一个变量的声明,从ρ_0中获取它的初始状态,并删除该变量在ρ_0中对应的映射.
3) 变量状态栈 c 在部分求值过程中维护变量及其状态的映射.每当一个新的作用域开始其部分求值过程时,一个空的变量状态表就被压入σ,在该作用域内,按照变量声明顺序将其添加到该作用域对应的变量状态映射表中,并依据ρ_0 初始化其在映射表中的状态.在部分求值过程中,当一个变量的值发生变化时,需要修改其在σ中相应的状态.一个作用域的部分求值过程结束后,其对应的变量状态映射表被删除,但当程序的部分求值过程结束后,变量状态栈σ将保存最终的状态.

例 2:如图 6 所示的输入源程序和部分求值过程中动态环境的变化(根据部分求值策略,部分求值过程对输入源程序中的赋值语句、if 条件语句进行分析).

- 在状态①中,全局变量状态ρ_0按照变量在输入程序中的声明顺序保存其与状态的映射.当 *main* 函数开始进行部分求值过程时,语句分析栈 *StmtEnv* 中保存待分析语句:声明语句 *DeclStmt* 和条件语句 *IfStmt*,并在变量状态栈σ中压入空的变量状态表σ_1.
- 在状态②中,声明语句 *DeclStmt* 结束了其部分求值过程,将 *a*,*b*,*c* 这 3 个变量保存进变量状态表σ_1中,根据全局变量状态ρ_0进行初始化,并从语句分析栈 *StmtEnv* 中删除 *DeclStmt*.
- 在状态③中,条件语句 *IfStmt* 开始部分求值过程.由于 *IfStmt* 对应一个新的程序块 *Block*,在变量状态栈中压入空的变量状态表σ_2.因为条件表达式已知为真,条件语句 *IfStmt* 将执行 then 分支,else 分支不会进行部分求值过程.语句分析栈 *StmtEnv* 收集条件语句中 then 分支的待分析语句.
- 在状态④中,条件表达式中的声明语句结束了其部分求值过程,将变量 *local* 保存进变量状态表σ_2 中,根据全局变量状态ρ_0进行初始化,并从语句分析栈 *StmtEnv* 中删除 *DeclStmt*.

- 在状态⑤中,条件表达式中的赋值语句 *BinaryOperator* 结束了其部分求值过程.由于赋值语句改变了变量 *a* 的状态,需要在变量状态表σ_1中更新它的状态.随着赋值语句的部分求值过程的结束,条件语句 *IfStmt* 结束了其部分求值过程.由于条件语句是语句分析栈 *StmtEnv* 中的最后一条待分析语句,语句分析栈 *StmtEnv* 置为空,变量状态表σ_1中保存最终的变量状态的映射.

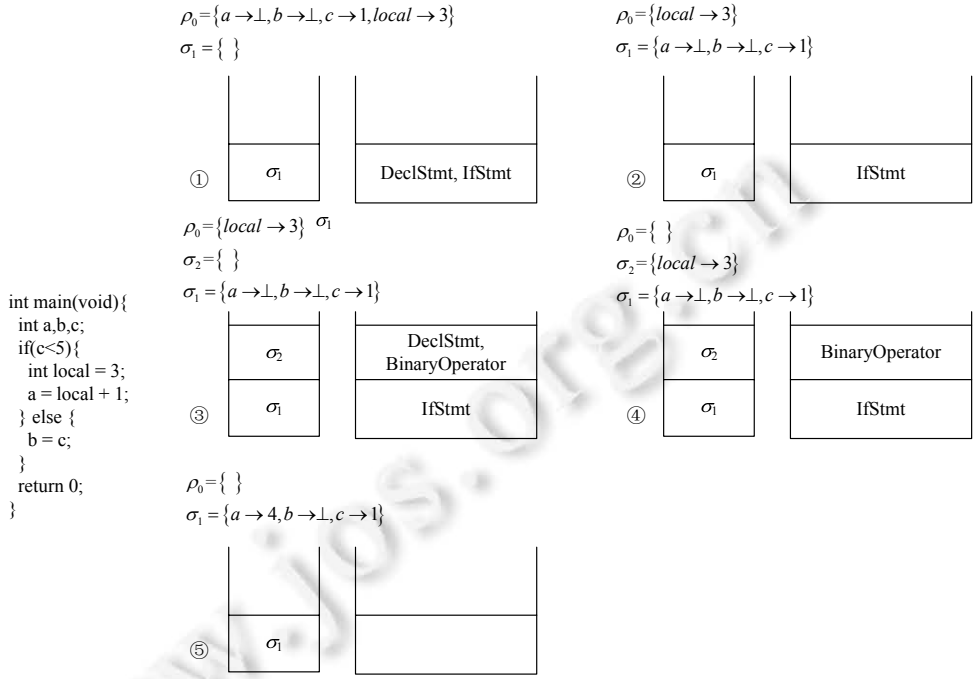

Fig.6 Example of dynamic environment maintenance in partial evaluation

图 6 部分求值中动态环境维护示例

2.4 语义等价变换

本文利用 AST 获取部分求值过程中所需要的信息,peC 语言的部分求值过程其实是修改 AST 和维护动态环境的过程.在修改 AST 的过程中,根据部分求值策略,对 AST 进行了替换、删除和转换操作.规则如下.

1) 表达式规则(图 7①):若表达式 *operand*1 *ω operand*2 的值可以被计算,其在 AST 中的子树会被 *PECodeFrag* 结点替换.其中,*PECodeFrag* 结点是 AST 中新增的自定义结点,保存表达式的值和值类型等信息.
2) 分支规则(图 7②、图 7③):当 if 语句的条件表达式 *E* 的值已知时,采用分支规则对 if 语句的子树进行变换.在图 7②中,当条件表达式为假时,if 语句执行 else 分支,在 AST 中,用 else 分支的子树替换 if 语句的子树;在图 7③中,当条件表达式为真时,if 语句执行 then 分支,在 AST 中,用 then 分支的子树替换 if 语句的子树.
3) 循环规则(图 7④~图 7⑥,图 7 以 while 语句对应的子树转换为例对循环规则具体说明):根据部分求值策略,while 循环语句对应的子树变换规则如下.
   a) 在图 7④中,当条件表达式 *E* 的值为未知时,循环展开一次,其中,*S′*为部分求值后的循环体对应的子树.
   b) 在图 7⑤中,当条件表达式 *E* 的值为真时,循环展开一次,其中,*S′*为部分求值后的循环体对应的子树.
   c) 在图 7⑥中,当表达式 *E* 的值为假时,整个 while 语句不会被执行,在 AST 中,用空结点替换 while 语句对应的子树.

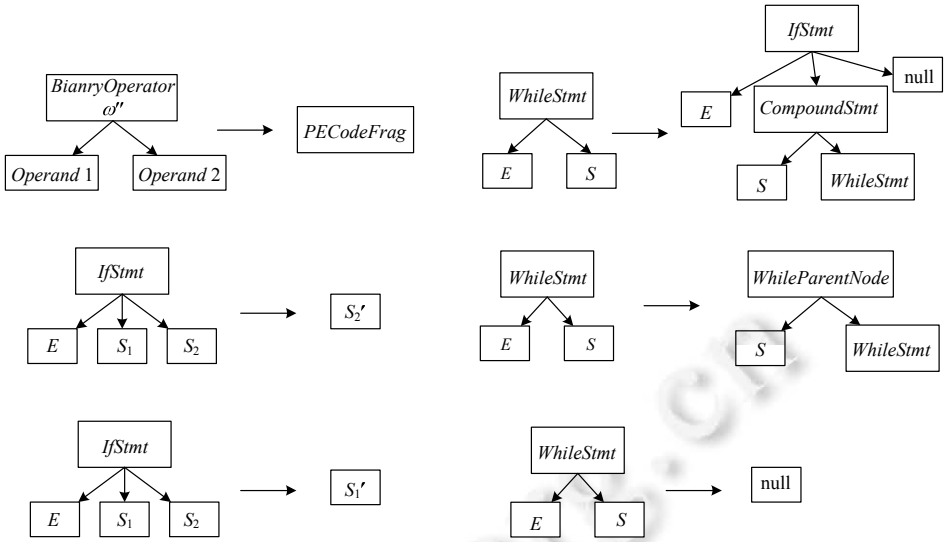

Fig.7 Transformation of AST

图 7 AST 的变换

2.5 剩余程序源代码生成

在本文实现的部分求值器中,部分求值过程是在 AST 上进行的,部分求值后的剩余程序是 AST 的形式,需要将其转换为源程序文本.

输入程序被 peC 语言的部分求值器读入后,其文本会被保存在内存中,利用 AST 中的信息,对内存中的输入程序文本自动地进行修改,最终生成剩余程序的文本.

3 编译器测试框架及实验

3.1 编译器测试框架

设有输入源程序 p,若已知有部分输入 $in1=\{d_0,\ldots,d_{n-1}\}$和剩余输入 $in2=\{d_n,\ldots,d_m\}$,则其剩余程序为 $r=[\![mix]\!][p,in1]$.理论上,在部分输入 $in1$ 确定的情况下,p 与 r 是等价的.用相同的编译器和编译选项对 p 与 r 编译并在剩余输入 $in2$ 下运行,其结果应相同.若出现不同的情况,则一定是该编译器中存在错误.基于以上分析,给出如图 8 所示的编译器测试框架.

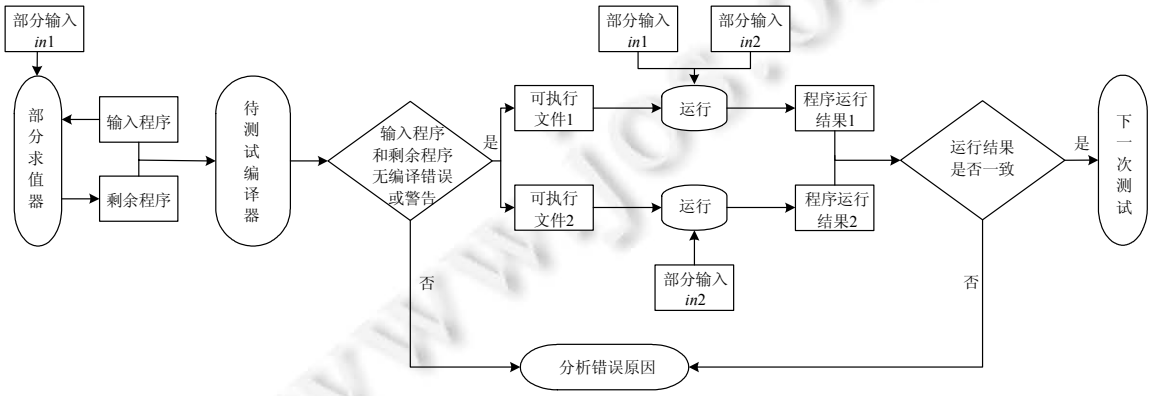

Fig.8 Compiler testing framework

图 8 编译器测试框架

3.2 实验与分析

本文的实验基于 x86-linux 平台,操作系统为 Ubuntu12.04(x86_64).对于 GCC 和 LLVM 编译器,本文使用 -*O*0,-*O*1,-*O*2,-*O*3,-*Os* 等编译选项对它们的若干个发行版本进行了大量的测试.基于图 8 所示的框架,本文做了如下两个方面的实验.

1) 在开源编译器框架 LLVM 中植入大量错误,利用本文的方法能够发现绝大部分错误.如例 3 中源程序 *p* 和剩余程序 *r*,利用相同的编译选项分别对 *p* 和 *r* 编译、运行,运行结果不一致,触发了 LLVM 编译器中人工植入的错误.
2) 利用不同的部分输入和现有的编译器测试用例[10,16,17]对 GCC,LLVM 编译器进行测试.如例 4~例 6 中源程序 *p* 和剩余程序 *r*,利用相同的编译选项分别对 *p* 和 *r* 编译或运行,编译或运行结果不一致,触发了 GCC 或 LLVM 编译器中不同的错误.

例 3:如图 9 输入程序 *p* 和图 10 在部分输入[*b*→1]时的剩余程序 *r*.在版本为 3.4 r182207 的 LLVM 编译器前端 Clang 中,分别使用"-*O*0","-*O*1","-*O*2","-*O*3"和"-*Os*"编译优化选项对程序 *p* 编译并运行,程序输出结果为 –1.同样条件下,对程序 *r* 进行编译并运行,程序输出结果为 1.显然,它们的运行结果是不一致的,说明了该版本的 LLVM 编译器中存在错误.通过实验结果分析可知,该例中的测试用例触发了 LLVM 编译器中人工植入的错误.

```
int a=0;
int main(void){
  int b,c;
  c=a+b;
  return 0;
}
```

Fig.9 Input program 1

图 9 输入程序 1

```
int a=0;
int main(void){
  int b,c;
  c=1;
  return 0;
}
```

Fig.10 Residual program 1

图 10 剩余程序 1

例 4:如图 11 输入程序 *p* 和图 12 在部分输入[*c*→0]时的剩余程序 *r*.在版本为 4.9.0 20131014 的 GCC 编译器中,使用"-*O*3"编译优化选项对程序 *p* 编译并运行,*p* 对应的可执行文件在运行时无法退出.在同样条件下,对 *r* 编译和运行,*r* 对应的可执行文件在运行时立刻退出.显然,它们的运行结果是不一致的.说明了该版本的 GCC 编译器中存在错误.通过实验结果分析可知:使用"-*O*3"编译优化后,优化后的程序 *p* 对应的可执行文件执行第 3 个 while 循环语句,进入无限循环,导致无法退出的情况.

```
int a,b,c,d;
int main(void){
  b=4;
  while (b>-5){
    while (c){
      while (1){
        d=a<2147483647-b;
        if (d)
          break;
      }
    }
    b=b-1;
  }
  return 0;
}
```

Fig.11 Input program 2[16]

图 11 输入程序 2[16]

```
int a,b,c,d;
int main(void){
  b=4;
  b=3;
  b=2;
  b=1;
  b=0;
  b=-1;
  b=-2;
  b=-3;
  b=-4;
  b=-5;
  return 0;
}
```

Fig.12 Residual program 2

图 12 剩余程序 2

例 5:同例 4 中的 *p* 和 *r*.在版本为 4.8 的 GCC 编译器中,使用"-*O*2"编译优化选项对程序 *p* 进行编译时,发生"未定义行为"的警告信息.在同样的条件下对 *r* 进行编译,没有出现警告信息.显然,它们的编译结果是不一致的.这说明该版本的 GCC 编译器中可能存在错误.通过实验结果分析可知,在使用"-*O*2"编译优化选项对程序 *p* 进行

优化时,循环不变式“d=a<2147483647b;”被外提,当 b 为–1 时,该循环不变式触发了“未定义行为”的警告.实际上,在该程序中,由于不可能进入到第 3 层循环,不会出现未定义行为的情况.

例 6:图 13 所示输入程序 p 和图 14 所示在部分输入[a→0]时的剩余程序 r.在版本为 3.4 r182207 的 LLVM 编译器前端 Clang 中,分别使用“-O2”和“-O3”编译优化选项对程序 p 编译,导致 Clang 崩溃.在同样条件下,对程序 r 进行编译,没有出现警告或者错误信息.显然,它们的编译结果是不一致的.这说明该版本的 LLVM 编译器中可能存在错误.值得一提的是,该错误虽然是由开发者向官方提交的错误,但是利用本文的方法能够检测出该错误,且并没有被 EMI 方法等编译器测试方法发现.通过实验结果的分析可知,编译器在对程序 p 进行编译时,在 LLVM IR 中生成了过长的乘法链,导致编译器崩溃.

```
int a;
int main(void){
  int i;
  while (a){
    i=0;
    while (i<32){
      a=a*a;
      i=i+1;
    }
  }
  return 0;
}
```

Fig.13 Input program 3[17]

图 13 输入程序 3[17]

```
int a;
int main(void){
  int i;
  return 0;
}
```

Fig.14 Residual program 3

图 14 剩余程序 3

基于上述实验,利用本文实现的 peC 语言的部分求值器,可以检测出 GCC,LLVM 编译器中关于循环不变式外提、死代码消除等错误,复现了上述两项工作中提及的大量错误.此外还发现了其他方法没有发现的错误,说明了将部分求值技术应用到编译器测试中是有效的.

EMI 方法通过消除“死代码”的方法生成变体程序,本质上是构造了一对具有等价蜕变关系的程序对.EMI 方法在特定输入下执行源程序,再利用代码覆盖检测工具获取每条代码的执行信息,本质上是动态生成变体程序的方法.本文的方法通过对 LLVM 编译器生成的抽象语法树剪枝的方法能够方便地对源代码进行修改,降低了在修改过程中造成变体程序语义错误的可能性.在大部分情况下,消除“死代码”仅仅是对源程序的局部进行修改,生成的变体程序和源程序之间的差别较小.本文提出的基于部分求值技术的编译器测试框架根据不同的部分输入,一个源程序不需要被执行,可以静态地得到多个语义等价的变体程序,组成多对等价测试用例,由于是基于部分求值技术的程序转换策略,得到的变体程序与源程序差异较大,更容易检测出编译器中的问题.本文通过修改 LLVM 编译器生成的抽象语法树的方法对源程序进行部分求值,相对于原先基于源程序文本标记的部分求值方法,获得的语义信息更加丰富,变体程序出现语义错误的可能性更小.

4 总 结

本文形式化描述了 C 语言子集 peC 的部分求值策略,并基于该策略利用 LLVM 编译器实现了 peC 语言的部分求值器.在给定部分输入的情况下,输入程序经该部分求值器转换后生成与之等价的剩余程序.利用该性质,设计了基于部分求值技术的编译器测试框架.实验结果表明,利用这种等价程序对,可以检测出大部分之前其他方法发现的 GCC,LLVM 编译器中的错误;此外还发现了其他方法不能发现的错误,表明将部分求值技术应用到编译器测试中是有效的.

本文将部分求值技术应用到编译器测试中,为编译器测试领域提供了测试用例生成的新思路.由于编译器是十分庞大的软件,由大量的开发者共同维护,使用现有的测试用例生成的等价程序对已经很难测试出新的编译器错误,利用 peC 的部分求值器对其进行检测必然存在一定的局限性.鉴于使用现有的测试用例生成的等价程序对已经很难测试出新的编译器错误这一问题,我们正尝试构造大量新的有效测试用例.由于 peC 语言是 C

语言的一部分且本文实现的部分求值策略并不完整,基于部分求值技术的编译器测试方法还有很大的提升空间,本课题组希望对 peC 语言进行扩展,描述并实现对应的部分求值策略,并在下一步工作中针对该问题进行重点研究,以期望其在编译器测试中产生较好的效果.

References:

[1] Beckman L, Haraldson A, Oskarsson Ö, Sandewall E. A partial evaluator, and its use as a programming tool. Artificial Intelligence, 1976,7(4):319−357. [doi: 10.1016/0004-3702(76)90011-4]

[2] Jones ND. An introduction to partial evaluation. ACM Computing Surveys, 1996,28(3):480−503. [doi: 10.1145/243439.243447]

[3] Adams MD, Farmer A, Magalhães JP. Optimizing SYB is easy! In: Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. 2014. 71−82. [doi: 10.1145/2543728.2543730]

[4] Brady EC, Hammond K. Scrapping your inefficient engine: Using partial evaluation to improve domain-specific language implementation. ACM SIGPLAN Notices, 2010,45(9):297−308. [doi: 10.1145/1932681.1863587]

[5] Birken K. Building code generators for DSLs using a partial evaluator for the Xtend language. In: Tiziana M, ed. Proc. of the Leveraging Applications of Formal Methods, Verification and Validation. New York: Springer-Verlag, 2014. 407−424. [doi: 10.1007/978-3-662-45234-9_29]

[6] Tripp O, Ferrara P, Pistoia M. Hybrid security analysis of Web javascript code via dynamic partial evaluation. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. 2014. 49−59. [doi: 10.1145/2610384.2610385]

[7] Razavi A, Kontogiannis K. Partial evaluation of model transformations. In: Proc. of the 34th Int'l Conf. on Software Engineering. New York: ACM Press, 2012. 562−572. [doi: 10.1109/icse.2012.6227160]

[8] Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2014. 216−226. [doi: 10.1145/2666356.2594334]

[9] Chen TY, Cheung SC, Yiu SM. Metamorphic testing: A new approach for generating next test cases. Technical Report, HKUST-CS98-01, 1998.

[10] Tao Q, Wu W, Zhao C, Shen W. An automatic testing approach for compiler based on metamorphic testing technique. In: Proc. of the 17th Asia Pacific Software Engineering Conf. IEEE, 2010. 270−279. [doi: 10.1109/apsec.2010.39]

[11] GNU compiler collection: gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[12] Glenstrup A, Makholm H, Secher JP. C-Mix-Specialization of C programs. In: Proc. of the Partial Evaluation. 1998. 108−154.

[13] Consel C, Hornof L, Lawall J, Marlet R, Muller G, Noyé J, Thibault S, Volanschi EN, Tempo: Specializing systems applications and beyond. In: Proc. of the ACM Computing Surveys, Symp. on Partial Evaluation. 1998. 19−23. [doi: 10.1145/289121.289140]

[14] Liu L, Zhen HJ, Jin CZ. The partial evaluation technique based on information flow analysis. Ruan Jian Xue Bao/Journal of Software, 1995,6(8):509−513 (in Chinese with English abstract). http://www.jos.org.cn/ch/reader/create_pdf.aspx?file_no=19950810&journal_id=jos

[15] The Clang Team. Clang documentation: LibTooling. http://clang.llvm.org/docs/LibTooling.html

[16] GCC bug. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58731

[17] LLVM bug. https://llvm.org/bugs/show_bug.cgi?id=16067

附中文参考文献:

[14] 刘磊,郑红军,金成植.基于信息流分析的部分求值技术.软件学报,1995,6(8):509−513. http://www.jos.org.cn/ch/reader/create_pdf.aspx?file_no=19950810&journal_id=jos

郭德贵(1972－),男,吉林安图人,博士,副教授,CCF 专业会员,主要研究领域为软件理论与技术,软件形式化方法,高级程序设计语言及实现.

吕帅(1981－),男,博士,副教授,CCF 高级会员,主要研究领域为人工智能,智能规划,自动推理.

王冠成(1993－),男,硕士生,CCF 学生会员,主要研究领域为软件理论与技术,软件形式化方法,高级程序设计语言及实现.

刘磊(1960－),男,教授,博士生导师,CCF 专业会员,主要研究领域为软件理论与技术,软件形式化方法,高级程序设计语言及实现.