可组合的描述符泄露类型检查

李 沁 缪 瑾
(安徽工业大学计算机科学与技术学院 马鞍山 243032)

摘 要 应用程序通过操作系统的系统调用对文件描述符进行操作并管理文件资源。如果应用程序对资源描述符的管理出现错误并发生描述符泄漏,会严重影响系统的可用性。据此,提出了一种检查程序是否会导致描述符泄漏的类型系统,给出了描述符操作方法的语义和类型约束,证明了类型系统的可靠性定理。此外,还初步讨论了该类型系统在并发程序下的扩展。

关键词 描述符泄露,类型检查,软件安全
中图法分类号 TP301 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.10.037

Compositional Type Checking of Descriptor Leaking

LI Qin MIAO Jin
(School of Computer Science and Technology, Anhui University of Science and Technology, Maanshan 243032, China)

Abstract Programs manage files, as a kind of resource, using system calls provided by operating systems to manipulate file descriptors. The availability of the system will be significantly degenerated if programs deal with file descriptors arbitrarily. We proposed a type system to check whether a program leaks some resource (typically, file) descriptors. We defined semantics for descriptor-related operations in sequential programs, and proved that the type system is sound with respect to our semantics. In addition, the extension of this type system with concurrent semantics was discussed.

Keywords Descriptor leaking, Type checking, Software safety

1 引言

信息资源管理的可靠性是信息系统可靠性的重要课题。操作系统通过文件描述符操作并管理文件资源,描述符不仅记录了文件的相关信息,还包括了获取、更改文件的权限。如果文件描述符的管理出现错误,会导致系统资源的浪费,并严重影响系统的可用性,极端情况下会导致系统崩溃。特别地,在Unix/Linux内核中,进程利用整数类型作为被打开文件的唯一标识,其中系统默认的3个描述符(标准输入、标准输出、标准错误)的描述符分别是0,1,2,如果程序继续打开新的文件,则其文件描述符为3,依次类推。实际上,这种管理和访问文件的方式在实践中经常会导致难以察觉的错误。以下段代码为例:

```
int fd1,fd2;
fd1=open("file1",O_RDONLY);
fd2=open("file2",O_RDONLY);
…
fd1=fd2;
…
```

上例是C程序中文件被打开时由于描述符被重新赋值而引起泄露的例子。fd1原本记录的是file1的描述符,其在被fd2赋值后失去了对file1的引用,而被打开的file1却驻留在内存中直至进程结束。如果该进程是服务器进程,则其用于管理file1的内存不能再被其它进程使用。实践中这种泄漏经常会导致服务器内存溢出而崩溃。注意到该代码中每一条的语法都是正确的,却在语义上造成了不可控制的描述符泄露,这表明对描述符的不恰当管理会导致系统资源的泄露与浪费。

描述符作为一种系统管理的资源已经获得了研究者持续的关注,目前对程序中描述符资源使用的正确性研究主要集中在函数式语言和演算语言方面。Albert[1,2]针对Java的字节码程序设计了静态分析工具COSTA,该工具可以给出程序运行时所需的资源上限;Albert[3]使用增量分析的方法进行资源使用量化分析。Jost[4]也对程序资源使用进行静态分析,但是他的分析对象包含了高阶运算的函数式语言。Bartoletti[6]采用了模型检验的方法对lambda演算资源管理的扩展进行资源访问检验,在检验过程中近似构造程序访问资源的历史表达式,同时确保通过检验的程序在运行时不会出现过度使用资源的状态。Kodayashi[5]针对Pi演算中的资源访问管理扩展了资源创建与访问的原语,并提出了基于行为类型论的类型推理方法。Antunes[7]从系统脆弱性的角度对程序的资源使用进行检测和预测,主要考虑分布式交互程序之间交互的脆弱性和程序资源泄漏之间的关系。Calcagno[8]使用形态分析的方法研究如何准确高效地分析可变数据结构;

到稿日期:2014-09-30 返修日期:2014-12-03 本文受国家自然科学基金(61170070),省教育厅科研项目(kj2012Z022),国家科技支撑计划(2012BAK30B049-02),安徽高校省级自然科学研究重大项目(KJ2014ZD05)资助。

李 沁(1976—),男,博士,副教授,主要研究方向为形式化方法,E-mail:linuxos2@163.com;缪 瑾(1989—),男,硕士生,主要研究方向为形式化方法。

Cordes[9]提出一种基于静态回路分析的框架，以对程序语义进行无损近似。Coughlin[10]针对必要的关系更新问题(Imperative Relationship Update Problem)提出了一个模块优化类型分析的通用扩展，其允许在类型检查中临时性地打破，接着重新恢复关系不变式。从广义上说，资源不仅包括文件和数据，还包括指针、链表、矩阵和存储空间等。Pugh[11]研究了针对矩阵依赖分析的新语言，通过引入约束用的原语"gist"将矩阵中的数据流分解为新语言系统中的项，提升了系统检查存储器中矩阵引用潜在的混叠的效率。Khedker[12]基于对操作堆结构的程序的研究，将操作堆结构的程序的时间空间结构抽象成一种有界表示，称为访问图(Access Graphs)，并将它应用于垃圾数据的收集。Hind[13]使用数据流的不敏感分析来处理多个指针表达式引用相同存储位置时引起的混叠。

本文分析了命令式语言中的描述符泄漏方式，设计了可有效避免描述符泄漏的语言 Rema 及其相应的依赖类型系统，同时也初步探讨了其在并发方面的扩展。

本文第 2 节给出了 Rema 语言操作的语法和形式语义，并形式化地规约了描述符的泄露；第 3 节给出了 Rema 的类型系统，证明了 Rema 对于描述符泄露的语义可靠性；第 4 节初步探讨了并发情形下 Rema 的扩展问题，采用并发原语 fork()扩展出 CRema。

2 Rema 语言的语法和语义

本节以 Rema 语言作为例子，它的语法对于文中所提出的方法是充分的。

2.1 语法

Rema(Resource Management)语言的语法如图 1 所示。r 的定义域为资源名称集 $\mathscr{R}$，变量 v 属于可列无穷集合 $\mathscr{V}$，n 代表整数集 $\mathbb{Z}$ 上的一个整数。$\vec{v_n}$(或 $\vec{r_n}$)是一个向量构造函数，用于构造具有 n 个变量(或资源名)的向量。当向量只有一个参数时，将 $\vec{v_1}$ 简写成 v。除了像赋值语句、条件语句、循环语句这样常见的结构之外，还引入了两个用于管理描述符(descriptor)的关键字 new 和 delete。非形式地说，就是给定一个资源名 r，关键词 new 产生一个可以储存在变量中的描述符(本文暂不考虑存储描述符的算术操作，其属于未来的工作)。关键词 delete 将存储在变量 v 中的描述符与该描述符绑定的资源名分开。一般来说，像文件描述符、套接字这样的资源，其名都是显式的；而像堆中动态分配的物理地址，由于其不可以被访问，因此它的资源名是隐式的。这里只考虑可以被访问的资源。

$$
\begin{aligned}
&r \in \mathscr{R}\\
&v \in \mathscr{V}\\
&\vec{x_n} ::= (x_1, \cdots, x_n)\\
&e ::= n \mid v \mid e \pm e\\
&m ::= \mathrm{new}(\vec{v_n}, \vec{r_n}) \mid \mathrm{delete}(\vec{v_n})\\
&c ::= m \mid \mathrm{skip} \mid c_1 ; c_2 \mid v := e\\
&\qquad \mid \mathrm{if}\ e\ \mathrm{then}\ c\ \mathrm{else}\ c\\
&\qquad \mid \mathrm{while}\ e\ \mathrm{do}\ c
\end{aligned}
$$

图 1 Rema 语言的语法

如果要用批处理的方式来构造描述符，只需要利用向量构造子(Vector Constructor) new 产生一个描述符向量，用一组资源名创造出一组描述符：每个 $\vec{r_i}$ 都有对应的描述符储存在 $\vec{v_i}$ 中。这里隐含了这样一个假设：这是构造批处理描述符的唯一途径。因而从描述符管理的角度而言，while 不再是一个必需的结构，这也意味着 new 和 delete 不会出现在循环结构中。这种批处理的方式本质上类似于 Python 这样的脚本语言中的列表解析(函数式语言中也是常见的)。第 3 节中的类型系统正是得益于这种设计。

2.2 语义

通常变量语义的存储函数记为 σ，是一个从变量集合到值集合(可能是数值也可能是描述符)的部分映射(Partial Mapping)。σ_0 将任意变量映射为 $\perp$，代表变量尚未定义。此外还需要一个将描述符集合 $\mathscr{D}$ 部分映射到资源名集合 $\mathscr{R}$ 的函数 δ(函数 δ_0 表示值域为空集的映射)。引用计数器 $\kappa: \mathscr{R} \to \mathbb{N}$ 来记录有多少个与资源名 r 绑定的描述符，κ_0 用于表示对任意 $r \in \mathscr{R}$ 都有 $\kappa(r)=0$ 的初始计数器。这些映射都不完全，为将其完全化(Totalize)，那些没有定义的变量均视为映射到 $\perp$。称由命令、存储、描述符映射和计数器组成的四元组 $\langle c, \sigma, \delta, \kappa\rangle$ 为一个格局(Configuration)，并记为 G。为了简化后面的叙述，将 $\sigma(v)$ 简写为 σv。

Rema 语言的操作语义定义为作用在格局集上的二元关系符 $\rightsquigarrow$，具体的规则由图 2 给出。将 i 步 $\rightsquigarrow$ 运算复合成的关系符简记为 $\rightsquigarrow^i$，而由 $\rightsquigarrow$ 合成的自反、可传递的闭包记为 $\rightsquigarrow^*$。对于一个格局 G，如果不存在另一个格局 G' 满足 $G \rightsquigarrow G'$，那么称 G 处于终止态(State of Termination)。对表达式 e 赋值的语义函数 $[\![\ ]\!]_\sigma$ 按如下方式无副作用地给出：

$$
[\![e]\!]_\sigma \in \begin{cases} \mathscr{D}, & \text{当 } e \text{ 具有变量 } v \text{ 的形式且 } \sigma v \in \mathscr{D}\\ \mathbb{N}, & \text{其它} \end{cases}
$$

$\sigma[v \mapsto n]$ 是一个将 v 变成 n 的 σ 映射，进一步扩展这个符号的使用方式，V 是一个变量集合，$\sigma[V \mapsto n]$ 将所有 V 中元素映射成 n。语义函数 $dp()$ 返回一个不属于 δ 的尚未使用的描述符。变换规则(delete)中的自然数减法记为 $\ominus$，具有规则：$0 \ominus 1 = 1$，$n>0$ 时 $n \ominus 1 = n-1$。这样的规则保证了关闭一个未打开的描述符不会产生任何效果。

$$
\langle \mathrm{skip}, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle -, \sigma, \delta, \kappa\rangle \ (\mathrm{skip})
$$

$$
\langle \mathrm{v} := \mathrm{e}, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle -, \sigma[\mathrm{v} \mapsto [\![\mathrm{e}]\!]_\sigma], \delta, \kappa\rangle \ (\mathrm{assign})
$$

$$
\frac{[\![\mathrm{e}]\!]_\sigma \neq 0 \quad \langle \mathrm{c}_1, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{c}_1', \sigma', \delta', \kappa'\rangle}{\langle \mathrm{if\ e\ then\ c_1\ else\ c_2}, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{c}_1', \sigma', \delta', \kappa'\rangle} \ (\mathrm{if1})
$$

$$
\frac{[\![\mathrm{e}]\!]_\sigma = 0 \quad \langle \mathrm{c}_2, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{c}_2', \sigma', \delta', \kappa'\rangle}{\langle \mathrm{if\ e\ then\ c_1\ else\ c_2}, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{c}_2', \sigma', \delta', \kappa'\rangle} \ (\mathrm{if2})
$$

$$
\frac{[\![\mathrm{e}]\!]_\sigma \neq 0 \quad \langle \mathrm{c}, \sigma, \delta, \kappa\rangle \rightsquigarrow^* \langle -, \sigma', \delta, \kappa\rangle}{\langle \mathrm{while\ e\ do\ c}, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{while\ e\ do\ c}, \sigma', \delta, \kappa\rangle} \ (\mathrm{loop1})
$$

$$
\frac{[\![\mathrm{e}]\!]_\sigma = 0}{\langle \mathrm{while\ e\ do\ c}, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle -, \sigma', \delta, \kappa\rangle} \ (\mathrm{loop2})
$$

$$
\frac{\langle \mathrm{c}_1, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{c}_1', \sigma', \delta', \kappa'\rangle}{\langle \mathrm{c}_1 ; \mathrm{c}_2, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle \mathrm{c}_1' ; \mathrm{c}_2, \sigma', \delta', \kappa'\rangle} \ (\mathrm{comp})
$$

$$
\frac{\forall \mathrm{i} \in [1 \cdots \mathrm{n}].\ \sigma(\mathrm{v_i}) = \perp \wedge \mathrm{d_i} = dp() \wedge \delta \mathrm{d_i} = \perp}{\langle \mathrm{new}(\vec{\mathrm{v_n}}, \vec{\mathrm{r_n}}), \sigma, \delta, \kappa\rangle \rightsquigarrow \langle -, \sigma[\mathrm{v_i} \mapsto \mathrm{d_i}], \delta[\mathrm{d_i} \mapsto \mathrm{r_i}], \kappa[\mathrm{r_i} \mapsto \kappa \mathrm{r_i} + 1]\rangle} \ (\mathrm{new})
$$

$$
\frac{\forall \mathrm{i} \in [1 \cdots \mathrm{n}]}{\langle \mathrm{delete}(\vec{\mathrm{v_n}}), \sigma, \delta, \kappa\rangle \rightsquigarrow \langle -, \sigma[\sigma^{-1}(\sigma \mathrm{v_i}) \mapsto \perp], \delta[\sigma \mathrm{v_i} \mapsto \perp], \kappa[\delta(\sigma \mathrm{v_i}) \mapsto \kappa(\delta(\sigma \mathrm{v_i})) \ominus 1]\rangle} \ (\mathrm{delete})
$$

图 2 Rema 语言的操作语义

总的来说，(assign)规则在保持 δ 和 κ 不变的前提下改变 σ，无论(assign)变换右侧存放的值是什么，都是为了与现在正在使用的命令式语言保持一致。如果将一个已经存放在某变

量中的描述符赋值给另一个变量，赋值结束后两个变量将存放相同的描述符，而与该描述符绑定的资源名相关联的引用计数器则不会发生变化。在(new)变换中，一个新的描述符将被创建，并通过复合映射 σ 和 δ 将 v_i 和 r_i 联系起来，同时 r 的引用计数器自增 1。当存放于 v_i 的描述符被关闭时，(delete)变换令所有原本映射到它的变量重新映射到 $\perp$，该描述符的引用计数器如果不是 0，也必须自减 1。

2.3 带有描述符泄露的格局

本节讨论了何种程序的运行格局可能产生描述符泄露。产生描述符泄露的原因有两种。第一种原因是将一个描述符存放在一个已经存有别的描述符的变量时，如果该描述符绑定的资源名之前没有存放在别的变量中，那么该资源名将无法被访问。在不同的语言上该问题的表现有所不同：对于 Python 而言这并不是问题，因为 Python 解释器会在资源失去关联之后将其返还给内核；而对于 C/C++来说，描述符会被保留，并且可以被应用进程通过内核为资源分配的整型描述符访问，因此下列代码存在安全隐患：

```
fd1=open("file1",O_RDWR);
/* fd1 is 3 */
fd2=open("file1",O_RDWR);
fd1=fd2;
/* now fd1 is 4 */
/* file1 can be accessed via 3 */
…
n=write(3,buf,8);
```

显然，上述示例代码的最后一行中的代码描述符如果运行于实际程序中，将产生无法预测的后果，因为编程人员无法预测在程序运行时描述符 3 究竟与哪个资源名相关联。根据最小特权原则(Principle of Least Privilege)，这种分离变量和描述符的方式是不安全的。

描述符泄露的另一个原因在于，即便通过绑定变量保证描述符是可以获取的，但仍可能存在使用描述符后忘记关闭的情形。Linux/Unix 可以调用_exit()函数来关闭这些描述符，它被封装在 exit()函数中，并默认在描述符被终结时被调用。保持已经没用的描述符继续开放既不必要，也违背了最小特权原则；如果放任不管，并且该程序是一个一经运行就不终止的守护程序时，将引起资源泄露。尤其是存在着已经打开描述符的库函数，且这些函数返回时没有关闭被打开的描述符，一旦守护进程调用这些库函数，资源泄露将成必然。因此，除非是一开始就被设计成永存的特殊描述符(如监听套接字、预分配描述符)，所有打开没有被关闭的描述符都有潜在的被泄露的风险。

现在规定两种格局，它们分别带有确定的或潜在的描述符泄露。

定义 1 一个格局 $G=\langle c,\sigma,\delta,\kappa\rangle$ 称作泄露格局(Leaking Configuration)，当 G 中存在描述符 $d\in dom(\delta)$，不存在变量 $v\in \mathrm{dom}(\sigma)$ 使得 $\sigma v=d\wedge\delta\circ\kappa d\neq 0$。将满足上述条件的 d 叫做悬疑描述符(Suspended Descriptor)。

对于每个命令 c，用如下的方式归纳定义出用于关闭 c 的函数 close(c)，其值域是一个变量集：

$\mathrm{close}(\mathrm{skip})=\emptyset$

$\mathrm{close}(\mathrm{new}(\vec{v_n},\vec{r_n}))=\emptyset$

$\mathrm{close}(v:=e)=\emptyset$

$\mathrm{close}(\mathrm{delete}(\vec{v_n}))=\vec{v_n}$

$\mathrm{close}(\mathrm{while}\ e\ \mathrm{do}\ c)=\mathrm{close}(c)$

$\mathrm{close}(\mathrm{if}\ e\ \mathrm{then}\ c_1\ \mathrm{else}\ c_2)=\mathrm{close}(c_1)\bigcup\mathrm{close}(c_2)$

$\mathrm{close}(c_1;c_2)=\mathrm{close}(c_1)\bigcup\mathrm{close}(c_2)$

需要注意的是，上述规则并不意味着与存在于 close(c)中变量相关连的描述符必须要被关闭，相反，它意味着与 close(c)之外的变量相关联的描述符无法被关闭。

定义 2 一个格局 $G=\langle c,\sigma,\delta,\kappa\rangle$ 叫做一个潜在泄露格局(Potentially Leaking Configuration)，当 G 满足：

1. 对于任何满足 $G\rightsquigarrow^k G'$，G' 都不是一个泄露格局，这里的 $k\leqslant\omega,\omega$ 是最小的无穷序数；

2. $r\in\mathscr{R}$，如果存在 $v\in(\sigma\circ\delta)^{-1}\ r$，那么 $v\notin\mathrm{close}(c)$。

注：条件 1 用于保证执行指令 c 时，后续(可数无穷个)格局中不存在悬疑描述符。条件 2 则表明描述符的生命周期不超过它所在进程的生命周期。描述符泄露的隐患来自于那些有意、无意或恶意设置成永不关闭的描述符。

3 Rema 程序的类型系统

本节提出一个用于制约 Rema 程序管理描述符行为的类型系统。

3.1 类型

资源名的类型是带有$\ominus$运算的自然数。变量的类型是 $\perp$、Int 或依赖类型 $ID(R)$，即未定义、整型或 $\mathscr{R}$ 中资源名的描述符组成的集，这里有 $R\in\mathscr{R}$。当 $R=\{r\}$ 时，也可以将 $ID(R)$ 写作 $ID(r)$。

资源环境 Γ 以推论形式给出变量和资源名的类型：$\Gamma\vdash e:\tau$，这里 $\tau\in\{\perp,\mathrm{Int},ID(R)\}$。整型表达式 e 的类型可以由通常的方式被归纳出。称一个变量 $v\in\mathrm{dom}(\Gamma)$ 是指 σv 是一个描述符，也就是 $\sigma v\in D$；称 $r\in\mathrm{dom}(\Gamma)$ 是指 $\kappa r\neq 0$，也就是 r 至少被一个 $d\in D$ 调用。对于 $\mathrm{dom}(\Gamma_1)$ 中任意的变量 v，由于 v 的类型为描述符类型，若对于另一个环境 Γ_2 有 $\Gamma_1(v)=\Gamma_2(v)$，就称 Γ_1 的描述符一致于 Γ_2，记为 $\Gamma_1\sqsubseteq\Gamma_2$。若同时有 $\Gamma_2\sqsubseteq\Gamma_1$，就称 Γ_1 和 Γ_2 是描述符一致的，并记为 $\Gamma_1=_{ID}\Gamma_2$。对于给定的两个环境 Γ_1 和 Γ_2，记 $\Gamma_1\uplus\Gamma_2$ 为它们联合生成的新环境，该环境中的新格局按下述方式映射 v：

$$\begin{cases}\Gamma_1(v), & v\in\mathrm{dom}(\Gamma_1)\backslash\mathrm{dom}(\Gamma_2)\\ \Gamma_2(v), & v\in\mathrm{dom}(\Gamma_2)\backslash\mathrm{dom}(\Gamma_1)\\ \Gamma_1(v)\bigcup\Gamma_2(v), & v\in\mathrm{dom}(\Gamma_1)\bigcap\mathrm{dom}(\Gamma_2)\end{cases}$$

以下 3 个定义是关于资源环境和格局之间关系的。

定义 3 对于给定的资源环境 Γ，称格局 $G=\langle c,\sigma,\delta,\kappa\rangle$ 关于描述符支持(support)Γ，当且仅当：

1. 对于任意变量 $v\in\mathrm{dom}(\Gamma)$，都存在资源名 r 使得 $v\in(\sigma\circ\delta)^{-1}\ r$；

2. 对于任意资源名 $r\in\mathrm{dom}(\Gamma)$，都有 $\Gamma(r)=\kappa r$。

定义 4 对给定格局 $G=\langle c,\sigma,\delta,\kappa\rangle$，称一个资源环境 Γ 关于描述符在 G 中可解释(interpreted)，当且仅当：对于任意变量 $v\in\mathrm{dom}(\Gamma)$，若 $\Gamma(v)=ID(R)$，则存在资源名 $r\in R$ 使得 $\sigma\circ\delta v=r$ 且有 $\Gamma(r)\leqslant\kappa r$。

定义 5 对于给定的格局 $G=\langle c,\sigma,\delta,\kappa\rangle$，称一个资源环境 Γ 关于描述符忠实(faithful)于 G 是指：对于任意变量 $v\in\mathrm{dom}(\Gamma)$，都存在形如 $\sigma\circ\delta v=r$ 的资源名 $r\in R$ 使得 $\Gamma(v)=ID$

(R)和 $\Gamma(r)=\kappa r$ 同时成立。

对于指令 c 的类型判定是这样一个三元组：

$\vdash\Gamma\{c\}\Gamma'$

其中，Γ 和 Γ'是两个分别称为前置和后置的类型环境。该表达式是说，给定的前置环境 Γ 在执行指令 c(可能是有限步也可能是可数无穷步)后变成后置环境 Γ'。如果一个格局 G 永不终止，那么存在一个足够大的n_0，当 $k\geqslant n_0$ 时都有$G\rightsquigarrow^k G^k$，因而 Γ'在 G^k 中可解释(特别地，一个终止了的格局也可看作一个无限循环)。

需要注意的是，环境对于格局的解释性允许那些永不终止的程序的类型获得 Hoare 逻辑风格的类型。对于一个永不终止的程序来说，后置的类型环境用于说明描述符的长期行为，也就是说，它仅仅记录下了格局变化中一直保持打开状态的描述符，而没有记录那些临时打开继而又会在运行中被关闭的描述符(因而后置的类型环境对于后续的格局来说不是忠实的)。当然这些没有被记录的描述符并不会影响 Γ'在后续格局中的可解释性，例如存在格局 G^{k_1}，当 $k_1\geqslant n_0$ 时 Γ'(因为某些文件被打开)对于 G^{k_1} 不是忠实的，这时显然存在某个 $k_2>k_1$ 使得 Γ'对于 G^{k_2} 又是忠实的(打开的文件又被关闭)，毕竟 Γ'在任何格局中都可以被解释。

图 3 给出了类型规则。(Skip)规则的意思是显而易见的，即保持自己不变。(Assign)规则中变量类型必须是$\perp$或者 Int，以确保该变量储存的描述符不会被覆盖。同理，(New)规则里的变量可以存放一个新的描述符，因此变量类型也必须是$\perp$或者 Int，而计数类型 r 自增 1。(Delete)规则将那些被 delete 函数关闭的变量的类型转变为$\perp$，以便这一变量的重新分配。(If)规则中两个环境的联合体将变量映射为一个通过两种选择路径可能产生的描述符组成的集。而(Loop)规则用于组织任何描述符操作出现在循环体里。

$$\frac{}{\Gamma\{\text{skip}\}\Gamma}(\text{Skip})$$

$$\frac{\Gamma\vdash e:\tau\quad \Gamma\vdash v:\perp\vee\ \Gamma\vdash v:\text{Int}}{\Gamma\{v:=e\}\Gamma[v\mapsto\tau]}(\text{Assign})$$

$$\frac{\Gamma\vdash r:k\quad \Gamma\vdash v:\perp\vee\Gamma\vdash v:\text{Int}}{\Gamma\{\text{new}(v,r)\}\Gamma[v\mapsto \text{ID}(r),r\mapsto k+1]}(\text{New})$$

$$\frac{\Gamma\vdash v:\text{ID}(R)\quad r\in R\quad \Gamma\vdash r:k}{\Gamma\{\text{delete}(v)\}\Gamma[v\mapsto\perp,r\mapsto k\ominus 1]}(\text{Delete})$$

$$\frac{\Gamma\{c_1\}\Gamma_1\quad \Gamma\{c_2\}\Gamma_2\quad \Gamma\vdash e:\text{Int}}{\Gamma\{\text{if e then } c_1 \text{ else } c_2\}\Gamma_1\uplus\Gamma_2}(\text{If})$$

$$\frac{\Gamma\vdash e:\text{Int}\quad \Gamma\{c\}\Gamma'\quad \Gamma=_{\text{ID}}\Gamma'}{\Gamma\{\text{while e do c}\}\Gamma'}(\text{Loop})$$

图 3　Rema 语言的类型规则

3.2　语义可靠性

本节给出 Rema 的类型系统语义可靠性的证明。

定理 1(可靠性)　对于一个指令 c，如果存在两个类型环境 Γ、Γ'满足 $\vdash\Gamma\{c\}\Gamma'$，那么对于一个支持 Γ 的格局 $G=\langle c,\sigma,\delta,\kappa\rangle$，不存在既满足 $G\rightsquigarrow^* G'$同时又是泄露格局的后续格局 G'。

证明：这是在不知道代码 c 的情形下对任何可能的 G'而言的。显然后续的代码可能是无限的 skip 流：skip; skip; skip; skip;…，这时就有 $G\rightsquigarrow^* G$，当这种情况出现时，上述定理仍应成立。因此要证的命题就可以通过拆分成两步证明：

A　G 支持 $\Gamma\rightarrow G$ 不泄露

B　G 不泄露$\rightarrow(G\rightsquigarrow G')\rightarrow G'$不泄露

对A的证明采用反证法，只需证明 G 泄露$\rightarrow G$ 不支持 Γ 即可。即证明若存在这样的 $d\in D$，没有相应的 $v\in V$ 使得$\sigma v=d$，且$\kappa(\delta d)\neq 0$，则 G 不支持 Γ。不防设 d_0 为满足这种条件的一个泄露描述符，$r_0=\delta d_0$，只需证明此时必有 $\Gamma(r_0)<\kappa(r_0)$。现令 $Q=\{d\in D\mid\delta(d)=r_0\}$，$P=\{d\in D\mid\delta(d)=r_0\wedge\exists v\in V:\sigma v=d\}$，显然可得$|Q|=\kappa(r_0)$和$|P|=\Gamma(r_0)$；且有$x\in P\rightarrow x\in Q$，即 $P\subseteq Q$。又因为没有相应的 $v\in V$ 使得 $\sigma v=d_0$，故 d_0 不属于 P，但显然 d_0 属于 Q，故 P 是 Q 的真子集，从而$\Gamma(r_0)<\kappa(r_0)$。这就证明了A。

B的证明采用对 c 的归纳证明。

对 $G=\langle\text{skip},\sigma,\delta,\kappa\rangle\rightsquigarrow G'=\langle-,\sigma,\delta,\kappa\rangle$而言，由于 σ、δ、κ都不发生改变，显然 G'不泄露。

对 $G=\langle\text{if } e \text{ then } c_1 \text{ else } c_2,\sigma,\delta,\kappa\rangle$而言，由图 3 的类型规则(If)可知 e 的类型为 Int，故不会改变 σ、δ、κ。当 $e\neq 0$ 时，$G\rightsquigarrow G'=\langle c_1,\sigma,\delta,\kappa\rangle$，不会泄露；同理当 $e=0$ 时，$G\rightsquigarrow G'=\langle c_2,\sigma,\delta,\kappa\rangle$亦不会发生泄露。上述证明方式同样适用于 $G=\langle\text{while } e \text{ do } c,\sigma,\delta,\kappa\rangle$的情形，由图 3 的类型规则(Loop)可知 e 的类型也为 Int。当 $e\neq 0$ 时，由 2.1 节可知，由于 new 和 delete 不会出现在循环体中，判断句 e 中也不会有赋值，因而 $G\rightsquigarrow G'=\langle\text{while } e \text{ do } c,\sigma',\delta,\kappa\rangle$不泄露；当 $e=0$ 时，$G\rightsquigarrow G'=\langle-,\sigma',\delta,\kappa\rangle$亦不会泄露。对 $G=\langle v:=e,\sigma,\delta,\kappa\rangle\rightsquigarrow G'=\langle-,\sigma[v\mapsto[\![e]\!]_\sigma],\delta,\kappa\rangle$，由类型规则(Assign)知，此时 v 的类型只能是$\perp$或者 Int，即 $\sigma v=\perp$或$\sigma v\in$Int，故 $v\notin\text{dom}(\Gamma)$，因而无论 v 如何改变都不会引起泄露。而$G=\langle\text{new}(\vec{v_n},\vec{r_n}),\sigma,\delta,\kappa\rangle\rightsquigarrow G'=\langle-,\sigma[v\mapsto[\![e]\!]_\sigma],\delta,\kappa\rangle$，$\vdash\Gamma\{\text{new}(\vec{v_n},\vec{r_n})\}\Gamma'$，有 $\delta'(\sigma' v_n)=\delta(\sigma' v_n)=r_n$，$\Gamma(r_n)=\kappa r_n$ 和 $\text{dom}(\Gamma')=\text{dom}(\Gamma)\cup\{v_n,r_n\}$。依图 2 的语义规则(new)有 $\kappa' r_n=\kappa r_n+1$。由于 new 是调用描述符构造函数 dp()来构造一个未被使用的新描述符，不妨设该描述符为 d_n，可知 $\sigma' v_n=d_n$ 和 $\delta d_n=r_n$。故$\Gamma'(r_n)=\Gamma(r_n)+1$，即 $\Gamma'(r_n)=\kappa' r_n$。因此 G'支持 Γ'，由A知 G'不泄露。

对 $G=\langle\text{delete}(\vec{v_n}),\sigma,\delta,\kappa\rangle\rightsquigarrow G'=\langle-,\sigma[\sigma^{-1}(\sigma v_i)\mapsto\perp],\delta',\kappa'\rangle$而言，令 $\sigma v_n=d_n$，则有 $\delta d_n=r_n$，由 G 不泄露可知 $\Gamma(r_n)=\kappa r_n$，依图 2 语义规则(delete)有 $\kappa' r_n=\kappa r_n\ominus 1$，由于 d_n 被删除，Γ'中指向 r_n 的描述符少了一个，故 $\Gamma'(r_n)=\Gamma(r_n)\ominus 1$。因此有 $\Gamma'(r_n)=\kappa' r_n$，G'支持 Γ'，由A可知 G'不泄露。

综上所述，对于任何原子指令 $G\rightsquigarrow G'$都不泄露，因此考虑一般意义上指令 c。$G=\langle c,\sigma,\delta,\kappa\rangle\rightsquigarrow G'=\langle c',\sigma',\delta',\kappa'\rangle$，$G'$都不泄露。因此对于复合指令 $c_1;c_2$ 亦有 $G=\langle c_1;c_2,\sigma,\delta,\kappa\rangle\rightsquigarrow G'=\langle c_1';c_2,\sigma',\delta',\kappa'\rangle$不泄露。而由$\rightsquigarrow^*$定义知它是$\rightsquigarrow$形成的可传递闭包，因此 $G=\langle c_1;c_2,\sigma,\delta,\kappa\rangle\rightsquigarrow^*\bar{G}=\langle-;c_2,\bar{\sigma},\bar{\delta},\bar{\kappa}\rangle=\langle c_2,\bar{\sigma},\bar{\delta},\bar{\kappa}\rangle$也不泄露。

同理可得$\bar{G}=\langle c_2,\bar{\sigma},\bar{\delta},\bar{\kappa}\rangle\rightsquigarrow^*\bar{\bar{G}}=\langle-,\bar{\bar{\sigma}},\bar{\bar{\delta}},\bar{\bar{\kappa}}\rangle$也不泄露，再由$\rightsquigarrow^*$的传递性知 $G=\langle c_1;c_2,\sigma,\delta,\kappa\rangle\rightsquigarrow^* G'=\langle-,\sigma,\delta,\kappa\rangle$对任何 G'都不泄露。

4　CRema：并发的 Rema 语言

4.1　并发语义

本节将用原语 fork(c)来扩展 Rema 语言，fork(c)产生一

个执行指令 c 的新进程，把包含了生成新进程的原语的 Rema 语言称作 CRema(Concurrent Rema)。不过现在只关注如何创建新的进程，至于进程相关的其他原语(比如 join)，本文暂不考虑；同样也不考虑进程间通信，比如管道和域套接字(domain socket)。将这些内容留给后续的工作。

CRema 语言的语义模型是一个格局集。每个进程都有其自己的格局。现为原语 fork(c)专门引入语义规则(fork)：

$$\langle \text{fork}(c_1); c_2, \sigma, \delta, \kappa\rangle \rightsquigarrow \langle c_1, \sigma, \delta, \kappa\rangle \parallel \langle c_2, \sigma, \delta, \kappa\rangle$$

符号“$\parallel$”代表两个格局的并行执行。并行格局执行的进度(schedule)既可以是未定义的，也可以参照某些策略。

注：fork 的语义是源自于 Linux 中的一个系统调用，父进程在产生子进程的过程中通过 fork 将初始化的数据、堆、栈拷贝给子进程。也就是说，产生的每一个进程都是与其他并行的进程完全独立的；每一个进程都有其自己的格局，这一格局的变量、描述符、资源名是从父进程那里拷贝过来的。每个进程的资源计数器都只负责记录自己的，因此，子进程的资源计数 κ 与父进程的相同。

需要强调的是，由于在实际中大多数描述符的管理都局限在本地进程中，因此两格局间彼此独立这一假设并非过度简化。即便是在一些极端情形中，上述假设亦成立，比如同一个描述符可以通过 Unix 域套接字在两个进程之间传递，进程对于描述符仍保持着本地化，因为接收描述符的接收进程收到的是一个内核提供的新描述符，与发送端的进程并无交集。

4.2 CRema 程序的类型检查

CRema 程序的类型检查并不需要额外增加类型规则。依照 fork 的语义，新进程自从被创建起，它的格局就支持它的父进程带给它的类型环境(这里从格局支持环境的角度来说明，而不是环境对于格局可解释的角度，这是因为调用 fork 之前的代码在生成新进程时已经执行完毕)。创建进程时所用的类型环境可以看作前置的类型环境，用以检查子进程的代码执行。因此无论何时遇到 fork(c)语句，都可将语句 c 暂时搁置，对它另做类型检查。对整个代码的检查只需各个独立部分分别通过类型检查即可。

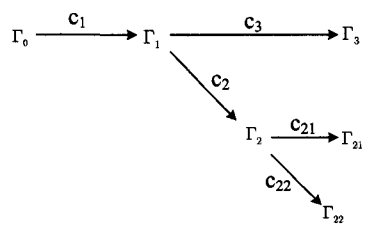

图 4　代码 c 类型检查流程

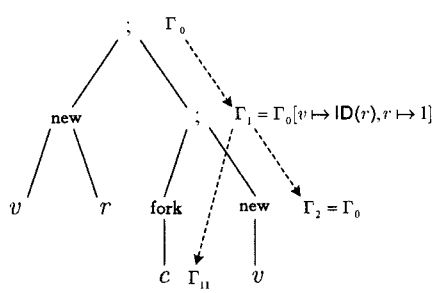

图 5　CRema 类型检查实例

例：如果要在形如 c_1；fork(c_2；fork(c_{22})；c_{21})；c_3 这样的代码中检查代码 c，并假设给定环境为 Γ_0，类型检查的过程可以按图 4 给出的方法。首先得出 $\Gamma_0\{c\}\Gamma_1$，接着以 Γ_1 做环境检查 c_2、c_3，并分别推导出 Γ_2 和 Γ_3。同样的方式作用于 c_{21}、c_{22} 并推出 Γ_{21} 和 Γ_{22}。图 5 的结构阐释了对 CRema 程序进行类型检查的方法：从空环境 Γ_0 出发沿程序的抽象语法树遍历检查每个结点。遇上 fork 结点时，将 fork 中的代码分离出来，以累积的类型环境作为前置类型环境单独做类型检查。

结束语　本文针对命令式语言程序中文件描述符泄漏问题，提出了一种依赖类型系统。在描述符操作的形式化语义上证明了该类型系统的可靠性，并讨论了该方法在并发程序中的应用。未来的工作包括继续扩展现有的 Rema 来统一多种资源管理原语和验证。

参 考 文 献

[1] Albert E, et al. Resource Usage Analysis and Its Application to Resource Certification[M]// Aldini V A, et al., eds. Foundations of Security Analysis and Design. Springer Berlin Heidelberg, 2009, 5705: 258-288

[2] Albert E, et al. On the Inference of Resource Usage Upper and Lower Bounds[J]. ACM Trans. Comput. Logic, 2013, 14(3): 1-35

[3] Albert E, et al. Incremental resource usage analysis[C]// Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation. Philadelphia, Pennsylvania, USA, 2012: 25-34

[4] Jost S, et al. Static determination of quantitative resource usage for higher-order programs[C]// Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Madrid, Spain, 2010

[5] Kobayashi N, et al. Resource Usage Analysis for the π-Calculus [M]// Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, 2006: 298-312

[6] Bartoletti M, et al. Local policies for resource usage analysis[J]. ACM Trans. Program. Lang. Syst., 2009, 31: 1-43

[7] Antunes J, et al. Detection and Prediction of Resource-Exhaustion Vulnerabilities[C]// 19th International Symposium on Software Reliability Engineering, 2008(ISSRE 2008). 2008: 87-96

[8] Calcagno C, et al. Compositional Shape Analysis by Means of Bi-Abduction[C]// POPL'09. 2009: 289-300

[9] Lokuciejewski P, et al. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models[C]// Code Generation and Optimization, 2009 (CGO 2009). 2009: 136-146

[10] Coughlin D, et al. Fissile type analysis: Modular Checking of Almost Everywhere Invariants [J]. SIGPLAN Notices, 2014, 49 (1): 73-85

[11] Pugh W, et al. Constraint-Based Array Dependence Analysis [J]. ACM Transactions on Programming Languages and Systems, 1998, 20(3): 635-678

[12] Khedker U, et al. Heap reference analysis using access graphs [OL]. http://www.cs.ox.ac.uk/seminars/584.html

[13] Hind M, et al. Interprocedural Pointer Alias Analysis[J]. ACM Transactions on Programming Languages and Systems, 1999, 21 (4): 848-894