



[12] 发明专利申请公开说明书

[21] 申请号 02823284.4

[43] 公开日 2005 年 3 月 2 日

[11] 公开号 CN 1589435A

[22] 申请日 2002.9.24 [21] 申请号 02823284.4

[30] 优先权

[32] 2001.9.24 [33] FR [31] 01/12278

[86] 国际申请 PCT/FR2002/003249 2002.9.24

[87] 国际公布 WO2003/027851 法 2003.4.3

[85] 进入国家阶段日期 2004.5.24

[71] 申请人 格姆普拉斯公司

地址 法国热姆诺

[72] 发明人 G·格里莫 D·德维尔

A·雷克特

[74] 专利代理机构 中国专利代理(香港)有限公司

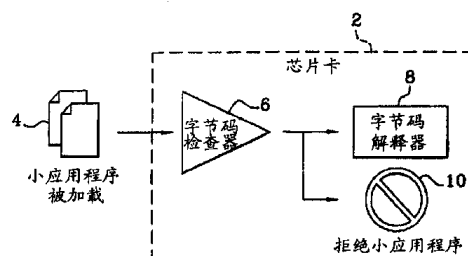
代理人 程天正 张志醒

权利要求书 3 页 说明书 20 页 附图 4 页

[54] 发明名称 用于优化的代码检查器的方法和设备

[57] 摘要

当正在把程序(4)集成到电子设备(2)中时,对程序(4)施加验证过程,所述程序是根据按照亲缘关系所链接的类型而加以组织的,每种类型由各自的代码加以标识。本发明的特征在于,它包括如下的一个阶段,所述阶段包括:对这些类型实施一致化的阶段以便确定这些类型最靠近的共同祖先所述代码包括根据如下形式体系的位模式,所述形式体系表示与所述代码相关联的类型的亲缘,所述确定是通过如下方式实施的:使用所述代码的形式;并且对分配给要一致化的类型的模式进行逻辑组合,产生作为这些类型最靠近的共同祖先的类型的模式。本发明能够在带有存储器资源有限的设备中进行验证,特别是在智能卡 2 中进行验证。



1.一种用于在把程序(4)集成到电子设备(2)中的阶段期间对程序(4)进行检查的方法,所述程序根据按照亲缘关系所链接的类型而加以组织,每种类型由各自的代码加以标识,所述方法包括:对这些
5 类型进行一致化的阶段,目的在于确定这些类型最靠近的共同祖先,特征如下:

所述代码是根据如下形式体系并采用位模式的形式加以描述的,所述形式体系表示与所述代码相关联的类型的亲缘;并且

所述一致化阶段包括:对分配给要一致化的类型的模式进行逻辑
10 组合,产生作为这些类型最靠近的共同祖先的类型的模式。

2.如权利要求1所述的方法,其特征在于,对于具有祖先的任何类型,它的模式包含至少一个位,所述位在值和位置上都与它的直接祖先的模式对应位完全相同。

3.如权利要求1或2所述的方法,其特征在于,一个给定类型的
15 模式包含多个处于预定逻辑状态的位,所述预定逻辑状态取决于所述类型到共同祖先的距离。

4.如权利要求1到3所述的方法,其特征在于,使用通过损耗性逻辑状态转换而经受损耗的存储器(22),以便存储预定要在所述一致化阶段中进行逻辑组合的至少一个模式,或存储在所述一致化阶段
20 中由逻辑组合所产生的至少一个模式。

5.权利要求4所述的方法,其特征在于,模式分配形式体系在模式组中建立逻辑状态的优势,所述优势使得在所述存储器(22)中实现模式的存储所需的损耗性转换的次数最小化。

6.如权利要求4或5所述的方法,其特征在于,所述存储器(22)
25 的存储单元被初始化为如下的逻辑状态,从所述逻辑状态起,所述转换处于与损耗性转换的方向相反的方向上。

7.如权利要求4到6任一项所述的方法,其特征是,所述损耗性转换处于从逻辑0状态到逻辑1状态的方向上,所述逻辑组合包括:在相同等级组合模式的每一对位之间执行与类型的逻辑操作。

8.如权利要求4到6任一项所述的方法,所述损耗性转换处于从
30 逻辑1状态到逻辑0状态的方向上,所述逻辑组合包括:在相同等级组合模式的每一对位之间执行或类型的逻辑操作。

9.如权利要求1到8任一项所述的方法,为处于一组代码段形式的程序而加以实现,每个代码段由一个标记加以标识,所述标记对应于可以从一个分支到达的指令,特征在于,所述检查在预定的试探法中采用给定的段顺序按段来实施。

- 5 10.权利要求9所述的方法,其特征在于,要检查的段的顺序是被检查的段的连续顺序,在被检查的段的后面跟随着作为它的后续者的段。

11.如权利要求9所述的方法,其特征在于,要检查的段的顺序由第一个加标记的段给出。

- 10 12.如权利要求1到11任一项所述的方法,其特征在于,使用易失性存储器(20)作为缓存器,用于为逻辑组合步骤存储候选模式,并且所述候选模式是根据在要检查的程序(4)中的分支点是动态且预测地加以选取的。

- 15 13.如权利要求12所述的方法,其特征在于,选择如下模式,以便存储到易失性存储器(20)中,所述模式对应于那些在给定时间后可能要经受一致化的类型,并且通过参考要检查的程序(4)的检查流程图,对要存储的模式作出选择。

- 20 14.如权利要求12或13中所述的方法,其特征在于,选择如下模式,以便清空所述易失性存储器(20),所述模式对应于那些在给定时间后更可能要经受一致化的类型,并且通过参考要检查的程序的检查流程图,对要清空的模式作出选择。

15.如权利要求13或14所述的方法,其特征在于,所述检查流程图是根据所述先前确定的标记和分支到这些标记的指令而建立的。

- 25 16.如权利要求1到15任一项所述的方法,其特征在于,所述检查是在芯片卡类型的电子设备(2)中实施的。

- 30 17.一种用于在把程序(4)集成到电子设备(2)中的阶段期间对程序进行检查的设备(6),所述程序根据按照亲缘关系所链接的类型加以组织,每种类型由各自的代码加以标识,所述检查设备包括:一致化装置,用于对这些类型进行一致化以便确定这些类型最靠近的共同祖先,特征在于:

所述设备包括用于标识类型的亲缘的装置,对类型的标识是以位

模式为基础进行的，所述位模式根据给定的形式体系构成了它的代码，并且

所述一致化装置包括：对分配给要进行一致化的类型的模式进行逻辑组合并产生作为它们最靠近的共同祖先的类型的模式的装置。

- 5 18.如权利要求 17 所述的设备，能够执行权利要求 1 到 15 任一项所述的方法。

19.一种用于加载并运行根据按照亲缘关系链接的类型而组织的程序的电子设备(2)，每种类型由各自的代码加以标识，特征是：所述设备集成符合权利要求 17 或 18 的检查设备。

- 10 20.如权利要求 19 所述的设备，其特征在于，所述设备采用芯片卡的形式而加以生产。

用于优化的代码检查器的方法和设备

5 本发明涉及对加载到电子设备例如一但不限于一存储器资源有限的设备中的程序代码进行检查，所述存储器资源有限的设备诸如是芯片卡或类似的设备。更准确的说，本发明的目的是对依照目前的高级语言编程技术、根据按照亲缘关系所链接的类型加以组织的程序进行检查，并且本发明以使用类型一致化过程的检查技术为基础，所述类型一致化过程目的在于确定这些类型共同的祖先。

10 这一目的是为了确保持所加载的程序与集成该程序的设备所期望的特异性相一致。为此，要使用一种检查器，该检查器的任务典型地用于防止：

- 使用显式值来产生和修改指针
- 攻击栈(“上溢”或“下溢”)

15 确保：

- 每条指令都有作为输入的、正确数量的参数，并且这些输入参数的类型都是正确的；以及
- 每个对象都已经被初始化过一次，且只被初始化过一次

20 这样就可以在集成程序之前过滤所加载的程序，特别是可以拒绝那些会有意地通过一个恶意的动作或者通过编制一个错误或者硬件故障的程序来破坏设备的正确运行的程序。

程序检查的全部问题会出现在存储器资源有限的设备的上下文环境中，在这一情形下，所述存储器资源有限的设备是带有所谓的开放式体系结构的芯片卡。这种类型的卡就由可能随时加载和修改高级语言所编制的程序。开放式智能卡因此构成了能够自包含式和通信的虚拟机，所述虚拟机能够根据它的持有者所需要的新应用演进。

图 1 说明了在这种上下文环境中的检查活动。芯片卡 2 能够加载一个或多个程序 4，在这一情形下，所述程序是“JAVA 小程序”类型的程序。

30 一旦程序 4 被加载到所述卡中，就必须继续进行集成阶段以使该程序可以被操作。检查就发生在这个阶段，并且用于确定所述卡是

必须集成该程序还是相反去拒绝它。这个任务是以对字节码或伪代码所进行的分析为基础而执行的，字节码在下文中将用一个专用术语“byte code”加以指定。它是由字节码检查器 6 实施的，字节码检查器 6 构成了处于字节码解释器 8 上游的、虚拟机的一个功能元件。

5 在运转中，检查器 6 确定所加载的程序 4 的结构在功能上是完整的以及卡 2 所期望的功能相兼容。如果检查器确认该程序有效，就把它传输到字节码解释器 8 以便执行它。相反，如果检测到一个可能危及该卡的故障，则检查器 6 会要求用于拒绝所加载的程序 4 的过程(由图所表示的符号 10)。

10 一般情况下，这种检查过程的实现实际上对在正在执行检查的设备的存储器资源造成了压力。特别是，检查的迭代性质要求对结果数据进行多次更新，以及因此对在存储单元中所进行逻辑状态转换更新多次。此外，所述检查过程专用的存储器空间对存储器资源有限的设备来说也是值得考虑的。

15 这是因为卡的形式在它的硬件资源方面施加了很显著的限制。该有限的空间被划分成了 6 个功能部件，如图 2 示意性地指示的，它们构成了卡 2 的微模块 12，即：

- 微处理器 14

- 安全单元 16

20 - 固定存储器 18，在(英语术语称谓中的 ROM 类型(只读存储器)的)只读模式下可被访问；

- (英语术语称谓中的 RAM 类型(随机访问存储器)，或者可能是 FlashRAM)的易失性工作存储器 20。

25 - 持久性存储器 22，或者非易失性存储器，不过它允许写和擦除周期。这个存储器通常是以所谓的电可擦除可重写 ROM 存储器技术产生的，英文术语称为 EEPROM(电可擦除可编程只读存储器)；和

- 输入/输出接口 24，

该整体由内部数据总线 26 相连接。

30 由于技术原因，给卡 2 的微模块 12 装备持久性存储器容量 22 要比易失性存储器 20 更好。但是，持久性存储器与易失性存储器相比具有很慢的写时间(其比例约为从 1000 到 10,000)，这对“卡”程

序员是很不利的。

更有甚者，持久性存储器 22 产生涉及到一个可靠性问题，涉及到它的使用期限。这是因为根据所用的技术，在写期间它们的存储单元状态的逻辑转换会导致耗损，而该耗损主要取决于转换方向；

- 5 随后会提及“压力”转换或操作。在持久性存储器中每次压力地写值时，所使用的存储单元出错的概率都会增加。在这一情况下，随后可以被读出的值将与曾被写入的值不一致。例如，制造商对使用 EEPROM 技术的持久性存储器保证有限次数(104 到 106 之间)的写操作。

- 10 对存储器来说，EEPROM 类型的持久性存储器有可用的四种操作：

- 读，为了读取先前存储的值；
- 擦除，包括实现从 0 到 1 的逻辑状态转换，存储单元缺省地处于状态 1；

- 15
 - 写，包括实现从 1 到 0 的逻辑状态转换；
 - 更新，包括实现之后是写的擦除；

擦除操作属于压力类型，并可能引起相关联的存储单元出错。芯片卡 2 和芯片卡与之通信的终端之间的通信模块是标准的。可将卡看作为受终端控制的服务器。在没有任何冲突的情况下发生运行。

- 20 通信属于“半双工”类型（在同一时刻只允许在一个方向上通信）。卡只在终端已经向其提出问题之后，才向其发送回应。

当前的趋势是在越来越多的无所不在的数据处理方面集成卡。为此，已经开发出了所谓的开放式操作系统，例如"JavaCard"、"Multos"、"用于智能卡的 Windows"和"Camille"。

- 25 这些操作系统使用户能够加载应用程序，并因此在每种情况下增强可以由卡形成的用途。另外，它们还使在卡中嵌入的应用程序能够相互通信。

- 关于“JavaCard”系统，它以一种四层的体系结构为基础，如图 3 所示。最高层包含用户应用，称为“板载” 28，它们以卡 2 专用的 Java 库 30 为基础。这些库是 Java API(来自英语"Application Program Interface"，即应用程序编程接口)的一个子集，并构成了
- 30 “JavaCard”系统的第二层。应用程序的代码以字节码表示，字节码

由虚拟机 32 解释。虚拟机 32 运行应用程序并且是以基本操作系统为基础的，所述基本操作系统管理在存储器的写、基本输入/输出、对确保安全的加密程序的调用、以及最后对不同 ISO 标准的支持。最低的两层 32 和 34 被存储在芯片卡 2 的固定存储器 ROM18 中。

- 5 每种方法都具有它自己的可利用的执行栈，该执行栈包含它操纵的变量和参数。虚拟机使用寄存器来存储所谓的局部变量(在方法期间所分配的变量及其参数)。

- 虚拟机 32 中的安全性是一个重要问题，这是因为该虚拟机是对执行移动代码的支持，而该移动代码可以来自各种不确定的源。因而可能使虚拟机 32 执行危害系统或其它部件的可靠性的代码。

考虑到这些困难，当前技术的目的是确保持续存储器资源有限的设备自己本身不必对它要集成的程序执行检查，这个任务在上游被执行或者被省略，依赖于确保程序完整性的认可证。

更特别地说，所提出的技术方案所依据的基础是：

- 15 - 防御性的虚拟机的使用；
 - 在确认代码无害的密码系统的帮助下外部认证实体的使用
 - “字节码”检查算法的使用

- 防御性虚拟机 据说是“多疑的”。在执行期间，它检查每个字节码元素不被判断不当地使用(例如，它检查每个字节码在执行栈上有正确个数的参数，或者还检查每个方法调用是有效的)。这一技术方案的缺点是大幅降低了虚拟机的性能。

使用外部认证实体的技术方案具有很少修改虚拟机的优点；但是，这种技术方案对于部署并大幅削弱虚拟机的开放式外部特性来说是非常昂贵的。

- 25 Java 规范根据在加载字节码期间对其的分析来定义了一种安全模型。这一分析的目的是证明所加载的程序无害；在相反情况的下，所加载的程序被拒绝。芯片卡具有的可用容量，被视为太小以致不能执行这样的分析。这就是为什么有多种技术方案被提出来的原因。

- 30 根据另一种已知的方法，执行外部预处理能够获得额外信息的代码或证据，使得卡能够在加载期间检查代码。其它方法的目的是在存储器有限的设备的资源中集成字节码检查器。

因此，在专利文献 WOA0114958 中就曾提出了执行程序代码的“卡外”修改，这样后者就保持了它的原始意义但变得更易于检查。这样就通过允许在单一阶段中进行检查而简化了检查过程。但是，这样做的缺点是需要对程序代码进行修改并排除了对特定优化技术的使用。

根据另一种特别用于 KVM(来自专用术语 K-虚拟机)虚拟机的方法，检查包括执行预计算并把它的结果加到代码上。随后由字节码检查器利用所加的预计算来允许使用更简单的算法。也就是称之为检查载体代码(术语还称之为“PCC”，由英文“Proof Carrying Code”而来，即证据携带代码)技术的简化版本。

在 Java 语言中，规定了字节码，这样它能够静态地检查程序操作的完整性。检查被分为两部分：

- 结构性检查；和
- 类型检查(类型推理)

类型推理可以被看作是数据流的分析，数据是由虚拟机的指令操纵的存储单元，以字节码表示。它的目的是通过拒绝可能危害它们的完整性的任何代码而保护虚拟机以及基本系统，但也保护用户应用程序。类型推理的主要任务是确保组成程序的每条指令操纵数据(存储器)并在考虑其它处理的规范时触发对它们的执行。

类型推理是分析静态代码的过程，该过程发生在加载应用程序期间。就执行时间和存储器消耗来说它被认为是一种昂贵的算法。这可能大大加重将代码加载到虚拟机 32 中的处理器的负担。因而当硬件约束被认为过大时(例如在一个芯片卡中)，类型推理通常由外部第三方执行。

对于这个方面的更多信息，可以参考下列出版物：Rose E. 和 Rose K.H. "Lightweight bytecode verification"; Workshop on Formal Underpinnings of the Java Paradigm, OOPSLA' 98, 1998; Necula G.C. 和 Lee P, "Proof Carrying Code" Proceedings of the 24thACMSIGPLAN - SIGACTSymposium on Principles of Programming Languages (POPL'97), Paris, January 1997. 在 Colby C., Necula G.C., 和 Lee P. 的文章中, "A proof carrying code architecture for Java" Computer Aided Verification, 2000,和 Necula

G.C. "A scalable architecture for proof carrying code", Fifth International Symposium of 2001, PCC 的发明者描述了如何将这项技术用于 Java 字节码检查和板载检查。

5 已知的解决方法提供了相同级别的安全性,并显著地简化了板载检查器的工作。但是,它们与完全嵌入式的技术方案相比都有些不够灵活,因为任何不带证书的应用程序都将被拒绝,即使它是有效的。PCC 技术方案的主要缺点是:证据必须被加载并存储在卡中。要加载的部件的附加无助于减少应用程序加载器构成的瓶颈。另外,由于大小的原因,证据必须被存储在持久性存储器(EPPROM)22
10 中,并因此而遭受速度和可靠性方面的问题。

由类型推理提供的保证在性质上是多的并且是可变的,最重要的利害关系是虚拟机 32 的安全性。要寻求保证它的相干性和一致性。

按照上面的论述,本发明根据第一个方面提出一种用于在把程序集成到电子设备中的阶段期间对程序进行检查的方法,所述程序根据按照亲缘关系链接的类型加以组织,每种类型都由各自的代码加以标识,该方法包括:对这些类型进行一致化的阶段,目的在于确定这些类型最靠近的共同祖先。
15

特征在于:所述代码是根据如下形式体系(formalism)并以位模式的形式加以描述的,所述形式体系表示与所述代码相关联的类型的亲缘;并且
20

特征还在于:所述一致化阶段包括:对分配给要一致化的类型的模式进行逻辑组合,产生作为这些类型最靠近的共同祖先的类型的模式。

有利地的是,对具有祖先的任何类型来说,它的模式包含至少一个位,所述位在值和位置上与它的直接祖先的模式的位置相同。
25

一个给定位的模式优选地包括多个处于预定逻辑状态的位,所述预定的逻辑状态是所述共同祖先的类型的距离的函数。

在优选实施例中,使用通过损耗性逻辑状态转换而经受损耗的存储器,以便存储预定要在所述一致化阶段中经受逻辑组合的至少一个模式,或存储在所述一致化阶段中由逻辑组合所产生的至少一个模式。
30

这样就能够设想一种模式分配形式体系,该形式体系在所有模式

中建立了逻辑状态的优势，该优势使在存储器中实现模式的存储所需的损耗性转换的次数最小化。

优选地，存储器的存储单元被初始化为一种逻辑状态，从这个状态开始，转换处于与损耗性转换的方向相反的方向上。

- 5 当损耗性转换处于从逻辑 0 状态到逻辑 1 状态的方向上时，逻辑组合优选地包括：在相同等级组合模式的每一对位之间执行与(AND)类型的逻辑操作。

- 当损耗性转换处于从逻辑 1 状态到逻辑 0 状态的方向上时，逻辑组合优选地包括：在相同等级组合模式的每一对位之间执行或(OR)类型的逻辑操作。
- 10

此外，所述方法可以为处于一组代码段形式的程序家加以实现，每个代码段由一个标记加以标识，所述标记对应于可以从一个分支到达的指令，按照给定的试探法以给定的段顺序按段进行检查来继续进行。

- 15 在这一情况下，要检查的段的顺序可以是被检查段的连续顺序，在被检查的段之后跟随着作为它的后续者的段。

要检查的段的顺序也可以由第一加标记的段给出。

- 有利的是，使用易失性存储器作为缓存器，用于在逻辑组合步骤中存储候选模式，并且根据要检查的程序中的分支点动态且预测地选择候选模式。
- 20

然后，可以选择如下模式，以便存储到易失性存储器中，所述模式对应于那些在给定时间后可能要经受一致化的类型，并且通过参考要检查的程序的控制流程图，对要存储的模式作出选择。

- 相反地，可以选择如下模式，以便卸载所述易失性存储器，所述模式对应于那些在给定时间后更可能要经受一致化的类型，并且通过参考要检查的程序的控制流程图，对要从存储器除去的模式作出选择。
- 25

- 控制流程图优选地是根据先前确定的标记和分支到这些标记的指令而建立的。在所设想的实施例，检查方法是用于如下设备的执行介质执行的，所述设备用于执行要经受检查的程序的代码。本发明尤其有利于在芯片卡类型的电子设备中执行检查。
- 30

依照第二方面，本发明涉及一种在把程序集成到电子设备中的阶

段期间对程序进行检查的电子设备，所述程序根据按照亲缘关系所链接的类型而加以组织，每种类型由各自的代码加以标识，所述设备包括：一致化类型装置，用于对这些类型进行一致化以便确定这些类型最靠近的共同祖先，

- 5 特征在于：它包括用于标识类型的亲缘的装置，对类型的标识是以位模式为基础进行的，所述位模式根据给定的形式体系构成了它的代码；

特征还在于：所述一致化装置包括：对分配给要一致化的类型的模式进行逻辑组合并产生作为它们最靠近的共同祖先类型的模式的装置。

10

可以生产这种检查设备以便能够执行依照第一个方面的方法。

依照第三个方面，本发明涉及用于加载并执行按照由亲缘关系所链接的类型所组织的程序，每种类型由各自的代码加以标识，

特征在于：它集成符合第二个方面检查设备。

- 15 在所设想的实施例中，这种设备是以芯片卡的形式生产的。

通过参考附图来阅读随后的、对优选实施例的说明，将会更了解本发明及其优势，优选实施例是以非限制性实例给出的，在附图中：

- 图 1，已描述过，是说明在像芯片卡这样的移动设备中集成代码检查器的原理的图；
- 20 - 图 2，已描述过，是构成开放式体系结构芯片卡的微模块的功能元件的框图；
- 图 3，已描述过，是说明构成虚拟机的各个层的图；
- 图 4 是以表格形式给出的、在结合假设的更改之后典型的和创新的推理跟踪的实例；
- 25 - 图 5 说明依照优选实施例的、非压力的一致化网格编码的实例；
- 图 6 是说明为了通过软件缓存器而加速检查而使用控制流程图的实例；

在描绘本发明的实施例之前，将给出通过所谓“引用”算法在虚拟机 32 中进行字节码检查的原理，在专利文献 USA5 668999 中描述了这种原理。

30

为了便于对引用算法-术语也称为“类型推理算法”的理解，将

通过应用于 Java 语言的实例来对其进行描述。

如前所述, Java 虚拟机是栈式机。它对称为局部变量的工作变量以及对栈执行操作。这个方法的目的的是为程序中的每个点确定每个局部变量以及栈中每个单元的类型。为此, 利用了 Java 字节码被给出类型的优势。

在下文中, 术语“执行上下文”被赋予局部变量/栈对。它包括类型信息。

类型推理以对代码的静态分析为基础。Java 字节码的每条指令都可以被模型化成采用它在栈上或在局部变量中的参数, 并在必要时在栈顶或局部变量中产生结果。对在方法期间分配的 Java 对象和变量来说, 它们的类型是通过用于它的创建的字节码而确定的。

在将要通过以下实例说明的类之间存在顺序关系。

有两个类 "Parent" 和 "Child", 其中 "Child" 是由如下代码链接的 "Parent" 的子类, :

```
15      Child class extends Parent
      {
          public static Parent m()
          {
              return new Child();
20      }
```

方法 "Child m()" 声明返回一个类型 Class "Parent", 或者返回类型元素 "Child". 问题在检查期间被提出: "Child" 与 "Parent" 兼容吗? 回答是肯定的, 因为 "Child" 是 "Parent" 的子类。这个例子说明需要有一个操作来确定一种类型是否与另一种类型兼容。

更简单地, 使用顺序关系来构建一个函数, 该函数使得声明是否可以用一种类型而不使用另一种类型成为可能。

关于 T 的顺序关系的行为

指定为 "T" 的类型是网格 (trellis) 的根; 它仅与它自己相兼容。它是网格中最大的元素。T 类型的变量是不可用的变量。

与 "null" 相关的关系顺序的行为

类型 "null" 是形成包含 Java 类的网格的元素。它是这个部分的最小元素。因此, 它与所有 Java 类相兼容。

合一符和固定点

图 4 中给出了一个实例的类型推理跟踪,该实例说明了类型推理算法的另一特殊性。对程序中的两条路径来说,在栈顶的变量类型上建立不同的假定。第一种技术方案是把其标记为类型 T,因而不可用。这样的限制过大:方法"must()"是在其 Java 代码有效的"Parent"类中定义的方法。这是因为"Child1"和"Child2"都继承了这个方法。

在从块"if-else"的输出上,必须按关系在栈顶变量上放置两个假定。不可能静态地确定这两个中的哪个是正确的。从注释: Parent 是"Child1"和"Child2"的母类开始,可以声明,在从块"if-else"的输出上,栈顶包含对"Parent"类型元素的引用,与通过我们的程序中每条路径上的类型推理所获得的类型相兼容的仅有假定。

这样就实现了对假定的一种连接。这种连接被称为“一致化”。用这种方式定义了一种新关系,包括为两种类型找到“共同亲代”。因为网格向上是有界的,所以可以定义并使用它。它被称为“合一符”并由符号“ $\sqcap$ ”指示。

在涉及网格的数学理论中,这个操作包括确定两个结点的共同亲代。这个结点被称为 LUB(来自英文术语“least-upper-bound”,最小上界)。这个操作的复杂度随网格的深度增加而增加。因而,在存储器有限的设备-例如芯片卡环境中把它放到合适的位置并非不重要。

另外还必须对一段代码重复类型推理过程。这是因为,如果在一段代码段的入口点上引入假定,就必须用新的假定重做。迭代的次数必定是有限的,因为每次一致化有助于把变量类型向 T 退回,在最坏情况下将退回到 T。这是因为所用的变换是单调函数。

下面的 Java 程序说明了对固定点的查找:

```
public static void m (boolean b)
{
    Object o =null;
    while (b)
    {
        o= new Object;
    }
}
```

}

在第一段，局部变量 v1 是"null"类型。第一个类型推理是在与"while"对应的代码段上执行的。在这段代码中，在局部变量 v1 中存储了一个"Object"。接下来，转到"while"的开始，因此在两个假定之间实现一致化。局部变量 v1 在第一段曾是"null"类型，现在是"ref Object"类型，因为"null"与所有 Java 类都相兼容。必须重做与"while"对应的代码段，以查看新的假定是否与这个代码段相兼容。在这个例子中，如果是，就接受该程序。

上述被称为对固定点的查找，因为当实现平衡时，类型推理过程是被连续迭代的，或者更准确地说一致化不再更改类型信息。

引用算法

这一段中的算法被以它最常见的形式进行描述，这种形式不会完全对应于后续所使用的算法，因为它没有任何形式的优化。

对所用的每条指令来说，使用一种表来包含每个局部变量的类型，栈中元素的类型以及栈索引的值，称为 TOS(来自英文“top of stack”，即栈顶)。(在英文术语中，这张表被称为“类型框架(type frame)”)。另外，对每条指令还有一个可用的标志；它说明对应指令的类型指令已经或还没有被上一次实现过的一致化之一修改过。这个标志可以有两个值"changed"(修改过)或"unchanged"(未修改过)。

引用算法包括两个连续的阶段，分别命名为"初始化"和"主循环"，它们的主要步骤总结如下：

初始化

- 初始化每条程序指令的信息。除将被定位在"changed"状态的第一条指令外，把所有指令都定位在"unchanged"状态；

- 在签名里所用的其类型信息中填充，没有类型的变量被初始化为 T，用值 0 初始化它的 TOS；

- 针对其它指令 I；

- 设置 I 的标志为值"unchanged"；

- 初始化 TOS 为-1(这用于标记这条指令还没有被访问过)

主循环

- 只要还有标志在值为"changed"的指令；

- 选择一条被标记为"changed"的指令，这条指令用 I 指示；
- 标记 I 为"unchanged"；
- 在与 I 相关的类型信息上模拟它的执行。如果模拟失败，该方法就被拒绝。"类型框架"由 R 表示；
- 5 - 针对 I 的每个后续者 S；
- 表示 I 的"类型框架"为 O，表示 S 的"类型框架"为 R；
- 如果与 S 相关的 TOS 等于 -1，则 $R = 0$ (并且 I 的 TOS 也被复制到 S 的 TOS 中)；
- 如果否，就对 R 和 O 实行一致化。结果由 $R \cap O$ 指示(如果两个"类型帧"的 TOS 不同，检查失败)；
- 10 - 如果 $R \cap O \neq R$ ，标记 S 为"changed"， $R \cap O$ 变成 S 的新"类型框架"。

只要还有被标记为"changed"的指令，就重复主循环。当到达稳定的位置时就停止，因此使用术语固定点。对执行的模拟(如前所述)

15 包括：存在于验证栈上的参数(它们有足够的数量并且它们的类型是所期望的类型)，并根据所执行的字节码更新"类型框架"。这样就验证了有没有栈发生上溢或下溢，并且类型也是正确的。

现在将处理为在芯片卡的上下文环境中掉换这些验证概念要考虑的方面。

20 主要问题在于卡中固有的处理和存储资源的限制，如上所述。

另外，从字节码来看，必须找到所操纵的每个字段的类型。这个操作是有技巧的。此外，在类型推理期间，它要被执行多次，因而它需要多次通过变量结构大小的转换。一种技术方案是构造有效的内部表示。

25 JavaCard 字节码中固有的类型

JavaCard 语言需要合适的类型，这涉及对上述算法的更改和调整，特别是关于类型和子例程网格的更改和调整。

类型的网格

图 5 中描绘了 JavaCard 类型网格的一个实例。

30 在上面的实例中，已经发现类型推理需要对一致化函数进行多次调用。

实现它的最标准的方式包括：构造有序的结构以便来代表网格。

在微处理器上下文环境中并不能容易地设想到这种技术方案，因为它会需要大量的存储器。在高度受约束的环境中实现的最标准的方式包括：根据要一致化的两种类型级联地使用一系列测试。这种技术方案不是最优的。

- 5 下面将在被加载到设备 - 尤其是芯片卡 2 中的程序的上下文环境下描述这些实施例中所采用的技术方案，这可以通过依照本发明的技术和优化来实现完全的类型推理。

首先描述一致化的加速机制，然后根据软件缓存器的使用来优化固定点的管理。

10 一致化的有效管理

该实施例通过两种方案解决来处理由推理的复杂性所提出的问题：一致化表和可选的、用于易受特定方向的转换而耗损的存储器的非压力一致化编码。

- 一致化表

- 15 实施例所用的第一技术方案包括：预先计算类型一致化规则。随后得到带有两个输入的表(要一致化的两个类型 $t1$ 和 $t2$)，该表提供对问题 $t1 \cap t2$ 等于什么的回答。随后提出涉及可用类型的无穷尽的问题：事实上，可以在芯片卡中加载大量的类。新类的加入随即将导致在一致化表中创建新行或新列，以及导致使用卡中此时已经存在的所有类型对其一致化结果进行计算。

- 20 下面的表 1 中给出了一致化加速表的一部分的例子。(符号 "T"(代表“顶部”)指示所有类型的共同亲代，即网格的顶部)。

表 1

一致化表的一部分的实例

	Top	int	Object	array	byte	short	int	null
Top	Top	Top	Top	Top	Top	Top	Top	Top
int	Top	int	Top	Top	Top	Top	Top	Top
Object	Top	Top	Object	Object	Object	Object	Object	Object
bool	Top	Top	Object	bool	Object	Object	Object	bool
byte	Top	Top	Object	Object	byte	Object	Object	byte
short	Top	Top	Object	Object	Object	short	Object	short
int	Top	Top	Object	Object	Object	Object	int	int
null	Top	Top	Object	bool	byte	short	int	null

为进行一致化使用该表容易去设立。另外，该表被计算一次而且仅计算一次，因而它可被存储在卡 2 的 ROM 存储器 18 中。

通过这种方法，在大多数情况下，一致化被减小到读取一张有两个输入 5 的表。

有效和非压力地一致化编码

在推理期间所操纵的信息是类型。对于每种类型，根据表示它的亲缘的形式体系而建立它的编码。这个编码被用来获得更有效的一致化。

10 为此，使用一种技术来实现对网格的操作到布尔操作的双射。所用的编码使得能够通过对逻辑函数的组合找到网格的两个元素的共同亲代。构成一致化的结果随后提交代码，所述代码根据所建立的形式体系而与两个类型的共同亲代相对应。

实际上，在大多数情况下，由于类型信息的大小，所以其必须存储 15 在持久性存储器 22 中，特别是存储在 EEPROM 类型的持久性存储器中。但是，这种类型的存储器易受到压力。更准确地说，这种存储器容易受损于由逻辑状态在特定方向转换的累积而导致的损耗，所述特定方向在这一情况下是从 0 到 1。相反，在 EEPROM 中位从 1 跌落到 0 的操作是非-压力的(它不会损害对应的存储单元)。

20 从在一致化阶段期间网格中的类型被退回到类型 T 的事实开始，所采用的形式体系的目的是确保这种操作只产生 EEPROM 的位

从 1 到 0 的转换。

图 5 中所描绘的实例说明了这种方法。

图 5 描绘了非-压力一致化网格的一个例子。位于网格顶部的所有类型的共同亲代被指定为顶(即英文中的"top")。

5 该实施例中所用的形式体系适用于尽可能地避免在卡 2 的 EEPROM 存储器 22 中进行压力转换, 所述形式体系为这种共同亲代固定作为代码的模式, 该模式由处在 0 状态的位所形成。形成这个模式的位数是根据网格中共享相同的共同祖先的类型的最大数而选择的。每种类型都用包括相同位数的模式来编码。在该实例中, 6
10 位构成一个模式, 共同亲代 T 具有"000000"的模式。特别的是要确保这个位数要使下列情况成为可能:

- 首先, 为网格中的每种类型分配一个单一的模式;
- 第二, 在处于 1 状态的位数的基础上表示从网格顶部 T 到对应类型的距离, 即它在网格中的深度(按照连接这个类型到共同亲代的祖先的个数)。在该实例中, 这个距离是直接由处于 1 状态的位数指示的, 除了构成网格底部的"Null"类型以外, 为它分配一个单独的模式, 在这一情况下, 在这个实例中是"011111"。

20 应该注意, 对于给定的类型(除共同亲代 T 之外), 它的模式包括至少一个位, 该至少一个位在值和位置上与它的直接祖先的模式都相同。

对于具有不是共同亲代的直接祖先的给定类型来说, 这个亲代的每个处于 1 状态的位都处于这个给定类型的模式中的相同位置。(后者还包括一个附加的处于 1 状态的位, 这个位是它所特有的并且能够区分这个直接共同祖先的其它直接后代)。

25 因而, 当存在具有一个共同祖先的几种类型时, 在两个模式的模式之间的单一逻辑与(AND)操作就能够重新产生直接共同祖先的模式。模式之间的与操作包括: 执行模式的、在相同位置上的每个位之间的与操作。这样, 通过对它们的模式执行与操作就实现类型的一致化。

30 为了说明起见, 在图 5 中确保将分配给类型"bool"和"short"模式的组合相与-即(010001) 与 (010100): 0 与 0 = 0, 1 与 1 = 1, ... 1 与 0 = 0), 产生对应于它们的共同亲代"Object"的模式 010000。

这种编码最重要的属性是它特别适合在 EEPROM 中存储：特别是能够在 EEPROM 中存储类型信息，因为前者与芯片卡的其它可用存储器相比可以较大。当位下落(从 1 到 0 变迁)时，对 EEPROM 的压力是有限的。一致化被简化到逻辑与，把它施加到两个类型上，
5 位只会从 1 变到 0，而不会从 0 变到 1。因而显著地减少了压力并延长了 EEPROM 的使用期限。

这种类型的编码可以适用于任意类型的 EEPROM。这是因为一些类型对于写模式下的压力有相反的特性，即压力转换通过从 1 到 0 的变迁导致。在这种情况下，编码被改变，以把一致化简化到逻辑
10 或。

更准确地说，模式的逻辑状态和为上述与组合所使用那些对称地相反。为了恢复如 5 的实例，接着就给共同亲代 T 分配"111111"，为类型"Int"分配"011111"等等。根据简单的布尔逻辑原理能够很容易地验证：在这种情况下使用逻辑或操作来代替图 5 的环境中的逻辑
15 与操作可以获得相同的结果。

在需要的地方，通过可能引起压力转换的、在 EEPROM 中进行的存储，并通过使用更复杂的布尔函数可以使编码更紧凑。

固定点的有效管理

您将回忆起类型推理是对固定点的搜索。前面所描述的类型编码
20 对减少 EEPROM 上的压力略微有些贡献。该实施例还能够：

- 加速一致化；
- 优化对 RAM 存储器 20 的使用，并
- 对下列两种情况有通用的算法：验证可以在 RAM 中进行，或者要由 EEPROM 的资源时。这些优势首先通过使用试探法来选择要
25 验证的标记(术语也称为"label(标签)")、其次通过组合这个选择和软件缓存器的控制而获得，所述缓存器即 RAM 存储器的一部分用作缓存器以对在不久的将来可能要处理的数据进行快速访问。目的于是包括：在所述缓存器中加载类型的模式，针对所述类型的模式设想了为短期的一致化。这些模式，称为"候选"模式，由称为"试探法"
30 的通用规则所定义的预测技术加以选择。软件缓存器的存储容量是受到限制的，试探法必须不仅能够预测很快就要被组合的模式，相反地好要预测那些更可能被组合的模式以消除它们。这需要动态改

变的管理。此外，试探法被有利地使用，试探发只需要少量的计算而不会加重所用资源的负担。

用于选择要被检查的标记的试探法

- 上述优化的类型推理算法工作在 JavaCard 程序的分支点上。在推理期间，可以有多个标签被标记为将要加以检查。这因此导致了选择它们中的一个并检查对应的代码段。因此，必须有一个选择标准。为了测量并比较可能的试探法，被标示为 r 的比率按如下加以定义：

r = 一致化的数量 / 标记的数量；

- 10 它使得能够测量每个代码段到达固定点所必需的变迁个数。

- "Random" 试探法(随机)

这个试探法可以用作参考。它能够在评估集上获得好的结果。这包括：在标记的类型之间随机地选择要检查的下一个标记。

- "Next" 试探法(下一个)

- 15 "Next" 试探法相对简单。它包括：选择这样的标记，该标记作为刚刚已经被检查过的标记的后续者。这样，就得到了线性检查遍历的必需数量。

- "Lazy(懒惰)" 试探法(搜索简易性)

- 20 "懒惰" 试探法按照功效和简易性是最佳的。它包括：先取一个要检查的标记首先加上标记。通常，JavaCard 程序产生于通过 Java 开发包(英语术语 JDK，代表 "Java Development Kit")所编译的 Java 代码的转换。但是，这产生了相当线性的代码。因而程序的停止点通常能在字节码结束时清楚地找到。因此，通过系统地用作为要检查的第一个标记，这一发现就被拓展了。这样，就计算出了程序的开始以便在后面就不再返回到这里。"懒惰" 试探法为上面定义的比率 r 能够获得值 1.3。

更细的试探法可以被加以施用，尤其是使用所检查的程序的检查流程图。该图尤其可用于控制软件缓存器。

最佳缓存器

- 30 明智地使用软件缓存器还能够获得加速度因子，在检查相对较大型的程序(例如小应用程序)期间尤其如此，因为它们略微减少了在 EEPROM 中进行写的次数。它们还通过在 EEPROM 中进行任何写

来显著地加速对小程序的检查。

为了进行类型推理，必须检查程序的消失点。在这样做的过程中，构建程序的检查流程图是容易和“自由”的。某些方法要求 4KB 以上的存储空间以便对其进行检查，现有技术中的高级芯片卡已经
5 有可用的 8KB 的 RAM。因而，可以设想使用这类方法来检查应用程序。

在所述实现中，在 EEPROM 和 RAM 之间建立软件缓存器，以便最好地操纵类型框架。这涉及到在检查期间从缓存器中选择要删除(消除)的数据的阶段，并因此而涉及到缓存器管理策略。

10 为此，可以使用常规策略 - 例如 LRU 类型，所述 LRU 类型适用消除最靠近最小使用的数据，英文是 "Lease Recently Used")。

但是，可以有利地拓展如下事实：因为可以使用检查流程图，所以就易于确定在缓存器中存在的、哪些是相关的数据以及哪些是因此而必须被保留的数据。知道了为要检查的标记所选择的试探法的
15 类型，就能够很容易地根据给定的机制估计缓存器中数据的相干性。举例来说，可以参考 Chi-Keung Luk 和 Todd C.Mowry 的文章 "Compiler-based prefetching for recursive data structures", ACM SIGPLAN Notices 31(9):222-333,1996 中所描述的机制。

将通过下面的例子展示为确定缓存器管理机制而对检查流程图
20 的使用。检查流程图是依照参考图 6 而说明的实例提供的。

表 II 指示了针对这个例子的类型推理的可能的执行跟踪。

表 II

针对图 6 的例子的类型推理的可能的执行跟踪

- 1.把当前上下文和程序开始的当前上下文一起加载
- 25 2.使用来标签 1 一致化
- 3.使用来标签 2 一致化
- 4.选择标签 1
- 5.使用标签 2 来一致化
- 6.选择标签 2
- 30 7.使用标签 3 来一致化
- 8.使用标签 2 来一致化
- 9.选择标签 2

10.使用标签 3 来一致化

11.标签 2 来一致化

12.选择标签 3

13.检查结束

5 采用这样的假定："懒惰"试探法被使用来选择下一个要检查的标记(标签),并且作出尝试来在 RAM20(即,在缓存器中)中实施一致化。对于最初为空并有两个单元的缓存器的情况来说,图 6 中给出的检查跟踪按如下产生:

10 - 第一步是:通过与程序开始 42 相关联的类型信息来加载当前描述符;

- 检查第一个代码段,并到达意图在于标记"标签 1"46 的条件分支 44。因此,有必要使用当前的类型框架来一致化标记"标签 1"的类型"框架"。因此,标记"标签 1"的类型信息将被加载到缓存器中;

15 - 然后,在遇到第二个分支 48 时必须用标记"标签 2"50 来一致化 48。因为具有可用的缓存器单元,所以可将标记"标签 2"的信息加载到其中;

- 根据对"懒惰"试探法的使用而选择标记"标签 1",以便检查对应的代码段;

20 - 必须用标记"标签 2"来进行一致化;但是,后者处在缓存器中。一致化因此就发生在 RAM 中;

- 选择下一个标记,即"标签 2";

25 - 到达条件跳转 52,在它的终点有标记"标签 3"54。在该缓存器中不存在任何对应的类型信息。后者是满的(它包含用于标记"标签 1"和"标签 2"的类型信息)。因此,必须从这些标记中选择一个标记,以便释放缓存器中的一个单元并在其中加载用于标记"标签 3"的类型信息。参考检查流程图,可以看出:只有标记"标签 2"和"标签 3"才可以从当前代码段中访问到(在该图的更深一级可以看到:从标记"标签 2",也可以访问到标记"标签 2"和"标签 3";从标记"标签 3",却访问不到任何标记)。因而,能够为缓存器消除标记"标签 1"并在
30 EEPROM 中更新它的副本。

- 检查毫无障碍地继续着,直到程序 56 结束为止。

如果不使用缓存器,则在这个例子中可能有必要在 EEPROM 中去执行七次一致化(即,在 EEPROM 中进行七次写)。如果使用缓存器,则在 EEPROM 中标记"标签 1"的类型信息就被更新一次且仅被更新一次。另外,这一更新不是必须的,因为程序中的任何路径在验证中的这个时刻都不再能够访问到标记"标签 1"。它的类型信息因而也再是必需的。

这里给出了为了最好地控制缓存器而使用待检查程序的检查流程图的各种可能性的说明。在这样做的过程中,近似达到了最优缓存器(即,在缓存器中在正确的时间有正确的信息)。

10 因为在讨论中的这一例子中,构造检查流程图是廉价的,所以使用它就不仅可以控制软件缓存器,还可以改善用于要检查的标记的选择试探法。

15 优选的实施例能够获取简单的检查系统,这种简单的检查系统展开了使用检查器的可能性,所述检查器被完全加载到微处理器卡或其它存储器有限的设备中。

对本领域的技术人员来说显而易见的是:本发明就逻辑、硬件和环境而言,根据它的使用条件和标准可以有多种变体和等价物。

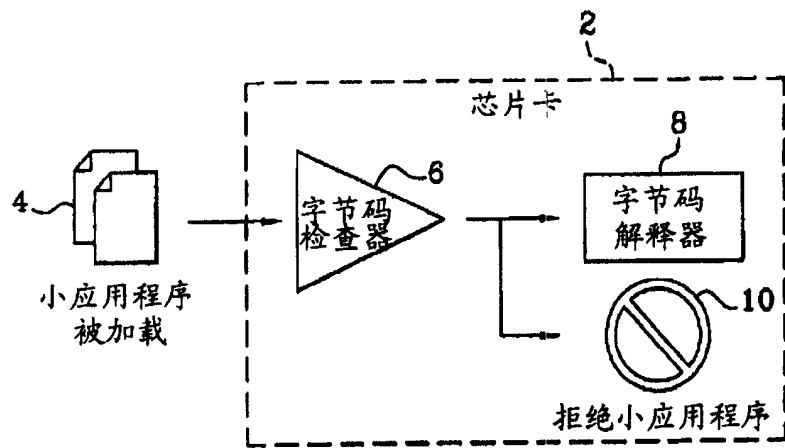


图 1

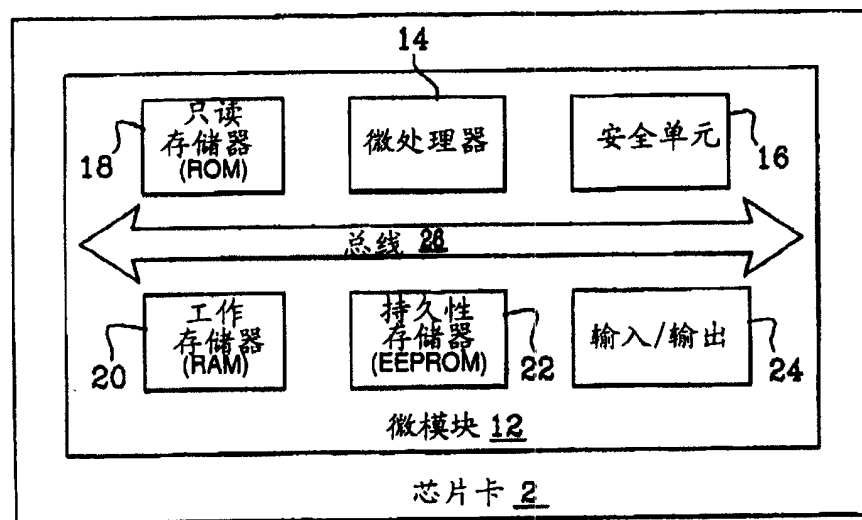


图 2

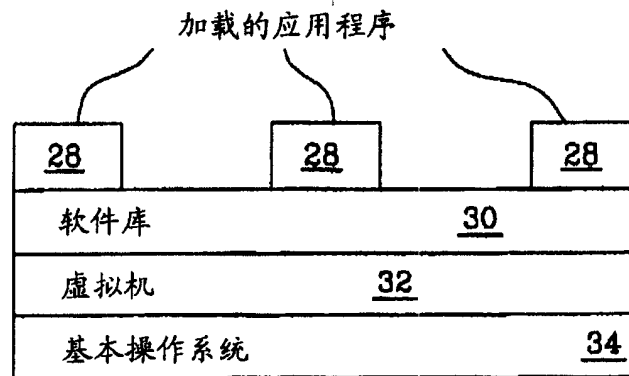


图 3

指令	v0	v1	s0	s1
iload_1	ref Parent	int		
ifeq L1	ref Parent	int	int	
new #1 <Class Filsl>	ref Parent	int		
dup	ref Parent	int	ref Child1	
invokespecial #2 <Method Filsl()>	ref Parent	int	ref Child1	ref Child1
goto L2	ref Parent	int	ref Child1	
L1: new #3 <Class Filsl2>	ref Parent	int		
dup	ref Parent	int	ref Child2	
invokespecial #4 <Method Filsl2()>	ref Parent	int	ref Child2	ref Child2
nop	ref Parent	int	ref Child2	
L2: invokevirtual #5 <Method void must()>	ref Parent	int	ref Parent	
return	ref Parent	int		

图 4

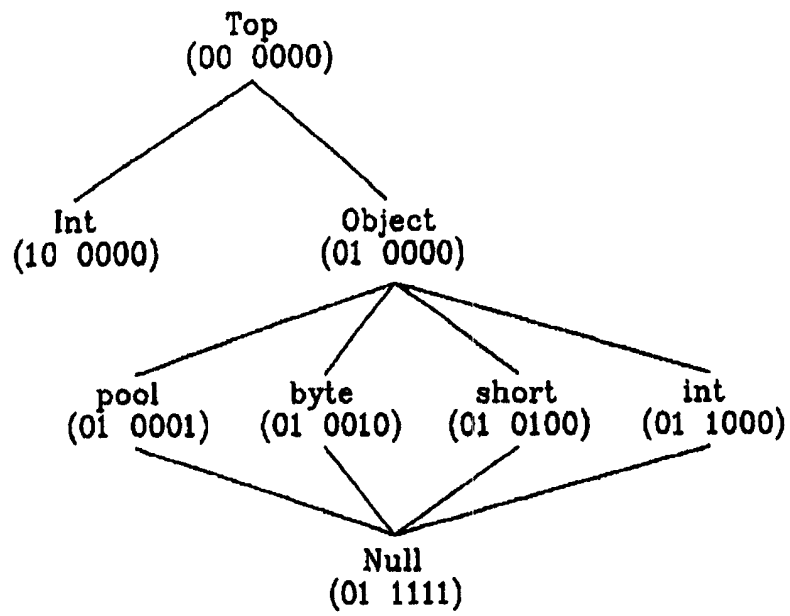


图 5

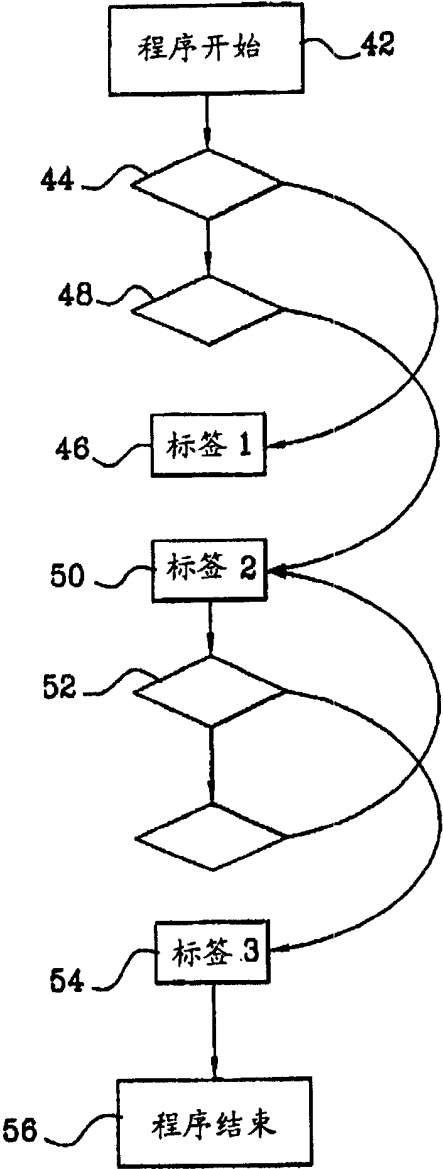


图 6