类型系统与程序正确性问题*)

丁志义[1,2] 宋国新[1] 邵志清[1]

(华东理工大学计算机科学与工程系 上海 200237)[1] (宁夏大学数学计算机学院 银川 750021)[2]

摘 要 类型系统能检出合法程序的语义错误,可以缩短调试时间,在执行程序之前捕获代码中的错误。类型系统的理论基础是类型化的λ演算。带子类型的高阶类型系统$F_{\leq}$已成为类型化语言的演算核心[5]。类型系统和直觉主义极小逻辑是同构的。证明系统的能力取决于类型系统,因而类型系统可以表达程序的性质,并自动进行验证。

关键词 类型系统,程序验证,λ演算,证明理论

Type Systems and the Correctness of Program

DING Zhi-Yi[1,2] SONG Guo-Xin[1] SHAO Zhi-Qing[1]

(Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237)[1]

(Institute of Mathematics and Computer Science, Ningxia University, Yinchuan 750021)[2]

Abstract When type systems detect legitimate program errors, they help to reduce the time spent debugging. Type systems catch errors in code that is not executed by the programmer. Proof generation capabilities of proof construction systems are based on type theory. The base of the theory is the typed λ-calculus. Higher-order type system of higher-order subtyping, known as $F_{\leq}$, has been used as a core calculus for typed languages. The Curry-Howard isomorphism is a correspondence between type systems and intuitionistic logic. Proof generation capabilities of proof construction systems are based on type theory. Type systems allow us to express program properties that are automatically verified.

Keywords Type system, Program verification, λ-calculus, Proof theory

1 引言

"类型良好的程序不会出错",这是由 Robin Milner 提出的一个著名思想。它从本质上断言了程序设计语言的类型系统的可靠性,问题是类型系统是否能让我们写出语义正确、免除错误的程序。类型理论是一个基础的形式理论,按照 Curry-Howard 同构原理,直觉主义逻辑可以解释于其中。正是由于这种构造特性,类型理论可以看作是一个函数式程序设计语言。这样,程序正确性的验证也就转变为类型检查。类型理论已得到了不同的实现,其中较著名的几个类型系统[2]有:

- ALF 基于扩展的马丁洛夫类型理论;
- Coq 基于构造演算;
- Isabelle/HoL 由简单的类型化 λ 演算发展而来,扩充了项的谓词演算;
- LEGO 基于构造演算和依赖类型理论,实现了逻辑框架;
- NuPrl 另一个马丁洛夫类型理论的实现;
- PVS 对简单类型理论的一个扩展,有依赖类型的高阶逻辑;
- Sparkle 函数式语言的一个定理证明器。

所有这些系统都可以用于软件和硬件的验证,也可以作为数学证明的形式化工具,以及作为理论研究的工具。

类型理论是用来作为构造程序的理论的。用该理论可以证明程序员编写的程序是否满足规范,这就是程序验证。程序验证要解决的问题是一个元组$\{Pre\}P\{Post\}$是否一致,可以形式地定义为

$$\forall s(Pre \rightarrow wp(P, Post))$$

在类型系统中可以表示程序的性质,并能自动地进行验证。另一个更为复杂的方法是从规范中推导出正确的程序,即为程序推导。类型系统同时支持以上两种方法,从而程序员可以沟通规范和程序之间的间隙。

在这一过程中,有两个不同的阶段和两种不同的语言:

- 规范过程——规范语言
- 程序设计过程——程序设计语言

规范被表示为一个集合,即所有满足规范的程序的集合。而一个程序则是程序员构造的该集合中的一个元素,也称为居元(inhabitation)。程序用函数式语言表示,类型系统用于证明程序的正确性。作为这些方法的一个结果,类型系统可以同时作为程序设计语言、规范语言和程序逻辑。

2 语义与类型

类型是不经执行程序就可以建立的程序性质[4]。习惯上,人们用类型一词表示对值集合的抽象。一个类型标明了计算表达式后应该得到的是哪一类值,即类型描述了一个程序所有的执行都能成立的不变式。

类型系统本质上是一个类型规则的集合,这些规则描述了如何确定一个表达式的类型。对每一种语法结构,至少应

*)本文工作得到国家自然科学基金和中科院计算机科学重点实验室资助(编号:60373075,SYSKF0305)。丁志义 副教授、博士研究生,主要研究领域为软件形式化方法;宋国新 教授、博士生导师,主要研究领域为软件形式化方法;邵志清 教授、博士生导师,主要研究领域为软件开发与验证方法。

有一条规则可以应用。这样，给定一个程序后，总会有一条规则可以应用到其子项上。通常，类型规则是递归的，一个表达式的类型是由它的各个子部分的类型推导得出的。以Scheme语言为例，假定我们只有两个类型number和function，即：

n : number

$\{\text{fun}\{i\}b\}$: function

对于标识符，需要一个确定其类型的环境(把标识符映射到类型的一个映射)，习惯上用 Γ 表示类型环境。所有的类型推导都有以下形式：

$\Gamma \vdash e : t$

其中，e 是表达式，t 是类型。即 Γ 证明了 e 具有类型 t。具体的规则如：

$\Gamma \vdash$ n : number

$\Gamma \vdash \{fun\{i\}b\}$: function

$\Gamma \vdash i : \Gamma(i)$

最后一个规则表明标识符 i 的类型是环境 Γ 中限定的任何类型。

以加法函数为例，它的类型规则是

$$\frac{\Gamma \vdash l : \text{number} \quad \Gamma \vdash r : \text{number}}{\Gamma \vdash \{+\, l\, r\} : \text{number}}$$

显然，此规则表达了操作语义的概念。之所以是语义，是因为它类似于给程序赋予了含义一样，使得类型的语法项在某一具体的环境中得到了解释；之所以是操作语义，是因为演算是以一个机械的模式进行的。

3 类型系统

设 $T \equiv (S, I, R)$ 是一个类型系统，其中 S 是表达式和类型的语法，I 是推导，R 是规则的集合，则类型系统是适合作为一般形式证明系统的框架的。较为知名的且已有大量研究的类型系统是：

- F_1 简单的类型化 λ 演算
- F_2 二阶(多型) 类型化 λ 演算
- $F_{\leq}$ 有子类型的类型化 λ 演算

在此基础上，还有各种为不同目的而扩展的类型系统，例如存在类型被引入后，可以研究抽象数据类型；递归类型被引入后便于描述递归数据结构；依赖类型也被作为依赖于表达式的类型引入，这种类型是构造演算(CC)和马丁洛夫类型理论的基本元素[8]。

3.1 类型系统 F_1

F_1 基于简单类型化 λ 演算，其语法 S 用 BNF 定义如下：

$type ::= \langle basetype\rangle \mid (\langle type\rangle \rightarrow \langle type\rangle)$

$\langle \lambda\_exp\rangle ::= \langle varible\rangle \mid (\lambda \langle varible\rangle : \langle rype\rangle . \langle \lambda\_exp\rangle)$
$\mid (\langle \lambda\_exp\rangle \langle \lambda\_exp\rangle)$

证明 I 的形式为：

$\Gamma \vdash wf$　　Γ 是一个合式环境

$\Gamma \vdash A$　　A 为 Γ 中的合式类型

$\Gamma \vdash E : A$　　E 在 Γ 中有类型 A

规则 R 为：

$$\frac{}{\varnothing \vdash wf} \quad [\text{ENV } \varnothing]$$

$$\frac{\Gamma \vdash A \quad x \notin dom(\Gamma)}{\Gamma, x : A \vdash wf} \quad [\text{ENV } x]$$

$$\frac{\Gamma \vdash wf \quad A \in basetype}{\Gamma \vdash A} \quad [\text{TYPE CONST}]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \rightarrow B} \quad [\text{TYPE ARROW}]$$

$$\frac{\Gamma', x : A, \Gamma'' \vdash wf}{\Gamma', x : A, \Gamma'' \vdash x : A} \quad [\text{VAL } x]$$

$$\frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A.\, E : A \rightarrow B} \quad [\text{VAL FUN}]$$

$$\frac{\Gamma \vdash E : A \rightarrow B = \Gamma \vdash F : A}{\Gamma \vdash EF : B} \quad [\text{VAL APPL}]$$

这些类型系统的规则主要用于类型的推导。例如，关于表达式 $\lambda x : \mathrm{N}.\, x$ 的类型，我们有

$$\cfrac{\cfrac{\cfrac{\cfrac{\varnothing \vdash \text{wf} \quad \mathrm{N} \in \mathrm{K}}{\varnothing \vdash \mathrm{N}} \quad x \in dom(\varnothing)}{x : N \vdash wf}}{x : N \vdash x : N}}{\varnothing \vdash (\lambda x : N.\, x) : N \rightarrow N}$$

β 归约定义为：

$(\lambda x : A.\, E) F \rightarrow_{\beta} E[x := F]$

可作为类型系统 F_1 操作语义的一部分，相应的类型规则为：

$$\frac{\Gamma \vdash (\lambda x : A.\, E) F : B}{\Gamma \vdash E[x := F] : B} [\text{CONV } \beta]$$

类型系统 F_1 具有 Church-Rosser 性质，即 F_1 是相容系统。

β 归约保持了表达式的类型。

定理 3.1 若 $E : A$ 且 $E \rightarrow_{\beta}^{+} F$，则 $F : A$。

另一个重要的结果就是所谓的强范式化定理。

定理 3.2 对任意的项，不存在无限的归约序列。

使用类型系统时，会遇到有关类型的三个问题：

①类型检查。给定 Γ, E 和 A，$\Gamma \vdash E : A$ 是否可导出；

②类型能力。给定 E，找出 Γ 和 A，使得 $\Gamma \vdash E : A$ 可导出；

③居元。给定 A，找出 Γ 和 E，使得 $\Gamma \vdash E : A$ 可导出。

3.2 有子类型的高阶 λ 演算

在类型系统 F_1 中引入类型变量就得到了类型系统 F_2。在 F_2 中，用表达式 $E \equiv \Lambda\alpha.\, F$ 描述多态函数和类型应用 $E[A]$。若 $F : B$，则函数 E 的类型为 $\forall \alpha.\, B$。

类型系统 F_2 的理论特性与 F_1 在本质上是相同的，而且更严格的递归机制在其演算中能更为简洁地表示出来。在 F_2 中可定义的函数类比原始递归函数的能力更强。例如，著名的 Ackerman 函数在 F_2 中就可以表示出来[8]。F_2 的函数将类型作为参数，返回一个项。在其后继的类型系统 F_3 中，可以定义从类型到类型的函数。

在类型系统 F_3 中，引入了类型抽象，从类型到类型的映射可以使用类型表达式。类型表达式的一般形式为 $\Lambda\alpha.\, B$。若 $E \equiv \Lambda\alpha.\, F$ 是类型 $\forall \alpha.\, B$ 的一个多态函数，则应用 $E[A]$ 的类型为类型应用 $(\Lambda\alpha.\, B)A$。$\Lambda\alpha.\, B$ 和 $\forall \alpha.\, B$ 的根本不同在于：$\forall \alpha.\, B$ 是多态函数的一个类型，而 $\Lambda\alpha.\, B$ 是一个从类型到类型的函数。

由于类型抽象和应用需要区分鉴别，就必须引入类型变量的类型，这样就得到了类型系统 F_4。类型的类型称为属(kind)。项的类型的属是常量 kind，用 * 命名。在该类型系统中，若 K 是一个 kind，则 $* \rightarrow K$ 也是一个 kind。在更高阶的 kind 之前加上量词，就得到了 $F_5, F_6, \cdots$ 等类型系统，所有这些类型系统的并集就是 F^{ω}：

$F^{\omega} \equiv F_1 \cup F_2 \cup F_3 \cup \cdots$

其中，$\langle kind\rangle ::= * \mid \langle kind\rangle \rightarrow \langle kind\rangle$。因此，$F^{\omega}$ 的语法规则为：

$type ::= \langle basetype \rangle$

$\quad | (\langle typ \rangle \rightarrow \langle type \rangle)$

$\quad | \forall \langle type\ varible \rangle : kind. \langle type \rangle$

$\quad | \Lambda \langle type\ varible \rangle : kind. \langle type \rangle$

$\langle \lambda\_exp \rangle ::= \langle varible \rangle | (\lambda \langle varible \rangle : \langle type \rangle . \langle \lambda\_exp \rangle)$

$\quad | (\langle \lambda\_exp \rangle \langle \lambda\_exp \rangle)$

$\quad | \Lambda \langle type\ varible \rangle : \langle kind \rangle . \langle \lambda\_exp \rangle$

再引入类型关系 $A \leqslant B$ 后，F^{ω} 就扩充为有子类型的类型系统 $F^{\omega}_{\leqslant}$。

现有一些可能的对 $F^{\omega}_{\leqslant}$ 进一步的扩充，用于对面向对象程序设计的一些重要概念进行建模[6]。特别是包含了记录、表、存在类型和递归类型的 $F^{\omega}_{\leqslant}$ 是构建程序设计形式化模型的一个基础。引入依赖类型

$\Pi \langle varible \rangle : \langle type \rangle . \langle \lambda\_exp \rangle$

之后，就是构造演算（以及马丁洛夫类型理论），它们都是定理证明环境的基础。

4 直觉主义逻辑

与古典逻辑不同，直觉主义逻辑的表述是基于证明的存在性的，而前者基于绝对真值的概念。例如，对任意命题 A，如果不存在证明 A 成立或证明 A 不成立的方法，则就不存在排中律 $A \vee \neg A$ 的证明，因而排中律就不是直觉主义有效的。

4.1 极小逻辑

极小逻辑是命题逻辑的一个变种，其惟一的联结词是蕴涵，因此被称作极小。公式 F 的语法定义为

$F ::= V | F \rightarrow F$

其中，V 可以是任意命题变元。这一逻辑的构造特性可表述为：

• 一个变元 A 解释为 A 的一个未指明的构造；

• $A \rightarrow B$ 的一个构造是将 A 的一个构造转换为 B 的一个构造的方法。

即，证明一个公式是与该公式的一个构造（证明、函数、程序）等同的。$A \rightarrow A$ 的一个构造是一个函数 $f \equiv \lambda x : A.x, fA \rightarrow_{\beta} A$。

为记法上的方便，这里约定一种下标写法。直觉主义极小逻辑的证明形式记为 $\Gamma \vdash_E A$，其中 A 是公式，E 是 A 的构造（证明），Γ 是环境，则前面的例子可写成下面的形式：

$\varnothing \vdash_{\lambda x : A.x} A \rightarrow A$

从而，自然演绎证明系统可用极小逻辑刻画为如下的公理和推理规则：

$$\frac{}{\Gamma, x : A \vdash_x A} \quad [\text{IDENT AXIOM}]$$

$$\frac{\Gamma \vdash_E B}{\Gamma, x : A \vdash_E B} \quad \text{ASSUMP A}]$$

$$\frac{\Gamma, x : A \vdash_E B}{\Gamma \vdash_{\lambda x : A.E} A \rightarrow B} \quad [\text{IMPL INTRO}]$$

$$\frac{\Gamma \vdash_E A \rightarrow B \quad \Gamma \vdash_F A}{\Gamma \vdash_{EF} B} [\text{IMPL ELIM}]$$

4.2 同构

容易看出，类型系统的项就是直觉主义极小逻辑的证明，表达式的类型就是已证明的公式。这两个系统还有更多的对应关系（表 1），这些对应关系即 Curry-Howard 同构[7]。

从表 1 还可以看到：一个公式 A 是可证的，等同于类型 A 有居元（表 2），即若公式 A 有一个证明，则存在着一个类型为 A 的表达式。类似地，证明检查等同于类型检查[3]，即若 P 是公式 A 的一个证明，则 P 就是证明一个项具有给定类型 A 的一个推导。

表 1 两个系统的对应关系

直觉主义极小逻辑	类型系统 F
命题变元	类型变量
→ 逻辑联结词	→ 类型构造算子
公式（命题）	类型
假设	变量
蕴涵引入	抽象
蕴涵消除	应用
证明，构造	表达式

表 2 Curry-Howard 同构

直觉主义极小逻辑	类型系统 F_i
可证性	居元
证明检查	类型检查

因此，有以下定理成立。

定理 4.1 类型 A 有一个封闭的 λ 表达式，当且仅当 A 是直觉主义极小逻辑可证明的公式。

5 程序验证

考察了类型系统和逻辑间的同构关系之后，本节我们关注于逻辑和程序验证之间的关系。

证明一个程序满足其规范即程序验证。程序验证已有多种方法[1]，其中一种是操作语义的方法，由分析得到给定程序的执行序列；还有一种是公理方法，该方法需要一个语言来刻画程序的性质，这个语言就是谓词逻辑，与合式公式是一致的，使用公理和推理规则证明程序满足所期望的性质。

目前，自动程序验证工作已有大量研究。如果一个程序仅对有限的数据类型运算，即计算是在有限状态空间上的，则自动程序验证的确是可行的。检查一个程序是否为其规范的一个模型称为模型检查。

5.1 形式证明系统

在程序验证理论中，用合式公式描述程序的性质，而证明系统的任务是证明程序满足这些性质。一个证明系统是由有限的公理和推理规则构成的集合。

若 φ 是一个公理，则它可看作是一个给定的事实。利用规则，更多的事实可由公式中进一步推导出来。在证明系统中，公式的一个证明是一个有穷的序列：

$\varphi_1 \varphi_2 \cdots \varphi_n$

其中 φ_1 是公理，$\varphi = \varphi_n$，并且每个公式 $\varphi_i (1 \leqslant i \leqslant n)$ 或者是公理，或者是对其它公式应用推理规则而得到的公式。因此，φ 是一个定理，记作 $\vdash \varphi$。

刻画程序性质的谓词公式也称为断言。证明理论要涉及的正确性公式的一般形式为：

$\{Pre\} P \{Post\}$

其中，Pre 和 $Post$ 是断言，P 是程序。若 P 的每一次计算开始于满足 Pre 的状态，执行终止且终止时的状态满足 $Post$，则正确性公式为真，程序 P 具有完全正确性。如果不考虑计算是否发散，则 P 具有部分正确性。完全正确性仅对终止的计算是有效的。

5.2 寻找证明

定理证明器基于类型理论，采用了目标制导的简化策略：为找出一个目标的证明，首先找出更简单的子目标的证明。

（下转第 157 页）

记，并通知安全服务器增加两类访问控制规则，表示该客体适用于两类访问控制规则。例如，某访问控制语句规定某主体可以读某一类文件，经过处理，我们假设安全服务器为主体分配整数标记为1，为客体分配整数标记为2，则安全服务器记录了标记为1的主体对标记为2的客体有读的权限。此时，如果该客体代表的某个文件已经被分配了标记9，则该客体被重新分配一个标记，假设为11。安全服务器在已有的访问控制规则中搜索包含标记9的访问控制规则，每找到一条，就生成一条新规则，以11替换9。另外，安全服务器再添加一条规则，记录标记为1的主体对标记为11的客体有读的权限。

FMAC实现了五个施加于全系统的安全策略，分别是满足文[3]中保护内核完整性及保护系统文件完整性要求的两个安全策略、基于角色的访问控制策略、能力策略和多级安全策略，这些策略都只对文件和目录进行约束。FMAC还实现了一个示例性的施加于某个普通用户的安全策略，该安全策略规定所有该用户的程序都不能访问其Home目录下的某个子目录。

对于一个访问控制决策请求（由MAC框架调用FMAC的钩子函数），FMAC的安全服务器可以直接判断其施加条件是否满足，并使用并行组合方式综合多个安全策略的决策结果。

小结 针对当前操作系统的安全需求，本文在FLASK安全体系结构的基础上提出了一种灵活的强制访问控制框架FMAC，FMAC模型由基于LTS的安全策略模型、对象管理器模型和安全服务器模型组成。本文论述了Unix类操作系统中FMAC的实现，并基于TrustedBSD实现了一个FMAC原型。

接下来的工作主要是完善FMAC模型和实现，并采用访问控制决策缓冲等机制解决实现原型中暴露的对操作系统性能的影响问题。

参考文献

1 易晓东，何连跃，杨学军. 安全操作系统基于角色的授权机制. 计算机工程与科学，2004(增刊)

2 Spencer R, Smalley S, Loscocco P, et al. The Flask Security Architecture: System Support for Diverse Security Policies. In: Proc. of the Eighth USENIX Security Symposium, 1999. 123～139

3 Loscocco P, Smalley S D. Meeting Critical Security Objectives with Security-Enhanced Linux. In: Proceedings of the 2001 Ottawa Linux Symposium, 2001

4 Watson R, Feldman B, Migus A, et al. Design and Implementation of the TrustedBSD MAC Framework. In: Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX III), 2003

5 Watson R, Morrison W, Vance C, et al. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In: Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference (FREENIX '03), 2003

6 Lampson B W. Protection. In: 5th Princeton Symposium on Information Science and Systems, 1971

7 Siewe F, Cau A, Zedan H. A compositional framework for access control policies enforcement. FMSE'03, ACM, 2003

8 Hale R W S. Programming in Temporal Logic: [PhD thesis]. Trinity College, University of Cambridge, 1988

9 Sandhu R S, Coyne E J, Feinstein H L, et al. Role-Based Access Control Models. IEEE Computer, 1996, 29(2): 38～47

10 Loscocco P, Smalley S. Integrating Flexible Support for Security Policies into the Linux Operating System. In: Proc. of the FREENIX Track of the 2001 USENIX Annual Technical Conf. 2001

（上接第143页）

然后，对子目标的证明应用推理规则，从而产生原始目标的证明。

这样的证明策略可以表示为一个函数，将一个证明目标映射为一个包含子目标列表和一个确认证明的序偶。一个确认证明可将子目标的证明映射为原始目标的证明，即

$$type\ tactic = goal \rightarrow (goal\ list \times (proof\ list \rightarrow proof))$$

函数 *tactic* 是证明系统中自动应用规则的一个基本工具，它检查给定目标，并将其分解成一些要解决的子目标，向前或向后推导出新的证明公式或确定一个证明是否是该公式的证明。*Tactic* 的主要用途是将一个证明转换成自然推理。例如，将函数应用到目标 g 上返回$([],f)$，则 $f[]$就是 g 的证明。

定理证明器通常使用交互式的目标制导策略，用户可以指定一些推理步骤，证明的细节是自动进行的。由此，用户可以组合所谓的原始证明，它们可以是推理规则，或是判定过程的调用。例如，PVS有许多原始证明和一个小的 *tactic* 语言，而Isabelle/HOL系统有一个完整的程序设计语言(ML)，作为 *tactic* 语言来组合复杂的原始证明。

如果希望通过选择、复合或重复的方法组合原始的 *tactic*，则可以构建组合子 *tacticals*，一些基本的组合子如：

T_1 then T_2　应用 T_1，然后应用 T_2；

T_1 orelse T_2　试应用 T_1，若失败，则再应用 T_2；

repeat T_1　重复应用 T_1，直至失败。

结论 很多广为使用的证明器都是基于某一类型理论的，这表明：

- 类型对于组织形式化的知识是有用的。
- 对象的类型可以将一些有用的信息传送给证明者。
- Curry-Howard同构给出了一种简便的方法，可以从证明中进行程序综合。

应该看到，目前形式化的正确性证明还是一项十分复杂和耗时的工作。证明简单的程序是十分容易的，一旦命题十分复杂，则证明的复杂度显著增长。当然，培训和利用工具软件有助于缩短所需时间。

形式化证明的一个优点也是明显的，至少可以使我们更为深刻地认识程序的规律，更好地理解算法，从而改进并简化代码及文档。

参考文献

1 Apt K R, Olderog E R. Verification of Sequential and Concurrent Programs Springer-Verlag, 1997

2 Csornyei Z. Type Systems, Lecture Notes (2003). http://people.inf.elte.hu/csz (In Hungarian)

3 Dunfield J, Pfenning F. Tridirectional Typechecking. In: POPL'04, 2004

4 Harper R, Pfenning F. Type Refinements, Project Description, 2001. http://www-2.cs.cmu.edu/1triple/triple.pdf

5 Pierce B C. Types and Programming Languages. The MIT Press, 2002

6 Schwartzbach M I. Polymorphic Type Inference BRICS Lecture Series, LS-95-3, 1995

7 Sørensen M H B, Urzyczyn P. Lectures on Curry-Howard Isomorphism. Lecture Notes, University of Copenhagen, University of Warsaw, 1999

8 Zoltan C. Type Systems and Program Verification. In: 6th International Conference on Applied Informatics, 2004, 27～31