

Зміст

1. Вступ	2
2. Опис мови ForTheL	4
3. ForTheL тексти	18
4. Технічні особливості реалізації системи САД на платформі Windows	22
5. Приклади використання мови ForTheL для запису математичних текстів	30
6. Висновки	42
7. Література	43

Вступ

Стрімкий розвиток обчислювальної техніки наприкінці минулого століття досягнув рівня достатнього для проведення дуже складних обчислень. Паралельно з технічними досягненнями, були отримані важливі результати в суто теоретичних областях. Виникла потреба використовувати можливості обчислювальних машин для обробки математичних текстів, зокрема, для автоматичного доведення теорем.

На початку 60-х років академік В. Глушков запропонував програму, яка отримала назву Алгоритму Очевидності. Він пропонував одночасно проводити дослідження в різних напрямках: по створенню формалізованих мов для подання математичних текстів у зручній для форми, розвитку логічних засобів для автоматизації обробки таких текстів, створення інформаційного середовища, яке б використовувалось в процедурі доведення. Така процедура повинна передбачати можливість інтерактивної взаємодії з користувачем у разі виникнення неоднозначних ситуацій. Ідея Алгоритму Очевидності полягала в створенні системи, яка б давала змогу автоматизувати частину рутинної роботи при розв'язуванні математичних проблем, а також проблем з інших областей знань, що допускають формалізацію.

Перша робота присвячена розробці процедури для доведення теорем теорії груп з'явилась в 1966 році. Подальші вдосконалення призвели до створення машинно-орієнтованої процедури пошуку виводу в логічних численнях. У той самий час розроблялась мова опису математичних теорій. Ця мова була схожа на мову логік першого порядку.

У 1970 році почався новий період реалізації програми алгоритму очевидності. Особливість цього періоду розробки полягала в орієнтації на створення єдиної системи обробки математичних текстів. Тоді була створена мова TL (Theory Language), яка була схожа на звичайну математичну мову та дуже зручна для використання як машиною так і людиною. У 1976 році почалось впровадження експериментальної системи обробки математичних текстів. У результаті в 1980 була створена перша версія системи. Система отримала назву САД (Система Автоматизації Дедукції).

Сучасна реалізація системи відноситься до 2002 року. Вхідною мовою стала мова ForTheL (Formal Theory Language), яка була створена на базі TL.

Особливість ForTheL полягає в тому, що вона дозволяє записувати математичні твердження мовою схожою на природну людську мову. У поточній реалізації в якості

такої мови виступає англійська. Очевидною перевагою такого підходу є зрозумілість написаних текстів, на противагу мові якої не будь традиційної логіки, як це зазвичай представлено в подібних системах. Іншим аргументом на користь використання формальної природної мови, є те, що багато інформації в такому тексті знаходиться поза логікою, особливо тієї, що втрачається при перекладі (з природної мови в мову логіки). Наприклад, у підручнику ми знаходимо аксіоми, визначення, теореми, твердження, схеми доведень пояснення, тощо. Там, де людина прагне до розрізнення класів тверджень, у формальних логічних записах вони зникають: означення, аксіоми, теореми, висловлення, інші судження перетворюються на формули, предикатні та функціональні символи, тощо. Це зумовлено передусім бідністю засобів мови логіки.

Система доступна в двох варіантах: через WEB вузол <http://ea.unicyb.kiev.ua> або у варіанті персонального використання. Персональний варіант орієнтований на платформу Linux(x86).

Система дозволяє виконувати декілька основних функцій: пошук доведення, верифікацію тексту, трансляцію тексту ForTheL у мову логіки першого порядку. Для пошуку доведення можуть бути використані як стандартні засоби системи, так і додаткові, для яких передбачено інтерфейс підключення.

Метою даної роботи було проведення аналізу мови ForTheL, переклад і верифікація реальних математичних текстів з підручників [2] та [3], і як результат розробка методики перекладу математичних текстів на формалізовану мову.

Іншим завданням стало перенесення системи САД на платформу Windows, розробка зручного інтерфейсу користувача та проведення тестувань.

Опис мови ForTheL

Текст ForTheL складається з послідовності речень та складених секцій. У свою чергу складені секції також являють собою послідовність згрупованих речень.

Речення будується за допомогою композицій та більш простих синтаксичних одиниць. Існує чотири види таких одиниць:

- **Поняття** визначає загальний, можливо параметризований, клас об'єктів (natural number, element of S, series that converges N).
- **Терм** визначає об'єкт, або вказуючи конкретне значення (N, the complement of S, $X*Y$), або за допомогою квантифікації поняття (every set, some divisor of M).
- **Предикат** визначає властивість об'єкта: empty, divides N, is a subset of S; застосований до терма, предикат формує твердження; застосовуючи предикат до поняття, отримуємо нове більш вузьке поняття.
- **Твердження** визначає логічний вираз, який може бути вірним або хибним. Твердження бувають прості і складені. Складені твердження утворюються з простих за допомогою композицій. Твердження перекладаються у формули мови першого порядку.

Для побудови синтаксичних одиниць в мові існує засіб, що має назву **синтаксичного примітиву**. На кожному етапі синтаксичного аналізу процесор мови працює з певною множиною примітивів, які формують сигнатуру оброблюваного тексту. Надалі цю множину будемо називати **поточним словником**. У процесі розбору тексту поточний словник може поповнюватись новими синтаксичними примітивами. Для кожного тексту властивий свій поточний словник.

Синтаксичні примітиви поділяються на шість різних класів, кожний з яких використовується для побудови різних синтаксичних одиниць.

- *Класовий іменник* – для формування простих понять: element of argument.
- *Визначений іменник* – для формування функціональних термів: zero, power set of argument
- *Прикметники та дієслова*, для побудови предикатів: equal to argument.
- *Функціональні символи*, для побудови символічних термів: argument1 + argument2.

- *Предикатні символи*, для побудови символічних тверджень: argument1 <= argument2.

Базові примітиви поміщуються прямо в текст за допомогою спеціальної конструкції, яка називається представленням (*introductor*). Кожне нове представлення може формувати додаткову групу примітивів. Наприклад, примітива іменників з конструкцією of – argument (a subset of argument, complement of argument) породжують спеціальні примітиви для використання в присвійних предикатних одиницях: *has an element of an infinite cardinality*.

Синтаксис представлень має таку структуру:

```

Introductor -> [ (a|an) pattern [ @ [a|an] nounNotion ]
               | [ the pattern | @ plainTerm ]
               | [ variable is pattern [ @ statement ] ]
               | [ variable pattern [ @ statement ] ]
               | [ symbPattern @ plainTerm ]
               | [ symbPattern @ statement ]

```

Кожне представлення містить шаблон (pattern), який визначає синтаксис нового примітива. Примітив може (символьний примітив обов'язково повинен) бути синонімом для синтаксичної одиниці підходящого вигляду, у такому випадку за шаблоном слідує символ @ і необхідна синтаксична одиниця (ціль). Нетермінали nounNotion, plainTerm та statement використовуються як цілі. Їх значення буде роз'яснене в подальшому тексті. Всі примітиви, що використовуються в цілі повинні бути визначеними в поточному словнику.

Шаблон – це непуста послідовність лексем (які визначають термінали для нового примітива) розділених змінними (які визначають місця для аргументів).

Синтаксис шаблонів такий:

```

Pattern -> {tokens} tokens [variable {tokens variable}]
Tokens  -> token [ '/' token]
Token   -> small {small}

```

```

symbPattern -> [variable] symbToken {variable symbToken} [variable]
               | word (variable {, variable} )

```

| word [variable]

symbToken -> symbol {symbol}

У не символних шаблонах можна використовувати декілька альтернативних варіантів для однієї лексеми. Це дає можливість використовувати іменники та дієслова у формі множини або однини. Наприклад, (subset/subsets). Артиклі та всі форми дієслова “to be” не повинні зустрічатись в шаблоні (так як вони є ключовими словами мови ForTheL).

В представленнях дієслів, змінна яка стоїть на початку не може позначатись літерою A або a, для уникнення колізій з представленнями класових іменників. Всі змінні в шаблоні повинні бути різними. У синонімах кількість вільних змінних повинна співпадати з кількістю вільних змінних у шаблоні.

Поняття.

Кожне поняття будується на основі основного поняття (primaryNotion), до якого застосовуються атрибути. Атрибути потрібні для звуження значення поняття, надання йому якихось додаткових властивостей. Синтаксис поняття:

primaryNotion -> primClassNoun | notionSymbol

primClassNoun -> (**set** | **sets**) [names]

| (**element** | **elements**) [names] (**of**) term

| (**function** | **functions**) [names] (**from**) term (**to**) term

|

notionSymbol -> primNotionSymbol | (primNotionSymbol)

primNotionSymbol -> names (<<) symbTerm

| names (:) symbTerm (->) symbTerm

| ...

names -> variable {, variable}

Символьні поняття (notionSymbol) – примітиви що успадковуються. Вони будуються автоматично з дескриптивних предикатних символів. Наприклад, визначені вище символьні поняття будуються з таких примітивів:

```
[ x << y @ x is an element of y]
[f : D -> R @ f is a function from D to R]
```

Кожне поняття має список імен. Ці ідентифікатори посилаються на конкретні об'єкти взяті з класу і відіграють ту ж роль, що і імена квантифікованих змінних. Використання імен можна побачити на такому прикладі:

Every natural number m greater than 0 divides m!

Як, уже зазначалось атрибути служать для обмеження класу який визначає поняття. Атрибути базуються на предикатах і виразах. Кожне поняття може мати лівий та правий атрибут:

```
leftAttribute -> primSimpleAdjective | primSimpleAdjectiveM
rightAttribute -> isPredicate { and isPredicate }
                | that doesPredicate { and doesPredicate }
                | such that statement
```

Прості прикметники (primSimpleAdjective) які формують лівий атрибут – також наслідувані примітиви. Вони будуються з тих прикметників та m-прикметників (див. далі) які не мають аргументів: кожний такий примітив просто копіюється в список простих (m-)прикметників:

```
primSimpleAdjective -> (prime)|(empty)|(natural)| ....
primSimpleAdjectiveM -> (equal)|(parallel)|(disjoint)| ....
```

Тепер ми можемо визначити повний синтаксис поняття:

```
Notion -> {leftAttribute} primaryNotion {rightAttribute}
```

Приклади понять:

Cyclic group G

Injective $f: \text{Nat} \rightarrow \text{Nat}$ that maps 0 to 0

Real number greater than 0 and less than 1

Терми.

У мові присутні два типи термів – визначені терми та квантифіковані поняття:

Term \rightarrow quantifiedNotion | definiteTerm

Квантифіковані поняття дозволяють формулювати загальні твердження про кожний об'єкт із класу, що визначений цим поняттям, або констатувати існування об'єктів класу наділених певними властивостями:

quantifiedNotion \rightarrow realQuantifiedNotion | { realQuantifiedNotion }

realQuantifiedNotion \rightarrow (every | each | all | any) notion

| some notion

| no notion

Визначений терм (definiteTerm) формується застосуванням функції до аргументів терму. Мова дозволяє писати такі застосування словами англійської мови або за допомогою функціональних символів. В ForTheL фортель діють такі домовленості щодо пріоритетів та асоціативності функціональних символів:

- **Постфіксні символи** – чиї примітиви закінчуються лексею (token), мають найвищий пріоритет.
- **Постфіксні символи** – чиї примітиви починаються з лексеми і закінчуються аргументом (змінною), мають менший пріоритет.
- **Інфіксні символи** – чиї примітиви мають аргументи з обох боків мають найнижчий пріоритет серед функціональних символів і праву асоціативність.

Для правильного запису термів розробниками рекомендується використовувати групування (взяття в скобки).

Синтаксис визначених термів:

definiteTerm -> realDefiniteTerm | (realDefiniteTerm)

realDefiniteTerm -> [the] primDefiniteNoun | symbTerm

symbTerm -> primInfixFunctionSymbol | prefixSymbTerm

prefixSymbTerm -> primPrefixFunctionSymbol | prefixSymbTerm

postfixSymbTerm -> primPostfixFunctionSymbol | (symbTerm) | variable

Приклади типових примітивів для визначених іменників і функціональних символів:

primDefinititeNoun -> (**zero** | **zeroes**)

| (**order** | **orders**) (of) term

|

primInfixFunctionSymbol -> prefixSymbTerm (*) symbTerm

|

primPrefixFunctionSymbol -> (**min**) prefixSymbTerm

|...

primPostfixFunctionSymbol -> (**0**)

| (**exp**) ('(' symbTerm (',' symbTerm (')')

|

Простий терм – це терм який не містить квантифікованих понять. Іншими словами, простий терм – це визначений терм, чиїми аргументами є також прості терми.

plainTerm -> realPlainTerm | (realPlainTerm)

realPlainTerm -> [**the**] primPlainNoun | symbTerm

primPlainNoun -> (**zero** | **zeroes**)

|

Кожний примітив визначеного поняття автоматично продукує два примітива для простих понять, замінюючи терми на прості терми в місцях для аргументів.

Предикати.

Функції та поняття дають нам можливість визначати об'єкти; предикати описують їх властивості та зв'язки між ними. Існує три типи основних (не складних) предикатів: побудовані на основі дієслів та прикметників, предикати які встановлюють належність до класу ("is a" – предикати) та предикати яка виражають існування об'єктів пов'язаних із суб'єктом ("has" – предикати). Основні предикати можуть бути заперечені або зв'язані в кон'юнкцію для формування складних предикативних одиниць.

Синтаксис предикатів визначений наступним чином:

```
doesPredicate -> [does|do] [not] primVerb
  | [does|do] [not] [pairwise] primVerb
  | (has|have) hasPredicate
  | (is|are|be) isPredicate { and isPredicate }
  | (is|are|be) is_aPredicate { and is_aPredicate }
```

```
isPredicate -> [not] primAdjective
  | [not|pairwise] primAdjectiveM
  | (with|of|having) hasPredicate
```

```
is_aPredicate -> [not] [a|an] nounNotion
  | [not] definiteTerm
```

```
hasPredicate -> [a|an|the] possessed { and [a|an|the] possessed }
  | no possessed
```

Прості прикметники та дієслова мають наступний вигляд:

```
primVerb -> (convereges | converge)
  | (divides|divide) term
  | (equals|equal) (to) term
  | .....
```

primAdjective -> (prime)

- | (dividing) term
- | (equal) (to) term
- | (less) (than) term
- |

Простий прикметник *equal to* визначений за замовченням.

Прості дієслова та прикметники можуть автоматично породжувати *мультиоб'єктні примітиви* або *м-примітиви*. Правило породження наступне. Візьмемо примітив з хоча б одним аргументом зі списку primVerb або primAdjective. Якщо першому аргументу передує прийменник та за ним не слідує лексема (**and**), то породжується новий примітив в primVerbM (primAdjectiveM), шляхом вилучення першого аргументу разом з передуючою лексемою.

Так, простий прикметник ((parallel) (to) term) породжує м-прикметник ((parallel)).

М-прикметники та м-дієслова можуть застосовуватись до кількох аргументів: parallel line m,n.

По домовленості вважаємо мультисуб'єктні предикати симетричними та транзитивними. Для не транзитивних предикатів, використовується ключове слово **pairwise** (попарно):

A,B,C are pairwise disjoint \leftrightarrow A is disjoint with B and A is disjoint with C and B is disjoint with C.

Синтаксис м-примітивів:

primVerbM -> (collides | collide)

- | (commutes | commute) (wrt) term
- |

primAdjectiveM -> (equal)

- | (adjacent) (in) term
- |

За допомогою “is a” предикатів можна сказати що об’єкт описаний поняттям або рівний визначеному терму: X is natural number, X is the order of G . Ми не можемо використовувати символні поняття в “is a” предикатах або цілях у представленнях (introducers), тому в ForTheL введено спеціальний нетермінал nounNotion для понять побудованих на класових-іменниках.

nounNotion -> {leftAttribute} primClassNoun {rightAttribute}

Для “has” – предикатів підтримується спеціальна група примітивів які спадкувались від простих функцій та понять:

Possessed -> {leftAttribute} primPossessedNoun {rightAttribute}

primPossessedNoun -> (element|elements) [names]

| (solution|solutions) [names]

|

Ці примітиви породжується автоматично за таким правилом. Для іменникового примітива з хоча б одним аргументом з primClassNoun або primDefiniteNoun. Якщо першому аргументу передує лексема (of) та за ним не слідує лексема (end), тоді породжується новий possessed noun примітив, шляхом вилучення першого аргументу разом з лексемою (of) та додаванням [names] при потребі.

Так, простий класовий-іменник:

(ambassador|ambassadors) [names] (of) term (in) term

породжує новий possessed noun примітив:

(ambassador|ambassadors) [names] (in) term

Хоча примітив ‘ union of _argument_ and _argument_ ‘ не породжує ніякого нового примітива.

Приклади тверджень з “has” – предикатними:

X has no elements

Every set of a finite cardinality is finite

F has the domain D and the range R such that D is a subset of R.

У другому твердженні “has” – предикат відіграє роль правого атрибута в понятті.

Твердження

Твердження в мові ForTheL – це аналог логічних формул. Спочатку розглянемо елементарні (атомні) твердження. Такі твердження бувають чотирьох типів:

```
atomicStatement -> simpleStatement
                  | thereIsStatement
                  | [we have] symbStatement
                  | [we have] specialStatement
```

Прості твердження (simpleStatement) будуються шляхом застосування предикатів до термів:

```
simpleStatement -> terms doesPredicate {and doesPredicate}
terms -> term { (,|and) term }
```

Так звані ‘there is’ – твердження підтверджують або спростовують не порожність деякого класу:

```
thereIsStatement -> there (exists|exist) notions
                  | there (exists|exist) no notion
```

```
notions -> [a|an] notion { (,|and) [a|an] notion }
```

Символьні твердження компонуються в традиційному синтаксисі мови першого порядку із символьних предикатів та термів:

```
symbStatement -> forall notionSymbol symbStatement
                 | exists notionSymbol symbStatement
```

| symbStatement \Leftrightarrow symbStatement
 | symbStatement \Rightarrow symbStatement
 | symbStatement \vee symbStatement
 | symbStatement \wedge symbStatement
 | **not** symbStatement
 | (Statement)
 | primPredicateSymbol

primPredicateSymbol \rightarrow symbTerms (=) symbTerm
 | symbTerms (!=) symbTerm
 | symbTerms (\Leftarrow) symbTerm
 | symbTerms (:) symbTerm (\rightarrow) symbTerm
 | (Nat) ('(') symbTerm (')')
 |

symbTerms \rightarrow symbTerm { , symbTerm }

Серед тверджень виділяють твердження спеціального типу:

specialStatement \rightarrow [**the**] **thesis** | [**the**] **contrary** | [**a|an**] **contradiction**.

Їх використовують при записі доведень теорем.

Елементарні твердження можна компонувати в більш складні за такими правилами:

Statement \rightarrow headStatement | chainStatement

headStatement \rightarrow **for** quantifiedNotion { **and** quantifiedNotion } statement
 | **if** statement **then** statement
 | (**it is wrong that|not**) statement

chainStatement \rightarrow andChain [**and** headStatement]
 | orChain [**or** headStatement]
 | (andChain|orChain) **iff** statement

andChain \rightarrow atomicStatement { **and** atomicStatement }

orChain -> atomicStatement {or atomicStatement }

Як уже було зазначено кожному твердженню мови ForTheL відповідає формула класичної логіки першого порядку. Така формула називається зображенням твердження. Зображення твердження S будемо позначати: $|S|$.

Опис змінних

Вільними змінними твердження S будуть ті змінні які є вільними в зображенні твердження $|S|$. Множину вільних змінних твердження $|S|$ будемо позначати: $FV(s)$ (free variables). Множину зв'язаних змінних будемо позначати: $BV(S)$ (bounded variables).

Правила мови ForTheL забороняють використовувати твердження, які містять вкладені квантори з однаковою змінною. Наприклад, твердження every number n divides some number n вважається не коректним.

Будемо говорити що твердження S описує (оголошує) змінну $v \in FV(S)$ кожний раз коли в S стверджується, що v належить до певного класу.

Терм t називається позитивним (реальним) коли t не є квантифікованим поняттям у формі заперечення (**no** notion) та всі аргументи терма t також позитивні терми.

Синтаксичний примітив I називається дескриптивним, якщо виконується одна з наступних умов:

- I – прикметник *equal to argument*;
- I – простий прикметник, дієслово або предикатний символ; I – представлений як синонім; шаблон представлення починається зі змінної v , яка описується і цілі.

Предикат P дескриптивний (описовий), якщо він не заперечується та виконується одна з умов:

- P – ‘is a’ предикат над іменниковим поняттям з позитивними аргументами.
- P – ‘is a’ предикат над позитивним визначеним термом.
- P – побудований над дескриптивним примітивом прикметника або дієслова з позитивними аргументами.

- P – композиція базових предикатів, один з яких є дескриптивним.

Нарешті, твердження S описує змінну v, якщо виконується одна з таких умов:

- S – дескриптивний предикатний символ і перший аргумент із списку змінних твердження містить v.
- S – просте твердження таке що підмет виступає в якості послідовності змінних, яка містить v та предикат твердження є дескриптивним.
- S – кон’юнкція декількох тверджень одне з яких є дескриптивним.

Визначення (*definition statements*)

У мові ForTheL присутні спеціальні типи тверджень, які використовуються у визначеннях.

Синтаксис таких тверджень має наступний вигляд:

defStatement -> notionDef | functionDef | predicateDef

notionDef -> [**a|an**] primClassNoun **is** [**a|an**] notion
 | primNotionSymbol **iff** variable **is** [**a|an**] notion

functionDef -> functionSym **is equal to** plainTerm

functionSym -> [**the**] primDefiniteNoun
 | primInfixFunctionSymbol
 | primPrefixFunctionSymbol
 | primPostfixFunctionSymbol

predicateDef -> predicateSym **iff** statement

predicateSym -> variable **is** primAdjective
 | variable ‘,’ variable **are** primAdjectiveM
 | variable primVerb
 | variable , variable primVerbM

|primPredicateSymbol

Кожне визначення S має так званий головний *junctor*, який може бути рівністю, еквівалентністю, або зв'язкою *is*. Синтаксична одиниця зліва від *junctor* називається *головою* визначення, а синтаксична одиниця справа – *тілом* визначення.

Визначення вважається коректним, якщо виконується одна з наступних умов:

- у голові визначення всі терми на місцях аргументів – змінні.
- Жодна змінна не зустрічається двічі в голові визначення.
- Кожна вільна змінна в тілі визначення є в голові визначення.
- Якщо примітив у голові визначення представлений синонімом тоді відповідна ціль – також повинна бути примітивом.
- Примітив що знаходиться в голові не повинен зустрічатись в тілі (в поточній реалізації мови, рекурсивні визначення не допускаються).
- Якщо S визначає поняття, тоді поняття що знаходиться в голові повинно мати хоча б одне ім'я
- Якщо S визначає символічне поняття тоді ім'я голови – суб'єкт у тілі.

Тепер ми можемо перейти до розгляду практичного використання мови.

ForTheL тексти.

У попередніх розділах вводились основні типи синтаксичних одиниць мови. З їх допомогою можна будувати твердження, які згодом транслюються у формули мови першого порядку. Але зрозуміло, що для написання математичних текстів, лише одних тверджень недостатньо. Текст записаний мовою ForTheL складається з представлень (*introductor*) та секцій верхнього рівня. Змістовно такими секціями є аксіоми, визначення та теореми.

Кожна з таких секцій містить заголовок, що описує її тип. Крім того, після заголовку може слідувати мітка (назва,label) секції. Мітки використовуються для посилань.

Загальний синтаксис ForTheL тексту визначається такими правилами:

```
Text -> { axiom | definition | proposition | introductor }
axiom -> axmHeader { assume } axmAffirm
axmHeader -> Axiom [label]
definition -> defHeader { assume } defAffirm
defHeader -> Definition [label]
proposition -> prpHeader { assume } prpAffirm
prpHeader -> (Proposition | Theorem | Lemma | Corollary) [label]
lable -> word
```

Крім заголовку та мітки кожна секція повинна містити твердження, відповідного до заголовку типу,(Affirmation, у граматичних правилах –Affirm) та може містити припущення (assume). Припущення служать для опису змінних які будуть використані в подальших твердженнях. Наприклад: Let S be a set. Описує змінну S, що являється множиною. Крім того, для опису змінних можуть бути використані селекції (selections), наприклад: Take an empty set E. Селекції складаються з понять та стверджують не порожність відповідних класів. (Нагадаємо, що поняття задають клас об'єктів)

```
assume -> asmPrefix statement
asmPrefix -> let | [let us|we can] (assume|suppose) [that]
select -> selPrefix notions [reference]
selPrefix -> [then|therefore|hence] [let us|we can] (take|choose)
```

axmAffirm -> statement

defAffirm -> defStatement

prpAffirm -> affPrefix statement [reference] ‘.’ [prfHeader proof]

|prfPrefix statement [reference] ‘.’ Proof

affPrefix -> [**then|therefore|hence**]

prfHeader -> proof [**by** method] ‘.’ | **indeed**

prfPrefix -> [**let us|we can**] (**prove|show**) [by method] [**that**]

method -> **contradiction** | **case analysis** | **induction** [on plainTerm]

reference -> ‘(**by** label { ‘,’ label })

Доведення (Proof Sections)

У коректному тексті кожне твердження (Affirmation) повинно бути обґрунтованим (за виключенням тверджень які приймаються як аксіоми або визначення). Воно повинно логічно слідувати з попереднього тексту. Автор тексту може зробити це слідування більш очевидним, як за допомогою посилань на необхідні секції верхніх рівнів так і використовуючи секцію доведень (proof section), у якій проводиться логічне виведення цього твердження.

Доведення в ForTheL складається з припущень (assumptions), селекцій(selections), та інших тверджень (affirmations), які можуть містити свої доведення, а також з особливих композицій. Такими композиціями може бути розгалуження або блок доведення.

Блок доведення служить для структуризації доведення. Вони визначають область дії припущень та описів змінних. Блок доведення може бути в будь-якому місці доведення.

Розгалуження служить для розгляду різних можливих випадків які виникають в процесі доведення. Розгалуження слідують одне за одним. Кожне розгалуження містить заголовок та доведення. Заголовок містить твердження, яке виражає випадок. Далі слідує доведення відповідного твердження в такому випадку.

Доведення представляється такими граматичними правилами:

```
proof -> [ {prfBody}prfLast ] qed
qed -> end. | qed. | obvious. | trivial.
prfBody -> assume | select | prpAffirm | block
prfLast -> prpAffirm | block | case {case}
block -> blkHeader proof
blkHeader -> Block [label] ‘.’ | now | first | second | ...
case -> Case statement [reference] ‘.’ Proof
```

При описі доведення можна описувати метод. ForTheL підтримує доведення від супротивного (by contradiction), аналіз випадків (by case analysis), та за індукцією (by induction).

Формалізація процесу доведення за індукцією в ForTheL ґрунтується на наступному загальному індукційному принципі:

$$\forall \vec{x}(IH(\vec{x}) \supset F[\vec{x}]) \supset \forall \vec{x}F[\vec{x}]$$

, де $IH(\vec{x}) = \forall \vec{y}((t[\vec{y}] \prec t[\vec{x}]) \supset F[\vec{y}])$, t терм, а \prec відношення порядку. Змістовно принцип виражає: доводячи, що властивість P має місце для деякого але фіксованого \vec{x} , можна припустити що ця властивість має місце для всіх \vec{y} менших за \vec{x} , зважаючи на визначене відношення порядку. Індукційний терм t використовується для групування кортежів значень в одне.

Отже, доведення за індукцією записуються та перевіряються враховуючи наведений принцип індукції. Ця гіпотеза формулюється автоматично і стає додатковим логічним попередником для аргументу індукції.

Наведемо приклад доведення за індукцією:

Proposition P_h. For all terms t, r $t + r = r + t$.

Proof by induction on r .

Let t, r be terms.

Case $r = 0$.

$r + 0 = r$.

$r = 0 + r$.

$r+0 = 0+r$.

end.

Case $r \neq 0$.

Take a term x such that $x' = r$.

$t + x = x + t$. #By induction hypothesis!

$t + (x') = (t + x)'$.

$(x')+t = (x + t)'$.

$(t + x)' = (x + t)'$.

$t+(x') = (x + t)'$.

$t+(x') = (x') + t$.

Hence $t + x = x + t \Rightarrow t + (x') = (x') + t$.

end.

end.

Технічні особливості реалізації системи САД на платформі Windows

Оскільки система оригінально розроблялась з орієнтацією на платформу Linux, є доцільним розглянути деякі особливості технічної реалізації, які характерні для цієї платформи. Ці особливості стосуються апарату вводу / виводу, потоків та процесів, а також способу роботи із зовнішніми модулями (програмами).

Для роботи з вище згаданими засобами технічної реалізації, операційна система на ядрі Linux використовує стандарт Posix, який описує низько рівневі механізми роботи із цими класами об'єктів. Платформою MS Windows стандарт Posix не підтримується, (хоча є спроби створити середовище, яке емулює Linux, наприклад, Cygwin) замість нього використовується власний апарат, заснований на об'єктах ядра. Для нас важливими об'єктами є процеси (process), потоки (thread) та канали вводу / виводу (pipe). Оскільки, для переносу системи на Windows платформу, нам необхідно замінити всі системно залежні процедури, на еквівалентні нові з використанням апарату об'єктів ядра.

Розглянемо більш детально, що собою являють об'єкти ядра. Кожний об'єкт ядра - насправді просто блок пам'яті, який був виділений ядром ОС і доступний тільки йому. Цей блок являє собою структуру даних, в елементах якої міститься інформація про об'єкт. Деякі елементи (дескриптор захисту, лічильник кількості користувачів) є спільними, але більша їх частина специфічна в залежності від типу об'єкта. Наприклад, в об'єкта „процес” є ідентифікатор, базовий пріоритет та код завершення, а в об'єкта „файл” – зміщення в байтах та режим доступу. Оскільки ці структури доступні тільки ядру, програма не може самостійно знайти їх в пам'яті і напряду модифікувати вміст.

При ініціалізації процесу система створює в ньому таблицю дескрипторів, що використовується тільки для об'єктів ядра. Відомості про структуру цієї таблиці і керуванні нею не задокументовані.

На схемі показано, як приблизно виглядає таблиця дескрипторів, що належить процесу. Це просто масив структур даних. Кожна структура містить вказівник на який-небудь об'єкт ядра, маску доступу й деякі флаги.

Індекс	Вказівник на блок пам'яті об'єкта ядра	Маска доступу	Флаги
1	0XXXXXXXXX	0XXXXXXXXX	0XXXXXXXXX
2	0XXXXXXXXX	0XXXXXXXXX	0XXXXXXXXX

Структура таблиці дескрипторів, що належить процесу (X = 0 або 1).

Коли процес ініціалізується в перший раз, таблиця дескрипторів ще порожня. Але як тільки один з його потоків викликає функцію, що створює об'єкт ядра (наприклад, CreateFileMapping), ядро виділяє для цього об'єкта блок пам'яті і заповнює початковими даними, далі ядро переглядає таблицю дескрипторів, що належить даному процесу, і відшукує вільний запис. Оскільки таблиця ще порожня, ядро виявляє структуру з індексом 1 і записує відомості про об'єкт туди. Вказівник встановлюється на внутрішню адресу структури даних об'єкта, маска доступу - на доступ без обмежень і, нарешті, визначається останній компонент – флаги.

От деякі функції, що створюють об'єкти ядра:

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreationFlags,
    PDWORD pdwThreadId);
```

```
HANDLE CreateFile(
    PCTSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);
```

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
```

```
DWORD dwMaximumSizeLow,  
PCTSTR pszName);
```

Всі функції, що створюють об'єкти ядра, повертають числові значення, які прив'язані до конкретного процесу і можуть бути використані в будь-якому потоці даного процесу. Ці значення являють собою індекси у таблиці дескрипторів, що належить процесу, і в такий спосіб ідентифікує місце, де зберігається інформація, пов'язана з об'єктом ядра.

Незалежно від того, як саме створений об'єкт ядра, по закінченні роботи з ним його потрібно закрити викликом `CloseHandle`.

```
BOOL CloseHandle (HANDLE hobj);
```

Ця функція спочатку перевіряє таблицю дескрипторів, що належить процесу, щоб переконатися, чи ідентифікує переданий їй індекс об'єкт, до якого цей процес дійсно має доступ. Якщо переданий індекс правильний, система одержує адресу структури даних об'єкта і зменшує в цій структурі лічильник числа користувачів; як тільки лічильник стане рівним нулю, ядро видалить об'єкт із пам'яті.

Тепер розглянемо модулі системи САД, які так чи інакше підлягають модифікації.

У системі САД існує модуль `Reason`, у його задачі входить верифікація текстів як за допомогою власних засобів так і з використанням зовнішніх. При використанні зовнішніх засобів виникає проблема взаємодії процесів. Така взаємодія на платформі `Windows` може бути організованою за допомогою апарату об'єктів ядра. Саме тому цей модуль системи зазнав найбільших змін в порівнянні з оригінальною версією при переносі на нову платформу.

Такими зовнішніми засобами є програми для доведення тверджень (`prover`). Кожна така програма має власний формат подання вхідних даних. Взаємодія між системою САД та зовнішніми програмами відбувається через стандартні потоки вводу / виводу, хоча можлива організація і інших способів взаємодії (оскільки система є відкритою і допускає розширення). Найпростіша взаємодія описується наступною схемою: модуль `Explore` отримує від модуля `Reason` завдання для доведення, а той в свою чергу передає його в структурованому вигляді зовнішнім засобам. Оскільки, кожний з них має власні вимоги до форми подання вхідних даних, то тут має місце задача трансляції.

Наведемо БНФ, які описують формат завдання для доведення, яке генерує модуль `Explore`:

```

Problem ::= Formula_list
          Proposition
          Number

```

```

Formula_list ::= { '#' | IDN
                  Fof
                  {Formula_list }*
                  }

```

```

Proposition ::= { '_'
                 Fof }
               '? ' Fof

```

```

Fof ::= ' | ' Fof Fof
        '&' Fof Fof
        '>' Fof Fof
        '~' { ':DHD' }
           Fof Fof
        '! ' Fof
        '=' Term Term
        '!=' Term Term
        '@' Term Fof
        '$' Term Fof
        Term

```

```

Term ::= = Func_Symbol '(' ( '{Term {',' Term}* } ')' |
Predicative_Symbol '(' ' Term {',' Term}* ')' | var | '+' | '- '

```

, тут фігурні дужки означають одне входження або відсутність, символи в лапках – термінальні символи, операція * - стандартна операція повторення (ітерація). IDN – ідентифікатор - це послідовність символів, яка починається з букви або символу підкреслення, Predicative_Symbol, Func_Symbol – ідентифікатори, що позначають предикат та функцію відповідно, '+' та '-' скорочені позначення констант істина (true) та хиба (false).

Формули записуються в оберненій польській нотації. Секція Formula_list – це список аксіом. Problem – це запис формул, що підлягає доведенню. Запис ‘_’ Fof1 ‘?’ Fof2 еквівалентний такій формулі: Fof1 => Fof2.

Тут, константи розглядаються, як нуль-арні функції. Наприклад: EmptySet() , Zero(), Jony(), etc.

Число в кінці подання – це обмеження на час, у секундах, який дається для доведення цієї проблеми.

Таке подання допускає трансляцію у вхідні дані для зовнішніх програм. У поточній реалізації використовується зовнішній prover SPASS, та існує відповідний модуль Naigha, який виконує трансляцію та обробку результатів.

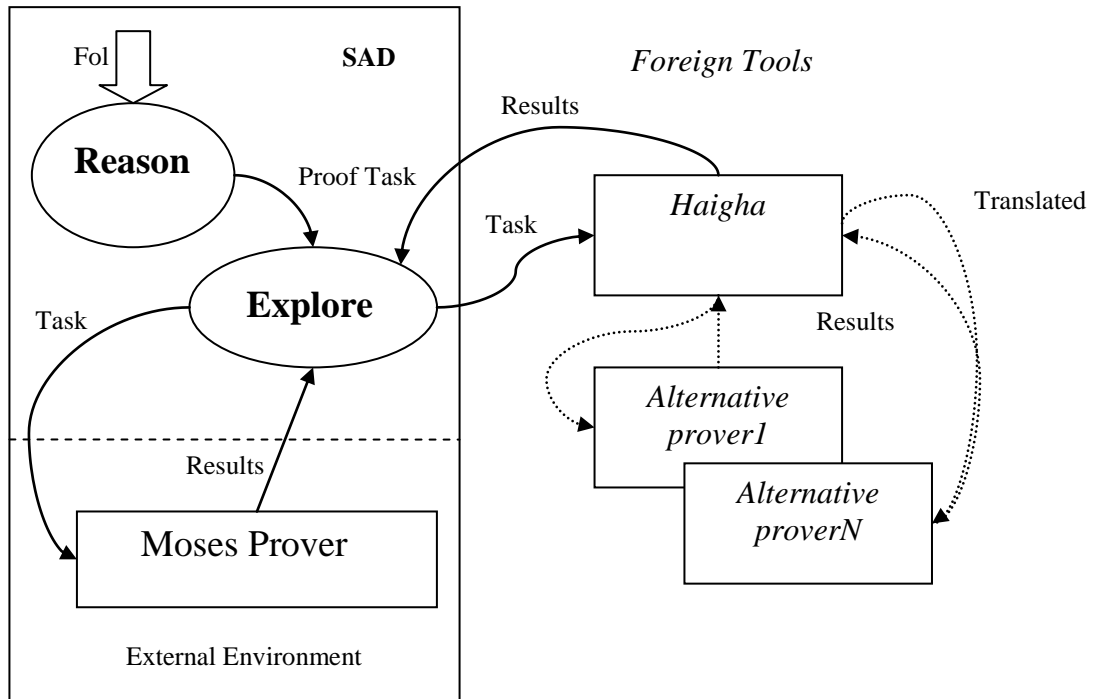
В ході робіт по перенесенню системи було розширено модуль Naigha для роботи з іншими зовнішніми prover-ами, такими як Otter та Vampire. Vampire працює під управлінням ОС Linux. Його підключення реалізовано для відповідної версії системи. Як уже зазначалось підключення такого модуля, полягає в розробці процедури трансляції у формат вхідних даних, процедури коректного запуску prover-а та обробки результатів його роботи, а також у деяких випадках процедур керування обмеженнями (пам'ять, час виконання). Vampire використовує вхідні дані у форматі бібліотеки TPTP (Thousands of Problems for Theorem Proving). Вона є доступною через Інтернет за адресою <http://www.tptp.org>.

Як уже зазначалось вище, при підключенні зовнішніх засобів до системи виникають проблеми узгодження їх взаємодії з модулем Reason, оскільки така взаємодія визначається операційною системою. Оригінальна версія була орієнтована на платформу Linux та стандарт Posix. Платформа Windows має свої альтернативні засоби. Задача узгодження яку необхідно було розв'язати можна звести до класичної задачі „постачальник – споживач”. Нагадаємо її формулювання:



Постачальник виробляє об'єкти і записує їх в буфер обміну. Коли буфер заповнений він не повинен постачати. Споживач не може споживати з порожнього буфера і на обробку одного об'єкту необхідний певний проміжок часу. Розв'язок такої задачі ґрунтується на використанні техніки семафорів.

В нашому випадку взаємодія більш складна. Її можна зобразити такою схемою:



Модуль Reason працює з внутрішнім представленням ForTheL тексту. Він генерує постановку задачі для доведення та передає її підмодулю Explore. В залежності від вхідних параметрів модуль Explore генерує задачу у вище згаданому форматі і передає її або до власного prover-а Moses або до іншої зовнішньої програми, а потім очікує на результати її роботи. В системі САД, Moses реалізований у вигляді окремої зовнішньої програми. Тому взаємодія з ним ні чим не відрізняється від взаємодії з будь-якою зовнішньою програмою. Після завершення роботи prover повинен повернути код завершення у форматі (у тому випадку, коли код завершення має інший формат, модуль що відповідає за підключення, повинен обробляти таку ситуацію і приводити його до коректного вигляду):

‘ / ‘

Код завершення

Код завершення може бути таким:

- ‘+’ – задача успішно доведена
- ‘!’ – задачу довести неможливо
- ‘-’ – досягнуто граничної глибини доведення
- ‘_’ – перевищено ліміт часу

Для підключення альтернативних prover-ів розробниками системи пропонується використовувати модуль Naigha. Його задача полягає в трансляції вхідного формулювання задачі в вхідні дані альтернативного prover-а , обробці результатів його роботи, а потім їх інтерпретація та повернення коду завершення для модуля Explore в коректному форматі.

Розглянемо як реалізується взаємодія між prover-ом та модулем Naigha в системі САД на платформі Windows.

Спочатку модуль Explore запускає програму Naigha, як окремий процес системи. Через канали вводу / виводу (pipes) відбувається передача формулювання проблеми. Далі Naigha проводить трансляцію вхідних даних в формат придатний для обраного prover-а і запускає його як окремий процес системи. З цим процесом пов'язується два потоки WorkerReader та WorkerWriter. Задача першого читати дані з вихідного каналу prover-а та проводити їх обробку. Задача WorkerWriter – записувати дані у вхідний канал prover у необхідному форматі. Канали вводу / виводу відіграють роль буферів між постачальниками даних та їх споживачами. В нашому випадку кожний з процесів може виступати як в ролі постачальника так і в ролі споживача. У тому випадку коли в каналі вводу / виводу не має даних, потік „споживач” призупиняє свою роботи. Аналогічно, якщо буфер каналу вводу / виводу заповнений „виробник” очікує доти поки „споживач” прочитає дані.

Далі приведемо уривок коду програми, який реалізує описану логіку роботи:

```
typedef struct {HANDLE Read;HANDLE Write;} IO_;  
DWORD WINAPI ReaderThreadFunc( LPVOID lpParam ){  
    DWORD bw,br; int c;  
    IO_ ioReader = *((IO_*)lpParam);  
    do{ReadFile(ioReader.Read,&c,1,&br,0); if (!br) break;  
        WriteFile(ioReader.Write,&c,1,&bw,0); } while (bw);  
    CloseHandle(ioReader.Read); CloseHandle(ioReader.Write);  
    return 0;  
}  
DWORD WINAPI WriterThreadFunc( LPVOID lpParam ){  
    DWORD bw,br;int c;IO_ ioWriter = *((IO_*)lpParam);  
    do{ReadFile(ioWriter.Read,&c,1,&br,0);if (!br) break;  
        WriteFile(ioWriter.Write,&c,1,&bw,0); } while (bw);  
    CloseHandle(ioWriter.Write);  
    CloseHandle(ioWriter.Read);  
    return 0;  
}  
int run_WR(char *cmd, HANDLE Input, HANDLE Output){  
    HANDLE fR,fW,fE; PROCESS_INFORMATION pInf;  
    DWORD dwReaderId; DWORD dwReaderExitCode = 0; HANDLE hReaderThread;
```

```

IO_ ioReader; DWORD dwWriterId; DWORD dwWriterExitCode = 0;
HANDLE hWriterThread; IO_ ioWriter;
spawnProc(cmd, &fW, &fR, &fE, &pInf);
ioWriter.Read = Input; ioWriter.Write = fW;
ioReader.Read = fR; ioReader.Write = Output;
hWriterThread=CreateThread(NULL, 0, WriterThreadFunc, &ioWriter, 0, &dwWriterId);
hReaderThread=CreateThread(NULL, 0, ReaderThreadFunc, &ioReader, 0, &dwReaderId);
if (hWriterThread == NULL || hReaderThread == NULL){return GetLastError();}
else {while (WaitForSingleObject(hWriterThread, 500) == WAIT_TIMEOUT) ;
      while (WaitForSingleObject(hReaderThread, 500) == WAIT_TIMEOUT) ;
      CloseHandle( hReaderThread );CloseHandle( hWriterThread ); }
return 0;
};

```

Процедура `SpawnProc` реалізована в іншому модулі. Вона запускає зовнішню програму, задану параметром `cmd`, як окремий процес системи та передає вказівники на канали вводу / виводу, канал помилок та структуру з інформацією про процес.

Тестування системи на платформі Windows проводилось з використанням прикладів оригінальної версії системи, та на спеціально розроблених прикладах, які описуються далі у роботі.

Крім того для зручної роботи з системою розроблено графічний віконний інтерфейс. Він написаний на крос-платформеній мові Java, і тому працює з обома версіями системи. Інтерфейс дозволяє редагувати тексти написані мовою ForTheL, зберігати та зчитувати результати роботи з файлів, а також виконувати операції верифікації, доведення, трансляції з використанням різних зовнішніх `prover`-ів.

Приклади використання мови ForTheL для запису математичних текстів

У цьому розділі ставиться за мету формалізувати основні поняття та визначення теорії множин, доведення теорем та розв'язку задач зі збірника задач [2].

Почнемо з визначення понять. Нові поняття вводяться в мову за допомогою представлень. Розглянемо такі представлення (introducor):

1: [a set/sets]

2: [an element/elements of x]

3: [x is in/from y @ x is an element of y]

4: [x belongs/belong to y @ x is in y]

5: [a subset/subsets of x]

6: [x is empty]

7: [x is non empty]

Перше, друге та п'яте представлення відносяться до першого типу (класовий – іменник). Вони формують поняття множини, елементу множини та підмножини. Третє та четверте представлення – синоніми твердження (x – елемент y), вони формують предикат належності елемента до множини. Шосте й сьоме представлення – прикметники. Вони виражають властивість множини – бути чи не бути порожньою.

Їм відповідають такі примітиви (правила):

```
primClassNoun -> (set|sets) [names]
                | (element|elements) [names] of term
                | (subset|subsets) [names] of term
```

```
primAdjective -> (empty)
                | (non empty)
                | (in | from) term
```

primVerb -> (belongs | belong) (to) term

За допомогою вказаних примітивів ми можемо записати такі аксіоми та визначення:

визначення підмножини

Definition. Let S be a set.

A subset of S is a set with no elements not in S .

визначення порожньої множини

Definition. Let S be a set.

S is empty iff S has no elements.

визначення не порожньої множини

Definition. Let S be a set.

S is non empty iff S is not empty.

аксіома існування порожньої множини

Axiom. There exists an empty set.

визначення рівності двох множин

Definition. Let A, B be a sets.

A is equal to B iff A is a subset of B and B is a subset of A .

Без визначення не порожньої множини можна обійтись, якщо змінити сьоме представлення, таким чином:

[x is non empty @ x is not empty]

Це представлення потрібне для більш зручного написання подальшого тексту.

Також введемо поняття різних множин, для цього додамо одне представлення та визначення:

[x is difers/different from/to B]

Definition *Disequality.* Let A, B be a sets.

A is difers from B iff there exists element of A not in B .

Тепер ми маємо можливість дати визначення основним теоретико-множинним операціям.

Для цього ми скористаємось представленням другого типу – визначений іменник. Він служить для формування функціональних термів. Слід також зауважити, що константа являє собою 0-арну функцію. Така домовленість діє і в мові ForTheL.

Нам буде достатньо визначити три операції:

[the union of x and y]

[the intersection of x and y]

[the substitution of x and y]

А відповідними визначеннями будуть такі:

Definition *DefUnion*. Let A,B be a sets.

The union of A and B is a set X such that for every element y of X y is an element of A or y is an element of B.

Definition *DefIntersection*. Let A,B be a sets.

The intersection of A and B is a set X such that every element of X is in A and in B.

Definition *DefSubst*. Let A,B be a sets.

The substitution of A and B is a set X such that every element of X is in A and not in B.

Маючи такі означення можна сформулювати декілька основних теорем, що виражають властивості вказаних операцій:

1. Транзитивність відношення включення (Якщо $A \subset B$ та $B \subset C$ то $A \subset C$). У термінах введених примітивів, це твердження можна записати таким чином:

Proposition. Let A,B,C be a non empty different sets.

if A is a subset of B and B is a subset of C then A is a subset of C.

2. $A \cap B \subseteq A$:

Proposition. Let A,B be a non empty different sets.

The intersection of A and B is a subset of A.

3. Асоціативність: $A \cap (B \cap C) = (A \cap B) \cap C$:

Proposition. Let A,B,C be a non empty different sets.

The intersection of A and intersection of B and C is equal to intersection of intersection of A and B and C.

Тепер можна спробувати провести верифікацію тексту в on-line версії системи.

Протокол роботи системи виявився наступним:

[ForTheL] parsing successful

[Reason] verification started

[Reason] line 55: goal: if A is a subset of B and B is a subset of C then A is a subset of C.

[MOSES] launch (time limit: 180 sec, initial db: 1, final db: 0)

[MOSES] proved in 266 ms -- proof tree nodes: 38 -- proof tree depth: 4

[MOSES] inference steps: 34647 -- born nodes: 100555 -- depth bound: 4

[Reason] line 59: goal: The intersection of A and B is a subset of A.

[MOSES] launch (time limit: 180 sec, initial db: 1, final db: 0)

[MOSES] proved in 110 ms -- proof tree nodes: 31 -- proof tree depth: 4

[MOSES] inference steps: 17198 -- born nodes: 49762 -- depth bound: 4

[Reason] line 63: goal: The intersection of A and intersection of B and C is equal to intersection of intersection of A and B and C.

[MOSES] launch (time limit: 180 sec, initial db: 1, final db: 0)

[MOSES] proved in 2141 ms -- proof tree nodes: 52 -- proof tree depth: 4

[MOSES] inference steps: 278749 -- born nodes: 604954 -- depth bound: 4

[Reason] verification successful

[Main] session finished in 00:02.93

[Main] 00:00.06 in [ForTheL] -- 00:00.21 in [Reason] -- 00:02.65 in [MOSES]

Таким чином, записаний текст виявився коректним з точки зору системи, а твердження такими, що їх доведення можливе в рамках введених понять та аксіом.

Тепер наведемо переклад частини глави „Формальная арифметика” з книги Е. Мендельсона „Введение в математическую логику” записаної мовою ForTheL:

```

#####
#                               Formal Arithmetic                               #
#                               originally introduced by Elliott Mendelson       #
#                               Introduction to Mathematical Logic              #
#                               D. VAN NOSTRAND COMPANY, INC                    #
#                               PRINCETON NEW JERSEY                          #
#                               TORONTO NEW YORK LONDON                        #
#                               1964                                           #
#                                                                              #
#                               ForTheL translation: Piter Protsyk             #
#                               Kiev National State University                 #
#                               faculty of Cybernetics                        #
#                               started: Ternopil, 1 April 2005               #
#                               last modified: Kiev, 22 April 2005             #
#####
[none]
[a number / numbers]
[x is natural]

[the zero]
[0 @ the zero]

[the successor of x]
[the multiplication of x and y]
[the mul of x and y @ the multiplication of x and y]
[the sum of x and y]

[x' @ the successor of x]
[x * y @ the multiplication of x and y]
[x+y @ the sum of x and y]

#Now, we are ready to formulate some basic axioms of
#natural numbers

Axiom P0. 0 is natural number.
Axiom P1. The successor of every natural number is natural number.
Axiom P2. The sum of every natural number and every natural number is natural number.
Axiom P3. The mul of every natural number and every natural number is natural number.
Axiom P4. For every natural number N N = 0 or
           there exists a natural number M such that N = M'.

# In first-order logic we cannot give a finite axiomatization of well-foundedness.
# Therefore ForTheL provides a preintroduced predicate symbol -<- witch is used
# as an abstract well-founded relation in induction arguments.
# To simulate the induction on natural numbers we define following axioms:

Axiom IndOrdering. For every natural number N N -<- N'.

# The following axioms discribes our theory:

Axiom S1. For all natural number A,B,C           A = B => (A = C => B = C).

```

```

Axiom S2. For all natural number A,B,C      A = B => A' = B'.
Axiom S3. For all natural number A          A'!=0.
Axiom S4. For all natural number A,B,C      A' = B' => A = B.
Axiom S5. For all natural number A          A + 0 = A.
Axiom S6. For all natural number A,B,C      A + (B') = (A+B)'.
Axiom S7. For all natural number A          A * 0 = 0.
Axiom S8. For all natural number A,B        A *(B')= (A*B) + A.

#Axiom S9. A(0) => (all x A(x) => A(x')) => all x A(x) ,
#       where A is any formula of the theory
# We don't need this axiom because it implemented as induction principle of SAD,
# see ForTheL language reference for details.

[a term/terms @ natural number]
#####
#Every arithmetic term represent some natural number.      #
#So we can say that every term is a representation of some natural number.      #
#Also we define three functions on natural numbers. They are the successor,      #
#the sum of two numbers and the multiplication.            #
#####
# Propostions L1-L8 directly derived from axioms S1-S8:

Proposition L1. For all terms t,r,s      t=r => (t=s => r=s) (by S1).
Proposition L2. For all terms t,r        t=r => t' = r' (by S2).
Proposition L3. For all term t           0 != t' (by S3).
Proposition L4. For all terms t,r        t'= r' => t = r (by S4).
Proposition L5. For all term t           t+0 = t (by S5).
Proposition L6. For all terms t,r        t+(r')=(t+r)' (by S6).
Proposition L7. For all term t           t*0 = 0 (by S7).
Proposition L8. For all terms t,r        t*(r')=(t*r)+t (by S8).

# Some basic properties of natural number terms:

Proposition P_a. For all term t          t=t.
Proposition P_b. For all terms t,r       t=r => t'=r'.
Proposition P_c. For all terms t,r,s     t=r => (r=s => t=s).
Proposition P_d. For all terms t,r,s     r=t => (s=t => r=s).
Proposition P_e. For all terms t,r,s     (t=r) => (t+s = r+s).

Proposition P_f. For all term t          t=0+t.
Proof by induction on t.
  Let t be a term.
  Case t = 0.
    0+0 = 0 (by L5).
  end.
  Case t!=0.
    Take a term x such that x' = t.
    x = 0 + x. # Hypothesis!
    (0+(x')) = (0+x)' (by L6,P0,P1,P2,P3).
    x' = (0+x)' (by L2,P0,P1,P2,P3).
    x' = 0 + (x') (by P_d).

```

$x = 0 + x \Rightarrow x' = 0 + (x')$.
 end.
 end.

Proposition P_g. For all terms t, r $(t') + r = (t+r)'$.
 Proof by induction on r .
 Let t, r be terms.
 Case $r = 0$.
 $(t') + 0 = t'$ (by L5, P0, P1, P2, P3).
 $t + 0 = t$ (by L5, P0, P1, P2, P3).
 $(t + 0)' = t'$ (by L2, P0, P1, P2, P3).
 $(t') + 0 = (t + 0)'$ (by P_d).
 end.
 Case $r \neq 0$.
 Take a term x such that $x' = r$.
 $(t') + x = (t + x)'$. # Hypothesis!
 $(t') + (x') = ((t') + x)'$ (by L6, P0, P1, P2, P3).
 $((t') + x)' = (t+x)''$ (by L2, P0, P1, P2, P3).
 $((t') + (x')) = (t+x)''$ (by P_c).
 $(t+(x')) = (t+x)'$ (by L6, P0, P1, P2, P3).
 $(t+(x'))' = (t+x)''$ (by L2, P0, P1, P2, P3).
 $(t')+(x') = (t+(x'))'$ (by P_d).
 Hence $((t')+x) = (t+x)' \Rightarrow (t') + (x') = (t+(x'))'$.
 end.
 end.

Proposition P_h. For all terms t, r $t+r=r+t$.
 Proof by induction on r .
 Let t, r be terms.
 Case $r = 0$.
 $r + 0 = r$ (by L5, P0, P1, P2, P3).
 $r = 0 + r$ (by P_f).
 $r+0 = 0+r$ (by P_c).
 end.
 Case $r \neq 0$.
 Take a term x such that $x' = r$.
 $t + x = x + t$. #By induction hypothesis!
 $t + (x') = (t + x)'$ (by L6, P0, P1, P2, P3).
 $(x')+t = (x + t)'$ (by P_g).
 $(t+x)' = (x+t)'$ (by L2, P0, P1, P2, P3).
 $t+(x') = (x + t)'$ (by P_c).
 $t+(x') = (x') + t$ (by P_d).
 Hence $t+x = x+t \Rightarrow t + (x') = (x') + t$.
 end.
 end.

Proposition P_i. For all terms t, r, s $t=r \Rightarrow s+t=s+r$.
 Proposition P_j. For all terms t, r, s $(t+r)+s=t+(r+s)$.
 Proof by induction on s .
 Let t, r, s be terms.
 Case $s=0$.
 $(t+r) + 0 = t+r$ (by L5, P0, P1, P2, P3).

$r+0 = r$ (by L5,P0,P1,P2,P3).
 $t+(r+0) = t+r$ (by P_i).
 $(t+r)+0 = t+(r+0)$.
end.
Case $s!=0$.
Take a term z such that $z' = s$.
 $(t+r)+z = t+(r+z)$. #By induction hypothesis
 $(t+r)+(z') = ((t+r)+z)'$ (by L6,P0,P1,P2,P3).
 $((t+r)+z)' = (t+(r+z))'$ (by L2,P0,P1,P2,P3).
 $(t+r)+(z') = (t+(r+z))'$ (by P_c).
 $r+z' = (r+z)'$ (by L6,P0,P1,P2,P3).
 $t+(r+(z')) = t+(r+z)'$ (by P_i).
 $t+(r+z)'$ = $(t+(r+z))'$ (by L6,P0,P1,P2,P3).
 $t+(r+(z')) = (t+(r+z))'$ (by P_d).
 $(t+r)+(z') = t+(r+(z'))$ (by P_d).
Hence $(t+r)+z = t+(r+z) \Rightarrow (t+r)+(z') = t+(r+(z'))$.
end.

end.

Proposition P_k. For all terms t, r, s $t=r \Rightarrow t*s=r*s$.

Proposition P_l. For all term t $0*t=0$.

Proof by induction on t .

Let t be a term.

Case $t=0$.

$0*0 = 0$ (by L7,P0,P1,P2,P3).

end.

Case $t!=0$.

Take a term x such that $x' = t$.

$0*x = 0$. #By induction hypothesis.

$0*(x') = (0*x) + 0$ (by L8,P0,P1,P2,P3).

$(0*x)+0 = 0$ (by L5,P0,P1,P2,P3).

Hence $0*x = 0 \Rightarrow 0*(x') = 0$.

end.

end.

Proposition P_m. For all terms t, r $(t')*r=(t*r)+r$.

Proof by induction on r .

Let t, r be terms.

Case $r = 0$.

$(t')*0 = 0$ (by L7,P0,P1,P2,P3).

end.

Case $r!=0$.

Take a term x such that $x' = r$.

$(t')*x = (t*x)+x$. #By hypothesis

$(t')*(x') = ((t')*x)+(t')$ (by L8,P0,P1,P2,P3).

$((t')*x)+(t') = ((t*x)+x)+(t')$ (by P_d).

$((t*x)+x)+(t') = (t*x) + (x+(t'))$.

$(t*x) + (x+(t')) = (t*x) + ((x+t)')$.

$((t*x)+(x+t))' = (t*x) + ((x')+t)$.

$(t*x)+((x')+t) = (t*x)+(t+(x'))$.

$(t*x)+(t+(x')) = ((t*x)+t)+(x')$.

$((t*x)+t)+(x') = (t*(x')) + (x')$ (by L8, P0,P1,P2,P3).

Hence $(t')*x = (t*x)+x \Rightarrow (t')*(x') = (t*(x'))+(x')$.

end.

end.

Proposition P_n. For all terms t, r $t*r=r*t$.

Proof by induction on r .

Let t, r be terms.

Case $r = 0$.

$t*0 = 0$ (by L7, P0, P1, P2, P3).

$0*t = 0$ (by P_1).

end.

Case $r \neq 0$.

Take a term x such that $x' = r$.

$t*x = x*t$. #By hypothesis.

$t*(x') = (t*x)+t$ (by L8, P0, P1, P2, P3).

$(t*x)+t = (x*t)+t$.

$(x*t)+t = (x')*t$ (by P_m).

Hence $t*x = x*t \Rightarrow t*(x') = (x')*t$.

end.

end.

Proposition P_o. For all terms t, r, s $t=r \Rightarrow s*t = s*r$.

A number of important properties of addition and multiplication are given

in following propositions:

Proposition P_p. For all terms r, s $(s=0 \text{ and } r+s =r)$

or

there exists term x such that $(x'=s)$ and $((r+s) = (r+x'))$.

Proposition P_3_4_a. For all terms t, r, s $t*(r+s) = (t*r) + (t*s)$.

Proof by induction on s .

Let t, r, s be terms.

Case $s = 0$.

$r+0 = r$ (by L5).

$t*(r+0) = t*r$.

$t*r = t*r + 0$.

$0 = t*0$ (by L7).

$t*r = t*r + t*0$.

$t*(r+0) = t*r + t*s$.

end.

Case $s \neq 0$.

Take a term x such that $x' = s$.

Then $t*(r+s) = t*(r+x')$ (by P_p).

$r+x' = (r+x)'$ (by L6).

$t*(r+x') = t*((r+x)')$ (by P_g, P_h).

$t*((r+x)') = (t*(r+x))+t$.

$t*(r+x) = (t*r) + (t*x)$.

$(t*(r+x))+t = ((t*r)+(t*x))+t$.

$((t*r)+(t*x))+t = (t*r)+((t*x)+t)$ (by P_j, P2, P3).

$(t*x)+t = t*(x')$ (by L8).

Hence $t*(r+x') = (t*r) + (t*x')$.

end.

end.

Proposition P_3_4_b. For all terms t, r, s $(r+s)*t = (r*t) + (s*t)$.

Proposition P_3_4_c. For all terms t, r, s $(t*r)*s = t*(r*s)$.

Proof by induction on s .

```

Let t,r,s be terms.
Case s = 0.
  (t*r)*0 = 0 (by L7,P0,P1,P2,P3,P4).
  r*0 = 0 (by L7,P0,P1,P2,P3,P4).
  t*(r*0) = 0 (by L7,P0,P1,P2,P3,P4).
end.
Case s!=0.
  Take a term x such that x' = s.
  (t*r)*x = t*(r*x). #By induction hypothesis
  (t*r)*(x')=((t*r)*x) + (t*r) (by L8,P0,P1,P2,P3,P4).
  ((t*r)*x)+(t*r) = (t*(r*x)) + (t*r).
  (t*(r*x))+(t*r) = t*((r*x)+r) (by P_3_4_a,P0,P1,P2,P3,P4).
  Hence t*((r*x)+r) = t*(r*(x')) (by L8,P0,P1,P2,P3,P4).
end.
end.
Proposition P_3_4_d. For all terms t,r,s t+s = r+s => t = r.
Proof by induction on s.
Let t,r,s be terms.
Case s = 0.
  t+0 = t.
  r+0 = r.
  Hence t+0 = r+0 => t=r.
end.
Case s!=0.
  Take a term x such that x' = s.
  t+x = r+x => t = r.
  t+x' = (t+x)'.
  r+x' = (r+x)'.
  (t+x)' = (r+x)' => t+x =r+x => t = r.
  Hence t+(x')=r+(x') => t = r.
end.
end.
[the one @ 0'] [1 @ the one] [the two @ 1'] [2 @ the two]
[the three @ 2'] [3 @ the three] [the four @ 3'] [4 @ the four]
[the five @ 4'] [5 @ the five] [the six @ 5'] [6 @ the six]
[the seven @ 6'] [7 @ the seven] [the eight @ 7'] [8 @ the eight]
[the nine @ 8'] [9 @ the nine] [the ten @ 9'] [10 @ the ten]
Proposition P_3_5_a. For all term t t + 1 = t'.
Proposition P_3_5_b. For all term t t * 1 = t.
Proposition P_3_5_c. For all term t t * 2 = t+t.
[none]
# Proving of this example is based on induction principle.
# But we cannot write the proofs because induction provides on formula.
# In current release induction is based only on terms
Proposition P_3_5_d. For all term t,s t +s = 0 => (t = 0) and (s = 0).
Proposition P_3_5_e. For all term t,s t != 0 => (s*t = 0 => s = 0).
Proposition P_3_5_f. For all term t,s (t+s = 1) => (t = 0 and s = 1) or (t=1 and s=0) .
Proposition P_3_5_g. For all term t,s (t*s = 1) => (t = 1 and s = 1).
Proposition P_3_5_h. For all term t if (t!=0) then there exists term y such that t = y'.
Proposition P_3_5_i. For all term t,s,r s!=0 => (t*s = r*s => t = r).
Proposition P_3_5_j. For all term t if (t!=0) then t is not equal to one and

```

there exists term y such that t is equal to y' .

[full]

[x is less than y]

[x is less or equal y]

[x is greater than y @ y is less than x]

[x is greater or equal y @ y is less or equal y]

[$x < y$ @ x is less than y]

[$x \leq y$ @ x is less or equal y]

[$x > y$ @ x is greater than y]

[$x \geq y$ @ x is greater or equal y]

Definition DefLess.

Let t, s be terms.

t is less than s iff there exists a term w such that w is not equal to zero and $t+w = s$.

Definition DefLessOrEqual.

Let t, s be terms.

t is less or equal s iff $t < s$ or $t = s$.

Proposition P_3_7_a. For every term t t is not less than t .

Proposition P_3_7_b. For every terms t, s, r $t < s \Rightarrow (s < r \Rightarrow t < r)$.

Proof.

Let t, s, r be terms.

Suppose $t < s$ and $s < r$.

Then (there exists a term w such that $w \neq 0$ and $t+w = s$) and

(there exists a term v such that $v \neq 0$ and $s+v = r$) (by DefLess).

Take a term w such that $t+w = s$.

Take a term v such that $s+v = r$.

$(t+w)+v = r$.

$t+(w+v) = r$.

$w \neq 0$ and $v \neq 0$.

$w+v \neq 0$.

$w+v \neq 0$ and $t+(w+v) = r$.

Then there exists a term u such that $u \neq 0$ and $t+u = r$.

Hence $t < r$.

Then $t < s \Rightarrow (s < r \Rightarrow t < r)$.

end.

Proposition P_3_7_c. For every terms t, s $t < s \Rightarrow$ not $s < t$.

Proposition P_3_7_d. For every terms t, s, r $t < s \Leftrightarrow t+r < s+r$.

Proposition P_3_7_e. For every term t t is less or equal t .

Proposition P_3_7_f. For every terms t, s, r $t \leq s \Rightarrow (s \leq r \Rightarrow t \leq r)$.

Proposition P_3_7_g. For every terms t, s, r $t \leq s \Rightarrow (t+r \leq s+r)$.

Proposition P_3_7_h. For every terms t, s, r $t \leq s \Rightarrow (s < r \Rightarrow t < r)$.

Proposition P_3_7_i. For every term t $0 \leq t$.

Proposition P_3_7_j. For every term t $0 < t'$.

Proposition P_3_7_k. For every terms t, r $t < r \Leftrightarrow t' \leq r$.

Proposition P_3_7_l. For every terms t, s, r $t \leq r \Leftrightarrow t < r'$.

Proposition P_3_7_m. For every term t $t < t'$.

Proposition P_3_7_n. $(0 < 1)$ and $(1 < 2)$ and $(2 < 3)$ and $(3 < 4)$ and $(4 < 5)$ and $(5 < 6)$ and $(6 < 7)$ and $(7 < 8)$ and $(8 < 9)$ and $(9 < 10)$.

Proposition P_3_7_o. For every terms t, r $t \neq r \Rightarrow (t < r$ or $r < t)$.

Proposition P_3_7_oo. For every terms t, r $(t = r)$ or $(t < r)$ or $(r < t)$.

Proposition P_3_7_p. For every terms t, r $(t \leq r)$ or $(r \leq t)$.

Proposition P_3_7_q. For every terms t, r $t+r \geq t$.

Proposition P_3_7_r. For every terms t, r if $r \neq 0$ then $t+r > 0$.
 Proposition P_3_7_s. For every terms t, r if $r \neq 0$ then $t*r \geq t$.
 Proposition P_3_7_t. For every terms r if $r \neq 0$ then $r > 0$.
 Proposition P_3_7_u. For every terms t, r $r > 0 \Rightarrow (t > 0 \Rightarrow r*t > 0)$.
 Proposition P_3_7_v. For every terms t, r $r \neq 0 \Rightarrow (t > 1 \Rightarrow t*r > r)$.
 Proposition P_3_7_w. For every terms t, s, r $r \neq 0 \Rightarrow (t < s \Leftrightarrow t*r < s*r)$.
 Proposition P_3_7_x. For every terms t, s, r $r \neq 0 \Rightarrow (t < s \Leftrightarrow t*r =< s*r)$.
 Proposition P_3_7_y. For every terms t it is wrong that $t < 0$.
 Proposition P_3_7_z. For every terms t, r if $(t < r)$ and $(r =< t)$ then $t = r$.

[x divide y]

[x | y @ x divide y]

Definition DefDivide.

Let t, s be terms.

t divide s iff there exists a term z such that $s = t*z$.

Proposition P_3_10_a. For all term t $t | t$.

Proposition P_3_10_b. For all term t $1 | t$.

Proposition P_3_10_c. For all term t $t | 0$.

Proposition P_3_10_d. For all terms t, s, r If $t | s$ and $s | r$ then $t | r$.

Proposition P_3_10_e. For all terms t, s If $s \neq 0$ and $t | s$ then $t =< s$.

Proposition P_3_10_f. For all terms t, s if $t|s$ and $s|t$ then $s = t$.

Proposition P_3_10_g. For all terms t, s, r If $t | s$ then $t | r*s$.

Proposition P_3_10_h. For all term t, s, r If $t|s$ and $t|r$ then $t | s + r$.

#Please try to write formal proofs of this exercises.

Proposition Exercise_1. For all term t If $t | 1$ then $t = 1$.

Proposition Exercise_2. For all terms t, s $(t | s \Rightarrow t | s') \Rightarrow t = 1$.

[none]

Proving of this example is based on induction principle.

But we cannot write the proofs because induction provides on formula.

In current release induction is based only on terms

Axiom P_3_11. For all term x, y If $y \neq 0$ then there exists terms u, v such that $((x = y*u + v$ and $v < y)$ and there exists terms m, n such that if $(x = y*m + n$ and $n < y)$ then $(u = m$ and $n = v)$).

[full]

Висновки

Метою даної роботи ставилось досягнення двох ключових цілей:

1. Перенесення системи САД з платформи під управлінням ОС Linux на платформу MS Windows
2. Використання вхідної мови системи ForTheL для написання формальних математичних текстів.

Система САД – розробляється в рамках робіт по алгоритму очевидності. Важливою складовою системи є мова ForTheL. Вона є формальною мовою і дуже близькою до природної англійської мови. В рамках даної роботи були виконані переклади реальних математичних текстів на мову системи. Як результат, були показані виразні можливості мови, а також вироблена певна методика по перекладу математичних текстів на формалізовану мову. Вдалося перекласти значну частину розділу „Формальна арифметика” з книги Е. Мендельсона „Введение в математическую логику”.

Не менш важливою частиною роботи є перенесення системи з платформи Linux на платформу MS Windows та створення персональної версії. Тобто, такої що не потребує доступу до мережі Інтернет, не вимагає додаткових налаштувань та має зручний графічний інтерфейс користувача.

Система була успішно перенесеною на платформу Windows та пройшла тестування як на прикладах з оригінальної версії системи так і на розроблених прикладах. Також були додані додаткові зовнішні засоби, що дало змогу оцінити їх продуктивність при використанні в системі.

Для зручної роботи з системою було створено віконний графічний інтерфейс, який дає доступ до всіх основних функцій системи: пошук виводу, верифікація текстів, доведення теорем та використання для цих цілей, як власних засобів так і різноманітних зовнішніх.

Таким чином поставлені в рамках роботи цілі були досягнені.

Подальший розвиток системи буде полягати в розширенні мови за допомогою додаткових конструкцій, створення бази знань, покращення засобів пошуку виводу, створення інтелектуальних графічних інтерфейсів користувача.

Література

1. Andrei Paskevich, Formal Theory Language (the reference). (<http://ea.unicyb.kiev.ua>)
2. Лавров И.А., Максимова Л. Л. Задачи по теории множеств, математической логике и теории алгоритмов. – 3-е изд. – М.: Физматлит, 1995. – ISBN 5-02-01 4844 – X.
3. Мендельсон Э. Введение в математическую логику: Пер. с англ. / Под ред. С.И. Адяна. – 3-е изд. – М.: Наука. Главная редакция физико-математической литературы, 1984. – 320 с.
4. Ершов Ю.Л., Палютин Е.А. Математическая логика: Учеб. Пособие для вузов. – 2-е изд., испр. и доп.. – М.: Наука, 1987. – 336 с.
5. Ю.В. Капітонова, С.Л. Кривий, О.А. Летичевський, Г.М. Луцький, М.К. Печурін. Основи дискретної математики. – підручник -: К.: Наукова думка, 2002
6. А. Роббинс. Linux программирование в примерах. – М.: Кудиц-образ, 2005.
7. Г. Шилдт. Полный справ очник по С. – 4-е изд. – М.: Издательский дом „Вільямс”, 2002.
8. П. Вейнер. Языки программирования Java и JavaScript. – изд. «Лори», 2001.
9. Arjan van IJzendoorn. Tour of the Haskell Syntax.
10. Hal Daume, Yet Another Haskell Tutorial.
11. Д. Рихтер. Создание эффективных WIN32 приложений. – С-П.: Питер, 2002
12. С. Weidenbach. Spass Input Syntax - Max-Planck-Institut fur Informatik
13. A.V. Lyaletski. Evidential paradigm: the logical aspect, - Cybernetics and System Analysis, Vol 39, No. 5, 2003