

**ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА
КІБЕРНЕТИКИ КИЇВСЬКОГО
НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ
ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Криволап А.В.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до курсу «Програмні логіки та їх застосування»

Київ - 2022

УДК 004.42

Криволап А.В. Методичні рекомендації до курсу «Програмні логіки та їх застосування» . – Київ, 2022. – 34 с.

Навчальний посібник призначений для студентів освітньої програми Інформатика за спеціальністю 122 «Комп'ютерні науки», які проходять курс «Програмні логіки та їх застосування» або споріднений, в якому розглядаються питання застосування програмних логік та систем допомоги доведення теорем до специфікації та верифікації програмних систем.

В посібнику надаються приклади застосування логіки Флойда-Хоара та інших програмних логік та використання системи Isabelle для доведення властивостей програмних систем.

Рецензенти А.Ю.Дорошенко, доктор фіз.-мат.наук
С.С.Шкільняк, доктор фіз.-мат.наук

*Затверджено Вченою радою
факультету комп'ютерних наук та кібернетики Протокол № 8 від 12
квітня 2022р.*

© А.В. Криволап, 2022

ЗМІСТ

| | |
|--|----|
| ВСТУП..... | 4 |
| 1. ЛОГІКА ФЛОЙДА-ХОАРА..... | 7 |
| 1.1. Номінативні дані та функції..... | 7 |
| 1.2. Програмна алгебра..... | 8 |
| 1.3. Аксиоматична система..... | 11 |
| 1.4. Приклади..... | 12 |
| 1.5. Вправи до розділу..... | 16 |
| 2. ЛОГІКА РОЗДІЛЕННЯ..... | 17 |
| 3. ДИНАМІЧНА ЛОГІКА | 20 |
| 3.1. Регулярні програми..... | 20 |
| 3.2. Семантика динамічних логік..... | 21 |
| 3.3. Вправи до розділу..... | 22 |
| 4. СИСТЕМА ISABELLE..... | 23 |
| 4.1. Теорії Isabelle..... | 23 |
| 4.2. Типи Isabelle..... | 24 |
| 4.3. Типи та функції визначені користувачем..... | 26 |
| 4.4. Леми та теореми..... | 28 |
| 4.5. Методи доведення..... | 29 |
| 4.6. Вправи до розділу..... | 32 |
| СПИСОК ЛІТЕРАТУРИ..... | 34 |

ВСТУП

Програмні логіки[1] є однією з математичних основ сучасних систем автоматичного доведення теорем. Вони дозволяють вирішувати проблему специфікації та верифікації програмних систем. Потреба у врахуванні різних аспектів сучасного програмного забезпечення призвела до появи цілого спектру програмних логік. Використання засобів програмного рівня разом з функціональним та предикатним дозволяє описувати властивості програм та верифікувати їх коректність.

Однією з перших програмних логік є логіка Флойда-Хоара[2], розгляду якої присвячено перший розділ даного посібника. Використання поняття трійки Хоара — предиката, що складається з передумови, програми та післяумови разом з аксіоматичною системою для імперативної мови програмування дозволило ефективно задавати програмні специфікації та верифікувати їх. Даний результат був розвинений Дейкстрою[3], який запропонував поняття перетворювачів предикатів, одним з яких є найслабкіша передумова. Наявність формул для обчислення найслабкішої передумови дозволило звести проблему доведення істинності формули програмної логіки до проблеми доведення істинності формули першого порядку. Таким чином уможливіючи використання вже існуючих засобів доведення теорем.

Поява поняття вказівників, динамічної пам'яті і їх широке використання в сучасних мовах програмування призвело до потреби пошуку відповідних формальних засобів. Адже значна частина помилок та проблем з продуктивністю в програмних системах пов'язана з роботою з динамічною пам'яттю. Саме тому проблема доведення коректності алгоритмів роботи з вказівниками, вивільненням пам'яті є дуже актуальним. Логіка розділення[4], що розглядається в другому розділі даного посібника є одним з мож-

ливих розв'язків даної проблеми. Визначена саме як розширення логіки Флойда-Хоара вона доводить релевантність підходу з використанням трійок Хоара. Розширення відбувається за рахунок використання спеціальної конструкції, що задає область динамічної пам'яті та спеціальних логічних зв'язок. Ці зв'язки дозволяють задавати твердження, що виконуються на ділянках пам'яті, що не перетинаються.

Розвиток розподілених систем та модальних логік призвів до появи динамічної логіки[5], яка є переосмисленням логіки Флойда-Хоара. В динамічній логіці програмний аспект задається за допомогою спеціального модального оператора, що позначає той факт, що після виконання певної дії має виконуватись відповідне твердження. Динамічна логіка розглядається в третьому розділі посібника. Модальності динамічної логіки за своєю суттю подібні до перетворювачів предикатів Дейкстри. Динамічні логіки також були досліджені на різних рівнях — пропозиційні динамічні логіки, динамічні логіки першого порядку та їх різноманітні розширення. Таким чином використання модальних логік дозволило замінити спеціальну композицію програмного рівня, що задає трійку Хоара на модальність, а інші композиції такі як послідовне виконання та цикл задавати за допомогою спеціальної мови дій. Це дозволило використовувати потужний апарат регулярних виразів та семантики можливих світів Кріпке для модальних логік.

Також варто зазначити інші розширення логіки Флойда-Хоара[6,7] та різноманітні числення[8-13]. Серед них застосування темпоральних логік для опису властивостей програм; числення процесів, що дозволяють описувати розподілені системи та алгоритми; логіка співставлення, що продовжує ідеї логіки розділення, використовуючи також поняття шаблону та співставлення виконання програми та шаблонів. Окремо слід виділити числення процесів направлені на опис та доведення властивостей розподілених систем.

В заключному, четвертому розділі розглядається система Isabelle[14] як

приклад системи автоматизованого доведення теорем, що активно розвивається в академічних колах та має широкий набір існуючих теорій. Дана система базується на логіці вищого порядку, що дозволяє формувати широкий клас властивостей а також має вбудовану функціональну мову програмування. В розділі описано базові можливості системи, та синтаксис, необхідний для формулювання та доведення базових лем та теорем. Також наводяться відповідні приклади .

1. ЛОГІКА ФЛОЙДА-ХОАРА

Логіку Флойда-Хоара визначимо використовуючи композиційно-номінативний підхід[15]. Для цього спочатку визначимо відповідну алгебру. Синтаксис в такому випадку буде визначатись термами алгебри. Також наведемо коректну аксіоматичну систему для простої імперативної мови програмування[13] та формули для перетворювача предикатів найслабкіша передумова.

1.1 Номінативні дані та функції

Дані будемо розглядати як номінативні дані, тобто дані як зв'язок між іменем змінної та її значенням. Для цього зафіксовано наступні множини: V – множина імен змінних, A – множина можливих значень змінних. В такому випадку простою іменною(номінативною) множиною будемо називати часткове відображення $V \xrightarrow{p} A$. А класом номінативних даних відповідно будемо називати клас часткових відображень ${}^V A = V \xrightarrow{p} A$. Важливою для подальших визначень є бінарна операція накладання, що дозволяє природним чином задавати важливе в сучасних мовах програмування поняття присвоєння $d_1 \nabla d_2 = [v \mapsto a \mid d_2(v) \downarrow = a \vee (d_1(v) \downarrow = a \wedge d_2(v) \uparrow)]$.

Визначимо також функції над номінативними даним, які будуть основами програмної алгебри, що задаватиме семантику програмної логіки Флойда-Хоара.

Для подання семантики умов в програмах будемо використовувати часткові предикати над простими іменними множинами, $Pr^{V,A} : {}^V A \xrightarrow{p} \{T, F\}$.

Семантику виразів в програмах будемо подавати за допомогою часткових функцій з іменних множин в множину базових значень $Fn^{V,A} : {}^V A \xrightarrow{p} A$.

Семантику програм та їх операторів будемо подавати використовуючи часткові функції над простими іменними множинами(що задають стани

програми), $FPr^g^{V,A} : {}^V A \xrightarrow{p} {}^V A$.

1.2 Програмна алгебра

Для подання семантики будемо використовувати алгебру наступного вигляду:

$$QPA(V, A) = \langle Pr^{V,A}, Fn^{V,A}, FPr^g^{V,A}; \vee, \neg, \{S_P^{\bar{v}}\}_{\bar{v} \in U(V)}, \{S_F^{\bar{v}}\}_{\bar{v} \in U(V)}, \{x\}_{x \in V}, \{\exists x\}_{x \in V}, id, \{AS^x\}_{x \in V}, \bullet, IF, WH, FH, PC \rangle$$

Операції предикатного рівня включають в себе композицію диз'юнкції, заперечення та композиція екзистенціальної квантифікації, їх визначення приведені далі.

Бінарна композиція диз'юнкції \vee має тип $Pr^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$, та визначається так, де $p \in Pr^{V,A}, q \in Pr^{V,A}, d \in {}^V A$:

$$(p \vee q)(d) = \begin{cases} T, & \text{якщо } p(d) \downarrow = T \text{ або } q(d) \downarrow = T, \\ F, & \text{якщо } p(d) \downarrow = F \text{ та } q(d) \downarrow = F, \\ \text{невизначено} & \text{інакше.} \end{cases}$$

Унарна композиція заперечення \neg має тип $Pr^{V,A} \xrightarrow{t} Pr^{V,A}$, та визначається наступним чином, де $p \in Pr^{V,A}, d \in {}^V A$:

$$(\neg p)(d) = \begin{cases} T, & \text{якщо } p(d) \downarrow = F, \\ F, & \text{якщо } p(d) \downarrow = T, \\ \text{невизначено} & \text{інакше.} \end{cases}$$

Унарна параметрична композиція екзистенціальної квантифікації $\exists x$ з параметром $x \in V$ має тип $Pr^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$, та визначається наступним чином, де $p \in Pr^{V,A}, q \in Pr^{V,A}, d \in {}^V A$:

$$(\exists x p)(d) = \begin{cases} T, & \text{якщо } p(d \nabla [x \mapsto a]) \downarrow = T \text{ для деякого } a \in A, \\ F, & \text{якщо } p(d \nabla [x \mapsto a]) \downarrow = F \text{ для всіх } a \in A, \\ \text{невизначено} & \text{інакше.} \end{cases}$$

На предикатно-функціональному рівні розглядаються композиції суперпозиції для предикатів та функцій а також параметрична функція розіменування.

Композиція суперпозиції для функцій $S_F^{x_1, x_2, \dots, x_n}$ є $n + 1$ -арною параметричною композицією з параметрами $x_1, x_2, \dots, x_n \in V$, має тип $(Fn^{V,A})^{n+1} \xrightarrow{t} Fn^{V,A}$, та визначається наступним чином, де $f, g_1, g_2, \dots, g_n \in Fn^{V,A}, d \in {}^V A$:

$$S_F^{x_1, x_2, \dots, x_n}(f, g_1, g_2, \dots, g_n)(d) \simeq f(d \nabla [x_1 \mapsto g_1(d), x_2 \mapsto g_2(d), \dots, x_n \mapsto g_n(d)])$$

.

Композиція суперпозиції для предикатів $S_P^{x_1, x_2, \dots, x_n}$ є $n + 1$ -арною параметричною композицією з параметрами $x_1, x_2, \dots, x_n \in V$, має тип $Pr^{V,A} \times (Fn^{V,A})^n \xrightarrow{t} Pr^{V,A}$, та визначається наступним чином, де $p \in Pr^{V,A}, g_1, g_2, \dots, g_n \in Fn^{V,A}, d \in {}^V A$:

$$S_P^{x_1, x_2, \dots, x_n}(p, g_1, g_2, \dots, g_n)(d) \simeq p(d \nabla [x_1 \mapsto g_1(d), x_2 \mapsto g_2(d), \dots, x_n \mapsto g_n(d)])$$

.

Нуль-арна параметрична композиція розіменування $\backslash x$ з параметром $x \in V$, $\backslash x \in Fn^{V,A}$, якщо $d \in {}^V A$ то значення функції що задається композицією визначається як значення відповідної змінної для даного d , якщо вона визначена:

$$\backslash x(d) \simeq d(x)$$

На програмному рівні розглядаються композиції, що відповідають основним програмним операторам мови, таким як умовний оператор, оператор циклу, оператор послідовного виконання, оператор присвоєння. До них також додаємо композицію, що задає трійку Флойда-Хоара а також композиція побудови умови за прообразом, яка дозволяє задати перетворювач предикатів найслабкіша передумова.

Унарна параметрична композиція присвоєння AS^x з параметром $x \in V$ має тип $Fn^{V,A} \xrightarrow{t} FPr_g^{V,A}$, та визначається наступним чином, де $f \in Fn^{V,A}, d \in {}^V A$:

$$AS^x(f)(d) \simeq d \nabla [x \mapsto f(d)].$$

Бінарна композиція послідовного виконання $\dot{\varepsilon}$ має тип $FPr_g^{V,A} \times FPr_g^{V,A} \xrightarrow{t} FPr_g^{V,A}$, та визначається наступним чином, де $pr_1, pr_2 \in FPr_g^{V,A}, d \in {}^V A$:

$$pr_1 \dot{\varepsilon} pr_2(d) \simeq pr_2(pr_1(d)).$$

Тернарна умовна композиція IF має тип $Pr^{V,A} \times FPr_g^{V,A} \times FPr_g^{V,A} \xrightarrow{t} FPr_g^{V,A}$, та визначається наступним чином, де $r \in Pr^{V,A}, pr_1, pr_2 \in FPr_g^{V,A}, d \in {}^V A$:

$$IF(r, pr_1, pr_2)(d) = \begin{cases} pr_1(d), \text{ якщо } r(d) \downarrow = T, \\ pr_2(d), \text{ якщо } r(d) \downarrow = F, \\ \text{невизначено інакше.} \end{cases}$$

Бінарна композиція циклу WH має тип $Pr^{V,A} \times FPr_g^{V,A} \xrightarrow{t} FPr_g^{V,A}$, та визначається наступним чином, де $r \in Pr^{V,A}, pr \in FPr_g^{V,A}, d \in {}^V A$:

$WH(r, pr)(d) = d'$, якщо існує послідовність іменних множин $d_0, d_1, \dots, d_n \in {}^V A$ таких, що $d_0 = d, d_n = d'$ та $d_1 = pr(d_0), d_2 = pr(d_1), \dots, d_n = pr(d_{n-1})$ разом з $r(d_0) = r(d_1) = \dots = r(d_{n-1}) = T, r(d_n) = F$. Інакше, $WH(r, pr)(d) \uparrow$.

Нуль-арна композиція тотожної програми $id \in FPr_g^{V,A}$, визначається наступним чином, де $d \in {}^V A$:

$$id(d) = d.$$

Тернарна композиція Флойд-Хоара FH має тип $Pr^{V,A} \times FPr^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$, та визначається наступним чином, де $p, q \in Pr^{V,A}, pr \in FPr^{V,A}, d \in {}^V A$:

$$FH(p, pr, q)(d) = \begin{cases} T, \text{ якщо } p(d) \downarrow = F \text{ або } q(pr(d)) \downarrow = T, \\ F, \text{ якщо } p(d) \downarrow = T \text{ та } q(pr(d)) \downarrow = F, \\ \text{невизначено інакше.} \end{cases}$$

Бінарна композиція побудови умови за прообразом PC має тип $FPr^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$, та визначається наступним чином, де $q \in Pr^{V,A}, pr \in FPr^{V,A}, d \in {}^V A$:

$$PC(pr, q)(d) = \begin{cases} T, \text{ якщо } pr(d) \downarrow \text{ та } q(pr(d)) \downarrow = T, \\ F, \text{ якщо } pr(d) \downarrow \text{ та } q(pr(d)) \downarrow = F, \\ \text{невизначено інакше.} \end{cases}$$

Для композиції побудови умови за прообразом, що задає семантику перетворювача предикатів найслабкіша передумов виконуються наступні формули, що дозволяє обчислити передумову для підпрограм без циклів.

$$PC(id, q) = q,$$

$$PC(AS^x(f), q) = S_p^x(q, f),$$

$$PC(pr_1 \bullet pr_2, q) = PC(pr_1, PC(pr_2, q)),$$

$$PC(IF(r, pr_1, pr_2), q) = (r \rightarrow PC(pr_1, q)) \wedge (\neg r \rightarrow PC(pr_2, q)) \wedge (r \rightarrow \neg r).$$

1.3 Аксіоматична система

Класична аксіоматична система запропонована Хоаром для випадку тотальних предикатів є коректною та екзистенційно повною.

$$R_{-AS} \frac{}{\{S_p^x(p, f)\} AS^x(f) \{p\}},$$

$$\begin{aligned}
R_SKIP & \frac{}{\{p\}id\{p\}}, \\
R_SEQ & \frac{\{p\}pr_1\{q\}, \{q\}pr_2\{r\}}{\{p\}pr_1 \bullet pr_2\{r\}}, \\
R_IF & \frac{\{r \wedge p\}pr_1\{q\}, \{\neg r \wedge p\}pr_2\{q\}}{\{p\}IF(r, pr_1, pr_2)\{q\}}, \\
R_WH & \frac{\{r \wedge p\}pr\{p\}}{\{p\}WH(r, pr)\{\neg r \wedge p\}}, \\
R_CONS & \frac{\{p'\}pr\{q'\}}{\{p\}pr\{q\}}, p \rightarrow p', q' \rightarrow q.
\end{aligned}$$

В класичному формулюванні для простої імперативної мови програмування система виводу має наступний вигляд:

$$\begin{aligned}
& \{P[x \mapsto A(a)]\} x := a \{P\} \\
& \{P\} skip \{P\} \\
& \frac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1;S_2\{R\}} \\
& \frac{\{B(b) \wedge P\}S_1\{Q\}, \{\neg B(b) \wedge P\}S_2\{Q\}}{\{P\} \text{if } b \text{ then } S_1 \text{ else } S_2\{Q\}} \\
& \frac{\{B(b) \wedge P\}S\{P\}}{\{P\} \text{while } b \text{ do } S\{\neg B(b) \wedge P\}} \\
& \frac{\{P'\}S\{Q'\}}{\{P\}S\{Q\}} \text{ if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q
\end{aligned}$$

1.4 Приклади

Приклад 1.1 Задамо програму та доведемо її коректність для функції $x!$.

Програма разом з післяумовою та передумовою матиме наступний вигляд:

$$\begin{aligned}
& \{x=n\} \\
& y := 1; \text{while } \neg(x=1) \text{ do } (y := y * x; x := x - 1) \\
& \{y=n! \wedge n > 0\}
\end{aligned}$$

Для того, щоб зафіксувати початкове значення змінної x введемо додаткову змінну.

Для циклу використаємо наступний інваріант:

$$INV(st) = st(x) > 0 \rightarrow (st(y) * st(x) != st(n)! \wedge st(n) \geq st(x))$$

Для другого оператора тіла циклу застосувавши правило для присвоєння отримуємо :

$$\{INV[x \mapsto x - 1]\} x := x - 1 \{INV\}$$

Тоді для першого оператора присвоєння, використовуючи як післяумову передумову з попереднього твердження.

$$\{(\{INV[x \mapsto x - 1]\})[y \mapsto y * x]\} y := y * x \{INV[x \mapsto x - 1]\}$$

Використовуючи правила для послідовного виконання.

$$\{(\{INV[x \mapsto x - 1]\})[y \mapsto y * x]\} y := y * x; x := x - 1 \{INV\}$$

Перевіримо, що $(\neg(x=1) \wedge INV) \Rightarrow (INV[x \mapsto x - 1])[y \mapsto y * x]$.

Розпишемо

$$(\neg(x=1) \wedge x > 0 \rightarrow y * x != n! \wedge n \geq x) \Rightarrow ((x > 0 \rightarrow y * x != n! \wedge n \geq x)[x \mapsto x - 1])[y \mapsto y * x]$$

Виконуємо першу підстановку:

$$(\neg(x=1) \wedge x > 0 \rightarrow y * x != n! \wedge n \geq x) \Rightarrow (x - 1 > 0 \rightarrow y * (x - 1) != n! \wedge n \geq x - 1)[y \mapsto y * x]$$

Виконуємо другу підстановку:

$$(\neg(x=1) \wedge x > 0 \rightarrow y * x != n! \wedge n \geq x) \Rightarrow (x - 1 > 0 \rightarrow y * x * (x - 1) != n! \wedge n \geq x - 1)$$

Отже маємо

$$\{(\neg(x=1) \wedge INV)\} y := y * x; x := x - 1 \{INV\}$$

Звідки за правилом для while

$$\{INV\} \text{while } \neg(x=1) \text{ do } (y := y * x; x := x - 1) \{ \neg \neg(x=1) \wedge INV \}$$

Покажемо, що $(\neg \neg(x=1) \wedge INV) \Rightarrow y = n! \wedge n > 0$

Для цього розпишемо $(x=1) \wedge (x > 0 \rightarrow y * x != n! \wedge n \geq x) \Rightarrow y = n! \wedge n > 0$.

Спростуємо $(y * 1 != n! \wedge n \geq 1) \Rightarrow y = n! \wedge n > 0$. Отже маємо

$$\{INV\} \text{while } \neg(x=1) \text{ do } (y := y * x; x := x - 1) \{y = n! \wedge n > 0\}$$

Для присвоєння можна застосувати правило $\{INV[y \mapsto 1]\} y := 1 \{INV\}$.
 Так як $x=n \Rightarrow INV[y \mapsto 1]$ то з того, що $x=n \Rightarrow (x > 0 \rightarrow y * x! = n! \wedge n \geq x)[y \mapsto 1]$
 можемо переписати $x=n \Rightarrow (x > 0 \rightarrow 1 * x! = n! \wedge n \geq x)$. В такому випадку
 $\{x=n\} y := 1 \{INV\}$

А отже застосовуючи правило для послідовного виконання отримаємо:

$$\{x=n\}$$

$$y := 1; \text{while } \neg(x=1) \text{ do } (y := y * x; x := x - 1)$$

$$\{y=n! \wedge n > 0\}$$

Що і потрібно було довести.

Приклад 1.2 Задамо програму та доведемо її коректність для функції x^y . Програма разом з післяумовою та передумовою матиме наступний вигляд:

$$\{y=n \wedge n \geq 0\}$$

$$r := 1; \text{while } (y > 0) \text{ do } (r := r * x; y := y - 1)$$

$$\{r=x^n \wedge n \geq 0\}$$

Для циклу використаємо наступний інваріант:

$$INV(st) = st(y) \geq 0 \wedge (st(y) \geq 0 \rightarrow (st(r) * st(x)^{st(y)} = st(x)^{st(n)} \wedge st(n) \geq st(y)))$$

З тіла циклу застосовуючи послідовно правила для присвоєння отримаємо наступне:

$$\{INV[y \mapsto y - 1]\} y := y - 1 \{INV\}$$

$$\{((INV[y \mapsto y - 1])[r \mapsto r * x]) r := r * x \{INV[y \mapsto y - 1]\}$$

Використовуючи правила для послідовного виконання.

$$\{((INV[y \mapsto y - 1])[r \mapsto r * x]) r := r * x; y := y - 1 \{INV\}$$

Перевіримо, що $((y > 0) \wedge INV) \Rightarrow (INV[y \mapsto y - 1])[r \mapsto r * x]$. Для цього розпишемо
 $((y > 0) \wedge (y \geq 0 \wedge (y \geq 0 \rightarrow r * x^y = x^n \wedge n \geq y)))[y \mapsto y - 1][r \mapsto r * x] \Rightarrow$
 $((y \geq 0 \wedge (y \geq 0 \rightarrow (r * x^y = x^n \wedge n \geq y)))[y \mapsto y - 1])[r \mapsto r * x]$

Виконуючи зовнішню підстановку
 $(y > 0) \wedge (y \geq 0 \wedge (y \geq 0 \rightarrow r * x^y = x^n \wedge n \geq y)) [r \mapsto r * x] \Rightarrow$
 $((y - 1 \geq 0 \wedge (y - 1 \geq 0 \rightarrow (r * x^{y-1} = x^n \wedge n \geq y - 1)))) [r \mapsto r * x]$

Виконуючи наступну підстановку
 $(y > 0) \wedge (y \geq 0 \wedge (y \geq 0 \rightarrow r * x^y = x^n \wedge n \geq y)) \Rightarrow$
 $(y - 1 \geq 0 \wedge (y - 1 \geq 0 \rightarrow (r * x * x^{y-1} = x^n \wedge n \geq y - 1)))$

Отже маємо

$$\{((y > 0) \wedge INV)\} r := r * x; y := y - 1 \{INV\}$$

Звідки за правилом для while

$$\{INV\} \text{while } (y > 0) \text{ do } (r := r * x; y := y - 1) \{(\neg(y > 0) \wedge INV)\}$$

Покажемо, що $(\neg(y > 0) \wedge INV) \Rightarrow r = x^n \wedge n \geq 0$ для цього розпишемо
 $(\neg(y > 0) \wedge (y \geq 0 \wedge (y \geq 0 \rightarrow r * x^y = x^n \wedge n \geq y))) \Rightarrow r = x^n \wedge n \geq 0$, спрощуючи отримав маємо
 $(y = 0 \wedge (y \geq 0 \rightarrow r * x^y = x^n \wedge n \geq y)) \Rightarrow r = x^n \wedge n \geq 0$, звідки
 $(y = 0 \wedge r * x^0 = x^n \wedge n \geq 0) \Rightarrow r = x^n \wedge n \geq 0$

Отже маємо $\{INV\} \text{while } (y > 0) \text{ do } (r := r * x; y := y - 1) \{r = x^n \wedge n \geq 0\}$

Для першої операції присвоєння маємо:

$\{INV[r \mapsto 1]\} r := 1 \{INV\}$. Так як $(y = n \wedge n \geq 0) \Rightarrow INV[r \mapsto 1]$, то з того, що
 $(y = n \wedge n \geq 0) \Rightarrow (y \geq 0 \wedge (y \geq 0 \rightarrow r * x^y = x^n \wedge n \geq y)) [r \mapsto 1]$ можемо переписати
 $(y = n \wedge n \geq 0) \Rightarrow (y \geq 0 \wedge (y \geq 0 \rightarrow 1 * x^y = x^n \wedge n \geq y))$.

В такому випадку $\{(y = n \wedge n \geq 0)\} y := 1 \{INV\}$.

А отже застосувавши правило для послідовного виконання отримуємо

$$\{y = n \wedge n \geq 0\}$$

$$r := 1; \text{while } (y > 0) \text{ do } (r := r * x; y := y - 1)$$

$$\{r = x^n \wedge n \geq 0\}$$

1.5 Вправи до розділу

Для наступних функцій написати програму, задати трійку Хоара, що виражає коректність програми, запропонувати інваріант циклу та довести коректність:

1. $x \operatorname{div} y$
2. $x \operatorname{mod} y$
3. $\lceil \log_x y \rceil$
4. $x!!$
5. $\lceil \sqrt{x} \rceil$

2. ЛОГІКА РОЗДІЛЕННЯ

Логіка розділення[4] є розширенням логіки Флойда-Хоара, що дозволяє оперувати твердженнями відносно динамічної пам'яті та вказівників. Для цього додатково вводяться поняття стеку як відображення імен змінних в їх значення та області динамічної пам'яті як часткового відображення адрес в їх значення. Також розглядаються спеціальні логічні зв'язки, такі як розділяюча кон'юнкція та імплікація. Дані логічні зв'язки дозволяють задавати твердження, що виконуються на ділянках динамічної пам'яті, що не перетинаються. Такі конструкції дозволяють адекватно моделювати процеси виділення пам'яті, оперування вказівниками.

Спеціальна зв'язка розділяюча кон'юнкція задається наступним чином: $P * Q(m) = \exists m_1 m_2. m_1 \oplus m_2 = m \wedge P(m_1) \wedge Q(m_2)$. Де P та Q це підформули, m, m_1, m_2 – стани динамічної пам'яті. А оператор \oplus задається як об'єднання без перетинів. Таким чином розділяюча кон'юнкція істина на певному стані динамічної пам'яті, якщо її можна розділити на дві області пам'яті, що не перетинаються, на одній з яких істинний один аргумент, а на іншій істинний другий аргумент. Так як область динамічної пам'яті задається у вигляді відображення з множини натуральних чисел, що задають адреси в множину значень, то умова того, що дві області пам'яті не перетинаються може бути сформульована як умова того, що області визначення відповідних функцій не перетинаються.

Визначення такої зв'язки дає змогу розширити логіку Флойда-Хоара спеціальним рамковим(frame) правилом, що дозволяє доводити локальні властивості підпрограм не враховуючи вплив зовнішніх по відношенню до підпрограми операторів.

Правило визначається наступним чином:

$$\frac{\{P\}S\{Q\}, \text{modv}(S) \cap \text{fv}(R) = \emptyset}{\{P * R\}S\{Q * R\}}$$

Тут $\text{modv}(S)$ позначаємо множину змінних, що були модифіковані програмою S . А $\text{fv}(R)$ позначаємо множину всіх вільних змінних R . Таким чи-

ном в даному правилі локальні властивості, що доводяться окремо позначаються $\{P\}S\{Q\}$. Твердження зовнішні до них — це R . Використання звичайної кон'юнкції в даному правилі не гарантувало того, що локальні зміни, викликані виконанням програми S не вплинуть на істинність зовнішньої властивості.

Також разом з даним правилом в логіці розширення вводяться додаткові операції та правила для роботи з ділянками динамічної пам'яті, зокрема оператор \oplus згаданий раніше. Для роділяючої кон'юнкції виконуються властивості комутативності, асоціативності, що випливає з того, що дані властивості мають оператор \oplus та кон'юнкція.

Іншим важливою зв'язкою є розширююча імплікація, що задається наступним чином $P-*Q(m) = \forall m_1 \text{ dom } m_1 \cap \text{ dom } m = \emptyset . P(m_1) \rightarrow Q(m_1 \oplus m)$. Іншими словами, для довільного розширення області динамічної пам'яті, якщо на цьому розширенні виконується ліва частина імплікації, то на області динамічної пам'яті, що отримана внаслідок розширення повинна виконуватись права частина імплікації.

Це дозволяє сформулювати аналог правила *modus ponens* для логіки розділення.

$$\frac{P*(P-*Q)}{Q}$$

Також варто зазначити, що наявність розділючої кон'юнкції дозволяє легко розширити логіку на випадок паралельних програм, розглядаючи процеси, що працюють з областями пам'яті, що не перетинаються.

Основними областями застосування логіки розділення є доведення властивостей базових алгоритмів для складних структур даних, таких як список, масив, стек, черга. Верифікується правильна робота відповідних структур даних, вивільнення пам'яті, що не використовується, неможливість доступу до пам'яті, що не відноситься до них.

Іншим важливим видом алгоритмів та програмних систем для верифікації яких широко використовується логіка розділення є алгоритми розподілених систем з критичною секцією.

Семантика логіки задається використанням спеціальної алгебри розділення, важливою частиною якої є спеціальні операції для роботи з областями динамічної пам'яті.

3. ДИНАМІЧНА ЛОГІКА

Динамічна логіка була запропонована Хареллом[5] як альтернатива використанню трійок Хоара та відповідної логіки. Динамічна логіка є модальною логікою, що використовує спеціальну модальність, що певним чином нагадує підхід Дейкстри[3] з перетворювачем предикатів найслабкіша передумова. В загальному випадку трійка Хоара $\{p\}f\{q\}$ еквівалентна формулі $p \rightarrow [f]q$

3.1 Регулярні програми

Динамічна логіка формулюється відносно спеціального класу програм, так званих регулярних програм. Регулярні програми, це програми, до яких можна звести довільну програму задану за допомогою імперативної мови програмування, прикладом якої є мова WHILE, що згадується в першому розділі і для якої задана класична аксіоматична система для логіки Флойда-Хоара.

Регулярні програми RP задаються наступним чином відносно множини атомарних програм Pr та тестів Ts .

- якщо $p \in Pr$ то $p \in RP$. Атомарні програми є базовими та неділимим.
- якщо $\phi \in Ts$ то $\phi ? \in RP$. Відповідна програма тестує чи виконується властивість ϕ в поточному стані. Якщо так, то програма продовжує свою роботу, якщо ні, то програма зациклюється.
- якщо $p_1, p_2 \in RP$ то $p_1 ; p_2 \in RP$. Відповідна програма є послідовним виконанням заданих програм.
- якщо $p_1, p_2 \in RP$ то $p_1 \cup p_2 \in RP$. Даний оператор називається оператором недетерміністичного вибору. Відповідна програма означає — вибрати одну з підпрограм — p_1 або p_2 і виконати її.
- якщо $p \in RP$ то $p^* \in RP$. Даний оператор називається оператором ітерації. Відповідна програма означає виконати p деяку обрану скінчену кількість разів.

За таких визначень звичні конструкції мови WHILE можна переписати наступним чином:

- $if \varphi then p_1 elst p_2 = \varphi ? ; p_1 \cup \neg \varphi ? ; p_2$
- $while \varphi do p_1 = (\varphi ? ; p_1)^* ; \neg \varphi ?$

В даному випадку програму з прикладу 1.2 можна задати використовуючи регулярні програми. $r := 1 ; while (y > 0) do (r := r * x ; y := y - 1)$ буде перетворено на $r := 1 ; ((y > 0) ? ; (r := r * x ; y := y - 1))^* ; \neg (y > 0) ?$

Формально семантика регулярних програм задається використовуючи послідовності обчислень, тобто множину всіх послідовностей атомарних програм, кожна з таких послідовностей задає один із можливих варіантів виконань.

3.2 Семантика динамічних логік

Семантика для Динамічних логік задається за допомогою фреймів Кріпке. Фреймом Кріпке називається пара $\langle K, m_k \rangle$, де K — множина можливих станів, а m_k — відображення, що задає значення програм. Дане відображення предикатам та умовам ставить у відповідність множину станів на яких вони істинні, а програмам відображення зі стану в стан.

Якщо задано відображення для значень базових атомарних програм та тестів, то значення для складніших програм та умов задається наступним чином:

$$m_k(\alpha \rightarrow \beta) = (K - m_k(\alpha)) \cup m_k(\beta)$$

$$m_k(0) = \emptyset$$

$$m_k([p]\alpha) = \{u \mid \forall v \in K \text{ if } (u, v) \in m_k(p) \text{ then } v \in m_k(\alpha)\}$$

$$m_k(p_1 ; p_2) = \{(u, v) \mid \exists w \in K (u, w) \in m_k(p_1) \text{ and } (w, v) \in m_k(p_2)\}$$

$$m_k(p_1 \cup p_2) = m_k(p_1) \cup m_k(p_2)$$

$$m_k(p^*) = m_k(p)^*$$

$$m_k(\alpha ?) = \{(u, u) \mid u \in m_k(\alpha)\}$$

3.3 Вправи до розділу

1. Перетворити програми для завдань з розділу 1 в вигляді регулярних програм.
2. Переписати твердження з завдань розділу 1 використовуючи динамічну логіку.

4. СИСТЕМА ISABELLE

Система Isabelle[14] це система підтримки автоматичних доведень, що з 1986 року розробляється вченими Кембриджського університеті спільно з вченими Мюнхенського технічного університету. Останні версії системи розповсюджуються разом з середовищем розробки, що базується на відкритому jEdit.

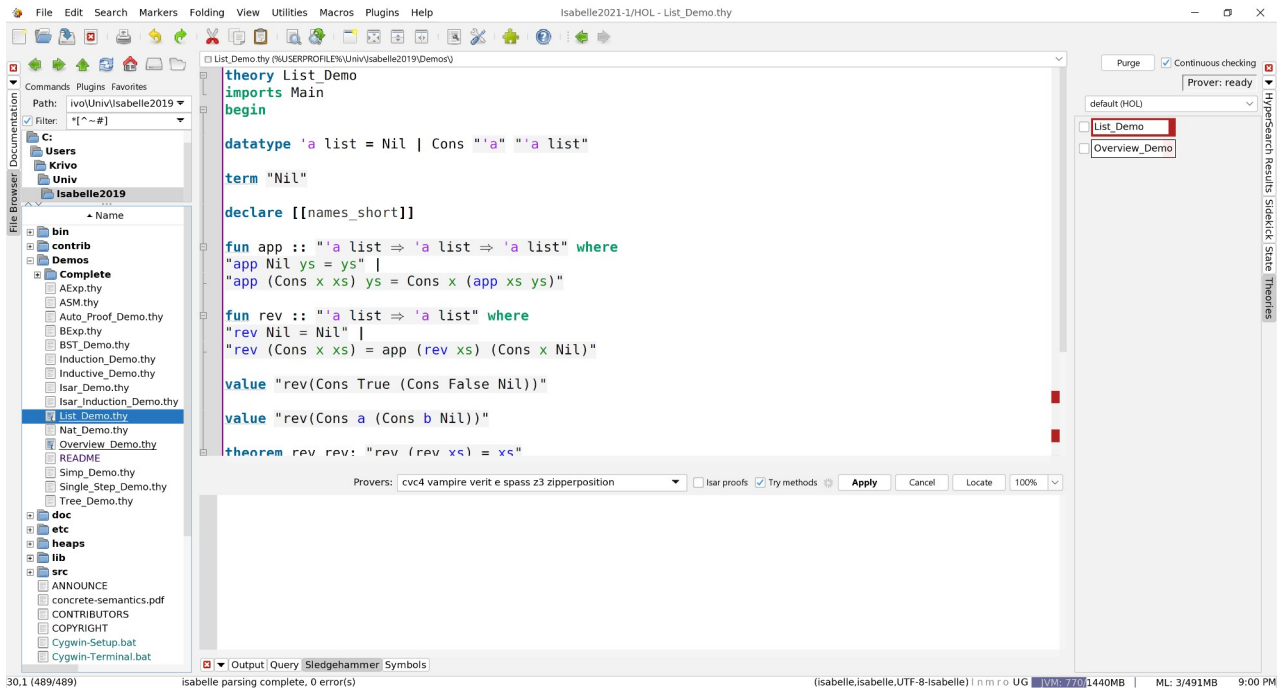


рис.1 середовище розробки Isabelle

Система Isabelle є узагальненою системою для застосування логічних формалізмів. Прикладом такого формалізму є Isabelle/HOL – застосування Isabelle для HOL, тобто Higher-Order Logic(логіки вищого порядку). HOL також це спеціальна мова для специфікацій, що дозволяє задавати леми, теореми та формулювати схему їх доведення. HOL є функціональною мовою програмування, що має розширені логічні засоби.

4.1 Теорії Isabelle

Всі визначення в системі формуються в теоремах, які є модулями, або бібліотеками в розумінні інших мов програмування. Кожна теорія має

міститись в окремому файлі, ім'я якого співпадає з іменем теорії, а розширення “thy”. Файл з теорією зазвичай має наступну структуру:

```
theory NameOfTheory  
imports SomeOtherTheory1, SomeOtherTheory2, SomeOtherTheory3  
begin  
  function ...  
  ...  
  lemma ...  
  ...  
end
```

Тут *NameOfTheory* – ім'я поточної теорії, *SomeOtherTheory1, SomeOtherTheory2, SomeOtherTheory3* – імена теорій, що імпортуються. Всі визначення, що належать до теорії містяться між **begin** та **end**. Імпортування теорії означає включення всіх лем, теорем та визначень відповідної теорії. Імпортування є транзитивним — якщо теорія А імпортується в теорії Б, а теорія Б імпортується в теорії В, то в теорії В будуть доступні всі визначення з теорії А. Разом з системою Isabelle постачається бібліотека теорії, зокрема теорія *Main*, що містить основні можливості, підтримку списків, арифметики та ін.

4.2 Типи Isabelle

Логіка HOL є строго типізованою, кожен терм має свій тип та відповідність типів перевіряється під час інтерпретації. Система підтримує наступні стандартні типи:

- **базові типи.** До базових типів відносять, зокрема, булевий тип *bool*, натуральні числа *nat*, цілі числа *int*.
- **складні типи.** До них відносять типи які дозволяють для існуючих типів будувати складніші типи, серед них списки *list*, множини *set*. Вони записуються постфікс після типу до якого за-

стосовується — наприклад $int\ set$ – тип множина, що складається з цілих чисел.

- **функціональні типи.** Позначаються за допомогою \Rightarrow . Наприклад, $nat \Rightarrow nat$ позначає унарну функцію над натуральними числами.
- **змінні типів.** Позначаються $'a$ – змінні, що дозволяють використовувати узагальнення для типів. Наприклад, $'a\ list \Rightarrow 'a$ позначає функцію, що приймає на вхід список елементів певного типу $'a$ та повертає елемент того ж типу.

Крім вбудованих можливостей, користувач може задавати власні типи. Формулами вважаються терми, що мають тип $bool$. Підтримується квантифікатори, лямбда-числення та рівність як вбудована функція.

Булевий тип в системі задається наступним чином — **datatype** $bool = True \mid False$. Змінні приймають одне з двох значень констант. Також основні логічні зв'язки, такі як кон'юнкція, диз'юнкція, заперечення, імплікація, задаються як функції над аргументами відповідного типу.

Натуральні числа в системі задаються як **datatype** $nat = 0 \mid Suc\ nat$. Значення генеруються за допомогою конструкторів 0 та Suc . Де перший конструктор задає константу нуль, а другий — наступне число. Маємо відповідно значення $0, Suc\ 0, Suc\ (Suc\ 0), \dots$ Що задають натуральні значення $0, 1, 2, \dots$ відповідно. В системі також можна використовувати числа для позначення натуральних та цілих значень, проте варто пам'ятати, що цілі та натуральні константи перевантажені, тому іноді потрібно явно вказувати тип константи — наприклад $0::nat$ для позначення нуля, який має тип натуральні числа. Також в системі задано вбудовані основні арифметичні операції та відношення як над натуральними так і над цілими числами.

Тип список задається наступним чином - **datatype** $'a\ list = Nil \mid Cons\ 'a\ "'a\ list"$. Тут ми маємо список елементів типу $'a$. Конструктором Nil позначається пустий список, а конструктором $Cons$ – список, що задається як

голова списку(перший аргумент) і хвіст списку(другий аргумент). Теорія *List* містить основні функції над списками та відповідні теореми. Для зручності можна використовувати позначення $[x \# xs]$. Де x — голова списку, а xs — хвіст.

4.3 Типи та функції визначені користувачем

Окремо слід відмітити можливість задавати синоніми типів, що не приводять до утворення нового типу, лише дозволяють задати зручніше ім'я для існуючого типу. Приклад синтаксису — **type_synonym** *string* = “*char list*”.

Загальна схема задання користувацького типу:

```
datatype ('a, 'b, 'c, ...) typeName =  
    Constructor1 “typeName1_1” ... “typeName1_n1”  
    | Constructor2 “typeName2_1” ... “typeName2_n2”  
    ....  
    | ConstructorM “typeNameM_1” ... “typeNameM_nM”
```

Тут *ConstructorI* імена конструкторів, кожен з яких приймає відповідну кількість аргументів відповідних типів. Кожен конструктор задає альтернативний випадок генерації представника типу(наприклад порожній список і список, що має голову та хвіст). Система автоматично генерує правила, суть яких полягає в тому, що значення задані різними конструкторами вважаються різними і два значення вважаються рівними, якщо їх конструктори і кожен аргумент конструктора співпадають.

Приклади визначень типів:

datatype 'a *bintree* = *Leaf* | *Node* “'a *bintree*” 'a “'a *bintree*” задає бінарне дерево з мітками типу 'a.

datatype 'a *tree* = *Leaf* 'a | *Node* 'a “'a *tree list*” задає довільне дерево з мітками типу 'a, де листок також має мітку.

Функції можуть бути визначені або використовуючи терми з існуючими функціями, або за допомогою рекурсивних визначень.

Синтаксис визначення функції:

```
definition functionName :: “function type” where  
“functionName arguments = expression”
```

Приклад 2х:

```
definition double :: “int  $\Rightarrow$  int” where  
“double x = 2*x”
```

Рекурсивне визначення функції задається використовуючи функціональну природу мови HOL. Основним механізмом є зіставлення шаблонів, або *pattern-matching*.

```
fun functionName :: “function type” where  
“functionName argumentsPattern1 = expression1” |  
“functionName argumentsPattern2 = expression2” |  
....  
“functionName argumentsPatternN = expressionN”
```

В визначенні шаблони співставляються згори вниз. Це дозволяє використовувати шаблони, що перетинаються. Також шаблон, що співставляється з рештою випадків не потрібно детально розписувати. В системі Isabelle всі функції мають завершуватись, система шукає доведення цього автоматично використовуючи лексикографічний порядок для аргументів функції. У випадку, якщо доведення не буде знайдено, користувач отримує повідомлення про помилку. Також для кожної функції заданої рекурсивно системою автоматично генерується правило *funcName.induct*. Дане правило генерується за кожним шаблоном з визначення функції та рекомендовано для використання в доведенні лем та теорем про властивості відповідної функції.

Приклад визначення функції:

```
fun mirror :: “a bintree  $\Rightarrow$  ‘a bintree” where  
“mirror Leaf = Leaf” |  
“mirror Node tree1 val tree2 = Node mirror tree2 val mirror tree1”
```

4.4 Лемми та теореми

Визначення лемми або теореми відбувається наступним чином:

lemma *lemmaName*[simp] “*formula*”

apply (*proofMethod1*)

apply (*proofMethod2*)

...

apply (*proofMethodN*)

done

Ім'я лемми або теореми може бути опущено, а позначка `simp` використовується для лем, які можуть бути використані, як правила спрощення. Інструкції **apply** мають привести до виведення всіх початкових та проміжних тверджень, що в системі називаються цілями(`goals`). Також комбінація **apply** (*proofMethodN*) **done** може бути замінена на **by** (*proofMethodN*), тобто якщо застосування методу доведення призводить до виводу заключної цілі.

Середовище розробки має спеціальну область(нижче редактора теорій) в якій відображається поточний стан доведення в якому зазначаються поточні цілі. Це дозволяє проглядати використання методів доведення один за одним і проводити відлагодження в разі потреби. Також інструкції **sorry** та **oops** застосовані замість **done** дозволяють відмітити лемми або теореми, які або будуть доведені пізніше у випадку **sorry**, або мають ігноруватись тимчасово у випадку **oops**. Таким чином **sorry** дозволяє проводити доведення згори вниз, припускаючи твердження, які потім будуть доведені і продовжуючи доведення поточної лемми.

Приклад лемми:

lemma “*mirror (mirror t) = t*”

apply (*induction t*)

apply (*auto*)

done

4.5 Методи доведення

Одним з основних методів доведення для типів даних та функцій, що задаються рекурсивно, є використання індуктивного методу. Застосування відповідного методу доведення для поточної цілі замінює її на декілька нових цілей згідно індуктивного правила. Евристичні правила стосовно застосування індуктивного методу наступні — якщо функція визначається відносно основних конструкторів типу, заданого рекурсивно — потрібно використовувати індукцію за відповідною змінною, якщо ж визначення основної функції, що використовується в лемі є більш складним, потрібно використовувати індуктивне правило згенероване для відповідної функції.

Індукція відносно змінної — **apply** (*induction var*). Розділяє ціль на базу індукції та індуктивний крок відносно визначення типу змінної *var*. Де базою індукції будуть твердження відносно конструкторів, що не є рекурсивними, тобто не залежать від аргументу того ж типу, а крок індукції сформовано для кожного рекурсивного конструктора.

Наприклад для типу *'a bintree* базою індукції буде твердження відносно *Leaf*. А кроком індукції буде доведення того, що якщо для *tree1* та *tree2* твердження виконується, то воно виконується і для *Node tree1 val tree2*. Проте варто зазначити, що в цьому випадку всі змінні у відповідних твердженнях крім змінної за якою застосовується метод індукції будуть фіксованими. Для того, щоб додати квантифікацію за відповідною змінною *val* необхідно використати інструкцію **apply** (*induction var arbitrary:val*). Таким чином база індукції та припущення індукції будуть доводитись для довільного *val*, а не для певного фіксованого.

Індукція відносно індуктивного правила для функції. **apply** (*induction arg1,arg2, ...,argN rule: funcName.induct*). Базою індукції є всі шаблони з визначення функції, що не містять рекурсії, а правила для кроку індукції визначаються аргументами рекурсивних викликів.

Для наступної функції:

fun *div2* :: “*nat* \Rightarrow *nat*” **where**

“ *div2* *O* = *O*” |

“ *div2* (*Suc* *O*) = *O*” |

“ *div2* (*Suc* (*Suc* *n*)) = *Suc*(*div2* *n*)”

Базою індукції буде доведення твердження для *O* та *Suc O*. А кроком індукції буде довести, що якщо твердження виконується для *n* (аргумент функції в правій частині останнього визначення) , то воно виконується і для *Suc (Suc n)*(аргумент функції в лівій частині останнього визначення).

Метод спрощення **apply** (*simp*). До першої з цілей застосовуються всі можливі правила спрощення. Даний метод не є дуже повним. Якщо доведення не можливо, стан доведення буде співпадати з тим станом, в якому система не змогла більше застосувати правила спрощення, що дозволяє спробувати знайти помилку в доведенні, сформулювати лему, яка потрібна для доведення. Також будь-які леми, що мають відношення рівності можуть бути відмічені, як леви для спрощення [*simp*]. Ці леми будуть застосовані зліва направо. Для леми “*a* = *b*” входження терма *a* буде замінено на терм *b*. Також додаткові правила спрощення можуть бути додані, або видалені для поточної інструкції, використовуючи наступний синтаксис — **apply** (*simp* *add*: *eqa1* ... *eqaN* *del*: *eqd1* ... *eqdM*). В такому випадку всі леми і теореми з іменами *eqa1* ... *eqaN* будуть додані до набору тих, що застосовуються, а *eqd1* ... *eqdM* – видалені. Проте слід відмічати леми як леми для спрощення і додавати їх обережно, бо системою не гарантується перевірка на зациклювання правил спрощення і на користувача покладається ця задача. Тому рекомендацією є перевірити той факт, що правила дійсно є правилами для спрощення і терм праворуч від знаку рівності є за тим чи іншим параметром “простішим” за терм ліворуч.

Метод *auto*. **apply** (*auto*) аналогічний методу спрощення за тим винятком, що метод застосовується для всіх поточних цілей та використовує

більше правил в своїй роботі. Для додавання нових правил спрощення потрібно використовувати синтаксис **apply** (*auto simp add: ... simp del: ...*).

Варто зазначити, що методи *simp* та *auto* є дуже неповними. Кращим є метод *fastforce*, цей метод дозволяє довести більшу кількість тверджень, використовує додаткові правила з логіки, теорії множин, відношень та арифметики. Даним методом також можна розширювати за допомогою правил спрощення. Проте цей метод має недолік — він або успішний, або ні. Таким чином немає змоги дослідити проміжний результат у випадку, коли твердження не було успішно доведено.

Для пошуку доведень для тверджень, що використовують лише логіку першого порядку, метод *blast* може бути використаний. Цей метод є повним для логіки першого порядку. Має переваги і недоліки аналогічно методу *fastforce*. Може бути розширений за допомогою додаткових правил виводу.

Метод доведення *arith* може бути використаний для доведення властивостей в межах формальної арифметики. Цей метод є повний для теорій першого порядку для натуральних та цілих чисел, проте для дійсної арифметики є повним для безкванторних формул.

Слід окремо відмітити вбудовану команду *sledgehammer*. Дана команда викликає список зовнішніх систем автоматичного доведення теорем для пошуку доведення. Таким чином система очікує певний проміжок часу відповіді від зовнішніх систем з методом, що потрібно використати в системі для доведення твердження. Команда *sledgehammer* використовує зовнішні системи для пошуку доведення засобами саме Isabelle а не доведення в рамках формалізмів зовнішніх систем. Тому немає гарантії, що доведення в цьому разі буде знайдено.

4.6 Вправи до розділу

1. Задати новий тип - дерево, не обов'язково бінарне, мітки вузлів — натуральні числа. Реалізувати функції обходу зліва на право і навпаки. Довести теорему, що обернувши список отриманий при обході зліва на право, результатом буде список отриманий при обході з права наліво.

2. Задати новий тип - бінарне дерево в якому мітками є натуральні числа з функцією видаленням всіх листків з міткою менше заданого числа, довести, що застосувавши операцію видалення повторно з параметром не більшим ніж той, що був заданий при першому застосуванні ми отримаємо те саме дерево. Приклад - (видалити 3 (видалити 5 дерево)) = (видалити 5 дерево).

3. Задати нові типи — черга та стек натуральних чисел, задати операції додавання нового елемента та отримання елемента з вершини стеку та початку черги (push, pop, enqueue, dequeue). Додати також операції додавання списку елементів в стек і чергу поелементно. Додати операції отримання всіх елементів стеку та черги поелементно у вигляді списку. Довести, що для довільного списку елементів — додавши їх до черги і потім отримавши всі елементи черги отримаємо той же список. Довести для стеку аналогічне твердження, лише в результаті ми отримуємо обернений список.

4. Задати новий тип — булеві вирази з операціями заперечення та кон'юнкції. Додати операції обчислення виразів, спрощення констант та однакових змінних. Довести, що введені операції спрощення не змінюють значення виразу.

5. Задати новий тип — булеві вирази з операціями заперечення та диз'юнкції. Додати операції обчислення виразів, спрощення констант та однакових змінних. Довести, що введені операції спрощення не змінюють значення виразу.

6. Задати новий тип — булеві вирази з операціями заперечення та імплікації. Додати операції обчислення виразів, спрощення констант та

однакових змінних. Довести, що введені операції спрощення не змінюють значення виразу.

7. Описати теорію абстрактних ґраток та напівґраток. Задати також теорію ґраток та напівґраток. Сформулювати та довести теореми про співвідношення операцій та відповідних відношень.

СПИСОК ЛІТЕРАТУРИ

1. Handbook of Logic in Computer Science. Edited by S. Abramsky, Dov M. Gabbay and T. S. E. Maibaum. – Oxford Univ. Press. – Vol. 1–5, 1993–2000.
2. Hoare C.A.R., Jifeng He. Unifying Theories of Programming. – London: Prentice Hall Europe, 1998.
3. Dijkstra E.W. A Discipline of Programming / E.W. Dijkstra // Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
4. Reynolds J.C. Separation Logic: A logic for Shared Mutable Data Structures / J.C. Reynolds // LICS, 2002 P. 55-74.
5. Harel D. Dynamic logic, Handbook of Philosophical Logic / D. Harel, D. Kozen, J. Tiuryn // 1984 P. 497-604.
6. O’Hearn P.W. The logic of bunched implications / P.W. O’Hearn, D. J. Pym // Bulletin of Symbolic Logic, Vol. 5(2), 1999 P. 215–244.
7. Rosu G., Ellison C., Schulte W. Matching Logic: An Alternative to Hoare/Floyd Logic LNCS 6486, 2010 P. 142-162.
8. Owicki S.S. Verifying properties of parallel programs: An axiomatic approach / S.S. Owicki, D. Gries // Communications of the ACM, Vol. 19(5), 1976 P. 279–285.
9. Bergstra J.A. et al Handbokk of Process Algebra – Holland 2001.
10. Hoare C. A. R. Communicating sequential processes. *Communications of the ACM* **21** (8): P. 666–677 , 1978.
11. Milner R. A Calculus of Communicating Systems, Springer Verlag, 1980.
12. Schneider K.: Verification of Reactive Systems. Formal Methods and Algorithms. – Berlin-Heidelberg: Springer-Verlag, 2004.
13. Hanne Riis Nielson, Flemming Nielson Semantics with Applications. An Appetizer, Springer-Verlag.– 2007.– 274p.
14. <https://isabelle.in.tum.de/>
15. Нікітченко М.С., Шкільняк С.С. Прикладна логіка. – К., 2013.