КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра математичної інформатики

«До захисту допущено»
Завідувач кафедри
В. М. Терещенко _____
(підпис)

«___» _____20_ р.

Дипломна робота
на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:
ВЕРИФІКАЦІЯ ПЕРЕТВОРЕННЯ МІЖ ВИНЯТКОВО ФУНКЦІЙНОЮ ТА ІМПЕРАТИВНОЮ ПРЕДМЕТНО-ОРІЄНТОВАНИМИ МОВАМИ ПРОГРАМУВАННЯ НА ПРИКЛАДІ HELIX

Виконав студент 4 курсу
Зайчук Ілля Костянтинович _____
(підпис)

Науковий керівник:
асистент
Бобиль Богдан Володимирович _____
(підпис)

Засвідчую, що в цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2021

РЕФЕРАТ

HELIX є формально верифікованим ланцюжком мов програмування та правил перекладу, призначеним для генерації високоефективних імплементацій ряду чисельних алгоритмів. Будучи заснованим на існуючій системі SPIRAL, HELIX додає строгість формального доведення коректності з використанням інструменту інтерактивного доведення теорем Coq. Він формально описує набір предметно-орієнтованих мов, починаючи з HCOL, що задає абстрактний потік даних обчислення. Робота HELIX полягає в перетворенні початкової програми через низку проміжкових мов, що закінчується на LLVM IR.

В цій роботі описані три мови цього ланцюжка та формальна верифікація переходу між ними. Розглянуті переходи є нетривіальним, адже кожна наступна мова ланцюжка вводить абстракції нижчого рівня, у порівнянні з попередньою. Відбувається перехід від чисто функційної мови змішаного рівня вбудови до глибоко вбудованої імперативної, від абстрактного алгебраїчного типу даних до машинних чисел з рухомою комою, додається модель пам’яті, області видимості змінних та монадна обробка винятків. Продемонстрована архітектура цих мов, автоматичний перехід між ними та автоматизоване доведення збереження семантики на кожному з кроків.

ЗМІСТ

ВСТУП

Актуальність роботи та підстави для її виконання.

Підвищена залежність сучасного суспільства від комп'ютерних програм та кібер-фізичних систем, у поєднанні з постійно зростаючим рівнем їх складності та витонченості робить проблему коректності та надійності як ніколи важливою. Коли код використовується в критично важливих системах, як-то в авіаіндустрії, медичних застосунках, безпілотних автомобілях та ін., ціною помилки можуть стати людські життя. В таких ситуаціях класичні методи перевірки коректності коду (такі як unit тестування та ін.) швидко стають недостатніми. На заміну їм приходять більш надійні, зокрема формальна верифікація - побудова точної математичної моделі коду та аналітичне доведення коректності його виконання.

Ця робота знаходиться на перетині областей високоефективних обчислювальних систем та формалізованих систем високого рівня надійності. **Мета роботи** полягає в розробці системи генерації високоефективних імплементацій широкого класу числових алгоритмів, яка при цьому була б повністю формально верифікованою та надавала найвищі можливі гарантії коректності та надійності.

Основними **задачами роботи** є:

1. Розробка та демонстрація нового підходу до двоетапного перекладу залежно-типізованої чисто функційної мови змішаного рівня вбудови у глибоко вбудовану імперативну, зі збереженням семантики.
    a. Початковий перехід від мови *Σ-HCOL* до проміжкової *MHCOL* зі зниженням рівня абстракції через введення низькорівневої моделі представлення даних та обробки помилок.
    b. Подальший перехід від мови *MHCOL* до *DHCOL*.

Формалізація моделі представлення пам’яті, обчислювального середовища, та структурної операційної семантики цільової мови. Перехід від абстракції часткових обчислень до конкретних незалежних операцій з пам’яттю. Формалізація та доведення відповідності між денотаційною семантикою *MHCOL* та операційною *DHCOL.*

2. Демонстрація переходу від абстракцій дійсних та натуральних чисел до машинних чисел з рухомою комою IEEE 754 та цілих заданої розмірності.

Взаємозв’язок з іншими роботами.

Представлена робота описує формальну верифікацію частини HELIX - проекту, направленого на імплементацію системи SPIRAL з доданням підвищеного ступеня надійності для критично важливих застосувань. Хоч HELIX і заснований на (та тісно пов’язаний з) SPIRAL, цей проект не є прямим доведенням її коректності. Навпроти: очікується, що HELIX використовуватиметься в поєднанні зі SPIRAL, адже має дещо відмінний набір функцій та вирішуваних задач:

- Головною метою HELIX є висока *надійність*, на відміну від високої *ефективності* в SPIRAL
- HELIX використовує вже існуючі правила оптимізації SPIRAL у якості оракула, та тільки валідує їх результати, не запроваджуючи власних верифікованих правил
- SPIRAL направлена на генерацію коду мовою C, в той час як HELIX націлений на LLVM IR

Існують також декілька проектів, направлених на верифіковану компіляцію функційних мов в імперативні.

Проект *CertiCoq* [9] перекладає програми мовою Gallina на обмежену підмножину мови C - мову Clight, запроваджену та формалізовану у проектом CompCert. Ця задача є значно ширшою за

описану в цій роботі, адже полягає в перекладі не предметно-орієнтованої мови (як *Σ-HCOL*), а повноцінної залежно типізованою мови загального призначення.

Ще один пов’язаний проект - *CakeML* [10] - також направлений на формалізацію підмножини мови програмування загального призначення. На відміну від HELIX, *CakeML* використовує логіку вищого порядку (в інтерактивному середовищі доведення теорем HOL) для задання великокрокової семантики. Між двома проектами є подібності: так, наприклад, в обох використовуються дефініційні інтерпретатори (у випадку HELIX - мовою Gallina) для задання семантики мови вищого рівня. Основними відмінностями є використання у HELIX структурної операційної семантики (проти великокрокової в *CakeML*) та підходу валідації перекладу замість верифікованої компіляції. Крім того, програми системи HELIX гарантовано завершуються з результатом (використання “палива” у HELIX не впливає на семантику, на відміну від *CakeML*).

1. МАТЕМАТИЧНІ ОСНОВИ

Формальна верифікація

Під *формальною верифікацією* деякої програми зазвичай розуміють формальне доведення відповідності заданої програми деякому набору властивостей - так званої *формальної специфікації*. Ця задача зазвичай включає в себе побудову формальної моделі мови - строгого опису її синтаксису та семантики деякою чітко визначеною формальною мовою, задання вимог до роботи програми, та їх аналітичне доведення на рівні моделі. До вимог формальної специфікації можуть входити будь-які особливості виконання програми, від чисто математичного відношення між вхідними та вихідними даними, до обчислювальної складності, використання машинних ресурсів, побічних ефектів та ін..

Результатом формальної верифікації стає формальний доказ відповідності програми заданій специфікації. Цей результат є сильнішим ніж будь-який (навіть автоматизований) рівень класичного тестування - в той час як тестування дозволяє емпірично перевірити роботу програми на деякій скінченній підмножині можливих вхідних даних, формальна верифікація надає аналітичні гарантії коректності її виконання на всіх можливих наборах вхідних даних та середовищах.

Для більшості практичних застосувань, доведення специфікацій набувають надто великих об’ємів для ручної перевірки. Через це зазвичай проекти з формальної верифікації виконуються машиночитано мовою, а перевірка коректності власне доведення тверджень є автоматизованим процесом. Багато доведень грунтуються на деяких припущеннях про формат вхідних даних чи стан середовища виконання. Такі припущення мають бути чітко вказані як частина специфікації, адже формальні гарантії, надані формальною верифікацією, обмежуються виконанням таких припущень.

Інтерактивний програмний засіб доведення теорем Coq

Coq [11] - інтерактивний програмний засіб доведення теорем, що використовує власну формально означену функційну мову програмування Gallina. Теоретичні основи Coq грунтуються на трьох основних концепціях: числення конструкцій Тьєррі Кокана (англ. calculus of constructions, CoC) [12], поняття залежного типу конструктивної теорії типів [13], та ізоморфізму Карі-Говарда [14]. Завдяки залежній типізації (англ. dependent typing) мови Gallina та відповідності Карі-Говарда, Coq дозволяє записувати математичні теореми (у вигляді сигнатур функцій) та їхні доведення (у формі відповідних імплементацій). Серед можливостей Coq виділимо:

- Означення функцій та предикатів.
- Задання математичних теорем та специфікацій ПЗ.
- Інтерактивне чи автоматичне виведення доведення цих теорем.
- Механічна перевірка цих доведення довіреним ядром.
- Синтез сертифікованих програми мовами загального призначення.

Весь код, описаний в цій роботі, включаючи означення, специфікації та доведення, було імплементовано в системі Coq.

2. ОГЛЯД SPIRAL

З сучасним рівнем розвитку апаратних архітектур, задача високоефективної імплементації числових алгоритмів стає складною для ручного виконання, навіть з використанням оптимізуючих компіляторів, і часто вирішуються використанням спеціалізованих систем генерації коду, таких як SPIRAL [1].

У розробці останні 20 років, система SPIRAL може використовуватись для генерації, синтезу та автоматичної оптимізації програм та бібліотек. Робота системи заснована на перетворенні вискорівневих специфікацій математичних алгоритмів в високо оптимізований код. SPIRAL було використано для імплементації багатьох обчислювальних алгоритмів з області обробки сигналів та зображень, включаючи алгоритми на графах, системи контролю роботизованої техніки, software-defined radio (SDR), числове розв’язання диференціальних рівнянь з частинними похідними та ін.. Система SPIRAL здатна генерувати код для багатьох платформ: від портативних та користувацьких (серверних) процесорів, до високоефективних та суперкомп’ютерних систем [2].

У випадках, коли згенерований SPIRAL високоефективний код використовується в критично важливому програмному забезпеченні, постає питання про надійність цього коду, та набір гарантій щодо його коректності та його відповідності абстрактній моделі. Задача проекту HELIX полягає в тому, щоб формально довести коректність застосовуваних SPIRAL оптимізацій та засобів генерації коду з використанням інструменту інтерактивного доведення теорем Coq.

Робота SPIRAL та HELIX полягає в поетапному перетворенні початкової математичної формули, через набір проміжкових мов, у

ефективний машинний код. Кроки перетворення відповідають різним рівням абстракції:

1. Математична формула
2. Потік даних (SPIRAL: мова OL, HELIX: мова HCOL)
3. Потік даних з неявними циклами (SPIRAL: мова Σ-OL, HELIX: мова Σ-HCOL)
4. Імперативна програма (SPIRAL: мова iCode, HELIX: мова FHCOL)
5. Код загальноприйнятої низькорівневої мови (SPIRAL: C програма, HELIX: інструкції LLVM IR)

Мова OL (та відповідна їй HCOL) дуже наближена до чистої математичної нотації та дозволяє відображати широкий клас корисних математичних формул. На першому кроці SPIRAL намагається представити початковий вираз у вигляді функційної композиції простіших, без втрати діаграми потоків даних обчислення [3]. Отриманий таким чином вираз перетворюється у мову наступного кроку, Σ-OL, набуваючи при цьому неявної структури циклічних обчислень. На наступному кроці вираз мовою Σ-OL зазнає ряд оптимізуючих перетворень згідно з набором правил SPIRAL, що допомагають у генерації найбільш ефективного машинного коду для заданої платформи. Після цього Σ-OL вираз перетворюється на проміжний код імперативної програми iCode. Таким чином, SPIRAL перекладає діаграму потоків даних на конкретну послідовність циклів та арифметичних операторів. Нарешті, проміжний імперативний код перетворюється на конкретну програму мовою C, компіляція якої оптимізуючим компілятором результує у вискоефективному машинному коді, що відповідному початковій математичній формулі.

У цій роботі розглянуто частину ланцюжка перетворень системи HELIX - формально верифікованої версії SPIRAL. Загалом, імплементація такої системи полягає в означенні та формалізації всіх проміжкових мов,

заданні кроків перетворення між ними та доведенні збереження семантики на кожному з них.

Частка ланцюжка, що описана в цій роботі позначена на **Рис. 1.** Ця робота зосереджена на архітектурі мов *MHCOL*, *DHCOL* та *FHCOL*.

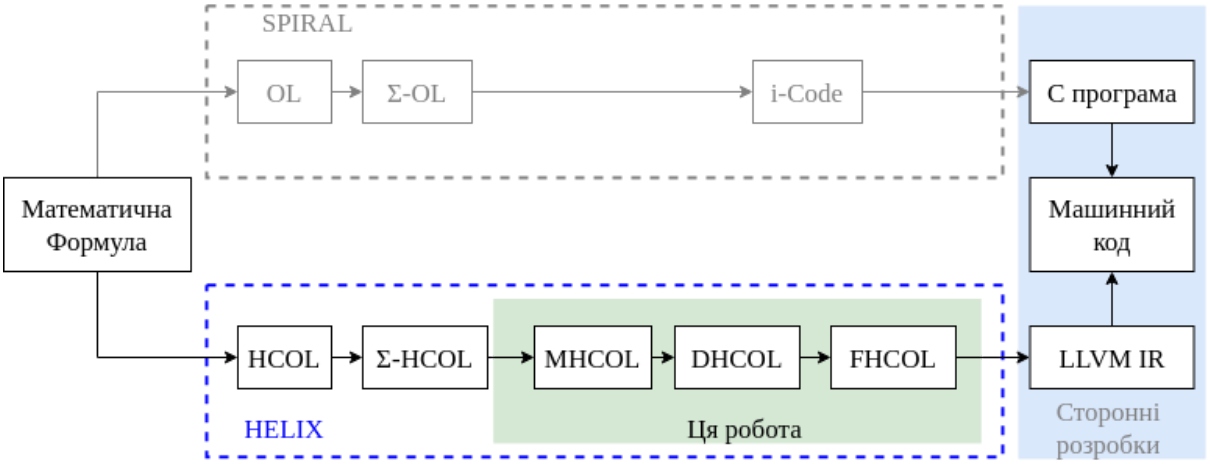

Рис. 1. Етапи перетворення SPIRAL та HELIX

Попередні етапи перетворення ланцюжка та доведення їх властивостей наведено у джерелах [4] та [5]. На відміну від SPIRAL, HELIX використовує LLVM IR замість C у якості фінальної мови ланцюжка. Таке рішення було прийнято через відносну простоту доведення збереження семантики на цьому кроці. Деякі особливості мови C, що використовуються оптимізованим SPIRAL кодом, наразі не мають повної підтримки в ком піляторі CompCert [6] [18] - найбільш розвиненому на сьогоднішній день формально верифікованому компіляторі мови C. З іншого боку проект Vellvm [7] надає формальні моделі майже всіх необхідних нам функцій LLVM IR.

3. ПІДХІД

Початком ланцюжка перетворень у HELIX є мова *Σ-HCOL*. Вона є чисто функційною операторною мовою, вбудованою у Coq на змішаному рівні, та заснованою на понятті *оператора багатолінійної алгебри*, означеного як відображення між векторними просторами. Вона працює на кінцевих розріджених векторах абстрактного *типу-носія*. Нашою частковою задачею є компіляція цієї мови у проміжкову імперативну - *DHCOL* - таку, що підлягатиме подальшому перетворенню у LLVM IR. Ця задача пов'язана з набором проблем:

1. Переклад чисто функційної програми на імперативну мову.
2. Задання відповідності між структурами даних *Σ-HCOL* і пам'яттю та змінними середовища *DHCOL*.
    a. Зокрема відповідність розріджених векторів та частково ініціалізованих блоків пам'яті.
3. Перехід від змішаного до глибокого рівня вбудови мови.
4. Обробка винятків.
5. Перехід від абстрактного *типу-носія* до чисел з рухомою комою IEEE 754.
6. Перехід від абстрактних натуральних чисел до апаратних цілих чисел з обмеженою бітовою довжиною.
7. Доведення семантичної відповідності між вихідним *Σ-HCOL* виразом та згенерованою *DHCOL* програмою.

В цій роботі описані розв'язки наведених проблем та деякі технічні особливості імплементації мови *DHCOL*.

Перетворення початкового оператора виконуються через низку проміжних мов, як показано на **Рис. 1**. З кожним кроком рівень абстракції зсувається в напрямку від чисто математичних операцій над абстрактними

алгебраїчними типами до низькорівневих інструкцій LLVM IR, що оперують регістрами та комірками пам'яті. Збереження семантики доводится між кожною послідовною парою мов ланцюжка.

Початкова мова - *Σ-HCOL* - вбудована в Coq на змішаному рівні (поєднання глибокого та мілкого рівнів), чисто функційна. Основним типом даних є скінченнорозмірний розріджений вектор *типу-носія*. Програми *Σ-HCOL* не мають обробки винятків, адже потенційні помилкові ситуації (такі як спроби доступу до даних за границями масиву) унеможливлюються сильною залежною типізацією.

В *Σ-HCOL* розріджені вектори використовуються як абстракція над частковими обчисленнями. Кожен оператор може виконувати операції лише над деякими елементами вектора, залишаючи інші невизначеними. Для виконання алгебраїчних перетворень виразів *Σ-HCOL*, зберігається інформація про розрідженість (набір індексів неозначених елементів) вектора, але невизначеним елементам назначається приховане нейтральне значення "за замовчуванням" . Не зважаючи на те, що ці значення використовуються в алгебраїчній теорії, на якій засновані правила оптимізації *Σ-HCOL*, вони не повинні змінювати остаточний результат обчислення. Значення будь-якого оператора залежить виключно від означених (непорожніх) елементів вхідних векторів.

Наведемо неформальний огляд 15 операторів мови *Σ-HCOL*:

1. `IdOp` – тотожний оператор.
2. `Embed i n` – бере елемент з одноелементного вхідного вектора та вставляє його під заданим індексом у розрідженому векторі заданої довжини.
3. `Pick i` – обирає елемент на заданій позиції з вхідного вектора та повертає його у одноелементному векторі.
4. `Scatter f` – "переставляє" елементи вхідного вектора згідно з функцією індексного відображення `f`. Відображення є ін'єктивним,

але не обов'язково сюр'єктивним, тобто результуючий вектор може бути розрідженим.

5. `Gather f` – працює подібно до `Scatter`, але відображення відбувається в зворотному напрямі - індекси вихідного вектора мапуються на індекси вхідного.
6. `SHPointwise f` – "приміняє" унарну функцію `f` до кожного елемента вектора
7. `SHBinOp f` – "приміняє" бінарну функцію `f` до пар елементів, взятих з першої та другої половини вхідного вектора відповідно.
8. `SHInductor n f` – послідовно застосовує функцію f до вхідного елементу n разів.
9. `liftM_HOperator hop` – "піднімає" оператори мови *HCOL* на *Σ-HCOL* рівень.
10. `HTSUMUnion sop1 sop2` – оператор вищого порядку, що застосовує два оператори до одного вхідного вектора та поєднує їх результати.
11. `SafeCast sop` – оператор вищого порядку, що огортає інший *Σ-HCOL* оператор. Не змінює результати обчислення, але додає монадний інтерфейс обробки властивостей розрідженості.
12. `UnSafeCast sop` – подібний до `SafeCast`, але використовує іншу монадну обгортку.
13. `IUnion f (fam: {x:nat|x<n} → SHOperator)` – послідовно застосовує індексовану сім'ю з n операторів до вхідного вектора та поєднує результати за допомогою бінарної функції f. Цей оператор є абстракцією над паралелізмом на рівні циклу (англ. loop-level parallelism).
14. `IReduction f (fam: {x:nat|x<n} → SHOperator)` – подібний до IUnion, але не припускає непересічності наборів непорожніх

індексів.

15. `SHCopmose sop1 sop2` – функційна композиция операторів.

У якості прикладу, розглянемо виконання *Σ-HCOL* оператора `HTSUMUnion`. Це оператор вищого рівня, параметризований двома операторами: `f` та `g`. `HTSUMUnion` застосовує обидва оператори до єдиного заданого вхідного вектора, та поєднує результати цих двох обчислень, використовуючи операцію об'єднання векторів, як показано на **Рис. 2.**

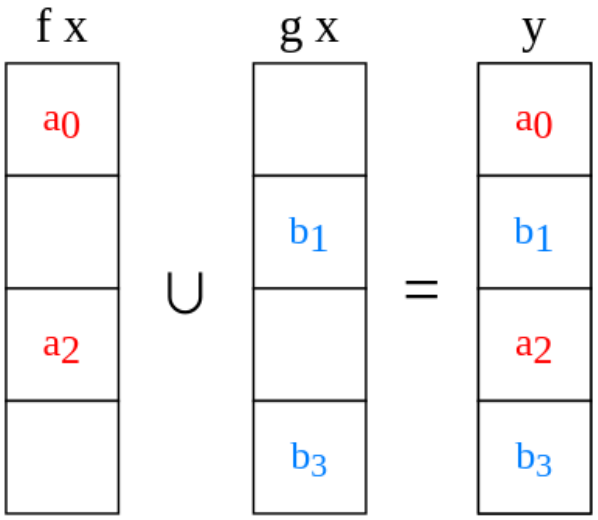

Рис. 2. `HTSUMUnion` в *Σ-HCOL*

У структурно коректному *Σ-HCOL* виразі, непересічність наборів індексів заповнених елементів цих двох векторів є гарантованою (доведеною у Coq). Таким чином, ситуація, в якій програма спробує об'єднати дві непорожні комірки, ніколи не виникає. Ця гарантія (як і властивість структурної коректності) є доведеним інваріантом попередніх кроків ланцюжка HELIX.

4. МОВА MHCOL

Розріджені вектори *Σ-HCOL* є абстракцією над блоками пам’яті. Явне відображення цього зв’язку є основним завданням проміжкової мови *MHCOL* (“M” відповідає англ. “memory” - пам’ять). Кожен блок пам’яті представлений в ній у вигляді асоціативного масиву, ключами в якому виступають натуральні числа, відповідні зсуву в пам’яті, а значеннями - елементи *типу-носія*, відповідні значенням пам’яті на даному зсуві. Порожньому елементу розрідженого вектора в такому представленні не відповідає жоден ключ.

Приклад одного розрідженого вектора в обох представленнях наведений на Рис. 3. На ньому зображено розріджений вектор з трьома ініціалізованими елементами, A, B, C, та відповідний йому асоціативний масив з трьома ключами.

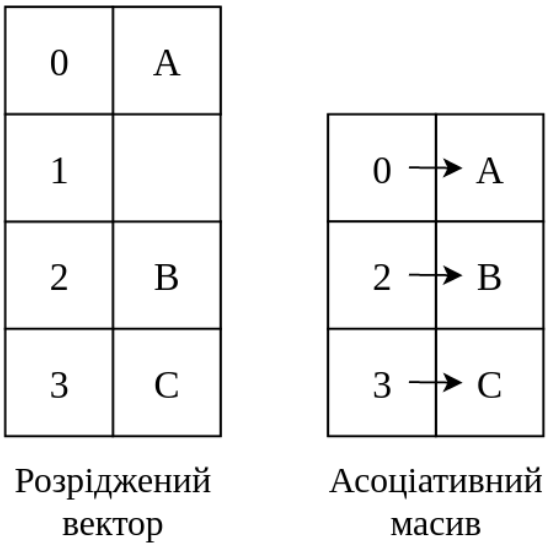

Рис. 3 Розріджений вектор представлений асоціативним масивом

На **Рис. 4** представлено схему роботи оператора `HTSUMUnion` в *MHCOL*. Кожен з двох операторів `f` та `g`, застосований до вхідного вектора

`x`, повертає відповідний асоціативний масив. Масиви мають непересічні набори ключів: відповідно, [0;2] та [1;3], що дозволяє об’єднати їх у фінальний масив `y`.

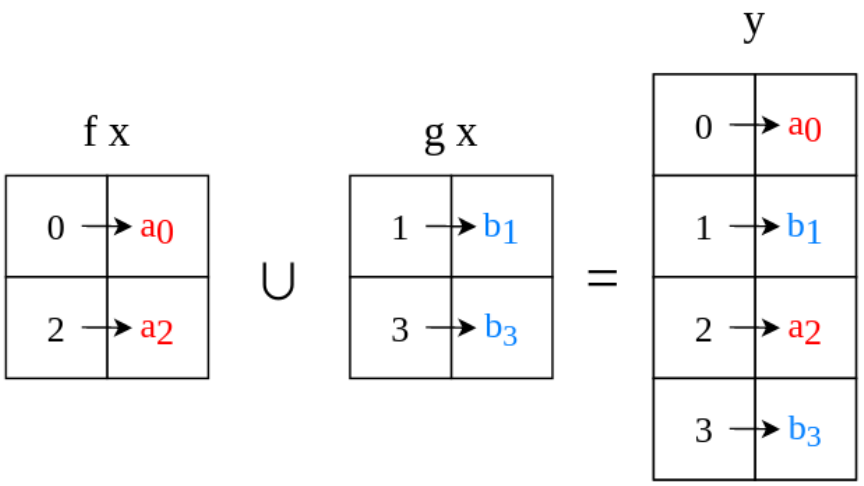

Рис. 4 `HTSUMUnion` в *MHCOL*

Така зміна представлення даних дозволяє перейти від високорівневого алгебраїчної природи *Σ-HCOL* ближче до апаратного рівня. В цьому представленні конкретне значення має бути пов’язано з ключем асоціативного масиву, перш ніж до нього можна отримати доступ. Спроба отримати значення за неініціалізованим ключем є помилкою (винятком). Таким чином, оператори мови *MHCOL* можуть повертати винятки та мають тип `mem_block → option mem_block`. Тим не менш, однією з наших гарантій при перетворенні структурно коректної *Σ-HCOL* програми на *MHCOL* є відсутність спроб доступу до порожніх комірок пам’яті, тобто, при виконанні, програма, отримана з коректного *Σ-HCOL* оператора, не повертатиме винятків.

Подібно до *Σ-HCOL*, в *MHCOL* для представлення операторів використано змішаний рівень вбудови. Наступна структура мови Gallina відповідає *MHCOL* оператору:

```
Record MSHOperator {i o : N} : Type :=
     mkMSHOperator {
          mem_op: mem_block → option mem_block;
          mem_op_proper:
               Proper ((equiv)=⇒(equiv)) mem_op;
          m_in_index_set: FinNatSet i;
          m_out_index_set: FinNatSet o;
     }.
```

Він параметризований розмірностями вхідного та вихідного блоків пам'яті. Поля включають: функцію, що імплементує операцію на блоці пам'яті (функція може зіткнутися з винятковою ситуацією, повертаючи `None`); доведення того, що ця функія є належним морфізмом [8] з точністю до сетоїдної еквівалентності (необхідне через абстрактність *типу-носія*); дві множини індексів, що задають структури доступу до вхідного та вихідного блоків пам'яті (відповідно до розрізненості векторів *Σ-HCOL*).

Кожен оператор мови *MHCOL* має відповідати певним обмеженням безпеки пам'яті. Ці обмеження виділені в окремий тайпклас, `MSHOperator_Facts`, та доведено, що кожен оператор його інстанціює. Цей підхід схожий на використаний з *Σ-HCOL*, але обмеження доводяться інші:

1. Будучи застосованим до блоку пам'яті, в якому заповнені всі комірки з множини `m_in_index_set`, `mem_op` ніколи не поверне помилку.
2. `mem_op` надає значення кожній комірці з множини `m_out_index_set` та не надає значення жодній поза цією множиною.
3. Блок на виході `mem_op` не має непорожніх комірок поза обмеженням розміру результату `o`.

Семантична еквівалентність пари операторів *Σ-HCOL* та *MHCOL* задається тайпклассом `SH_MSH_Operator_compat`. Він забезпечує однакові розмірності, відповідні множини вхідних та вихідних індексів, та структурну коректність обох операторів (через належність тайпкласам

`SHOperator_Facts` та `MSHOperator_Facts`). В додаток до цих властивостей, основне твердження семантичної еквівалентності має вигляд:

```
mem_vec_preservation:
∀ (x: svector i),
     (∀ (j: N) (jc: j < i),
       in_index_set sop (mkFinNat jc) →
       Is_Val (Vnth x jc)) →
     Some (svector_to_mem_block (op sop x)) =
     mem_op mop (svector_to_mem_block x)
```

Неформально цей предикат можна описати так:

Для будь-якого вектора, що відповідає контракту розрідженості входу оператора *Σ-HCOL*, застосування оператора *MHCOL* до такого вектора, перетвореного в блок пам'яті, має пройти успішно і повернути блок пам'яті, що дорівнює блоку, отриманому прямим перетворенням вектора-результату оператора *Σ-HCOL* на блок пам'яті.

Для звичайних операторів, `SH_MSH_Operator_compat` доводиться напряму. Для операторів вищого рівня, твердження ґрунтуються на припущеннях виконання `SH_MSH_Operator_compat` для всіх задіяних операторів. Деякі оператори потребують також додаткових припущень. Так, наприклад, для `HTSUMUnion` множини індексів `f` та `g` мають бути непересічними.

Перехід між *Σ-HCOL* та *MHCOL* імплементовано з використанням мови мета-програмування Coq, TemplateCoq [16]. Для перекладених програм, виконання обмежень `SH_MSH_Operator_compat` доводиться за допомогою автоматизованої системи. Це відповідає підходу валідації перекладу (англ. translation validation).

5. МОВА DHCOL

Наступна мова ланцюжка перекладів - *DHCOL* (“d” позначає англ. “deep-embedded” - глибоко вбудована). У той час як між операторами *Σ-HCOL* та *MHCOL* є відповідність один до одного, при переході на *DHCOL* ситуація дещо ускладнюється. *DHCOL* є низькорівневою мовою, тому кожен оператор мови *MHCOL* відповідає не одному, а комбінації з декількох *DHCOL* операторів. Ще однією особливістю є те, що *DHCOL* (вперше в ланцюжку) є глибоко вбудованою мовою. Але головною відмінністю на цьому кроці є *імперативність* мови *DHCOL*. На відміну від чисто функційної *MHCOL*, *DHCOL* покладається на середовище (набір змінних) та змінну пам’ять. Оператори цієї мови можуть мати сторонні ефекти, змінюючи стан пам’яті, але не можуть впливати на середовище.

На архітектурні рішення в імплементації *DHCOL* вплинула потреба в мові проміжній між декларативною MHCOL та низькорівневою LLVM IR. Вона не була побудована як мова загального призначення, та може має лише функції, необхідні для представлення *MHCOL* програм. Таке обмеження дозволило зменшити розмір мови, спростити її властивості та доведення тверджень про неї.

Наш попередній приклад, оператор `HTSUMUnion`, можна представити імперативно як послідовне виконання двох операторів та об’єднання їх результатів. Оскільки множини індексів вихідних блоків гарантовано не перетинаються, ці два оператори можуть бути обчислені незалежно один від одного (навіть паралельно), і записуватися в один (спільний) вихідний блок пам’яті, як показано на **Рис. 5.**, не турбуючись про ризик перезаписування результатів одне одного.

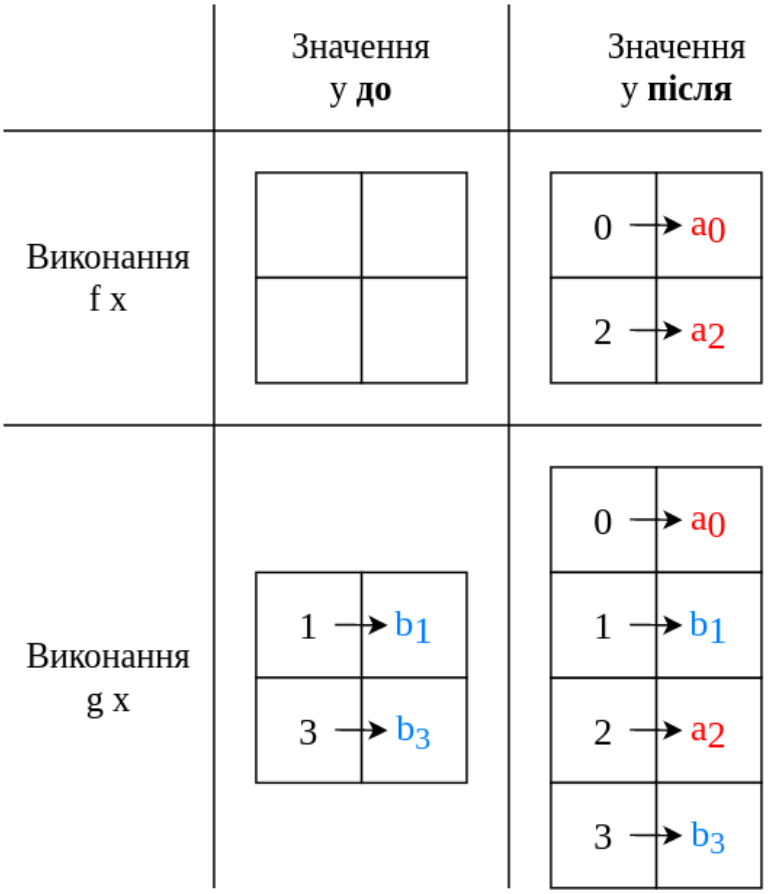

Рис. 5. `HTSUMUnion` в *DHCOL*

Нарешті, оскільки перед нами постає необхідність працювати з декількома блоками пам'яті одночасно, ми організуємо їх у *пам'ять*, яку представляємо у вигляді ще одного асоціативного масиву - цього разу зі значеннями, відповідними цілим блокам. Така дворівнева ієрархічна модель пам'яті схожа на модель, що використовується проектами CompCert та Vellvm.

Окрім пам'яті, до архітектури мови додається також *середовище обчислення*, в якому зберігаються всі змінні в області видимості. Змінні є типізованими, та можуть зберігати натуральні числа, значення *типу-носія* та вказівки на пам'ять.

Мові *DHCOL* надається структурна (малокрокова) операційна семантика за допомогою функції `evalDSHOperator : evalContext →`

`DSHOperator → memory → fuel → option (err memory)`, яка, за заданим середовищем, станом пам'яті та оператором, обчислює новий (змінений) стан пам'яті.

Для спрощення доведення того, що оператор завжди припиняється (обчислюється за скінченну кількість кроків), ця функція приймає допоміжний параметр `fuel`, що зменшується на кожному рекурсивному кроці, аж до 0, після чого виконання гарантовано завершується. Варто зазначити: було доведено, що, при виконанні структурно коректної функції, `fuel` ніколи не "закінчується" - він не впливає на семантику функції та використовується лише для спрощення деяких доведень. З цієї ж причини `evalDSHOperator` ніколи фактично не повертатиме `None`.

Зауважимо: семантичний крок виражається як перехід між станами пам'яті. Середовище обчислення залишається незмінним, і побічні ефекти виконання операторів обмежуються зміною значень у пам'яті. Причиною цьому є побудова мови: змінні мають статичну область видимості та представлені у формі SSA (англ. single static assignment).

Перетворення *MHCOL* на *DHCOL* також імплементоване мовою Template Coq. На додаток до операторів, перекладаються також арифметичні вирази на натуральних числах та значення типу-носія. При перекладі виразів вводиться обмеження на арифметичні оператори, що підтримуються мовою.

4.2 Визначення

Вирази мови *DHCOL* складаються зі змінних середовища обчислення (доступ до яких організовано за індексами де Бройна), операцій (таких як + та -) та констант. Існують чотири типи виразів: `NExpr` - вирази на натуральних числах, `AExpr` - вирази на значеннях типу-носія, `PExpr` - вирази на вказівках пам'яті та `MExpr` - операції над блоками

пам’яті. Перші два, `NExpr` та `AExpr`, обчислюють натуральні числа та значення типу-носія відповідно. Два інших, `PExpr` та `MExpr`, обидва повертають блоки пам’яті. Відповідно, змінні середовища набувають значень одного з трьох типів: натуральні числа, значення типу-носія чи вказівники на блоки пам’яті.

Існують 10 операторів мови *DHCOL*, визначених у вигляді наступного індуктивного типу:

```
Inductive DSHOperator :=
| DSHNop (* порожній оператор *)
| DSHAssign (src dst: MemVarRef)
  (* запис комірки пам’яті *)
| DSHIMap (n:N) (x_p y_p: PExpr) (f: AExpr)
  (* індексована функція [map] *)
| DSHBinOp (n:N) (x_p y_p: PExpr) (f: AExpr)
  (* [map2] над двома половинами блока під [x_p] *)
| DSHMemMap2 (n:N) (x0_p x1_p y_p: PExpr) (f: AExpr)
  (* [map2] над двома окремими блоками*)
| DSHPower (n:NExpr) (src dst: MemVarRef)
           (f: AExpr) (initial: CT.t)
  (* рекурсивне застосування [f] *)
| DSHLoop (n:N) (body: DSHOperator)
  (* виконати [body] [n] разів.
     індекс циклу доступний у середовищі *)
| DSHAlloc (size:NT.t) (body: DSHOperator)
  (* виділити новий порожній блок у пам’яті та
     надати доступ до нього за вказівкою з контекста
     під час виконання [body] *)
| DSHMemInit (size:NT.t) (y_p: PExpr) (value: CT.t)
  (* заповнити індекси блока пам’яті від [0] до
     [size] заданим значенням [value] *)
| DSHMemCopy (size:NT.t) (x_p y_p: PExpr)
   (* копіювати блок пам’яті. перезаписує блок
```

```
      призначення, якщо він вже наявний *)
| DSHSeq (f g: DSHOperator)
   (* виконати [g] після виконання [f] *).
```

4.3 Збереження семантики

У той час як звичайним операторам мови *MHCOL* відповідають одиничні інструкції мови *DHCOL*, оператори вищого порядку перетворюються на комбінації декількох інструкцій. Наприклад, оператор `(MSHIReduction i o n z f op_family)` мови *MHCOL* перетворюється на наступну *DHCOL* програму:

```
DSHSeq
  (DSHMemInit o y_p z)
  (DSHAlloc o
    (DSHLoop n
      (DSHSeq dop_family
              (DSHMemMap2 o y_p' (PVar 1)
                          y_p' df)))))
```

Параметри `MSHIReduction` наступні: розмірності вхідного та вихідного векторів (`i` та `o` відповідно), розмір `n` сім'ї операторів `op_family` та початкове значення `z`. У *DHCOL*, `df` та `dop_family` відповідають власне функції `f` та сім'ї `op_family`.

Оператори `DSHAlloc` та `DSHLoop` вводять дві нові змінні в середовище: вказівник на щойно виділений блок пам'яті та індекс циклу. Всередині циклу доступ до них можливий за індексами де Бройна `(PVar 1)` та `(Pvar 0)`. Використовуючи індекс циклу, `dop_family` виконує на кожній ітерації відповідний оператор з сім'ї та записує результат у виділений тимчасовий блок. Припускається, що результат обчислення `MSHIReduction` записується у блок під вказівником зі змінної `y_p`, а `y_p'` -

та ж змінна, лише зі збільшеним на 2 індексом де Бройна (для врахування двох нових змінних - індекса циклу та вказівника на новий блок).

Необхідно довести, що переклад з мови *MHCOL* в *DHCOL* зберігає семантику оператора. Подібно до інших переходів ланцюжка HELIX, використаємо підхід автоматизованої *валідації перекладу*. Для автоматичного доведення коректності перекладу, потрібно довести твердження про коректність та відповідність кожного оператора мови *MHCOL* та відповідного йому набору інструкцій *DHCOL*. Після чого такі твердження можна буде застосовувати рекурсивно, ієрархічно спускаючись структурою виразу.

Першим кроком буде формалізація поняття семантичної еквівалентності між чисто функційною мовою з денотаційною семантикою (*MHCOL*) та імперативною мовою з оперативною семантикою (*DHCOL*). Кожен оператор мови *MHCOL* являє собою функцію `x ↦ y`, де `x` та `y` є блоками пам’яті. Це *чиста* функція без сторонніх ефектів, результат котрої залежить від `x` та інших змінних контексту. З іншого боку, представлення цього оператора мовою *DHCOL* є імперативною програмою, що може зчитувати змінні з *середовища обчислення*, а також зчитувати та змінювати стан пам’яті. Один блок пам’яті відповідає “вхідному” `x`, інший блок - “вихідному” `y`.

Тим не менш, будь-яка програма мови *DHCOL*, що є імперативним представленням чистої функції, в деякому сенсі зберігає цю властивість - вона може змінювати лише свій “вихідний блок” в пам’яті - `y`. Формальне представлення класу *DHCOL* програм, що представляють чисті функції було визначено за допомогою тайпкласу `DSH_Pure`:

```
Class DSH_pure (d: DSHOperator) (y: PExpr) := {
  mem_stable: ∀ σ m m' fuel,
    evalDSHOperator σ d m fuel = Some (inr m') →
    ∀ k,
```

```
      mem_block_exists k m ↔ mem_block_exists k m';

  mem_write_safe: ∀ σ m m' fuel,
    evalDSHOperatorσd m fuel = Some (inr m') →
    (∀ y_i, evalPexp σ y = inr y_i →
     memory_equiv_except m m' y_i)
}.
```

Він складається з двох обмежень:

- *стабільність пам'яті* - оператор не може ані виділяти, ані звільнювати блоки пам'яті
- *безпека пам'яті* - оператор може може вносити зміни лише в блок пам'яті під вказівником зі змінної `y`, котра має бути коректною в середовищі `σ`.

Клас "чистих" програм *DHCOL* повністю покриває набір програм, з якими ми працюємо в ланцюжку HELIX та значно спрощує доведення тверджень про них.

Тепер можемо перейти до формулювання семантичної еквівалентності між оператором мови *MHCOL* та "чистою" *DHCOL* програмою. Оскільки *MHCOL*-частина цього відношення є функцією, маємо квантифікувати його над усіма можливими вхідними блоками. Оскільки оператори мови *DHCOL* зчитують та перезаписують пам'ять, вхідний та вихідний блоки оператора мають відповідати деяким існуючим блокам у пам'яті. Доступ до блоків пам'яті у *DHCOL* можливий лише за змінними-вказівниками, а тому додамо вимогу про існування в середовищі двох таких коректних змінних, що відповідають вхідному та вихідному блокам.

Семантичну еквівалентність означемо як тайпклас, параметризований власне операторами *MHCOL* та *DHCOL*, середовищем обчислення, та іменами змінних-вказівників на вхідний та вихідний блоки

у контексті. Крім того, чистота *DHCOL* оператора має також бути забезпечена через інстанціювання ним тайпкласу `DSH_pure`.

```
Class MSH_DSH_compat
  {i o:N} (σ: evalContext) (m: memory)
  (mop: @MSHOperator i o) (dop: DSHOperator)
  (x_p y_p: PExpr) `{DSH_pure dop y_p} :=
{
  eval_equiv: ∀ (mx mb: mem_block),
    (lookup_Pexp σ m x_p = inr mx) →
    (lookup_Pexp σ m y_p = inr mb) →
    (h_opt_opterr_c
      (λ md m' ⇒ err_p
        (λ ma ⇒
          SHCOL_DSHCOL_mem_block_equiv mb ma md)
        (lookup_Pexpσm' y_p))
      (mem_op mop mx)
      (evalDSHOperator σ dop m (estimateFuel dop)));
}.
```

Розглянемо наведене відношення більш детально. `h_opt_opterr_c` відповідає за обробку винятків. На відміну від `mem_op`, що має прості помилки (повертає `None` зіткнувшись з винятком), `evalDSHOperator` використовує дворівневу систему, розрізняючи нестачу “палива” (монада `option`) та інші помилки (монада `err`). Рівність виконується у випадку, коли оба оператора повертають помилку (за будь-яких причин), або коли оба спрацьовують коректно. У другому випадку на результатах обох має виконуватись вбудоване відношення. Це відношення (задане у вигляді лямбда-виразу) виконує додаткову перевірку винятків з предикатом `err_p` - щоб запевнити успішний доступ до блоку під змінною-вказівником `y_p` в `m'`. В кінцевому рахунку, відношення рівності зводится до виконання предикату `SHCOL_DSHCOL_mem_block_equiv` між блоками `mb`, `ma` та `md`.

Схему виконання операторів *MHCOL, DHCOL* та описане відношення між їх результатами наведено на **Рис. 6.** `σ` позначає середовище обчислення, `m` та `m′` - стани пам'яті відповіжно до і після виконання функції `evalDSHOperator`. `ma` відповідає блоку пам'яті під `m′` вказівником з `y_p`. `md` є результатом застосування *MHCOL* оператора до `mx`.

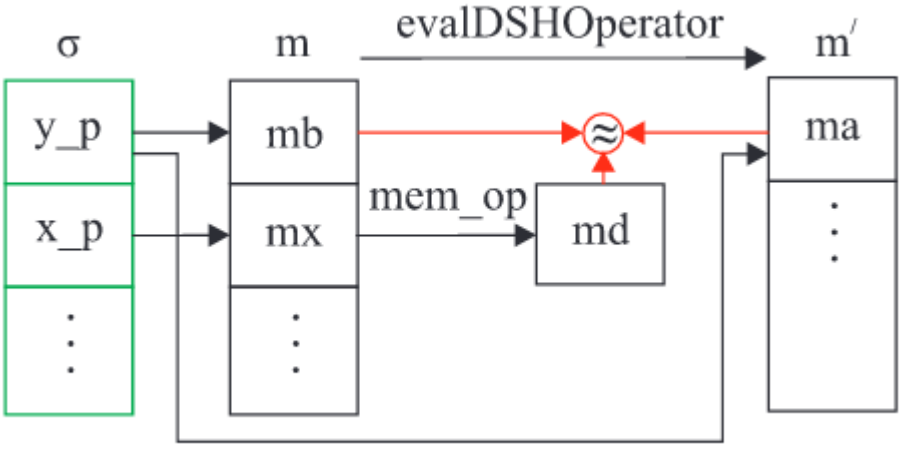

Рис. 5. `HTSUMUnion` в *DHCOL*

Нагадаємо: в мові *Σ-HCOL* розрізнені вектори є абстрактним представленням часткових обчислень. Порожні елементи вектора відповідають поки що не обчисленим значенням, заповнені - обчисленим. Об'єднання двох розрізнених векторів відповідає поєднанню декількох часткових обчислень в одне. Перехід від незмінних векторів до змінюваних станів пам'яті дозволяє нам замінити операцію об'єднання векторів простою операцією запису в пам'ять. Згідно з такими міркуваннями, блок-результат обчислення *MHCOL* оператора (названий на **Рис. 5.** `md` від англ. "memory delta" - "різниця в пам'яті") є таким блоком, що містить значення на тих позиціях, які потрібно оновити в пам'яті. Значення на всіх інших позиціях мають залишатися незмінними. З іншого боку, в *DHCOL* нам відомий стан пам'яті до виконання оператора та оновлений стан після

його виконання. Таким чином, `SHCOL_DSHCOL_mem_block_equiv` задає відношення між:

- `mb` - (від англ. “memory before” - “пам’ять до”) стан вихідного блоку в пам’яті *DHCOL* до початку виконання оператора
- `ma` - (від англ. “memory after” - “пам’ять після”) стан вихідного блоку в пам’яті *DHCOL* після виконання оператора
- `md` - значення змінених елементів блоку після обчислення оператора *MHCOL*

Це відношення імплементоване з використанням поелементного порівняння `MemOpDelta`, “піднятого” до блоків пам’яті через `SHCOL_DSHCOL_mem_block_equiv`.

```
Definition SHCOL_DSHCOL_mem_block_equiv
           (mb ma md : mem_block) : Prop :=
  ∀ i, MemOpDelta
        (mem_lookup i mb)
        (mem_lookup i ma)
        (mem_lookup i md).

Inductive MemOpDelta (b a d : option CarrierA)
  : Prop :=
| MemPreserved : is_None d →
                 b = a → MemOpDelta b a d
| MemExpected : is_Some d →
                a = d → MemOpDelta b a d.
```

Неформально його можна описати так:

Для кожного індекса блоку `md`, в якому присутнє деяке значення, значення під цим індексом в блоці `ma` має бути таким самим. Для кожного індекса блоку `md`, що не має значення, значення під цим індексом в блоці `ma` має залишатись рівним такому в блоці `mb`.

Після того, як виконання `MSH_DSH_compat` доведено для всіх загальних операторів мови *MHCOL* та відповідних їм *DHCOL* програм, доведення семантичної еквівалентності довільного *MHCOL* оператора та його перекладу мовою *DHCOL* може бути проведене автоматично через виведення інстанціювання ними цього ж тайпкласу. Рекурсія при цьому виконується за структурою *MHCOL* оператора. Така особливість пов’язана з тим, що відображення між *MHCOL* та *DHCOL* не є ін’єктивним, та переклад двох різних операторів мови *MHCOL* може створити дуже схожі за структурою *DHCOL* оператори, які буде складно розрізнити простим зіставленням зі зразком, в той час як структуру *MHCOL* розрізнити легко.

6. МОВА FHCOL

Робота з числами з рухомою комою представляє собою певний набір нових проблем. Щоб запобігти необхідності вводити числа з рухомою комою на самому початку ланцюжка та працювати з ними впродовж всіх перетворень HELIX, було введено абстрактний тип-носій даних, що використовувався до мови *DHCOL* включно. Врешті, перехід до чисел з рухомою комою виконано з уведенням ще однієї проміжкової мови - *FHCOL* (“F” від англ. “floating-point”), що оперує над числами у форматі, заданому стандартом IEEE 754. *FHCOL* і *DHCOL* використовують однакову модель пам’яті, з єдиною відмінністю - *FHCOL* використовує конкретні значення IEEE 754 замість абстрактного типу-носія. Цей рівень все ще вищий за кінцевий - модель пам’яті Vellvm, фінального кроку ланцюжка, використовує байтові послідовності напряму. Тим не менш, модель *FHCOL* може бути напряму перекладена на таку.

Схожий підхід використовуємо і для арифметичних виразів, що задають індекси в пам’яті. Оскільки індекси є невід’ємними за означенням, в усіх мовах до *FHCOL* вони задаються абстрактними натуральними числами. На етапі ж *FHCOL* вони перетворюються на конкретні значення типу `int64`.

Перехід між мовами *DHCOL* та *FHCOL* є тривіальним, адже обидві мови імплементовані через інстанціювання одного модуля (*AHCOL*), лише параметризованого різними типами (натуральні числа, тип-носій - машинні цілі, числа з рухомою комою). Такий підхід дозволив з легкістю означити синтаксис та семантику обох мов без дублювання коду, а також дозволяє довести деякі леми для обох мов одразу. Перехід від *DHCOL* до *FHCOL* імплементований мовою Gallina, та, на відміну від інших кроків ланцюжка, не використовує TemplateCoq.

5.1 Доведення коректності з використанням чисельних методів

Співвідношення між результатами обчислення структурно подібних виразів у цих двох мовах можна описати методами чисельного аналізу, такими як межі помилок та чисельна стійкості.

Для цілих чисел достатньо простої перевірки границь для уникнення програмного переповнення. Обмеження арифметичних виразів можливо оцінити використовуючи обмеження їх складових. В *DHCOL* та *FHCOL* такими компонентами можуть бути виключно константи чи індекси ітерацій циклу. Оскільки вони обмежені постійними розмірностями циклів, програма може бути проаналізована на потенційне переповнення.

Для чисел з рухомою комою зазначимо три можливих підходи. Жоден з них не було імплементовано - вони представляють можливі напрямки майбутніх досліджень.

1. Загальне поширення невизначеності (англ. uncertainty propagation). У найзагальнішому випадку, може бути застосований підхід поширення невизначеності. На кожному рівні *FHCOL* оператора можуть бути підраховані межі похибки підрахунку. На жаль, у більшості випадків, межі похибки будуть надто широкими для практичної корисності результату.
2. Оцінка похибки конкретного застосування. Одним з основних застосувань HELIX є валідація кібернетично-фізичних систем. В таких системах часто існують додаткові обмеження, що здатні покращити нашу оцінку похибки. Такий аналіз може бути запроваджений користувачем та підключений до нашого кроку перетворення *DHCOL* в *FHCOL*.
3. Живе поширення невизначеності. Результат обчислення може бути представлений у вигляді інтервалу замість одного числа. Оскільки цей інтервал обчислюється для конкретних значень, він буде помітно

точнішим, ніж “холодна” оцінка похибки, описана в п. 1.

5.2 Компіляція в LLVM IR

Останній крок ланцюжка HELIX - компіляція програм мовою *FHCOL* у фінальне, наближене до машинного, представлення - мову LLVM IR. Компілятор з *FHCOL* в LLVM IR було імплементовано мета-мовою TemplateCoq. Для доведення семантичної еквівалентності між *FHCOL* програмами та їх IR версіями, покладаємося на проект Vellvm, що зосереджується на наданні формальної семантики власне LLVM IR. Доведення базується на концепції дерев інтеракцій (англ. interaction trees) [17], та його детальне обговорення виходить за рамки цієї роботи.

ВИСНОВКИ

Нами було успішно завершено передостанній крок доведення коректності ланцюжка HELIX - формально верифікованої системи генерації високоефективного коду, заснованій на SPIRAL.

Основним досягненням цієї роботи є демонстрація підходу до верифікації перекладу з чисто функційної предметно-орієнтованої мови змішаного рівня вбудови (*Σ-HCOL*) до глибоко вбудованої імперативної мови (*DHCOL*) з використанням проміжкової моделі (*MHCOL*).

Було поступово введено та доведено коректність: роботи з винятками, модель пам’яті та імперативного середовища обчислення. Наш підхід обмежує програми кожної мови набором властивостей: структурна коректність *Σ-HCOL,* безпека пам’яті *MHCOL*, та властивість чистоти програм *DHCOL*, та використовує ці обмеження для подальшого доведення кожного кроку перетворення. Доведені властивості переносять корисну інформацію про код на різних кроках у зручній та готовій до використання формі. Наприклад, уводячи обробку винятків в мову *MHCOL*, маємо змогу довести властивість, згідно з якою помилки ніколи не виникають в *MHCOL* програмах, отриманих зі структурно коректних *Σ-HCOL* операторів. Схожим чином, загальна *DHCOL* програма може змінювати довільні блоки в пам’яті, але ми можемо обмежитись простішою підмножиною усіх операторів *DHCOL* - тими операторами, що описують чисті функції та модифікують лише єдиний вихідний блок.

Виділимо два основних напрямки майбутньої роботи. Перший з них - завершення ланцюжка перетворень HELIX, додавання останнього кроку системи - генерації коду LLVM IR. Другий напрямок пов’язаний з доповненням набору гарантій нашої системи обмеженнями на точність

обчислень виразів у числах з рухомою комою. Можливі підходи для цього було описано у розділі 5.1.

ПЕРЕЛІК ДЖЕРЕЛ

1. P¨uschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M.M., Singer, B.W.,Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W.,Rizzolo, N.: Spiral: Code generation for DSP transforms. Proceedings of the IEEE 93(2), 232–275 (Feb 2005). https://doi.org/10.1109/JPROC.2004.840306
2. Franchetti, F., Low, T.M., Popovici, T., Veras, R., Spampinato, D.G., Johnson, J.,P¨uschel, M., Hoe, J.C., Moura, J.M.F.: SPIRAL: Extreme performance portability.Proceedings of the IEEE, special issue on “From High Level Specification to High Performance Code”106(11) (2018)
3. Franchetti, F., Voronenko, Y., P¨uschel, M.: Formal loop merging for signal transforms. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 315–326. PLDI ’05, ACM, New York,NY, USA (2005). https://doi.org/10.1145/1065010.1065048
4. Zaliva, V., Franchetti, F.: Formal verification of HCOL rewriting (2015), http://www.crocodile.org/lord/FormalVerificationofHCOLRewritingFMCAD15.pdf
5. Zaliva, V., Franchetti, F.: HELIX: A case study of a formal verification of high performance program generation. In: Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing. pp. 1–9. FHPC 2018, ACM, New York, NY, USA(2018). https://doi.org/10.1145/3264738.3264739, http://doi.acm.org/10.1145/3264738.3264739
6. Leroy, X., Appel, A., Blazy, S., Stewart, G.: The CompCert memory model, version2. Tech. rep., INRIA (2012)

7. Zhao, J.: Formalizing the SSA-based compiler for verified advanced program transformations. Ph.D. thesis
8. Cast´eran, P., Sozeau, M.: A gentle introduction to type classes and relations in Coq. Tech. rep., Technical Report hal-00702455, version 1 (2012)
9. Anand, Abhishek, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. "CertiCoq: A verified compiler for Coq." In The third international workshop on Coq for programming languages (CoqPL). 2017.
10. Kumar, Ramana, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: a verified implementation of ML." ACM SIGPLAN Notices 49, no. 1 (2014): 179-191.
11. T. C. development team, The Coq proof assistant reference manual. LogiCal Project, 2004. Version 8.0.
12. T. Coquand and G. Huet, The calculus of constructions. PhD thesis, INRIA,1986.
13. P. Martin-L¨of, “Notes on constructive mathematics,” *Almqvist &amp*, 1970.
14. M. H. B. Sørensen and P. Urzyczyn, “Lectures on the Curry-Howard Isomor-phism,” 1998.
15. Zaliva, V., Sozeau, M.: Reification of shallow-embedded DSLs in Coq with automated verification. CoqPL, Cascais, Portugal (2019)
16. M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha,N. Tabareau, and T. Winterhalter, “The metacoq project,”Journal of Auto-mated Reasoning, pp. 1–53, 2020.

17. L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic, “Interaction trees: representing recursive and impure programs in Coq,”Proceedings of the ACM on Programming Languages, vol. 4, no. POPL, pp. 1–32, 2019.
18. X. Leroy, “Formal verification of a realistic compiler,” Commun. ACM, vol. 52, pp. 107–115, July 2009.

ДОДАТКИ

Використані терміни

В даній роботі використовується низка позначень, запозичених з англомовної літератури, що не мають загальноприйнятих аналогів в українській мові. Наведемо визначення ключових серед них:

- **Вбудована мова** (англ. “embedded language”) - мова програмування, задана у вигляді бібліотеки іншої мови (загального призначення), що використовує синтаксис чи деяку підмножину мови-носія для задання власних елемнтів, таких як типи даних, функції, методи та ін.
    - **Вбудова глибокого рівня** (англ. “deep embedding”) - ситуація, в якій вбудована мова має власне означення абстрактного синтаксичного дерева. В такому випадку програма написана вбудованою мовою буде єдиною змінною мови-носія (деякого типу “АСД”).
    - **Вбудова мілкого рівня** (англ. “shallow embedding”) - вбудована мова використовує абстрактне синтаксичне дерево мови-носія. В такому випадку програмі, написаній вбудованою мовою, відповідатиме один чи декілька елементів мови-носія (часто функція вбудованої мови відображається напряму функцією мови-носія).
    - **Вбудова змішаного рівня** (англ. “mixed embedding”) - поєднання глибокого та мілкого рівнів у вбудові однієї мови. Мова, вбудована на змішаному рівні, може задавати власну структуру АСД, але при цьому використовувати в ньому деякі елементи мови-носія.

- **Розріджений вектор** (англ. “sparse vector”). В класичному розумінні, *розріджений вектор* - вектор, в якому багато елементів мають нульові (в абстрактній алгебрі - нейтральні) значення. В контексті цієї роботи, поняття розрідженого вектора використовується в дещо зміненому сенсі - нейтральні елементи вектора вважаються не нульовими, а відсутніми.
- **Валідація перекладу** (англ. “translation validation”) - метод забезпечення збереження семантики програми при перекладі. Замість доведення коректності власне перекладача (що означало б повне покриття доведенням усіх можливих програм), підхід валідації перекладу полягає в перевірці відповідності для кожної пари перекладених програм окремо та генерації, коли це можливо, формального доведення семантичної відповідності між ними.