TRIP REPORT

PLDI 2024

24.06.24 ~ 24.06.30 COPENHAGEN, DENMARK

흥미로운 발표들

KEYNOTE 1. THE FUTRE OF FAST CODE: GIVING HARDWARE WHAT IT WANTS

효율적인 연산을 위해 매우 복잡하고 이해하기 어려운 프로그래밍을 해본 적이 있는가? 멀티스레딩이나 CUDA 프로그래밍 같은 것들이 여기에 해당될 것이다. 이 발표의 목표는 하드웨어의 성능을 극한으로 끌어 올리려면 복잡하게 프로그램을 만들어야 하는 현 상황을 극복하기 위한 컨셉 제시였다.

첫번째는 복잡한 프로그램을 자동으로 증명할 수 있는 도구를 만드는 것이다. 두번째는 Domain Specific Langauge(DSL)를 디자인하는 것이다. 각 컨셉에 맞추어 Jonathan 교수님의 연구실에서 어떤 연구를 해왔고 현주소가 어디 쯤인지 알려주었다. 특히, 두번째 내용에 많이 공감하며 인상깊게 들었다. Domain Specific보다 더 디테일한 Data Structure Language가 필요하다고 주장하였다. 발표에서 든 예시는 DB Query 최적화였는데 DB 구성에 따라 효율적인 쿼리가 정해지는 사실을 생각해보면 아주 적절하다고 생각했다.

발표에서는 하드웨어에 집중하였지만 더 나아가면 복잡해지는 경향이 있는 모든 분야에 통용되는 말이라고 생각이 들었다. PLDI 2024에 참석하게 된 것도 Graph를 위한 DSL을 제시한 연구였던 걸 생각해보면 Data Structure Language는 점점 트렌드가 되어가지 않을까 싶다.

PAPER 1. ASSOCIATED EFFECTS

논문 제목이 두 글자여서 지나칠 수 없었던 발표였다. 여기서 Effect는 흔히 알고 있는 Computational Effect를 뜻한다. 예를 들어, 예외 처리나 객체 상태 변화 혹은 입출력 등이 있다. 논문에서 제시하는 Associated Effect란, Type Class를 정의할 때 Effect를 고려하는 member를 미리 정의해두어 Type Class의 instance를 만들 때 instance에 맞추어 Effect를 계산하는 메커니즘을 뜻한다. 예를 들어, 사칙연산 Type Class가 있다고 해보자. Int로 해당 클래스를 instantiate하면 DivisionByZero Effect를 계산해야 한다. 반면에, Float으로 instantiate하면 해당 Effect는 더 이상 고려할 필요가 없다 (부동소수점으로 인해). 이러한 Effect를 유저가 직접해야 하는 것이 아닌 언어 차원에서 지원해주는 것이 이 논문의 핵심이다.

더 놀라운 점은 Associated Effect를 구현한 언어가 Flix라는 언어였는데, Flix라는 언어는 덴마크의 Aarhus University에서 만든 언어였던 것이다. 이 연구도 Aarhus University에서 진행이 되었다. 다시 말해, 학교 차원에서 언어를 디자인하고 필요한 feature들을 연구하여 구현 후 논문으로 내는 것이었다. 이 프로젝트는 덴마크의 주요 기관 중 하나인 Independent Research Fund Denmark (IRFD)에서 지원 받아 진행된 프로젝트라고 한다. 즉, 덴마크 국가로부터 지원을 받아 Flix라는 언어가 탄생했다는 것이다. 국가 차원에서 고유한 문제를 가지고 있으며 그 위에서 다양한 문제를 해결하고 있는 모습이 굉장히 인상 깊었다.

PAPER 2. DIFFY: DATA-DRIVEN BUG FINDING FOR CONFIGURATIONS

이 논문은 모두가 공감할 수 있는 어려운 문제인 Configuration의 버그를 찾는 것이었다. 발표는 한국인의 밤에 참석하느라 듣지는 못했지만, Abstract가 인상 깊어서 논문을 직접 찾아보았다.

Configuration 버그 찾는 것이 매우 어려운 이유는 정답이 정해져 있지 않기 때문이다. 따라서, 기존 연구에서는 타입오류만 찾는다거나 특정 key-value 조합만 고려하였다. 그러나, 이 연구는 기존 허들을 넘어 복잡한 구조도 포함한 arbitrary json에 대해 가능한 방법을 제시하고 있다.

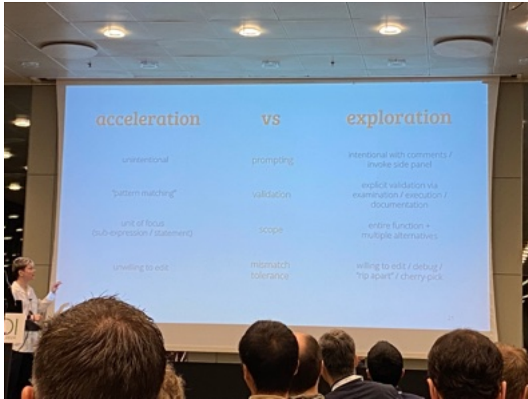
직관은 단순하다. 버그는 “이상행동”이라는 직관 하에 해당 문제를 해결하였다. 이상행동을 탐지하기 위해 해당 프로그램에서 사용된 Configuration 파일들을 다 모은다. 그 후, Configuration을 대표할 수 있는 Template을 생성한다. 해당 Template은 Configuration에 존재하는 특정 구조들을 abstract하여 그 구조가 가질 수 있는 값들을 counting 해둔 형태이다. 예를 들어, {“Path”: “/var”}라는 configuration이 100개 있고 {“Path”: True}라는 configuration이 1개 있다고 가정하자. 그러면 Template은 {“Path” : [{“CASES”: 100, “TEMPLATE”: “/var”}, {“CASES”: 1, “TEMPLATE”: True}]} 이런 식으로 만들어진다.

Template을 생성했다면, 각 구조마다 지배적인 값이 존재할 것이다. 이 연구에서는 지배적인 값을 올바른 행동이라 간주한다. 그 후, 소수의 값들에 대해 지배적인 값과 비교했을 때 비슷하다면 OK, 그렇지 않다면 직관 하에 버그라고 판단을 내린다.

현재 진행하고 있는 연구인 Pyinder에서도 “정답이 정해져 있지 않은 경우”가 너무 많았다. 그렇기에, 이 논문에서 사용된 디테일들을 적용해볼 수 있는 여지 또한 많았다. 나의 연구 세계를 더 발전시킬 가능성을 찾은 것 같았고, 또 이러한 문제를 많이 경험하고 있을 거 같은 여러 사람들을 위해 Trip Report에 남기기로 하였다.

KEYNOTE 2. AI-ASSISTED PROGRAMMING TODAY AND TOMORROW

AI를 활용하는 측면은 크게 두 가지: (1) Acceleration과 (2) Exploration가 있다.



Acceleration은 개발자가 개발 방법을 충분히 인지하고 있지만 이를 가속화 하기 위해 AI를 사용하는 경우이다. 패턴매칭의 경우가 대부분이며 한 개의 Statement 혹은 Expression을 합성하게 된다. 예를 들어, Ocaml으로 사칙연산 모듈의 to_string을 작성한다고 해보자.
| Add -> “+”
| Sub -> “-“
| Mul -> “*”
까지 작성한 뒤 물 한 모금을 마신 상황을 가정해보자. 그러면 Copilot은 자연스레 | Div -> “/”를 추천해줄 것이고 개발자는 자신이 충분히 할 수 있지만 ‘오 대박’ 라고 생각하며 Copilot의 코드를 사용할 것이다. 이러한 상황처럼 AI를 이용하는 것이 Acceleration이다.

반면에, Exploration은 개발자가 언어나 방법에 대해 익숙하지 못하여 의도적으로 AI에게 코드 생성을 맡기는 상황이다. 예를 들어, Python의 라이브러리 matplotlib으로 plot을 그린다고 생각해보자. 옛날이나 일일이 검색해가며 코드를 작성했지, 지금은 ChatGPT나 Copilot한테 스펙을 알려주고 코드를 받아 사용하는 것이 일반적인 선택이다. 이 경우 Acceleration과는 달리 여러 줄의 Statement를 한꺼번에 받게 된다. 이처럼 개발자가 개발 방법을 모르는 상황에서 도움을 받기 위해 코드 전체 합성을 AI로 하는 것이 Exploration이다.

키노트 발표에서 지적한 문제는 두 가지 사용법 모두 개발자가 검증하고 디버깅하는 시간이 꽤나 오래 걸린다는 사실이었다. 설문조사 결과 ChatGPT의 52% 정도의 답변은 틀린 답이었고 (Kabir et al, CHI’24), 더 재밌는 사실은 (특히 보안 쪽에서) AI가 틀린 답변을 대체로 주었지만 사용자는 대체로 그것이 올바른 답변이라고 오해했다는 것이다 (Perry et al, CCS’23). 즉, AI-Assisted를 사용할 때는 검증/디버깅을 잘 도와주는 작업이 추가적으로 요구된다는 것이다.

본 키노트 발표가 굉장히 유의미하며 공감이 되었는데, 현재 같이 진행하고 있는 LLM-based Python Library Migration 연구에서도 똑같은 문제에 봉착했기 때문이다. 현재 LLM code를 어떻게 검증할지 혹은 정답처리를 어떻게 할지 고민을 하고 있는 상황이다. 그냥 뭉갤까 싶었는데, 이게 우리에게만 국한된 문제가 아닌 모두가 다 같은 고민을 하고 있는 좋은 연구 주제임을 확인할 수 있었던 발표였다.

PAPER 3. EQUIVALENCE BY CANONICALIZATION FOR SYNTHESIS-BACKED REFACTORING

해당 논문은 앞서 말한 LLM-based Python Library Migration에 일부 사용할 수 있을 법한 아이디어를 제시해준 논문이었다. 문제는 간단하다. Component-based Refactoring 도중 합성 중인 Sketch가 Input 프로그램과 같은 프로그램인지의 여부를 어떻게 판단할까였다. I/O spec은 unsound한 문제가 있으며 Logical Checking은 Spec 추론이 어렵다는 고전적인 문제가 존재한다. 따라서, 이 논문에서는 Syntax를 활용한 Equivalence Checking을 시도하였다.

Syntatic Equivalence를 확인하기 위한 방법으로 Canonicalization을 제안한 것이 본 논문의 키포인트였다. 방법론은 이러하다. Input과 Sketch를 Canonicalization을 통해 비슷한 형태로 만든다. 이 때, Canonicalized Sketch에는 Hole이 존재할텐데 이 Hole들을 Canonicalized Input에 대응되어 채워 넣을 수 있으면 동일한 프로그램이라는 것이다. 다시 말해 Canonicalized Sketch로 Canonicalized Input를 만들 수 있으면 이는 동일한 프로그램일 것이며, Sketch에 존재하는 Hole들을 대응되는 Input Component로 치환하면 되는 것이다. 이 때, Canonicalization은 General한 방법은 존재하지 않고 Domain마다 직접 정의해야하는 한계가 있긴 하다.

Migration도 일종의 Refactoring이라고 생각해보면 해당 방법을 충분히 사용할 여지가 있다고 생각했다. 특히, 본 논문에서 Python API Refactoring도 가능함을 보였기 때문에 (numpy에 국한되어 있지만) Canonicalization만 잘 정의한다면 연구에 큰 진척을 보일 수 있을 것이라 생각했다.

PAPER 4. SCALING TYPE-BASED POINTS-TO ANALYSIS WITH SATURATION

해당 발표는 추후 Pyinder를 연구하며 발견한 문제들에 충분히 활용할 여지가 있어 Trip Report에 남겨두려 한다.

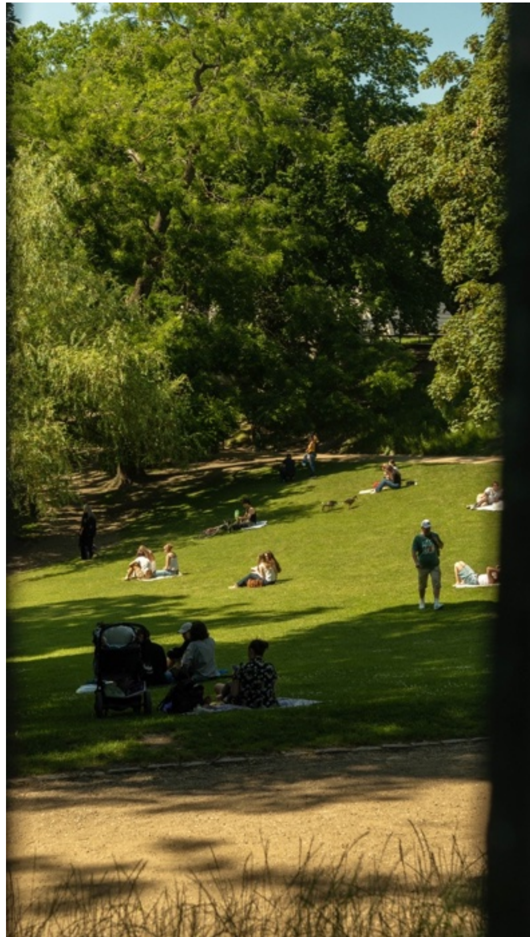
이 논문에서는 두 가지 포인트를 제시한다. (1) allocation-site가 아닌 type-based points-to analysis를 활용하여도 real-world Java에서 충분히 유의미한 precision을 유지한다는 사실과 (2) type-based points-to analysis를 효율적으로 (O(n)) 사용하는 방법인 Saturation을 소개하는 것이다. 특히, Saturation의 경우 변수마다 모든 타입을 유지하는 것이 아닌, precision을 충분히 유지할 정도로만 타입 정보를 유지하는 것이 핵심인 아이디어이다. 이 때, 유지하지 않는 타입 정보는 Saturated Information이라고 뭉쳐서 관리하게 된다. 핵심은 어떤 정보를 남기고 어떤 정보를 뭉칠 것이냐는 선택 부분인데, 자바 bytecode에 대한 사전지식이 없어 이해하기가 조금 어려웠다. High-level 자체는 우리 연구실에서도 진행했던 idea였고, 특히 Pyinder에서도 Type-based 분석을 하고 있는데 잘 버무릴 수 있을 거 같아 기록으로 남겨 두었다.

덴마크

밤은 언제예요?

북유럽의 나라답게 여름철 해가 떠 있는 시간이 너무너무너무 길었다. 해가 오전 5시쯤 떠서 오후 11시쯤 지는데, 시간 개념이 없어지는 거 같아 미쳐버리는 줄 알았다. 해가 쨍쨍한데 저녁을 먹으러 다니는 묘한 기분을 견디기 어려웠다. 집에 냉장고를 열어 콜라인 줄 알고 마셨던 게 간장이었을 때 배신감 같은 느낌이었다. 덴마크는 행복지수가 굉장히 높은 나라로 알려져 있는데, 도대체 어떻게 그럴 수 있는지 궁금했다.

휘게 라이프 (HYGGE LIFE)



덴마크는 휘게 라이프로 높은 행복도를 유지하고 있는 나라라고 한다. 휘게란 이런 것이라고 한다. 대낮에 공원에서 누워 책 읽기, 밤에 모닥불 앞에 앉아 코코아, 이불 덮고 에어컨 틀기, 삼겹살에 소주, 치킨에 콜라, 수육과 육전에 막걸리.

진짜 놀랍게도 평일 낮에 큰 공원을 가보면 사람들이 벌거벗고 썬텐하고, 돗자리 펴고 피크닉을 하고, 누워서 책을 읽고 있다. 여유로움의 끝판왕이며 낭만이 넘치는 도시였다. 참고로 이 낮은 아침 9시도 물론이고 저녁 8시도 당연히 포함이다. 덴마크 사람들은 일 안하나 싶어 검색을 해보았는데 석유가 나온단다. 바로 납득이 갔다.

낭만에 미쳐버린 나라에서 나도 한 번 도전해 보았다. 평일 낮에 공원에서 햇빛 받으며 벤치에 앉아 책 읽기. 내 옆은 같이 학회에 참석한 연구실 친구이기도 했고, 느린 걸음걸이로 데이트하는 노부부도 있었으며, 웨딩 촬영을 하는 신혼 부부도 있었다. 꽤나 괜찮았다.

또 다른 휘게도 도전했다. 숙소에서 맥주 마시며 다 같이 게임하기. 경쟁 게임이 아닌 모두가 한 마음 한 뜻이 되어 적을 물리치는 게임이었다. 나의 답답한 플레이에 다른 사람들이 가끔씩 짜증도 내긴 했지만, 마지막에 한 목소리로 “이제 간다!”를 외칠 때의 쾌감은 휘게가 분명했다.

그런데 덴마크의 삶에는 크나큰 치명적인 문제가 있었다.

유명한 음식 있어요? 호밀빵이요. 추천 음식 있어요? 호밀빵이요. 색다른 음식은요? 호밀빵이요.

덴마크 추천 음식이 호밀빵 밖에 없다는 사실이었다. 호밀빵에 베이컨, 호밀빵에 연어, 호밀빵에 코울슬로… 탄수화물의 왕인 쌀에 절여진 나에게는 조금 힘든 싸움이었다. 다행히 학회에서 제공되는 식사에는 호밀빵 대신 다양한게 나와서 생존할 수 있었다 (그리고 연구실 사람이 가져온 라면 덕분에).

매슬로우가 제안한 욕구 이론에 의하면 하위 욕구를 충족하지 못하면 상위 욕구를 충족하지 못한다고 한다. 해가 주구장창 떠있거나 혹은 달이 주구장창 떠있는데 수면욕이 제대로 충족되는지 의문이었다. 또, 시그니처 음식이 호밀빵 밖에 없는데 식욕도 제대로 충족되는지 의문이었다. 휘게 라이프는 사실 생존본능이 아니었을까.

결국은 행복한 나라 덴마크

앞의 말은 농담 반 진담 반이었고, 이러니 저러니 해도 덴마크 삶의 양식은 마음에 들었다. 여유로움에서 나오는 행복, 그리고 행복에서 나오는 여유로움의 선순환. 바다를 바라보며 다 같이 맥주를 마시는 노인들, 강가에서 다이빙 하며 놀던 청년들, 자전거에 깃발 꽂고 달리고 있는 아이들. 이 글을 읽는 모두에게 휘게의 축복이 내려지길 바라며 글을 마친다.

마지막으로, PLDI 2024에서 뜻 깊은 경험을 하게끔 도와준
오학주 교수님, 전민석 박사님, 그리고 연구실 사람들에게 감사의 인사 올립니다.