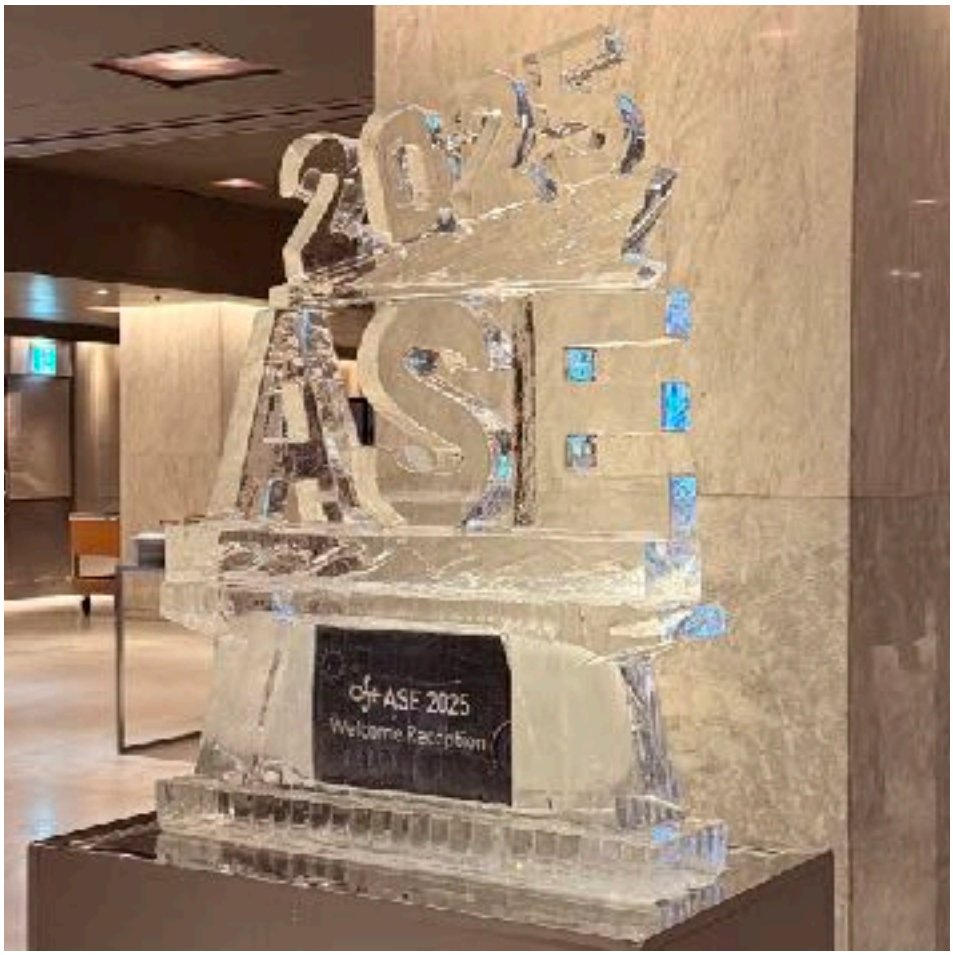
Trip Report for Attending ASE25



2025. 11. 17. ~ 11. 19.

최민석

ASE (IEEE/ACM International Conference on Automated Software Engineering)는 소프트웨어 공학 학술대회로, 프로그램 분석, 설계, 테스트, 유지보수의 자동화가 주요 관심사이다. 올해 서울에서 40번째 ASE가 개최되어 참석할 수 있게 되었다. 이번에는 주로 Distinguished Paper를 받은 것들 위주로 발표를 들었다.

Characterizing Multi-Hunk Patches: Divergence, Proximity, and LLM Repair Challenges 이 연구는, 기존 APR 연구가 대부분 단일한 수정 위치를 가정했기 때문에, multi-hunk bug(여러 코드 영역을 동시에 수정해야 하는 버그)가 APR 연구에서 충분히 다뤄지지 않았다는 점을 문제로 삼고, 이를 위한 데이터셋을 만들고 (LLM) multi-hunk 평가 도구를 개발했다. 각 bug의 수정 난이도를 측정하기 위해 hunk divergence라는 metric을 정의하고, multi-hunk 패치의 난이도를 metric에 따라 분류해 각 난이도마다 LLM이 버그를 얼마나 잘 해결하는지 측정했다. 가장 성능이 높은 모델도 multi-hunk 버그 중 약 1/4 정도만 수리할 수 있었다. 논문은 결국 multi-hunk 패치에서는 단순 local code transformation 이상의 semantic coordination이 필요하며, 현재 LLM은 이를 수행할 수 있는 구조화된 추론 능력이 부족하다는 결론을 제시한다. 실험에 사용한 가장 성능이 높은 모델이 o4-mini 였다는 점이 좀 아쉽긴 했지만, 어려운 multi-hunk 패치를 위해서는 컨텍스트를 더 많이 고려해야 된다는 점에서, 컨텍스트 크기 제약이 고질적인 문제인 LLM이 multi-hunk를 잘 처리하지 못한다는 결론 자체는 타당하다고 느껴졌다. 이를 보면서 한 가지 떠오른 고민은, 이 coordination을 도와주는 lightweight 분석이 patching 성능을 개선할 수 있지 않을까 하는 점이다. 예를 들어 스칼라처럼 강력한 overloading과 implicit resolution이 있는 언어에서는 특정 호출이 어느 정의를 가리키는지 같은 것들을 표시해주는 것만으로도 모델이 hunk 간 semantic 연계를 더 잘 추론할 수 있게 도울 가능성이 있을 것 같다. 의존성을 가진 코드들임을 알려주는 힌트가 주석으로 제공된다면, 난이도가 높은 multi-hunk 케이스에서 LLM의 실패율이 줄어들 수도 있지 않을까 하는 생각이 든다.

Seeing is Fixing: Cross-Modal Reasoning with Multimodal LLMs for Visual Software Issue Repair 이 연구는 기존 APR 연구가 대부분 텍스트 기반 코드 입력만을 고려했다는 점을 문제 삼는다. 그러나 프론트엔드 UI 개발에서는 문제가 텍스트가 아니라 화면 상에서 시각적으로 드러나는 경우가 많다. 이 연구는 이러한 관찰을 바탕으로, UI 화면과 코드 컨텍스트를 동시에 입력하여 문제를 이해하고 코드를 수정하는 방법을 제안한다.
평소 프론트엔드 개발에 관심이 있는 편이라 연구가 흥미로웠다. 예전에 연구실에서 CSS differential testing을 농담처럼 이야기한 적이 있었는데, 이 연구는 그 가능성을 꽤 진지하게 현실화한 사례처럼 보였다. 여러 브라우저와 UI 렌더링 조합을 켜놓고 differential testing을 수행하면 visual bug를 포착할 수 있지 않을까?

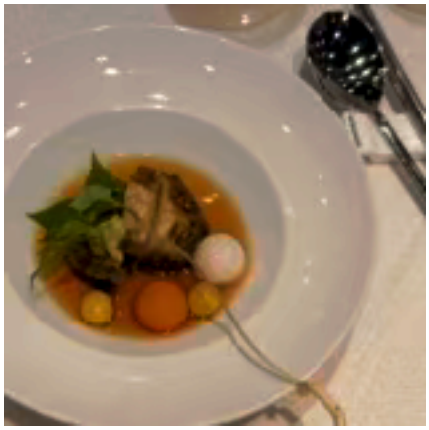
Finding Bugs in MLIR Compiler Infrastructure via Lowering Space Exploration 이 연구는 MLIR의 lowering 과정에서 한 프로그램이 여러 pass 경로를 통해 낮은 수준의 IR로 변환될 수 있다는 점에 주목했다. lowering equivalence가 성립하는지 확인하기 위해 연구자들은 lowering space exploration이라는 접근을 제안하였다. 방법론 자체는 잘 이해하지 못했지만, 대체적으로 Lowering Space의 크기가 매우 크기 때문에 봐야하는 Space의 크기를 효과적으로 줄이는 방법에 대한 것이었다. 결과적으로, 40개의 미확인 버그를 발견했다.

개인적으로, 예전에 PLDI에서 컴파일러 optimization 옵션 조합에 따른 실행 성능 차이를 분석한 연구를 본 적이 있었다. 그 당시 “optimization pass의 순서 조합 자체를 탐색하는 방식은 왜 나오지 않았을까?” 라는 생각을 했었는데, 이 논문이 바로 그 접근을 실제로 수행한 사례라는 점에서 흥미로웠다. 의미 보존을 가정하는 변환 규칙들의 조합이 모두 일관적인지 확인한다는 점에서 transpiler rule correctness 검증과도 비슷한 면이 있다고 생각한다. 차후에 논문을 직접 읽어보면, JavaScript transpiler나 codemod 기반 refactoring 도구의 correctness 검증에도 유사한 구조가 적용될 수 있을지 탐색해 볼 수 있을 것 같았다.

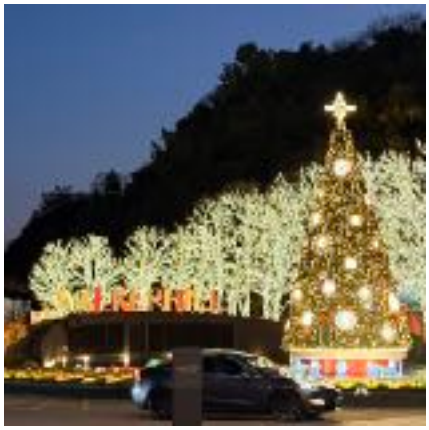
LSPFuzz: Hunting Bugs in Language Servers 코드 편집기에서 고급 언어 기능을 제공하는 Language Server에 대한 버그를 탐지하는 연구이다. 일반적인 컴파일러와 달리 LSP 서버는 프로그램과 더불어 사용자 상호작용이라는 추가적인 입력을 가지기 때문에, 프로그램 텍스트만을 변형하는 fuzzing으로는 핵심 로직까지 도달하기 어렵다는 관찰에서 출발한다. LSPFuzz는 이를 해결하기 위해 2-stage mutation을 사용하는데, 첫 단계에서 문법 기반으로 소스 코드를 (전형적인 방법으로) 변형하고, 두 번째 단계에서 hover 등 LSP operation을 적용하여 상호작용 기반 버그를 유발한다.
LS를 대상으로 버그를 찾겠다는 아이디어 자체가 좋은 연구라는 생각이 들었다. LS는 나도 매번 쓰고 ESMeta IR에 대해서 LS를 만들어보려고 짧게나마 LSP에 대해 이것저것 찾아봤었는데, 그 과정에서 LS를 테스팅하는 것 자체가 매우 어려웠어서 이 연구가 지적한 testing difficulty가 현실적인 문제라는 점에 공감이 갔다. 나도 얼추 겪어본 문제이지만 연구로 생각해보진 않았는데 이런 점은 좀 배울 수 있으면 좋을 것 같다. 얼마 전에 한창 논문 읽을 때도 SpecGen이나 Laurel 같은 연구들이 어떻게 보면 단순히 LLM을 활용해 Program Generation/Reapir를 하면서도 Software Verification이라는 도메인에 잘 붙인게 유효하다고 생각했는데, 이 연구 Language Server에 대해 버그를 찾겠다는 데에서 출발한 뒤로는 자연스럽게 방법론이 확장될 수 있었겠다는 생각이 들었다.

TEPHRA: Principled Discovery of Fuzzer Limitations 이 연구는 여러 연구에도 불구하고 fuzzer가 커버리지에 있어 자주 비슷한 지점을 뚫지 못한다는 점에서, “실패하는 입력을 발견하는 것이 fuzzer 자체의 근본적 진전을 이끈다”는 문제의식을 출발점으로 삼는다. 스택 기반 언어를 사용해 입력 공간을 정의하고, 다양한 속성을 갖는 syntactic/semantic “hard cases”를 자동으로 생성하여 fuzzers가 어디에서 실패하는지를 관찰한다.
흥미로웠던 지점은 이 연구가 fuzzer의 성능을 평가하는 방식 자체에 새로운 관점을 제시했다는 점이다. 일반적으로 fuzzer 연구는 동일한 프로그램 집합을 기준으로 “누가 더 좋은 coverage를 달성했냐”로 비교되지만, 이 논문은 입력 구조의 난이도를 점진적으로 높여가며, 어떤 조건에서 다양한 fuzzer가 강하거나 약한지를 보여주는 방법론이라는 점에서, 기존 fuzzing 연구가 벤치마크를 기반으로 서로를 평가하는 방식 자체에 문제를 제기한다. 다른 연구들이 Evaluation 수치를 개선하고자 할 때 이에서 벗어나 fuzzing 매커니즘 자체를 발전시키기 위한 더 나은 평가 방식을 제안한다는 면에서 매우 본질적인 연구라는 생각에 인상적으로 느껴졌다.

실시간으로 음악을 작성하는 라이브 코딩(Live Coding)



연회에서 제공된 식사



학회 장소 주변 모습