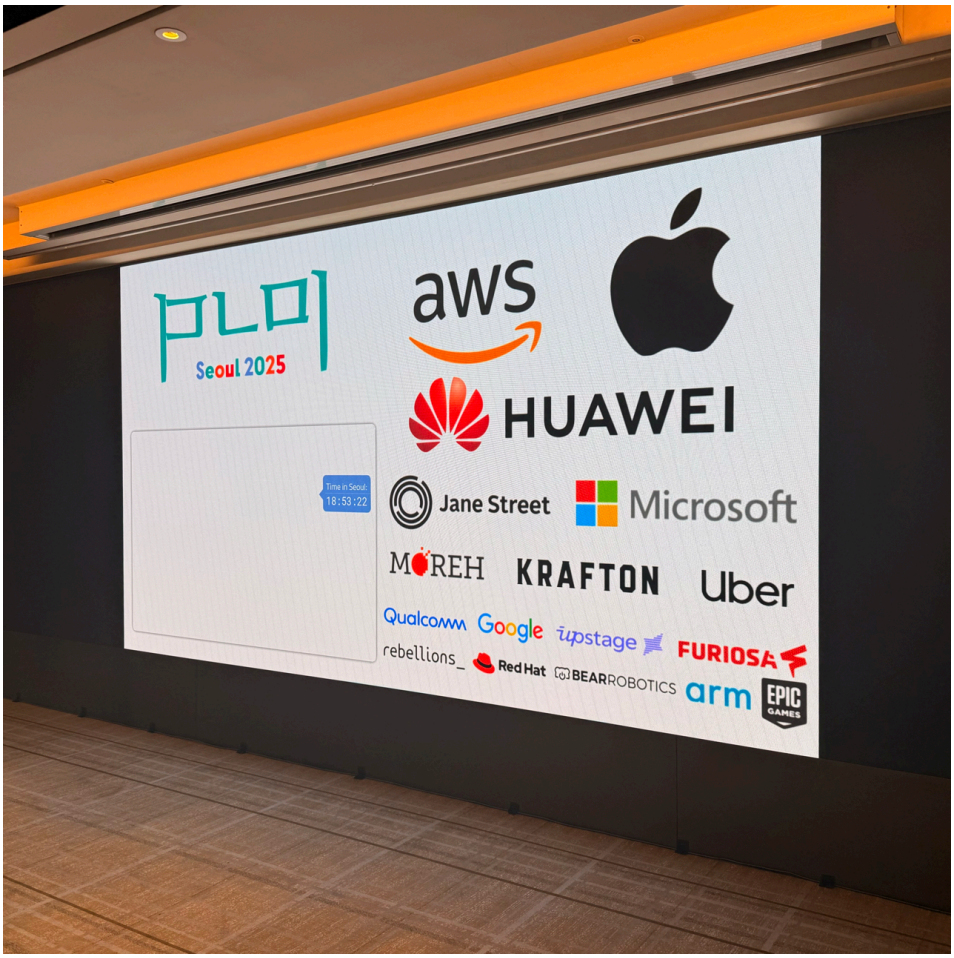
Trip Report for Attending PLDI25



2025. 06. 16. ~ 06. 20.

최민석

PLDI (Programming Language Design and Implementation) 는 언어 설계 및 구현, 컴파일러, 프로그램 분석 및 최적화, 정형 검증 등을 다루는 연례 학술대회이다. 대한민국 서울에서 46번째 학회가 열렸다. 지난 해에 이어 올해에도 직접 들을 기회를 얻게 되었다. 지난해보다 1년간 이것저것 보고 들은 것이 있기 때문에 작년보다 더 보람찬 참여였던 것 같다. 참가하면서 인상 깊었던 몇가지 발표나 행사를 소개한다.

Co-located Workshops - 튜토리얼 세션: 파이썬 바이트코드 디컴파일레이션 툴 PyLingual이라는 Python 바이트코드 디컴파일레이션 도구에 대한 튜토리얼 세션에 참여했다. Python 바이트코드 파일(pyc)이 주어지면 원본 Python 코드를 찾아주는 것이 목표이다. 이 도구는 CPython 바이트코드의 다양한 명세 버전에 자동으로 대응하고, 원본 Python 파일과 syntactic하게 동일한 "퍼펙트 디컴파일레이션"을 목표로 한다는 점이 특징이었다.
디컴파일 분야에 대해 아는 건 없이 Python이라는 이유만으로 들었던 세션인데, 다버전 대응이라는 문제를 해결하는 방식이 흥미로웠다. CPython의 바이트코드 명세는 버전마다 호환에 대한 고려 없이 바뀌는 편이라 지금까지 대응이 쉽지 않았다. PyLingual은 다버전에 대응하기 위해 바뀌는 명세를 데이터 기반으로 처리할 수 있도록 NLP 모델을 사용한다. NLP 모델의 학습 데이터는 주로 오픈소스 프로젝트로부터 얻은 코드를 CPython으로 컴파일하여 얻은, 자동으로 얻은 데이터로 되어있다. 모든 부분을 자동으로 처리하지는 않는데, 변수 이름, 상수, 메서드 이름 등에 NLP 모델이 휘둘려서 지나치게 많은 학습을 필요로 하는 문제를 막기 위해 학습 데이터를 마스킹 토큰으로 처리해주고, 번역 후 컨트롤 플로우를 복원하는 부분은 CFG로부터 CDG(Control Dependency Graph)를 그려 복원한다.
흥미로웠던 점은, PyLingual이 넓은 범위에서 Mechanized Specification을 떠올리게 했다는 점이다. 물론 ESMeta나 SpecTec같은 (내가 아는) Mechanized Spec 연구는 주로 전통적인 PL 방법을 활용하기 때문에, 인공지능을 사용하지는 않는다. 하지만 PyLingual에서는 " '문서화가 잘 되지 않은, 여러 버전의 구현체(명세)'에 동시에 대응해야 한다는 문제"를 자동으로 풀기 위해, NLP와 기존 PL 방법론(CDG 등)을 적절히 결합하여 자동화하고 있다는 점에서 Spec스러운 문제를 적절한 방식으로 풀고 있다는 인상을 받았다.

Co-located Workshops - RPLS 둘째 날에는 RPLS (Reproducible and Practical Language Systems) 워크샵이 진행되었으며, 이 워크샵에서는 현실 세계에서 사용되는 프로그래밍 언어들의 명세를 기계적으로 표현하고 자동화하는 다양한 프레임워크들이 소개되었다. 이 날은 현장 참석은 하지 못했지만, 발표는 스트리밍으로 시청하였다. SpecTec 관련 발표도 들었는데, 이전에 몇 번 발표를 접한 적이 있음에도 불구하고, 이번에는 특히 더 잘 이해가 되었다. 아마도 여러 번 반복해서 들은 효과와 함께, 류석영 교수님의 발표가 명확하게 잘 정리되어 있었던 점이 크게 작용한 것 같다.
인상 깊었던 부분은, 명세(specification)라는 것이 단순한 기술 문서가 아니라 그 생태계가 가진 문화, 철학, 요구사항까지 반영하고 있다는 점이었다. 예를 들어 어떤 명세는 표준화 기구에 의해 엄격히 관리되고, 어떤 명세는 커뮤니티 주도로 유지되며, 어떤 명세는 애초에 공식적인 governance 자체가 명확하지 않기도 하다. 각 명세마다 '어떤 식으로 접근해야 할지'의 기준과 분위기가 다르다는 점이 잘 와닿았다. 이전에는 명세를 일종의 기술 문서로만 인식했던 것 같은데, 이번 발표들을 통해 그 안에 담긴 조직 구조, 커뮤니티 운영 방식, 역사성까지 고려해야 한다는 시각을 얻을 수 있었다.

Partial Evaluation, Whole-Program Compilation 기존 인터프리터를 거의 수정하지 않고 컴파일러로 변환할 수 있는 방법인 weval을 제안한다. 기존 partial evaluation은 인터프리터를 특정 DSL로 작성

해야 하거나, JIT를 사용하는 경우가 많아 실제 산업 환경에 적용하기 어려운 한계가 있었다. 반면 이 방법은 C/C++ 기반의 상용 인터프리터 코드를 이용해도 해당 인터프리터를 컴파일러로 전환할 수 있음을 보여준다. SpiderMonkey 자바스크립트 엔진의 WebAssembly 포팅 버전을 대상으로, 벤치마크에서 모든 기법을 적용한 결과 최대 x2.5배의 성능 향상을 보였다고 한다.
Partial evaluation(부분 실행)을 연구에 사용해보기 위해 잠깐 공부해본 입장에서, 그 내용을 현실(?)에 이렇게 잘 적용할 수 있다는게 대단해보였다. 인터프리터를 부분 실행하기 위해 program counter-sensitive한 방식으로 분석하는 것 자체는 당연한 건데, 그걸 상용 인터프리터(SpiderMonkey) 등에 실제로 가능하도록 한 게 인상적이다. 내가 ESMeta위에서 부분실행을 구현할 때는 메모리 위에서 돌아다니는 자료구조를 잘 처리하는게 까다로운 일 중 하나였는데, 이 논문에서는 모든 자료구조에 대해 까다롭게 분석하지 않고, load_register(index), store_register(index, value) 같은 intrinsic을 인터프리터에 (annotation을 달 듯이) 달아줘서 자동화를 살짝 포기한 대신 매우 좋은 정확도와 성능 향상을 얻은게 그게 가능한 이유였을 것 같다.
지금까지 부분실행이 이미 다 죽은 오래된 개념이 아닐까 생각했었는데 지난번에 José 교수님이 얘기한 것도 그렇고, 이번 PLDI25 학생 포스터에 이걸 활용하는 방법에 대해 포스터가 나온 것도 그렇고, '아직까지 할 게 많은 쪽이었나? 그냥 하면 됐나?' 싶기도 하다. 부분 실행이라는 개념 자체가 매우 매력적이라고 생각해서 아직 그 쪽으로 연구해보고 싶은 마음이 남아 있는데 SpiderMonkey에 어떻게 적용한건지 구현체도 보고 좀 자세하게 읽어보고 싶은 마음이 든다.

Exploiting Undefined Behavior in C/C++ Programs for Optimization C/C++ 언어의 Undefined Behavior가 실제로 컴파일러 최적화에서 얼마나 성능 향상에 기여하는지를 분석한 경험적 연구이다. LLVM 컴파일러를 수정하여 UB 관련 최적화를 각각 비활성화할 수 있는 플래그를 구현하고, 이를 기반으로 다양한 실제 오픈소스 C/C++ 프로그램(총 730만 LOC)을 대상으로 성능 및 바이너리 크기 변화를 측정하였다. 일반적으로 UB를 활용한 최적화는 성능에 큰 영향을 주지 않으며, 대부분의 성능 저하는 컴파일러의 소규모 개선을 통해 다시 성능 향상이 가능했다고 한다, 오히려 가끔은 UB에 대한 최적화가 잘못된 수정을 유발해 보안 취약점의 원인이 되기도 한다. 실험적으로 UB의 실효성을 입증함으로써 안전성과 성능 간의 균형에 대해 다시 생각해보게 만든 의미 있는 발표였다.
정확한 연구 주제는 기억이 안나지만, 예전에 교수님이 프로그램의 다양한 옵션 조합을 바꿔가면서, 그에 따른 실행 속도였는지 Fuzzing과 관련된 성능 향상이었는지를 보는 연구에 대해 말씀하신 적이 있는데, 이 논문도 비슷한 방향에서 실증적으로 접근한 사례라는 느낌을 받았다. 아마 실험 결과가 일반적으로 알려진 직관에 반대된다는 점이 PLDI에 붙을 수 있었던 중요한 점이지 않았을까 싶다.

Relaxing Alias Analysis: Exploring the Unexplored Space 본 논문은 컴파일러 최적화에서 Alias Analysis에 대해 기존과는 반대되는 관점을 제시하는 연구이다. 기존 연구들이 alias 정보를 가능한 정밀하게 추론함으로써 컴파일러가 더 효과적인 최적화를 수행할 수 있다고 가정해 온 반면, 이 논문은 이 가정이 반드시 성립하지 않음을 실험적으로 보인다. 저자들은 LLVM 컴파일러 등에서 alias 정보를 의도적으로 완화함으로써 그 영향력을 분석하였다. 그 결과, 전체 alias 쿼리 중 약 3%만이 최종 바이너리에 영향을 주며, alias 분석을 완전히 제거해도 대부분의 경우 실행 시간과 바이너리 크기에 큰 변화가 없다는 사실을 발견했다. 저자들은 그 이유가 alias 정보가 컴파일러 최적화 패스에서 앞에서는 이득을 준 것으로 보이지만 뒤에서 일어날 수 있는 최적화를 방해하기 때문이라고 한다. 예를 들어 Dead Store

Elimination(DSE)은 두 포인터가 alias라고 판단되면 앞선 store를 제거하지만, 이로 인해 벡터화가 제한되어 오히려 성능 저하로 이어질 수 있다.
내 생각에는 앞선 'Exploiting Undefined Behavior in C/C++ Programs for Optimization' 와 비슷하게 컴파일러 내의 최적화 패스를 다양한 조합으로 수정했을 때 미치는 영향도 알아야 할 것 같은데, 그렇지 않고 단순히 'alias를 대충 알아도 좋을 수 있어'라고 결론 내리기에는 성급한 결론일 수도 있을 것 같다는 생각이 들었다. 컴파일러 내의 최적화 패스 순서를 바꾸는 것 자체가 엔지니어링적으로 수고스러운 일이라 안 한 게 아닐까 싶은데, 이 순서를 다양하게 자동화해서 성능이나 최적화에 미치는 영향을 파악하는 경험적 연구를 빠르게 해봐도 좋을 것 같다는 생각이 들었다.

Program Synthesis From Partial Traces Program Synthesis from Partial Traces라는 문제를 정의하고, 이 문제를 해결하기 위한 합성 기법 Syren을 제안한다. 기존의 program synthesis 연구들은 완전한 입출력 예시 또는 명시적 사양을 필요로 했던 반면, 본 연구는 API 호출 등의 side-effect만 포함된 불완전한 trace들만으로도 프로그램 전체를 복원하고 일반화된 스크립트를 생성할 수 있음을 보였다. 핵심 기법은 (1) 모든 입력 trace를 재현하는 초기 프로그램을 구성한 뒤, (2) 프로그램 재작성(rewriting) 및 (3) syntax-guided synthesis를 통해 조건문, 반복문, 숨겨진 함수 호출 등을 점진적으로 삽입하는 방식이며 사용자가 정의한 비용 함수에 따라 프로그램 품질을 최적화할 수도 있다.
본 연구의 가장 큰 독창성은 "부분 로그만으로도 프로그램 합성이 가능하다"는 점을 보였다는 점인 듯했다. 학부 인턴을 처음 시작했을 때 구현체로부터 스펙을 자동으로 합성하는 연구 주제를 받았었는데, 이 연구도 매우 비슷한 방향이라는 생각이 든다. JavaScript 객체에 Proxy를 적용해 실행 trace를 수집하고, 그로부터 일반화된 동작 스펙을 유도하는 방식과도 자연스럽게 접목될 수 있어, JavaScript 스펙 합성에 바로 접목할 수 있을 것 같다는 생각이 들었다. 물론 방법을 그대로 가져다 쓰기만 하면 논문이 되지는 않겠지만, 나중에 스펙 합성에 도전하겠다는 학생이 있으면 이 논문도 참고 자료로 추천해주면 좋지 않을까 싶다.