Oregon Programming Summer School 2010

June 15-25, Eugene, Oregon, USA

오레곤은 녹색이 진정 잘 어울리는 주였습니다. 울창한 숲을 어디서든 만날 수 있습니다. 유진은 도시지만 시골 느낌에 더욱 가까운 곳이었습니다. 곰이 어슬렁 나타나 연어를 사냥할만한 분위기의 강이 있고 숲속을 달리는 느낌을 주는 경치 좋은 자전거 도로가 잘 되어 있는 곳입니다. 하루가 지나자 왜 이런 곳에서 여름학교가 열리는 지 알 것 같더군요. 수업이 끝나면 산책과 자전거 타기 외에는 할 일이 없는 장소입니다. 공부하기 좋은 유진에서 PL 여름학교가 열렸습니다. 같이 다녀온 학주나 현승이가 모든 수업에 대해서 비슷한 양으로 정리를 했는데 저는 관심이 많았던 수업 위주로 정리를 하고 싶습니다.

Type and Proof Theory Foundations – Frank Pfenning and Robert Harper
Software Foundation in Coq – Andrew Tomach and Benjamin C. Pierce

이번 여름학교에 주된 내용은 단연 Coq을 배우고 활용하기 였습니다.

사실 Frank Pfenning, Robert Harper 교수님의 이론적인 강의도 Coq에서 사용하는 기본적인 논리에 대해서 설명하는 내용이었습니다. Pfenning의 강의는 이광근 교수님께 수업시간에 배운 내용과 겹치는 부분이 많아서 듣기가 쉬었습니다. 논리를 어떻게 정의해야 할까라는 기본 물음에서 차근차근 자연 연역 (natural deduction)을 만들어 가는 과정을 보여줬습니다. 또,

Curry-howard 동일성 (isomorphism)을 건설 논리 (constructive logic)과 람다 계산 (lambda calculus)의 연관성으로 보여줬습니다. 예로, 람다 계산의 함수 요약 (lambda abstraction)은 로직의 implication (A -> B)에 해당하고, lambda calculus에서 일어나는 치환이 논리에서는 A와 A -> B를 알때 B를 알 수 있다는 것에 해당한다는 사실도 강의했습니다. 증명하는 과정은 논리를 분석하는 과정이라는 말씀을 했습니다. Robert Harper 교수님은 프로그램 식을 실행하여도 타입이 변하지 않는다는 사실 (preservation lemma)과 타입을 가지고 있는 식은 실행을 마친 값이거나 실행을 할 수 있다는 사실 (progress lemma)을 증명했습니다. 또 종료 증명 (termination proof)을 설명하면서 abstraction rule에 대해서 증명하는 것의 어려움 (open term에 의한)을 강의했습니다. 이는 제가 다단계 프로그램 (multi-staged program)의 타입을 가지고, 보통 프로그램을 다단계 프로그램으로 변환하는 연구를 할 때 겪었던 어려움과 비슷한 문제였습니다. 함수의 경우 인자가 뭔지 모르는 상태에서 함수의 어떤 성질을 증명해야 하기에 어렵더군요. 이런 모든 내용이 추후에 Tomach와 Pierce의 강의를 통해 Coq으로 다시 한번 어떻게 구현이 되고, 증명이 되는지를 확인 할 수 있었습니다. 대부분의 강의가 유기적으로 짜여져 있어서 집중하기에 좋은 커리큘럼이었습니다.

Tomach와 Pierce의 강의는 책 Software Foundation in Coq을 통해 진행이 되었습니다. 훌륭한 교재이면서 그 자체가 Coq 프로그램이라서 Coq을 처음 접하는 사람들도 쉽게 따라갈 수 있게 잘 만들어져 있습니다. Coq을 배우기를 원하는 사람 누구에게나 내용도 이해하면서 Coq 프로그래밍도 배울 수 있는 진정 훌륭한 교재입니다. 기본적인 Coq의 작동 방식을 하나하나 알기 쉽게 배우는 귀중한 자리였습니다. 앞에 들었던 수업들과 어떻게 연관이 되는지 생각하면서 Coq에 대해 배울 수 있었습니다. 중간 중간에 연습문제를 해결해가는 재미도 있습니다.

교재: http://www.cis.upenn.edu/~bcpierce/sf/

Coq을 직접 접하기 전에는 Coq이 자동 증명기인 줄 알고 있었다가, Coq을 조금 써보고 나서는 증명 보조기인데 거의 사람이 다 해야하는구나라는 느낌을 받았었습니다. 그런데 여름학교에서 강의를 들어보니 사람이 해야하는 일을 많이 줄여주는 전략 (tactic)들을 많이 배울 수 있었습니다. 또, 프로그램 관련한 증명을 많이 하다보면 자주 쓰게 되는 구조에 대한 가설 (induction hypothesis)을 자동으로 찾아 주고, 증명의 모든 경우를 놓치지 않게 체크해 주는 등 유용한 기능을 많이 제공한다는 사실도 알게 되어 제 편협한 생각을 고치는 좋은 기회가 되었

습니다. Pierce는 간단한 증명이라서 당연하다 여기고 쉽게 넘어간 경우 실제로는 그렇지 않아 증명에 오류가 있는 경우가 많고, Coq이 그런 것들을 놓치지 않고 잘 찾아주어 손으로 증명할 경우 생길 수 있는 오류를 Coq은 미리 찾아줄 수 있다고 했습니다.

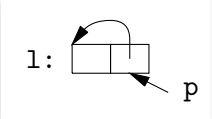
Coq으로 간단한 프로그램을 정의하고, 그 프로그램 타입 시스템을 만들어서 옳다는 것을 증명했습니다. 타입 시스템 뿐 아니라 형식 의미 (formal semantics)를 정의하고, 간단한 프로그램 변환 (constant propagation이나 짝수 값만을 보전하는 프로그램)이 프로그램의 의미를 바꾸지 않는다는 사실을 증명하기도 했습니다. 학부 PL 시간에서 프로그램을 정의하고 해석기 (interpreter)를 구현하고 간단한 분석을 구현해보는 강의 내용을 Coq을 이용해서 복습하는 재미가 있었습니다.

Coq을 배우면서 알게된 이해가 가지 않던 사실 중 하나는 $(\forall P,\ P \vee \neg P)$ 을 증명할 수 없다는 것이었습니다. Coq의 건설 논리로 위의 사실을 증명한다는 이야기는 P나 P의 거짓 중의 하나를 증명할 수 있다는 것인데 둘 중에 어떤 것도 증명할 수 없기 때문에 증명을 못하는 것입니다. 증명 가능하다와 참이다가 엄연히 다르다는 것을 평소에는 자주 혼동하고 있는 것 같습니다.

Proving a Compiler: Mechanized Verification of Program Transformations and Static Analyses - Xavier Leroy

개인적으로 여름학교 수업 중에 가장 흥미롭고 인상적이었던 발표였습니다. 컴파일러가 올바른지를 Coq으로 증명한 일들을 정리하는 발표였습니다. 실제로 C 컴파일러를 새로 만들었는데 3-4년에 걸리는 작업으로 소스가 8천줄인데 Coq 증명이 무려 5만줄! 이라고 합니다. Xavier Leroy는 2006년부터 올해까지 POPL에 세편, PLDI에 한편의 논문을 발표하였는데 모두 이 컴파일러의 최적화(optimization)가 옳다는 것을 증명하는 일이었습니다.

프로그램이 하는 일이 무엇인지 알아내는 일! 제가 가장 관심 있는 일입니다. Xavier Leroy는 그보다 전에 프로그램이 자신이 의도한대로 제대로 컴파일이 되어서 돌아가는 지를 먼저 보이기 싶어했습니다. 이 일은 간단하지 않은 것이 아래와 같이 명백해 보이는 프로그램 변환도 때로는 틀린 변환이 될 수 있습니다.




```
struct list { int head; struct list * tail; };

struct list * foo(struct list ** p)
{
  return ((*p)->tail = NULL);          (*p)->tail = NULL;
                                       return (*p)->tail;
}
```

위의 그림과 같이 (*p)->tail이 p가 가리키는 주소가 되는 경우라면 오른쪽 프로그램은 없던 널 포인터 참조 (null dereference) 오류가 발생할 수 있습니다. 이렇듯 부작용 (side effect)이 쉽게 생길 수 있는 프로그램 변환이 그럼에도 불구하고 옳다는 것이 증명한 것입니다. 진심으로 경의를 표할 만한 일을 해냈습니다.

Xavier Leroy는 강의를 통해 Coq을 이용하여 프로그램 변환이나 분석을 수식화하고, 안전성을 증명하는 법을 가르쳐 주었습니다. Coq 파일들과 강의 자료들은 여기에 http://gallium.inria.fr/~xleroy/courses/Eugene-2010/ 있습니다.

자 이제 어떻게 컴파일러를 증명하는지 구체적으로 알아보겠습니다. 우선 간단한 프로그램(IMP)을 정의해야 하겠지요. 이것은 이미 앞서 Benjamin Pierce 강의에서도 했던 것입니다. 그리고, 여지없이 프로그램의 의미도 클 걸음 실행 의미 (big-step operational semantics) 형식으로 정의 합니다. 그리고, 바로 기계어로 컴파일해버리면 증명하기 어려우니까 간단한 가상 머신(virtual machine)으로 변환을 합니다. 가상머신위에서의 프로그램은 프로그램 카운터와 스택, 코드의 연속으로 이루어져 있습니다. 당연히 가상 머신의 명령문들의 의미도 정의를 해야합니다. 이 과정들은 모두 Coq 위에서 진행이 됩니다. 여기까지는 우리가 잘 알고 있는 전형적인 것들로 이루어져 있습니다.

우리는 이제 프로그램을 두 가지 방법으로 실행할 수 있습니다. 프로그램의 의미를 따라 AST를 방문하며 해석하는 방법이랑 프로그램을 컴파일해서 가상머신 위에서 실행하는 방법입니다. 이 둘 사이에 의미가 변하지 않는다는 사실을 증명해야만 합니다. 증명은 귀납법(induction)으로 비교적 간단하게 됩니다만 변환된 프로그램이 최종 상태 (final state)에 도달했을때에만 옳다는 것을 증명할 수 밖에 없다는 문제가 있습니다. 프로그램이 끝나지 않고 무한히 돌거나 중간에 잘못되는 경우가 발생하지 않다는 것을 증명하기 위해서는 더욱 자세한 기술이 필요합니다.

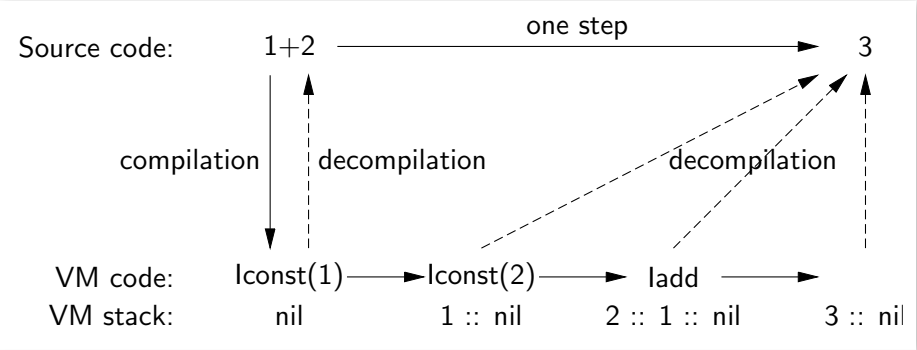
여기서 관찰가능한 행동양식 (observable behaviors)을 정의하게 됩니다. 중간 과정이 어떻든간에 무한히 도는 두 프로그램 혹은 에러를 발생시키는 두 프로그램은 같은 것으로 봅니다. 그리고, 프로그램의 동일성을 한번에 증명하는 대신 두 방향으로 나누어 증명을 합니다. 프로그램 P_1의 관찰가능한 행동양식이 변환된 프로그램 P_2의 그것을 포함하면 백워드 시뮬레이션(backward simulation - refinement)이라고 합니다. $\mathcal{B}(P_1) \supseteq \mathcal{B}(P_2)$ 컴파일 과정에서 몇몇 행동양식들이 제거된 것입니다.

Michael Sasa

아니... 그런데 프로그램 P_1에서 나타나던 결과가 변환된 프로그램 P_2에서 나타나지 않게 되는데 어떻게 프로그램 동일성을 증명한다는 것이지?라는 의문이 들었습니다. 근데 이런 느슨화는 꼭 필요한 일입니다. 이유는 컴파일러가 최적화를 하기 때문입니다.

`x := 1 / y; x := 42` (goes wrong if y = 0)	optimized to	`x := 42` (always terminates normally)

위와 같은 변환 예를 보면 처음 프로그램은 프로그램이 잘못 될 수가 있습니다. 하지만, 변수 x가 바로 재정의 되기 때문에 최적화를 통해 첫 할당문은 제거가 가능합니다. 그런 경우 두 번째 프로그램은 실행이 잘 끝나게 되지요. 이와 같이 두 프로그램의 행동양식이 완전히 같지 않더라도 우리가 원하는 안전한 컴파일러의 성질을 잘 지키는 경우가 있습니다.



이제 backward simulation을 증명하려고 하는데 문제가 또 있습니다. 원래 프로그램에서 하나의 실행문이었던 것이 가상머신 위에서는 여러 단계로 쪼개지는 경우가 발생하는 것이지요. 그래서, 중간과정에 있는 Iconst(2)와 같은 가상머신 코드는 일치하는 코드를 찾을 수가 없습니다. 그리고, 증명을 위해 역변환 (decompilation) 함수를 모든 코드에 대해서 일반적으로 정의하는 일은 어렵습니다. 반면 forward simulation은 $\mathcal{B}(P_1) \subseteq \mathcal{B}(P_2)$ 은 증명하기가 훨씬 더 쉽습니다. 그래서, 전략을 이렇게 세웁니다. 우선 forward simulation을 증명한 후에 머신 코드가 결정적(deterministic)이라는 성질을 증명합니다. 프로그램이 결정적이면 관찰가능한 행동양식이 하나라는 뜻이므로 자동으로 backward simulation까지 증명이 된 셈입니다.

프로그램이 발산하거나 오류 상태로 간다는 것을 수식화 하고 싶습니다. 이때 큰 걸음 실행 의미로 정의하는 것은 그런 것들을 구별할 수가 없습니다. 반면 실행 단계 하나하나를 표현하는 작은 걸음 의미로는 표현이 가능합니다. 하지만 단점은 있습니다. 강력한 귀납법을 쓰기가 어렵고, 해석기는 이런 식으로 코딩이 되지 않습니다.
어차피 해석기 자체가 Coq에서는 논리에서의 서술식이나 서술 함수이기 때문에 스타일이 맞지 않는다는 것이 큰 문제가 될 수 있습니다. 혹시 새로운 구문이 추가되는 경우 많은 부분을 변경해야 하는 문제도 있습니다. Leroy는 큰 걸음 실행의미 스타일의 장점을 살리면서 발산과 오류 상태를 구분할 수 있는 표현 방식을 고안했습니다.

$$\frac{c_1/s \Rightarrow \infty}{c_1; c_2/s \Rightarrow \infty} \qquad \frac{c_1/s \Rightarrow s_1 \quad c_2/s_1 \Rightarrow \infty}{c_1; c_2/s \Rightarrow \infty}$$

$$\frac{c_1/s \Rightarrow \infty \text{ if } \texttt{beval } s\ b = \texttt{true} \quad c_2/s \Rightarrow \infty \text{ if } \texttt{beval } s\ b = \texttt{false}}{\texttt{IFB } b \texttt{ THEN } c_1 \texttt{ ELSE } c_2 \texttt{ FI}/s \Rightarrow \infty} \qquad \frac{\texttt{beval } s\ b = \texttt{true} \quad c/s \Rightarrow \infty}{\texttt{WHILE } b \texttt{ DO } c \texttt{ END}/s \Rightarrow \infty}$$

$$\frac{\texttt{beval } s\ b = \texttt{true} \quad c/s \Rightarrow s_1 \quad \texttt{WHILE } b \texttt{ DO } c \texttt{ END}/s_1 \Rightarrow \infty}{\texttt{WHILE } b \texttt{ DO } c \texttt{ END}/s \Rightarrow \infty}$$

그런데 문제는 $c/s \Rightarrow \infty$ 을 만족시키는 기본 공리 (axiom)가 없다는 겁니다. 항상 만족시키지도 못한 조건이 있는 규칙은 의미가 없지요.

여기서 정말 중요한 개념을 명료한 설명을 통해 배웠습니다. 예전에 덕환이형이 석사 논문의 중요한 정리를 증명하기 위해서 찾아 보길래 같이 살펴 보았으나 어려워서 이해하지 못했던 co-induction입니다. co-induction은 그냥 귀납법과는 다르게 무한한 증명 나무 (infinite derivation tree)에 대해서도 결론을 도출하는 것이 가능합니다.

$$\dfrac{\texttt{beval } s\ \texttt{true} = \texttt{true} \quad \texttt{SKIP}/s \Rightarrow s \qquad \dfrac{\texttt{beval } s\ \texttt{true} = \texttt{true} \quad \texttt{SKIP}/s \Rightarrow s}{c/s \Rightarrow \infty} \quad \dfrac{\texttt{beval } s\ \texttt{true} = \texttt{true} \quad \texttt{SKIP}/s \Rightarrow s}{c/s \Rightarrow \infty}}{c/s \Rightarrow \infty}$$

예로 위와 같이 조건문이 항상 참이고 몸체는 skip문 하나로 되어 있는 while문이 무한히 실행된다는 사실을 증명하기 위해서는 무한한 증명 나무를 생성해야 합니다. co-induction을 통해 위의 증명이 맞다는 사실을 보이고, 큰 걸음 의미와 작은 걸음 의미에서 정의한 발산의 의미가 서로 치환가능하다는 사실을 보입니다. 즉, c/s가 발산하면 작은 걸음 의미에서 무한한 길이의 명령문들이 존재하고 그 역도 성립한다는 사실을 증명합니다.

For all c and s,
either c/s reduces infinitely,
or there exists c', s' such that $c/s \xrightarrow{*} c'/s' \not\rightarrow$

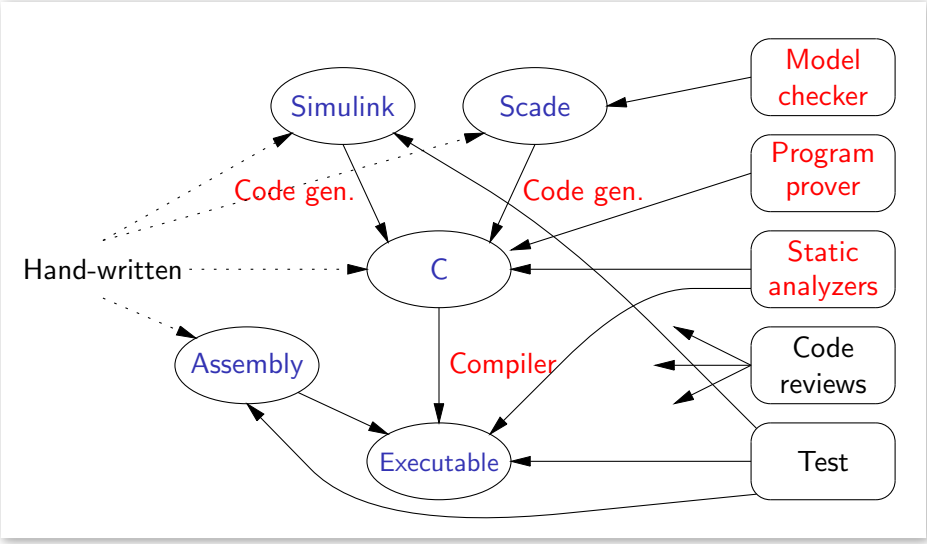
$(\forall P,\ P \vee \neg P)$

위와 같이 or로 묶인 결론을 내기 위해서는 앞에서도 말했듯이 Coq에서는 둘 중에 하나를 증명할 수 있어야 하는데 그 것이 불가능하기 때문에 (앞에 것을 증명하는 것은 종료 문제 (halting problem)를 푼다는 것이고 그건 Leroy 표현대로 좋은 일이 아니므로) 위의 공리를 추가해서 증명합니다.

이제 컴파일러의 최적화의 안전성을 증명합니다. 근데 최적화를 잘 하기 위해서는 프로그램을 분석할 필요가 있습니다. 따라서, 최적화의 안전성을 증명하는 일은 정적 분석의 안전성을 증명하는 것과 불가분의 관계에 있습니다. Leroy는 강의에서 생존 분석 (liveness analysis), 죽은 프

로그램 제거 (dead code elimination), 레지스터 할당 (register allocation)의 안전성을 증명하였습니다. 생존 분석은 컴파일러 책에 있는 전형적인 분석 방법을 따르고 있습니다. 그런데, 유한한 집합들에 포함 관계는 잘 정의된 순서 관계 (well-founded order)가 아니므로 유한번만 계산하고 잘 모르겠으면 안전한 큰 근사치 (over-approximation)을 쓴다고 합니다. 변수의 개수가 실제로는 유한하지만 유한하다는 정의를 하기 어려워서 그런 것 같습니다. 일종의 버티다가 축지법 (delayed widening)을 적용하는 것입니다. 분석 결과를 이용해서 안전한 죽은 프로그램 제거와 레지스터 할당을 Coq으로 증명했습니다.

다음은 일반적인 요약 해석에 기반한 정적 분석기를 증명하는 연구에 대해서 설명을 했습니다. 요약 해석의 기본에 대해서 설명을 해주었습니다. 이광근 교수님의 강의나 논문을 통해서 익히 알고 있었기에 복습하는 기분으로 쉽게 따라 갈 수 있었습니다. 요약 해석기를 Coq을 통해 구현하는 과정은 Ocaml로 분석기나 해석기 (interpreter)를 구현하는 과정과 많이 닮아 있습니다. 그리고, 가장 간단한 분석기의 구현에서 시작해서 이것 저것 분석 속도와 정확도를 높이는 기술들을 더 구현했습니다. 정확도를 높이는 기술이라고 해야 분기문에 있는 조건식이 정확히 참이거나 거짓일때 실행되지 않는 실행문은 제거하는 수준의 간단한 것들입니다. 분석 속도 높이는 것은 축지법 (widening)이구요. 그래도, 축지법과 좁히기 (narrowing)을 그래프로 보여준 것은 신선했습니다. 아직 연구중이라서 그런지 요약 해석기의 단조성 (monotonicity)이나 축지법, 좁히기의 성질들이 아직 증명이 안되어서 소개하는 정도에서 그쳤습니다.



강의에서 소개하는 내용은 극히 일부일 뿐이고 실제 중요한 임베디드 소프트웨어를 검증하기 위해서 위와 같이 여러 방법을 총동원한다고 합니다. 컴파일러 최적화도 여러가지를 다 쓰고 있고요. 더 대단한 것은 실험 결과에서 이 컴파일러로 생성한 코드가 gcc에서 최적화를 끈 코드보다는 모두 빠르고, 대부분의 프로그램에 대해서 최적화 레벨을 O1이나 O3의 gcc와 비슷한 성능을 냈다는 것입니다. 앞으로 언어확장, 더 많은 검증 기술 도입, 병렬 프로그램에 적용하기등 다양한 연구를 진행할 계획이라고 합니다.

의미 있는 연구 주제에 멋지게 정진하는 모습이 감동적이었습니다.

다른 강의들

Anupam Datta의 강의 Programming Language Methods for Compositional Security는 한참 Coq을 재밌게 배우고 있을때 갑자기 끼어 든 형태라 주제와 동 떨어진듯한 강의였습니다. 각각의 콤포넌트들이 안전하다는 사실이 증명되었을때 그 합의 안전성은 어떻게 증명할 것인가에 대해 강의를 했습니다. 보안성 규약 (security protocol)을 표현하기 위한 프로그래밍 언어를 정의하고 이를 Hoare 논리로 증명하는 아이디어를 보여주었습니다.

Greg Morisett은 Ynot 프로그래밍에 대해서 강의를 하였습니다. Hoare 타입 이론에 대한 강의를 하였으며 마지막에 분리 논리 (seperation logic)에 대하여도 언급을 하였습니다. Ocaml의 예외 사항 처리가 인터페이스 파일에 나와 있지 않아서 불편하다는 이야기를 했습니다. 절대적으로 공감이 가는 이야기였습니다. Ocaml 프로그래밍을 하다보면 예외 사항 처리가 잘 되지 않아 문제를 많이 겪습니다. Java에서는 인터페이스에 어떤 예외가 발생할 수 있는지 모두 나와 있다고 하는데 문제는 너무 부정확한 경우가 많아서 경고가 너무 많다고 합니다. Greg은 의존 타입 (dependent type) 시스템과 비슷한 방식으로 예외 사항이 발생하는 조건들을 기록하여 그런 문제를 해결하려고 노력한다고 했습니다. 강의를 하다 보면 사람들이 Greg의 Coq 코딩 스타일에 대하여 질문을 하는 것을 느꼈고, 특히 Constable이 손으로 직접 증명하는 것과 스타일이 너무 다르지 않느냐는 질문에 Greg은 Xavier Coq 프로그램도 다른 사람들이 보면 이해하기 어렵다고 답했습니다. 그러면서 자신은 프로그래밍의 관점에서 자신의 프로그램의 재사용성을 높이는 것에 보다 신경을 쓴다고 했습니다.

맺으며...

월드컵 기간과 겹치는 바람에 개인적으로는 힘든 여름학교였습니다. 우리나라 16강전은 비행기 시간과 겹쳐서 놓쳤는데 아직까지 시청하지 못했습니다. 보다 많이 예습과 복습을 하였다면 더 많이 얻어왔을텐데라는 아쉬움이 남습니다. 몇몇 강의(Constable, McBride)는 너무 따라가기 벅차고 교재도 없어서 힘들었습니다. 그래도, 우리 분야의 대가들과 안면을 트는(?) 기회도 있었고 MIT에 머물때 친하게 지내던 Sasa를 다시 보게 되어서 기뻤습니다. 여러 나라의 친구들과 우리만의 월드컵을 했던 것도 즐거운 추억으로 남아있습니다. 제가 Benjamin C. Pierce의 공을 두 번이나 뺐었습니다. 포항공대 현승이와 종현이, 카이스트 지응이와 준형이도 함께 해서 즐거웠습니다.