Program Verification via Type Theory

CS242
Lecture 17
Program Verification

• Proving properties of programs

• But not just that programs are well-typed
  • Much deeper, almost arbitrary properties
  • And often verifying full functional correctness

• Components
  • A specification: What property the program is supposed to have
  • A proof: Written mostly manually
  • A proof assistant: Supports defining the concepts, managing the proof, checking the proof, some automation of easy parts of the proof

• Proof assistants are based on type theory
Type Theory

- Pioneered by Bertrand Russell in the early 20th century
  - And greatly extended in computer science

- Original goal: A basis for all mathematics
  - An alternative to set theory

- Allows the formalization of
  - Programs
  - Propositions (types)
  - Proofs that programs satisfy the propositions
  - Uniformly in one system
Caveats

• There are multiple versions of type theory

• We will look at one, and mostly by example
  • At the level we consider, there aren’t significant differences with other approaches

• Type theory is a big topic
  • Whole courses are devoted to it
  • (But the same is true of other topics in this class!)
Lambda Application and Abstraction Rules

### App

\[
A \vdash e_1 : t \rightarrow t' \\
A \vdash e_2 : t \\
\hline
A \vdash e_1 e_2 : t'
\]

If \( e_1 : t \rightarrow t' \) and \( e_2 : t \), then \( e_1 e_2 \) has type \( t' \).

### Abs

\[
A, x : t \vdash e : t' \\
\hline
A \vdash \lambda x.e : t \rightarrow t'
\]

If assuming \( x : t \) implies \( e : t' \), then \( \lambda x.e : t \rightarrow t' \).

Function Type Elimination

Function Type Introduction
Ignore the Programs for a Moment ...

\[ A \vdash e_1 : t \to t' \]
\[ A \vdash e_2 : t \]
\[ \frac{}{A \vdash e_1 e_2 : t'} \quad \text{[App]} \]

From a proof of \( t \to t' \) and and a proof of \( t \), we can prove \( t' \).

Implication Elimination (modus ponens)

\[ A, x : t \vdash e : t' \]
\[ \frac{}{A \vdash \lambda x.e : t \to t'} \quad \text{[Abs]} \]

If assuming \( t \) we can prove \( t' \), then we can prove \( t \to t' \).

Implication Introduction
Types As Propositions

\[
\frac{A \vdash e_1 : t \rightarrow t'}{
\frac{A \vdash e_2 : t}{A \vdash e_1 \, e_2 : t'}}
\]

\[
\frac{A, x : t \vdash e : t'}{A \vdash \lambda x.e : t \rightarrow t'}
\]

From a proof of \(t \rightarrow t'\) and a proof of \(t\), we can prove \(t'\).

If assuming \(t\) we can prove \(t'\), then we can prove \(t \rightarrow t'\).

Here we regard the types as propositions: If we can prove certain propositions are true, then we can prove that other propositions are true.

But what are the proofs?

Alex Aiken   CS 242   Lecture 17
Programs as Proofs

\[
\begin{align*}
A \vdash e_1 : t &\rightarrow t' \\
A \vdash e_2 : t &\quad \text{[App]} \\
A \vdash e_1 e_2 : t' \\
\end{align*}
\]

From a proof of \( t \rightarrow t' \) and a proof of \( t \), we can prove \( t' \).

\[
\begin{align*}
A, x : t \vdash e : t' \\
A \vdash \lambda x.e : t \rightarrow t' &\quad \text{[Abs]} \\
\end{align*}
\]

If assuming \( t \) we can prove \( t' \), then we can prove \( t \rightarrow t' \).

Answer: The programs! \( e : t \) is a proof that there is a program of type \( t \).
The Curry-Howard Isomorphism

• There is an isomorphism between programs/types and proofs/propositions.

• Two interpretations of \( \vdash e : t \)

• We have a proof that the program \( e \) has type \( t \)
  • \( \to \) is a constructor for function types

• \( e \) is a proof of \( t \)
  • \( \to \) is logical implication
Discussion

• This seems interesting ... but is it useful?

• Not so far

• If we use more expressive types, we can express more propositions.

• We need more than implication!
Propositional Logic

• As an example, we show how to define the rest of propositional logic

• This is just one of many theories we could define
  • But a particularly useful one

• We will define:
  • And
  • Or
  • Not
And

$$\frac{\text{A} \vdash e_1 : t_1 \quad \text{A} \vdash e_2 : t_2}{\text{A} \vdash ? : t_1 \land t_2}$$  [And-Intro]

$$\frac{\text{A} \vdash ? : t_1 \quad \text{A} \vdash e : t_1 \land t_2}{\text{A} \vdash e : t_1 \land t_2}$$  [And-Elim-Left]

$$\frac{\text{A} \vdash e : t_1 \land t_2 \quad \text{A} \vdash ? : t_2}{\text{A} \vdash ? : t_2}$$  [And-Elim-Right]

What program is a proof of $t_1 \land t_2$?
Pairs

[And-Intro]

\[
\begin{align*}
A & \vdash e_1 : t_1 \\
A & \vdash e_2 : t_2 \\
A & \vdash (e_1, e_2) : t_1 \land t_2
\end{align*}
\]

[And-Elim-Left]

\[
\begin{align*}
A & \vdash e : t_1 \land t_2 \\
A & \vdash e.left : t_1
\end{align*}
\]

[And-Elim-Right]

\[
\begin{align*}
A & \vdash e : t_1 \land t_2 \\
A & \vdash e.right : t_2
\end{align*}
\]
Or

\[ A \vdash e : t_1 \]
\[ \underline{A \vdash e : t_1 \lor t_2} \]  [Or-Intro-Left]

\[ A \vdash e : t_2 \]
\[ \underline{A \vdash e : t_1 \lor t_2} \]  [Or-Intro-Right]

\[ A \vdash e : t_1 \lor t_2 \]
\[ \underline{A \vdash ? : ?} \]  [Or-Elim]
Hmmm ... 

• The Or-Elim rule isn’t obvious

• We need to exhibit a program that works regardless of whether \( e \) is an element of \( t_1 \) or \( t_2 \).

• Solution
  • The elimination is done by another program that does a case analysis
Or Elimination

\[ A \vdash e_0 : t_1 \lor t_2 \quad A, x : t_1 \vdash e_1 : t_0 \quad A, x : t_2 \vdash e_2 : t_0 \]

\[ A \vdash (\lambda x. \text{case } x \text{ of } t_1 \to e_1; t_2 \to e_2) \ e_0 : t_0 \]
Discussion

• Using a case analysis makes sense to computer scientists
  • Do one thing if the list is Nil / \( n = 0 \)
  • Do something else if the list has at least one element/ \( n > 0 \)

• But this is not the “or” of classical logic
  • In constructive logic, we must construct evidence for everything we prove
  • To use a disjunction, we must know which case we are in

• A dual explanation
  • To create a disjunction, we must compute a value of one of the types

• Thus \( \mathfrak{t} \lor \neg \mathfrak{t} \) is not an axiom of this system!
  • And this is the only classical axiom that must be excluded
Negation

• \( \neg p \) is defined as \( p \rightarrow \text{false} \)
  • Proposition \( p \) implies a contradiction

• \text{False} is the empty type – there is no evidence for \text{false}

• Thus \( \neg p \) either does not have any elements, or only non-terminating functions
  • Depending on what else is included in the theory we are using
What is Negation Good For?

• There can be uses for negation

• If we are just interested in proving things, proof by contradiction is an important technique
  • Recall one goal is to formalize mathematics

• But there are also computational interpretations
Type Theory for Continuations (Sketch)

Recall $\neg p = p \rightarrow false$

In pure lambda calculus, a function of type $\neg p$ can’t be called
  • Because false has no elements in its type

• But in a language with continuations:
  • Recall that a continuation has the form $\lambda v. e$ and does not return when called
  • So it is sensible to give continuations a type $p \rightarrow false = \neg p$
Constructive vs. Classical Logic

• Constructive logic gives us programs we can run

• Type theory can also have classical axioms
  • What axioms are used is not the distinguishing feature of type theory
  • But if we use classical logic, we also lose the ability to use the proofs as programs, as they are no longer constructive

• In applications to software, we are generally interested in constructive proofs
Summary

- We have shown how to define propositional logic in type theory
  - Give sensible type rules for and, or and not
  - Show how to construct programs that have the postulated types

- Example: We can prove \((a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow (a \rightarrow b \land c)\)
Taking It to the Next Level

- We want to be able to define new kinds of theories within the system
- and, or, & not should be definable within the system
- The type checking rules should also be definable
Boolean Connectives Revisited

- What are **and**, **or** and **not**?

- They are functions that take types and construct new types

- Introduce a new type **Type** that contains all types
  - $\text{Type} = \{ \text{Int, Bool, Int} \rightarrow \text{Int, ...} \}$

- and: Type $\rightarrow$ Type $\rightarrow$ Type
- or: Type $\rightarrow$ Type $\rightarrow$ Type
- not: Type $\rightarrow$ Type
Inference Rules Revisited

• An inference rule is a function that takes proofs of propositions as arguments and produces a proof of a proposition as a result

• Define a new type \textit{Proof}

• And-Intro: \textit{Proof} \rightarrow \textit{Proof} \rightarrow \textit{Proof}
• And-Elim-Left: \textit{Proof} \rightarrow \textit{Proof}
• And-Elim-Right: \textit{Proof} \rightarrow \textit{Proof}
Review

So now we can:

• Define new types
• Define new type combinators (and, or, not ...)
• Define new inference rules (and-intro, ...)

• All using a uniform system based on types
• Note the system also checks type functions and inference rules are correctly used
  • E.g., we can only build valid proofs
Are We Done?

• Not yet

• There are three more important features of type theories:
  • Type stratification
  • Inductively defined data types
  • Pi types
Type Stratification

• Recall we "Introduce a new type Type that contains all types"
  • Type = { Int, Bool, Int → Int, ... }

• So is Type ∈ Type ?
And Now ... A Little Set Theory

• Recall in the early 20th century there was a systematic effort to understand the foundations of logic
  • As part of the goal of formalizing mathematics

• *Set theory* was recognized as a potential foundation
Why Set Theory?

• A function $f$ can be represented as a set of (input, output) pairs:

$$\{(x_i, y_i) \mid f(x_i) = y_i\}$$

• Natural numbers:

  $0 \equiv \emptyset$

  $\text{Succ}(n) \equiv n \cup \{n\}$

• And so on ...
Russell’s Paradox

Consider \( R = \{ x \mid x \notin x \} \)

Now we can easily show:

\[
\begin{align*}
R \notin R & \implies R \in R \\
R \in R & \implies R \notin R
\end{align*}
\]

So we conclude:

\[ R \in R \iff R \notin R \]
Implications

• Russell’s paradox showed naïve set theory is inconsistent
  • Can prove ``false is true’’ and so can prove anything
  • Not a great foundation for mathematics!

• Led to a reconsideration of the foundations of set theory
  • Over a couple of decades

• One conclusion: No set could be an element of itself
  • Set theory should be well-founded
What Does Well-Founded Mean?

• There is no set of all sets
• Instead, there is an infinite hierarchy of stratified sets

• We define ``small’’ sets at stratum 0
• The set of all level 0 sets is a stratum 1 set
• The set of all level 1 sets is a stratum 2 set
• ...

• In this way no set can be an element of itself
  • Stratum $n$ sets can only contain small sets of stratum $n$ and sets of strata less than $n$
• Similar to the definition of ordinals
Back To Types ...

• Recall that types are sets
  • So Russell’s paradox applies to types as well

• Implies we will need a type hierarchy
  • In a consistent type system
  • The set of all types lives at a higher level in the hierarchy than ordinary types
Ordinary Types

0 : Int
succ : Int → Int
add: Int → Int → Int

true: Bool
false: Bool
and: Bool → Bool → Bool
Next Level ...

• What are Int, Bool, $\alpha \rightarrow \beta$, ...?

• They are types
  • Int : Type
  • Bool: Type
  • Int $\rightarrow$ Int: Type

• Int, Bool, etc. are at level 0 of the type hierarchy
• Type is at level 1
Next Level ...

• What are $\rightarrow$ and $\mathsf{and}$?

• They are functions of types that produce types
  • $\rightarrow : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$
  • $\mathsf{and} : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$

• These are functions that operate on elements of type level 1
Inductively Defined Data Types

• Dependent type theories generally include inductively defined data types as a primitive concept
  • So users can define natural numbers, lists, trees, etc.
  • With constructors of the appropriate types

• We have already talked about how to represent inductively defined data types as lambda terms in previous lectures.
  • Nothing new here ...
Pi Types

• What we have discussed so far is still missing an important feature

• We can’t express type functions that depend on their arguments

• Example \texttt{cons}: \(\alpha \rightarrow \text{List}(\alpha) \rightarrow \text{List}(\alpha)\)
  
  • What is the type of \texttt{cons}?
  
  • Explanation 1: \texttt{cons} has a family of types indexed by a parameter \(\alpha\)
  
  • Explanation 2: \texttt{cons} has many types, one for each \(\alpha\)
    • a product or intersection of an infinite set of types
Pi Types

Defining the List data type:

List: Type → Type
Cons: Π α : Type. α → List(α) → List(α)
Nil: Π α : Type. List(α)

Polymorphic types are an example of dependent types: The type depends on a parameter. Note how Π functions like ∀.

There is also a corresponding sum type Σ that functions like ∃
Pi Types

The parameter in a Pi type doesn’t have to range over Type.

A polymorphic array that includes its length in the type:

Array: Type → Int → Type

mkarray: Π α : Type. Π β : Int. α → β → Array(α, β)

Here β is an integer – which could be any expression of type Int!
Discussion

• Without Pi types, type theory is very limited
  • E.g., simply typed lambda calculus

• Pi types are extremely powerful
  • The construct for creating infinite families of types
  • The signature feature of dependent type theories
  • Play a somewhat similar role to set comprehension in set theory

• Dependent type systems are often undecidable
  • Performing computation as part of type checking is bound to quickly run into computability issues!
Type Theory

• A foundation for all mathematics
  • Especially constructive mathematics
  • Sufficiently powerful to prove anything we can think of proving
  • And thus also a foundation for verifying the correctness of software

• Key features
  • Isomorphism of programs/types with proofs/propositions
  • Type hierarchy allows uniform definition of types, type operations, proofs, ...
  • Dependent types allow very expressive (even to the point of undecidability) types to be constructed
Type Theory in the Real World

• Type theory has been used to verify the correctness of real systems

• CompCert
  • A formally verified (subset of) C compiler

• Sel4
  • A formally verified OS microkernel
  • Has many but not all features of a real OS
State of Practice

• Compcert and Sel4 show that formal verification of significant systems using type theory-based proof assistants is possible.

• Compcert and Sel4 have very high levels of assurance
  • Debugging is not an issue
  • Guaranteed, for example, to be extremely secure

• But Compcert and Sel4 have shown the software engineering costs of full formal verification are still high
  • Sel4 has over 1M lines of proofs
  • Modifications may require much more reproving than recoding

• The biggest barrier for most systems, though, is having the specification
  • To use a theorem prover, you first have to state a theorem to prove!