Program Verification via Type Theory

CS242
Lecture 12
Projects

There will be three projects:

• Program verification using dependent types
  • Today’s lecture

• Gradual typing: Integrating static and dynamic type checking
  • Wednesday’s lecture

• Writing an async library using Rust’s novel features
  • Already covered ...
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 the 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

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

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

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

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

Function Type Elimination

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

\[ \begin{align*}
A \vdash e_1 : t \to t' \\
A \vdash e_2 : t \\
\hline
A \vdash e_1 e_2 : t' \\
\end{align*} \]

[App]

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

Implication Elimination (modus ponens)

\[ \begin{align*}
A, x : t & \vdash e : t' \\
A & \vdash \lambda x.e : t \to t' \\
\hline
A & \vdash \lambda x.e : t \to t' \\
\end{align*} \]

[Abs]

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

Implication Introduction
Types As Propositions

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

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

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 12
Programs as Proofs

\[ A \vdash e_1 : t \to t' \]
\[ A \vdash e_2 : t \]
\[ A \vdash e_1 e_2 : t' \]

[App]

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

[Abs]

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

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

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

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

• Two interpretations of $\vdash e : t$

• We have a proof that the program $e$ has type $t$
  • $\rightarrow$ is a constructor for function types

• $e$ is a proof of $t$
  • $\rightarrow$ 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

A ⊢ e₁ : t₁
A ⊢ e₂ : t₂
A ⊢ ? : t₁ ∧ t₂

[And-Intro]

A ⊢ e : t₁ ∧ t₂
A ⊢ ? : t₁

[And-Elim-Left]

A ⊢ e : t₁ ∧ t₂
A ⊢ ? : t₂

[And-Elim-Right]

What program is a proof of \( t₁ ∧ t₂ \)?
Pairs

\[
\begin{aligned}
\text{A} \vdash e_1 : t_1 \\
\text{A} \vdash e_2 : t_2 \\
\hline
\text{A} \vdash (e_1, e_2) : t_1 \land t_2
\end{aligned}
\]  
[And-Intro]

\[
\begin{aligned}
\text{A} \vdash e : t_1 \land t_2 \\
\hline
\text{A} \vdash e.left : t_1
\end{aligned}
\]  
[And-Elim-Left]

\[
\begin{aligned}
\text{A} \vdash e : t_1 \land t_2 \\
\hline
\text{A} \vdash e.right : t_2
\end{aligned}
\]  
[And-Elim-Right]
Or

\[
\begin{align*}
A \vdash e : t_1 \\
\hline
A \vdash e : t_1 \lor t_2
\end{align*}
\]  [Or-Intro-Left]

\[
\begin{align*}
A \vdash e : t_2 \\
\hline
A \vdash e : t_1 \lor t_2
\end{align*}
\]  [Or-Intro-Right]

\[
\begin{align*}
A \vdash e : t_1 \lor t_2 \\
\hline
A \vdash ? : ?
\end{align*}
\]  [Or-Elim]
Hmmmm ...

• The Or elimination 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

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

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

• This is not the “or” of classical logic
  • In constructive logic, we must construct evidence for everything we prove

• More restricted
  • 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 $t \lor \neg t$ 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 are 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 $\neg p = p \rightarrow \text{false}$

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 $p \rightarrow \text{false} = \neg p$
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 definable within the system

• The type checking rules should also be definable
Boolean Connectives Revisited

• What are \textit{and}, \textit{or} and \textit{not}?

• They are functions that take types and construct new types

• Introduce a new type \texttt{Type} that contains all types
  • \texttt{Type} = \{ Int, Bool, Int \to Int, ... \}

• \texttt{and}: Type \to Type \to Type
• \texttt{or}: Type \to Type \to Type
• \texttt{not}: Type \to 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 \textbf{Proof}

  • And-Intro: \textbf{Proof} $\rightarrow$ \textbf{Proof} $\rightarrow$ \textbf{Proof}
  • And-Elim-Left: \textbf{Proof} $\rightarrow$ \textbf{Proof}
  • And-Elim-Right: \textbf{Proof} $\rightarrow$ \textbf{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 \rightarrow Int, \ldots \}

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

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

• \textit{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:

$$\begin{align*}
0 &\equiv \emptyset \\
\text{Succ}(n) &\equiv n \cup \{n\}
\end{align*}$$

• And so on ...
Russell’s Paradox

Consider $R = \{ x \mid x \notin x \}$

Now we can easily show:

- $R \notin R \Rightarrow R \in R$
- $R \in R \Rightarrow R \notin R$

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 \texttt{and}? 

• They are functions of types that produce types
  • \( \rightarrow : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \)
  • \texttt{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 cons: $\alpha \rightarrow \text{List}(\alpha) \rightarrow \text{List}(\alpha)$
  • What is the type of cons?
  • Explanation 1: cons has a family of types indexed by a parameter $\alpha$
  • Explanation 2: 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: \( \Pi \alpha : \text{Type}. \, \alpha \to \text{List}(\alpha) \to \text{List}(\alpha) \)
Nil: \( \Pi \alpha : \text{Type}. \, \text{List}(\alpha) \)

Polymorphic types are an example of dependent types: The type depends on a parameter. Note how \( \Pi \) functions like \( \forall \).
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:

\[ \text{Array: Type } \rightarrow \text{ Int } \rightarrow \text{ Type} \]

\[ \text{mkarray: } \Pi \alpha : \text{ Type}. \Pi \beta : \text{ Int}. \alpha \rightarrow \beta \rightarrow \text{ Array}(\alpha, \beta) \]

Here \( \beta \) is an integer – which could be any expression of type \( \text{Int}! \)
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 there for 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 using type theory-based proof assistants is becoming practical

• 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
  • Have to know what to prove to use a theorem prover!