

## Dependent Type Systems

Lecture 20  
CS263

CS 263

1

### Dependent Types

- Say that we have the functions  
zero : nat → vector (creates vector of requested length)  
dotprod: vector → vector → real (dot product)
- The types do not prevent using dotprod on vectors of different length
  - If they could, maybe we could catch more bugs through type checking!
- Idea: Make "vector" a type family annotated by a natural number  
"vector n" is the type of vectors of length n  
dotprod: vector n → vector n → real (where is n bound?)  
zero : nat → vector ? (need a way to refer to the value of the first argument in the type!)

CS 263

3

### Another Example

- Consider the familiar sprintf function (C, C++)  
sprintf : Πf:Format.Data(f)
- Here Data is a function from format strings to types
  - Data("") = string
  - Data("%s" ++ f) = string → Data(f)
  - Data("%d" ++ f) = int → Data(f)
  - Data("c" ++ f) = Data(f)
- For example:
  - sprintf "%d little %s" : int → string → string
- Ocaml "format" works like this
  - Catch: the format string must be a literal
  - In those cases "gcc -Wall" does the same checking

CS 263

5

### Review

- We studied a variety of type systems
- We repeatedly made the type system more expressive to enable the type checker to catch more errors
- But we have steered clear of undecidable systems
  - Thus there must still be many errors that are not caught by the type checker
- Now we explore more complex type systems that bring type checking closer to program verification

CS 263

2

### Dependent Types. Notation

- How to write the type of zero : nat → vector ?
- Given two sets A and B verify the isomorphism  
 $A \rightarrow B \simeq \prod_{x \in A} B$ 
  - The latter is the cartesian product of B with itself as many times as there are elements in A
  - Also written as  $\prod x:A.B$
  - The name x in this scenario plays no role.
  - But now we can make B depend on x !
- Definition:  $\prod x:A.B$  is the type of functions with argument in A and with the result type B (possibly depending on the value of the argument x)
  - We write "zero :  $\prod n:\text{nat}. \text{vector } n$ "
  - In the special case when  $x \in B$  we abbreviate as  $A \rightarrow B$

CS 263

4

### Dependent Types and Program Specifications

- Types act as specifications
- With dependent types we can specify any property !
- For example, define the following types:
  - "eq e" - the type of values equal to "e".  
Also named "sng e" (the singleton type)
  - "ge e" - the type of values larger or equal to "e"
  - "lt e" - the type of values smaller than "e"
  - "and  $\tau_1 \tau_2$ " - the type of values that have both type  $\tau_1$  and  $\tau_2$
- Need appropriate typing rules for the new types
- The precondition for vector-read
  - read:  $\prod n:\text{nat}. \text{vector } n \rightarrow (\text{and } (ge\ 0) (lt\ n)) \rightarrow \text{int}$
- The type checker must do program verification

CS 263

6

## Dependent Types. Logical Frameworks

- Dependent types play an important role in the formalization of logics
  - Started with Martin-Lof
- Need a language in which we can write proofs
  - E.g., for writing down formal arguments in various logics
  - E.g., for theorem provers that generate proofs
  - E.g., for proof-carrying code

CS 263

7

## Desired Characteristics

- General framework
  - Applicable to many logics
  - Allows high-level description of the logic
- Simple and fast proof checking
  - Parameterized by the logic (so we don't have to rewrite it over and over)
- Compact representations of proofs
  - Reduces bandwidth needed in Proof-Carrying Code
  - Reduces space required for storage of proofs
  - Speeds-up proof validation

## Ad-Hoc Proof Representation (1)

- Proofs are derivation trees
  - Each internal node is an instance of an inference rule (with subtrees being proofs of hypotheses)
- First impulse is to use an ad-hoc representation of proof trees
- Nodes are annotated with inference rule names
- Example:

The conjunction introduction inference rule

$$\frac{\vdash P \quad \vdash Q}{\vdash P \wedge Q} \text{ andi}$$

An instance is represented as a node labeled with "andi" and with two children proving P and Q



## Ad-Hoc Proof Representation (2)

- Example:
  - Implication introduction

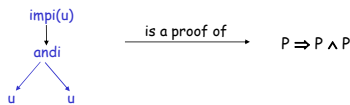
$$\frac{\begin{array}{c} P^u \\ \vdots \\ \vdash Q \end{array}}{\vdash P \Rightarrow Q} \text{ impi}^u$$

- An instance is a node labeled with "impi(u)" and one child proving Q (the local nature of the assumption is explicit)



## Example of Ad-Hoc Representation

- The representation of a proof of " $P \Rightarrow (P \wedge P)$ "



- How do we check such proofs?
  - We know what predicate is proved at the root
  - We read the inference rule used at each node
  - We find the predicates that must be proved by children
  - We descend recursively in the proof

## Ad-hoc Proof Validation

How do we validate such proofs?

- Checking: we know what formula is supposed to be proved
- Inference: same but we must infer what predicate is proved (an instance of bidirectional checking)

```

fun check(andi(D1,D2), and(P1,P2)) : bool =
  check(D1,P1) andalso check(D2,P2)
| check(andel(D), P) = let and(P1,P2) = infer D in P = P1 and
| check(eqid(E), eq(E1,E2)) = E1 = E andalso E2 = E
    
```

```

fun infer(andi(D1,D2)) : pred =
  let P1=infer(D1) and P2=infer(D2) in and(P1,P2) end
| infer(andel(D)) = let and(P1,P2) = infer(D) in P1 and
| infer(eqid(E)) = eq(E,E)
    
```

## Ad-hoc Representation Lessons

- Proofs are small
  - As small as they could be and still allow simple checking
- We need two validation modes
  - Checking when the proved predicate is known
  - Inference when the proved predicate is not known
- The implementation of validation is
  - Large (1000 lines for 80-axiom logic)
  - Error-prone implementation
  - Very much dependent on the logic
  - But relatively regular (same building blocks repeated)

## Predicates As Lambda Expressions

- Type constructors
  - "exp" for expressions
  - "pred" for predicates
- Term constructors for formulas
  - and :  $\text{pred} \rightarrow \text{pred} \rightarrow \text{pred}$
  - imp :  $\text{pred} \rightarrow \text{pred} \rightarrow \text{pred}$
  - all :  $(\text{exp} \rightarrow \text{pred}) \rightarrow \text{pred}$
- We can then represent predicates
  - $P \Rightarrow (P \wedge P)$  is represented as "imp P (and P P)"
  - $\forall x.P$  is represented as "all ( $\lambda x : \text{exp}. P$ )"
- Higher-order abstract syntax representation
  - Binding in the logic is binding in the representation
  - Substitution in logic is substitution in the representation
    - $P[E/x]$  is represented as " $(\lambda x : \text{exp}. P) E$ "

## Proofs as Lambda Expressions

- We change the declarations of proof constructors
  - and<sub>i</sub> :  $\Pi p_1 : \text{pred}. \Pi p_2 : \text{pred}. \text{pf } p_1 \otimes \text{pf } p_2 \otimes \text{pf } (\text{and } p_1 \ p_2)$
  - imp<sub>i</sub> :  $\Pi p_1 : \text{pred}. \Pi p_2 : \text{pred}. (\text{pf } p_1 \otimes \text{pf } p_2) \otimes \text{pf } (\text{imp } p_1 \ p_2)$
- Now the proof from before is
  - imp<sub>i</sub> P (and P P) ( $\lambda u : \text{pf } P. \text{and } P \ P \ u$ )
  - Can verify that it has type "pf (imp P (and P P))"
- This system is the Edinburgh Logical Framework (LF)
  - A logic is represented as a series of constant declarations
  - LF itself is completely independent of the logic
  - Not even  $\dot{U}$  or  $\dot{U}$  are built-in

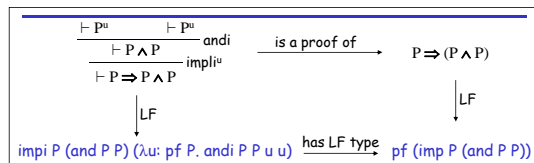
## Alternative Proof Representation

- The Curry-Howard isomorphism teaches us that proofs can be encoded as terms and formulas as types
  - formulas ~ types in simply typed lambda calculus
  - proofs ~ terms in simply typed lambda calculus
  - proof checking ~ type checking
- Extensions desired:
  - Handle more than intuitionistic logic
    - Martin-Lof: judgments as types, derivations as terms
  - An extensible language of types and terms
    - We want to add new term and type constructors

## Why Dependent Types?

- We now add proof constructors
  - and<sub>i</sub> :  $\text{pf} \rightarrow \text{pf} \rightarrow \text{pf}$
  - imp<sub>i</sub> :  $(\text{pf} \rightarrow \text{pf}) \rightarrow \text{pf}$
- The proof from before is represented as
  - imp<sub>i</sub> ( $\lambda u : \text{pf}. \text{and } u \ u$ )
  - We can verify that it has type "pf"
- Problem
  - Type checking ensures well-formedness (almost)
  - But does not tell which predicate is proved
- Need richer types
- Let "pf P" be the type of proofs of the predicate P
  - Types now encode the predicate that is proved
  - Meaning that type checking is more effective

## LF Representation of Proofs



- LF type checking  $\equiv$  Proof checking [Harper, et al.]
  - A theorem valid for any representable logic
- LF type checking is
  - very simple (15 typing rules) and ☺
  - general and logic independent ☺

## Formalization of Dependent Types

- More complex type structure:
  - int, int → int, "vector 5" are all types, but "vector true" is not
  - We must "type check" the types!
  - What is the "type" of "vector"?
- We introduce kinds (type families) to classify types
  - int : type
  - vector : int → type
- Syntax:
  - $e ::= x \mid \lambda x.e \mid e_1 e_2$  (terms, expressions)
  - $\tau ::= \text{int} \mid \Pi x:\tau_1.\tau_2 \mid \tau e$  (types)
  - $K ::= \text{type} \mid \tau \rightarrow K$  (kinds, type families)
- Remember:  $A \rightarrow B$  is the same as  $\Pi x:A.B$

CS 263

19

## Typing Rules

- Typing contexts  $\Gamma ::= \cdot \mid \Gamma, x:\tau$
- Well-formed contexts:  $\vdash \Gamma \quad \frac{}{\vdash \cdot} \quad \frac{\Gamma \vdash \tau : \text{type}}{\vdash \Gamma, x:\tau}$
- Well-formed types  $\Gamma \vdash \tau : K$

$$\frac{\Gamma \vdash \tau_1 : \text{type} \quad \Gamma \vdash \tau_2 : \text{type}}{\Gamma \vdash \Pi x:\tau_1.\tau_2 : \text{type}}$$

$$\frac{\Gamma \vdash \tau : \tau_1 \rightarrow K \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash \tau e : K}$$

CS 263

20

## Typing Rules (II)

- Well-formed expression  $s: \Gamma \vdash e : \tau$ 

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash \tau_1 : \text{type} \quad \Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x:\tau_1.e : \Pi x:\tau_1.\tau_2}$$

$$\frac{\Gamma \vdash e_1 : \Pi x:\tau_1.\tau \quad \Gamma \vdash \tau_1 \equiv \tau_2 : \text{type} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau[e_2/x]}$$
- Now type equivalence plays a big role
  - vector 5 ≡ vector (2 + 3)
  - Equivalence is typed to simplify the theory (no need to consider untyped normalization)
$$\frac{\Gamma \vdash \tau \equiv \tau' :: \tau_e \rightarrow K \quad \Gamma \vdash e \equiv e' : \tau_e}{\Gamma \vdash \tau e \equiv \tau' e' :: K}$$

CS 263

21

## Dependent Type Checking

- We call this language  $\lambda\text{LF}$
- The key difficulty is that the type checker needs to reason about expression equivalence
  - This makes type checking as powerful as program verification
  - Of course, type checking may be undecidable
- The key design decision is what limits to put on equivalence
  - Limit the expressions that can appear in types to a language with decidable equality (e.g., variables + linear arithmetic)
  - Or, not limit to get the extra power
  - Or, check equivalence at run-time (hybrid type checking)

CS 263

22

## Dependent Types. Normalization

- The operational semantics is the same as for simply typed lambda calculus
- This language is normalizing!
  - We can compile  $\lambda\text{LF}$  to simply-typed expressions
  - Compile types
    - For each type family constant, create a new type
      - $c(\text{int}) = \text{int}$ ,  $c(\text{vector}) = \text{vector}$
      - $c(\Pi x:\tau_1.\tau_2) = c(\tau_1) \rightarrow c(\tau_2)$ ,  $c(\tau e) = c(\tau)$
    - E.g.,  $c(\text{vector } 5) = \text{vector}$ ,  $c(\text{vector } \dots) = \text{vector}$
  - Compile expressions by compiling all types within them

CS 263

23

## Normalization

- Show that if  $\Gamma \vdash e : \tau$  (dependent) then  $c(\Gamma) \vdash c(e) : c(\tau)$  (simply-typed)
  - Lemmas, If  $\Gamma \vdash \tau \equiv \tau' :: K$  then  $c(\tau) = c(\tau')$
- Furthermore, the evaluation of  $e$  is isomorphic with that of  $c(e)$
- If follows from normalization of simply-typed lambda calculus that  $\lambda\text{LF}$  is normalizing
  - This means that  $\Gamma \vdash \tau \equiv \tau' :: K$  is decidable
  - This means that type checking is decidable
  - Type inference is a different story ...
  - Adding recursion to the language also changes the story ...

CS 263

24

## Dependent Sum Types

---

- Say now that we want to pack a vector with its length
  - $e = (n, v)$  where " $v : \text{vector } n$ "
  - The type of an element of a pair depends on the value of another element
  - This is another form of dependency
  - The type of  $e$  is " $\text{nat} \times \text{vector } ?$ "
- Given two sets  $A$  and  $B$  verify the isomorphism
$$A \times B \simeq \sum_{x \in A} B$$
  - The latter is the disjoint union of  $B$  with itself as many times as there are elements in  $A$
  - Also written as  $\sum x:A.B$
  - $x$  here plays no role.
  - But now we can make  $B$  depend on  $x$ !

CS 263

25

## Dependent Sum Types

---

- Definition:  $\sum x:A.B$  is the type of pairs with first element of type  $A$  and the second element of type  $B$  (possibly depending on the value of first element  $x$ )
  - Now we can write  $e : \sum x:\text{nat. vector } x$
- Let's now write a function that computes the length of a vector
  - $\text{length} : \prod n:\text{nat. vector } n \rightarrow \text{nat}$  (the result is not constrained)
  - $\text{sng } n : \prod n:\text{nat. vector } n \rightarrow \text{sng } n$
  - " $\text{sng } n$ " is a dependent type that contains only  $n$
  - called the singleton type
- What if the vector is packed with its length?
$$\text{length} : (\sum n:\text{nat. vector } n) \rightarrow \text{nat}$$

CS 263

26

## Dependent Sum Types. Static Semantics

---

- Typing rules:
$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : [e_1/x]\tau_2}{\Gamma \vdash (e_1, e_2) : \sum x : \tau_1. \tau_2}$$
$$\frac{\Gamma \vdash e : \sum x : \tau_1. \tau_2}{\Gamma \vdash \text{snd } e : [\text{fst } e/x]\tau_2}$$
  - Note how this rule reduces to the usual rules for tuples when there is no dependency
- The evaluation rules are unchanged

CS 263

27