Functional Programming

WS 2014/15

Cezary Kaliszyk (VO+PS)
Yann Savoye (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 13

Overview

- Week 13 - Dependent Types
  - Summary of Week 12
  - Simple Type Theory
  - Curry-Howard
  - Dependent Types

Overview

- Week 13 - Dependent Types
  - Summary of Week 12
  - Simple Type Theory
  - Curry-Howard
  - Dependent Types

Lazyness

Support for Lazyness

- keyword **`lazy`** (`'a -> 'a Lazy.t`)
- function `Lazy.force : 'a Lazy.t -> 'a`

Lazy Lists

```
type 'a cell = Nil
             | Cons of ('a * 'a llist)
and 'a llist = 'a cell Lazy.t
```

Overview

- Week 13 - Dependent Types
  - Summary of Week 12
  - Simple Type Theory
  - Curry-Howard
  - Dependent Types

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction,
reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing,

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, dependent types monads

Safety of programs

- Type checking
    - If a program compiles, some guarantee about safety
- Type inference
    - Further helps in the interaction
- What kind of errors can it detect?
- Is it always an advantage?
    - Constructions that are not allowed?
    - Complicated error messages?

Errors in programs

- A boolean is added to a list of integers
- First element of an empty list
- `zip` of lists with different lengths
- Division by zero
- $0! = 0$
- `fib 100` returns a negative integer?
- Program loops

Hierarchy of types

- No types (Asm)
- Artificially added types (C)
- Simple types (λ)
- Overloading, type classes.
- Polymorphism (OCaml+)
- Types that depend on expressions

Convenience of types

- More errors found $\rightarrow$ less programs accepted?
- Less programs accepted $\rightarrow$ less expressive?

Convenience of types

- More errors found $\rightarrow$ less programs accepted? NO!
- Less programs accepted $\rightarrow$ less expressive? NO!
- There are stronger type systems
  - A bit more complicated than λ-calculus

Convenience of types

- More errors found $\rightarrow$ less programs accepted?
- Less programs accepted $\rightarrow$ less expressive?
- There are stronger type systems
    - A bit more complicated than λ-calculus
- Type inference becomes harder
    - Back to type annotations?

Overview

Simple Type Theory (STT) or $\lambda_\rightarrow$

Types

- Atomic types $\alpha \quad \beta \quad \gamma \quad \ldots$,
- Function types $\alpha \rightarrow \beta$.

For example: $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

Terms

- Variables with explicit types: $x_1^\sigma, x_2^\sigma, \ldots$
  - Countably many for each σ
- Applications: if $M : \sigma \rightarrow \tau$ and $N : \sigma$ then $(MN) : \tau$
- Abstractions: if $P : \tau$ then $(\lambda x^\sigma . P) : \sigma \rightarrow \tau$

Examples

$$\lambda x^\sigma . \lambda y^\tau . x : \sigma \rightarrow \tau \rightarrow \sigma$$

$$\lambda x^{\alpha \rightarrow \beta \rightarrow \gamma} . \lambda y^{\alpha \rightarrow \beta} . \lambda z^\alpha . xz : \beta \rightarrow \gamma$$

Conventions

Parentheses

- Types associate to the right
- Applications associate to the left

α-convertibility

$$\lambda x^\sigma. \ldots x \ldots x \ldots \approx_\alpha \lambda y^\sigma. \ldots y \ldots y \ldots$$

Capture avoiding substitution

$$M[x := N]$$

β-reduction

$$(\lambda x^\sigma.M)N \longrightarrow_\beta M[x := N]$$

Terms in STT ($\lambda_{\rightarrow}$)

- Can we find a term for every type?

Terms in STT ($\lambda_{\rightarrow}$)

- Can we find a term for every type?

$$x^{\alpha} : \alpha$$

- Can we find a closed term for every type?

Terms in STT ($\lambda_\rightarrow$)

- Can we find a term for every type?

$$x^\alpha : \alpha$$

- Can we find a closed term for every type?

$$(\alpha \rightarrow \alpha) \rightarrow \alpha$$

- No! Not every type is inhabited.

Overview

- Week 13 - Dependent Types
  - Summary of Week 12
  - Simple Type Theory
  - Curry-Howard
  - Dependent Types

Relation to logic

A typing judgement $M : \sigma$ can be read in two ways:

M is a function with the type σ

- term is an algorithm (program)
- type is its specification

M is a proof of the proposition σ

- type is a proposition
- term is its proof

One to one correspondence between

- Terms in $\lambda_{\rightarrow}$ (typable)
- Derivations in minimal propositional logic

Proof interpretation

Proof of $A \to B$

Function that maps proofs of A to proofs B

Proof of $A \land B$

Pair of proofs of A and B

Proof of $A \lor B$

Either a proof of A or a proof of B

Proof of $\forall x.P(x)$

Function that maps an object x to a proof of $P(x)$

Proof of $\bot$

Does not exist.Negation of A turns a proof of A into a nonexistant object

Typical questions in Type Theory

TCP (type checking problem)

$M : \sigma$?

TSP (type synthesis problem)

$M :?$

TIP (type inhabitation problem)

$? : \sigma$ (by a closed term)

Typical questions in Type Theory

TCP (type checking problem)

$M : \sigma?$

TSP (type synthesis problem)

$M :?$

TIP (type inhabitation problem)

$? : \sigma$ (by a closed term)

- For $\lambda_{\rightarrow}$ all are decidable
  - both with type annotations and without
- TCP and TSP are usually equivalent
  - application typing rule is to blame
- For more complicated systems TCP and TSP become undecidable
  - TIP corresponds to provability in some logic

Properties of $\lambda_\rightarrow$

- Uniqueness of Types

$$\text{If } \Gamma \vdash M : \sigma \text{ and } \Gamma \vdash M : \tau \text{ , then } \sigma = \tau.$$

- Subject Reduction

$$\text{If } \Gamma \vdash M : \sigma \text{ and } M \rightarrow_{\beta\eta} N \text{ , then } \Gamma \vdash N : \sigma.$$

- Strong Normalization

$$\text{If } \Gamma \vdash M : \sigma, \text{ then all } \beta\eta\text{-reductions from } M \text{ terminate.}$$

- Substitution Property

$$\text{If } \Gamma, x : \tau, \Delta \vdash M : \sigma, \Gamma \vdash P : \tau, \text{ then } \Gamma, \Delta \vdash M[x := P] : \sigma.$$

- Thinning

$$\text{If } \Gamma \vdash M : \sigma \text{ and } \Gamma \subset \Delta, \text{ then } \Delta \vdash M : \sigma.$$

- Strengthening

$$\text{If } \Gamma, x : \tau \vdash M : \sigma \text{ and } x \notin FV(M), \text{ then } \Gamma \vdash M : \sigma.$$

Overview

- Week 13 - Dependent Types
  - Summary of Week 12
  - Simple Type Theory
  - Curry-Howard
  - Dependent Types

Dependent types and more

Printf

What type does it have?

Dependent types and more

Printf

What type does it have?

Bit-strings of length n

- Type of bit-strings: $bs : \mathbb{N} \to \star$
- Bit-string made of zeros: $0_{bs} : (\forall n : \mathbb{N}) bs(n)$
- $\mathbb{R}^{\mathbb{N}}$

Vectors

- Type of hd?

Constructive Division

$$a/b/\!\!/P \quad \approx \quad \frac{a}{b \neq 0} \; .$$

Intuition behind λ_P

functions from A to B

$$A \to B$$

dependent functions from A to B

$$\Pi x : A.B$$

- Also called: dependent product
- Type of B can now depend on the argument x
- arrow type becomes a special case of dependent product

Properties of λ_P

- Possible to extend by an existential quantifier
    - Disjoint union (coproduct) of types
- Strong Normalization (using forgetting map)
- Church-Rosser (corollary)
- Subject Reduction
- Type reconstruction is decidable in PTIME.
    - Type checking is undecidable!
- Type inhabitation in λ_P is undecidable
    - ?

Properties of λ_P

- Possible to extend by an existential quantifier
  - Disjoint union (coproduct) of types
- Strong Normalization (using forgetting map)
- Church-Rosser (corollary)
- Subject Reduction
- Type reconstruction is decidable in PTIME.
  - Type checking is undecidable!
- Type inhabitation in λ_P is undecidable

  First order intuitionistic logic is undecidable

More logic

Types

- Types can show that programs have certain properties
- Types can show that programs terminate

But specifying and proving all the constraints gets tedious...

Proof Assistants

- Programs designed to prove properties
- Properties of programs (algorithms) and mathematical

Outlook

- Simple type theory corresponds to propositional logic
  - A proof of a proposition corresponds to a program of a type
- With dependent types
  - Predicate logic, Closer to Math
  - Epigram, Cayenne, Mizar
- Polymorphism
  - Limited in OCaml
- All possible dependencies
  - Foundation for Coq, Agda, Matita