Types

Informatics 2A: Lecture 20

Mirella Lapata (slides by SA, BW, JL)

School of Informatics
University of Edinburgh

12 November 2010

A variety of things

- Programming languages (particularly OO languages) give us the means to model things in the world.
- In OO languages, a class represents a whole family of possible objects corresponding to things of some particular kind (e.g., Student, Degree Programme, Start Date).
- In a computer, all kinds of data are represented by bit sequences (0s and 1s); but for different kinds of things, the representations may overlap.
- Sometimes this is tolerated: e.g., in C or Smalltalk we can be rather casual about representations.
- At other times, we want to keep track of the kind of thing we're working with. A type system is the usual way to do this.

The potential for error

- In programming languages, we have means of transforming data. The addition operation that takes two numbers and gives us back a new number.
- Transformations often only make sense on some kinds of data. E.g., what does "$3 + \textit{false}$" mean?
- So unless we check we are applying transformations to the right kind of thing, there is a potential to introduce information that doesn't really make sense.
- We often don't notice this until long after the first piece of rubbish is created, so it can be hard to track down such errors.

The potential for error

- In programming languages, we have means of transforming data. The addition operation that takes two numbers and gives us back a new number.
- Transformations often only make sense on some kinds of data. E.g., what does "$3 + \textit{false}$" mean?
- So unless we check we are applying transformations to the right kind of thing, there is a potential to introduce information that doesn't really make sense.
- We often don't notice this until long after the first piece of rubbish is created, so it can be hard to track down such errors.

So how do we go about reducing error?

Reducing error

Laissez faire: even if a transformation is not well defined for the data it's being used on, just go ahead and see what happens — in some cases it might be useful. (Common in C.)

Dynamic checking: system tags all representations with a record of what they're intended to represent, and all transformations check they're being applied to the right kind of thing. Then we can give better runtime errors. (Common in Python.)

Static Checking: define rules for the language that ensure a range of type errors cannot occur. A type error is where a transformation is applied to the wrong kind of thing. (Typical of Java or Haskell.)

Types in Programming Languages
Types in Natural Language
Introduction
Types and Type Systems
Types and Expressions
Type Structures

Reducing error

Laissez faire: even if a transformation is not well defined for the data it's being used on, just go ahead and see what happens — in some cases it might be useful. (Common in C.)

Dynamic checking: system tags all representations with a record of what they're intended to represent, and all transformations check they're being applied to the right kind of thing. Then we can give better runtime errors. (Common in Python.)

Static Checking: define rules for the language that ensure a range of type errors cannot occur. A type error is where a transformation is applied to the wrong kind of thing. (Typical of Java or Haskell.)

We'll concentrate on static checking today — how to capture aspects of the language that aren't easily captured by CFGs.

Types and Type Systems

A type

Is a collection of values (or the computer representation of those values) all of which have some similarity in the roles they can play. E.g., numbers, boolean truth values, characters, . . .

A type system

- Defines a collection of atomic or basic types.
- Provides ways of building complex types out of simple ones.
- Allows us to assign a type to certain programming language phrases, e.g. expressions. Expressions in programming languages (e.g. `x + 3`, `P & Q`) are a way of talking about values, and type systems allow us to say which type the value should belong to.

An Example Type System

Basic Types: bool (truth val), num (numbers), char (characters).

Type Constructors: operators that take types and combine them to form more complex types. If A and B are types then:

- **Product:** $A \times B$ is a type. Its values are pairs (a, b) where a has type A and b has type B.
- **Sum:** $A + B$ is a type. Values are of the form $i(a)$ or $j(b)$, where a has type A, b has type B.
- **List:** $\mathrm{List}(A)$ is a type with sequences of values of type A.
- **Record:** If $A_1, \ldots, A_k$ are types then $\{f_1 : A_1, \ldots f_k : A_k\}$ is a type whose values are *labelled records* with labels $f_1, \ldots f_k$
- **Function:** In languages where functions are "first-class objects" (e.g. Haskell), the type of all functions (representable in the language) from A to B is $A \to B$.

Connecting Types and Expressions in the Language

- For the base types, we usually have some direct way of writing down values. For example, tt: **bool**, ff: **bool**, 1: **num**, ...
- For the structured types:
  if $a : A,\quad b : B,\quad a_i : A_i,\ 1 \leq i \leq k,\quad b_1 : B, \ldots b_n : B$
  then $(a, b) : A \times B,\quad i(a) : A + B,\quad [b_1, \ldots b_n] : \mathrm{List}(B),$
  $\{f_1 = a_1, \ldots f_k = a_k\} : \{f_1 : A_1, \ldots f_k : A_k\}$

$$\{\mathrm{place} = \{\mathrm{north} = (55, 57), \mathrm{west} = (3, 13)\}, \mathrm{time} = (6, (12, 07))\}$$

has type

$$\{\mathrm{place} : \{\mathrm{north} : \mathrm{num} \times \mathrm{num}, \mathrm{west} : \mathrm{num} \times \mathrm{num}\},\ \mathrm{time} : \mathrm{num} \times (\mathrm{num} \times \mathrm{num})\}$$

Computing with structured values

For each of the ways of building up values of types, there is also a way of taking them apart again:

- for pairs: $\mathbf{fst}((x, y)) = x$ and $\mathbf{snd}(x, y) = y$
- for sums: **case** x **in** $\{i(a) \Rightarrow e_1 \mid j(b) \Rightarrow e_2\}$
- for records: **open** x **in** $\{\{f_1 = x_1, \ldots f_k = x_k\} \Rightarrow e\}$
- for functions: $f(x)$

open x **in**
  { { place = a, time = b } ⇒
    open a **in**
      { { north = c, west = d } ⇒
        c
      }
    }
  }

Dealing with variables

- Expressions often contain *variables*, e.g. `x+1`.
- To deal with variables, we define the type of expressions relative to an *environment* `E` that tells us the types of any variables involved.
- In the example, we ask the question: what is the type of `x+1` when we know the environment is { `x: num` }
- In this case the expression has type `num`.
- What happens with {`bool x; x = x+1;` }?

Functions

- If we use `fn` to stand for the λ of Haskell, then if we are trying to find a type for `fn x.e` in environment `E` we:
  - Find the type of `e` in the environment `E` augmented with {`x:A`} for some appropriate type `A`
  - If `e` has type `B` in this environment then we can say `fn x.e: A->B` in environment `E`.
- We can express such ideas formally by means of *typing rules*:

$$\frac{E, x : A \vdash e : B}{E \vdash \mathtt{fn}\ x.e : A\text{->}B} \qquad \frac{E \vdash f : A\text{->}B \qquad E \vdash e : A}{E \vdash f(e) : B}$$

Rules like this allow us to capture a lot of ‘rules of the language’ that can’t (readily) be captured by CFGs alone.

Structure in the basic types

- Often in programming languages there is not much relationship between the basic types.
- But in some languages there can be: for example, if we had two number types **float** and **int** standing for floating point and integer numbers.Then anywhere we can use a **float**, we can also use an **int**.
- This is the beginning of the notion of *subtyping*. We write **int** $<:$ **float** to mean **int** is a subtype of **float**. (Idea: anywhere a **float** is allowed, an **int** is allowed too).
- These ideas extend to records where, in general: $\{f_1 : A_1, \ldots f_k : A_k\} :> \{f_1 : A_1, \ldots f_k : A_k, b_1, \ldots b_m : A_m\}$. (I.e., anywhere we can use something of the left hand type, we can also use something of the right hand type.)

Subtyping for function types

What relationship must hold between types to have $A \to B <: C \to D$?

$$\frac{C <: A \quad B <: D}{A \to B <: C \to D}$$

This sort of thing is relevant to understanding e.g., how *method overriding* works in languages like Java.

Agreement phenomena

In programming languages, typing rules can be used to enforce *type agreement* between widely separated parts of a program.

```
(fn x. if x==1 then ...) (2)
```

There are similar phenomena in NL: constituents of a sentence (often widely separated) may be constrained to agree on an attribute such as person, number, gender.

- You, I imagine, are unable to attend.
- The hills are looking lovely today, aren't they?
- He came very close to injuring himself.

Types in Natural Language Semantics

Types are also very useful if we wish to describe the semantics (i.e., meaning) of natural languages. For example, we can use types employed in logic to model the meanings of various NL phrase types.

Basic Types

1. e — the type of real-world *entities* such as Inf2a, Stuart, John.
2. t — the type of *facts with truth value* like 'Inf2a is amusing'.

These two basic types enable us to construct complex types using e.g., the function type constructor.

From basic to complex formal types

Complex Types

- $<e,t>$: **unary predicates** – things that are functions from entities to facts.
- $<e, <e,t>>$: **binary predicates** – things that are functions from entities to unary predicates.
- $<<e,t>, t>$: **type-raised entities** – things that are functions from unary predicates to truth values.

N.B. where computer scientists write $\sigma \rightarrow \tau$, linguists sometimes write $< \sigma, \tau >$.

- Inf2a, Stuart **:** e
- enjoys **:** $<e, <e,t>>$
- enjoys Inf2a, is amusing **:** $<e,t>$
- Inf2a is amusing, Stuart enjoys Inf2a **:** t
- every student **:** $<<e,t>, t>$

Subtypes in NL

```
hamburger <: sandwich <: food item <: food
< substance <: matter <: physical entity <: entity
```

- To deal with meanings in NL, much more fine-grained classifications (of varying levels of specificity) are often useful.
- There are also many other more abstract types of entities to which a NL expression may refer: e.g., locations, points in time, time spans, events, beliefs, desires, possibilities, . . .
- This leads to a vast system of subtypes capturing information about real-world concepts and their relationships.
  (Cf. the **WordNet** database.)

Selectional restrictions

We can often characterize verbs and other predicates in terms of their selectional restrictions — constraints on the type of entities or expression can serve as their arguments. arguments.

- I want to eat somewhere close to Appleton Tower.
- I want to eat some Thai food.
- I want to eat some radio.

- The object of eating is usually something *edible*: Its semantic type is *edible things*.
- The location of an event is usually a *place*: Its semantic type is *location*.

Selectional restrictions

Selectional restrictions are associated with word senses, not words:

- Do any international airlines serve vegan meals?
  (ie, *provide food or drink*)
- Do any international airlines serve Edinburgh?
- (ie, *provide a service*)
- ?? Do any international airlines serve Edinburgh and vegan meals?

Selectional restrictions vary in their specificity:

OBJECT(imagine): a situation
OBJECT(diagonalise): a matrix

⇒ Verbs vary in the specificity of their argument types.

Selectional restrictions and type coercion

Selectional restrictions can change the way we interpret a term:

- Jane Austen wrote 'Emma'.
- I used to read Jane Austen a lot.

- The chicken was domesticated in Asia.
- The chicken was overcooked.

Metonymy is when the referent of a term changes to a related entity, often associated with the demands of a verb's selectional restrictions.

Clicker Questions

Which of the following is *not* a type for *ostrich*?

1. bird
2. vertebrate
3. artefact
4. animal

Clicker Questions

Which of the following is *not* a type for *ostrich*?

1. bird
2. vertebrate
3. artefact
4. animal

Which is the correct type for the expression *I love Inf2a!*?

1. $<e, <e,t>>$
2. t
3. $<<e,t>, t>$
4. $<e,t>$

Summary

- Types help avoid failure in computation
- We can use the structure of the program to check that type constraints are being observed.
- Type systems for programming languages can become quite complex, particularly for OO and functional languages.
- Types are also relevant in Natural Languages.
- There are general types associated with syntax, and more specific types associated with verb (predicate) arguments.
- Type coercion is common in Natural Language, changing the type (and often the referent) of an expression to one that fits the verb (predicate) to which it serves as an argument.