

Types

Informatics 2A: Lecture 20

Stuart Anderson & Bonnie Webber

School of Informatics
University of Edinburgh
soa@inf.ed.ac.uk, bonnie@inf.ed.ac.uk

10 November 2009

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 “class” of instances or objects that we can use to model things in the world.
- In any computer:
 - we use some bit sequence to represent particular objects
 - the representations are usually fairly uniform for similar things
 - for different kinds of things the representations may overlap
- Sometimes this is fine e.g. in C or Smalltalk we can be rather casual about representations.
- In other circumstances we want to keep track of the kind of thing we are working with.
- A *type system* is the usual way to do this.

1 Types in Programming Languages

- Introduction
- Types and Type Systems
- Types and Expressions
- Type Structures

2 Types in Natural Language

- Formal types
- Types of entities
- Types as selectional restrictions

Reading:

J&M, 2nd ed., Chapter 17.2 – 17.3, 17.5, Chapter 19.4

The potential for error

- In programming languages we have ways of structuring information and a means of transforming old information into new.
- For example, we have the addition operation that takes two numbers and gives us back a new number.
- These transformations often only make sense on some kinds of data, they are not defined for all data e.g. what does $3 \times \text{false}$ mean?
- So in any programming system, 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 very hard to track down such errors.

Reducing error

- As programming languages have developed a number of distinct approaches have developed:
 - Laissez faire*: even if a transformation is not well defined for the data its being used on just go ahead and see what happens - in some cases it might be useful.
 - Dynamic checking*: where the system tags all representations with a record of what they are intended to represent and all transformations check they are being applied to the right kind of thing. The we can give better runtime errors.
 - Static Checking*: where we 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.
- We just consider static checking today because it helps provide a link to the lectures on the semantics of programming languages.

Types and Type Systems

- A *type* is a collection of values (or the computer representation of those values) all of which have some similarity e.g. numbers, truth values, characters, ...
- A *type system* does three things:
 - It identifies a collection of *atomic* or *basic* types.
 - It identifies ways of building new *structured* types from other types
 - It allows us to provide a type for some programming language terms. In particular there is usually a syntax for expressions. Expressions in programming languages are a way of talking about values and type systems say which type the value should belong to.

An Example Type System

Basic Types: **bool** – truth values, **num** – numbers, **char** – characters.

Type Constructors: Here we have some operators that take types and combine them to form more complex types. If A and B are types then we might have the following operators:

- Product: $A \times B$ is a type and the values of $A \times B$ are pairs (a, b) where a is a value of type A and b is a value of type B .
- Sum: $A + B$ is a type and the values are of the form $i(a)$ and $j(b)$ where a is a value of A and b is a value of B
- List: $\text{List}(A)$ is a type whose values are finite sequences of values of A .

Example Type System ctd.

More Type Constructors:

- Record: If A_1, \dots, A_k are types then $\{f_1 : A_1, \dots, f_k : A_k\}$ is a type whose values are labelled records and the labels are f_1, \dots, f_k
- Function: In languages where functions are “first-class objects” the type of all functions (representable in the language) from A to B is $A \rightarrow B$.

Structure in the basic types

- Often in programming languages there is no relationship between the basic types.
- In some languages there can be, for example, if we had two number types **float** and **int** standing for floating point and integer numbers. The anywhere we can use a **float** we can use an **int**.
- In natural language there is a rich structure in the basic types.
- This is the beginning of the notion of *subtyping* we write **float** > **int** to mean **int** is a *subtype* of **float** and anywhere we can use an **int** instead.
- These ideas extend to records where, in general:
 $\{f_1 : A_1, \dots, f_k : A_k\} > \{f_1 : A_1, \dots, f_k : A_k, b_1, \dots, b_m : A_m\}$
 (i.e. anywhere we can use the left hand type we can also use something of the righthand type.)

Functions

- What relationship must hold between types to have $A \rightarrow B > C \rightarrow D$?

Entities and formulae as a basis for formal types

We can use the types employed in **first-order logic** to specify formally some of the types needed for Natural Languages.

Basic types:

- e – the type of entities such as *Inf2a*, *Stuart*, *Bonnie*
- t – the type of formulae (expressions with truth values) such as *Inf2a is amusing*.

These two basic types enable us to construct **complex types** as functions from one type σ to another type τ .

From basic formal types to complex formal types

- $\langle e, t \rangle$ – things that are functions from entities to truth values – i.e. **unary predicates**.
- $\langle e, \langle e, t \rangle \rangle$ – things that are functions from entities to unary predicates – i.e. **binary predicates**.
- $\langle \langle e, t \rangle, t \rangle$ – things that are functions from unary predicates to truth values, called **type-raised entities**.

- e : inf2a, Stuart
- $\langle e, \langle e, t \rangle \rangle$: enjoy
- $\langle e, t \rangle$: enjoy inf2a, amusing
- t : inf2a is amusing, Stuart enjoys inf2a

These basic and complex types underpin the semantics used in Lecture 22, NLTK, Tutorial 8 and Assignment 3.

Different types of entities in NL

We can distinguish those entities we can **count** and those we can't:

- A student kept **a chicken** in her room.
 - A student kept **two chickens** in her room.
 - I ate **rice** and drank **milk**.
 - *I ate **two rices** and drank **two milks**.
- **individuals** (things we can count): one student, two students, one chicken, many chickens, one room, many rooms
 - **mass** (things we can't count): rice, milk

Switching types: Coercion

Sometimes changing syntactic form changes type interpretation:

- I drank **beer** last night.
- I had **a beer** last night. (⇒ one glass of beer)
- I tasted **two beers** last night. (⇒ two kinds of beer)
- I ate **chicken** for lunch. (⇒ some meat, not one chicken!)
- There were **chickens** all over the floor. (⇒ many chickens)
- There was **chicken** all over the floor. (⇒ bits of chicken)

This is one illustration of **type coercion** in Natural Language.

Other general types

Individuals entities can form **groups**, distinct from their individual components:

- The two students embraced.
- *Each of the students (separately) embraced.
- The Inf2a students danced "strip the willow" in the AT foyer.
- *Each Inf2a student danced "strip the willow" in the AT foyer.

Besides individuals, masses and **groups**, **events** can be considered entities:

- John heard me.
- John heard me eating a chicken sandwich.
- John heard that I ate a chicken sandwich.

Other general types

Time point: 4pm, 1600, November 2, 2007

Time span: an hour, 30 days, two months, from 4pm onwards

Start of an event:

Culmination of an event:

Type coercion can happen here too:

- They danced "strip the willow" for 30 minutes.
- They danced "strip the willow" at 4pm (⇒ started dancing)
- They danced "strip the willow" until 4pm (⇒ finished dancing)

Other general types

Beliefs: *I believe John ate a chicken sandwich.*

Desires: *I want to eat a chicken sandwich.
I want Fred to eat a chicken sandwich.*

Possibilities: *Fred might eat a chicken sandwich.*

Other "attitudes": ...

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.

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

How do we know that *Thai food* is the **object** of the eating event in the second sentence, and that *somewhere close to BP* is the **location** of the eating event in the first?

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 with words:

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

Selectional restrictions vary in their specificity:

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

⇒ Verbs as operators vary in the specificity of their argument types.

Selectional restrictions and type coercion

Selectional restrictions can change the way we interpret a term:

- o IBM makes inexpensive laptops.
- o IBM fell 10 points today on the NY Stock exchange.
- o IBM announced the introduction of new laptop.

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

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.