Computing Fundamentals 2
Introduction to CafeOBJ

Lecturer: Patrick Browne
Lecture Room: K408
Lab Room: A308

Based on work by: Nakamura Masaki, João Pascoal Faria, Prof. Heinrich Hußmann. See notes on slides for details.
Why a specification language?

- Programming languages (PLs), like Java, are formal specifications. They specify what the machine should do. They must convert language keywords into the actual machine instructions that the CPU can execute.
Why a specification language?

- However, PLs are not good at expressing our understanding of a domain. They are too low level and contain too much detail. There is no distinction between what the system does and what it **should do**. Using PLs it is difficult to identify design errors, assumptions are not made explicit, they are implementation (or language) dependent. They focus on **how** system tasks are performed. They don’t easily allow us to **reason about** the system or describe **what** the system does.
Why a specification language?

- The following equation is a specification and when it is run it becomes part of a program to perform actual sorting (see footnote for full program)

- \( \text{ctrans} (E \; E') \Rightarrow (E' \; E) \text{ if } (E' \leq E) \text{ and } (E \neq E') \).
Why a specification language?

- Domains of interest include the theory of sets, the theory of integers, the theory of bank accounts, the theory of a hospital record systems. For this course we are primarily interested in specifying mathematical systems, such as set theory or arithmetic.
Why a specification language?

- CafeOBJ allows us to reason about what system a system does. It is also a programming language that allows us to encode algorithms.
- CafeOBJ is itself is based on rigorous mathematic (equational logic, a lattice of types, functions, equations, axioms, rigorously defined syntactic and semantic domains). This makes CafeOBJ useful for teaching mathematics and computing.
Why a specification language?

- CafeOBJ is executable. Even though it allows us to focus the **what** a program **should** do (i.e. the intended specification), CafeOBJ also allows to run ‘specifications’ and produce results. Thus we can check our work (prototype) **and** get a result (compute). In short we can:
  - Specify
  - Prototype
  - Reason and study proofs
  - Compute
Why a specification language?

- CafeOBJ has conceptual semantics based on the logic of equations.
- CafeOBJ has operational semantics based on term rewriting.
- These two interpretations are closely related. Usually we talk about syntactic 'terms' and semantic 'expressions'.
- A good specification should be written in such a way that it can be used to predict how the system will behave.
What can CafeOBJ be used for?

– Formal description of systems
– Automatic checking for
  • Ambiguity
  • Incompleteness
  • Inconsistency
Don’t Panic!

• The role of CafeOBJ on this course is to provide a functional and logic based language that can be used to represent the mathematical concepts such as logic, sets, algebras and functions. While you are expected to be able to read and understand small pieces of CafeOBJ code it is not intended to study the entire language in great depth.
## Predefined Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Type</th>
<th>Operators</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>Bool</td>
<td>not and and-also xor or or-else</td>
<td>true, false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>implies iff</td>
<td></td>
</tr>
<tr>
<td>NAT</td>
<td>Nat</td>
<td>* + &gt;= &gt; &lt;= &lt; quo rem divides s p</td>
<td>0 1 2 3 ...</td>
</tr>
<tr>
<td></td>
<td>NzNat</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>Int</td>
<td>Idem, mais - e –</td>
<td>-2 ...</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Float</td>
<td>* / + - &lt; &lt;= &gt; &gt;= exp log sqrt abs sin atan etc.</td>
<td>1.2 ... pi</td>
</tr>
<tr>
<td>STRING</td>
<td>String</td>
<td>string= string&gt; string&lt; string&gt;= string&lt;= string&lt; length substring ++ upcase downcase etc.</td>
<td>“a string”</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Based on notes by João Pascoal Faria at: http://paginas.fe.up.pt/~jpf/teach/MFES/index.html
CafeOBJ Modules - syntax

module module_name {
    imports {
        module_element ...
    }
    signature {
        module_element ...
    }
    axioms {
        module_element ...
    }
}
Modules

• Imports
  – Very like the concept of extends in Java
  – Allows us to reuse other modules

• Signature definitions
  – Like method declarations in Java
  – Describe operations without implementation details (a bit like interfaces in Java)
Modules

• Axioms
  – The basis for a logical argument
  – Axioms can be assumptions which
    • Are reasonable in a given domain
    • Related by logical steps
    • Allow us to draw a conclusion
    • Allow us to reason about our modules
    • Allow us to express restrictions (or rules) on our models.
Axioms — unconditional equations

• A basic equation is an axiom

• \[ eq \ X + 0 = X \]

• Built of
  – Sorts
  – Operators & Constants
  – Variables
module SIMPLE-NAT {
  signature {
    [ Zero NzNat < Nat ]
    op 0 : -> Zero
    op s : Nat -> NzNat
    op _+_ : Nat Nat -> Nat
  }
  axioms {
    vars N N' : Nat
    eq 0 + N = N .
    eq N + s(N') = s(N + N') .
  }
}
Axioms

• Axioms from previous example:
  
  \[ \text{eq} \ 0 + N = N \ . \]
  \[ \text{eq} \ N + s(N') = s(N + N') \ . \]

• Using axioms, we can **reduce** (or execute) terms
  
  reduce \ (0 + N) + N \ .
  reduce \ 0 + s(N) \ .
Axioms – conditional equations

• Axioms can also be conditional:

\[
\begin{align*}
\text{op } _\leq_ & : \text{DInt DInt} \to \text{Bool} \\
\text{ceq } M \leq N & = \text{true if } M < N . \\
\text{ceq } M \leq N & = \text{true if } M == N . \\
\text{ceq } M \leq N & = \text{false if } N < M . \\
\text{eq } M \leq (M + N) & = \text{true} . \\
\text{eq } N \leq (M + N) & = \text{true} .
\end{align*}
\]
Downloading CafeOBJ

- You can download binary packages for Windows, Mac, and Linux from http://www.ldl.jaist.ac.jp/cafeobj/
CafeOBJ

• CafeOBJ is a declarative language with firm mathematical and logical foundations. The mathematical semantics of CafeOBJ is based on state-of-the-art algebraic specification concepts and results, and is strongly based on category theory and the theory of institutions.

• Similar to the course text by Gries Schneider, CafeOBJ also uses *Equational logic*, which is the calculus of replacing terms by equal terms.
CafeOBJ

- Equational calculus derives (proves) a term equation from a conditional-equational axiom set. The deduction rules in this calculus are:
  - **Reflexivity**: Any term is provably equal to itself \((t = t)\).
  - **Transitivity**: If \(t_1\) is provably equal to \(t_2\) and \(t_2\) is provably equal to \(t_3\), then \(t_1\) is provably equal to \(t_3\).
  - **Symmetry**: If \(t_1\) is provably equal to \(t_2\), then \(t_2\) is provably equal to \(t_1\).
  - **Congruence**: If \(t_1\) is provably equal to \(t_2\), then any two terms are provably equal which consist of some context built around \(t_1\) and \(t_2\). Compare congruence like Leibniz.
  - \(\text{op}(a) = \text{op}(a') \iff a = a'\)
equational entailment

\[ R. \quad \frac{}{\{(\forall X)t = t\}} \]

\[ S. \quad \frac{\{(\forall X)t = t'\}}{\{(\forall X)t' = t\}} \]

\[ T. \quad \frac{\{(\forall X)t = t', (\forall X)t' = t''\}}{\{(\forall X)t = t''\}} \]

\[ OC. \quad \frac{\{(\forall X)t_i = t'_i | 1 \leq i \leq n\}}{\{(\forall X)\sigma(t_1,...,t_n) = \sigma(t'_1,...,t'_n)\}} \quad \text{for each operation symbol } \sigma \in F \]

\[ \Gamma. \quad \frac{\Gamma \cup \{(\forall X)H\theta\}}{\{(\forall X)C\theta\}} \quad \text{for all } (\forall Y)C \text{ if } H \in \Gamma, \text{ subst } \theta : Y \rightarrow X \]

inference rules are sound and complete,
CafeOBJ executes expressions using Term Rewriting

Apply the given equations to

- **ground terms** (terms without variables), as left-to-right rewrite rules, progressively transforming them until a
- **normal form** is reached (where no further rule is applicable).
CafeOBJ operations can have Mathematical properties

• **Associative**
  
  \[(X + (Y + Z)) = ((X + Y) + Z)\]
  \[(X \text{ and } (Y \text{ and } Z)) = ((X \text{ and } Y) \text{ and } Z)\]

• **Commutative**
  
  \[(X + Y) = (Y + X)\]
  \[(X \text{ and } Y) = (Y \text{ and } X)\]

• **Identity**
  
  \[(X + 0) = (0 + X) = X\]
  \[(X \text{ and } \text{true}) = (\text{true and } X) = X\]
Functions 1) as program and 2) as equations

- Functions can be considered as a restricted form of relation. This is useful because the terminology and theory of relations carries over to function.
- In programming languages like C or CafeOBJ a function can have a signature, which includes its name, type of argument and the type of the expected return value.
Fibonacci Function in Java and CafeOBJ

Function Signature has:

- a name
- argument(s) type
- and return type

## Class Fibonacci

```java
class Fibonacci {
    static int fib(int n) {
        if (n < 2) { return n; }
        else { return fib(n-1) + fib(n-2); }
    }
    -- test main
}
```

---

```obj
mod* FIBO-NAT {
    pr(INT)
    op fib : Int -> Int
    var N : Int
    eq fib(0) = 0.
    eq fib(1) = 1.
    ceq fib(N) = fib(p(N)) + fib(p(p(N)))
    if N > 1.
}
```
Fibonacci Function in Java and CafeOBJ

Signature of a function consists of a name, argument(s) type, and return type.

class Fibonacci {
    static int fib(int n) {
        if (n < 2) {return n;}
        else { return 
            fib(n-1)+fib(n-2);}
    }
    -- test main}

mod* FIBO-NAT {
    pr(INT)
    op fib : Int -> Int
    var N : Int
    eq fib(0) = 0 .
    eq fib(1) = 1 .
    ceq fib(N) =
        fib(p(N)) + fib(p(p(N)))
    if N > 1 .}

Return type

Argument constraints
Writing equations in CafeOBJ

• In general when writing algebraic specification one should include axioms that apply each observer or extension to each generator, except where an error might occur.

• The RHS should not be a single variable, unless you want it to hold a value (rather than a definition).

• In general, all variables in the RHS must appear in the LHS.

• The scope of a constant is the module it is declared in (or a proof score or interactive session).

• The scope of a variable is only inside of the equation. So, during evaluation a variable, say X, will have the same value in a given equation, but it may have different values in different equations in the same module.

• We would like to use equations to partition sets.
Types and subtypes¹

mod SORTS {
  [ s4 < s2 s3 < s1, s5 < s1, s6, t]
  op f1 : -> s1
  op f2 : -> s2
  op f3 : -> s3
  op f4 : -> s4
  op f5 : -> s5
  op f6 : -> s6
  op g : s2 -> t
}
Types and subtypes

\[ s_4 < s_2 \ s_3 < s_1 \ s_5 < s_1 \ s_6 \ t \]

- op f1 : \(\rightarrow s_1\)
- op f2 : \(\rightarrow s_2\)
- op f3 : \(\rightarrow s_3\)
- op f4 : \(\rightarrow s_4\)
- op f5 : \(\rightarrow s_5\)
- op g : s_2 \rightarrow t

-- A set of constants and a function

Types form a partial order (reflexive, antisymmetric, and transitive) interpreted as:

\[ t, s_6, s_5 < s_1, s_1, s_2 \ s_3 \ s_5 < s_1, s_3, s_4 < s_3 < s_1, s_2, s_4 < s_2 < s_1, s_4, s_4 < s_2 \ s_3 \]

\[ \text{OK} \] \quad \text{Error Sort} \quad \text{No Parse}

- g(f2)
- g(f4)
- g(f1)
- g(f3)
- g(f5)
- g(f6)
A partial order of Types

[ s4 < s2, s3 < s1, s5 < s1, s6 < t ]
Types and subtypes

{signature
  [ s4 < s2, s3 < s1, s5 < s1, s6 t]
  op f1 : -> s1
  op f2 : -> s2
  op f3 : -> s3
  op f4 : -> s4
  op f5 : -> s5
  op f6 : -> s6
  op g : s2 -> t
}

OK
  g(f2)
  g(f4)

Error Sort
  g(f1)
  g(f3)
  g(f5)

No Parse
  g(f6)
Subtraction

-- SUB extends NAT with subtraction (see note)
module SUB {
  using (NAT)
  [Nat]
  op _ - _ : Nat Nat -> Nat
}

vars N M : Nat
-- base case
eq N - N = 0 .
-- conditional equation for recursive case
ceq N - M = s(N - s(M)) if N > M .
}
-- execute specification.
open SUB
reduce 5 - 3 .
reduce 3 - 5.
SUB extends NAT with subtraction (see note)
module SUB {
    using (NAT) import [Nat]
    op _ - _ : Nat Nat -> Nat
}

vars N M : Nat
base case
eq N - N = 0 . -- (1)
conditional equation for recursive case
ceq N - M = s(N - s(M)) if N > M . --(2)
}
execute specification.
select SUB
reduce 5 - 3 .
reduce 3 - 5.

(3) eq s(N) = #! (+ n 1) (defined in NAT)

Three equations are used for the calculation (5-3), one from NAT. See notes section

In the definition of NAT there is a successor function call s() which is called recursively while N>M. So at this point the recursion ‘unwinds’
Running CafeOBJ

• Typically there are four things that you need to do to work on a file:
  1. Start CafeOBJ, using CafeOBJ.bat
  2. Set CafeOBJ to point to files in your current working directory (or folder):
     \texttt{cd yourDir}
  3. Load the file you wish to work on:
     \texttt{in workingFile}
  4. Open the module you wish to working with:
     \texttt{open ModuleName}

• The current open module is called the 'current context'. Type '?' for help.

• The following slides are from: How to use CafeOBJ systems, NAKAMURA, Masaki, Language Design Laboratory, Japan Advanced Institute of Science and Technology
Module elements

- A (basic) module consists of a signature and an axiom:

```plaintext
mod! NAT+ {
  [Zero NzNat < Nat]
  op 0 : -> Zero
  op s : Nat -> NzNat
  op + : Nat Nat -> Nat
}
```

Signature

```plaintext
vars M N : Nat
eq +( 0, N ) = N .
eq +(s(M), N ) = s(+{M, N}) .
```

Axiom
Proof score

- For more complicated verification, a proof score is used.
- A proof score begins at "open" command which opens a module, and ends with "close" command.
- While a module is opening (between open and close), we can declare operations and equations to do verification.

```plaintext
NAT+> open NAT+
  -- opening module NAT+.. done.
%NAT+> op n : -> Nat .
%NAT+> eq n = 0 .

---
%NAT+> red +(n, n) == 0 .
*
  -- reduce in %NAT+ : +(n,n) == 0
true : Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 4 matches)
%NAT+> close
NAT+>
```

Lecture note 2 for i613 (2005.10)

NAKAMURA, Masaki, Language Design Laboratory, Japan Advanced Institute of Science and Technology
Arbitrary element

• After opening a module, a declared constant operation 
  \texttt{\texttt{op e : -> s .}} stands for an arbitrary element of the 
  sort \texttt{s} whose scope is from its declaration to the end of 
  a proof score (i.e. \texttt{close}).

\begin{verbatim}
NAT+> open NAT+
  -- opening module NAT+.. done.
%NAT+> op n : -> Nat .
%NAT+> red +(0, n) == n .
  -- reduce in %NAT+ : +(0,n) == n
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
%NAT+> close
NAT+>
\end{verbatim}

This is a proof score for the claim that \texttt{\texttt{+(0, N) = N for any}}
\texttt{natural number N}}. Since all reductions return \texttt{“true”}, it holds.

Lecture note 2 for i613 (2005.10)

NAKAMURA, Masaki, Language Design Laboratory, Japan Advanced
Institute of Science and Technology
While a module is opening, a declared equation stands for an assumption of the proof score.

```
NAT+> open NAT+
    -- opening module NAT+.. done.
%NAT+> op n : -> Nat .
%NAT+> eq +(n, 0) = n .
-
%NAT+> red +(s(n), 0) == s(n) .
*
    -- reduce in %NAT+ : +(s(n),0) == s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
%NAT+>
```

This is a proof for “$+(N, 0) = N$ implies $+(s(N), 0) = s(N)$ for any natural number $N$” (it holds).

Lecture note 2 for i613 (2005.10)
Constant v.s. variable

- If we use a variable instead of a constant, the meaning of the proof score may be changed.
  - The scope of a constant is to the end of a proof score.
  - The scope of a variable is inside of the equation.

```
open NAT+
op n : -> Nat .
eq +(n, 0) = n .
red +(s(n), 0) == s(n) .
close
```
```
open NAT+
var N : Nat .
eq +(N, 0) = N .
red +(s(N), 0) == s(N) .
close
```

Constant: \( \forall N: \text{Nat}. \ [ + (N, 0) = N \Rightarrow + (s(N), 0) = s(N) ] \)"

Variable: \( \forall N: \text{Nat}. [ + (N, 0) = N ] \Rightarrow \forall N: \text{Nat}. [ + (s(N), 0) = s(N) ] \)"

NAKAMURA, Masaki, Language Design Laboratory, Japan Advanced Institute of Science and Technology
Inductive theorem

- For a tight module, we can use the structural induction to do verification.
- Structural induction for “∀N: Nat. P(N)” is done as follows:
  - **Base case**: Prove “P(0)”.
  - **Induction step**: Assume “P(n)” and prove “P(s(n))” where “P(n)” is called the induction hypothesis.
- If the above holds, “∀n: Nat. P(n)” is called an inductive theorem.
The following is a proof score of \( \forall n: \text{Nat}. + (n, 0) = n \):

```
open NAT+
red + (0, 0) == 0 .
op n : -> Nat .
eq + (n, 0) = n .
red + (s(n), 0) == s(n) .
close
```

```
-- opening module NAT+.. done.
%!NAT+> -- reduce in %NAT+ : +(0,0) == 0
true : Bool
(0.000 sec for parse, 2 rewrites(0.000 sec), 2 matches)
...
-- reduce in %NAT+ : +(s(n),0) == s(n)
true : Bool
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
%!NAT+>
NAT+>
```
Automatic Theorem Proving

The CafeObj environment includes automatic theorem proving (ATP) software. ATP will try to show that some statement (often called a conjecture) is a logical consequence of a set of statements (often called axioms). The ATP in CafeOBJ is called PIGNOSE and is based on OTTER. Most of the logic problems on this course use PIGNOSE. For our purposes we set up the problems such as Portia, Superman, and Family using a convenient\textsuperscript{1} logical notation (\(\rightarrow, \leftarrow, \mid, \& ~, \neg, \text{ax, goal}\))
Automatic Theorem Proving

The basic steps in our examples are:

1) That we declare some predicates, with the keyword `pred`.

2) Then we define the axioms with a convenient logical notation (`\rightarrow`, `\leftarrow`, `|`, `&`, `\sim`, `ax`)

3) Then we set the conjecture to be proved.

4) If PIGNOSE can prove the conjecture we can examine the proof. We may compare it to the manual proof.
A Proof

• Prove that 0 is the right identity of ‘+’
• Base case:
  \[ 0 + 0 = 0 \]
• Inductive hypothesis:
  if \( a + 0 = a \) -- assumed
  then \( s(a) + 0 = s(a) \) -- expected
Proofs

module SIMPLE-NAT { 
    [Zero NzNat < Nat]

    op 0 : -> Zero
    op s : Nat -> Nat
    op _+_ : Nat Nat -> Nat

    vars X, Y : Nat
    eq 0 + X = X .
    eq s(X) + Y = s(X + Y) .
}
Operator Attributes

• Attributes of the operator rather than the operands

• Categories:
  – Equational theory
  – Parsing
  – Constructor
  – Evaluation strategy
Operator Attributes

- Equational Theory Attributes
  - Associativity
    \[(x + y) + z = x + (y + z)\] for any \(x, y, z\)
  - Commutativity
    \[x + y = y + x\] for any \(x, y\)
Operator Attributes

• \_and\_ is idempotent.

\[ \text{op } \_\text{and}_\_ : \text{Bool } \rightarrow \text{Bool} \text{ } \{ \text{assoc comm id: true idem } \} \]

• \text{idem is equivalent to the equation}

• \text{eq } X \text{ and } X = X .

• In this case idempotence means that \( X \) anded with itself gives \( X \). For example, “true and true” and “true” matches modulo idempotency.
Operator Attributes

- Equational Theory Attributes
  - Identity
    
    id: \( I \) (where \( I \) is the identity in question)
    
    \( x + 0 = x \) for any \( x \)
    
    eg: \( \text{op } _+\_ : \text{Nat Nat } \rightarrow \text{Nat \{id: 0\}} \)
Operator Attributes

• Parsing Attributes
  – Precedence
    • \texttt{op \_+_ : Int Int \rightarrow Int \{ prec: 33 \}}
    • \texttt{op \_*_ : Int Int \rightarrow Int \{ prec: 31 \}}
  – Defaults
    • Operators with no external arguments (standard notation, and similar) have precedence 0
    • Prefix unary operators (eg \_\_) have precedence 15
    • Otherwise, 41
Operator Attributes

• Parsing Attributes
  – Left or right associativity
    • Purely for disambiguating parsing
    • Not the same thing as equational theory associativity
    • $l$-assoc, $r$-assoc
Operator Attributes

• Why?
• Sometimes you save yourself bother:
  – The lines:
    \[ \text{eq } a + 0 = a . \]
    \[ \text{eq } 0 + a = a . \]
  – Are not necessary if + is declared with id:
    \[ \text{op } _+_{-} : \text{Nat Nat } \rightarrow \text{Nat} \ {\text{id: 0}} \]
Operator Attributes

• Why bother?
• Sometimes it’s necessary:
  – The underlying TRS will not terminate if you have:
    • eq a + b = b + a.
  – You must use:
    • op _+_ : Nat Nat -> Nat {comm}
CafeOBJ theorem prover

**Input:** clausal set.

**Output** on termination: a proof of unsatisfiability or a saturated clause set.

New added to Usable after interreduction.

Usable clauses \(\xrightarrow{\text{Selected clause}}\) Worked Off Clauses \(\xleftarrow{\text{Interreduction}}\) New clauses

Inferences \(\xleftarrow{\text{Newly generated clauses}}\)