

Tasks graph and data dependencies

Towards an object-oriented logic


Benoît Perrot <benoit@lrde.epita.fr>

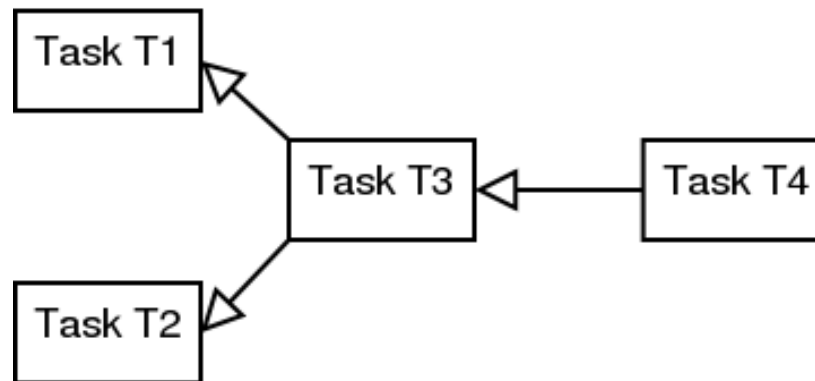
LRDE seminar, May 28, 2003



Context

- ▶ Building a dependency graph, illustrating task dependencies and implying the order of execution of modules

Module M1  — — Module M2



Common resolution

- ▷ Topological sort:
“linear ordering of all vertices of a directed graph G such that if G contains an edge (u, v) , then u appears before v ”
- ▷ Complexity:
 $O(V + E)$ (depth-first search complexity) cor (2002)

Table of Contents

Expressing dependency	1
Analogy with deduction	7
Towards an object oriented logic	12
Dependency engine	20
Extending application domain	24
Conclusion	27
References	28

Expressing dependency

Searching for a syntax to write task-dependencies:

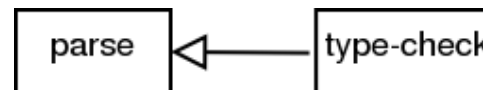
- ▷ What must absolutely be written?
- ▷ What could easily be “forgotten”, what might be inferred?
- ▷ What should be factored?

Two possibilities:

1. name-driven syntax
2. data-driven syntax

Name-driven task-dependency expression

- ▷ Assign a unique identifier to each task
- ▷ Dependency \equiv a “temporal link” between two identifiers
Example: Tiger Compiler



The task “type-check” occurs **after** “parse”

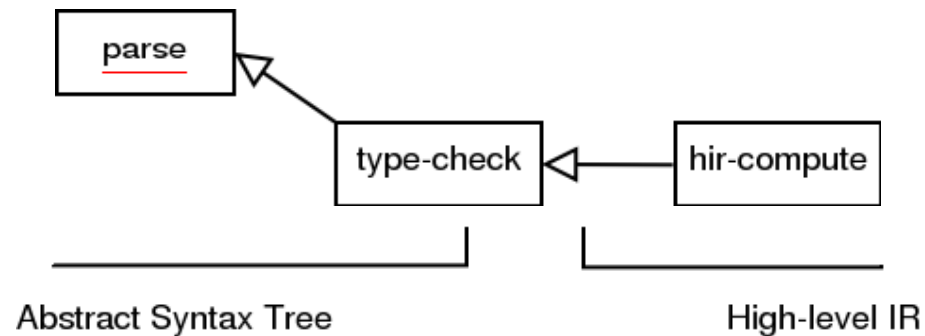
- ▷ Explicit graph description \Rightarrow Easy to implement

... And its limitations

- ▶ Low expression power: “scan-trace” is optional but flows in “parse”



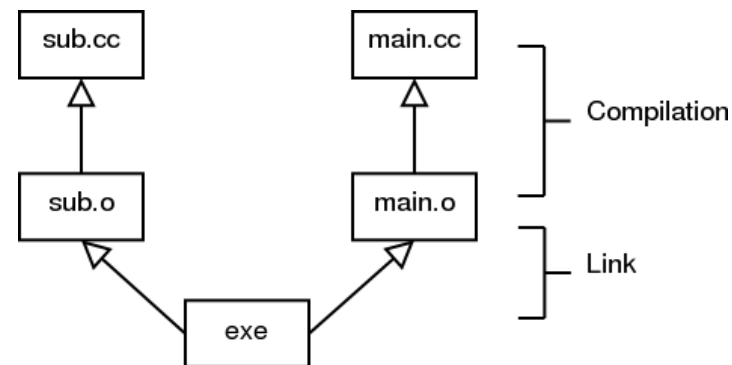
- ▶ Hide generated data streams



Data-driven task-dependency expression

- ▷ Task \equiv reactants and products (like chemistry equation)
- ▷ Considering Makefile Scheme:

```
## Compilation
main.o: main.c
    $(CXX) -c $<
sub.o: sub.c
    $(CXX) -c $<
## Link
exe: main.o sub.o
    $(CXX) main.o sub.o -o $@
```



- ▷ Tasks' implicit dependency \Rightarrow Hard to setup

... And its possibilities

▷ Good expression power

- Factored transformations (reusable scheme):

```
% .o: % .c  
$(CXX) -c $<
```

▷ Show data streams

- More intuitive ⇒ Easier to understand and to manipulate
- Allow liveness analysis

Focus the needs

- ▷ Formalize data-ruled dependencies
- ▷ Maximize expression power
 - do not limit to “make”!
- ▷ Minimize resolution engine
 - do not re-create Prolog!

Analogy with deduction

- ▷ Solving task dependencies \equiv given a theory, demonstrating a theorem
- ▷ A Makefile may be seen as a theory:

```
## Compilation
main.o: main.c
    $(CXX) -c $<
sub.o: sub.c
    $(CXX) -c $<
## Link
exe: main.o sub.o
    $(CXX) main.o sub.o -o $@
```

```
“Theory”:  
    main.c → main.o  
    sub.c  → sub.o  
main.o ^ sub.o → exe  
  
Data:  
    → main.c  
    → sub.c  
  
To prove:  
    ?  
    → exe
```

Logic summary [1/3] :: universe

- ▷ An alphabet $\Gamma = \mathcal{C} \cup \mathcal{P} \cup \mathcal{A}$ with
 - $\mathcal{C} = \{\vee, \wedge, \rightarrow, \leftrightarrow, \neg\}$: the set of connectors
 - $\mathcal{P} = \{(,)\}$: the set of parenthesis
 - \mathcal{A} : the set of atoms

- ▷ Set of well-formed formulae \mathcal{F}
 - includes \mathcal{A}
 - stable by: $u, v \mapsto (u \alpha v)$ with $\alpha \in \mathcal{C}_{bin}$
 - stable by: $u \mapsto (\neg u)$

Logic summary [2/3] :: deduction

- ▷ Write all formulae as Horn clauses: $(p_i, q_j) \in \mathcal{A}^2, \bigwedge p_i \rightarrow \bigvee q_j$
- ▷ Cut elimination:
 - Consider two clauses

$$\begin{cases} C_1 = \bigwedge p_i \rightarrow \bigvee q_j \\ C_2 = \bigwedge r_i \rightarrow \bigvee s_j \end{cases}$$

- If $\exists(a, b), p_a = s_b$ then consider the new clause

$$C' = \bigwedge_{i \neq a} p_i \bigwedge r_i \rightarrow \bigvee q_j \bigvee_{j \neq b} s_j$$

Logic summary [3/3] :: cut example

- ▷ C : Clean
- ▷ W : Wash, D : Dishes, T : Tidy

$$\frac{W \wedge D \wedge T \rightarrow C}{\rightarrow W}$$
$$D \wedge T \rightarrow C$$

Lack of abstraction

Logic theories are...

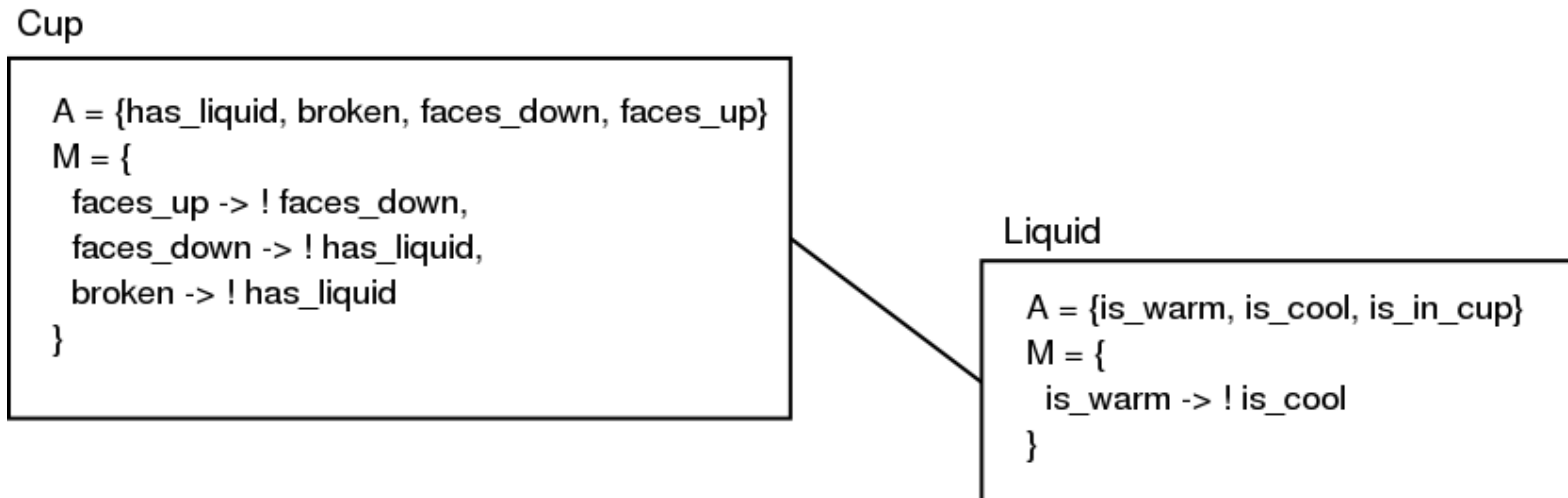
▷ Exhausting to build

▷ Difficult to reuse

⇒ Need object oriented paradigm!

Towards an object oriented logic

- ▷ A theory \equiv a graph of smaller theories Amir (2001)



Reasoning within objects

- ▷ An axiom \equiv an assignable attribute
- ▷ A clause \equiv an activation method
- ▷ Obtaining a property \equiv proving a theorem, by applying a chain of methods

... Yes, within

Run the coffee-machine !

- ▷ A class CoffeeMachine with its set of attributes = {is-plugged, is-switched-on, is-running, has-water, has-coffee}
- ▷ CoffeeMachine's theory:
is-plugged \wedge is-switched-on \rightarrow is-running
- ▷ Solving "running CoffeeMachine":

$$\begin{array}{rcl} \text{is-plugged} \wedge \text{is-switched-on} & \rightarrow & \text{is-running} \\ & \rightarrow & \text{is-plugged} \\ \hline & \text{is-switched-on} & \rightarrow \text{is-running} \end{array}$$

Reasoning between objects

- ▷ Perpetual link between objects
- ▷ Constructor \equiv converter (directed link) between objects
- ▷ Destructor \equiv converter (directed link) between an object and the “garbage”

... Yes, between

Want a coffee ?

- ▷ A class Coffee with its set of attributes = {is-warm, is-cool}
- ▷ “warm Coffee” constructor:
CoffeeMachine.{is-running \wedge has-water \wedge has-coffee} \rightarrow Coffee.is-warm
- ▷ Making “warm Coffee” from scratch

\rightarrow CoffeeMachine.is-plugged

CoffeeMachine.is-switched-on \rightarrow CoffeeMachine.is-running

CoffeeMachine.{is-running \wedge has-water \wedge has-coffee} \rightarrow Coffee.is-warm

CoffeeMachine.{is-switched-on \wedge has-water \wedge has-coffee} \rightarrow Coffee.is-warm

Formal definition

Defining a dependency scheme

- ▷ A set of classes that includes the “garbage” \perp and the “origin” \top
- ▷ A set of converters between these classes

Defining a class C

- ▷ A set of properties \mathcal{P}_C
- ▷ A set of activation methods \mathcal{M}_C
A method is a tuple of a clause, an optional identifier and a list of actions

Formal definition

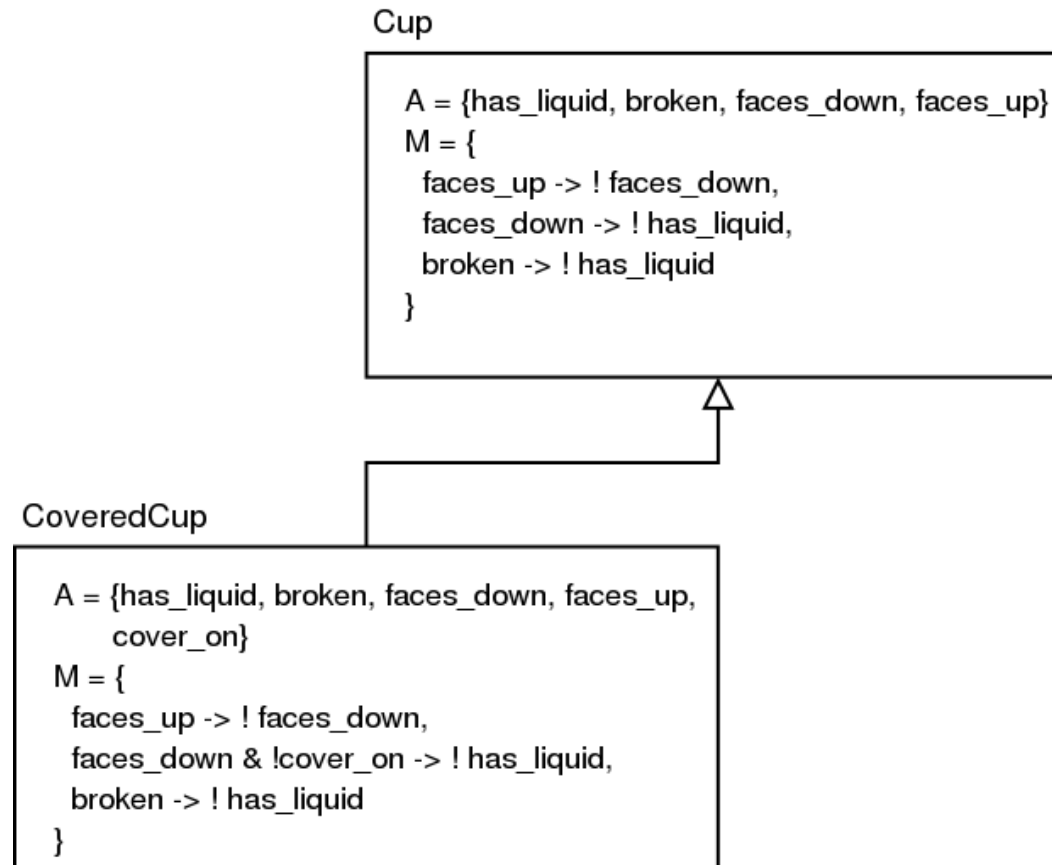
Inheritance

A subclass inherits all the properties of its superclass and may add symbols to the vocabulary

Polymorphism

A subclass inherits and may redefine or add activation methods

Considering a covered cup



Dependency engine

Using F-Logic syntax [Kifer et al. \(1995\)](#)

Clause within object (method)

$A[id @ p \ \& \ q \rightarrow r]$

Inheritance

$A :: B$ (A is a subclass of B)

Clause between objects (converter)

$id @ A[p \ \& \ q] \rightarrow B[t]$

:: for Tiger Compiler

Simple deduction system

```

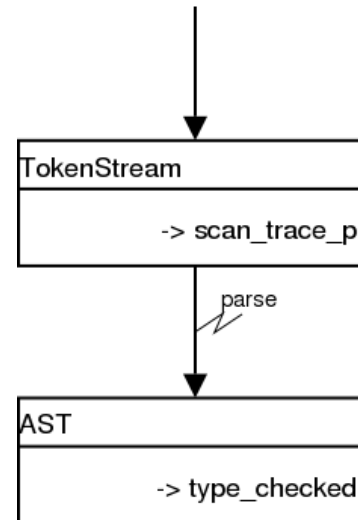
-> TokenStream
{
    echo "open_file"
}

TokenStream[scan_trace @ -> scan_trace_p]
{
    echo "scan_trace"
}

parse @ TokenStream -> AST
{
    echo "parse"
}

AST[types_check @ -> type_checked]
{
    echo "types_check"
}

```



:: for make utility

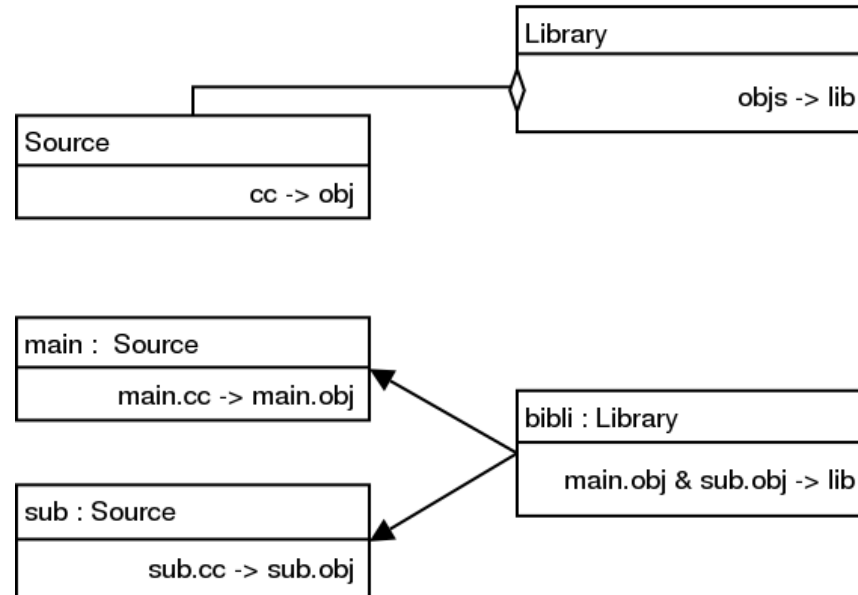
Deduction system with environment table (need variables)

```

## Source rules.
Source [cc -> obj := "$0.o"]
{
    echo "Compiling_$1_..."
    gcc -c $1 -o $2
}
Source [obj ->]
{
    rm -f $1
}

## Library rules.
Library [ objs -> lib := "$0"]
{
    ar $2 $1
    ranlib $2
}

## Project
main, sub: Source [cc := "$0.cc"]
bibli : Library [objs = (main.obj, sub.obj)]
    
```



Parallelism, ambiguities, cycles

Considering cycle

$A[\rightarrow p]$ $A[p \rightarrow]$

- ▷ Shortest path
- ▷ Never happen!

Solving ambiguities

$A[\rightarrow q]$ $A[\rightarrow r]$ $A[r \rightarrow q]$

- ▷ Shortest path
- ▷ Let the user choose!

Extending application domain

- ▷ Combination of object oriented ideas and logical knowledge representation

- ▷ Performance improvement
 - Reuse results
 - Parallelism

:: Hierarchical reasoning

A robot receives the assignment of navigating from one office to another...

▷ Robot constraints:

- Must be run and monitored in real-time
- Should reason about the behavior of other objects/agents (elevator, doors, etc.)

⇒ brute-force reasoning intractable, knowledge modeling difficult

▷ What allows object-oriented logic:

- Model environment and robot by a logic object
- Reasoning about several object concurrently

:: Compiler coercion inference

Building an `std::string` from an `int`...

- ▷ A constructor of an `std::string` is defined from a `char*` as argument
- ▷ `itoa` function allows to build a `char*` from an `int`
- ▷ Imagine a compiler that implicitly defined a constructor of an `std::string` from an `int`!

Conclusion

- ▷ Object-oriented logic
- ▷ Reduce complexity of deduction?

References

(2002). *Introduction to Algorithms*.

Amir, E. (2001). Object-oriented first-order logic.

Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages.