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
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>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressing dependency</td>
<td>1</td>
</tr>
<tr>
<td>Analogy with deduction</td>
<td>7</td>
</tr>
<tr>
<td>Towards an object oriented logic</td>
<td>12</td>
</tr>
<tr>
<td>Dependency engine</td>
<td>20</td>
</tr>
<tr>
<td>Extending application domain</td>
<td>24</td>
</tr>
<tr>
<td>Conclusion</td>
<td>27</td>
</tr>
<tr>
<td>References</td>
<td>28</td>
</tr>
</tbody>
</table>
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

\[ \text{parse} \quad \xrightarrow{} \quad \text{type-check} \]

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
Expressing dependency

... And its possibilities

▷ Good expression power
  □ Factored transformations (reusable scheme):
    
    %.o: %.c
    $(CXX) -c <$

▷ Show data streams
  □ More intuitive \(\Rightarrow\) 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` \(\rightarrow\) `main.o`
- `sub.c` \(\rightarrow\) `sub.o`
- `main.o \land sub.o` \(\rightarrow\) `exe`

Data:
- \(\rightarrow\) `main.c`
- \(\rightarrow\) `sub.c`

To prove:
- `?` \(\rightarrow\) `exe`
An alphabet $\Gamma = C \cup P \cup A$ with

- $C = \{\lor, \land, \rightarrow, \leftrightarrow, \neg\}$: the set of connectors
- $P = \{(,)\}$: the set of parenthesis
- $A$: the set of atoms

Set of well-formed formulae $F$

- includes $A$
- stable by: $u, v \mapsto (u \alpha v)$ with $\alpha \in C_{bin}$
- stable by: $u \mapsto (\neg u)$
Write all formulae as Horn clauses: \((p_i, q_j) \in \mathcal{A}^2, \land p_i \rightarrow \lor q_j\)

Cut elimination:
- Consider two clauses

\[
\begin{align*}
C_1 &= \land p_i \rightarrow \lor q_j \\
C_2 &= \land r_i \rightarrow \lor s_j
\end{align*}
\]

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

\[
C' = \land_{i \neq a} p_i \land r_i \rightarrow \lor_{j \neq b} q_j \lor s_j
\]
Logic summary [3/3] :: cut example

▷ $C$: Clean

▷ $W$: Wash, $D$: Dishes, $T$: Tidy

\[
W \wedge D \wedge T \rightarrow C
\]
\[
\rightarrow W
\]
\[
D \wedge T \rightarrow C
\]
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)

Cup

\[
A = \{ \text{has\_liquid}, \text{broken}, \text{faces\_down}, \text{faces\_up} \} \\
M = \{ \\
\quad \text{faces\_up} \rightarrow ! \text{faces\_down}, \\
\quad \text{faces\_down} \rightarrow ! \text{has\_liquid}, \\
\quad \text{broken} \rightarrow ! \text{has\_liquid} \\
\}
\]

Liquid

\[
A = \{ \text{is\_warm}, \text{is\_cool}, \text{is\_in\_cup} \} \\
M = \{ \\
\quad \text{is\_warm} \rightarrow ! \text{is\_cool} \\
\}
\]
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:
  \[ \text{is-plugged} \land \text{is-switched-on} \rightarrow \text{is-running} \]

- Solving “running CoffeeMachine”:

  \[
  \begin{align*}
  \text{is-plugged} \land \text{is-switched-on} & \rightarrow \text{is-running} \\
  \text{is-switched-on} & \rightarrow \text{is-plugged} \\
  \hline
  \text{is-switched-on} & \rightarrow \text{is-running}
  \end{align*}
  \]
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 \land has-water \land has-coffee\} \implies \text{Coffee.is-warm}

▷ Making “warm Coffee” from scratch
  \[
  \begin{align*}
  \text{CoffeeMachine.is-plugged} & \quad \text{CoffeeMachine.is-switched-on} \implies \text{CoffeeMachine.is-running} \\
  \text{CoffeeMachine.}\{\text{is-running} \land \text{has-water} \land \text{has-coffee}\} & \implies \text{Coffee.is-warm}
  \end{align*}
  \]
Towards an object oriented logic

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
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

Cup

\[ A = \{ \text{has\_liquid}, \text{broken}, \text{faces\_down}, \text{faces\_up} \} \]
\[ M = \{ \]
\[ \text{faces\_up} \rightarrow \neg \text{faces\_down}, \]
\[ \text{faces\_down} \rightarrow \neg \text{has\_liquid}, \]
\[ \text{broken} \rightarrow \neg \text{has\_liquid} \]
\[ \} \]

CoveredCup

\[ A = \{ \text{has\_liquid}, \text{broken}, \text{faces\_down}, \text{faces\_up}, \]
\[ \text{cover\_on} \} \]
\[ M = \{ \]
\[ \text{faces\_up} \rightarrow \neg \text{faces\_down}, \]
\[ \text{faces\_down} \& \neg \text{cover\_on} \rightarrow \neg \text{has\_liquid}, \]
\[ \text{broken} \rightarrow \neg \text{has\_liquid} \]
\[ \} \]
Using F-Logic syntax Kifer et al. (1995)

Clause within object (method)
A@\text{id} \land p \land q \rightarrow r

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

Clause between objects (converter)
id \land A[p \land q] \rightarrow B[t]
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"
}
```
## 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

\[
A \rightarrow p, \quad A[p \rightarrow] \\
\]

▷ Shortest path
▷ Never happen!

Solving ambiguities

\[
A \rightarrow q, \quad A \rightarrow r, \quad A[r \rightarrow q] \\
\]

▷ Shortest path
▷ Let the user choose!
Extending application domain

- Combination of object oriented ideas and logical knowledge representation
- Performance improvement
  - Reuse results
  - Parallelism
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.)

  \[ \rightarrow \text{brute-force reasoning intractable, knowledge modeling difficult} \]

- What allows object-oriented logic:
  - Model environment and robot by a logic object
  - Reasoning about several object concurrently
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

