Modular Action Language $\mathcal{ALM}$

Michael Gelfond (joint work with Daniela Inclezan)

Texas Tech University

March, 2015

The Goal

When we started our work on $\mathcal{ALM}$ Daniela and I wanted to: *design a language to support stepwise development of action theories, improve their readability, and facilitate reuse of knowledge.*

We believed that the absence of such language was one of the most serious impediments to the *creation of knowledge libraries* – an important scientific and engineering goal of KRR.

The Design Principles

In the matters of language design we were strongly influenced by work of Dijkstra, Hoare, Wirth, and McCarthy from which we extracted the following principles:

- **The main function of the language is not communication but rather *helping people to acquire ability for a richer understanding of the world.***

  When E. Dijkstra says that Lisp "assisted a number of our most gifted fellow humans in thinking previously impossible thoughts" *he clearly expresses this view.*

The Design Principles

- *Elegance*, understood as the right combination of simplicity and power is *"not a dispensable luxury but a quality that decides between success and failure"*.
- The syntax and semantics of the language should be *as simple as possible* but not simpler.
- Language should ensure that *"different components of the program correspond clearly to different components of its specification, so you can reason compositionally about it"* (C. A. R. Hoare).
- Language should support elaboration tolerant representation of knowledge.

Example of a KR problem: Monkey and Banana

A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. In the room there is also a box. The ceiling is just the right height so that a monkey standing on the box under the bananas can reach the bananas. The monkey can move around, carry other things around, climb on the box, and grasp the bananas. What is the best sequence of actions for the monkey to get the bananas?

Monkey and Banana– how to start?

I used to start representation of knowledge relevant to the problem with asking: *What are the objects of our domain?*.

After some experience I came to (now obvious) realization that the question is wrong. It works for simple problems but does not lead to general and elaboration tolerant solutions.

The right question is " *What are the SORTS of objects relevant to the domain and what is the relationship between these sorts*"?

Monkey and Banana: Sorts and Sort Hierarchy

$\mathcal{ALM}$ **assumes that most sorts of the domain are organized into an inheritance hierarchy (a DAG) and provides means for its description.**

The story explicitly mention four types of action: *move*, *carry*, *climb*, and *grasp*.

Clearly, *carry* understood as *"moving while holding an object"* and *climb* understood as *"moving on top of an elevation"* are special kinds of action *move*.

This is captured by the hierarchy on the next slide:

Monkey and Banana: the action hierarchy

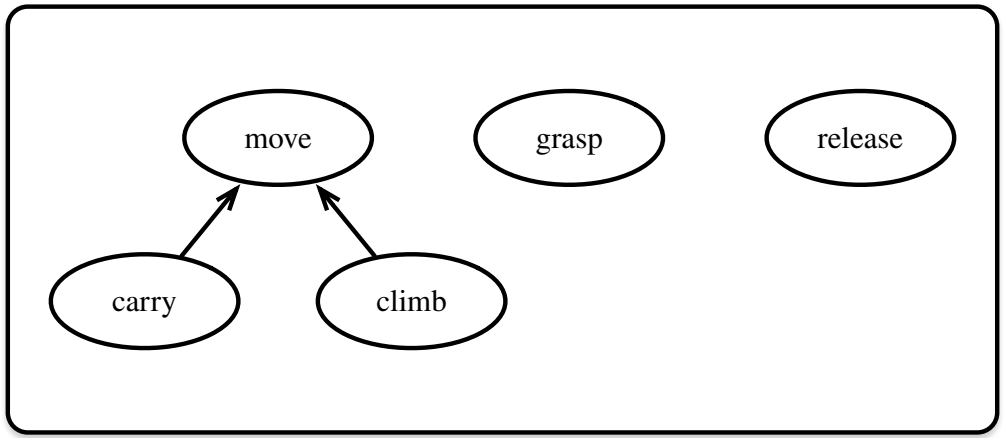


Figure : Action hierarchy for the Monkey and Banana Problem

It is often advisable to build hierarchies which contain actions together with their opposites, which explains the appearance of *release*.

$\mathcal{ALM}$ module

Now we are ready to describe the first component of our $\mathcal{ALM}$ theory – a *module* which describes actions of sort move and their properties.

Intuitively, an $\mathcal{ALM}$ *module* is a formal description of a specific piece of knowledge packaged as a unit.

Modules are used for structuring knowledge, reuse, and stepwise refinement.

They are normally organized into a module inheritance hierarchy, which is referred to as an $\mathcal{ALM}$ *theory*.

$\mathcal{ALM}$ module: Sort Declarations

Module description starts with the title, followed by the declaration of object sorts:

module moving
  sort declarations
    points, things :: universe
    agents :: things

    move :: actions
      attributes
        actor : agents
        origin : points
        dest : points

$\mathcal{ALM}$ module: Sort Declarations

Pre-defined sorts *universe* and *actions* contain all the elements of the sort hierarchy and all the domain actions respectively.

The construct :: , called *specialization*, corresponds to the hierarchy's links.

***Attributes* are (possibly partial) functions which describe intrinsic properties of objects of a given sort.**

This sort, which serves as a parameter of the attribute, is not explicitly mentioned in the declaration, i.e. the signature of actor is:

$$actor : move \rightarrow agents$$

$\mathcal{ALM}$ module: Function Declarations

In addition to attributes properties of the domain are described by functions which can be *static* or *fluent* and *basic* or *defined*. Here is an $\mathcal{ALM}$ declaration:

function declarations
statics
basic
symmetric_connectivity : booleans
transitive_connectivity : booleans
fluents
basic
connected : points × points → booleans
total loc_in : things → points

total **indicates that** loc_in **is defined for all things.**

axioms

$$\text{occurs}(M) \textbf{ causes } \text{loc\_in}(A) = D \textbf{ if } \quad \begin{array}{l} \text{instance}(M, \text{move}), \\ \text{actor}(M) = A, \\ \text{dest}(M) = D. \end{array}$$

The axiom says that *an occurrence of action* M *of sort* move *causes the move's actor to arrive at its destination.*

Note that M here is a variable and hence the axiom is more general then a dynamic causal law of $\mathcal{A}$-like languages.

$\mathcal{ALM}$ module: Axioms

As expected, $\mathcal{ALM}$ also has *state constraints*, e.g.

$$\begin{aligned} \text{connected}(X, Z) \quad \textbf{if} \quad & \text{connected}(X, Y), \\ & \text{connected}(Y, Z), \\ & \text{transitive\_connectivity}. \end{aligned}$$

and *executability axioms*, e.g.

$$\begin{aligned} \textbf{impossible} \quad \text{occurs}(M) \quad \textbf{if} \quad & \text{instance}(M, \text{move}), \\ & \text{origin}(M) = L_1, \\ & \text{dest}(M) = L_2, \\ & \neg\text{connected}(L_1, L_2). \end{aligned}$$

with a self-explanatory reading.

Semantics of an $\mathcal{ALM}$ Module: Interpretations

Similar to FOL, axioms of an $\mathcal{ALM}$ module are *uninterpreted*. Their meaning is determined by an *interpretation* of the sorted signature, Σ, of the module.

An *interpretation* of Σ consists of the universe $\mathcal{U}$ and a mapping I such that

- **For every sort c of the sort hierarchy $\mathcal{H}$ of Σ, $I(c)$ is a non-empty subset of $\mathcal{U}$ and for every object constant o, $I(o) \in \mathcal{U}$.**
- **For every function symbol $f : c_1 \times \cdots \times c_n \rightarrow c$, $I(f)$ is a function from $I(c_1) \times \cdots \times I(c_n)$ into $I(c)$.**
- **Special functions like is_a, $link$, etc. are interpreted in accordance with their meaning in the hierarchy. Similarly for boolean function dom_f determining the domain of f.**

Semantics of an $\mathcal{ALM}$ Module: Interpretations

An interpretation I is divided into two parts:

- ***fluent interpretation*** **consisting of the universe of I and the restriction of I on the sets of fluents, and**
- ***static interpretation*** **consisting of the same universe and the restriction of I on the remaining elements of the signature.**

Example of an Interpretation

- **Static Interpretation**

$I_1(points) = \{paris, london\}$

$I_1(agents) = \{bob\}$

$I_1(actions) = I_1(move) = \{m\}$

$I_1(actor)(m) = bob$

$I_1(dest)(m) = london$

$I_1(origin)(m) = paris$

$I_1(things) = I_1(agents)$

$I_1(universe) = I_1(things) \cup I_1(points) \cup I_1(actions)$

- **Fluent Interpretation**

$I_1(loc\_in)(bob) = paris$

$I_1(connected)(paris, london) = true$

etc.

Pre-Models of $\mathcal{ALM}$ theory

Let T be an action theory with signature Σ and U be a collection of strings in some fixed alphabet.

By Σ_U we denote the signature obtained from Σ by expanding its set of object constants by elements of U.

A static interpretation M of Σ_U is called a *pre-model* of T (with the universe U) if

- $M(universe) = U$ **and**
- **for every object constant o of Σ_U that is not an object constant of Σ , $M(o) = o$.**

Models of $\mathcal{ALM}$ theory

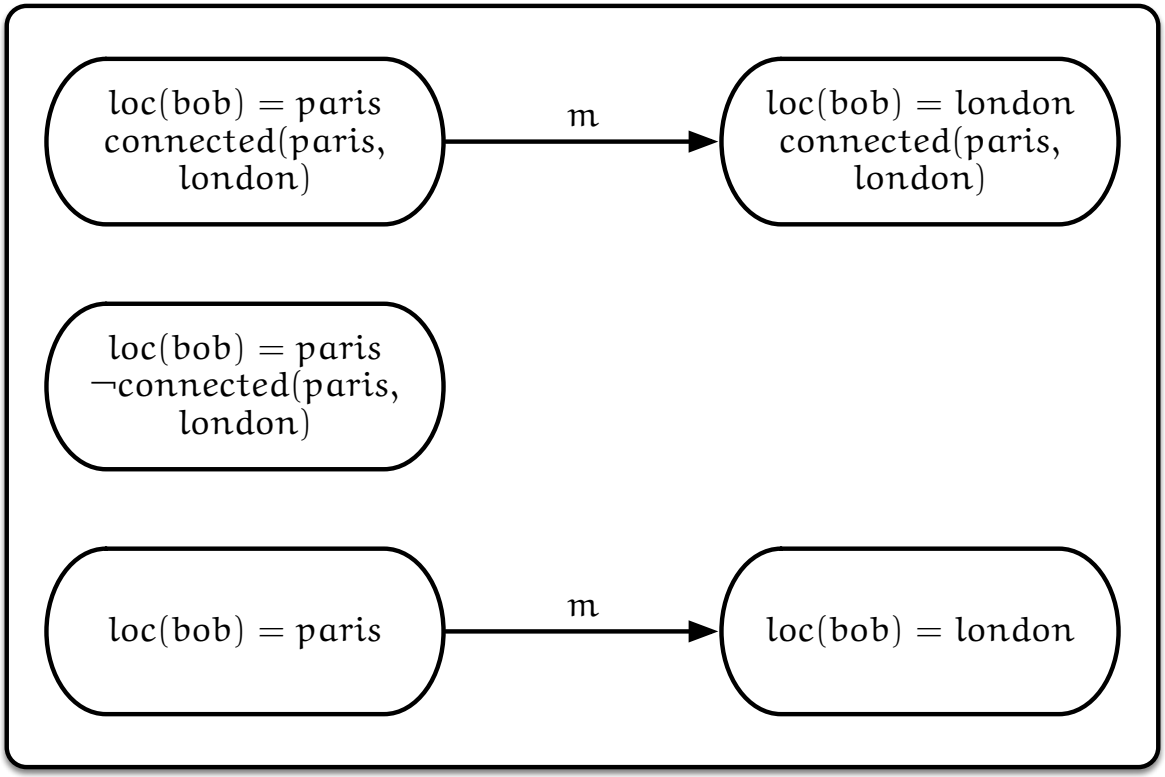
A pre-model M defines a *model* of an $\mathcal{ALM}$ theory T – a transition diagram τ(M) containing all the possible trajectories of the dynamic system described by T.

A *state* of τ(M) is an interpretation I such that

- **static part of I is M;**
- **I satisfy state constraints and definitions of T.**

***Transitions* of τ(M) are defined by the translation of M and the axioms of T into an ASP program.**

Example of a Model



Note that in the last transition, $connected(paris, london)$ is undefined.

Specifying models of an $\mathcal{ALM}$ theory in $\mathcal{ALM}$

Usually, a knowledge engineer is interested in one particular model of his domain characterized by the domain's sorts hierarchy and values of statics.

This is done in the second part of $\mathcal{ALM}$'s system description called *structure*.

So syntactically we have

system description ⟨name⟩
  theory⟨name⟩
    module ⟨name⟩
      ⟨module body⟩
    ...
  structure ⟨name⟩
    ⟨structure body⟩

Structure of $\mathcal{ALM}$

structure test
  instances
    bob **in** agents
    paris, london **in** points
    m **in** move
      actor = bob
      origin = paris
      dest = london

The module *moving* can be tested by using it for reasoning about different transition diagrams formed by the module in conjunctions with various structures.

The testing is done by translating the corresponding system descriptions into logic programs.

Axiomatization of *carry*: Sorts

Now we move toward axiomatization of the second component of our specification – an action *carry*, understood as *move while holding*.

We introduce a new subsort of things:

sort declarations

carriables :: things

and action sort *carry* viewed as a subsort of *move*

carry :: move

attributes

carried_object : carriables

Note that, since carry is defined as a special case of move, it automatically inherits the attributes of move; hence those attributes do not have to be repeated in the declaration of carry.

Axiomatization of *carry*: Fluents and Axioms

function declarations

fluents

basic

total $holding : agents \times things \rightarrow booleans$

axioms

$loc\_in(C) = P$ **if** $holding(T, C)$,
$loc\_in(T) = P$.

impossible $occurs(X)$ **if** $instance(X, carry)$,
$actor(X) = A$,
$carried\_object(X) = C$,
$\neg holding(A, C)$.

etc.

Combining the components

There are two ways to combine $\mathcal{ALM}$'s representation of move and carry:

- **make the representation of carry to be a part of module *moving*;**
- **put our formalization of carry in a separate module and make it a submodule of *moving*.**

The latter can be done by using a construct *depends on* as follows:

module carrying_things
    depends on moving
    ⟨sort declarations and axioms⟩
    ⟨from two previous slides⟩

How many modules?

To serve their purpose modules should be easily comprehensible, testable, and reusable.

This puts serious limitation on the module size.

On another hand, a module inherits knowledge represented in its ancestors. So for the sake of comprehension the hierarchy of modules shall not be too deep.

This means that, when deciding how many actions to describe in one module, one should consider balancing the size of the module with the depth of the module dependency hierarchy.

Module *carrying_ things*

We are still learning techniques for achieving such balance. But here is an example based on our current experience.

Actions *carry*, *grasp* and *release* understood as *move while holding*, *take and hold*, and *stop holding* respectively can be put in the same module, *carrying_ things*, since

- **The definitions of these actions share a fluent *holding* and sorts *things*, *agents* and *points* and**
- **A things-carrying agent usually also executes actions *grasp* and *release*.**

Axiomatizing *grasp* and *release*: sort declarations

Module *carrying_ things* will be expanded by formalization of actions *grasp* and *release*.

grasp :: actions
  attributes
    grasper : agents
    grasped_thing : things

release :: actions
  attributes
    releaser : agents
    released_thing : things

Axiomatizing *grasping*: function declarations

Function *can_reach* will be needed as a precondition for the executability of *grasp*.

It will be defined in terms of locations of things.

defined

$$can\_reach : agents \times things \rightarrow booleans$$

Other sorts and functions will be inherited from module *moving*.

Axiomatizing *grasping*: Axioms

We'll also expand axioms of *moving* by:

$$\text{occurs}(A) \textbf{ causes } \text{holding}(X, Y) \textbf{ if } \text{instance}(A, \text{grasp}), \text{grasper}(A) = X, \text{grasped\_thing}(A) = Y.$$

$$\neg\text{holding}(X, Y_2) \textbf{ if } \text{holding}(X, Y_1), Y_1 \neq Y_2.$$

$$\text{can\_reach}(M, O) \textbf{ if } \text{loc\_in}(M) = \text{loc\_in}(O).$$

$$\textbf{impossible } \text{occurs}(A) \textbf{ if } \text{instance}(A, \text{grasp}), \text{grasper}(A) = X, \text{grasped\_thing}(A) = Y, \text{holding}(X, Y).$$

etc.

Module for problem specific knowledge

Modules we have built so far for Monkey and Banana problem (together with a module *climbing* which can be built in a similar way) are rather general and one can easily imagine them to be stored in the library for further reuse.

The next module, *main*, contains specific information needed for the problem solution.

Among other things it defines different sorts of points and their connectivity, specifies which position located under the banana and that the banana is reachable from the top of the box located at this position, etc.

Points and Connectivity

The problem contains points located on the floor, which are all connected with each other and a point on the ceiling on which banana is located.

In addition, it seems to contain one extra point, *top(box)*, which travels together with the box.

Constants of this type, denoted by *top(E)* where E is an *elevation*, are defined in module *climbing*.

We assume that *top(box)* is connected with the location of the box.

Module *main*

In module *main* this is expressed as

module main
  depends on carrying_things, climbing

sort declarations
  floor_points, ceiling_points, movable_points :: points

object constants
  monkey : agents
  box : carriables, elevations
  banana : carriables

Module *main*: function declarations and axioms

function declarations

statics

basic $under : floor\_points \times things \rightarrow booleans$

axioms

$$can\_reach(monkey, banana) \quad \textbf{if} \quad loc\_in(box) = P,\ under(P, banana),\ loc\_in(monkey) = top(box).$$

Module *main*: function declarations and axioms

$$\begin{aligned}
\text{connected}(\text{top}(\text{box}), P) \quad &\textbf{if} \quad \text{loc_in}(\text{box}) = P, \\
&\quad\ \ \text{instance}(P, \text{floor_points}).\\
\neg\text{connected}(\text{top}(\text{box}), P) \quad &\textbf{if} \quad \text{loc_in}(\text{box}) \neq P, \\
&\quad\ \ \text{instance}(P, \text{floor_points}).\\
\text{connected}(P_1, P_2) \quad &\textbf{if} \quad \text{instance}(P_1, \text{floor_points}), \\
&\quad\ \ \text{instance}(P_2, \text{floor_points}).\\
\neg\text{connected}(P_1, P_2) \quad &\textbf{if} \quad \text{instance}(P_1, \text{ceiling_points}), \\
&\quad\ \ \text{instance}(P_2, \text{points}), \\
&\quad\ \ P_1 \neq P_2.
\end{aligned}$$

The Structure

Now we define the remaining instances of sorts and assign values to statics.

structure monkey_and_banana
  instances
    under_banana, init_monkey, init_box **in** floor_points

    init_banana **in** ceiling_points

    top(box) **in** movable_points

    move(P) **in** move **where** instance(P, points)
      actor = monkey
      dest = P

The Structure

carry(box, P) **in** carry **where** instance(P, floor_points)
  actor = monkey
  carried_object = box
  dest = P

grasp(O) **in** grasp **where** instance(O, carriables)
  grasper = monkey
  grasped_thing = O

climb(box) **in** climb
  actor = monkey
  elevation = box

values of statics
  under(under_banana, banana).
  symmetric_connectivity.
  ¬transitive_connectivity.

Finding a Plan

It is not difficult to check that our system description for Monkey and Banana problem defines a transition diagram τ that contains the path which starts with the initial state of our problem and is generated by actions $move(initial\_box)$, $grasp(box)$, $carry(box, under\_banana)$, $release(box)$, $climb(box)$, $grasp(banana)$.

The final state of this path will contain a fluent $holding(monkey, banana)$.

This sequence of actions can be found in the usual way by translating the system description, the initial situation and the goal into a logic program and using ASP planning techniques.

Conclusion

This concludes our introduction to $\mathcal{ALM}$. *Did we achieve our goal?*

The jury is still out. In our view $\mathcal{ALM}$ is the best modular action language to date.

- **It allowed us to structure and reuse knowledge, to create library modules, to define actions and other sorts in terms of their supersorts, to do stepwise refinement, etc.**
- **It also has fairly natural and transparent syntax and semantics (even though its complexity is slightly increased by the necessity to cover partial functions and sorted signatures.)**
- **Most importantly it allowed us to *"think new thoughts"* – our KR styles changed significantly as the result of this work.**

Conclusion

However, we need to

- **Implement the language and gain much more experience in its use for KRR. This will allow to see if the language can be simplified, to test our design decisions and to check that the language does not contain hidden traps for a user.**
- **We need to develop solid mathematical theory of $\mathcal{ALM}$. It will be especially nice to find sound and complete inference system for the $\mathcal{ALM}$ consequence relation and to investigate properties of $\mathcal{ALM}$ theories which remain unchanged after theories are expanded by adding new modules.**