(De)constructing the Frame

Marcus Kracht
Fakultät LiLi
Universität Bielefeld
Postfach 10 01 31
33501 Bielefeld
marcus.kracht@uni-bielefeld.de

§1. Constructing the Frame

When interpreting a modal language, we need to construct the frame and the objects out of “what is there”. There can be several ways to do this, leading ultimately to different logics.

Specifically, I shall take a look at dynamic logic and its semantics.

§2. Necessity of Identity

In Kripke-style modal logic:

(1) $x = y \rightarrow \Box(x = y)$

Worlds are built “around” objects.

This idea has been challenged many times over (Kripke, Bressan, Lewis, Priest, Schurz, Belnap & Müller).

There are several tracks:

1. Go completely worldbound (Lewis) using objects and counterparts
2. Consider the value of variables uniformly as individual concepts (Bressan)
3. Distinguish artefacts from substances (Fine)

§3. Counterpart Frames

Counterpart theory allows individuals to have counterparts in this world. This world we are in is possible in several ways, depending on how a counterpart of the objects gets chosen.

This deconnects possibility and object constitution. Consider a single world w with $w\,R\,w$, $D(w) = \{a, b\}$ and $C = \{\langle a, a\rangle, \langle a, b\rangle, \langle b, a\rangle, \langle b, b\rangle\}$. Then $\Diamond(x = y) \rightarrow \Box(x = y)$ fails even though there is only one alternative world.

§4. Coherence Frames

Coherence frames overcome this weakness.

(2) $\mathfrak{C} = \langle W, R, U, T, \tau \rangle$

where

- W is the set of worlds;
- $R \subseteq W \times W$ the accessibility relation;
- U a set of objects;
- T a set of things (world bound individuals);
- $\tau : U \times W \to T$ the trace function.

$\mathfrak{M} := \langle \mathfrak{C}, \mathcal{I} \rangle$, $\mathcal{I} = (\mathcal{I}_w)_{w \in W}$ interpreting the functions and relations (separately for each world).

(3)
$$\begin{aligned} \langle \mathfrak{M}, \beta, w \rangle \vDash P(t_1, \cdots, t_n) &\Leftrightarrow \langle t_1^\beta, \cdots, t_n^\beta \rangle \in \mathcal{I}_w(P) \\ \langle \mathfrak{M}, \beta, w \rangle \vDash x = y &\Leftrightarrow \tau(\beta(x), w) = \tau(\beta(y), w) \end{aligned}$$

§5. Problem

Objects are no longer uniquely determined. For analytic purposes, we need to

- determine the structure of the frame (worlds and accessibility); and
- construct the objects from their traces.

These two

- are not independent;
- come down to a theoretical commitment.

§6. Dynamic Logic I

Dynamic logic: reasoning about computations.

Var $:= \{x_i : i \in \omega\}$ a set of variables. A *state* is a function $\sigma :$ Var $\to \mathbb{Z}$. There are primitive functions $(+, -, \times)$, and relations $(=, <)$. A *program* is

- an assignment $x_i := t$, t a term; or
- $\pi; \rho$, $\pi \cup \rho$, π^* (π and ρ programs), or
- $\varphi?$, φ a formula.

A *proposition* is

- $P(t_1, \cdots, t_n)$, P an n-ary predicate;
- $\neg\varphi$, $\varphi \wedge \chi$ (φ, χ propositions);
- $[\pi]\varphi$ (π program, φ proposition).

§7. Dynamic Logic II

Programs are interpreted as relations between states. $\sigma R_\pi \sigma'$ if there is an execution of π that leads from σ to σ'. Eg if $\pi := x_0 := x_1 + x_0$ then $\sigma R_\pi \sigma'$ iff

1. $\sigma'(y) = \sigma(y), y \neq x_0$;
2. $\sigma'(x_0) := \sigma(x_0) + \sigma(x_1)$.

Further: $R_{\pi;\rho} = R_\pi \circ R_\rho$, $R_{\pi\cup\rho} = R_\pi \cup R_\sigma$, $R_{\pi^*} = (R_\pi)^*$, and

(4) $R_{\varphi?} = \{\langle\sigma, \sigma\rangle : \varphi \text{ is true in } \sigma\}$

§8. Nondeterminism

The semantics is not deterministic.

- $x_1 := x_0 + x_1 \cup x_0 := x_0 + x_1$
- $(x_1 := x_1 + 1)^*$

More indeterminism: random assignment $x_i :=?$ (“assign any value to x_i”). Effectively, $[x_i :=?]\varphi$ is equivalent to $(\forall x_i)\varphi$.

§9. Necessity of Identity

In DL, $x = y \to [\alpha]x = y$ does *not* generally hold.

(5) $\quad \neg(x_1 = 0) \to (x_0 = x_1 \to [x_0 := x_0 + x_1]\neg(x_0 = x_1))$

Thus we need to distinguish between what variables refer to (objects) and their values in a state (traces).

$x = y$ is true only if they have the same value. They need not be identical in terms of location. Values can be reassigned.

§10. DL and First-Order Logic

The computational behaviour is fixed by the first-order interpretation of the functions and relations! However, first-order logic is not expressive enough (lack of transitive closure).

§11. The Physical Layer(s)

1. A computer is a physical object which consists of electrical circuits. We may use physics /electrical engineering to describe its behaviour.
2. Typically, we consider only the logical behaviour: certain parts are identified as eg memory locations, and we acknowledge two abstract states “1” and “0”.
3. On top of the logical structure, various layers of abstraction are considered, with special programming languages to deal with them (machine language, assembler, C, Python, Fortran, OCaml).

§12. The Logical Layer

An actual computer consists of several parts (memory, arithmetic and logical unit, program memory, I/O devices etc.). Memory consists in

- locations, which can be addressed.
- Values stored in the locations.

The machine executes an eternal loop:

1. Fetch the next instruction
2. Decode the instruction
3. Fetch the operands (if any)
4. Execute the instruction

This exection is *deterministic* (modulo the input).

§13. The Logical Layer II

- The memory address is a number. Identical numbers point to the same physical location.
- Locations can store a fixed number of bytes.
- Basic computation is in terms of this size. (In my implementation of OCaml, integers range from -2^{63} to $2^{63} - 1$. Addition is as in $\mathbb{Z}/2^{64}\mathbb{Z}$.)

§14. DL at the Logical Layer?

- One may analyse the logical layer using DL. This allows to use variables for physical memory locations. However, DL is quite unlike machine language or assembler.
- DL uses terms, tests and logical program constructs reminiscent of higher programming, but lacks the ability to define subroutines.
- DL does know binding effects of predicate logic: it is possible to bind a variable again inside a program, which makes it impossible to think of DL variable names as coding the address.

$$(\forall x)(\exists y)(P(y,x) \wedge (\forall x)(P(y,x) \rightarrow x < y))$$

or, in DL:

$$[x_0 :=?]\langle x_1 :=?\rangle P(x_1,x_0) \wedge [x_0 :=?](P(x_1,x_0) \rightarrow x_0 < x_1))$$

§15. Variables and Objects

"*x*" is not a pointer to a physical location. It becomes that during execution of a program. Hence, the value of *x* may change because

1. *x* gets connected to another location;
2. the value stored in the location of *x* is changed.

"*x*" plays the role of the individual concept, while the memory locations are the individuals.

NB The linking between variables and locations must be tabulated. In reality, variables denote stacks of locations.

§16. Compilation

The linkage between variable name and physical location is organised by the compiler. The compiler does the storage allocation and produces also the runtime environment (or execution stack).

Call-by-Name When a procedure is called, the procedure code is copied with the parameters of the function call inserted.

Call-by-Value When a procedure is called, the values for its parameters are passed on to the called procedure.

Call-by-Address When a procedure is called, the locations for the parameters are passed on to the called procedure.

The coherence of the “individual concept” denoted by a variable is effected by the execution stack.

§17. An example

$x_0 := 1; x_1 := 1;$
while $x_1 < 6$ do
$x_0 := x_0 * x_1; x_1 := x_1 + 1$
done

Cycle:	0	1	2	3	4	5
x_0	1	1	2	6	24	120
x_1	1	2	3	4	5	6

§18. Lifespan

In functional programming, no reassigning is possible. To assign a new value, the variable needs to be thrown away. "$i := i + 1$" can be effected like this:

1. Create j.
2. $j := i$.
3. Discard i.
4. Create i.
5. $i := j + 1$.

§19. Indeterminism Revisited

A physical computer is a *deterministic* finite state automaton. We can approximate the memory by a function $\mu : R \to 2^8$ from registers to bytes (microstate). The relation between a (macro)state σ and a microstate μ is complicated.

1. numbers are larger than one byte;
2. addresses of register cells are physically given, while the variable names are not uniquely assignable to some such addresses.
3. some registers may be used internally only, that is, do not count towards the macrostate. (Indeed, some registers are not even available for the programmer.)

§20. Dynamic Logic

Consider a function calculating a random number. Only if the values in some registers are hidden can this be a truly random function.

> Depending on how the microstates are mapped onto macro-states we may consider the computer as a deterministic or as a nondeterministic device.

NB The programs $\pi \cup \rho$ or π^* are typically nondeterministic (in theory). Practical implementations often turn out to be deterministisc, though.

§21. Thank you!