MCDPL tutorial

MCDPL basics

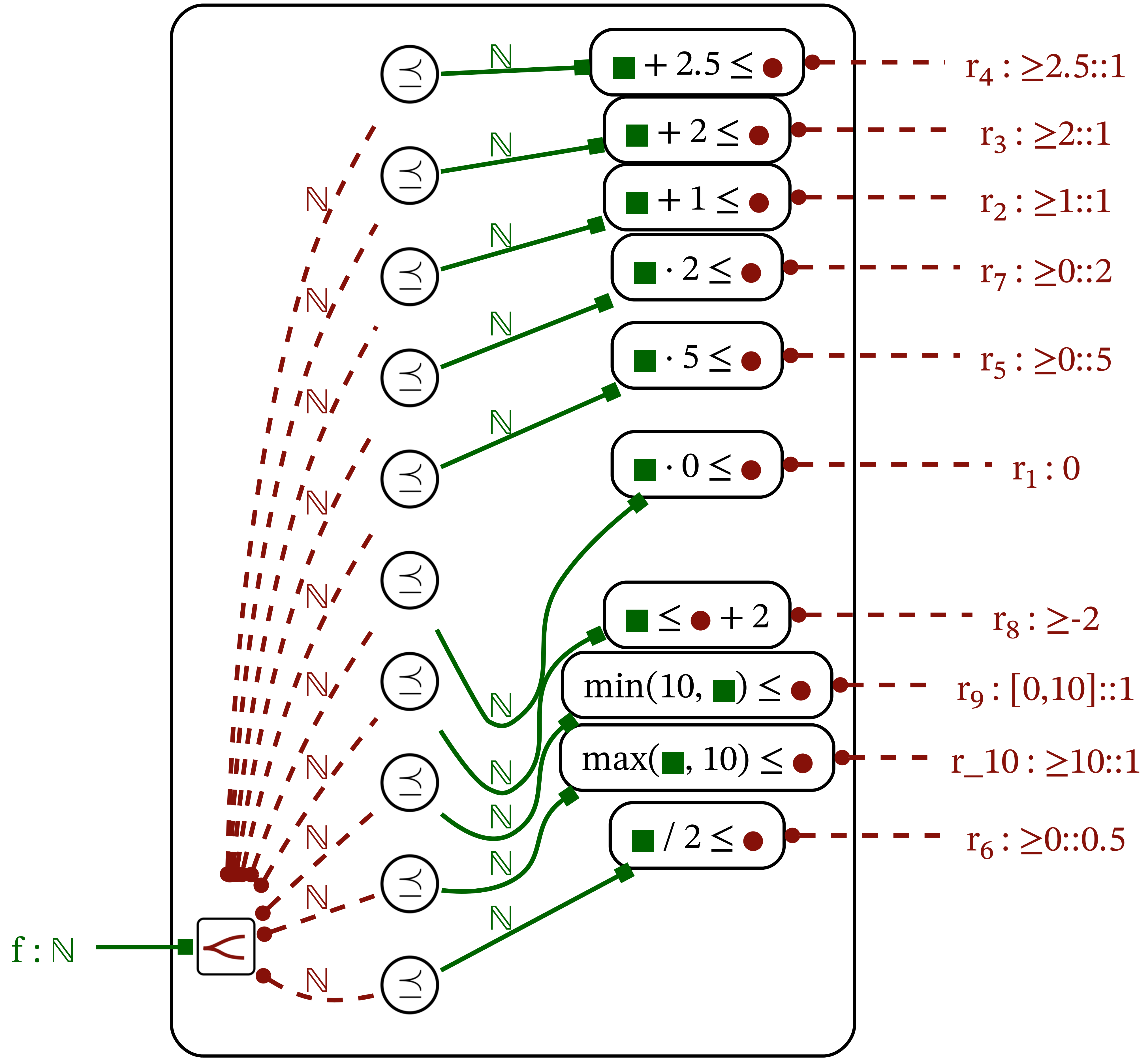
- MCDPL is a **modeling language** for **monotone co-design problems.**
- All **"primitive values"** belong to **partially ordered sets** (posets).
- MCDPL allows to define **monotone relations** between variables.
- Models can be **composed** according to **arbitrary graphs.**
- Models can be **composed** hierarchically.
- **Templates** ("diagrams with holes") allows "operadic composition".

From the programming language perspective

- **MCDPL is not meant to be a programming language.**

- The graph obtained by the interpreter is **not a computation graph.**
  - The nodes in the graph are **relations rather than functions.**

- **When choosing a query**, we do create from the graph of relations a computation graph: **a streaming algorithm** that iteratively computes the solutions.



From the optimization perspective

- **An MCDP is not an optimization problem;** rather, it roughly corresponds to just the constraints of an optimization problem.

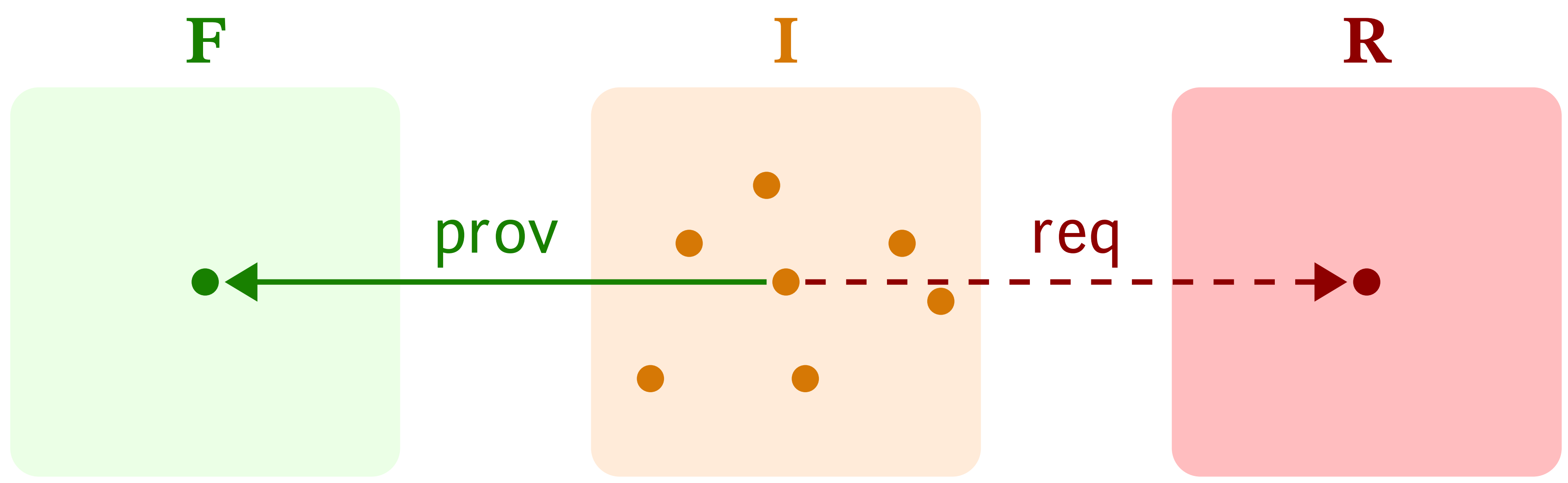
- In co-design, there are various optimization problems based on the same model, which we call **queries**.

- There is a **functorial relation** between the category of design proglems and **various categories of queries**.
  - This makes the solution "compositional".

- Our research effort so far has been focused on **arbitrary posets and monotone relations**; not much work went into special techniques for special types of problems (monotone + linear, monotone + convex, etc.)

Defining Design Problems

- We need to define the **posets** of **functionalities**, **implementations**, and **requirements**.
- Moreover, we need to define the **req** and **prov** maps.



- Two ways to do this in MCDPL:
  - By using **catalogues**, which describe these posets and functions **explicitly**.
  - By defining **MCDPs**, in which you describe these posets and functions **implicitly, as a set of monotone constraints.**

Catalogues

- The simplest way to think co-design models is using catalogues.
- These simply **enumerate the relation between functionality and requirements.**

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    # records go here
}
```

Definition of the interface

Catalogues

- The simplest way to think co-design models is using catalogues.
- These simply **enumerate the relation between functionality and requirements.**

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    # records go here
}
```

Definition of the interface

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    10 W ⟷ 10 s
    20 W ⟷ 20 s
}
```

Enumerate some elements of the relation.

We then use the **monotone closure.**

Catalogues

- The simplest way to think co-design models is using catalogues.
- These simply **enumerate the relation between functionality and requirements.**

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    # records go here
}
```

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    10 W ⟷ 10 s
    20 W ⟷ 20 s
}
```

“relations”

Anonymous implementations

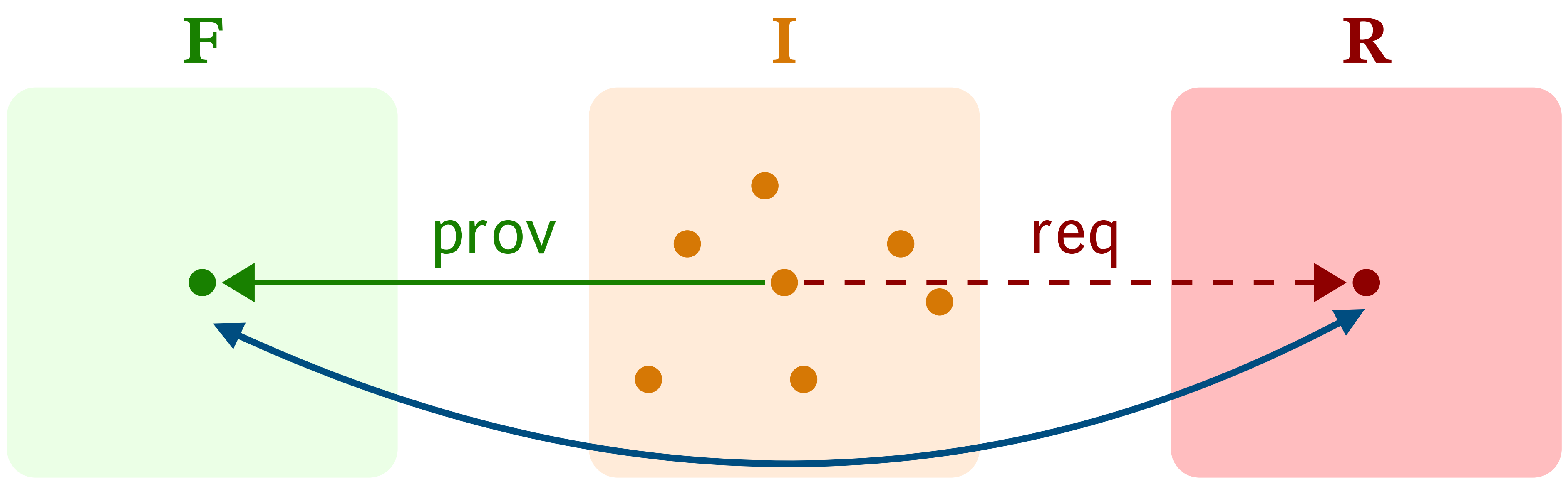
```
catalogue {
    provides f1 [W]
    requires r1 [s]

    10 W ↤ imp1 ⟼ 10 s
    20 W ↤ imp2 ⟼ 20 s
}
```

“decorated relations”

Named implementations

Catalogues

- The simplest way to think co-design models is using catalogues.
- These simply **enumerate the relation between functionality and requirements.**



feasibility relation

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    10 W ⟷ 10 s
    20 W ⟷ 20 s
}
```

“relations”

Anonymous implementations

```
catalogue {
    provides f1 [W]
    requires r1 [s]

    10 W ↤ imp1 ⟼ 10 s
    20 W ↤ imp2 ⟼ 20 s
}
```

“decorated relations”

Named implementations

Catalogues

- The simplest way to think co-design models is using catalogues.
- These simply **enumerate the relation between functionality and requirements.**

- Note that **in each row you have to use units;**
  which can be different from the interface units.

```
catalogue {
    provides distance [m]
    requires duration [s]
    5 miles ⟷ 10 hours
}
```

automatic internal conversion

- **In general, use proper units everywhere.**

Catalogues

- The simplest way to think co-design models is using catalogues.
- These simply **enumerate the relation between functionality and requirements.**

- **For multiple functionalities/requirements**, use commas betweeen numbers.

```
catalogue {
    provides f1 [W]
    provides f2 [m]
    requires r1 [s]
    requires r2 [s]

    5 W, 5 m ↤ imp1 ↦ 10 s, 10 s
}
```

- In the special cases of no functionalities/requirements, use empty tuples.

```
catalogue {
    requires r1 [s]
    ⟨⟩ ↤ imp1 ↦ 10 s
}
```

```
catalogue {
    provides f1 [s]
    5s ↤ imp1 ↦ ⟨⟩
}
```

Thinking in relations: True and False

- The **empty catalogue** has no functionalities and no requirements.

```
catalogue {}
```

- Nothing is asked but there is no way to do it! We can call it "**false**".
- Dually, this is "**true**": there is a way to provide nothing from nothing.

```
catalogue {
    ⟨⟩ ⟷ ⟨⟩
}
```

- We can make a catalogue **even more true:**

```
catalogue {
    ⟨⟩ ↤ even    ⟼ ⟨⟩
    ⟨⟩ ↤ more    ⟼ ⟨⟩
    ⟨⟩ ↤ ways    ⟼ ⟨⟩
    ⟨⟩ ↤ to      ⟼ ⟨⟩
    ⟨⟩ ↤ provide ⟼ ⟨⟩
    ⟨⟩ ↤ nothing ⟼ ⟨⟩
}
```

- There is **only one function $\mathbf{1} \to \mathbf{1}$**; there are **exactly 2 relations $\mathbf{1} \to \mathbf{1}$**; there are **infinite more "DPIs" $\mathbf{1} \nrightarrow \mathbf{1}$**.

Solution cardinality

- This is a basic example that shows that the number of minimal solutions is not monotone in the functionality required.

```
catalogue {
    provides capacity [J]
    requires mass [g]
    requires cost [USD]

    500 kWh ↤ model1 ⟼ 100 g, 10 USD
    600 kWh ↤ model2 ⟼ 200 g, 200 USD
    600 kWh ↤ model3 ⟼ 250 g, 150 USD
    700 kWh ↤ model4 ⟼ 400 g, 400 USD
}
```

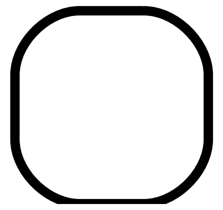
Functionality required	Optimal implementation(s)	Minimal resources needed
0 kWh $\leq f \leq$ 500 kWh	model1	⟨100 g, 10 USD⟩
500 kWh $< f \leq$ 600 kWH	model2 or model3	⟨200 g, 200 USD⟩ or ⟨250 g, 150 USD⟩
600 kWh $< f \leq$ 700 kWH	model4	⟨400 g, 400 USD⟩
700 kWh $< f \leq$ Top kWh	(unfeasible)	$\emptyset$

Monotone Co-Design Problems

- The construct **mcdp {}** allows defining **compositional** problems, hierarchically.
- The building blocks:
  - **catalogues**
  - **numerical operations** (+, /, -, pow, ...) **interpreted as monotone relations**
  - **other MCDPs** (hierarchical composition)

- This is the simplest MCDP:

```
mcdp {

}
```



- This is an MCDP with no functionalities, requirements, or constraints.
- Because the intersection of no conditions is true ($\wedge\varnothing = \top$), this is **True.**

```
mcdp {

}
```

=

```
catalogue {
    ⟨⟩ ⟷ ⟨⟩
}
```

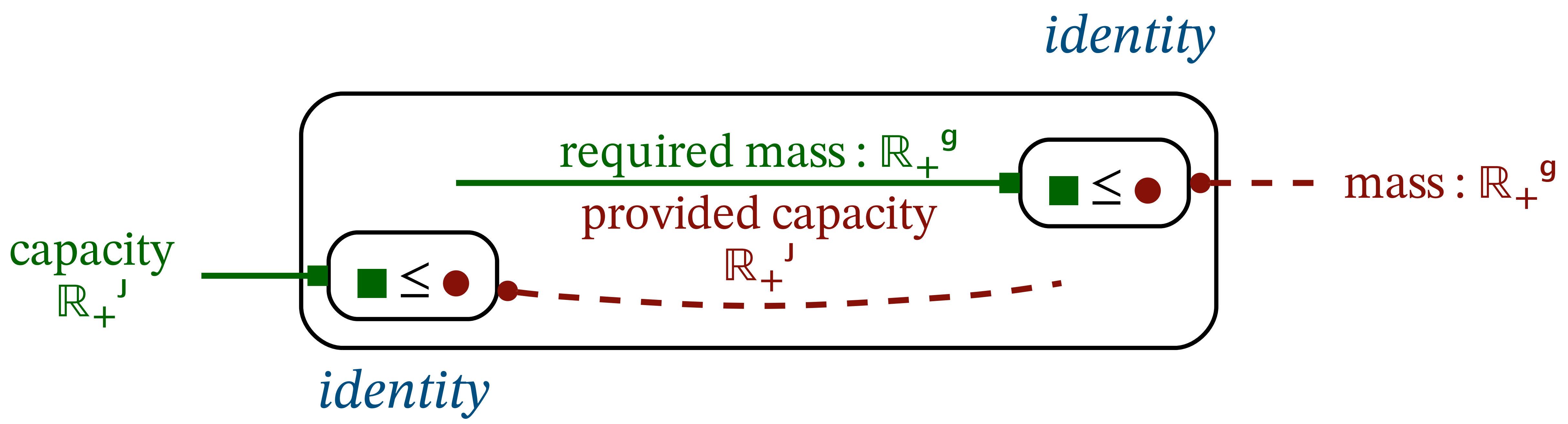
- How to describe **False**?

Defining MCDPs

- Declare functionalities and requirements using **provides** and **requires** **clauses.**

```
mcdp {
    provides capacity [J]
    requires mass [g]
}
```

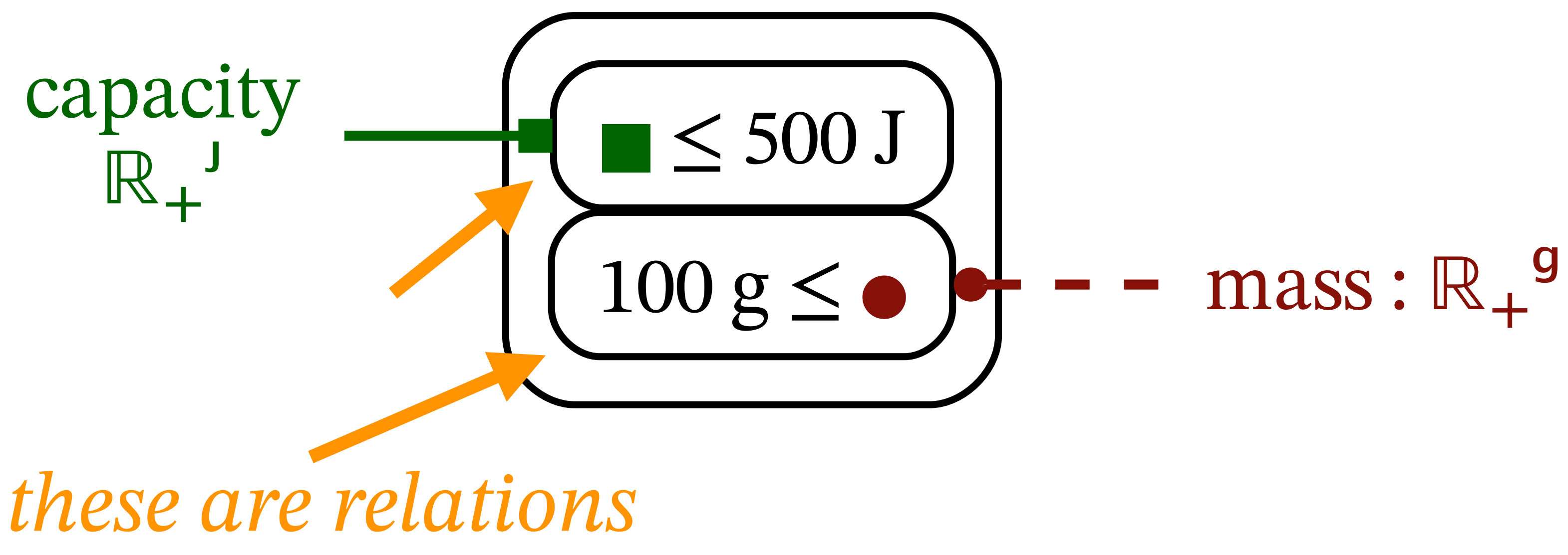
- Note that **this is not a complete model** because we did not define any constraint on functionalities/requirements.
- The UI representation will show that:
  - there are some “hanging threads”
  - if the model provides capacity, then provided capacity is a <u>requirement</u> from inside the model
  - if the model requires mass, then required mass is a <u>functionality</u> from inside the model



Using constants

- The simplest non-trivial complete model: put **constant bounds**.

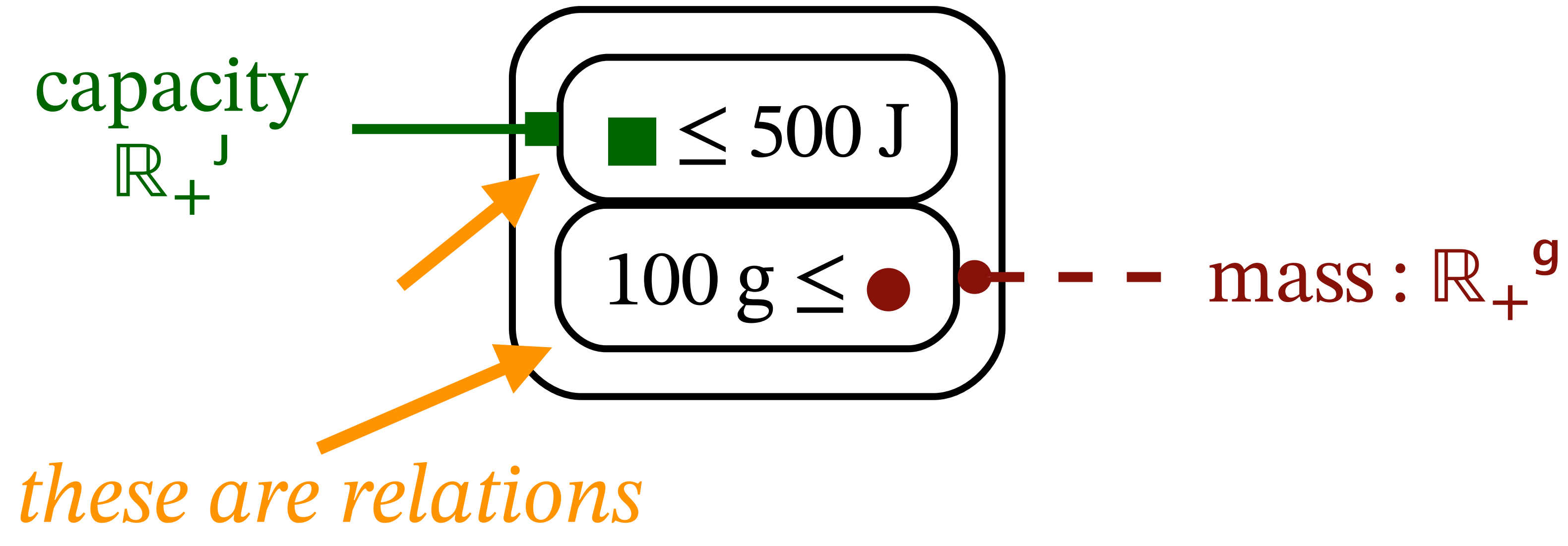
```
mcdp {
    provides capacity [J]
    requires mass [g]
    provided capacity ≤ 500 J
    required mass ≥ 100 g
}
```



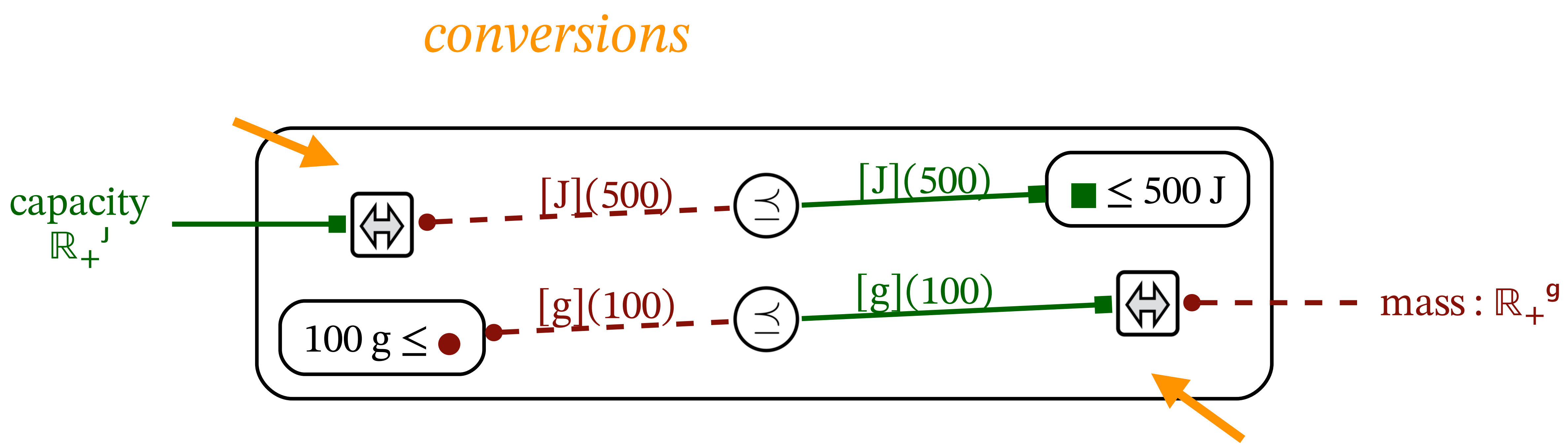
Using constants

- The simplest non-trivial complete model: put **constant bounds**.

```
mcdp {
    provides capacity [J]
    requires mass [g]
    provided capacity ≤ 500 J
    required mass ≥ 100 g
}
```



- **Actually,** that was a **simplified** diagram. This is how it looks like:

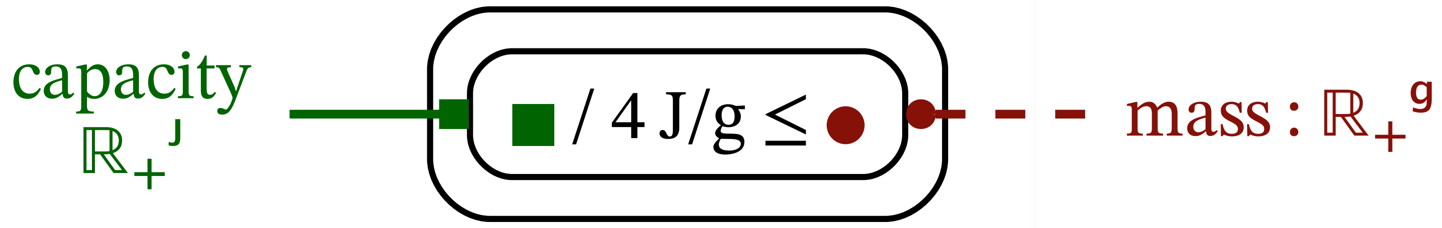



```
CompositeNDP
capacity    NWU[J](≥0)

 3 nodes, 2 edges
 connections:
(Connection(dp1=_fun_capacity, s1=capacity, dp2=_conversion1, s2=_op0),
 Connection(dp1=_conversion1, s1=_res, dp2=_lim1, s2=_l))
├ _fun_capacity: SimpleWrap
│                 capacity    NWU[J](≥0)
│                 capacity    NWU[J](≥0)
│                 └ dp: IdentityDP NWU[J](≥0) ↠ NWU[J](≥0)
│                       f ≤ r
├ _lim1          : SimpleWrap
│                 _l     NWU[J](500)
│                 └ dp: Limit NWU[J](500) ↠ 1 f ≤ 500 J
│                       f ≤ 500 J
│                      └ c: NWUValueWithUnits
│                           ├ value: 500
│                           └ unit : NWU[J](500)
└ _conversion1 : SimpleWrap
                  _op0     NWU[J](≥0)
                  _res     NWU[J](500)
                  └ dp: AmbientConversion NWU[J](≥0) ↠ NWU[J](500)
                        f ≤ r
                        common: NWU[J](Decimals)
```

Link between functionality and requirements

- **The simplest link is a linear relation.**

```
mcdp {
    provides capacity [J]
    requires mass [g]
    ρ = 4 J / g
    required mass ≥ provided capacity / ρ
}
```

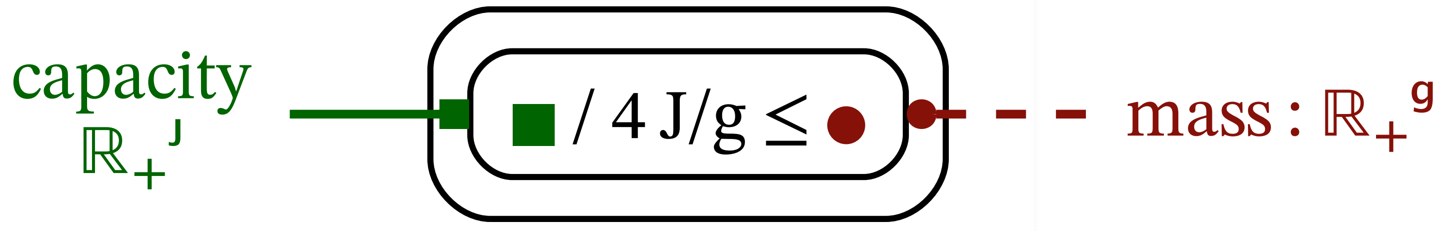


  - Note that dividing **by a constant** is a monotone operation.

Link between functionality and requirements

- **The simplest link is a linear relation.**

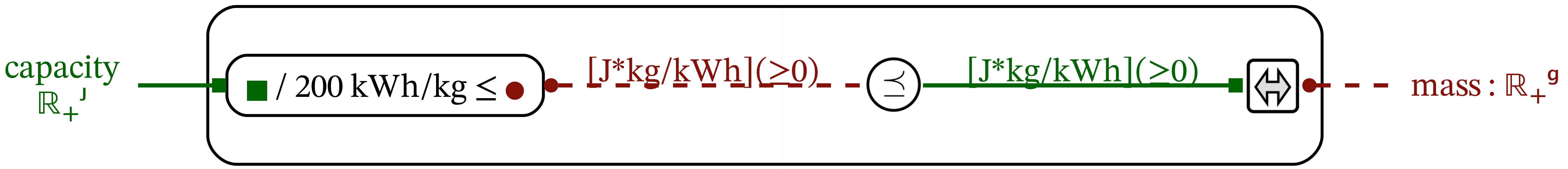
```
mcdp {
    provides capacity [J]
    requires mass [g]
    ρ = 4 J / g
    required mass ≥ provided capacity / ρ
}
```



  - Note that dividing **by a constant** is a monotone operation.

- As long as the **dimensionality** is correct, the software will take care of units.

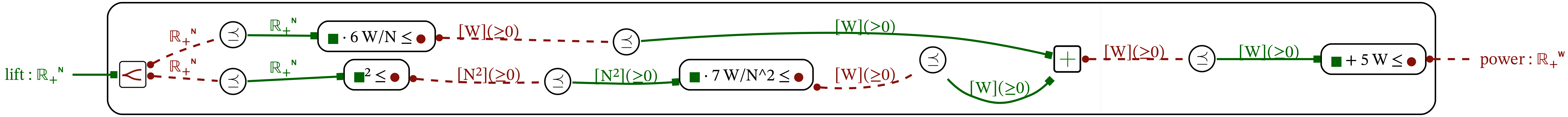
```
mcdp {
    '''
    Simple model of a battery as a linear relation
    between capacity and mass.
    '''
    provides capacity [J] 'Capacity provided by the battery'
    requires mass [g] 'Battery mass'
    ρ = 200 kWh / kg 'Specific energy'
    required mass ≥ provided capacity / ρ
}
```



Other example of numerical constraints

- In this example we have a (**positive**!) **polynomial constraint**:

```
mcdp {
    provides lift [N]
    requires power [W]

    l = provided lift
    p₀ = 5 W
    p₁ = 6 W/N
    p₂ = 7 W/N²
    required power ≥ p₀ + p₁ · l + p₂ · l²
}
```

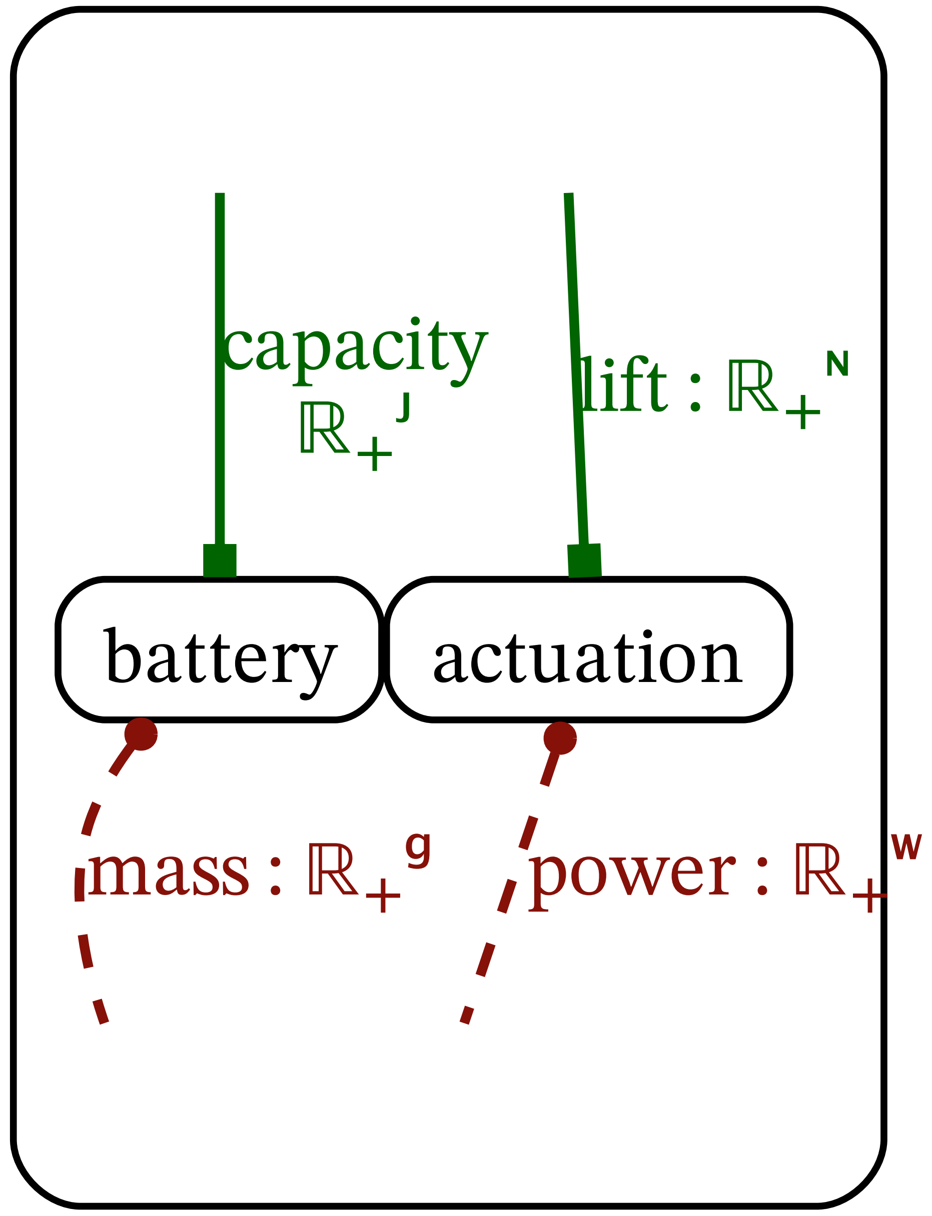


Hierarchical composition of MCDPs

- The **backtick syntax** refers to another model in the same library.
- Then we **instance** the MCDP “type” in current model.

```
mcdp {
    actuation = instance `Actuation1
    battery = instance `Battery1
}
```

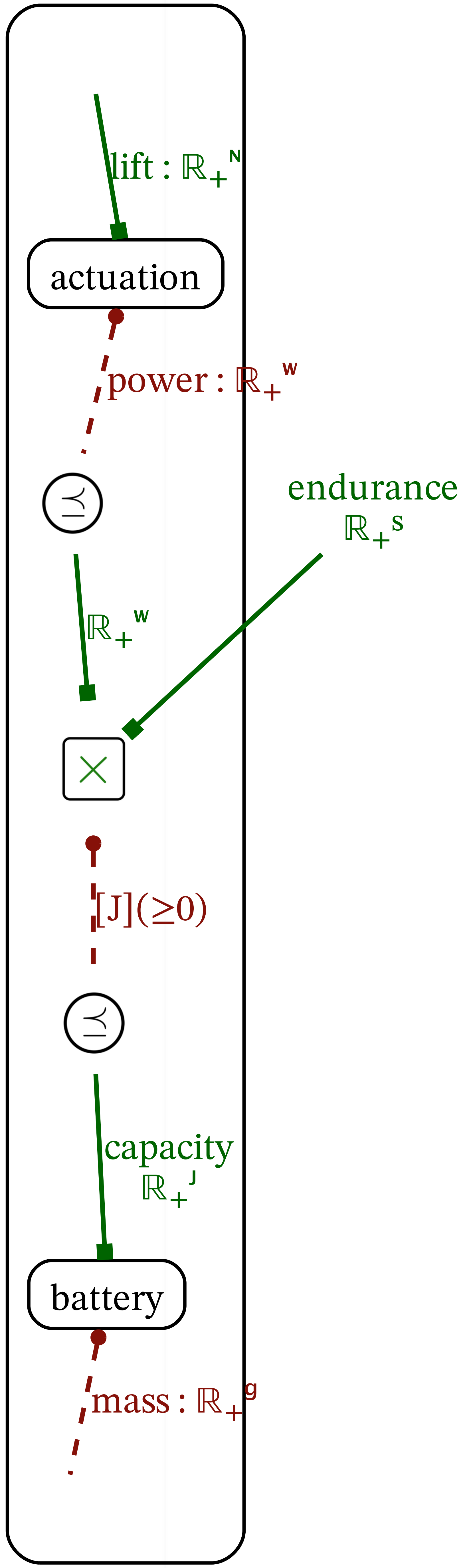
- We obtain an unconnected graph, because we need to say how to relate the functionality and requirements of the sub-design problems.



Writing co-design constraints

- If we have chosen the right formalization for functionality and requirements, **we expect that the co-design constraints will be easy to write.**
- This is not supposed to be an exercise of cleverness!
- Use the syntax “f provided by M” or “r required by M” to refer to the functionality/requirements of the subproblems.

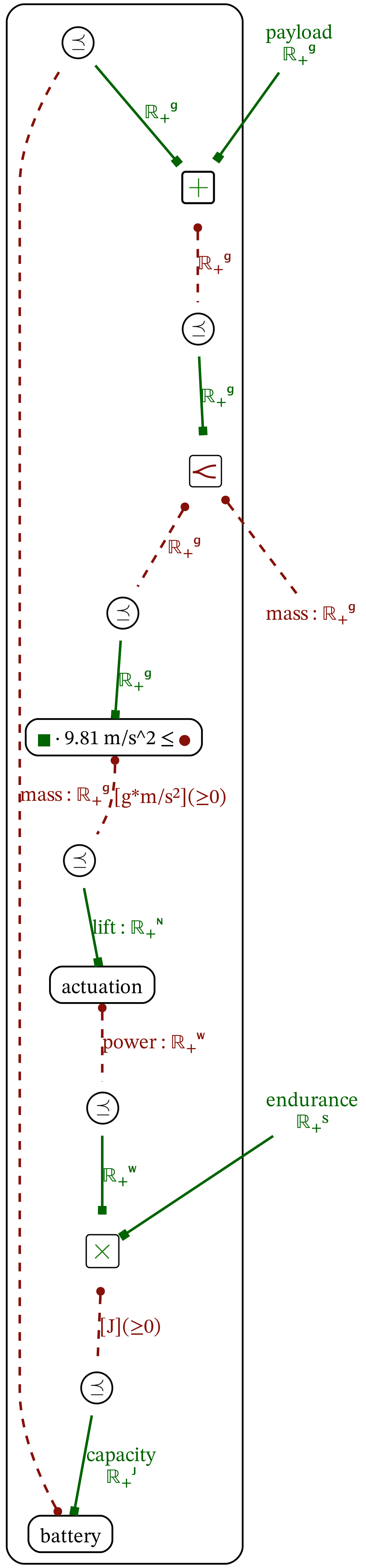
```
mcdp {
    provides endurance [s]

    actuation = instance `Actuation1
    battery = instance `Battery1

        # battery must provide power for actuation
    energy = provided endurance · (
                power required by actuation)
    capacity provided by battery ≥ energy
        # still incomplete...
}
```



Co-design loops

- **Any non-trivial design problem will introduce a loop.**

```
mcdp {
    provides endurance [s]
    provides payload [g]

    actuation = instance `Actuation1
    battery = instance `Battery1

    # battery must provide power for actuation
    energy = provided endurance · (
        power required by actuation)

    capacity provided by battery ≥ energy

    # actuation must carry payload + battery
    gravity = 9.81 m/s²
    total_mass = (mass required by battery +
     provided payload)
    weight = total_mass · gravity
    lift provided by actuation ≥ weight

    # minimize total mass
    requires mass [g]
    required mass ≥ total_mass
}
```



Numerical posets

- Some pre-defined posets in the language:
  - Nat: **natural numbers** including +inf
  - Int: **integers** including -inf and +inf
  - Rcomp = dimensionless: **non-negative real numbers**
  - m, J, W, s, miles, hours ... = **non-negative real numbers <u>with units</u>**

- The **internal representation** of these posets is an **implementation detail.**

- **Some representations:**
  - **[default]** Decimal numbers with $n = 9$ digits of precision.
  - Integer fractions
  - float64
  - Intervals of [*other representation*]

Numerical concerns

- Some of the **usual numerical analysis concerns do not apply.**

- We do not care about
  - **commutativity**: $a + b = b + a$
  - **associativity**: $a + (b + c) = (a + b) + c$
  - **inverses**: $a + b - b = a$

- **We only ask for monotonicity!** (Scott continuity)

New numerical concerns!

- We need **poset completeness** - numerical posets need to have a top = +inf
  - We need to extend all operations on +inf,-inf in a way that is also Scott-Continuous.
  - Example: +inf * 0 = +inf or +inf * 0 = 0 ?

- **Results containing** +inf are meant to be **interpreted as infeasible.**
  - Example: x + +inf <= x has +inf as a solution.

- **All operations come in pairs**, one for **lower sets** and one for **upper sets**
  - For example, consider: $f_1 *_L f_2 <= r_1 *_R r_2$
  - The **two multiplications** $*_L$ **and** $*_R$ **are different operations**!
  - Scott Continuity is defined with respect to either the Upper Sets or the Lower Sets topology.

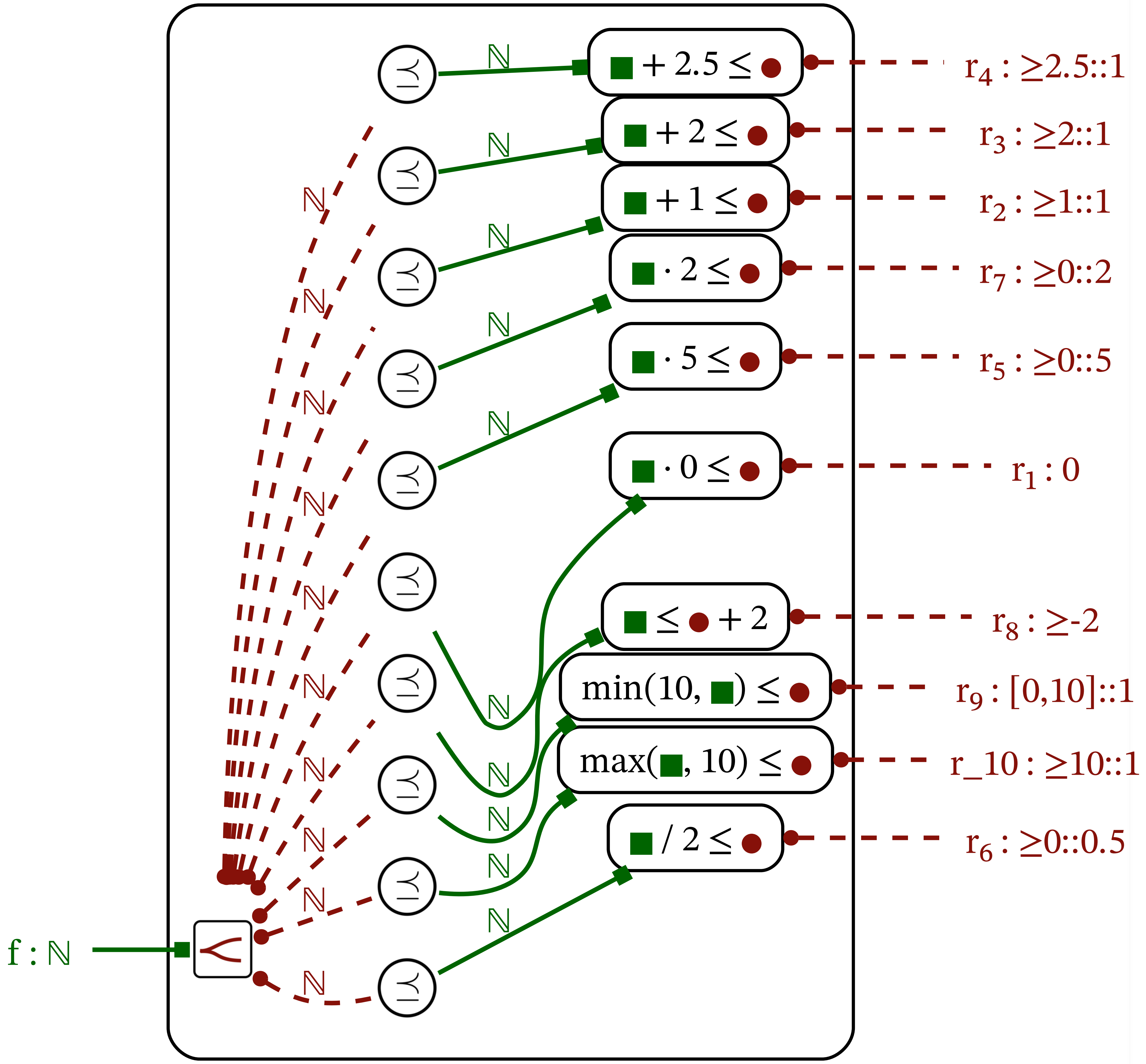
- **Some upper/lower sets are not representable!**
  - Example: solutions to $f_1 + f_2 <= 1$

Numerical poset inference

- The code has some capability of performing **numerical poset inference** by considering **refinement** that takes into account **lower bound**, **upper bound**, and **discretization**.
- This enables many **static optimizations**.

- **Example**: the functionality is a **natural number** but the requirements are not.

```
mcdp {
    provides f [ℕ]
    requires r₁ = provided f · 0
    requires r₂ = provided f + 1
    requires r₃ = provided f + 2.0
    requires r₄ = provided f + 2.5
    requires r₅ = provided f · 5
    requires r₆ = provided f / 2
    requires r₇ = provided f · 2
    requires r₈ = provided f - 2
    requires r₉ = min(provided f, 10)
    requires r_10 = max(provided f, 10)
}
```



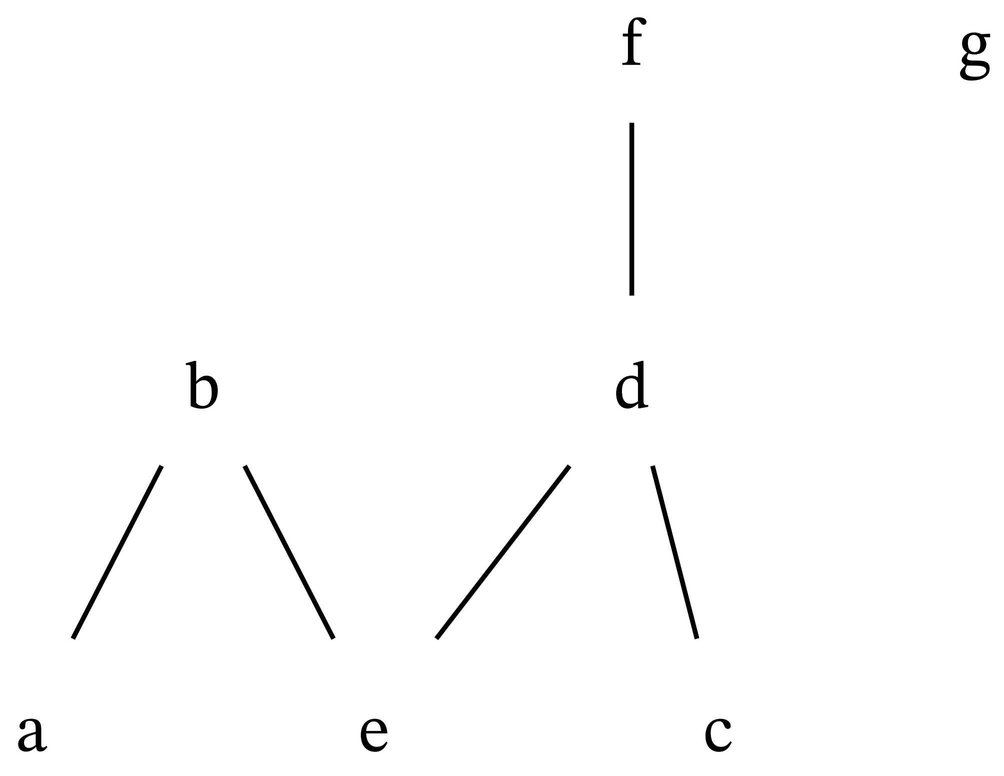
::5 = step equal to 5

Defining discrete posets

- We can define an arbitrary discrete poset by populating a.mcdp_poset file.

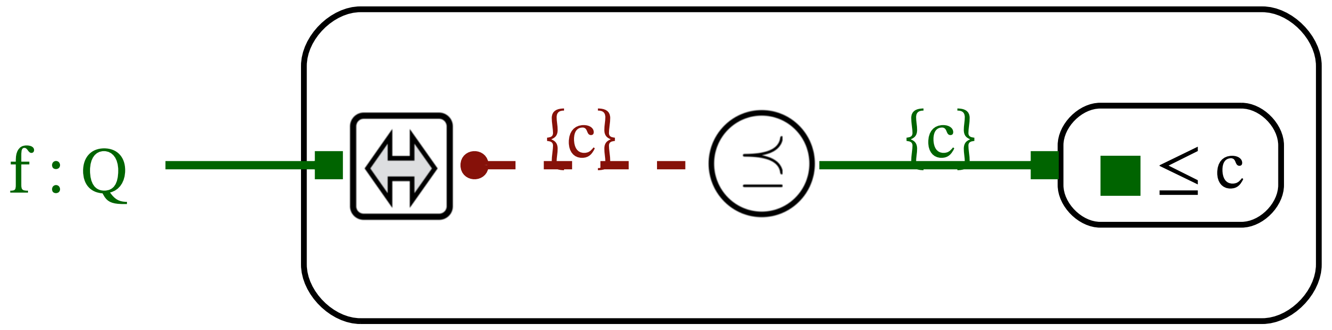
Q.mcdp_poset

```
poset {
    a ≤ b
    c ≤ d
    e ≤ d ≤ f
    e ≤ b
    g
}
```



- We can then refer to the poset using the backtick syntax wherever a poset is expected.
- Use the syntax `PosetName: Element to refer to an element of the poset.

```
mcdp {
    provides f [`Q]
    provided f ≤ `Q: c
}
```



Different ways to describe uncertainty

- Using the "**between**".

```
c = 10 kg
δ = 50 g
x = between c - δ and c + δ
```

- Using **absolute tolerances:**

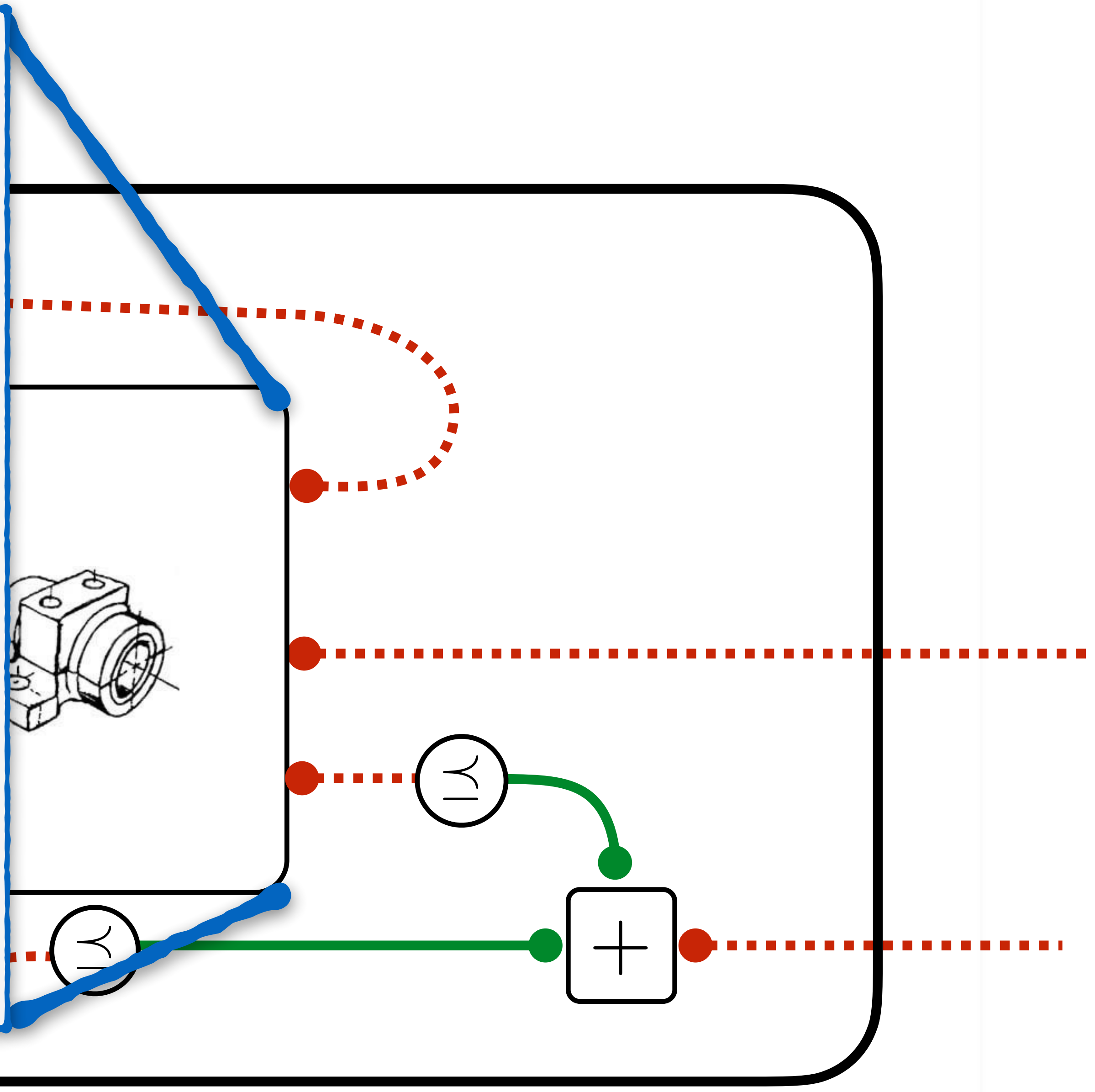
```
x = 10 kg ± 50 g
```

- Using **percentage tolerances** (10 kg +- 5%)

Uncertainty as a modeling tool

battery_uncertain.mcdp

```
mcdp {
    provides capacity [kWh]
    requires mass [g]
    requires cost [$]
    energy_density = between 140 kWh/kg and 150 kWh/kg
    specific_cost = 200 $/kWh
    required mass · energy_density ≥ provided capacity
    required cost ≥ provided capacity · specific_cost
}
```



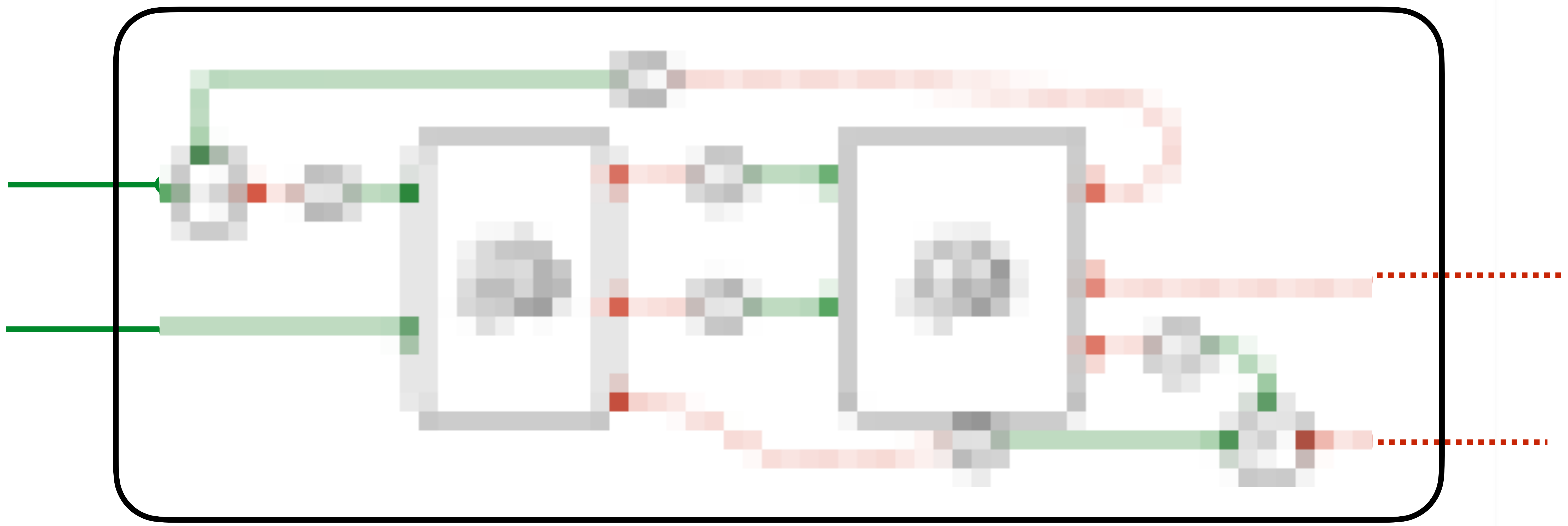
no uncertainty: "To obtain an endurance of **15 min**, the minimal cost is **$230**"

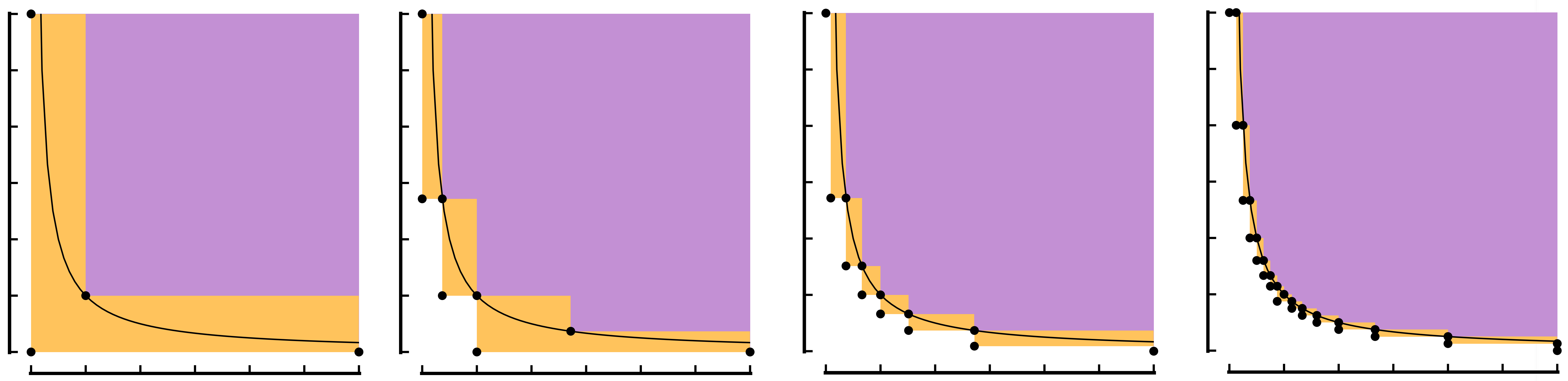
low uncertainty: "To obtain an endurance of **15 min**, the minimal cost is **between $220 and $240**"

high uncertainty: "To obtain an endurance of **15 min**, the minimal cost is **$220 in the best case, and in the worst case the problem is not feasible**"

Uncertainty for relaxation

- Algorithmically, to consider continuous posets (infinite number of solutions) we build a **sequence of design problems intervals** that **converge to the real problem.**





Templates

- **Templates are diagrams with typed holes.**

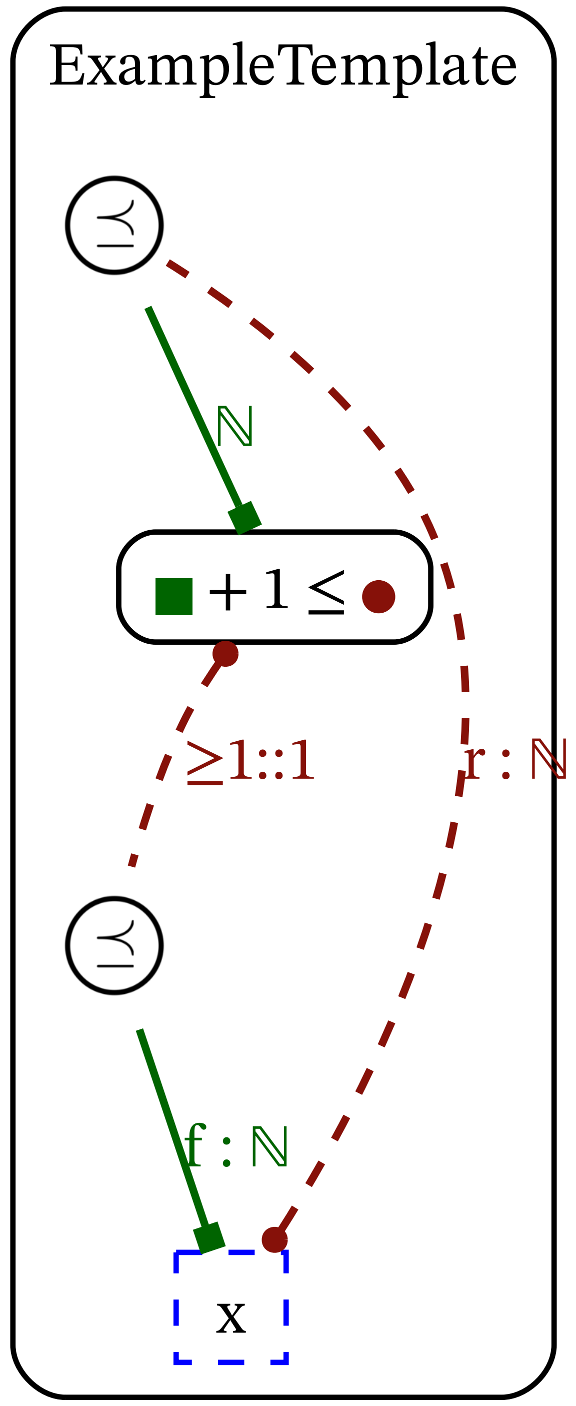
```
template [name1: interface1, name2: interface2]
mcdp {
    # usual definition here
}
```

- Inside the mcdp block, the template parameters are in scope.

ExampleInterface.mcdp

```
interface mcdp {
    provides f [ℕ]
    requires r [ℕ]
}
```

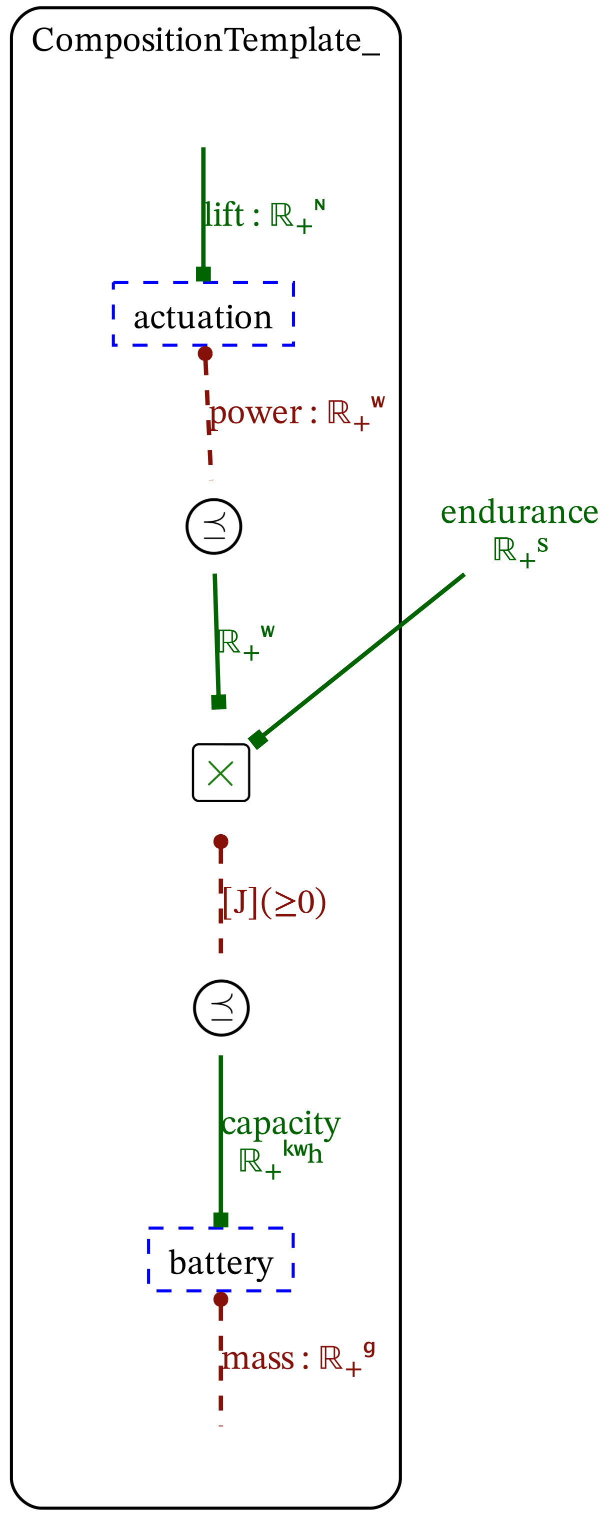
```
template [T: `ExampleInterface]
mcdp {
    x = instance T
    f provided by x ≥ r required by x + 1
}
```



Generalization of battery/actuation example

- We can abstract over the type of battery/actuation using the template construction:

```
template [
    generic_actuation: `ActuationInterface,
    generic_battery: `BatteryInterface
]
mcdp {
    actuation = instance generic_actuation
    battery = instance generic_battery
    # battery must provide power for actuation
    provides endurance [s]
    energy = provided endurance ·
    (power required by actuation)
    capacity provided by battery ≥ energy
    # only partial code
}
```



- And then **we use the template by specialization:**

```
specialize [
        generic_battery: `Battery1,
        generic_actuation: `Actuation1
] `CompositionTemplate
```

Loading models from other libraries and repositories

- The **backtick syntax** loads an object from the **current library.**

```
mcdp {
    T = `other_model
    a = instance T
}
```

- We can **qualify the name to refer to a different library**.

```
mcdp {
    T = `other_library.other_model
    a = instance T
}
```

- We can refer to a **different repository** with the “**from shelf**” syntax.

```
from shelf "github.com/org/repo@branch" import library other_library
mcdp {
    T = `other_library.other_model
    a = instance T
}
```

Union of models

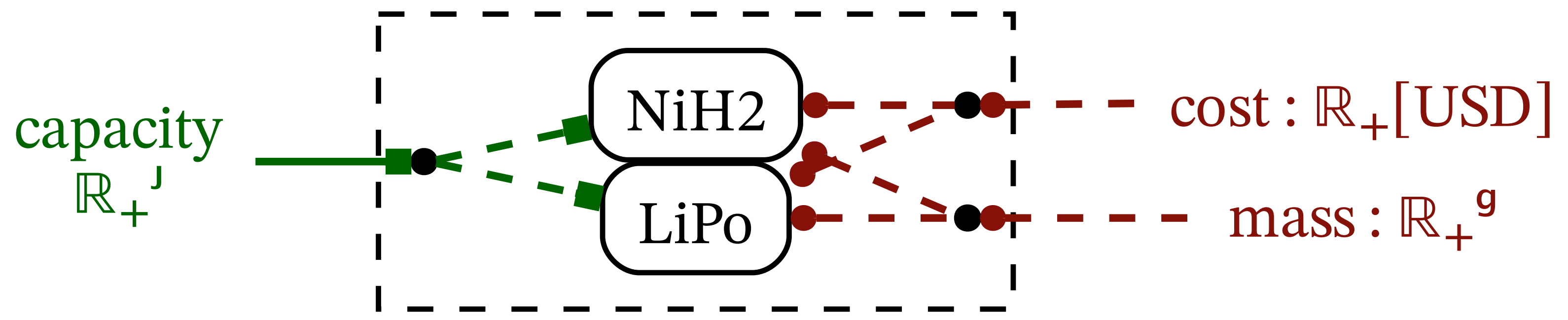
- We can specify **a model as the union of a finite number of known models using the "choose" keyword.**
- Example: we have two different battery technologies:


```
mcdp {
    provides capacity [J]
    requires mass [g]
    requires cost [USD]
    ρ = 150 Wh/kg
    α = 2.50 Wh/USD
    required mass ≥ provided capacity / ρ
    required cost ≥ provided capacity / α
}
```


```
mcdp {
    provides capacity [J]
    requires mass [g]
    requires cost [USD]
    ρ = 45 Wh/kg
    α = 10.50 Wh/USD
    required mass ≥ provided capacity / ρ
    required cost ≥ provided capacity / α
}
```

- We let the system choose the best:

```
choose(
    NiH2: `Battery1_LiPo,
    LiPo: `Battery1_NiH2
)
```



- (Future vision: "take the union of all compatible models in the world".)