Configuration with Model-Based Dependencies

— an experience report —

Gabor Greif
mailto:gabor.greif@alcatel-lucent.com

BOBkonf 2015
January 23

About me

With Alcatel-Lucent since 2000

Currently (also) working on *Safe and Secure European Routing* („SASER“), a BMBF-funded project

Coaching Bachelor Student: Philip Ottinger

The context we assume for this talk

Our setting is

- Embedded devices
- Haskell: no mutation, expressive types

Show of hands

- Experience with Haskell? Monads?
- GADTs?
- Proofs?
- Chemistry?

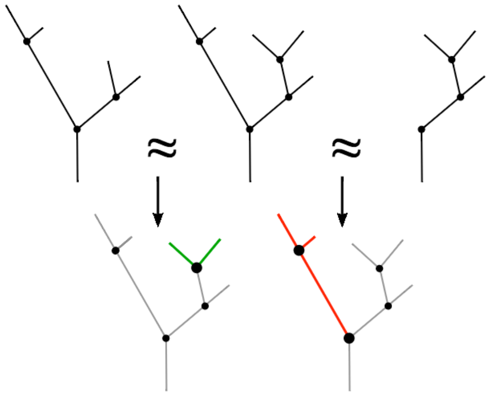
Agenda

1. How to use `gdiff` for computing effectful actions
2. How to ensure correct effect ordering

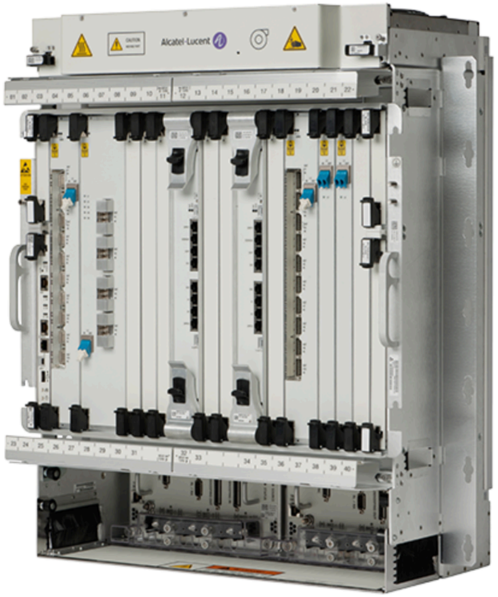
`gdiff` is a Haskell library for comparing values

Part One

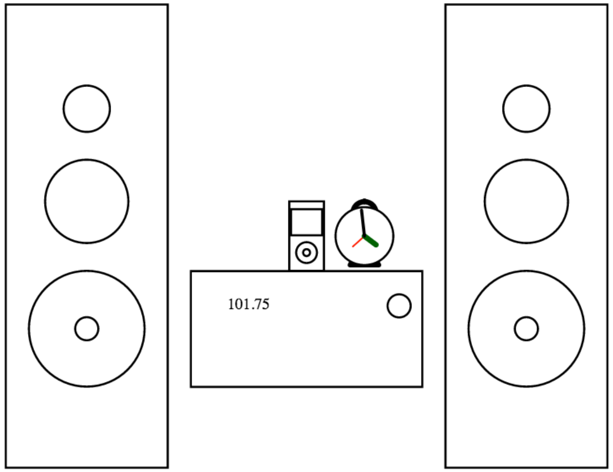
How to obtain configuration actions
by
comparing trees



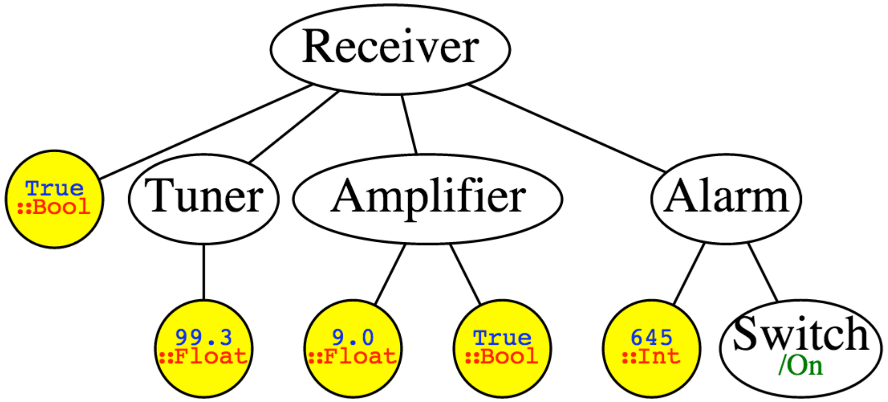
We are building these (1830 PSS)



But I shall explain things in terms of this familiar device



Configuration tree



`gdiff`: a fundamental utility

Like the well-known UNIX® programs `diff` and `patch`

Lempsink and Löh, 2010

  Generalized to arbitrary algebraic datatypes

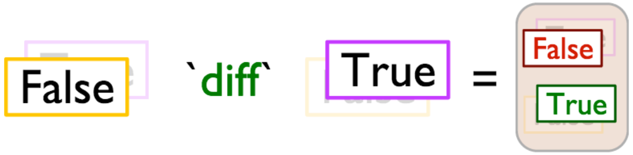
  Formally verified in Agda, library ported to Haskell

Comparing two trees of the same type (old vs. new)

```
diff :: a ⟶ a ⟶ EditScript_Fam a a
t_n `diff` t_n+1 = Δ
```

Simplest example:

```
λ> let delta = False `diff` True
 ➪ Ins True $ Del False $ End
```

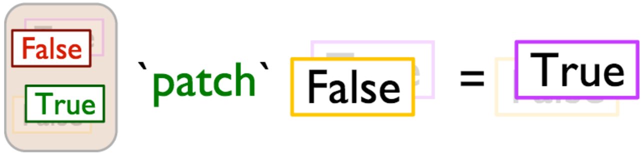


Applying edit script to a previous value

```
patch :: EditScript_Fam a a ⟶ a ⟶ a
Δ `patch` t_n = t_n+1
```

Example:

```
λ> delta `patch` False
 ➪ True
```



Designed to work on pure Haskell values (e.g. ADTs, tree-like data)

How it works (in a nutshell)

`diff` needs a view to nodes (locally), so the programmer
is in charge of supplying following infrastructure:

- Family GADT categorises all nodes occurring in tree (`data Fam`)

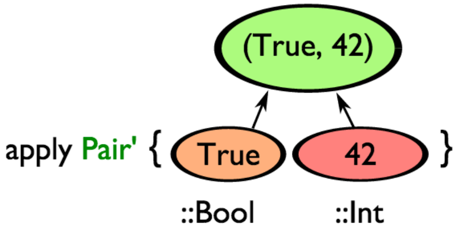
```
data Fam :: ★ → ★ → ★ where
  False' :: Fam Bool {}
  True' :: Fam Bool {}
```

- Each occurring type mapped to a subset of these by (`class Type`$_{Fam}$)

```
instance Type Fam Bool where
  constructors = [False', True']
```

- `class Family` mediates:
  - `decEq` compares node categories
    returning proofs that the node types match
  - `fields` returns a heterogeneous list of subtrees of a node
    effectively exposing the node structure to recursive invocations

  - `apply` creates a new tree, given a node descriptor and subtrees
    for use by `patch`

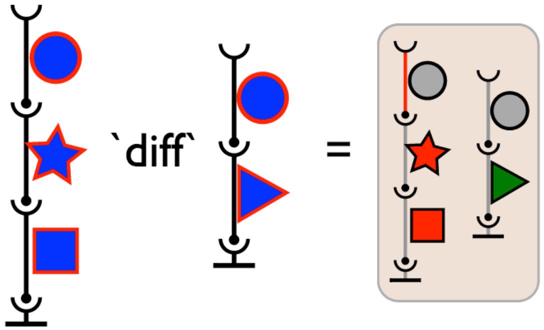


```
instance Family Fam where
  False' `decEq` False' = Just (Refl, Refl)
  True' `decEq` True' = Just (Refl, Refl)
  _ `decEq` _ = Nothing

  fields False' False = Just {}
  fields True' True = Just {}
  fields _ _ = Nothing
```

We added

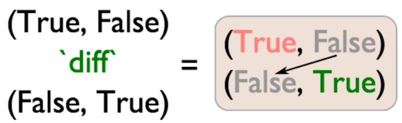
- polymorphic containers, e.g. $\texttt{Type}_{\mathit{Fam}}\ \texttt{a} \Longrightarrow \texttt{Type}_{\mathit{Fam}}\ \texttt{[a]}$
- …other features, described later



Encountered problems

- `diff` moves subtrees around, e.g.

```
λ> (True, False) `diff` (False, True)
  ➪ Ins True $ Cpy False $ Del True $ … $ End
```



Same thing happens with textual `diff`:
⌘-⌥-☞ see github

While this is an intentional optimization, it leads to *unphysical moves*

When hardware-related configuration parameters change, we always require

$\texttt{Ins}\ v_{n+1}\ \$\ \texttt{Del}\ v_n\ \$\ \ldots$

in edit scripts, corresponding to *APIs*

We added (cont'd)

- polymorphic containers, e.g. `Type a` ⟹ `Type [a]`
- Locations added to data types to pin them
  `Bool` becomes $\texttt{Bool}_{\texttt{loc}}$
- …other features, described later

Locations for our radio device

on the *type* level

```
data Loc
  = Receiver
  | Amplifier Loc
  | Tuner Loc
  | Pod Loc
  | Alarm Loc
```

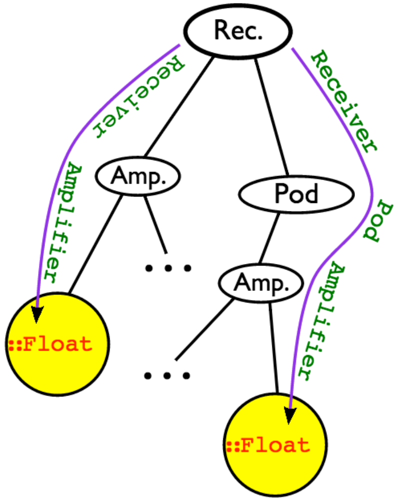
describe *paths* to nodes

Volume setting of the iPod earphones is then

$\texttt{Float}_{\texttt{Amplifier (Pod Receiver)}}$

Loudness of the speakers

$\texttt{Float}_{\texttt{Amplifier Receiver}}$



We use datatype promotion
to obtain a Loc kind:
{-# LANGUAGE DataKinds #-}

```
newtype Located t (l :: Loc) = Loc t
```

$$\text{Float}_{\text{Amplifier Receiver}} \equiv \text{Located Float (Amplifier Receiver)}$$

At this point

We can create (pure) edit scripts without unphysical movements

EditScript$_{Fam}$ Configtree Configtree

But we would like patch to have an effectful (i.e. monadic) result:

IO Configtree

with potentially non-trivial actions included

For this (deducing backwards) our scripts must have following type:

EditScript$_{Fam}$ (IO Configtree) (IO Configtree)

So diff must also be called with IO Configtree

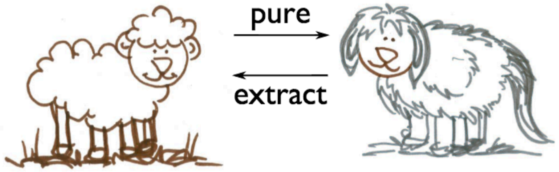
```
loop :: Configtree → IO ()
loop conf_n = do
  conf_{n+1} ← runUI conf_n
  let delta = ? conf_n `diff` ? conf_{n+1}
  patch delta (? conf_n)
  loop conf_{n+1}
```

Idea: `diff` of pure **actions**

for example

$\texttt{t} \xrightarrow[\texttt{pure}]{} \texttt{IO t} \xrightarrow[\texttt{unsafePerformIO}]{} \texttt{t}$

is the identity

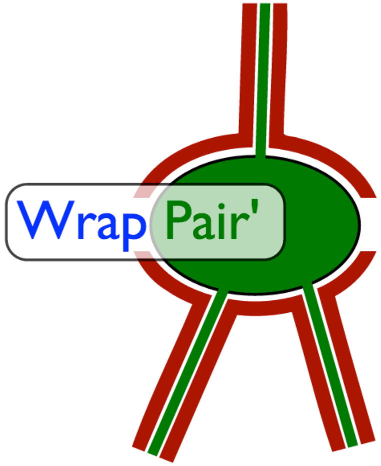


(fortunately many monads/applicatives like this with *disciplined* extraction exist)

All we need to do is to wrap existing `Fam` GADT descriptors:

$\texttt{Wrap :: Fam t sub}_{\texttt{t}} \longrightarrow \texttt{Fam (IO t) (Map IO sub}_{\texttt{t}}\texttt{)}$

```
fields (Wrap desc) action
  = wrapIO (fields desc $ extract action)
```

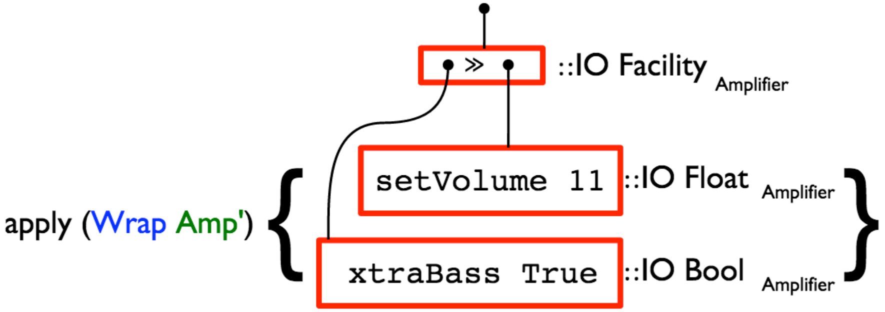


At this point we have

$\texttt{patch} :: \texttt{EditScript}_{\mathit{Fam}}\ (\texttt{IO Configtree})\ (\texttt{IO Configtree}) \longrightarrow$
$\quad\texttt{IO Configtree} \longrightarrow \texttt{IO Configtree}$

Locations permit specialization of actions created:

$\quad\texttt{Float}_{\texttt{Tuner} \ldots} \Longrightarrow \texttt{setTunerFrequency}$
$\quad\texttt{Float}_{\texttt{Amplifier} \ldots} \Longrightarrow \texttt{setVolume}$



Departing from the IO monad

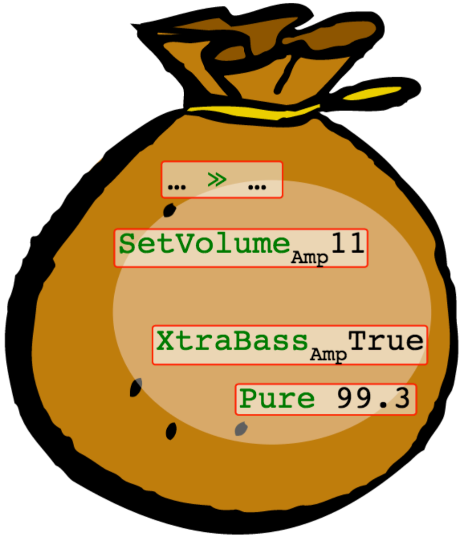
IO actions are too restricted for our purposes

Generalization to Monad m ⇒ m Configtree is straightforward, and permits, e.g.

- tracing of execution
- timing measurements
- mobile code
- visualization
- property-based testing (e.g. QuickCheck)

For the rest of the talk we assume a Bag implementation, that supports

- injection of Pure values
- parallelism of actions (Par)
- sequencing, essentially a monadic (>>)
- a range of primitive actions (e.g. SetVolume, etc.)


... ≫ ...
$\texttt{SetVolume}_{\text{Amp}}\ 11$
$\texttt{XtraBass}_{\text{Amp}}\ \texttt{True}$
Pure 99.3

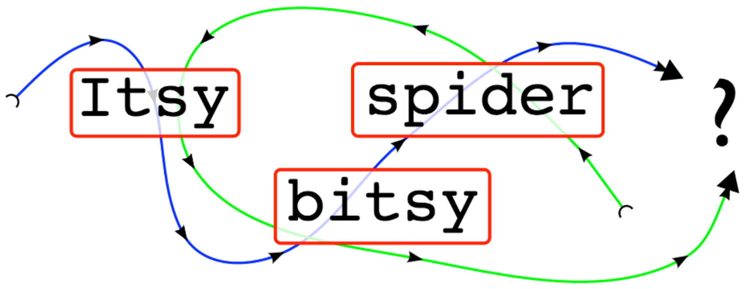
Part Two

A sequencing problem

and

the molecular analogy

(ongoing work)



Configuration by remote commands (CLI)

Running example is this command

```
$ set-alarm -time Now -active Off
```

This should be interpreted as one transaction
Hardware should be updated on commit

The non-obvious problem: effect ordering matters

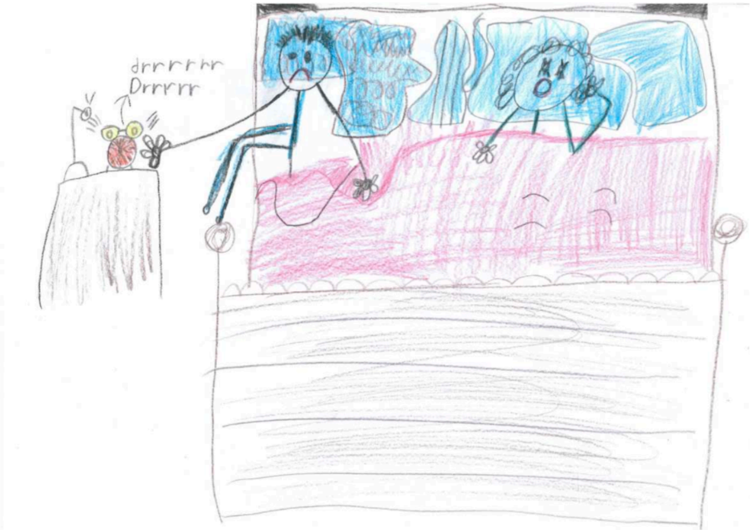
Let's assume the alarm clock is switched on

The CLI command

```
  $ set-alarm -time Now -active Off
```

when implemented naïvely (e.g. by performing actions as written)

may cause a transient **beep!**



Fixed reordering does not help,
example:

```
  $ set-alarm -active On -time 6:45
```

We have to deal with **context-dependency**!

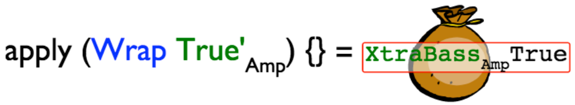
Caveat: hardware cannot be updated atomically

Atomic actions

Actions coming out of a leaf diff are considered atomic:

Atomic — The name comes from the Greek ἄτομος („indivisible“)

(e.g. our primitives `SetVolume`, etc.)



apply (Wrap True'$_{Amp}$) {} = XtraBass$_{Amp}$ True

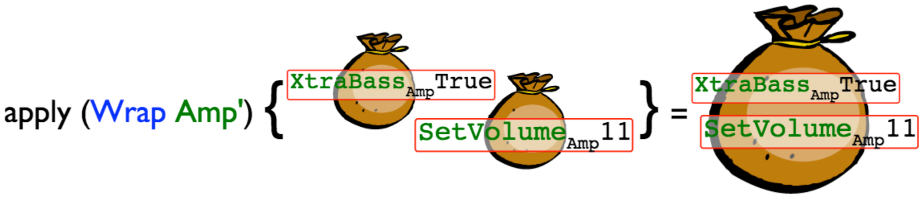
— vs. —

Compound actions

at each structured node its sub-actions are absorbed by a bag, so they become inherently parallel

We intend to exploit *dependencies* for sequencing

Embarrassing parallelism needs to be controlled



apply (Wrap Amp') { XtraBass$_{Amp}$ True SetVolume$_{Amp}$ 11 } = XtraBass$_{Amp}$ True SetVolume$_{Amp}$ 11

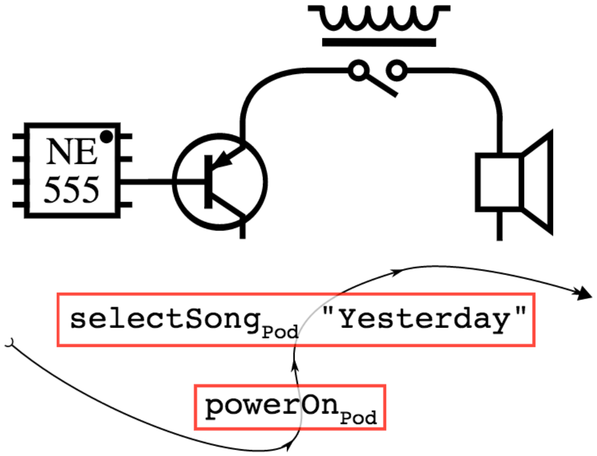
Where dependencies arise

Dependencies are dictated by the hardware

Configuring enclosing units before its parts
Dually, reversed order for controlled removal

Other model-specific dependencies, such as:

- suppressing transients
- modelling resources: buses, CPU cores



$\texttt{selectSong}_{\texttt{Pod}}$ `"Yesterday"`

$\texttt{powerOn}_{\texttt{Pod}}$

A DSL for stating dependencies

`make` is a decent language for describing dependencies

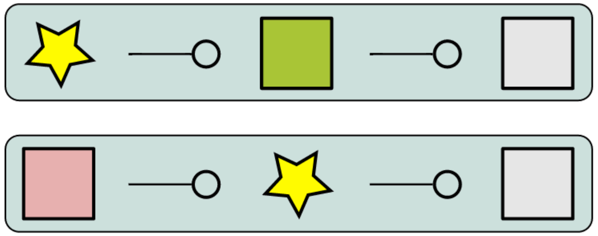
```
%.c.o: %.c
  gcc $< -o $@
```

We'll add rules to our `Bags`
but these serve to only model ordering

Our rules are written in terms of (abstract) locations
and strongly resemble Haskell function signatures

- `Time (Alarm …) ⊸ Switch (Just True) (Alarm …) ⊸ Switch Nothing (Alarm …)`
- `Switch (Just False) (Alarm …) ⊸ Time (Alarm …) ⊸ Switch Nothing (Alarm …)`

In symbols:



How rules consume inputs

Rule evaluation is reminiscent of organic chemistry:

- rules can be seen as catalysts (enzymes), which bind atoms to obtain sequenced molecules
- partially saturated molecules are the other active substances
- reactions in `Bags` run until a fixpoint is reached

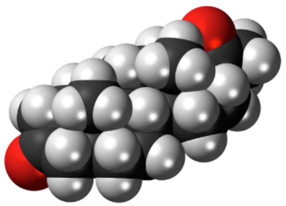
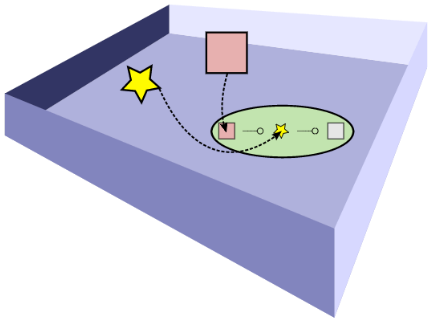


Image credit: Wikipedia.org

(N.B.: In informatics this is also called the *linear lambda calculus*)

Binding

- requires a proof that locations match
- changes ⊸ to ≫



Responsibilities

The author of the rules needs to ensure that the rules

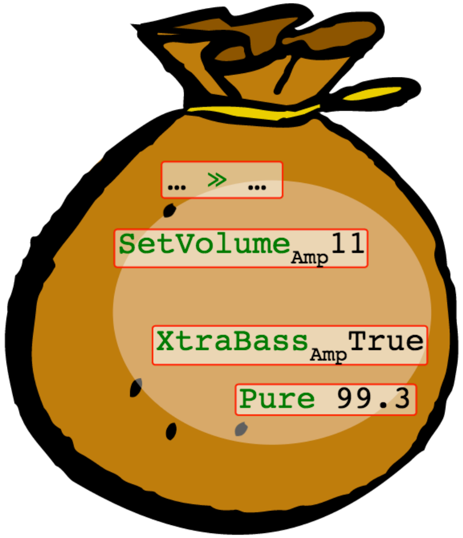
1. are terminating
2. and confluent

Our evaluator takes care of linearity

The molecular analogy

In summary, we can establish the following correspondence between `Bag` constructors and chemical substances

- `Pure`: (*irrelevant*)
- `ActivateAlarm`, etc.: **atoms**
- `(>>)`, sequencing: **molecules** (compounds)
- `Rule`: **catalysts**
- `Par`: **free substances**, unordered in reaction container



Conclusions

We sketched a declarative way to model
the profoundly effectful domain of HW configuration, by

- teaching `gdiff` to handle effectful actions
- starting out with maximal parallelism, and describing dependencies with a DSL
- obtaining strong guarantees by requiring proofs for type equalities

Thanks for listening!

Questions?

Trademarks:

UNIX is a registered trademark of The Open Group

iPod is a trademark of Apple Inc., registered in the U.S. and other countries

Image credits:

Wikipedia.org

Funding:

This work has been funded in parts by the German Ministry for Research and Education (BMBF grant „SASER“)

Alcatel·Lucent

Bundesministerium
für Bildung
und Forschung

SASER
SaveNet

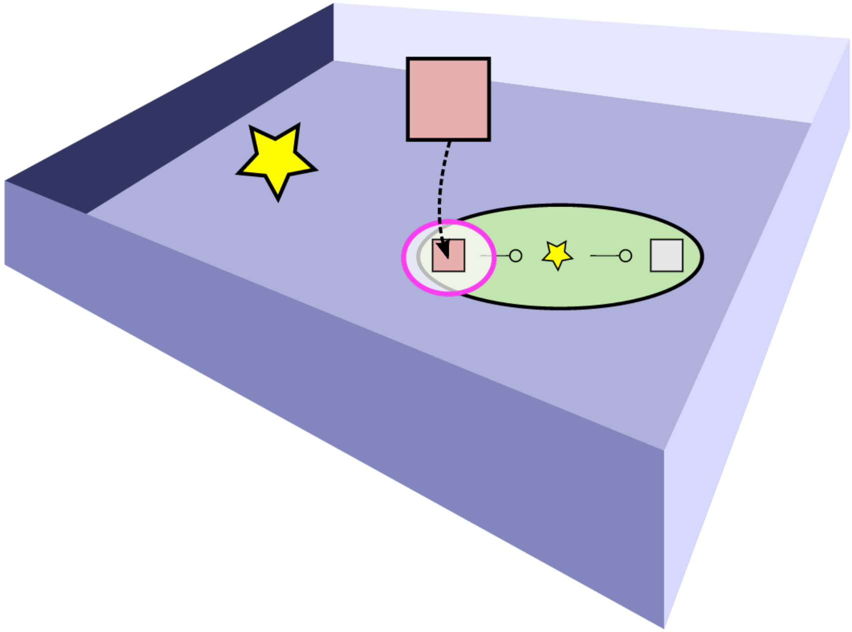
Backup Slides

Fixpoint reaction

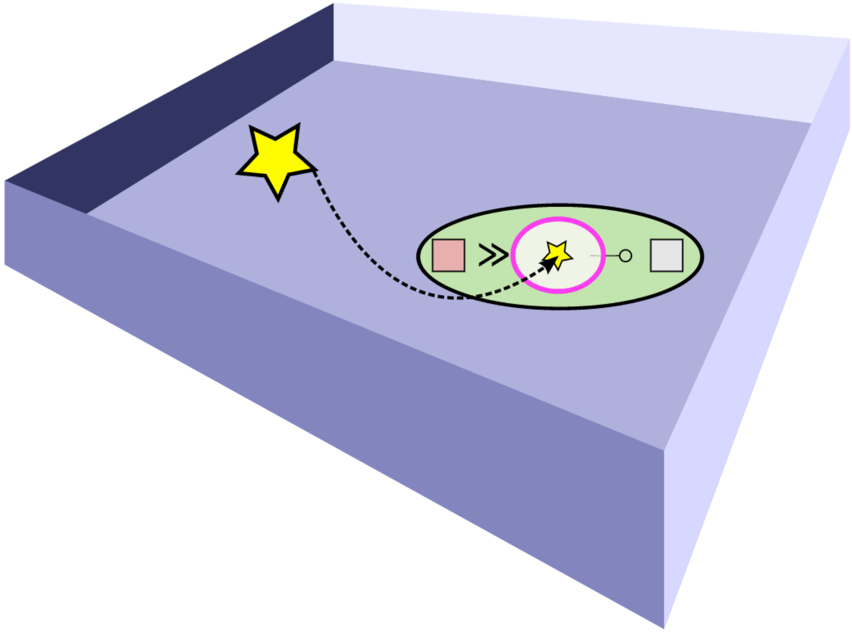
with

a rule

Fixpoint reaction: start



Fixpoint reaction: bind first



Fixpoint reaction: bind second