

Clojure

a language for problem solvers



@stuarthalloway
stu@thinkrelevance.com
stuarthalloway@datomic.com

Software Should Be

Knowledgeable

Powerful

Flexible

Intelligent

Effective

Pervasive

The Laws

Memory is expensive

Storage is expensive

Machines are precious

Resources are dedicated

C, C#, and Java:

Bringing hardware
presumptions from the 1970s
to bear on today's problems

Problem 1: Transience

characteristic	transient structure
sharing	difficult
distribution	difficult
concurrent access	difficult
access pattern	eager
caching	difficult
examples	Java and .NET collections relational databases NoSQL databases

Problem 2: Complexity

Nonlinear difficulty maintaining systems as they grow

Inability to make incremental change

Complete collapse of large efforts

Clojure

Persistent data structures

Powerful and flexible

State is a *succession of values*

Lisp

Designed by and for professionals

Extensible Data Notation (edn)

Rich set of built in data types

Generic extensibility

Language neutral

Represents values (not identities, objects)

type	example	java equivalent
string	"foo"	String
character	\f	Character
a. p. integer	42	Int/Long/BigInteger
double	3.14159	Double
a.p. double	3.14159M	BigDecimal
boolean	true	Boolean
nil	nil	null
ratio	22/7	N/A
symbol	foo, +	N/A
keyword	:foo, ::foo	N/A

type	properties	example
list	singly-linked, insert at front	(1 2 3)
vector	indexed, insert at rear	[1 2 3]
map	key/value	{ :a 100 :b 90 }
set	key	# { :a :b }

Clojure programs are written
in data, not text

Function Call

semantics:

fn call

arg

(println "Hello World")

structure:

symbol

string

list

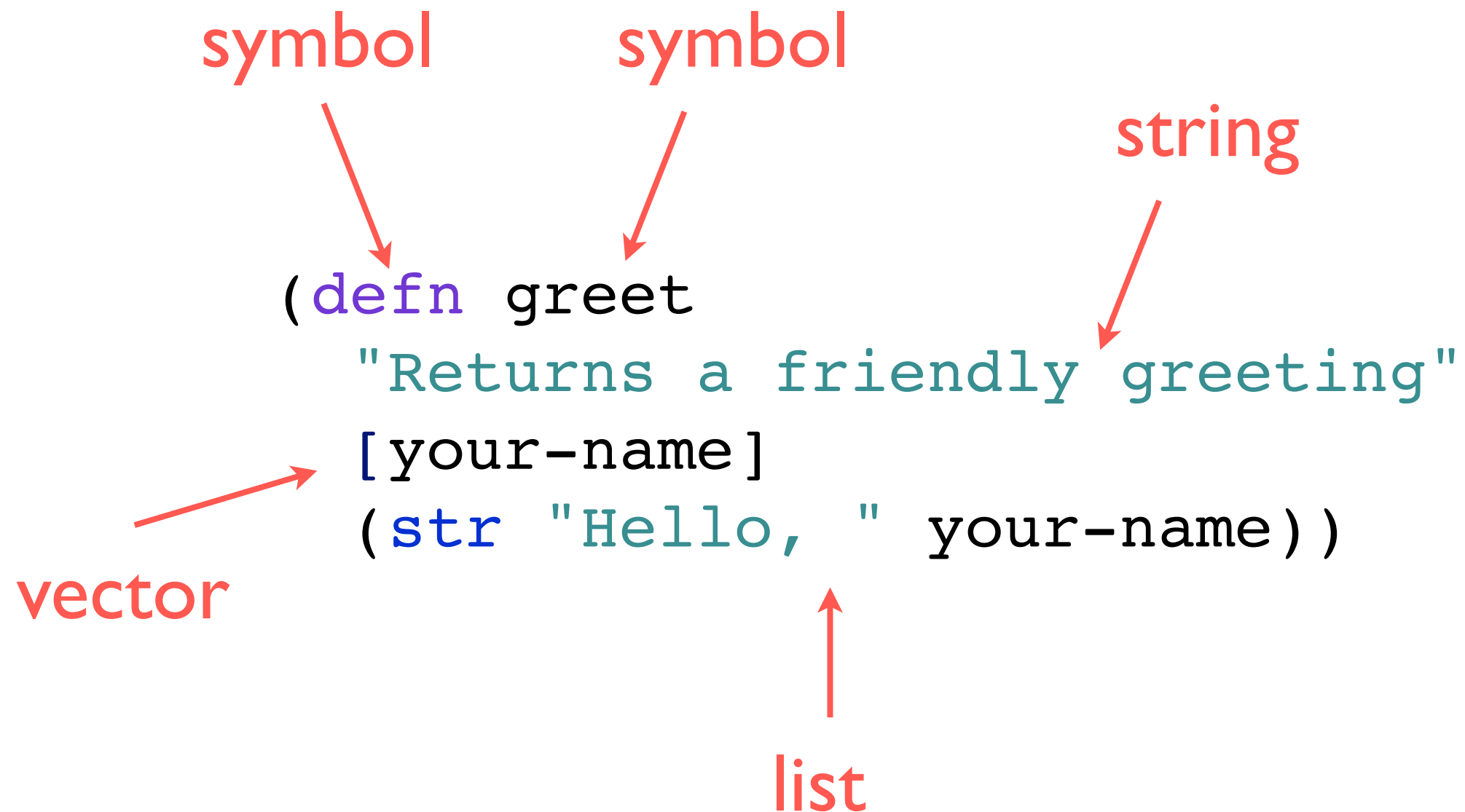
Function Definition

The diagram illustrates the components of a function definition in Clojure. Red text labels with arrows point to specific parts of the code:

- define a fn** points to `(defn`
- fn name** points to `greet`
- docstring** points to `"Returns a friendly greeting"`
- arguments** points to `[your-name]`
- fn body** points to `(str "Hello, " your-name)`

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

Still Just Data



Software Should Be

Knowledgeable: **Persistent Data Structures**

Powerful

Flexible

Intelligent

Effective

Pervasive

Persistent Data Structures

Immutable

“Change” by function application

Maintain performance guarantees

Full-fidelity old versions

Transience vs. Persistence

characteristic	transient	persistent
sharing	difficult	trivial
distribution	difficult	easy
concurrent access	difficult	trivial
access pattern	eager	eager or lazy
caching	difficult	easy
examples	Java, .NET collections relational databases NoSQL databases	Clojure, F# collections Datomic database

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures." - Alan J. Perlis

Vectors

```
(def v [42 :rabbit [1 2 3]])
```

```
(v 1) -> :rabbit
```

```
(peek v) -> [1 2 3]
```

```
(pop v) -> [42 :rabbit [1 2 3]]
```

```
(subvec v 1) -> [:rabbit [1 2 3]]
```

Maps

```
(def m {:a 1 :b 2 :c 3})
```

```
(m :b) -> 2
```

```
(:b m) -> 2
```

```
(keys m) -> (:a :b :c)
```

```
(assoc m :d 4 :c 42) -> {:d 4, :a 1, :b 2, :c 42}
```

```
(dissoc m :d) -> {:a 1, :b 2, :c 3}
```

```
(merge-with + m {:a 2 :b 3}) -> {:a 3, :b 5, :c 3}
```

Nested Structure

```
(def jdoe {:name "John Doe",  
          :address {:zip 27705, ...}})
```

```
(get-in jdoe [:address :zip])
```

```
-> 27705
```

```
(assoc-in jdoe [:address :zip] 27514)
```

```
-> {:name "John Doe", :address {:zip 27514}}
```

```
(update-in jdoe [:address :zip] inc)
```

```
-> {:name "John Doe", :address {:zip 27706}}
```

Sets

```
(use clojure.set)
(def colors #{"red" "green" "blue"})
(def moods #{"happy" "blue"})
```

```
(disj colors "red")
-> #{"green" "blue"}
```

```
(difference colors moods)
-> #{"green" "red"}
```

```
(intersection colors moods)
-> #{"blue"}
```

```
(union colors moods)
-> #{"happy" "green" "red" "blue"}
```

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: **Full Platform Access**

Flexible

Intelligent

Effective

Pervasive

Constructor

platform	<code>new Widget("foo")</code>
Clojure	<code>(Widget. "red")</code>

Static Member

platform	Math.PI
Clojure	Math/PI

Instance Method

platform	<code>rnd.nextInt()</code>
Clojure	<code>(.nextInt rnd)</code>

Instance Field

platform	pixels.data
Closure	(.-data pixels)

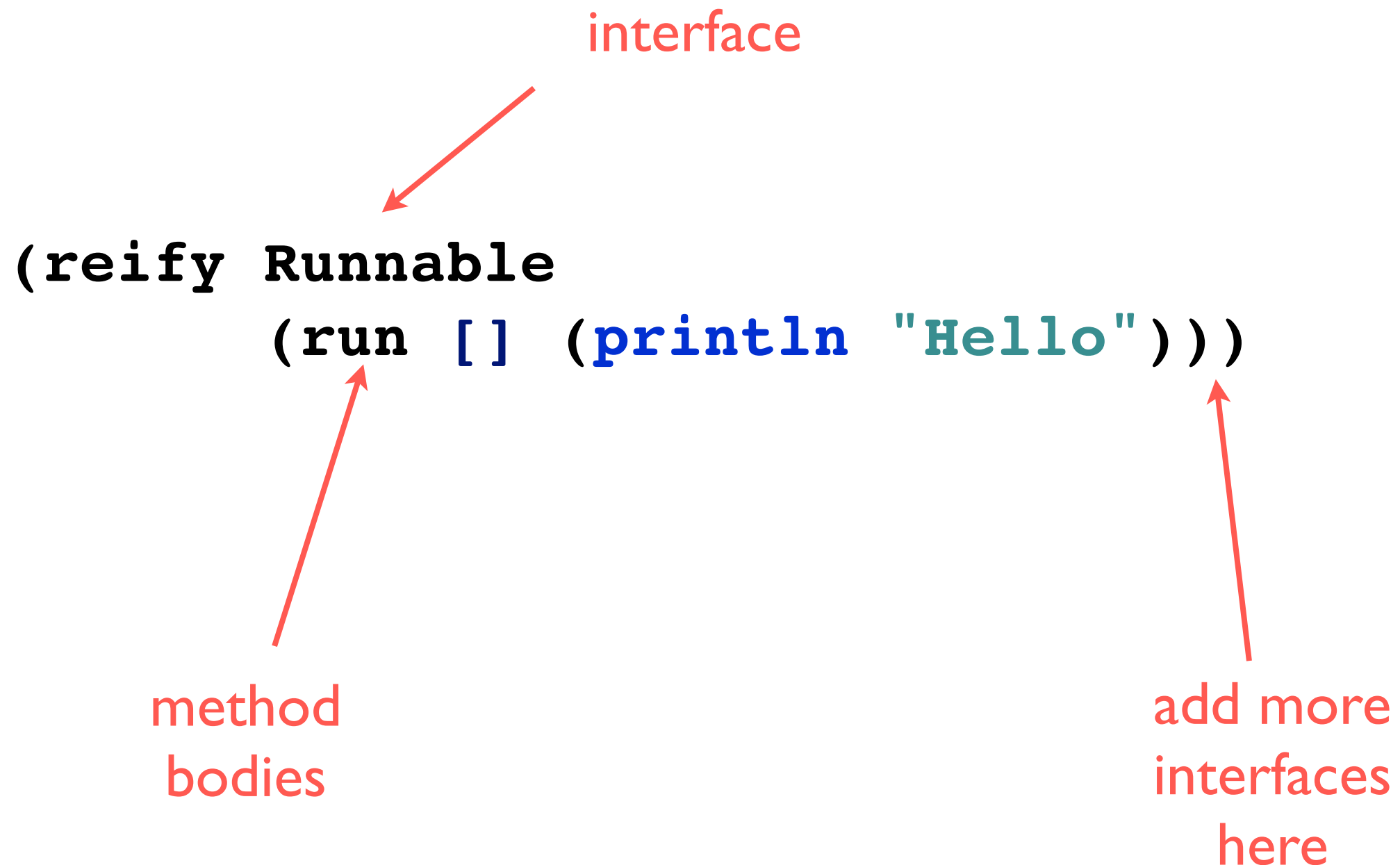
Chaining Access

platform	<code>person.getAddress().getZipCode()</code>
Clojure	<code>(.. person getAddress getZipCode)</code>

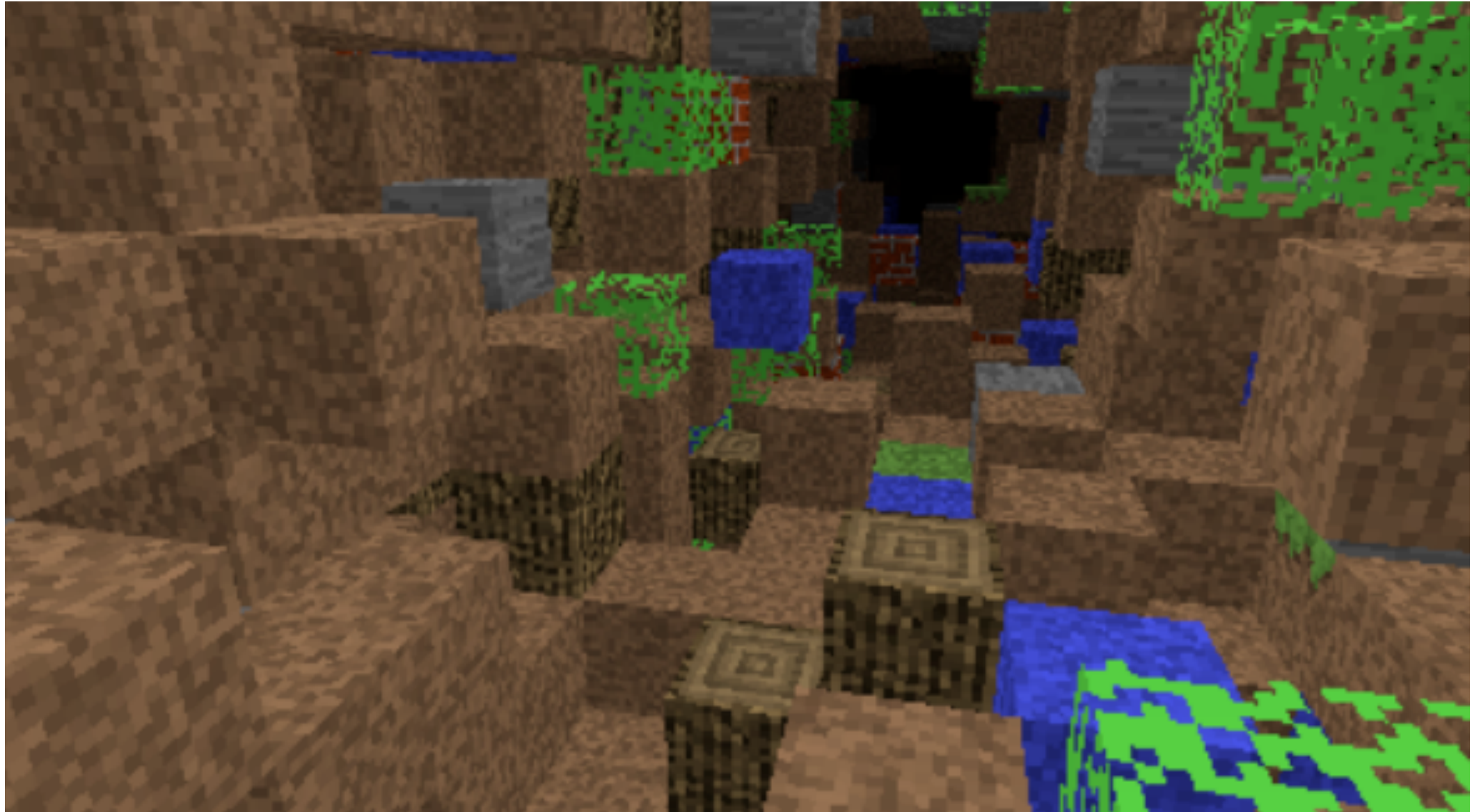
Paren Count

platform	() () () ()
Clojure	() () ()

Implement Interface



Full-power JavaScript



<http://swannodette.github.io/2013/06/10/porting-notchs-minecraft-demo-to-clojurescript/>

Platform Interop

Simple

Wrapper-Free

Performant

Conformant

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: **Interactive**

Intelligent

Effective

Pervasive



Give me some Clojure:

>

[links](#)

[about](#)

Welcome to Clojure! You can see a Clojure interpreter above - we call it a *REPL*.

Type `next` in the REPL to begin.

©2011-2012 Anthony Grimes and numerous contributors.

<http://tryclj.com/>

Himera REPL v0.1.5

`cljs.user>`

[View source on Github](#)



////////////////////////////////////
[ClojureScript at a glance - PDF](#) | [Searchable docs](#) | [Translations from JavaScript](#)

Datatypes

MAP `{:key1 :val1, :key2 :val2}`

VECTORS `[1 2 3 4 :a :b :c 1 2]`

SETS `#{:a :b :c 1 2 3}`

SCALARS `a-symbol, :a-keyword, "a string"`

ARRAYS `(array 1 2 3)`

Useful Functions

MATH `+ - * / quot rem mod inc dec max min`

COMPARISON `= == not= < > <= >=`

PREDICATES `nil? identical? zero? pos? neg? even? odd?
true? false?`

DATA PROCESSING `map reduce filter partition split-at split-
with`

<http://himera.herokuapp.com/index.html>

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: **Interactive**, **Dynamic**

Intelligent

Effective

Pervasive

Example: refactor Apache
Commons isBlank

Initial Implementation

```
public class StringUtils {  
    public static boolean isBlank(String str) {  
        int strLen;  
        if (str == null || (strLen = str.length()) == 0) {  
            return true;  
        }  
        for (int i = 0; i < strLen; i++) {  
            if ((Character.isWhitespace(str.charAt(i)) == false)) {  
                return false;  
            }  
        }  
        return true;  
    }  
}
```

Remove Type Declarations

```
public class StringUtils {  
    public isBlank(str) {  
        if (str == null || (strLen = str.length()) == 0) {  
            return true;  
        }  
        for (i = 0; i < strLen; i++) {  
            if ((Character.isWhitespace(str.charAt(i)) == false)) {  
                return false;  
            }  
        }  
        return true;  
    }  
}
```

Remove Enclosing Class

```
public isBlank(str) {  
    if (str == null || (strLen = str.length()) == 0) {  
        return true;  
    }  
    for (i = 0; i < strLen; i++) {  
        if ((Character.isWhitespace(str.charAt(i)) == false)) {  
            return false;  
        }  
    }  
    return true;  
}
```


Introduce HOF

```
public isBlank(str) {  
    if (str == null || (strLen = str.length()) == 0) {  
        return true;  
    }  
    every (ch in str) {  
        Character.isWhitespace(ch);  
    }  
    return true;  
}
```

Corner Cases Fall Away

```
public isBlank(str) {  
    every (ch in str) {  
        Character.isWhitespace(ch);  
    }  
}
```

Clojure Syntax

```
(defn blank? [s]  
  (every? #(Character/isspace %) s))
```

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: Interactive, Dynamic

Intelligent: **Declarative**

Effective

Pervasive

Destructuring

DSL for binding names

Works against *abstract* structure

Available wherever names are bound

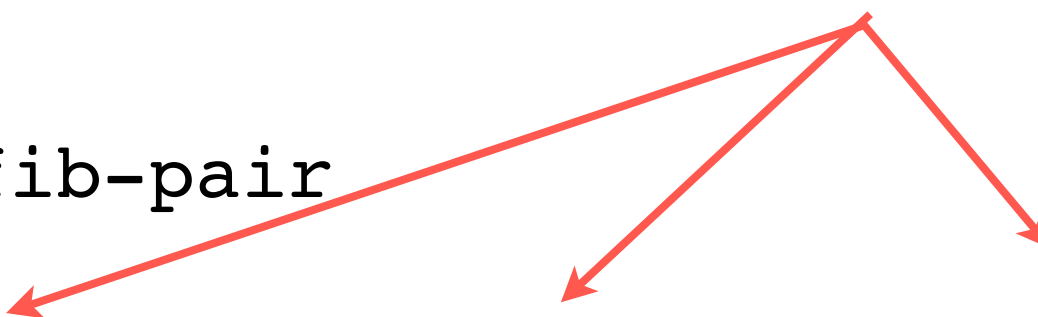
Vector form performs sequential destructure

Map form performs associative destructure

Motivation

can't see the logic for the
structure

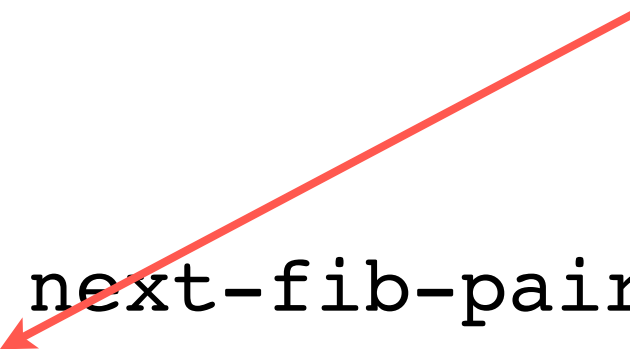
```
(defn next-fib-pair  
  [pair]  
  [(second pair) (+ (first pair) (second pair))])
```



```
(iterate next-fib-pair [0 1])  
-> ([0 1] [1 1] [1 2] [2 3] [3 5] [5 8] [8 13]...)
```

Sequential Destructure

vector form indicates
“assign a to first, b to second”

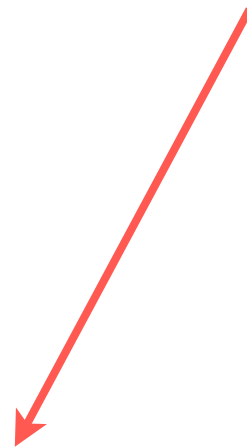


```
(defn next-fib-pair  
  [a b]  
  [b (+ a b)])
```

```
(iterate next-fib-pair [0 1])  
-> ([0 1] [1 1] [1 2] [2 3] [3 5] [5 8] [8 13] ...)
```

Sequential Destructure

inline next-fib-pair

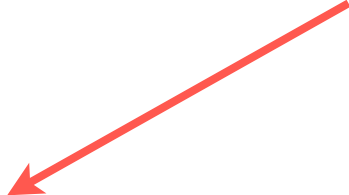


```
(defn fibs
  []
  (map first
    (iterate (fn [[a b]] [b (+ a b)]) [0 1])))
```


Associative Destructure

fn body dominated by
associative lookups

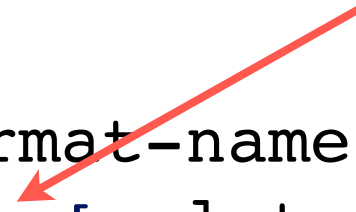
```
(defn format-name  
  [person]  
  (str/join " " [(:salutation person)  
                  (:first-name person)  
                  (:last-name person)]))
```



```
(format-name  
  { :salutation "Mr." :first-name "John" :last-name "Doe" })  
-> "Mr. John Doe"
```

Bind Names by Keyword Lookup

`{:keys [names]}` picks apart
argument by looking up keyword
names



```
(defn format-name
  [{:keys [salutation first-name last-name]}]
  (str/join " " [salutation first-name last-name]))

(format-name
  {:salutation "Mr." :first-name "John" :last-name "Doe"})
-> "Mr. John Doe"
```

Java Destructuring?

```
public String formatName(Person p)
{
    stringJoin(" ", p.getFirstName(),
               p.getMiddleName(),
               p.getLastName());
}
```

```
public String formatName
(Person {:keys [firstName, middleName, lastName]})
{
    stringJoin(" ", firstName, middleName, lastName);
}
```

Coding Inside Out?

```
{:name "Jonathan"  
 :password "secret"}
```

Coding Inside Out?

```
(assoc { :name "Jonathan"  
         :password "secret"  
         :nickname "Jon" })
```

Coding Inside Out?

```
(dissoc  
  (assoc  
    { :name "Jonathan" :password "secret" }  
    :nickname "Jon" )  
  :password)
```

Thread First ->

```
(-> { :name "Jonathan" :password "secret" }  
    (assoc :nickname "Jon")  
    (dissoc :password) )
```

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: Interactive, Dynamic

Intelligent: Declarative, Functional

Effective

Pervasive

Sequence Library

first / rest / cons

```
(first [1 2 3])  
-> 1
```

```
(rest [1 2 3])  
-> (2 3)
```

```
(cons "hello" [1 2 3])  
-> ("hello" 1 2 3)
```

take / drop

```
(take 2 [1 2 3 4 5])  
-> (1 2)
```

```
(drop 2 [1 2 3 4 5])  
-> (3 4 5)
```

Predicates

```
(every? odd? [1 3 5])
```

```
-> true
```

```
(not-every? even? [2 3 4])
```

```
-> true
```

```
(not-any? zero? [1 2 3])
```

```
-> true
```

```
(some nil? [1 nil 2])
```

```
-> true
```

Lazy and Infinite

```
(set! *print-length* 5)  
-> 5
```

```
(iterate inc 0)  
-> (0 1 2 3 4 ...)
```

```
(cycle [1 2])  
-> (1 2 1 2 1 ...)
```

```
(repeat :d)  
-> (:d :d :d :d :d ...)
```

Map / Filter / Reduce

```
(range 10)
```

```
-> (0 1 2 3 4 5 6 7 8 9)
```

```
(filter odd? (range 10))
```

```
-> (1 3 5 7 9)
```

```
(map odd? (range 10))
```

```
-> (false true false true false true  
false true false true)
```

```
(reduce + (range 10))
```

```
-> 45
```

Seqs Work Everywhere

Collections

Directories

Files

XML

JSON

ResultSets

Consuming JSON

What actors are in more than one movie currently topping the box office charts?



[http://developer.rottentomatoes.com/docs/
read/json/v10/Box_Office_Movies](http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies)

Consuming JSON

find the JSON input
download it
parse json
walk the movies
accumulating cast
extract actor name
get frequencies
sort by highest frequency



[http://developer.rottentomatoes.com/docs/
read/json/v10/Box_Office_Movies](http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies)

Consuming JSON

```
(->> box-office-uri  
      slurp  
      json/read-json  
      :movies  
      (mapcat :abridged_cast)  
      (map :name)  
      frequencies  
      (sort-by (comp - second)))
```



[http://developer.rottentomatoes.com/docs/
read/json/v10/Box_Office_Movies](http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies)

Consuming JSON

```
[ "Shiloh Fernandez" 2 ]  
[ "Ray Liotta" 2 ]  
[ "Isla Fisher" 2 ]  
[ "Bradley Cooper" 2 ]  
[ "Dwayne \"The Rock\" Johnson" 2 ]  
[ "Morgan Freeman" 2 ]  
[ "Michael Shannon" 2 ]  
[ "Joel Edgerton" 2 ]  
[ "Susan Sarandon" 2 ]  
[ "Leonardo DiCaprio" 2 ]
```



[http://developer.rottentomatoes.com/docs/
read/json/v10/Box_Office_Movies](http://developer.rottentomatoes.com/docs/read/json/v10/Box_Office_Movies)

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: Interactive, Dynamic

Intelligent: Declarative, Functional, **Logical**

Effective

Pervasive

Logic Programming

Cascalog

core.logic

Datomic Datalog

Cascalog Queries

parallel,
please

variables

```
( ??- (<- [ ?name ?age ]  
      ( people ?name ?age )  
      ( < ?age 40 ) )  
      (<- [ ?name ?age ]  
        ( people ?name ?age )  
        ( < ?age 50 ) ) )
```

dataset

constraint

<https://github.com/nathanmarz/cascalog/wiki/Defining-and-executing-queries>

core.logic

```
(defne moveo [before action after]
  ([[:middle :onbox :middle :hasnot]
    :grasp
    [:middle :onbox :middle :has]]))
([[:pos :onfloor pos has]
  :climb
  [:pos :onbox pos has]]))
([[:pos1 :onfloor pos1 has]
  :push
  [:pos2 :onfloor pos2 has]]))
([[:pos1 :onfloor box has]
  :walk
  [:pos2 :onfloor box has]]))
```

<https://github.com/clojure/core.logic/wiki/Examples>

Datomic Datalog

```
[ :find ?p2  
  :in $ %  
  :where (expensiveChocolate ?p1)  
          (relatedProduct ?p1 ?p2) ]
```

return ?p2

rules

repeated variable causes join

“Find all products related to expensive chocolate.”

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: Interactive, Dynamic, **Composable**

Intelligent: Declarative, Functional, Logical

Effective: **Value Succession Model**

Pervasive

In-Place Effects

Subprograms are machines

Programming: sticking together a bunch of moving parts

Reasonable if memory is *very* (1970s) expensive

A Better Way: References

New memories use new places

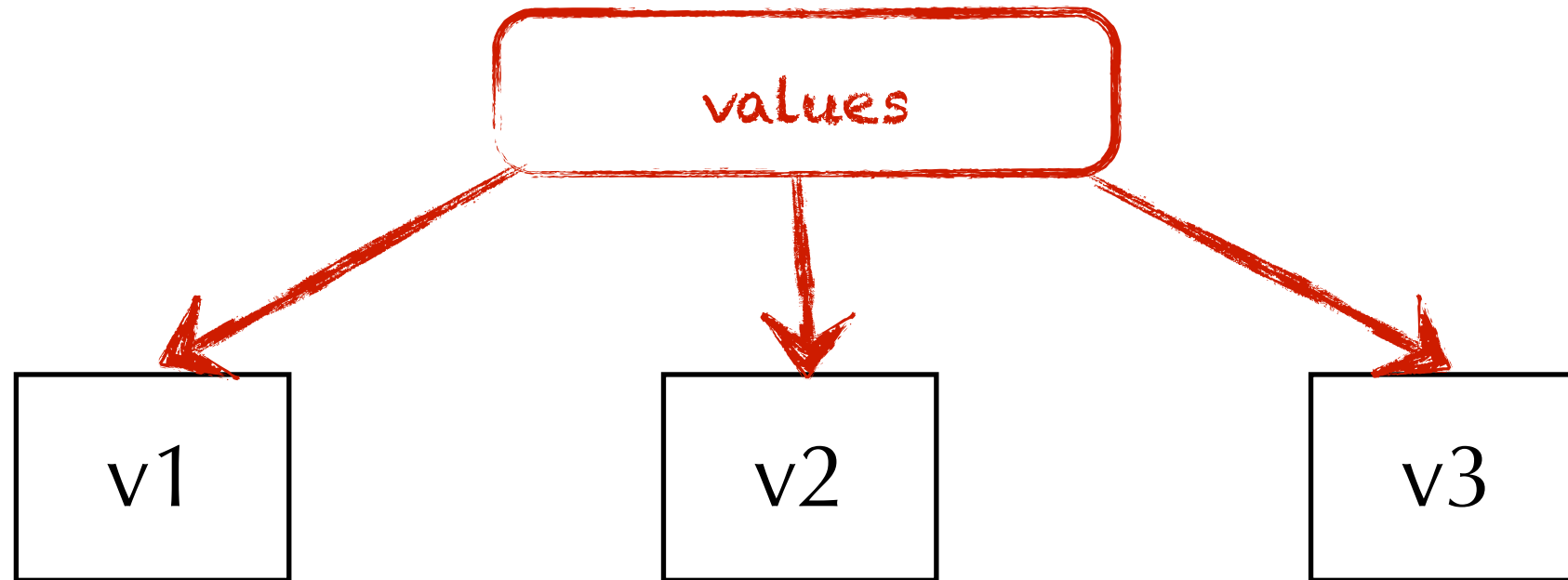
Change encapsulated by constructors

References refer to point-in-time value

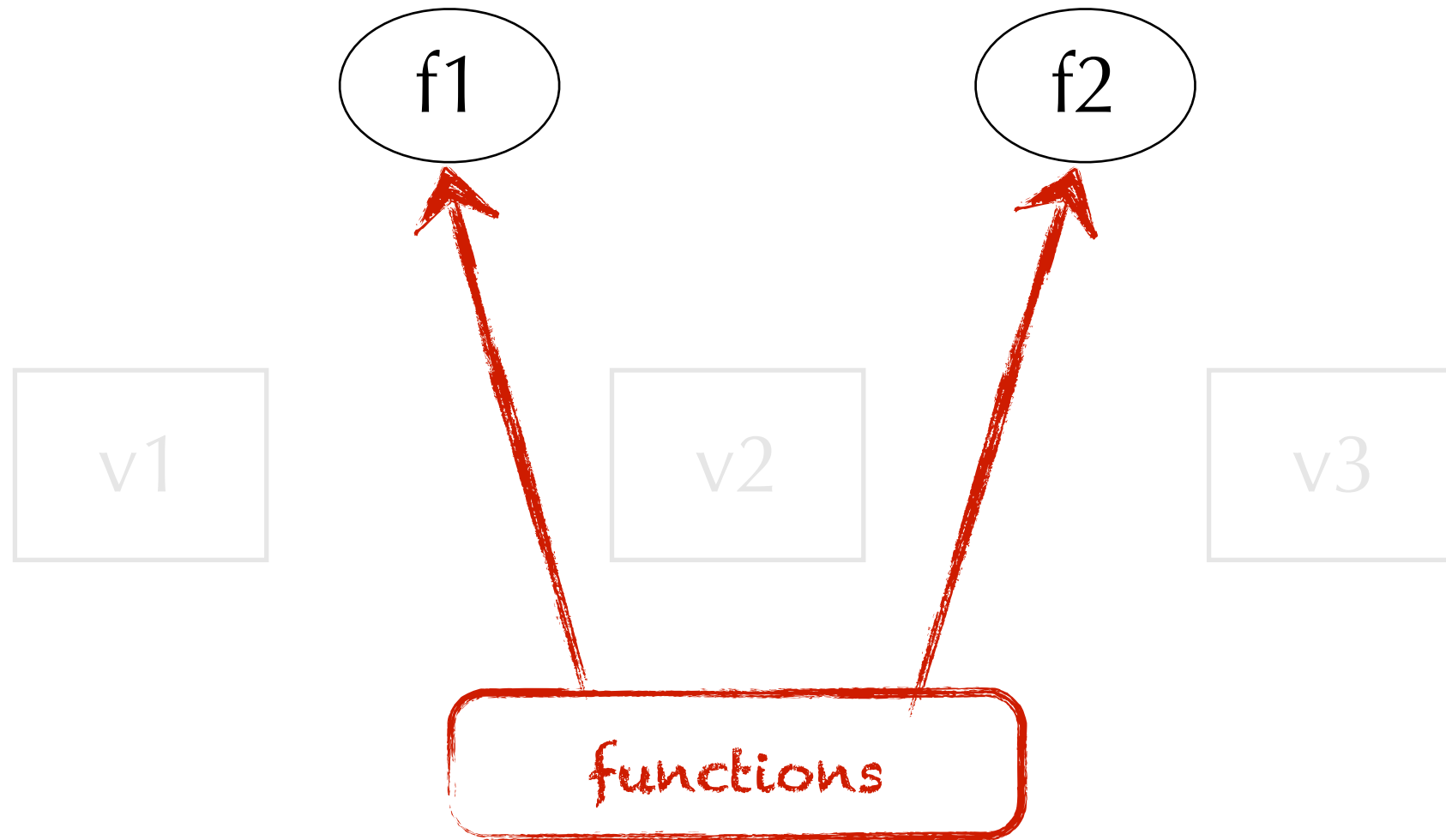
References see a *succession of values*

Compatible with many update semantics

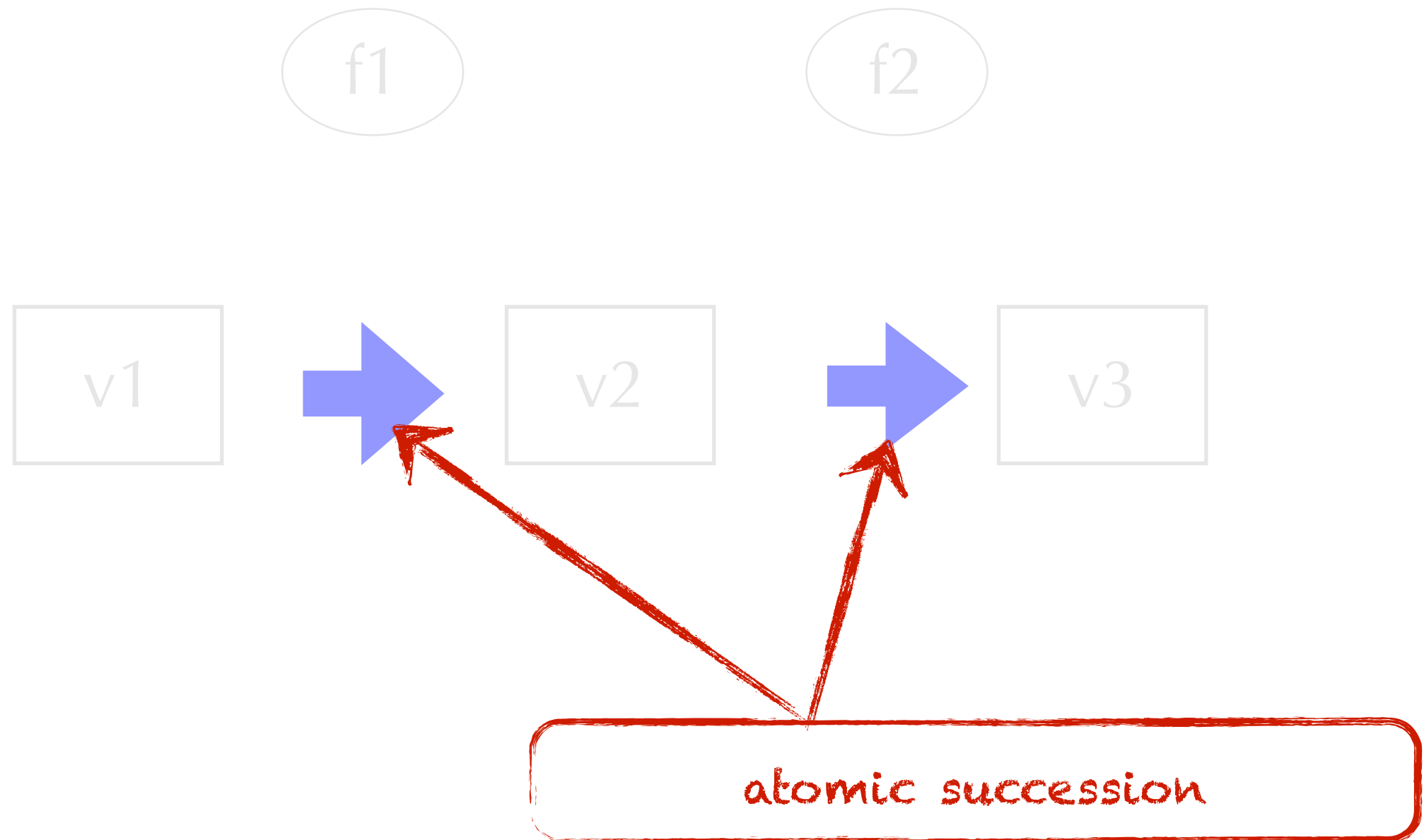
Value Succession Model



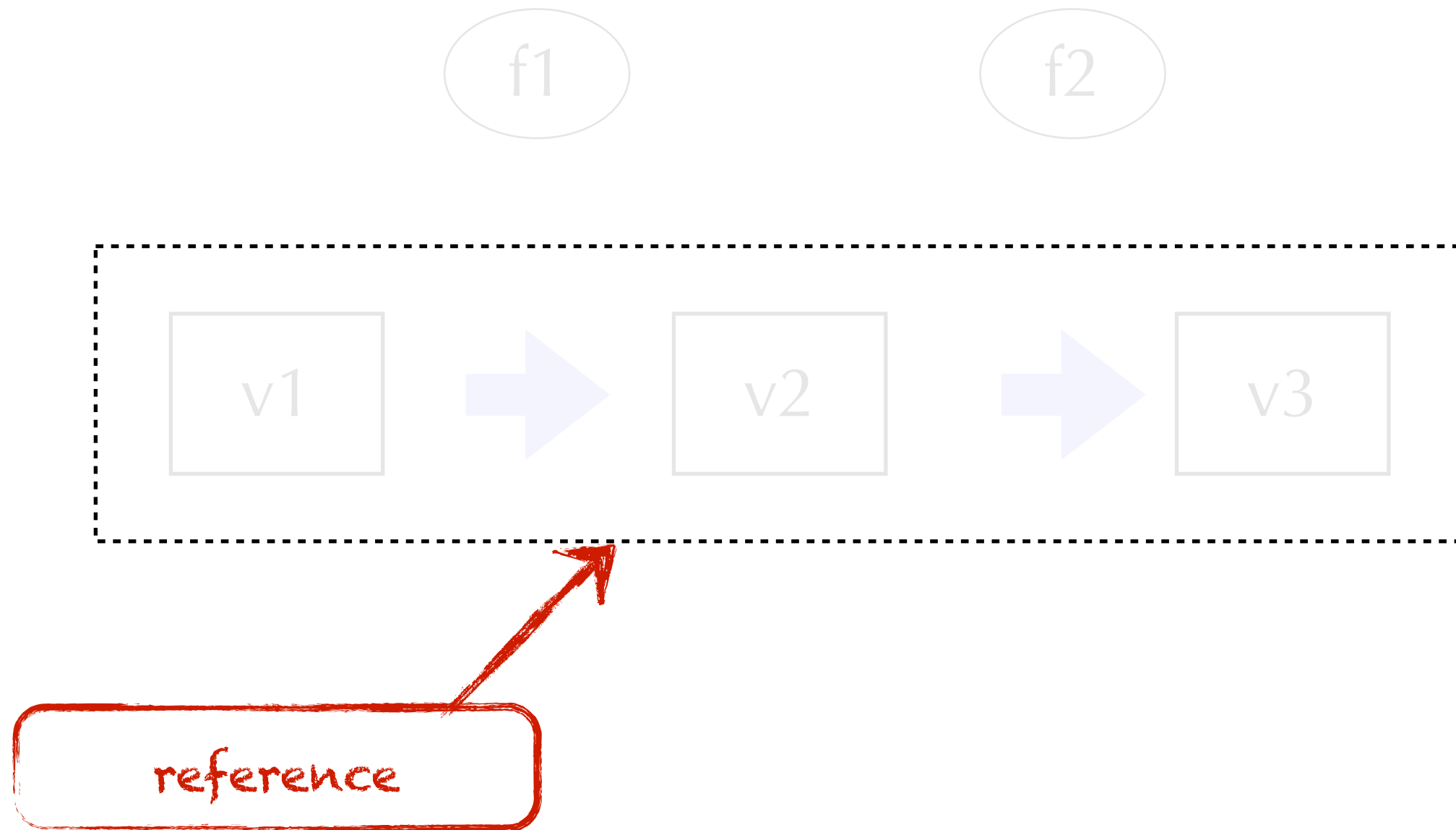
Value Succession Model



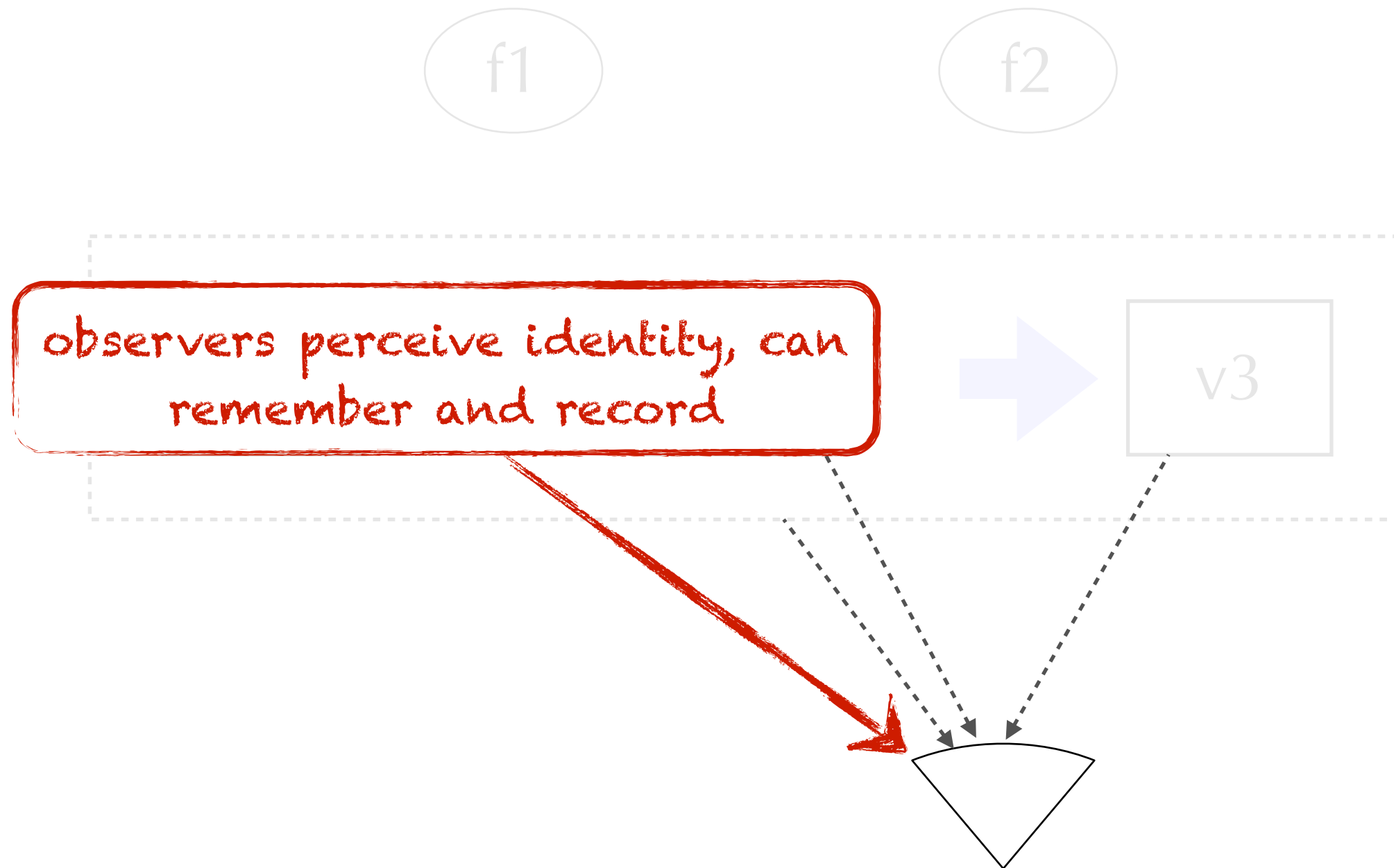
Value Succession Model



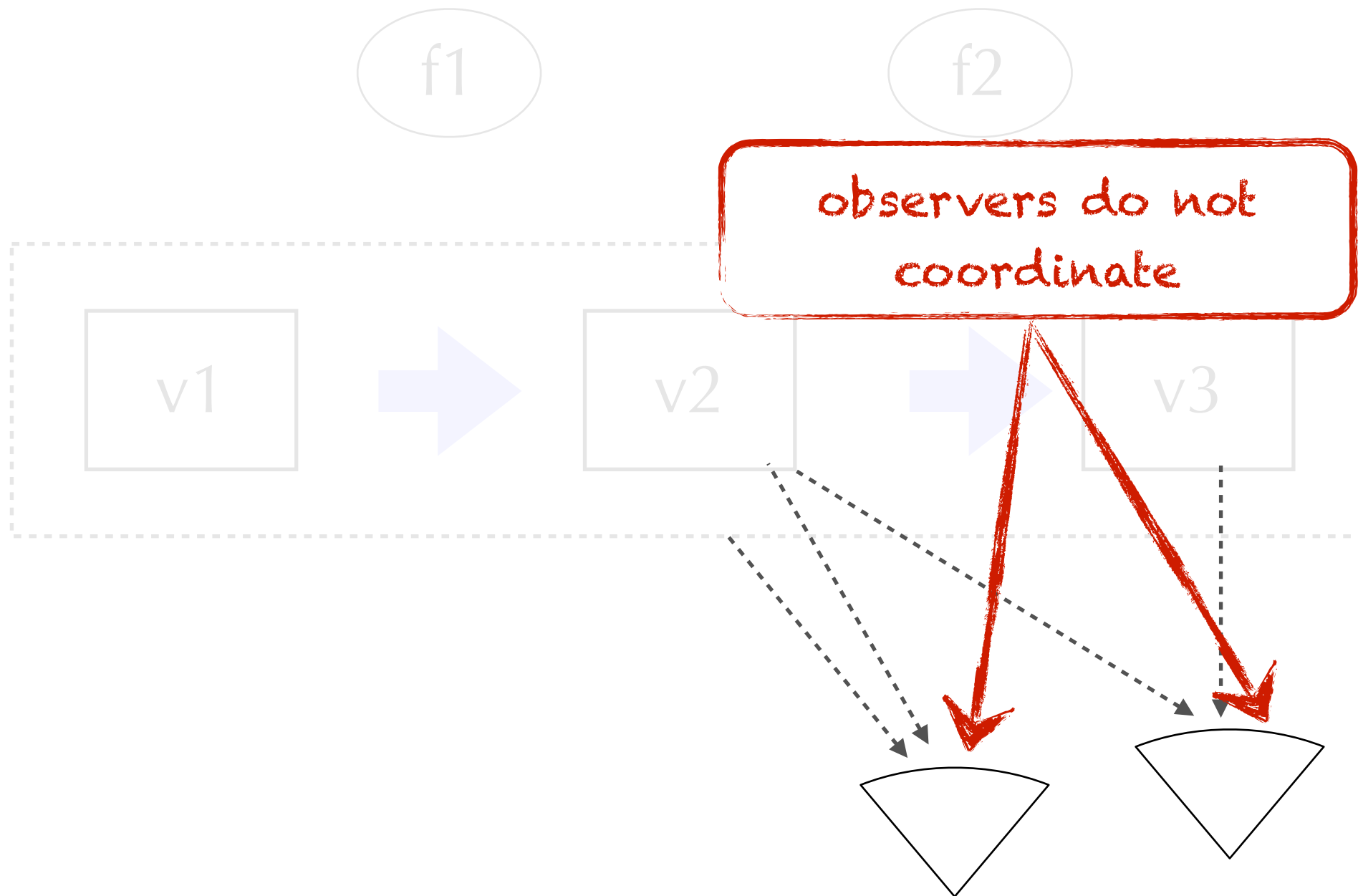
Value Succession Model



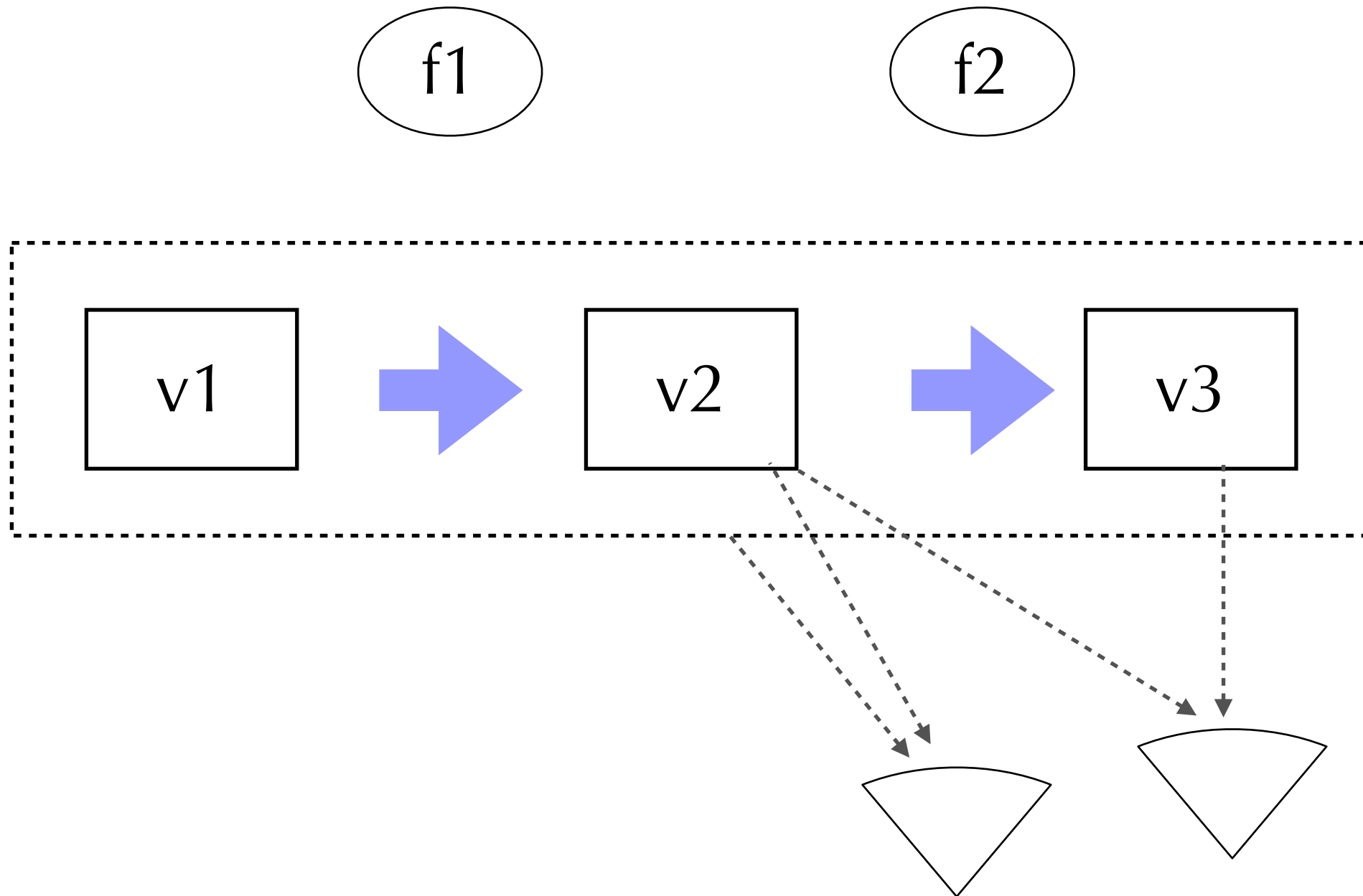
Value Succession Model



Value Succession Model



Value Succession Model




Atoms

```
(def counter (atom 0))  
(swap! counter + 10)
```

Atoms

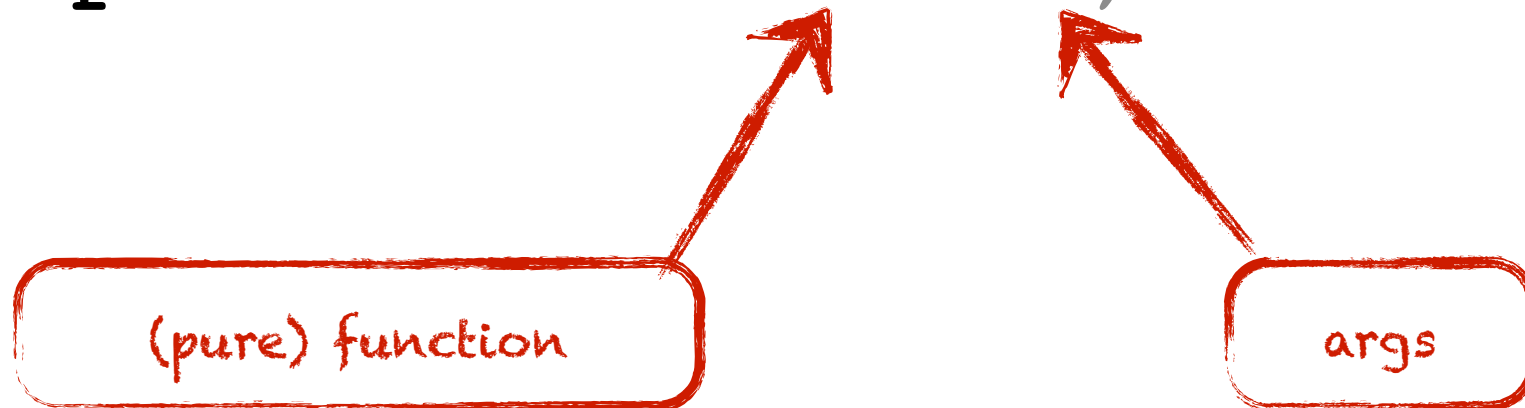
reference constructor



```
(def counter (atom 0))  
(swap! counter + 10)
```

Atoms

```
(def counter (atom 0))  
(swap! counter + 10)
```

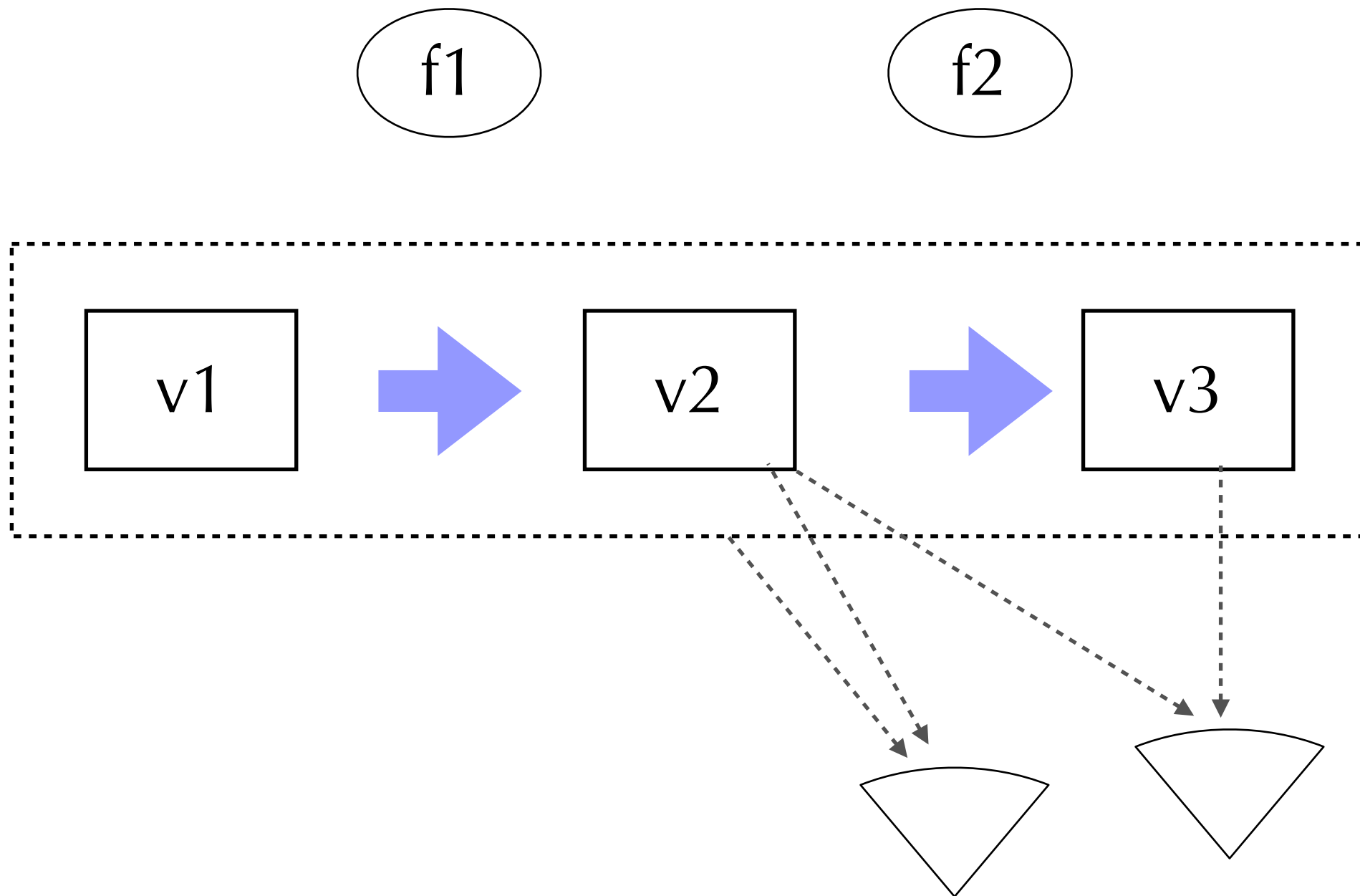


Atoms

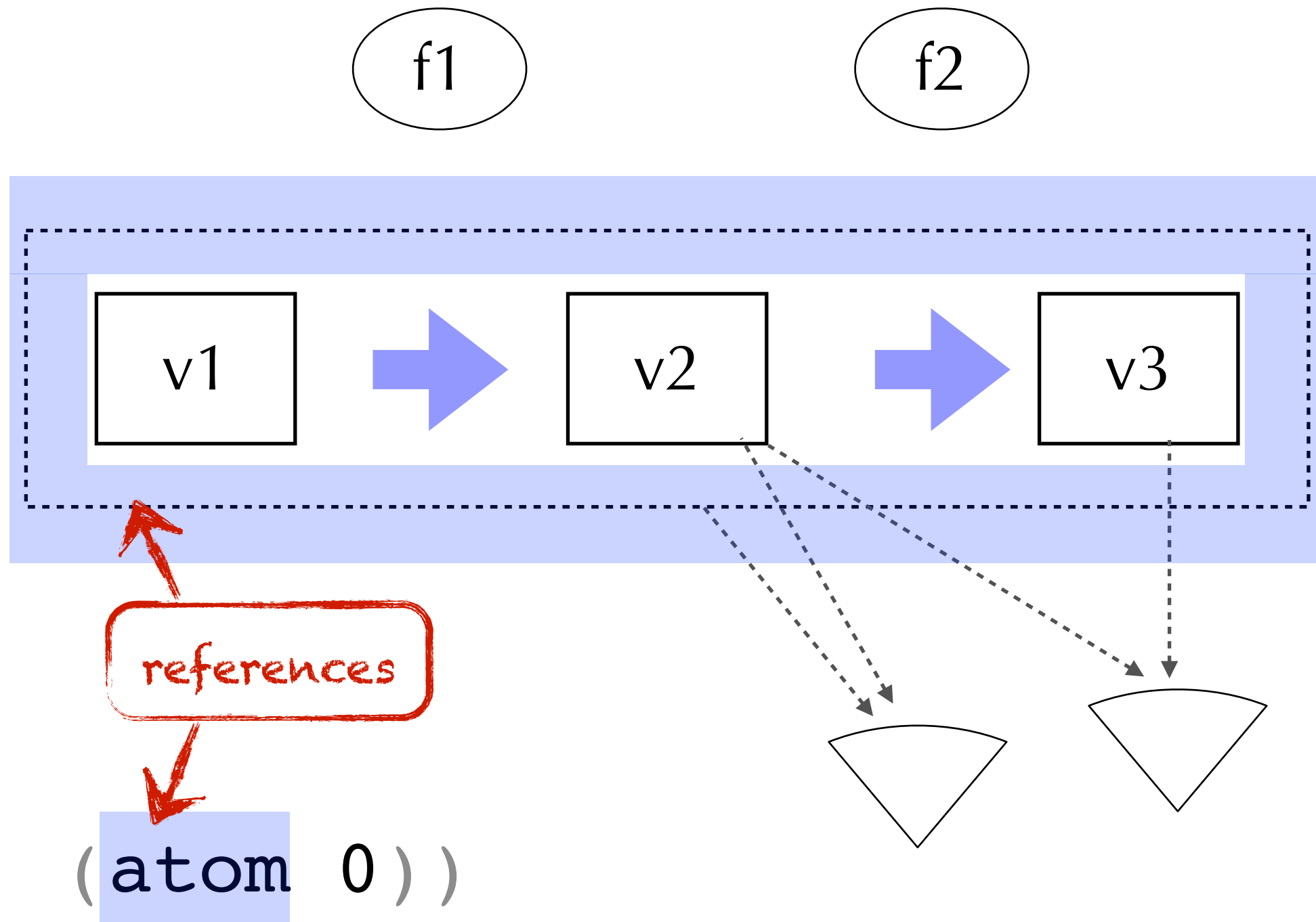
```
(def counter (atom 0))  
(swap! counter + 10)
```



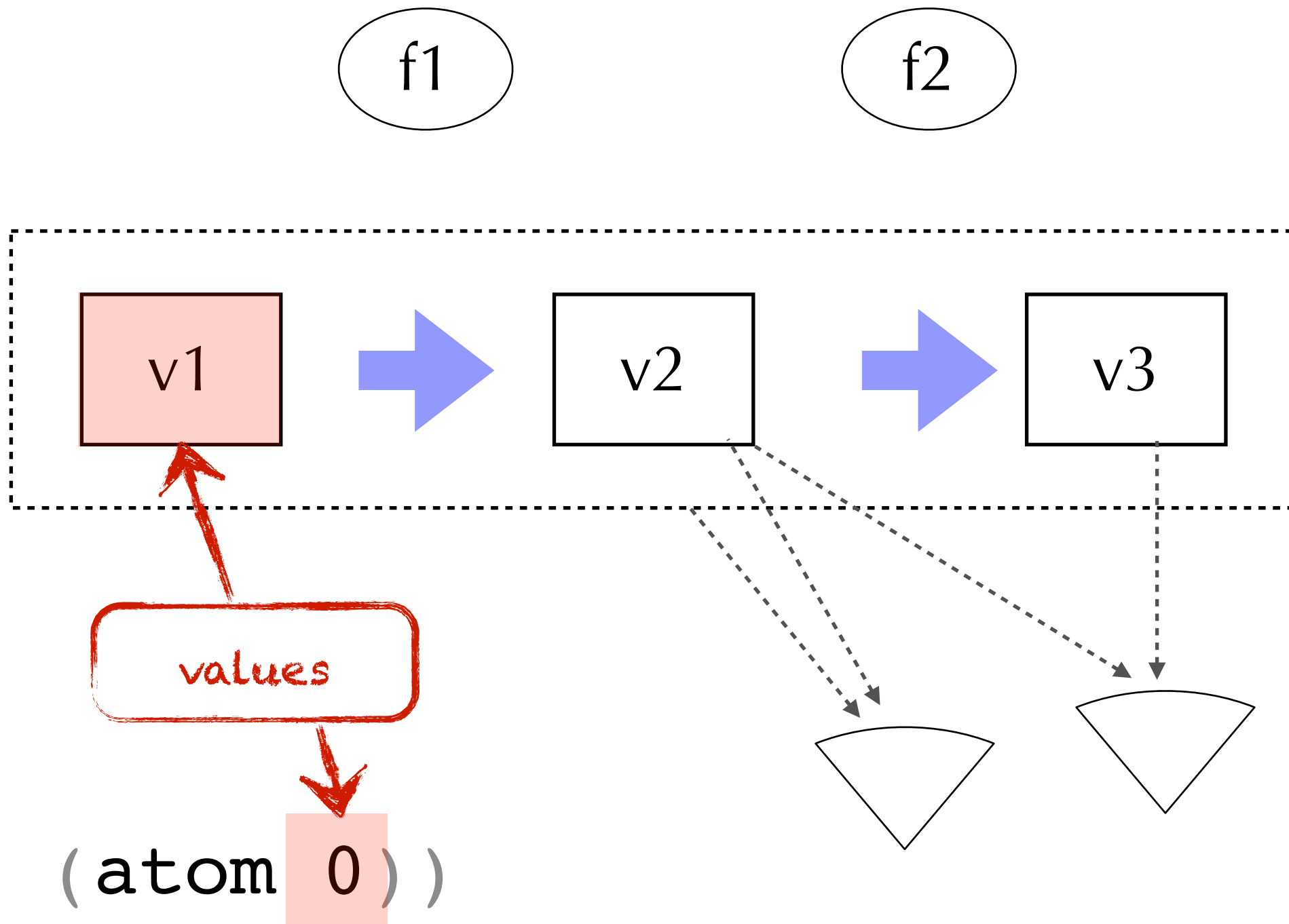
Atoms



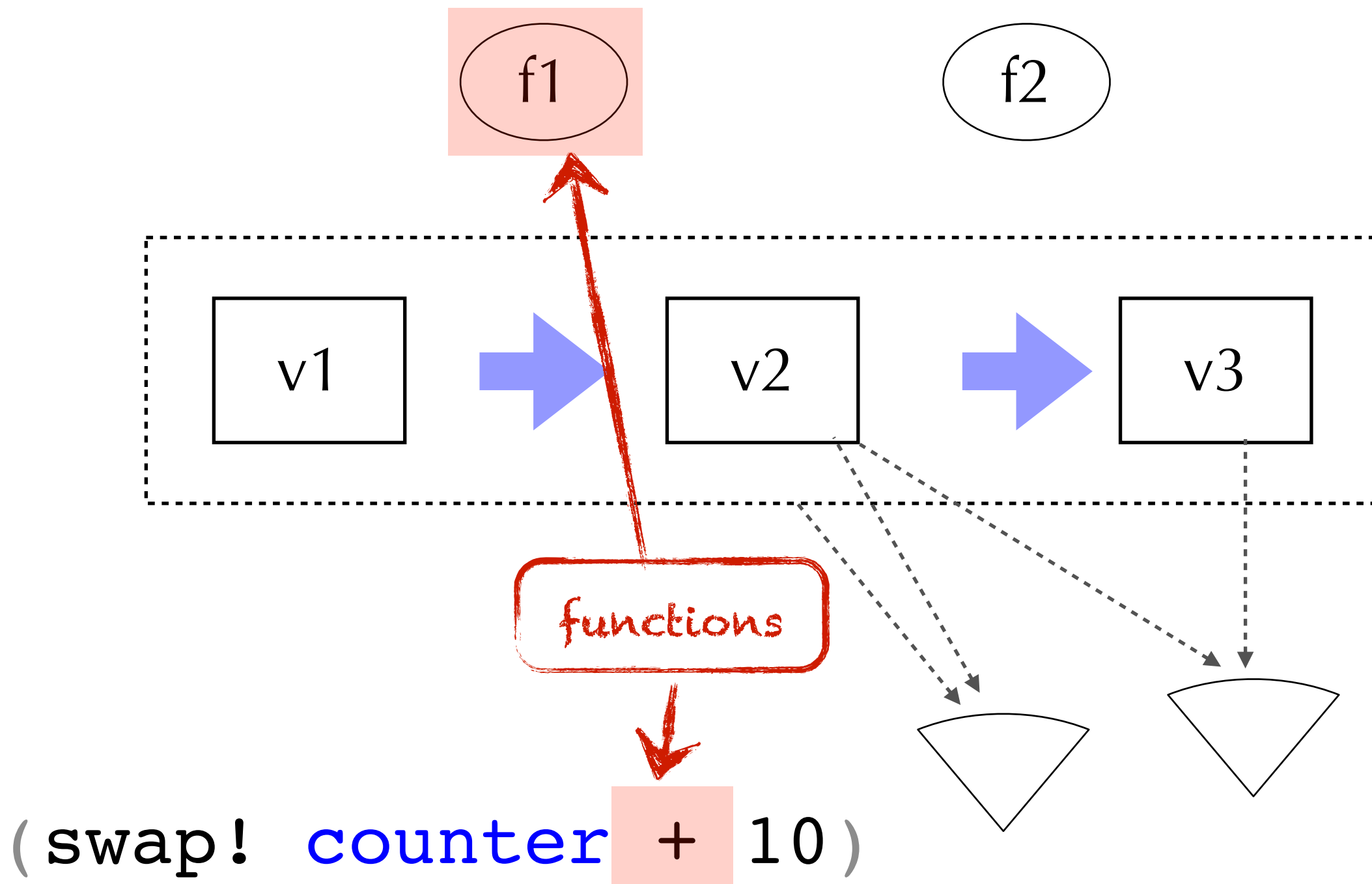
Atoms



Atoms



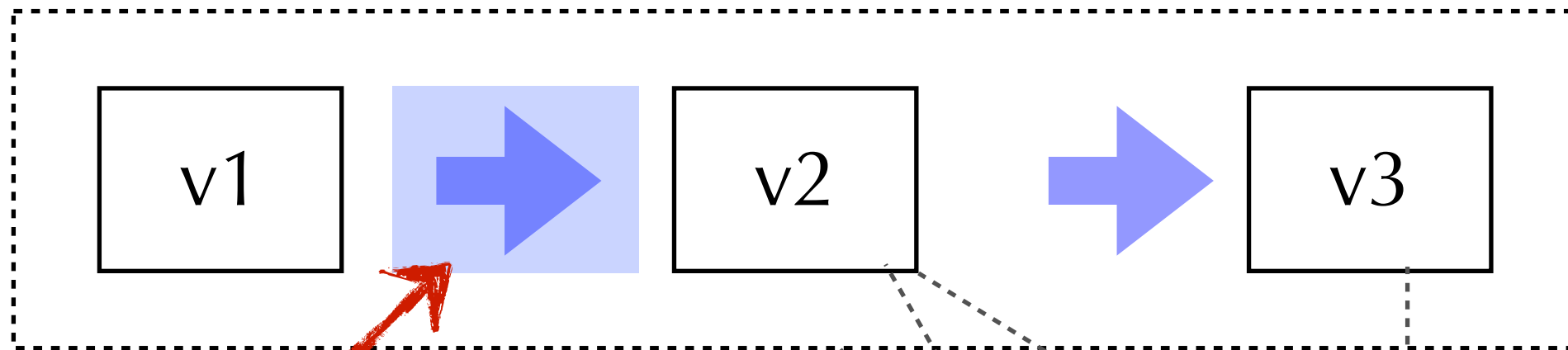
Atoms



Atoms

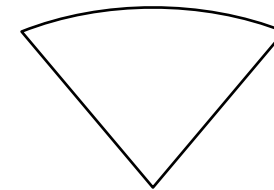
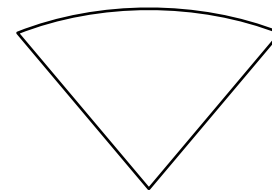
f1

f2



succession

(swap! counter + 10)



Bigger Structures

different data

```
(def person (atom (create-person)))  
(swap! person assoc :name "John")
```

same ref type
and succession fn

Varying Semantics

different kind of ref



```
(def number-later (promise))  
(deliver number-later 42)
```

different succession



Entire Database

```
(def conn (d/connect uri)  
(transact conn data)
```

Entire Database

```
(def conn (d/connect uri))  
(transact conn data)
```



agent →

send	processor-derived pool
send-off	IO-derived pool
send-via	user-specified pool

atom ⇔

compare-and-set!	conditional
reset!	boring
swap!	functional transformation

connection 

transact	⇔	ACID
transact-async	→	ACID

ref ⇔

alter	functional transformation
commute	commutative

var ⇔

alter-var-root	application config
----------------	--------------------

var binding ⇔

binding, set!	dynamic, binding-local
---------------	------------------------

Software Should Be

Knowledgeable: Persistent Data Structures

Powerful: Full Platform Access

Flexible: Interactive, Dynamic, Composable

Intelligent: Declarative, Functional, Logical

Effective: Value Succession Model

Pervasive: **Target Mainstream Platforms**

The Mainstream

Clojure (JVM)

ClojureScript (JavaScript)

ClojureCLR (.NET)

Conclusions

Transient data structures don't compose

Clojure solves this with *persistence* and *value succession*

Better semantics allow powerful, flexible programs

Clojure semantics available where you need them

Resources

Clojure

<http://clojure.com>. The Clojure language.

<http://tryclj.com/>. Try Clojure.

<http://hимерa.herokuapp.com>. Try ClojureScript.

<http://thinkrelevance.com/blog/tags/podcast>. The Relevance Podcast.

<http://www.datomic.com/>. Datomic.

<http://clojure.in/>. Planet Clojure.

<http://pragprog.com/book/shcloj2/programming-clojure>. *Programming Clojure*.

Stuart Halloway

<https://github.com/stuarthalloway/presentations/wiki>. Presentations

<http://www.linkedin.com/pub/stu-halloway/0/110/543/>

<https://twitter.com/stuarthalloway>

<mailto:stu@thinkrelevance.com>