

A Formal Model of Tutuca

Components, views, and the binding that lowers them to a virtual DOM

This document specifies the meaning of tutuca as a PLT Redex model. Tutuca's side is declarative, in three parts. Components are classes: typed fields, invariants, and a protocol of messages whose handlers are deterministic steps from one instance to the next, in the style of TLA+. Views evaluate to a semantic tree that says which parts are the render of a component, which are rendered at a frame, and which are the items of a loop; events on the tree route to messages. The host composes them into a task queue.

A binding layer lowers the semantic tree to virtual DOM nodes, encoding wrappers as metadata comments and handlers as data-eid attributes, and decodes events back from the DOM. The binding theorem checks that decoding agrees with routing on the semantic tree. The virtual DOM itself, with reconciliation and node identity, knows nothing about tutuca and is specified in Appendix A.

Conventions. A metafunction undefined on its input stands for a thrown JS exception. The optimizations in `src/optimize.js` and the fast paths in `vdom.js` are erased; they must preserve the meaning given here. Shaded fragments inside a rule are Racket code. The rules are executable: `raco test spec/` checks them against scenarios ported from the JS test suite and against random properties.

Contents

1 Values	3
1.1 Runtime values	3
1.2 Property access	4
2 Paths	5
2.1 Kinds of path	5
2.2 Resolving	6
2.3 sourceKey and frames	7
2.4 Selectors	8
3 Components	9
3.1 Expressions	9
3.2 Field types	10
3.3 Classes and protocols	13
3.4 Actions and checks	13
4 Views	14
4.1 The semantic tree	14
4.2 View syntax	16
4.3 Registration	17
4.4 Render contexts and view selection	18
4.5 Evaluation	20
4.6 Routing on the tree	23
4.7 Well-formed views	24
5 Binding	25
5.1 Metadata	25
5.2 Lowering	26
5.3 Decoding events	27
6 The host	29
6.1 Host configurations	29
6.2 The task queue	30
6.3 Settled states	30
A Virtual DOM	31
A.1 Normalized vnodes	31
A.2 DOM output	33
A.3 Keyed sibling matching	34
A.4 Labeled DOM and mounting	36
A.5 Morphing	37

Values

The values views read and message handlers write. State is a value, a component instance is a value, and the effect of a handler is a new value, so everything later is built on this chapter.

1.1 Runtime values

Atoms, null (which also stands for undefined), arrays, records and components. A component instance is a record tagged with its class name. These are the only objects: JS functions appear later as expression terms (Chapter 3).

atom ::= *string* | *number* | *boolean*
fname ::= (variable-except null)
cname ::= (variable-except null)
value ::= *atom* | null
 | (arr *value* ...)
 | (rec (*fname value*) ...)
 | (comp *cname (fname value)* ...)
pkey ::= *natural* | *string* | *fname*

Figure 1. Runtime values: atoms, null (also undefined), arrays, records and components.

1.2 Property access

Reading and writing one property, as immer drafts see it. A missing property reads as null; reading a property of null has no rule, which is the `TypeError`. Writes never mutate: they build a new array, record or component. `entries-of` gives the pairs a loop iterates over.

prop-get: $v[k]$; $null[k]$ has no rule (`TypeError`)

```
prop-get[(arr value ...), natural] = nth((value ...), natural)
  where (< natural length((value ...)))
prop-get[(arr value ...), any] = null
prop-get[(rec anyf ...), pkey] = field-get[(anyf ...), key(pkey)]
prop-get[(comp cname anyf ...), pkey] = field-get[(anyf ...), key(pkey)]
prop-get[(rec anyf ...), any] = null
prop-get[(comp cname anyf ...), any] = null
prop-get[(atom, any)] = null
```

field-get

```
field-get[(any1 ... (fname value) any2 ...), fname] = value
field-get[(anyfs, any)] = null
```

prop-set: $v[k] = x$

```
prop-set[(arr value ...), natural, valuenew] =
(arr (list-set (value ...) natural valuenew))
  where (< natural length((value ...)))
prop-set[(arr value ...), natural, valuenew] =
(arr value ... valuenew)
  where (= natural length((value ...)))
prop-set[(rec anyf ...), pkey, valuenew] =
(rec anyr ...)
  where (anyr ...) = field-set[(anyf ...), key(pkey), valuenew]
prop-set[(comp cname anyf ...), pkey, valuenew] =
(comp cname anyr ...)
  where (anyr ...) = field-set[(anyf ...), key(pkey), valuenew]
```

field-set

```
field-set[(any1 ... (fname valueold) any2 ...), fname, value] =
(any1 ... (fname value) any2 ...)
field-set[(anyf ...), fname, value] =
(anyf ... (fname value))
```

entries-of: array entries or `Object.entries`

```
entries-of[(arr value ...)] =
(for/list ([v (value ...)] [i (in-naturals)]) (list i v))
entries-of[(rec (fname value) ...)] =
(for/list ([f (fname ...)] [v (value ...)]) (list (symbol->string f) v))
entries-of[(comp cname anyf ...)] =
entries-of[(rec (filter (λ (f) (not (eq? (car f) @))) (anyf ...)))]
entries-of[(atom)] =
()
```

Figure 2. Property access, as immer drafts see it: writes build new values.

Paths

How places are named. There are four kinds of path, one per job: update paths address state, dispatch paths record rendered ancestry, selectors describe targets read at commit time, and routes pair a dispatch path with its update path. A route is where a message is delivered, so paths come before components and views.

2.1 Kinds of path

The syntax of frames, dispatch paths, update paths, selectors and routes. A frame reads its keys, then its item key, from its base: the previous frame's value, or the value a provider captured at some render depth. `get-in` and `set-in` are `getIn` and `setIn`, the only operations that touch state through an update path.

```

key ::= natural | string
base ::= ∅ | (prov fname natural (skey ...))
item? ::= skey | ∅
dframe ::= (frame base (skey ...) item? boolean)
dpath ::= (dframe ...)
upath ::= (skey ...)
route ::= (dpath upath)
pb-op ::= (pb-field fname) | (pb-item fname skey) | (pb-seq-item fname fname)
sel ::= (pb-op ...)

```

access-keys: what a frame reads from its base

```

access-keys[(frame base (skey ...) ∅ boolean)] = (skey ...)
access-keys[(frame base (skey ...) skeyi boolean)] = (skey ... skeyi)

```

get-in: getIn on update paths

```

get-in[((), value)] = value
get-in[(skey0 skey ...), value] = get-in[(skey ...), prop-get[value, skey0]]

```

set-in: setIn on update paths

```

set-in[((), valueroot, value] =
value
set-in[(skey0 skey ...), valueroot, value] =
prop-set[valueroot, skey0, set-in[(skey ...), prop-get[valueroot, skey0], value]]

```

Figure 3. Update paths, dispatch frames, selectors and routes are different things.

2.2 Resolving

From a dispatch path to the state keys it stands for, frame by frame from left to right. A provider frame restarts at the end of the frames up to its capture depth, followed by its suffix. to-component-path drops trailing frames that are not component frames, so an event inside a loop item belongs to the component whose view has the loop; route-for pairs a dispatch path with its update path.

resolve

$\text{resolve}[\![dpath]\!] = \text{last}(\text{resolve-from}[\![dpath, (())]\!])$

resolve-from

$\text{resolve-from}[\![(\text{frame } \emptyset \text{ any}_k \text{ item? boolean}) dframe \dots], (upath \dots)] =$
 $(upath \dots)$

$\text{resolve-from}[\![(\text{frame } \emptyset \text{ any}_k \text{ item? boolean}) dframe \dots], (upath \dots)] =$

$\text{resolve-from}[\![dframe \dots], (upath \dots (skey_p \dots skey \dots))]\]$

where $(skey_p \dots) = \text{last}((upath \dots))$,

$(skey \dots) = \text{access-keys}[\![(\text{frame } \emptyset \text{ any}_k \text{ item? boolean})]\!]$

$\text{resolve-from}[\![(\text{frame } (\text{prov } fframe \text{ natural } (skey_s \dots)) \text{ any}_k \text{ item? boolean}) dframe \dots], (upath \dots)] =$

$\text{resolve-from}[\![dframe \dots], (upath \dots (skey_p \dots skey_s \dots skey \dots))]\]$

where $(skey_p \dots) = \text{nth}((upath \dots), \text{natural})$,

$(skey \dots) = \text{access-keys}[\![(\text{frame } \emptyset \text{ any}_k \text{ item? boolean})]\!]$

to-component-path

$\text{to-component-path}[\![dframe \dots (\text{frame } base \text{ any}_k \text{ item? \#t})]\!] =$
 $(dframe \dots (\text{frame } base \text{ any}_k \text{ item? \#t}))$

$\text{to-component-path}[\![dframe \dots dframe_{last}]\!] =$

$\text{to-component-path}[\![dframe \dots]\!]$

$\text{to-component-path}[\![]] =$

$()$

to-update-path

$\text{to-update-path}[\![dpath]\!] = \text{resolve}[\![\text{to-component-path}[\![dpath]\!]]\!]$

route-for

$\text{route-for}[\![dpath]\!] = (dpath \text{ to-update-path}[\![dpath]\!])$

Figure 4. Resolving: frames to keys; a provider jump restarts at its capture depth.

2.3 sourceKey and frames

sourceKey is the item key of the frame that leads out of the receiver's view, or null when the receiver itself or an ancestor sent the message. promote marks the frame before a component boundary as a component frame, and item-frame is the frame a loop item adds over its source.

source-key-in

```
source-key-in[(dframe ...), (dframer ...)] = null
where (or (>= length((dframer ...)) length((dframe ...)))
      (not (frames-prefix? (dframer ...) (dframe ...))))
source-key-in[dpath, (dframer ...)] = key-at[dpath, length((dframer ...))]
```

key-at

```
key-at[dpath, natural] =
key-at[dpath, succ(natural)]
where (< natural (sub1 length(dpath))),
      (frame base anyk item? #f) = nth(dpath, natural)
key-at[dpath, natural] =
frame-item-key[nth(dpath, natural)]
```

frame-item-key

```
frame-item-key[(frame base anyk ∅ boolean)] = null
frame-item-key[(frame base anyk skey boolean)] = skey
```

promote: a component boundary follows the frame

```
promote[()] =
()
promote[(dframe ... (frame base anyk item? boolean))] =
(dframe ... (frame base anyk item? #t))
```

item-frame: a loop item over a source frame

```
item-frame[(frame base anyk item? boolean), skey] =
(frame base access-keys[(frame base anyk item? boolean)] skey #f)
```

Figure 5. sourceKey, component frames and item frames.

2.4 Selectors

A selector is a sequence of path builder calls (`field`, `item`, `seqItem`). It freezes at commit, one call at a time, against the value it starts from, and the frames it freezes to extend its base. Checked: freezing below a base equals resolving the base with the frozen keys appended.

freeze-op

```
freeze-op[(pb-field fname), value] =  
(frame Ø ((symbol->string fname)) Ø #t)  
freeze-op[(pb-item fname skey), value] =  
(frame Ø ((symbol->string fname)) skey #t)  
freeze-op[(pb-seq-item fname fnamek), value] =  
(frame Ø ((symbol->string fname)) item-of[prop-get[value, fnamek]] #t)
```

freeze-sel

```
freeze-sel[()], value] =  
()  
freeze-sel[(pb-opo pb-op ...), value] =  
(dframeo dframe ...)  
  where dframeo = freeze-op[pb-opo, value],  
        (dframe ...) = freeze-sel[(pb-op ...), get-in[access-keys[dframeo], value]]
```

Checked: `resolve(base ++ freeze-sel(sel, get-in(resolve(base), root)))`
= `resolve(base) ++ the frozen keys`.

Figure 6. Selectors freeze against the values they read.

Components

The model half of `component()`: what state an instance holds and which messages it handles. A class has typed fields, invariants, and a protocol: each message has a contract and a handler. Instances are values, with no identity and no inheritance; where an instance lives is its only identity. A handler is an action in the style of TLA+, a deterministic step from self to next together with the effects it emits.

3.1 Expressions

The pure JavaScript a component supplies, as a small language. Field predicates, invariants, contracts, handler bodies, loop hooks and scope binds are all terms in it. A throw is an undefined evaluation.

$x ::= \text{variable}$	
$\text{env} ::= ((x \text{ value}) \dots)$	
$\text{hx} ::= \text{value} \mid x$	
$\mid (\text{get } \text{hx } \text{fname}) \mid (\text{at } \text{hx } \text{hx})$	
$\mid (= \text{hx } \text{hx}) \mid (< \text{hx } \text{hx}) \mid (\text{not } \text{hx}) \mid (\text{and } \text{hx } \dots) \mid (\text{if } \text{hx } \text{hx } \text{hx}) \mid (+ \text{hx } \text{hx})$	
$\mid (\text{push } \text{hx } \text{hx})$	
$\mid (\text{mk-rec } (\text{fname } \text{hx}) \dots) \mid (\text{mk-arr } \text{hx } \dots)$	
eval-hx	
$\text{eval-hx}[\text{env}, \text{value}]$	$= \text{value}$
$\text{eval-hx}[(\text{any}_1 \dots (x \text{ value}) \text{any}_2 \dots), x]$	$= \text{value}$
where $(\text{not } (\text{assq } x (\text{any}_1 \dots)))$	
$\text{eval-hx}[\text{env}, (\text{get } \text{hx } \text{fname})]$	$= \text{prop-get}[\text{eval-hx}[\text{env}, \text{hx}], \text{fname}]$
$\text{eval-hx}[\text{env}, (\text{at } \text{hx}_1 \text{hx}_2)]$	$= \text{prop-get}[\text{eval-hx}[\text{env}, \text{hx}_1], \text{eval-hx}[\text{env}, \text{hx}_2]]$
$\text{eval-hx}[\text{env}, (= \text{hx}_1 \text{hx}_2)]$	$= (\text{equal? } \text{eval-hx}[\text{env}, \text{hx}_1] \text{eval-hx}[\text{env}, \text{hx}_2])$
$\text{eval-hx}[\text{env}, (< \text{hx}_1 \text{hx}_2)]$	$= (< \text{number}_1 \text{number}_2)$
where $\text{number}_1 = \text{eval-hx}[\text{env}, \text{hx}_1], \text{number}_2 = \text{eval-hx}[\text{env}, \text{hx}_2]$	
$\text{eval-hx}[\text{env}, (\text{not } \text{hx})]$	$= (\text{not } \text{truthy?}(\text{eval-hx}[\text{env}, \text{hx}]))$
$\text{eval-hx}[\text{env}, (\text{and})]$	$= \#t$
$\text{eval-hx}[\text{env}, (\text{and } \text{hx}_0 \text{hx } \dots)]$	$= \text{eval-hx}[\text{env}, (\text{and } \text{hx } \dots)]$
where $\text{truthy?}(\text{eval-hx}[\text{env}, \text{hx}_0])$	
$\text{eval-hx}[\text{env}, (\text{and } \text{hx}_0 \text{hx } \dots)]$	$= \text{eval-hx}[\text{env}, \text{hx}_0]$
$\text{eval-hx}[\text{env}, (\text{if } \text{hx}_c \text{hx}_t \text{hx}_e)]$	$= \text{eval-hx}[\text{env}, \text{hx}_t]$
where $\text{truthy?}(\text{eval-hx}[\text{env}, \text{hx}_c])$	
$\text{eval-hx}[\text{env}, (\text{if } \text{hx}_c \text{hx}_t \text{hx}_e)]$	$= \text{eval-hx}[\text{env}, \text{hx}_e]$
$\text{eval-hx}[\text{env}, (+ \text{hx}_1 \text{hx}_2)]$	$= (+ \text{number}_1 \text{number}_2)$
where $\text{number}_1 = \text{eval-hx}[\text{env}, \text{hx}_1], \text{number}_2 = \text{eval-hx}[\text{env}, \text{hx}_2]$	
$\text{eval-hx}[\text{env}, (\text{push } \text{hx}_1 \text{hx}_2)]$	$= (\text{arr value } \dots \text{eval-hx}[\text{env}, \text{hx}_2])$
where $(\text{arr value } \dots) = \text{eval-hx}[\text{env}, \text{hx}_1]$	
$\text{eval-hx}[\text{env}, (\text{mk-rec } (\text{fname } \text{hx}) \dots)]$	$= (\text{rec } (\text{fname } \text{eval-hx}[\text{env}, \text{hx}]) \dots)$
$\text{eval-hx}[\text{env}, (\text{mk-arr } \text{hx } \dots)]$	$= (\text{arr } \text{eval-hx}[\text{env}, \text{hx}] \dots)$

Figure 7. Expressions.

3.2 Field types

The `t.*` builders of `schema.js` as terms. `valid` is `isValid`: the value has the type's kind and meets its constraints. An `index` is a key of another field of the owner, the component holding the field; inside arrays, maps and records there is no owner. `make-field` keeps a valid argument and otherwise takes the default, which must meet the constraints; a record makes each key, and a union its first alternative, the same way. `Checked`: `make` keeps valid arguments, and yields a valid value whenever every `.default(v)` has its type.

```
ty ::= Any | String | Number | Boolean | (Enum value ...) | (Option ty) | (Array ty) | (Map ty)
      | (Record (fname ty) ...) | (OneOf ty ...) | (Index fname) | (Component cname ...)
      | (Refine ty check ...) | (Default ty value)
check ::= int | (min number) | (max number) | (len natural) | (len natural natural) | (pred x hx)
owner ::= value |  $\emptyset$ 
arg? ::= value | absent
```

Figure 8. Field types, constraints and owners.

$\frac{}{\text{valid}[\text{Any}, \text{value}, \text{owner}]} [\text{any}]$	
$\frac{\text{isString}(\text{value})}{\text{valid}[\text{String}, \text{value}, \text{owner}]} [\text{string}]$	
$\frac{\text{isFinite}(\text{value})}{\text{valid}[\text{Number}, \text{value}, \text{owner}]} [\text{number}]$	
$\frac{\text{isBoolean}(\text{value})}{\text{valid}[\text{Boolean}, \text{value}, \text{owner}]} [\text{boolean}]$	
$\frac{(\text{member } \text{value} \text{ } (value_m \dots))}{\text{valid}[(\text{Enum } value_m \dots), \text{value}, \text{owner}]} [\text{enum}]$	
$\frac{}{\text{valid}[(\text{Option } ty), \text{null}, \text{owner}]} [\text{option-null}]$	
$\frac{\text{valid}[ty, \text{value}, \text{owner}]}{\text{valid}[(\text{Option } ty), \text{value}, \text{owner}]} [\text{option}]$	
$\frac{\text{valid}[ty, \text{value}, \emptyset] \dots}{\text{valid}[(\text{Array } ty), (\text{arr } \text{value} \dots), \text{owner}]} [\text{array}]$	
$\frac{\text{valid}[ty, \text{value}, \emptyset] \dots}{\text{valid}[(\text{Map } ty), (\text{rec } (fname \text{ value}) \dots), \text{owner}]} [\text{map}]$	
$\frac{(\text{equal? } (\text{sort } (fname \dots) \text{ symbol}<?) (\text{sort } (fname_s \dots) \text{ symbol}<?)) \quad \text{valid}[ty, \text{field-of}[(fname \text{ value}), \dots, fname_s], \emptyset] \dots}{\text{valid}[(\text{Record } (fname_s \text{ ty}) \dots), (\text{rec } (fname \text{ value}) \dots), \text{owner}]} [\text{record}]$	
$\frac{\text{valid}[ty_i, \text{value}, \text{owner}]}{\text{valid}[(\text{OneOf } ty_1 \dots ty_i \text{ } ty_2 \dots), \text{value}, \text{owner}]} [\text{one-of}]$	
$\frac{(\text{or } (\text{exact-nonnegative-integer? } \text{value}) \text{ isString}(\text{value}))}{\text{valid}[(\text{Index } fname), \text{value}, \emptyset]} [\text{index-kind}]$	
$\frac{(\text{member } \text{value} \text{ } (\text{keys-in } value_o \text{ } fname_o))}{\text{valid}[(\text{Index } fname), \text{value}, value_o]} [\text{index}]$	
$\frac{(\text{or } (\text{null? } (cname \dots)) (\text{memq } cname_v \text{ } (cname \dots)))}{\text{valid}[(\text{Component } cname \dots), (\text{comp } cname_v \text{ any } \dots), \text{owner}]} [\text{component}]$	
$\frac{\text{valid}[ty, \text{value}, \text{owner}] \quad (\text{andmap } (\lambda (c) (\text{check-holds? } c \text{ value})) (\text{check } \dots))}{\text{valid}[(\text{Refine } ty \text{ check } \dots), \text{value}, \text{owner}]} [\text{refine}]$	
$\frac{\text{valid}[ty, \text{value}, \text{owner}]}{\text{valid}[(\text{Default } ty \text{ value}_d), \text{value}, \text{owner}]} [\text{default}]$	

Figure 9. valid: isValid.

default-of: makeDefault; none for a required component or an empty index

```
default-of[[Any, owner]] =  
null  
default-of[[String, owner]] =  
""  
  
default-of[[Number, owner]] =  
0  
default-of[[Boolean, owner]] =  
#f  
default-of[[Enum value0 value ...], owner]] =  
value0  
default-of[[Option ty], owner]] =  
null  
default-of[[Array ty], owner]] =  
(arr)  
default-of[[Map ty], owner]] =  
(rec)  
default-of[[Record (fname ty) ...], owner]] =  
(rec (fname make-field[[ty, absent, ∅]] ...) ...)  
default-of[[OneOf ty0 ty ...], owner]] =  
make-field[[ty0, absent, owner]]  
default-of[[Index fname], value]] =  
first((skey0 skey ...))  
  where (skey0 skey ...) = (keys-in value fname)  
default-of[[Refine Number check ...], owner]] =  
(clamp 0 (check ...))  
default-of[[Refine ty check ...], owner]] =  
default-of[[ty, owner]]  
default-of[[Default ty value], owner]] =  
value
```

meets?: only the type's own constraints

```
meets?[[Refine ty check ...], value]] =  
(and meets?[[ty, value]] (andmap (λ (c) (check-holds? c value)) (check ...)))  
meets?[[Default ty valued], value]] =  
meets?[[ty, value]]  
meets?[[ty, value]] =  
#t
```

make-field

```
make-field[[ty, value, owner]] = value  
  where valid[[ty, value, owner]]  
make-field[[ty, arg?, owner]] = valued  
  where valued = default-of[[ty, owner]], #t = meets?[[ty, valued]]
```

Figure 10. Making a field: defaults, constraints, make.

3.3 Classes and protocols

A class: fields with their types, invariants on self, and handlers. A handler's contract is MessageSpec: argument types, the source key's type, pre on self and the arguments, post relating self and next, and the messages it may emit. Its body is one step or cases, the first enabled one running. A step's updates assign into next, reading self only; a field no update names is unchanged. Its effects run in order and may read next.

```
class-def ::= (class cname csection ...)
csection ::= (fields (fname ty) ...)
            | (invariants (fname hx) ...)
            | (on handler ...)

            | (answers (fname answer-def ...) ...)
answer-def ::= (answer-kind (x ...) contract ... body)
answer-kind ::= ok | error | unhandled
hkey ::= fname | (answer fname answer-kind)
handler ::= (fname (x ...) contract ... body)
contract ::= (args ty ...) | (source-key ty) | (pre hx) | (post hx) | (emits (emit-kind fname ...) ...)
emit-kind ::= notify | ask | send-at
body ::= (step part ...) | (cases (hx body) ...)
part ::= (next (lv ...) hx) | (become hx) | eff
lv ::= fname | (at hx)
eff ::= (notify fname hx ...) | (ask fname hx ...)
      | (send-at fname (hx ...) sel) | (forward-at sel)
      | (stop)
      | (log hx)
      | (fail hx)
```

Figure 11. Classes, contracts and actions.

3.4 Actions and checks

act steps one instance: it picks the first enabled case, applies the updates over self (or the replacement become names), and evaluates the effects. A body with no enabled case, or an effect that fails, aborts; effects evaluated before the abort survive, as JS side effects do. message-issues checks a handled message against its contract, like messageIssues. invalid-in is findInvalidFields: undeclared and invalid fields, then broken invariants, skipping subtrees that did not change. Checked: a handler refines its contract from every valid state meeting pre.

```
fx ::= (notify fname value ...) | (ask fname value ...)
      | (send-at fname (value ...) sel) | (forward-at sel) | (stop) | (log value)
stepped ::= (ok value (fx ...)) | (aborted (fx ...))
issue ::= string
```

act(class, self, m, args, sourceKey): the first case whose guard is truthy;
next = become or self, then each update next.p = e with e read against self;
then each effect in order, against self and next. fail, a throw, or no enabled
case gives (aborted fx...), the effects evaluated before it.

Figure 12. Evaluated effects and the outcome of a step.

Views

The view half of `component()`: templates of expressions that evaluate to a semantic tree. Elements keep their handlers, and wrapper nodes say what the tree means: this is the render of a component, this is rendered at a frame, these are the items of a loop. Routing is defined on that tree, and well-formed views only send messages the class handles. How the DOM records any of this is Chapter 5's business.

4.1 The semantic tree

Elements with props and handlers, text, the view's own comments, explicit fragments, and three wrappers: `component`, `enter` (rendered at a frame: a field, the selected item of a field, or a provided value) and `items`, one entry per loop item with its key and frame. `normalize` is `addChild`: arrays and fragments splice, null drops out, nodes stay, other values become text.

```

tag ::= string
ns? ::= ns | ∅
tid ::= natural
key? ::= string | ∅
prop-value ::= value | snode | (frag prop-value ...)
on-attr ::= (on fname action)
action ::= (send fname arg ...)
arg ::= value | (target-value) | (source-key) | (drag-key) | (if arg arg arg)
snode ::= (el tid tag ns? key? (prop ...) (on-attr ...) (snode ...))
        | (text string)
        | (comment string)
        | (group key? (snode ...))
        | (component cname (snode ...))
        | (enter dframe (snode ...))
        | (items entry ...)
entry ::= (item key? dframe (snode ...))

```

Figure 13. The semantic tree: elements keep their handlers; wrappers say what their children are.

```

result ::= value | snode | (frag result ...)

normalize
normalize[[null]]           = ()
normalize[[snode]]         = (snode)
normalize[(frag result ...)] = (snode ... ...)
  where ((snode ...) ...) = (normalize[[result]] ...)
normalize[(arr value ...)]  = (snode ... ...)
  where ((snode ...) ...) = (normalize[[value]] ...)
normalize[[value]]         = ((text String(value)))

key-of: key != null ? String(key) : undefined
key-of[[null]]           = ∅
key-of[[result]]        = String(result)

```

Figure 14. addChild: from evaluated results to a flat node sequence.

4.2 View syntax

A view definition belongs to a class: views, loops, scopes and provides. `component()` is both halves in one form; registration splits it. Every form a template can use: `field`, `seqItem`, `provided`, `lookupBind`, `$if`, `h`, `render`, `each`, `renderEach`, `renderIt`, `scope` and `pushView`, plus constants and multi-root fragments.

```
view-def ::= (view-of cname vsection ...)
vsection ::= (views (view-name e) ...)
            | (loops loop ...)
            | (scopes (fname hx) ...)
            | (provides (fname sel) ...)
loop ::= (fname loop-opt ...)
loop-opt ::= (key hx) | (filter hx) | (enrich hx) | (view view-name)
component-def ::= (component cname section ...)
e ::= value | snode
    | (field fname)
    | (seq-item fname fname)
    | (provided fname)
    | (lookup-bind bname)
    | (if e e e)
    | (h tid tag (attr ...) e ...)
    | (frag e ...)
    | (x-render src view-name?)
    | (each src e loop-name?)
    | (render-each src loop-name?)
    | (render-it view-name?)
    | (scope fname e)
    | (push-view e e)
src ::= (field fname) | (seq-item fname fname) | (provided fname)
attr ::= (prop-name e) | (key e) | (namespace ns) | on-attr
ctx ::= (rctx value binds dyn (value ...) integer value)
binds ::= ((bname value) ...)
dyn ::= ((fname value natural upath) ...)
table ::= ((tid (on-attr ...)) ...)
registration ::= (reg cname table class-def view-def)
registry ::= (registration ...)
```

Figure 15. View definitions, expressions, attributes and render contexts.

4.3 Registration

component() and the Collect walker. Each class gets one table from template id to the event handlers of that element, built by walking every view: an element's entry, then its attribute expressions, then its children. A class needs a main view, and a template shared between views keeps one entry.

split-component

```
split-component[(component cname section ...)] =  
((class cname csection ...) (view-of cname vsection ...))  
  where (csection ...) = (filter (λ (s) (redex-match? View csection s)) (section ...)),  
        (vsection ...) = (filter (λ (s) (redex-match? View vsection s)) (section ...))
```

add-entry

```
add-entry[tid, (), table] =  
table  
add-entry[tid, (on-attr ...), (any1 ... (tid (on-attr ...)) any2 ...)] =  
(any1 ... (tid (on-attr ...)) any2 ...)  
add-entry[tid, (on-attr ...), ((tidt anyt) ...)] =  
((tidt anyt) ... (tid (on-attr ...)))  
  where (not (member tid (tidt ...)))
```

collect

```
collect[(h tid tag (attr ...) e ...), table] =  
collect-all[(ea ... e ...), add-entry[tid, on-attrs[(attr ...)], table]]  
  where (ea ...) = attr-exprs[(attr ...)]  
collect[(if ec et ee), table] =  
collect-all[(ec et ee), table]  
collect[(frag e ...), table] =  
collect-all[(e ...), table]  
collect[(each src e loop-name?), table] =  
collect[e, table]  
collect[(scope fname e), table] =  
collect[e, table]  
collect[(push-view ev e), table] =  
collect-all[(ev e), table]  
collect[e, table] =  
table
```

register

```
register[component-def] =  
(reg cname collect-all[(e ...), ()] class-def (view-of cname vsection ...))  
  where (class-def (view-of cname vsection ...)) = split-component[component-def],  
        ((view-name e) ...) = (def-section (view-of cname vsection ...) views),  
        (member "main" (view-name ...))
```

Figure 16. Registration: split the halves; Collect maps template ids to handlers.

4.4 Render contexts and view selection

A RenderContext holds the current value, binds, provided values, the view stack and the depth. Entering a frame adds one depth and captures the component's providers there, shadowing outer ones; a loop item's view keeps the depth and clears the binds. best-view picks the view to render: the explicit name, then the view stack, then main.

enter-frame: one more path frame; providers captured at its depth

```
enter-frame[registry, (rctx valueit bindsold dyn (valuev ...) integer valuec), value, binds] =  
(rctx value binds merge-dyn[registry, dyn, value, integerd] (valuev ...) integerd valuec)  
where integerd = succ(integer)
```

merge-dyn

```
merge-dyn[registry, dyn, (comp cname any ...), natural] =  
( ([dyn dyn]) ([p (def-section view-def provides)])  
  (match-define (list name sel) p)  
  (define suffix resolve[freeze-sel[sel, (comp cname any ...)]]]  
  (define v get-in[suffix, (comp cname any ...)])  
  (cons (list name v natural suffix)  
        (filter (λ (d) (not (eq? (first d) name))) dyn)))
```

where *view-def* = view-def-of[registry, (comp cname any ...)]

```
merge-dyn[registry, dyn, value, natural] =  
dyn
```

root-ctx

```
root-ctx[registry, value] = enter-frame[registry, (rctx null () () -1 value), value, ()]
```

for-item-view

```
for-item-view[(rctx valueit binds dyn anyv integer valuec), value] =  
(rctx value () dyn anyv integer valuec)
```

Figure 17. RenderContext.

best-view

```
best-view[[view-def, view-name, (value ...)]] =  
best-view-in[[view-def, (view-name value ...)]]  
best-view[[view-def,  $\emptyset$ , (value ...)]] =  
best-view-in[[view-def, (value ...)]]
```

best-view-in

```
best-view-in[[view-def, ()]] =  
(second (lookup-named (def-section view-def views) "main"))  
best-view-in[[view-def, (value0 value ...)]] =  
(second (lookup-named (def-section view-def views) value0))  
where (lookup-named (def-section view-def views) value0)  
best-view-in[[view-def, (value0 value ...)]] =  
best-view-in[[view-def, (value ...)]]
```

Figure 18. View selection: explicit name, then the view stack, then main.

4.5 Evaluation

Every template node has one eval rule. A render source enters a frame and wraps the component's render; renderView evaluates the component's best view inside a component wrapper. Loops make one entry per item, after the loop's hooks pick entries, keys and extra binds.

to-frame

```
to-frame[ctx, (field fname)] =  
(frame () ((symbol->string fname)) () #t)  
to-frame[(rctx valueit any ...), (seq-item fname fnamek)] =  
(frame () ((symbol->string fname)) item-of[prop-get[valueit, fnamek]] #t)  
to-frame[(rctx valueit binds dyn anyv integer valuec), (provided fname)] =  
(frame (prov fname natural upath) () () #t)  
  where (value natural upath) = lookup-dyn[dyn, fname]
```

render-view

```
render-view[registry, (comp cname any ...), (rctx valueit binds dyn (valuev ...) integer valuec), view-name?] =  
(component cname  
  normalize[  
    eval[registry, with-component[(rctx valueit binds dyn (valuev ...) integer valuec), (comp cname any ...)],  
      best-view[view-def-of[registry, (comp cname any ...)], view-name?, (valuev ...)]])])
```

eval-attrs

```
eval-attrs[registry, ctx, ()] =  
(() () ())  
eval-attrs[registry, ctx, ((key e) attr ...)] =  
((prop ...) key-of[eval[registry, ctx, e]] ns?)  
  where ((prop ...) key? ns?) = eval-attrs[registry, ctx, (attr ...)]  
eval-attrs[registry, ctx, ((namespace ns) attr ...)] =  
((prop ...) key? ns)  
  where ((prop ...) key? ns?) = eval-attrs[registry, ctx, (attr ...)]  
eval-attrs[registry, ctx, (on-attr attr ...)] =  
eval-attrs[registry, ctx, (attr ...)]  
eval-attrs[registry, ctx, ((prop-name e) attr ...)] =  
(((prop-name eval[registry, ctx, e]) prop ...) key? ns?)  
  where ((prop ...) key? ns?) = eval-attrs[registry, ctx, (attr ...)]
```

Figure 19. Render sources and renderView.

eval

```
eval[registry, ctx, value] =
value
eval[registry, ctx, snode] =
snode
eval[registry, (rctx valueit any ...), (field fname)] =
prop-get[valueit, fname]
eval[registry, (rctx valueit any ...), (seq-item fname fnames)] =
prop-get[prop-get[valueit, fname], prop-get[valueit, fnames]]
eval[registry, (rctx valueit binds dyn any, integer valuee), (provided fname)] =
(let ([d (assq fname dyn)]) (if d (second d) null))
eval[registry, (rctx valueit binds any ...), (lookup-bind bname)] =
bind-of[binds, bname]
eval[registry, ctx, (if ec ee es)] =
eval[registry, ctx, es]
where truthy?(eval[registry, ctx, ec])
eval[registry, ctx, (if ec ee es)] =
eval[registry, ctx, es]
eval[registry, ctx, (h tid tag (attr ...) e ...)] =
(eltid tag ns? key? (prop ...) on-attrs[(attr ...)] (snode ... ...))
where ((prop ...) key? ns?) = eval-attrs[registry, ctx, (attr ...)],
((snode ...) ...) = (normalize[eval[registry, ctx, e]] ...)
eval[registry, ctx, (frag e ...)] =
(frag eval[registry, ctx, e] ...)
eval[registry, ctx, (x-render src view-name?)] =
(enter to-frame[ctx, src] (render-view[registry, value, enter-frame[registry, ctx, value, ()], view-name?]))
where value = eval[registry, ctx, src]
eval[registry, ctx, (each src e loop-name?)] =
eval-each[registry, ctx, src, e, loop-name?]
eval[registry, ctx, (render-each src loop-name?)] =
eval-each[registry, ctx, src, render-each, loop-name?]
eval[registry, (rctx valueit binds any ...), (render-it view-name?)] =
render-view[registry, value, for-item-view[(rctx valueit binds any ...), value], view-name?]
where value = bind-of[binds, value]
eval[registry, (rctx valueit binds dyn anye integer valuee), (scope fname e)] =
eval[registry, (rctx valueit ((bname value) ... anye ...) dyn any, integer valuee), e]
where (anye ...) = binds,
(fname hx) = (lookup-named (def-section view-def-of[registry, valuee] scopes),
fname)
(rec (bname value) ...) = (eval-hx[(self valuee), hx]
[(list rec fs ...) (quasiquote(rec ,@(filter (λ (f) (not (eq? (first f) 'value))) fs))))])
eval[registry, (rctx valueit binds dyn (valuee ...) integer valuee), (push-view ev e)] =
eval[registry, (rctx valueit binds dyn (valuee valuee ...) integer valuee), e]
where valuee = eval[registry, (rctx valueit binds dyn (valuee ...) integer valuee), ev]
```

Figure 20. Evaluation: every template node has one eval.

eval-each

```
eval-each[[registry, (rctx valueit anyr ...), src, anybody, loop-name?]] =  
(items ( ([entry entries-of[[eval[[registry, ctx, src]]]]  
          #:when (keep? view-def loop-name? entry valuec))  
          eval-item[[registry, ctx, view-def, dframe, src, anybody, loop-name?, entry]]))  
where ctx = (rctx valueit anyr ...),  
      (rctx anyi ... valuec) = ctx,  
      view-def = view-def-of[[registry, valuec],  
      dframe = to-frame[[ctx, src]]
```

eval-item

```
eval-item[[registry, ctx, view-def, dframe, src, anybody, loop-name?, (skey value)]] =  
(item key-of[[valuekey]] item-frame[[dframe, skey]] normalize[[result]])  
where (rctx anyi ... valuec) = ctx,  
      valuekey = (item-key view-def loop-name? skey value valuec),  
      binds = (item-binds view-def loop-name? skey value valuec),  
      ctxitem = enter-frame[[registry, ctx, value, binds],  
      result = eval-body[[registry, ctxitem, view-def, anybody, loop-name?]]
```

eval-body

```
eval-body[[registry, (rctx valueit anyr ...), view-def, render-each, loop-name?]] =  
render-view[[registry, valueit, for-item-view[[rctx valueit anyr ...), valueit,  
      (or loop-opt[[view-def, loop-name?, view]] ∅)]]  
eval-body[[registry, ctx, view-def, e, loop-name?]] =  
eval[[registry, ctx, e]]
```

render-root

```
render-root[[registry, value]] =  
normalize[[render-view[[registry, value, root-ctx[[registry, value], ∅]]]]
```

Figure 21. Loops: one entry per item, with its key and frame.

4.6 Routing on the tree

Which handler an event on an element runs, and where. An element's dispatch path is the frames of the wrappers above it, a component wrapper making the frame before it a component frame. Its candidate handlers are its own, then those of the elements above it up to the nearest component wrapper; the first for the event's type wins. Drop events bubble: without a handler there, the next enclosing component's are tried, at the frame of the child the event came through.

sites-of

```
sites-of[(el tid tag ns? key? (prop ...) (on-attr ...) (snode ...)), (anyhs ...) dpath, (anyl ...) boolean] =
()
sites-of[(snodeo snode ...), anyhs, dpath, anyl, boolean] =
(siteo ... site ...)
where (siteo ...) = sites-in[(snodeo, anyhs, dpath, anyl, boolean),
      (site ...) = sites-of[(snode ...), anyhs, dpath, anyl, boolean]
```

sites-in

```
sites-in[(el tid tag ns? key? (prop ...) (on-attr ...) (snode ...)), (anyh ...) dpath, (level ...) #t] =
((elem (((on-attr ...) anyh ...) dpath) level ...) sitec ...)
where (sitec ...) = sites-of[(snode ...), ((on-attr ...) anyh ...) dpath, (level ...) #t]
sites-in[(el tid tag ns? key? (prop ...) (on-attr ...) (snode ...)), anyhs, dpath, anyl, #f] =
((elem ()) sitec ...)
where (sitec ...) = sites-of[(snode ...), (), dpath, (), #f]
sites-in[(component cname (snode ...)), anyhs, dpath, (level ...) #t] =
sites-of[(snode ...), (), promote[dpath], ((anyhs outside[dpath]) level ...) #t]
sites-in[(component cname (snode ...)), anyhs, dpath, anyl, #f] =
sites-of[(snode ...), (), promote[dpath], (), #t]
sites-in[(enter dframe (snode ...)), anyhs, (dframep ...) anyl, boolean] =
sites-of[(snode ...), anyhs, (dframep ... dframe), anyl, boolean]
sites-in[(items (item key? dframe (snode ...) ...) anyhs, dpath, anyl, boolean] =
(site ... ...)
where ((site ...) ...) = (sites-in[(enter dframe (snode ...)), anyhs, dpath, anyl, boolean] ...)
sites-in[(group key? (snode ...)), anyhs, dpath, anyl, boolean] =
sites-of[(snode ...), anyhs, dpath, anyl, boolean]
sites-in[(snode, anyhs, dpath, anyl, boolean] =
()
```

outside: the enclosing component's route

```
outside[(el)] =
()
outside[(dframe ... (frame base anyk item? boolean))] =
(dframe ... (frame base anyk item? #f))
```

site-info

```
site-info[(elem (level ...)), fname] =
(or ( ([l] ( ([ls (level ...)])
              (if (drop-event? fname) ls (take ls (min 1 (length ls))))))
    [h (append* (first l))]
    #:when (eq? (second h) fname))
(list h ((second l) to-update-path[(second l)])))
∅)
```

Figure 22. Sites: each element with its handler chain and dispatch path.

4.7 Well-formed views

Every event binding sends a message of the class's protocol, with the arity its contract declares and constants of the declared types: lint's unknown-handler, send-arity and send-arg-type. Checked: on random trees of well-formed components, every event finds its handler.

$$\frac{\begin{array}{c} handler = (handler\text{-}of\ class\text{-}def\ fname_m) \\ \#t = (args\text{-}ok?\ (contract\text{-}of\ handler\ args)\ (arg\ \dots)) \\ wf\text{-}send[[class\text{-}def, (on\ fname_e\ (send\ fname_m\ arg\ \dots))]] \end{array}}{[send]}$$
$$\frac{\begin{array}{c} (on\text{-}attr_b\ \dots) = (on\text{-}attr\ \dots\ \dots) \\ wf\text{-}send[[class\text{-}def, on\text{-}attr_b]]\ \dots \end{array}}{wf\text{-}registration[[reg\ cname\ ((tid\ (on\text{-}attr\ \dots))\ \dots)\ class\text{-}def\ view\text{-}def]]\ [registration]}$$
$$\frac{wf\text{-}registration[[registration]]\ \dots}{wf\text{-}registry[[registration\ \dots]]\ [registry]}$$

Figure 23. Well-formed sends, registrations and registries.

Binding

What binds the declarative side to the virtual DOM of Appendix A. Lowering encodes each wrapper as a metadata comment, its children and a closing comment, and each element with handlers as a data-eid attribute. Decoding reads them back from the DOM. Checked, the binding theorem: for every element of a semantic tree, decoding the rendered lowering picks the same handler and route as routing on the tree. With the virtual DOM's own theorem, the same holds on every DOM the host morphs into.

5.1 Metadata

A metadata comment says what the wrapper it opens is: a component boundary or one path frame, as JSON between § marks. A frame's cf flag is not written: decoding restores it from the component boundary that follows.

$meta ::= \text{component} \mid (\text{frame } base \ (skey \ \dots) \ item?)$

frame-meta: cf is not written

$\text{frame-meta} \llbracket (\text{frame } base \ any_k \ item? \ boolean) \rrbracket = (\text{frame } base \ any_k \ item?)$

meta-frame

$\text{meta-frame} \llbracket (\text{frame } base \ any_k \ item?) \rrbracket = (\text{frame } base \ any_k \ item? \ \#f)$

Figure 24. Metadata: what an opening comment says.

5.2 Lowering

From the semantic tree to vnodes. Wrappers become balanced comments; a loop becomes an unkeyed group of keyed item groups, each loop its own sibling key scope; an element with handlers gets data-eid; elements in props become vnode values.

wrap

$\text{wrap}[\![meta, (node \dots)]\!] = ((\text{comment } \text{metaText}(meta)) \text{ node } \dots (\text{comment } "\$/\$"))$

eid-props

$\text{eid-props}[\![tid, ()]\!] = ()$

$\text{eid-props}[\![tid, (on-attr_0 \text{ on-attr } \dots)]\!] = ((\text{data-eid } tid))$

lower

$\text{lower}[\![(\text{text } string)]\!] =$
 $((\text{text } string))$

$\text{lower}[\![(\text{comment } string)]\!] =$
 $((\text{comment } string))$

$\text{lower}[\![(\text{group } key? (snode \dots))]\!] =$
 $((\text{group } key? \text{ lower-all}[\![(snode \dots)]\!]))$

$\text{lower}[\![(\text{el } tid \text{ tag } ns? \text{ key? } (prop \dots) (\text{on-attr } \dots) (snode \dots))]\!] =$
 $(\text{make-el}[\![tid, tag, ns?, key?, (prop_l \dots prop_{eid} \dots), \text{lower-all}[\![(snode \dots)]\!]]])$
where $(prop_l \dots) = \text{lower-props}[\![(prop \dots)]\!]$,
 $(prop_{eid} \dots) = \text{eid-props}[\![tid, (\text{on-attr } \dots)]\!]$

$\text{lower}[\![(\text{component } cname (snode \dots))]\!] =$

$\text{wrap}[\![\text{component}, \text{lower-all}[\![(snode \dots)]\!]]\!]$

$\text{lower}[\![(\text{enter } dframe (snode \dots))]\!] =$

$\text{wrap}[\![\text{frame-meta}[\![dframe]\!], \text{lower-all}[\![(snode \dots)]\!]]\!]$

$\text{lower}[\![(\text{items } entry \dots)]\!] =$

$((\text{group } \emptyset (\text{lower-entry}[\![entry]\!] \dots)))$

lower-entry

$\text{lower-entry}[\![(\text{item } key? dframe (snode \dots))]\!] =$

$(\text{group } key? \text{ wrap}[\![\text{frame-meta}[\![dframe]\!], \text{lower-all}[\![(snode \dots)]\!]]\!])$

lower-all

$\text{lower-all}[\![(snode \dots)]\!] = (\text{node } \dots \dots)$

where $((\text{node } \dots) \dots) = (\text{lower}[\![snode]\!] \dots)$

lower-root

$\text{lower-root}[\![registry, value]\!] = \text{lower-all}[\![\text{render-root}[\![registry, value]\!]]\!]$

Figure 25. Lowering wrappers to comments and handlers to data-eid.

5.3 Decoding events

getEventInfoFor. The DOM holds no component references, only data-eid attributes and metadata comments. Decoding walks from the target up, collects eids until the first component boundary, and reads the metadata enclosing each ancestor to rebuild the dispatch path; closing comments hide openings and look-alikes are skipped. The handler is the first one for the event type, innermost eid first, in the table of the class the path reaches; a drop event tries each enclosing component in turn.

```
addr ::= (natural ...)
info ::= (on-attr route) | ∅
```

enclosing-meta

```
enclosing-meta[(dom ...), addr] = scan[reverse(siblings-before[(dom ...), addr]), 0]
```

scan: closing comments hide openings; look-alikes are skipped

```
scan[(), natural] = ()
scan[((comment string) dom ...), natural] = scan[(dom ...), succ(natural)]
  where (equal? string "$/$")
scan[((comment string) dom ...), natural] = scan[(dom ...), pred(natural)]
  where metaLike?(string), (> natural 0)
scan[((comment string) dom ...), 0] = (meta metarest ...)
  where meta = parseMeta(string), (metarest ...) = scan[(dom ...), 0]
scan[(dom0 dom ...), natural] = scan[(dom ...), natural]
```

walk: eids with the component level they're at

```
walk[(dom ...), (), anye, (meta ...)] =
  (anye (meta ...))
walk[(dom ...), (naturala ... naturallast), ((naturale naturali) ...), (meta ...)] =
  walk[(dom ...), (naturala ...), ((naturale naturali) ... anynew ...), (meta ... metahere ...)]
  where addr = (naturala ... naturallast),
    (anynew ...) = ( ([eid eid-of[node-at[(dom ...), addr]]])
      (if eid (list (list eid (count-components (meta ...)))) ()) ),
    (metahere ...) = enclosing-meta[(dom ...), addr]
```

Figure 26. From the event target back to a handler and its component's path.

frames-of

```
frames-of[(dom ...), dpath] =  
dpath  
frames-of[(component meta ...), dpath] =  
frames-of[(meta ...), promote[dpath]]  
frames-of[(meta0 meta ...), (dframe ...)] =  
frames-of[(meta ...), (dframe ... meta-frame[meta0])]
```

event-info

```
event-info[registry, value, (dom ...), fname, addr] =  
( ( [metas (meta ...)]  
  [ends (for/list ([m metas] [i (in-naturals)]) #:when (eq? m component)) i]]  
  [levels (if (drop-event? fname) (length ends) (min 1 (length ends)))]  
  (or ( ([level levels]  
        [dpath (in-value frames-of[(reverse (drop metas (if (zero? level) 0 (add1 (list-ref ends (sub1 level))))))],  
        (0))]  
        [upath (in-value to-update-path[dpath]])]  
        [eid ( [e ((naturale natural) ...)] #:when (= (second e) level))  
              (first e))]  
        [h handlers-for[registry, comp-class[get-in[upath, value]], eid]]  
        #:when (eq? (second h) fname)]  
        (list h (list dpath upath)))  
      (0))  
  where (((naturale natural) ...) (meta ...)) = walk[(dom ...), addr, (0), (0)]
```

path-of: where a dragged element was rendered

```
path-of[(dom ...), addr] =  
(if (memq component (meta ...)) frames-of[reverse((meta ...)), (0)] (0))  
where (anye (meta ...)) = walk[(dom ...), addr, (0), (0)]
```

Checked: event-info(render(lower-root(r, root)), a_i) = site-info(site_i)
for the i-th element, on random component trees.

Figure 27. Rebuilding the path and choosing the handler.

Chapter 6

The host

Closing the loop. An event on the DOM is decoded into a task, which runs as one atomic transaction over the root state, and the DOM is then morphed to the new root. Read as TLA+, a configuration is the state and the reduction relation is Next: each rule is one action, and all of the system's nondeterminism is which rule fires. Checked: after every drain, the DOM is the render of the root.

6.1 Host configurations

What the host holds: the root state, the task queue, pending async replies, the DOM with identities and the vnodes and root it was rendered from, the events still to arrive, the observations (logs, failures, renders and dev-mode reports), the next fresh label, and the root when the queue was last empty. Tasks are plain, notify, ask or event, and target a base route plus a selector.

```
kind ::= plain | (notify boolean) | ask | (answer fname answer-kind) | (event action value drag?)
task ::= (task kind fname (value ...) dpath target route?)
target ::= (at dpath sel)
route? ::= (dpath upath any) | ∅
ask-def ::= (fname ask-result)
ask-result ::= (ok hx) | (error hx) | (async (ok hx)) | (async (error hx))
evt ::= (event fname addr value)
obs ::= (log value) | (failed fname) | (unhandled fname) | (stale fname) | rendered
| (state-invalid any ...) | (message-invalid fname issue ...)
outcome ::= (ok value (task ...) (task ...) (obs ...) natural) | (failed (obs ...))
cfg ::= (host registry (ask-def ...) value (task ...) (task ...)
        (ldom ...) (node ...) value (evt ...) (obs ...) lbl value drag? natural)

eval-arg: send arguments in the message context
eval-arg[(target-value), valuetv, any, anyd] = valuetv
eval-arg[(source-key), valuetv, any, anyd] = any
eval-arg[(drag-key), valuetv, any, anyd] = anyd
eval-arg[(if arge argv argd), valuetv, any, anyd] = eval-arg[arge, valuetv, any, anyd]
where truthful?(eval-arg[arge, valuetv, any, anyd])
eval-arg[(if arge argv argd), valuetv, any, anyd] = eval-arg[arge, valuetv, any, anyd]
eval-arg[value, valuetv, any, anyd] = value

route-event
route-event[registry, value, (ldom ...), (event fname addr valuetv), drag?] =
  ( ( (lexn:fail? (λ (x) (∅)))
    event-info[registry, value, erase-all[(ldom ...)], fname, addr]
  ) (∅) )
  ((list (list on action) (list path upath)))
  (define route (quasiquote(path upath (id-of (term get-in upath value))))
    (list (quasiquote(task (event action (term valuetv) (term drag?)) (second action) () path (at path ()) route))))

drag-key-in: Drag.keyIn
drag-key-in[(∅, dpath)] =
  null
drag-key-in[(drag (dframer ... dframeo dframe ...), (dframer ...))] =
  source-key-in[(dframer ... dframeo dframe ...), (dframer ...)]
drag-key-in[(drag?, dpath)] =
  null

drag-from: dragstart starts a drag
drag-from[(ldom ...), (event dragstart addr value), drag?] =
  ( ( [doms erase-all[(ldom ...)]
    ]a ( ( (λ (in-range length(addr) 0 -1))
      #:when (node-at-doms (take addr n))
        ((quasiquote(element _ _ _ props _)) (equal? (assq draggable props) (draggable "true")))
        [ #f ]))
    (take addr n))
    [path (and a path-orig doms a)]))
  (if (and path (not (eq? path ∅))) (quasiquote(drag ,path)) drag?)
drag-from[(ldom ...), evt, drag?] =
  drag?

drag-after: dragend ends it
drag-after[(event dragend addr value), drag?] = ∅
drag-after[(evt, drag?)] = drag?
```

Figure 28. Host configurations.

6.2 The task queue

host-> has six rules: event decodes against the current DOM, run commits one transaction, fail aborts one, resolve settles a pending async ask in any order, render morphs the DOM when the root changed, and settle checks the invariants once the queue is empty. Any exception aborts the whole transaction: the root and the queue stay as they were. After a commit, the fields that changed are checked against their types and the handler that ran against its contract; invariants wait for the queue to empty, since a task may break them for a task it queued.

```

(host registry anye value (tsk ...) anye (ldom ...) any, any. → [event]
  (evte evt ...) anye lbl any, drag? natural)
(host registry anye value (tsk ... tske ...) anye (ldom ...) any, any.
  (evt ...) anye lbl any, drag-after[evte, drag?new] natural)
  where drag?new = drag-from[(ldom ...), evte, drag?], (tske ...) = route-event[registry, value, (ldom ...), evte, drag?new]
(host registry (ask-def ...) value (tsko tsk ...) (tskp ...) anyd any, any. → [run]
  any, (obs ...) lbl any, anye natural)
(host registry (ask-def ...) valuenew (tsko ... tsk ... tske ...) (tskp ... tskr ...) anyd any, any.
  anye (obs ... obsnew ...) lbl any, anye naturalnew)
  where (ok valuenew (tsko ...) (tskr ...) (obsnew ...) naturalnew) = handle-task[registry, (ask-def ...), value, tsko, natural]
(host registry (ask-def ...) value (tsko tsk ...) anyp anye any, any. → [fail]
  any, (obs ...) lbl any, anye natural)
(host registry (ask-def ...) value (tsk ...) anye anyd any, any.
  anye (obs ... obsnew ...) (failed task-name[tsko]) lbl any, anye natural)
  where (failed (obsnew ...)) = handle-task[registry, (ask-def ...), value, tsko, natural]
(host registry any, value (tsk ...) (tsk1 ... tskr tsk2 ...) anye any, any. → [resolve]
  any, any, lbl any, any, natural)
(host registry anye value (tsk ... tsk2 ...) anyd any, any.
  any, any, lbl any, anye natural)
(host registry anye value anye anye (ldom ...) (nodeold ...) valuer → [render]
  anye (obs ...) lbl any, anye natural)
(host registry any, value anye anye (ldomnew ...) (nodenew ...) value
  any, (obs ... rendered) lblnew any, any, natural)
  where (not (equal? value valuer)), (nodenew ...) = lower-root[registry, value], ((ldomnew ...) lblnew) = morph-children[(ldom ...), (nodeold ...), (nodenew ...), HTML-NS, lbl]
(host registry anye value () anye, anyd any, any, anye (obs ...) lbl value, anye natural) → [settle]
(host registry anye value () anye, anyd any, any, anye (obs ... obsnew ...) lbl value anye natural)
  where (not (equal? value valuer)), (obsnew ...) = settle-reports[registry, value, valuer]

```

handle-task: read the leaf at the update path; run the task by kind (plain, notify, ask, event), stepping the handler with act; set the new leaf; freeze each child's selector against the new root; report invalid fields and contract issues. Any failure aborts the transaction.

Figure 29. The host as a reduction relation; handle-task runs one transaction.

6.3 Settled states

checkSettled: when the queue is empty and the root changed since it last was, the invariants of what changed are reported.

settle-reports

```

settle-reports[registry, value, values] =
  ( ([issues invalid-in[classes-of[registry], value, values, (invariants)]] )
    (if (null? issues) () (list (quasiquote(state-invalid ,@issues)))) )

```

Figure 30. Invariants hold of settled states.

Virtual DOM

vdom.js, which knows nothing about tutuca: normalized vnodes, the DOM they render to, and reconciliation of an existing DOM against new vnodes. DOM nodes carry labels that stand for their identity, so the model says which nodes survive a morph, not only what the DOM looks like after it. Checked: `erase(morph(mount(old), old, new)) = render(new)`, and morphing a tree into itself keeps every label. The binding relies on nothing else.

A.1 Normalized vnodes

Elements, wrapperless groups with an optional key, text and comments. An element carries its template id `tid`, which is its Shape: only elements of the same template are the same kind. Props are sorted by name, since attribute order is not observable, and `svg` and `math` set their own namespace.

```

tag ::= string
ns  ::= string
ns? ::= ns | ∅
tid ::= natural
key? ::= string | ∅
prop-name ::= (variable-except key namespace null)
prop-value ::= value | node | (frag prop-value ...)
prop ::= (prop-name prop-value)
leaf ::= (text string) | (comment string)
node ::= (el tid tag ns? key? (prop ...) (node ...))
        | (group key? (node ...))
        | leaf

```

Figure 31. Normalized vnodes. `tid` is the element's template (its Shape); props are sorted by name.

shape-of: svg and math set their own namespace

```
shape-of[[tag, ∅]] = (lower(tag) SVG-NS)
  where (string-ci=? tag "svg")
shape-of[[tag, ∅]] = (lower(tag) MATH-NS)
  where (string-ci=? tag "math")
shape-of[[tag, ns?]] = (tag ns?)
```

make-el: new VNode(shape, values, childs, key)

```
make-el[[tid, tag, ns?, key?, (prop ...), (node ...)]] =
  (el tid tags ns?s key? (sort-props (prop ...)) (node ...))
  where (tags ns?s) = shape-of[[tag, ns?]]
```

Figure 32. Element construction.

A.2 DOM output

toDom into a fresh container. Groups contribute their children with no wrapper, and the namespace flows from parent to child, with foreignObject switching SVG back to HTML. A fresh render is the reference every reconciliation must agree with.

$dom ::= (\text{element } ns \text{ tag } (prop \dots) (dom \dots)) \mid (\text{text string}) \mid (\text{comment string})$

child-ns: foreignObject resets SVG to HTML

$\text{child-ns}[\![tag, ns]\!] = \text{HTML-NS}$

where $(\text{and } (\text{equal? } ns \text{ SVG-NS})$
 $\quad (\text{string-ci=? tag "foreignObject"}))$

$\text{child-ns}[\![tag, ns]\!] = ns$

to-dom

$\text{to-dom}[\![(\text{text string}), ns]\!] =$
 (text string)

$\text{to-dom}[\![(\text{comment string}), ns]\!] =$
 (comment string)

$\text{to-dom}[\![(\text{group key? } (node \dots)), ns]\!] =$
 $\text{render-into}[\![(node \dots), ns]\!]$

$\text{to-dom}[\![(\text{el tid tag } \emptyset \text{ key? } (prop \dots) (node \dots)), ns]\!] =$

$\text{to-dom}[\![(\text{el tid tag ns key? } (prop \dots) (node \dots)), ns]\!]$

$\text{to-dom}[\![(\text{el tid tag ns}_{el} \text{ key? } (prop \dots) (node \dots)), ns]\!] =$
 $(\text{element } ns_{el} \text{ tag } (prop \dots) \text{ render-into}[\![(node \dots), \text{child-ns}[\![tag, ns_{el}]\!]]])$

render-into

$\text{render-into}[\![(node \dots), ns]\!] = (dom \dots \dots)$

where $((dom \dots) \dots) = (\text{to-dom}[\![node, ns]\!] \dots)$

render: the container's children

$\text{render}[\![(node \dots)]\!] = \text{render-into}[\![(node \dots), \text{HTML-NS}]\!]$

Figure 33. DOM output: groups have no wrapper.

A.3 Keyed sibling matching

The first half of `morphChildren`: which old sibling each new sibling reuses. Keyed nodes are looked up by key and unkeyed nodes are taken in order, independently of each other. Matching stays within one element's or fragment's children and identifies whole virtual nodes, empty groups included. Duplicate keys on either side are an error.

$$\begin{aligned} & \text{index} ::= \text{natural} \\ \text{match-result} & ::= \text{index} \mid \text{fresh} \\ \text{matches} & ::= (\text{match-result } \dots) \mid \text{duplicate-keys} \end{aligned}$$

node-key

$$\begin{aligned} \text{node-key}[(\text{el } \text{tid } \text{tag } \text{ns? } \text{key? } (\text{prop } \dots) (\text{node } \dots))] &= \text{key?} \\ \text{node-key}[(\text{group } \text{key? } (\text{node } \dots))] &= \text{key?} \\ \text{node-key}[\text{leaf}] &= \emptyset \end{aligned}$$

find-key

$$\begin{aligned} \text{find-key}[\text{string}, (), \text{index}] &= \text{fresh} \\ \text{find-key}[\text{string}, (\text{node}_o \text{ node } \dots), \text{index}] &= \text{index} \\ &\text{where } \text{string} = \text{node-key}[\text{node}_o] \\ \text{find-key}[\text{string}, (\text{node}_o \text{ node } \dots), \text{index}] &= \text{find-key}[\text{string}, (\text{node } \dots), \text{succ}(\text{index})] \end{aligned}$$

unkeyed-indices

$$\begin{aligned} \text{unkeyed-indices}[(), \text{index}] &= () \\ \text{unkeyed-indices}[(\text{node}_o \text{ node } \dots), \text{index}] &= (\text{index } \text{index}_{\text{rest}} \dots) \\ &\text{where } \emptyset = \text{node-key}[\text{node}_o], (\text{index}_{\text{rest}} \dots) = \text{unkeyed-indices}[(\text{node } \dots), \text{succ}(\text{index})] \\ \text{unkeyed-indices}[(\text{node}_o \text{ node } \dots), \text{index}] &= \text{unkeyed-indices}[(\text{node } \dots), \text{succ}(\text{index})] \end{aligned}$$

Figure 34. Matching is local to each element or explicit fragment's children.

match-next: keyed lookup and unkeyed order are independent

```
match-next[(nodeold ...), (), (index ...)] =  
()  
match-next[(nodeold ...), (node0 node ...), (index ...)] =  
(find-key[string, (nodeold ...), 0] match-result ...)  
  where string = node-key[node0],  
        (match-result ...) = match-next[(nodeold ...), (node ...), (index ...)]  
match-next[(nodeold ...), (node0 node ...), (index0 index ...)] =  
(index0 match-result ...)  
  where ∅ = node-key[node0],  
        (match-result ...) = match-next[(nodeold ...), (node ...), (index ...)]  
match-next[(nodeold ...), (node0 node ...), ()] =  
(fresh match-result ...)  
  where ∅ = node-key[node0],  
        (match-result ...) = match-next[(nodeold ...), (node ...), ()]
```

match-siblings: duplicate keys in either sequence are rejected

```
match-siblings[(nodeold ...), (nodenew ...)] =  
match-next[(nodeold ...), (nodenew ...), unkeyed-indices[(nodeold ...), 0]]  
  where #t = keys-unique?[(nodeold ...)],  
        #t = keys-unique?[(nodenew ...)]  
match-siblings[(nodeold ...), (nodenew ...)] =  
duplicate-keys
```

Figure 35. Matched indices identify whole virtual nodes, including empty groups.

A.4 Labeled DOM and mounting

The DOM with identities. `mount` is `toDom` minting a fresh label for every node it creates. A virtual node may own zero, one or several DOM nodes, so `dom-width` and `split-groups` cut a DOM sequence into the range each virtual node owns.

```
lbl ::= natural
ldom ::= (element lbl ns tag (prop ...) (ldom ...)) | (text lbl string) | (comment lbl string)
        | (foreign lbl dom)
lgroups ::= ((ldom ...) ...)
out ::= ((ldom ...) lbl)

el-ns
el-ns[[], ns] = ns
el-ns[nsel, ns] = nsel

mount: toDom, minting labels
mount[(text string), ns, lbl] =
  (((text lbl string)) succ(lbl))
mount[(comment string), ns, lbl] =
  (((comment lbl string)) succ(lbl))
mount[(group key? (node ...)), ns, lbl] =
  mount-all[(node ...), ns, lbl]
mount[(el tid tag ns? key? (prop ...) (node ...)), ns, lbl] =
  (((element lbl nsel tag (prop ...) (ldom ...))) lblnext)
  where nsel = el-ns[ns?, ns],
        ((ldom ...) lblnext) = mount-all[(node ...), child-ns[tag, nsel], succ(lbl)]

mount-all
mount-all[[], ns, lbl] =
  () lbl
mount-all[(node0 node ...), ns, lbl] =
  ((ldom0 ... ldom ...) lbl2)
  where ((ldom0 ...) lbl1) = mount[node0, ns, lbl],
        ((ldom ...) lbl2) = mount-all[(node ...), ns, lbl1]
```

Figure 36. Labeled DOM: a label is a browser node's identity.

dom-width

```
dom-width[(group key? (node ...))] = (apply + (dom-width[(node) ...])  
dom-width[(node)] = 1
```

split-groups

```
split-groups[(()], ()) =  
(  
split-groups[(ldom ...), (node0 node ...)] =  
(take((ldom ...), natural) anyrest ...)  
where natural = dom-width[(node0),  
      (anyrest ...) = split-groups[(drop((ldom ...), natural), (node ...))]
```

Figure 37. Each virtual node owns a range of DOM nodes.

A.5 Morphing

morphNode keeps a node when the leaf kind matches or, for elements, when the template and key match; it then patches every prop and morphs the children. Otherwise it mounts a replacement. morphChildren reuses each matched source's DOM range and mounts the rest. It matches against the nodes render made only: foreign nodes, which other code put in a rendered element, don't shift the match and stay.

morph-node

```
morph-node[(text lbln stringold), (text strings), (text stringt), ns, lbl] =  
(((text lbln stringt)) lbl)  
morph-node[(comment lbln stringold), (comment strings), (comment stringt), ns, lbl] =  
(((comment lbln stringt)) lbl)  
morph-node[(element lbln nsn tagn (propold ...) (ldom ...)), =  
      (el tid tags nss key? (props ...) (nodes ...)),  
      (el tid tag ns? key? (prop ...) (node ...)), ns, lbl]  
(((element lbln nsn tag (prop ...) (ldomnew ...)) lblnext)  
where ((ldomnew ...) lblnext) = morph-children[(ldom ...), (nodes ...), (node ...), child-ns[tag, nsn], lbl]  
morph-node[(ldom, nodes, nodet, ns, lbl] =  
mount[(nodet, ns, lbl]
```

Figure 38. morphNode: same leaf kind, or same template and key, keeps the node.

morph-children

```
morph-children[(ldom ...), (nodeold ...), (nodenew ...), ns, lbl] =  
(keep-foreign[(ldom ...), (ldomnew ...)] lblnext)  
where (match-result ...) = match-siblings[(nodeold ...), (nodenew ...)],  
      ((ldomnew ...) lblnext) = morph-seq[split-groups[owned[(ldom ...)], (nodeold ...), (nodeold ...), (nodenew ...),  
      (match-result ...), ns, lbl]
```

morph-seq

```
morph-seq[lgroups, (nodeold ...), (), (), ns, lbl] =  
  (()) lbl  
morph-seq[lgroups, (nodeold ...), (nodeo node ...), (match-resulto match-result ...), ns, lbl] =  
  ((ldomo ... ldom ...) lbl2)  
where ((ldomo ...) lbl1) = morph-one[lgroups, (nodeold ...), nodeo, match-resulto, ns, lbl],  
      ((ldom ...) lbl2) = morph-seq[lgroups, (nodeold ...), (node ...) , (match-result ...) , ns, lbl1]
```

morph-one

```
morph-one[lgroups, (nodeold ...), node, fresh, ns, lbl] =  
  mount[node, ns, lbl]  
morph-one[lgroups, (nodeold ...), (group key?t (nodet ...)), index, ns, lbl] =  
  morph-children[nth(lgroups, index), (nodes ...), (nodet ...), ns, lbl]  
  where (group key?s (nodes ...)) = nth((nodeold ...), index)  
morph-one[lgroups, (nodeold ...), node, index, ns, lbl] =  
  morph-node[(first nth(lgroups, index)), nodes, node, ns, lbl]  
  where nodes = nth((nodeold ...), index),  
    (not (redex-match? Morph (group any ...) nodes)),  
    (not (redex-match? Morph (group any ...) node))  
morph-one[lgroups, (nodeold ...), node, index, ns, lbl] =  
  mount[node, ns, lbl]
```

owned: the nodes render made

```
owned[(ldom ...)] = (filter (λ (d) (not (eq? (first d) foreign))) (ldom ...))
```

keep-foreign: foreign nodes stay

```
keep-foreign[(ldomold ...), (ldomnew ...)] =  
  ( ([(lead runs) (foreign-runs (ldomold ...))] )  
    (define kept (map second (ldomnew ...)))  
    (append lead  
      (append* ( ([(d (ldomnew ...))] )  
        (cons d (hash-ref runs (second d) ())))  
      (append* (for/list ([l run] runs) #:unless (member l kept)) run))))
```

erase: forget identities

```
erase[(text lbl string)] =  
  (text string)  
erase[(comment lbl string)] =  
  (comment string)  
erase[(foreign lbl dom)] =  
  dom  
erase[(element lbl ns tag (prop ...) (ldom ...))] =  
  (element ns tag (prop ...) (erase[(ldom ...) ]))
```

erase-owned: what render made

```
erase-owned[(text lbl string)] =  
  (text string)  
erase-owned[(comment lbl string)] =  
  (comment string)  
erase-owned[(element lbl ns tag (prop ...) (ldom ...))] =  
  (element ns tag (prop ...) erase-owned-all[(ldom ...) ])
```

Checked: erase(morph(mount(old), old, new)) = render(new); morph(x, x) keeps every label; with foreign nodes anywhere, erase-owned(morph(...)) = render(new) and the foreign nodes stay.

Figure 39. morphChildren: matched targets reuse their source's DOM; the rest mount.