Reactive Programming

Table of Contents

1. Concept

In version 0.6.0, Red introduced support for "reactive programming" to help reduce the size and complexity of Red programs. Red's reactive model relies on dataflow and object events to construct a directed graph and propagate changes in objects. It uses a "push" model. More specifically, Red implements the object-oriented reactive programming model, where only object fields can be the source of change.

The reactive API and its use are simple and practical, even if the description is abstract. Here are some diagrams to help visualize reactive relationships.

[react-simple] | *react-simple.png*

Graph A & B show simple relations between one or several reactors (objects that act as a reactive source).

[react-graph] | *react-graphs.png*

Graphs C, D & E show chained reactions, where some targets are, themselves, reactors, setting up a chain of relations that can have any shape.

Reactions are run asynchronously, when a source field's value is changed. The reaction relationship is maintained until the reaction is explicitly destroyed using `react/unlink` or `clear-reactions`.

Only the source objects in a reactive expression need to be a reactor. The target can be a simple object. If the target is also a reactor, reactions are chained and a graph of relations is constructed

implicitly.

NOTE

- Red's reactive support could be extended in the future to support a "pull" model.
- This is not a FRP framework, though event streams could be supported in the future.
- The Red/View GUI engine relies on *face!* objects in order to operate on graphic objects. Faces are reactors, and they can be used for setting reactive relations between faces or with non-reactor objects.

1.1. Glossary

Expression	Definition
reactive programming	A programming paradigm, a subset of dataflow programming, based on events "pushing" changes.
reaction	A block of code which contains one or more reactive expressions.
reactive expression	An expression which references at least one reactive source.
reactive relation	A relation between two or more objects implemented using reactive expressions.
reactive source	A `path!` value referring to a field in a reactive object.
reactive formula	A reaction which returns the last expression result on evaluation.
reactive object	An object whose fields can be used as reactive sources.
reactor	Alias for "reactive object".

1.2. Static Relations

The simplest form of reactions is a "static relation" created between *named* objects. It is *static* because it statically links objects. It uniquely applies to its source reactors, and cannot be re-used for other objects.

Example 1

```
view [
    s: slider return
    b: base react [b/color/1: to integer! 255 * s/data]
]
```

This example sets a reactive relation between a slider named `s` and a base face named `b`. When the slider is moved, the red component of the base face's background color is changed accordingly. The reactive expression cannot be re-used for a different set of faces. This is the simplest form of reactive behavior for graphic objects in Red/View.

Example 2

```
vec: reactor [x: 0 y: 10]
box: object [length: is [square-root (vec/x ** 2) + (vec/y ** 2)]]
```

Another form of static relation can be defined using the `is` operator, where the value of the reaction evaluation is set to a word (in any context).

This example is not related to the GUI system. It calculates the length of a vector defined by `vec/x` and `vec/y` using a reactive expression. Once again the source object is statically specified by its name (`vec`) in the reactive expression.

Example 3

```
a: reactor [x: 1 y: 2 total: is [x + y]]
```

The word `total` above has its value set to the `x + y` expression. Each time the `x` or `y` values change, `total` is immediately updated. Notice that paths are not needed in this case, to specify the reactive sources, because `is` is used directly inside the reactor's body and so knows its context.

Example 4

```
a: reactor [x: 1 y: 2]
total: is [a/x + a/y]
```

This variation of Example 3 shows that a global word can also be the target of a reactive relation (though it can't be the source). This form is the closest to that of a spreadsheet's (e.g. Excel) formula model.

NOTE	due to the size of global context, making it reactive (as above with `total`) could have significant performance overhead, though that could be overcome in the future.

1.3. Dynamic Relations

Static relations are easy to specify, but they don't scale well if you need to provide the same reaction to a number of reactors, or if the reactors are anonymous (reminder: all objects are anonymous by default). In such cases, the reaction should be specified with a **function** and `react/link`.

Example

```
;-- Drag the red ball up and down with the mouse. Watch how the other balls react.

win: layout [
    size 400x500
    across
    style ball: base 30x30 transparent draw [fill-pen blue circle 15x15 14]
    ball ball ball ball ball ball ball b: ball loose
    do [b/draw/2: red]
]

follow: func [left right][left/offset/y: to integer! right/offset/y * 108%]

faces: win/pane
while [not tail? next faces][
    react/link :follow [faces/1 faces/2]
    faces: next faces
]
view win
```

In this example, the reaction is a function (`follow`) which is applied to the ball faces by pairs. This creates a chain of relations that link all the balls together. The terms in the reaction are parameters, so they can be used for different objects (unlike static relations).

2. API

2.1. react

Syntax

```
react <code>
react/unlink <code> <source>

react/link <func> <objects>
react/unlink <func> <source>

react/later <code>

<code>    : block of code that contains at least one reactive source (block!).
<func>    : function that contains at least one reactive source (function!).
<objects> : list of objects used as arguments to a reactive function (block! of
object! values).
<source>  : 'all word, or an object, or a list of objects (word! object! block!).

Returns   : <code> or <func> for further references to the reaction.
```

Description

`react` sets a new reactive relation, which contains at least one reactive source, from a block of code (sets a "static relation") or a function (sets a "dynamic relation" and requires the `/link` refinement). In both cases, the code is statically analyzed to determine the reactive sources (in the form of `path!` values) that refer to reactor fields.

By default, the newly formed reaction **is called once on creation** before the `react` function returns. This can be undesirable in some cases, so can be avoided with the `/later` option.

A reaction contains arbitrary Red code, one or more reactive sources, and one or more reactive expressions. It is up to the user to determine the set of relations which best fit their needs.

The `/link` option takes a function as the reaction and a list of arguments objects to be used in evaluation of the reaction. This alternative form allows dynamic reactions, where the reaction code can be reused with different sets of objects (the basic `react` can only work with statically *named* objects).

A reaction is removed using the `/unlink` refinement and with one of the following as a `<source>` argument:

- The `'all` word, will remove all reactive relations created by the reaction.
- An object value, will remove only relations where that object is the reactive source.
- A list of objects, will remove only relations where those objects are the reactive source.

`/unlink` takes a reaction block or function as argument, so only relations created from **that** reaction are removed.

2.2. is

Syntax

```
<word>: is [<body>]
<word>: is [[<default>] <body>]

<word>    : word to be set to the result of the reaction (set-word!).
<body>    : arbitrary Red code that contain at least one reactive source.
<default> : arbitrary Red code that return a value used as initial default for <word>.
```

Description

`is` creates a reactive formula whose result will be assigned to a word. The `<code>` block can contain references to both the wrapping object's fields, if used in a reactor's body block, and to external reactor's fields. For specifying a default value, a block returning the default value can be inserted as the first element of the reactive formula block. This is especially useful when using forward reference(s) for reactive source(s) that are unset at the time of the formula's evaluation.

NOTE	This operator creates reactive formulas which closely mimic Excel's formula model.

Examples

```
a: reactor [x: 1 y: 2 total: is [x + y]]

a/total
== 3
a/x: 100
a/total
== 102
```

```
reactor [a: 1 b: is [[none] c] c: is [a < 4]]
== make object! [
    a: 1
    b: true
    c: true
]
```

2.3. react?

Syntax

```
react? <obj> <field>
react?/target <obj> <field>

<obj>   : object to check (object!).
<field> : object's field to check (word!).

Returns : a reaction (block! function!) or a none! value.
```

Description

`react?` checks if an object's field is a reactive source . If it is, the first reaction found where that object's field is present as a source, will be returned, otherwise `none` is returned. `/target` refinement checks if the field is a target instead of a source, and will return the first reaction found targeting that field or `none` if none matches.

2.4. clear-reactions

Syntax

```
clear-reactions
```

Description

Removes all defined reactions, unconditionally.

2.5. dump-reactions

Syntax

```
dump-reactions
```

Description

Outputs a list of registered reactions for debug purposes.

3. Reactive Objects

Ordinary objects in Red do not exhibit reactive behaviors. In order for an object to be a reactive source, it needs to be constructed from one of the following reactor prototypes.

3.1. reactor!

Syntax

```
make reactor! <body>

<body> : body block of the object  (block!).

Returns : a reactive object.
```

Description

Constructs a new reactive object from the body block. In the returned object, setting a field to a new value will trigger reactions defined for that field. `reactor` function is a convenient shortcut for this type of constructor.

> **NOTE** The body may contain `is` expressions.

3.2. deep-reactor!

Syntax

```
make deep-reactor! <body>

<body> : body block of the object  (block!).

Returns : a reactive object.
```

Description

Constructs a new reactive object from the body block. In the returned object, setting a field to a new value or changing a series the field refers to, including nested series, will trigger reactions defined for that field. `deep-reactor` function is a convenient shortcut for this type of constructor.

NOTE	The body may contain `is` expressions.

Example

This shows how change to a series, even a nested one, triggers a reaction.

NOTE	It is up to the user to prevent cycles at this time. For example, if a `deep-reactor!` changes series values in a reactor formula (e.g. `is`), it may create endless reaction cycles.

```
r: deep-reactor [
    x: [1 2 3]
    y: [[a b] [c d]]
    total: is [append copy x copy y]
]
append r/y/2 'e
print mold r/total
```