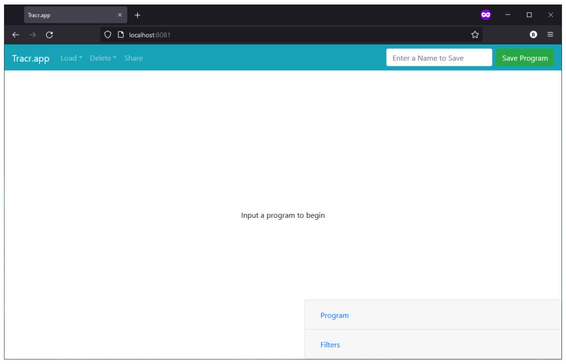
1.3 Example

To best illustrate how TRACR can be used, a brief example is presented in this section to showcase some of its features.

When opening the application (by navigating to a URL), the user is presented the new page screen, as seen in Figure 1.2.



Input a program to begin

Figure 1.2: New page

To begin with, we will use one of the example programs, "twice-fact", available in the interface. This sample program uses 2 function definitions to compute `twice fact 2`, where `twice` and `fact` functions are defined as follows (shown in Haskell):

```haskell
twice :: (a -> a) -> a -> a
twice f x = f (f x)

fact :: Int -> Int
fact x | x == 0    = 1
       | otherwise = fact (x-1) * x
```

The function `twice` will apply a function twice, and `fact` will compute the factorial of it's input. In the example program, `fact` is applied to `2`, and then `fact` is again applied to that value. In Xtra, this program looks a little different but functions identically. The same syntax is actually valid in Haskell. The exact text of the program we will load is below:

```
let twice = \f -> \x -> f (f x) in let fact = \x -> case x of 0 -> 1; y ->
↪  x*fact (x-1); in twice fact 2
```

To load this example program, click "Load" on the right side of the navigation bar to expand the menu, and click the program name under the "Example Programs" header (see Figure 1.3).

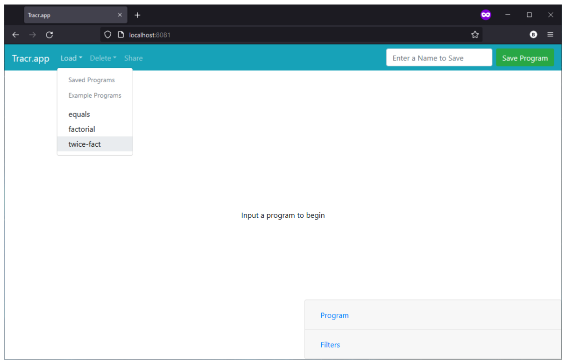
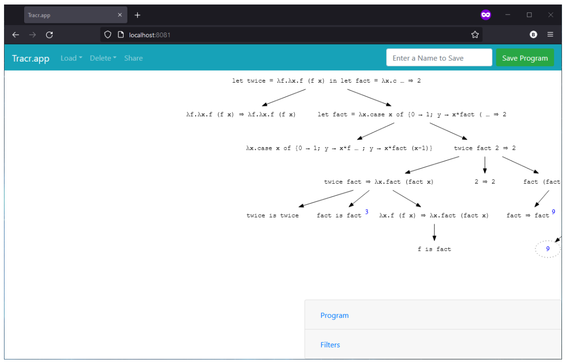


Figure 1.3: Selecting an example program to load

This will load the selected program, displaying the trace in the middle area of the screen (see Figure 1.4).



```
let twice = λf.λx.f (f x) in let fact = λx.c … ⇒ 2
λf.λx.f (f x) ⇒ λf.λx.f (f x)
let fact = λx.case x of {0 → 1; y → x*fact ( … ⇒ 2
λx.case x of {0 → 1; y → x*f … ; y → x*fact (x-1)}
twice fact 2 ⇒ 2
twice fact ⇒ λx.fact (fact x)
2 ⇒ 2
fact (fact
twice is twice
fact is fact 3
λx.f (f x) ⇒ λx.fact (fact x)
fact ⇒ fact 9
f is fact
9
```

Figure 1.4: Example program loaded

The view of the trace can be panned and zoomed using a mouse to drag the trace, or the scroll wheel to zoom. In Figure 1.5, the trace is panned left and zoomed out to fit the screen. Also, the full node contents of the parent node can be seen as a tooltip. TRACR will shorten nodes that are very long to allow them to better fit the screen. However, their full contents can always be seen by hovering the cursor over the node.

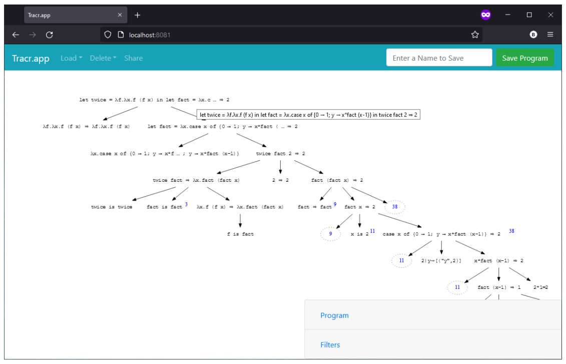

Tracr.app
localhost:8081
Tracr.app
Load
Delete
Share
Enter a Name to Save
Save Program
let twice = λf.λx.f (f x) in let fact = λx.c … ⇒ 2
let twice = λf.λx.f (f x) in let fact = λx.case x of {0 → 1; y → x*fact (x-1)} in twice fact 2 ⇒ 2
λf.λx.f (f x) ⇒ λf.λx.f (f x)
let fact = λx.case x of {0 → 1; y → x*fact (… ⇒ 2
λx.case x of {0 → 1; y → x*f … ; y → x*fact (x-1)}
twice fact 2 ⇒ 2
twice fact ⇒ λx.fact (fact x)
2 ⇒ 2
fact (fact x) ⇒ 2
twice is twice
fact is fact 3
λx.f (f x) ⇒ λx.fact (fact x)
fact ⇒ fact 9
fact x ⇒ 2
38
f is fact
9
x is 2 11
case x of {0 → 1; y → x*fact (x-1)} ⇒ 2 38
11
2|y~[("y",2)]
x*fact (x-1) ⇒ 2
11
fact (x-1) ⇒ 1
2*1=2
Program
Filters

Figure 1.5: Fitting the trace to the screen

To begin reducing the size of the trace, the Filters block can be expanded in the bottom right of the screen by clicking on "Filters". This will reveal a green button that can be clicked to add a new filter. In Figure 1.6, one filter has been added showing its default setting.

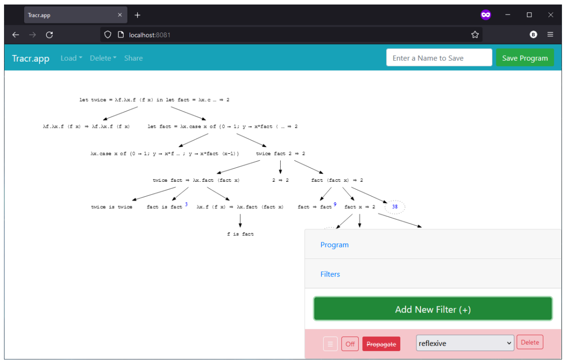


Figure 1.6: Adding a filter

The first filter we will apply is `reflexive`. This filter will target nodes in the trace which are the result of a reflexive judgement. Such a node is of the form $x \Rightarrow x$. This type of node generally doesn't add any additional information, and we can improve the explanatory power of the trace by hiding unneeded nodes.

To enable this filter, the “Off” button can be pressed to toggle it to “On”. This will change the background of the filter to green, indicating that it is enabled (see Figure 1.7). When changes are made to the program or to the filters, TRACR will automatically update the trace on the screen. Here the updated trace can be seen with the reflexive filter applied.

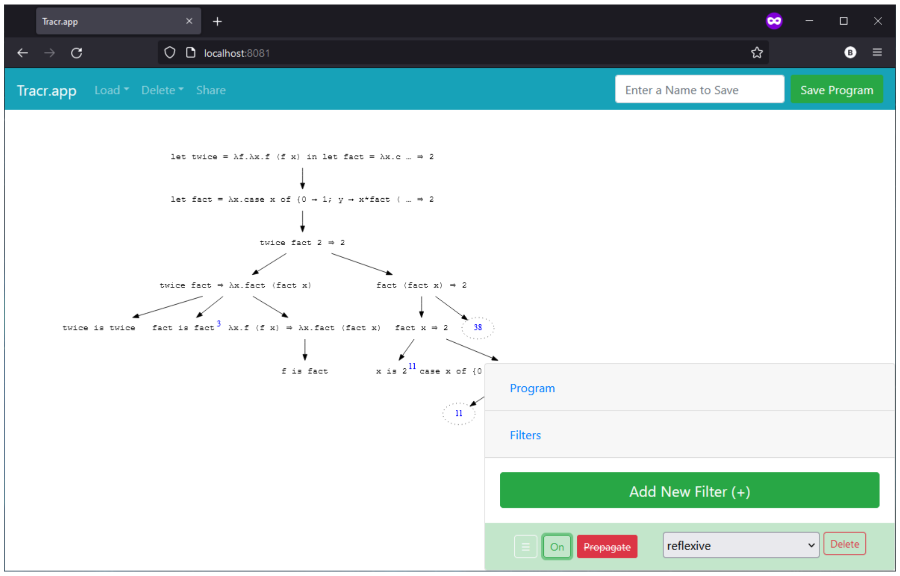


Figure 1.7: Enabling a filter

Filters also have another option that can be toggled: “Propagate”. This has the effect of propagating the changes into other nodes (instead of only hiding nodes selected by the filter).

Propagation could be useful when the binding of a value to a variable is hidden. In such a case, only the variable name remains. By propagating the result of the hidden node, the value replaces the variable name in the still visible nodes.

Propagating some filters may be more useful than others, and this may be program dependent. Determining the best set of filters is an exercise for the user, and TRACR aims to assist in the process of iteration.

In Figure 1.8, the new trace can be seen after enabling propagation for the reflexive filter.

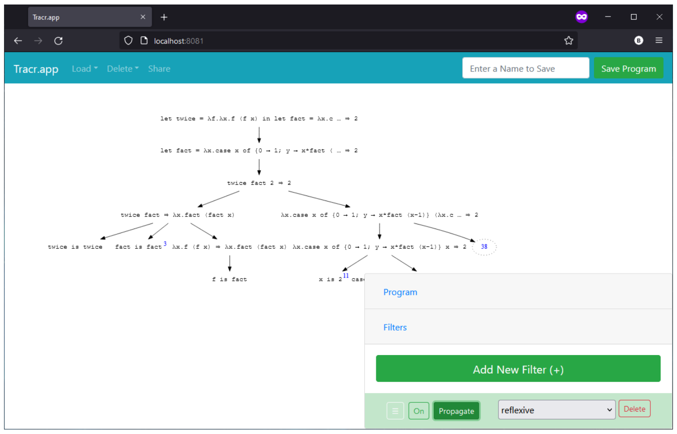

Tracr.app
localhost:8081
Tracr.app
Load
Delete
Share
Enter a Name to Save
Save Program
let twice = λf.λx.f (f x) in let fact = λx.c … ⇒ 2
let fact = λx.case x of {0 → 1; y → x*fact (… ⇒ 2
twice fact 2 ⇒ 2
twice fact ⇒ λx.fact (fact x)
λx.case x of {0 → 1; y → x*fact (x-1)} (λx.c … ⇒ 2
twice is twice
fact is fact 3
λx.f (f x) ⇒ λx.fact (fact x)
λx.case x of {0 → 1; y → x*fact (x-1)} x ⇒ 2
38
f is fact
x is 2 11
Program
Filters
Add New Filter (+)
On
Propagate
reflexive
Delete

Figure 1.8: Enabling a propagation for a filter

Next, a new filter can be added by pressing the “Add New Filter” again. This will add a new filter with the default options. This filter can then also be customized. The pattern filter targets nodes which are the result of pattern matching evaluations. These nodes contain a ⇝ symbol, which is used to denote pattern matching in the trace output. Such a node can be seen in near the center of the screen in Figure 1.9.

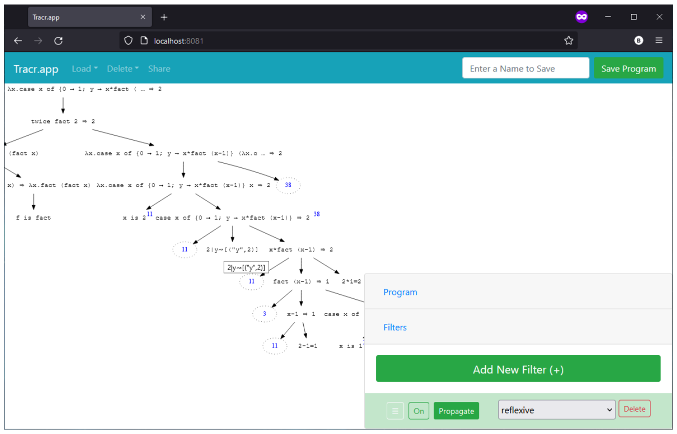
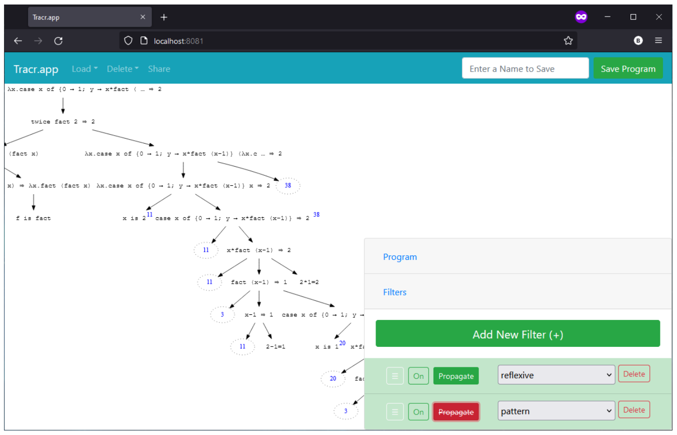


Figure 1.9: Pattern matching node

Next, we'll apply the pattern filter to hide this node in Figure 1.10

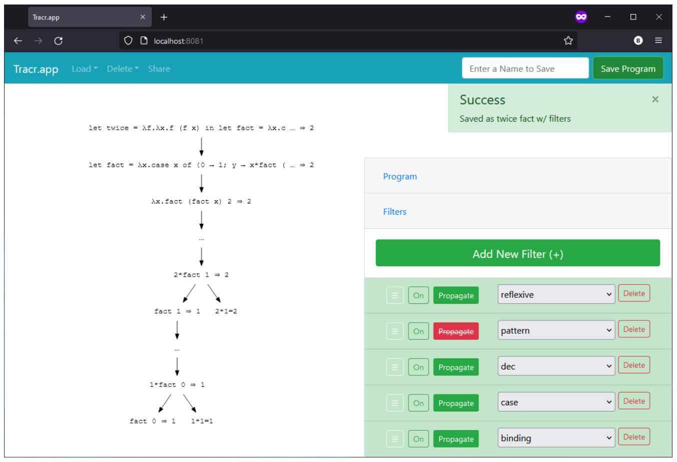


```
λx.case x of {0 → 1; y → x*fact ( … ⇒ 2
twice fact 2 ⇒ 2
(fact x)
λx.case x of {0 → 1; y → x*fact (x-1)} (λx.c … ⇒ 2
x) ⇒ λx.fact (fact x)
λx.case x of {0 → 1; y → x*fact (x-1)} x ⇒ 2
38
f is fact
x is 2 11
case x of {0 → 1; y → x*fact (x-1)} ⇒ 2 38
11
x*fact (x-1) ⇒ 2
11
fact (x-1) ⇒ 1
2*1=2
3
x-1 ⇒ 1
case x of {0 → 1; y →
11
2-1=1
x is 1 20
x*fa
20
fac
3
```

reflexive
pattern

Figure 1.10: Adding the pattern filter

This process of adding and modifying filters can continue until the user is satisfied with the result. At any point, the program and the set of filters can be saved. The saved results will be saved to the browser, able to be loaded whenever the user would like to view the result, or iterate on their work. To save the work, a unique name can be typed in the text box in the upper left portion of the screen, clicking the "Save Program" button to confirm.



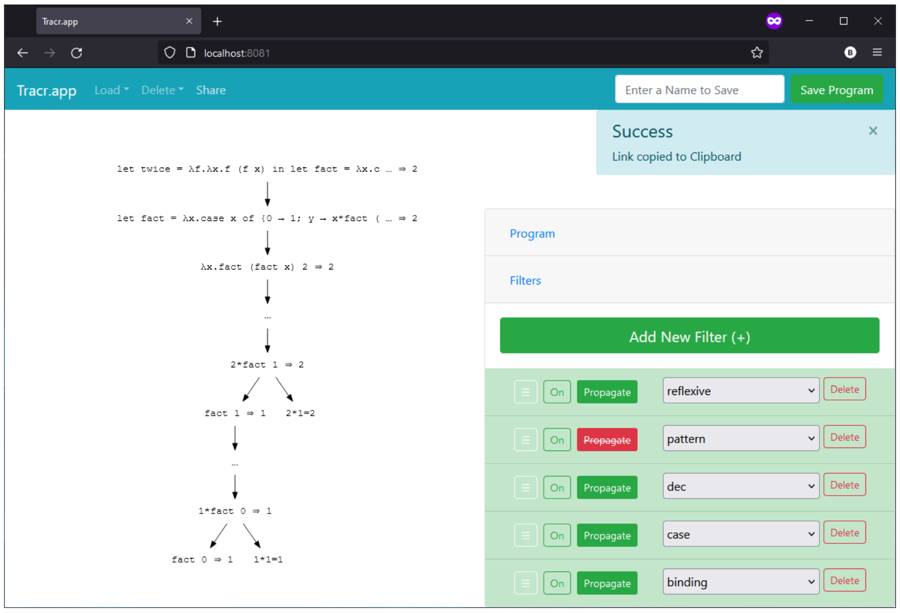
Success
Saved as twice fact w/ filters

```
let twice = λf.λx.f (f x) in let fact = λx.c … ⇒ 2
let fact = λx.case x of {0 → 1; y → x*fact ( … ⇒ 2
λx.fact (fact x) 2 ⇒ 2
…
2*fact 1 ⇒ 2
fact 1 ⇒ 1
2*1=2
…
1*fact 0 ⇒ 1
fact 0 ⇒ 1
1*1=1
```

reflexive
pattern
dec
case
binding

Figure 1.11: Saving a program

To share a program and filters with someone else, the "Share" button in the top navigation bar can be used. Clicking "Share" will copy a link to the clipboard. The link can be used to load the same program (and filters) as if currently displayed on the screen. An alert box in the top right will show that the link has been copied successfully.



Success
Link copied to Clipboard

```
let twice = λf.λx.f (f x) in let fact = λx.c … ⇒ 2
let fact = λx.case x of {0 → 1; y → x*fact ( … ⇒ 2
λx.fact (fact x) 2 ⇒ 2
…
2*fact 1 ⇒ 2
fact 1 ⇒ 1    2*1=2
…
1*fact 0 ⇒ 1
fact 0 ⇒ 1    1*1=1
```

On	Propagate	reflexive
On	~~Propagate~~	pattern
On	Propagate	dec
On	Propagate	case
On	Propagate	binding

Figure 1.12: Sharing a program