Proto Quick Start

Jacob Beal

Last Revision: May 27, 2008

This is a quick-start guide for Proto, giving a quick tour of commonly used features of the language and simulator. For installation instructions, see the **Proto Installation Guide**. For a tutorial on the Proto language, see the document **Thinking In Proto**. For a reference to the Proto language, see the **Proto Language Reference**. For a usee manual for the simulator, see the **Proto Simulator User Manual**. For information on how to extend the functionality of the simulator, see the **Proto Simulator Developer Reference**.

1 Credits for Proto

The Proto language was developed in partnership by Jonathan Bachrach and Jacob Beal. Jonathan Bachrach is the primary programmer for the Proto compiler, kernel, and 1st generation simulator. Jacob Beal is the primary programmer for the 2nd generation simulator.

Additional programming by: Joshua Horowitz, Omari Stephens, Mark Tobenkin, Dan Vickery

2 Smoke Test

Let us assume you have already successfully built Proto. Go to the directory `resrc/` and type:

```
./proto -n 1000 -r 10 -l -c -T -v "(red (gradient (once (< (rnd 0 1) 0.01))))"
```

You should see a green network with blue unreadable text at the intersections. Red dots will spread through the screen from several starting locations as a magenta number in the lower left counts upward.

Drag with your left mouse button to rotate the display. You will see that the red dots form a mountainous landscape. Drag with your right mouse button to zoom and get a better view.

Let's break down this command: `-n 1000` means use 1000 devices and `-r 10` means connect devices within 10 meters of one another. The `-c` means show the network: it's the green thing; hit 'c' to turn it off. You'll notice the magenta number in the lower right go up when you do that: it's displaying how many frames per second the simulator is rendering, and the one in the lower left is displaying the number of simulated seconds that have elapsed. The timing display was invoked by `-T`; hit 'T' (shift-t) to turn them off.

Now you are looking at just the blue numbers and the red dots, and it's time to delve into the Proto expression a little: it's the big thing inside the quotes. The expression `(rnd 0 1)` means that all 1000 devices should pick random floating point numbers between 0 and 1. The comparison `(< (rnd 0 1) 0.01)` turns these into a field of boolean numbers which is true at approximately 10 devices. The `once` function wrapped around this says to do this once and remember the result. Then we feed this boolean field into the `gradient` function, which finds the distance from each device to the nearest device with a true value—that is, the nearest one that picked a random number less than 0.01. Finally, these distances are fed to the `red` LED actuator that produces those dots on the screen. If you zoom in one the blue numbers, you'll see that the dots float above the number at a height equal to the number. The blue numbers are the output of the Proto expression, and are visible because of the `-v`: you can hit 'n' to turn them on and off. The LEDs are visible because of the `-l` and you can use 'L' to turn them on and off.

3 Proto Language Essentials

Here are some essentials to the Proto language, along with the most frequently used functions.

Evaluation Proto is a purely functional language. Proto is written using s-expressions in a manner very similar to Scheme. Evaluating a Proto expression produces a program: a dataflow graph that may be evaluated against a space to produce an evolving field of values at every point on the space.

Data Types All Proto expressions produce fields that map every point in space to a value. The values produced are categorized into four basic types: fields, lambdas, tuples, and scalars. A number is a scalar or vector (tuple with scalar values), a local is anything but a field, and a boolean is a scalar interpreted as a logical value: false is 0, anything else is true.

Namespaces and Bindings Proto is a lexically scoped language. Names are not case sensitive. Bindings contain values and are looked up by name. Lexical bindings are visible only within the scope in which they are bound, and shadow bindings of the same name from enclosing scopes.

When the Proto compiler encounters an unknown identifier *name*, it searches its path for a file named *name*`.proto`. If it finds such a file, then it loads the contents of the file and looks up the identifier again. Definitions in subdirectories can be accessed with identifiers of the form *dir*`/`*name*.

> `(def .name (.arg ...) ,@body)`: Define a function `name` in the current scope, with as many arguments as there are `arg` identifiers. The body is evaluated within an extended scope where the `arg` identifiers are bound to arguments to the function. `fun`, which omits `name`, creates anonymous functions.
>
> `(let ((.var ,value) ...) ,@body)`: Extends scope, binding all `var` identifiers to their associated `value` in parallel. The `body` is evaluated in the extended scope. `let*` is like let, except that identifies are bound sequentially, so later ones can use earlier ones in their definition.

Control Flow

> `(all ,@forms)`: All `forms` are evaluated in parallel and the value of the last form returned.
>
> `(mux ,test ,true ,false)`: Evaluates both `true` and `false` expressions. When `test` is true, returns the result of the `true` expression, otherwise returns the result of the `false` expression. The `true` and `false` expressions must return the same type.
>
> `(if ,test ,true ,false)`: Restricts execution to subspaces based on `test`. Where `test` is true, the `true` expression is evaluated; where `test` is false, the `false` expression is evaluated. The `true` and `false` expressions must return the same type.

State Because Proto is a purely functional language, we create state using feedback loops. A state variable is initialized at some value, then evolves that value forward in time. In regions where the feedback loop is not evaluated, the state variable is reinitialized, resuming evolution when the feedback loop begins to be evaluated again.

For example, the expression:

```
(rep t 0 (+ t (dt)))
```

creates a timer that returns how long evaluation has been proceeding at each device.

`(letfed ((.var ,init ,evolve) ...) ,@body)`: Creates a state variable for each `var`. `var` is initially bound to the value of expression `init`, and at each time step the state is evolved forward using expression `evolve`. The body is evaluated within an extended scope including the state variables.

In the `evolve` expression, each `var` is bound to an old value and `(dt)` is set to the time since the last step. All `init` and `evolve` expressions are evaluated in parallel, so no variable can reference another value in its `init`, but variables can use one another's old values in their `evolve` statements. *Capable of violating the continuous space/time abstraction..*

`(rep .var ,init ,evolve)`: Create a single feedback variable and return its value. Equivalent to `(letfed ((.var ,init ,evolve)) .var)`. *Capable of violating the continuous space/time abstraction.*

`(once ,expr)`: Evaluates `expr` once, then always returns that value.

Logic and Arithmetic

- Logical operators: `and`, `or`, `not`
- Constants: `(inf)`, `(e)`, `(pi)`
- Arithmetic: `+`, `-`, `*`, `/`, `neg` (negation)
- Comparison: `=`, `<`, `>`, `<=`, `>=`
- Functions: `pow`, `min`, `max`, `sqrt`, `abs`, `sin`, `cos`, `atan2`
- Random Numbers: `(rnd ,min ,max)` gives a floating point random number in the range $[min, max]$.
- Vectors: `(vdot ,a ,b)`, `(normalize ,v)`, `(polar-to-rect ,v)`, `(rect-to-polar ,v)`

Tuples

`(tuple ,v ++)`: Creates a tuple with the set of `v` arguments as its elements.

`(elt ,tuple ,i)`: Returns the `i`th element of `tuple`, counting from zero. `1st`, `2nd`, and `3rd` are functions for getting elements 0, 1, and 2.

Neighborhoods There are two types of neighborhood functions: functions that create fields, and functions that summarize fields into local values. In between, any pointwise function can be applied to fields, producing a field whose values are the result of applying the pointwise operation to the values of the input fields.

- `(nbr ,expr)`: Returns a field mapping neighbors to their values of `expr`.
- `(nbr-range)`, `(nbr-angle)`, `(nbr-lag)`, `(nbr-vec)`: return a field of distances, bearings, time lags, and vectors to neighbors, respectively.
- `min-hood`, `max-hood`, `all-hood`, `any-hood`, `int-hood`: summarize a field into a scalar that is, respectively, minimum, maximum, for-all, existence, and integral over the values of the field.

  `(fold-hood ,fold ,base ,value)`: Collects `value` from each of the neighbors, then folds these into a summary value, using `fold` to combine elements into `base` one at a time. *Capable of violating the continuous space/time abstraction.*

Sensor and Actuators Actuators reset themselves to a null value whenever they are not actively being invoked. Thus, for example,

```
(if (sense 1) (swim (tup 2)) (red (tup 1)))
```

will cause devices move to the right only when `(sense 1)` is true, and to turn on their red LED only when `(sense 1)` is false.

`(swim ,velocity)`: Attempt to move at `velocity`. The return echoes `velocity`.

`(red ,n)`: Set red LED to intensity `n`. Intensity ranges from 0 to 1, but overloading of display can show values outside this range. The return echoes `n`. `green` and `blue` are identical, but set the green and blue LEDs instead.

`(probe ,value ,i)`: Posts `value` to the `i`th probe (valid indices are 0 to 2).

`(sense ,i)`: Returns the `i`th user sensor value.

`(clone ,now)`: When `now` is true, the device attempts to reproduce. The return echoes `now`.

`(die ,now)`: When `now` is true, the device attempts to suicide. The return echoes `now`.

`(coord)`: Returns the device's estimated coordinates.

`(radio-range)`: Returns the maximum expected range at which devices can communicate.

Library Functions These are not primitive functions, but are frequently used building blocks which have been included in Proto's distribution library, in the directory `lib/`.

`(distance-to ,source)`: Calculates the shortest-path distance from every device to the set of devices where `source` is true. The function `gradient` is an alias.

`(broadcast ,source ,value)`: Flow `value` outward from devices in the `source` to all other devices. Each device takes its value from the nearest `source` device.

`(dilate ,source ,d)`: Returns true for every device within distance `d` of the `source`.

`(distance ,region1 ,region2)`: Calculates the distance between `region1` and `region2` and broadcasts it everywhere.

`(disperse)`: Devices repel from one another using spring forces.

`(dither)`: Devices wander randomly in a 2D plane.

`(elect)`: Devices choose a leader by mutual exclusion and maintain precisely one leader within a given distance.

`(timer)`: Return the length that this device has been evaluating this expression (i.e. not going in different branches of an `if`)

.

4 Simulator Essentials

Key: `q`
Quit the simulator.

Argument: `-v`, Key: `n`
Show value computed at each device. Toggled by key.

Argument: `-sv`, Key: `v`
When device outputs a 2- or 3-tuple, display it as a vector (toggled by key).

Argument: `-T`, Key: `T`
Display simulator time in lower left corner and frames-per-second in lower right corner. Toggled by key.

Argument: `-step`, Key: `s`
Use stepping mode, advancing one step on key `s`

Key: `x`
Execute freely (ending stepping mode).

Argument: `-s N`
Set simulated seconds per step, default 0.01/`ratio`

Display:

Argument: `-f`, Key: `f`
Full screen display (toggled by key)

Key: `z`
Reset display to initial view.

Key: ARROW KEYS
Shift simulation display in the direction of the arrow, in the simulation's coordinate system.

Mouse: LEFT DRAG
Rotate display.

Mouse: RIGHT DRAG
Zoom display; toward center zooms out, away zooms in.

Selection:

Mouse: LEFT CLICK
Select the devices clicked on.

Mouse: RIGHT CLICK
Select, then print the state of the selected device(s) to standard out.

Mouse: SHIFT LEFT DRAG
Toggle the selection of all devices in a rectangular display region.

Mouse: SHIFT RIGHT DRAG
Move selected devices.

Device Distribution Simulations are created with `n` devices distributed through a bounded 2D or 3D volume according to one of several distribution rules.

Argument: `-n N`
Number of devices.

Argument: `-3d`
Use a 3D distribution; default is 2D.

Argument: `-grid`
Distributes devices in a grid, rathen than uniformly randomly.

Unit Disc Radio Communication

Argument: `-r N`
Transmission range for radio, default 15.

Argument: `-ns N`
Set transmission range to get an expected neighborhood size of `N`. Overrides `-r`.

Argument: `-c`, Key: c
Display network connections (toggled by key).

Dynamics There are two dynamics packages: "simple dynamics," which does not handle collisions and allows instantaneous shifts in velocty, and "ODE dynamics," which is based on the Open Dynamics Engine, a Newtonian physics simulator. Simple dynamics is the default.

Argument: `-ODE`
Use ODE dynamics for physics.

Argument: `-rad N`
Radius of a device body, defaults from the initial distribution bounds: $\sqrt{0.087 * (width * height/n)}$

Argument: `-w`, Key: w
Use walls to keep devices inside the initial distribution bounds.

Argument: `-m`, Key: m
Enable movement (toggled by key).

Sensors and Actuators

Argument: `-probes N`, Key: p
Display the first `N` of the three probes. The `p` key cycles through how many are shown.

Argument: `-l`, Key: L
Display LEDs (toggled by key).

Key: t
Toggle user sensor 1 on selected devices.

Key: y
Toggle user sensor 2 on selected devices.

Key: u
Toggle user sensor 3 on selected devices.

Key: B
Clone selected devices.

Key: K
Kill selected devices.