ERGO on a Page (v1.5)

The *ERGO* File

An *ERGO* file for an application typically has three parts: a basic action theory, the definitions of some *ERGO* procedures (using `define` as usual), and an optional robotic interface. These three parts are described further below, and may be loaded from other files by using `include`. The *ERGO* file usually ends with a `main` function that calls `ergo-do` or one of the planning functions:

`(define (main) (ergo-do [#:mode` *mode*`]` *pgm*`))`

The *pgm* here is an expression that evaluates to an *ERGO* program. If the execution *mode* is specified, it should be one of `'offline`, `'first` (the default), or `'online`. (The robotic interface part is needed only when the mode is `'online`.)

Using Scheme

Within an *ERGO* file, *ERGO* can be intermingled with Scheme variables and function definitions that appear in the usual way. In what follows, we use "*fexpr*" to mean any Scheme expression where the fluents of the basic action theory (see `define-fluents` below) may appear as global variables.

Running *ERGO*

Once *ERGO* has been properly installed, it is possible to call the `main` function in an *ERGO* file called `my-ergo-app.scm` as follows:

```
> racket -l ergo -f my-ergo-app.scm -m
```

Basic Action Theory

A basic action theory has definitions for the fluents and actions.

Fluents

The fluents of a basic action theory are defined using one or more expressions in the file of the form

`(define-fluents`
  fluent ini
  ...
  fluent ini `)`

where each *fluent* is a symbol and each *ini* is a Scheme form that provides the value of the fluent in the initial state. This has the effect of defining the fluents as global variables that can then be used later in fexprs for actions and programs.

Any valid Scheme datum can be used as the value of a fluent, including lists, vectors, hash-tables, and functions.

Actions

Each action is defined by an expression of the following form:

`(define-action` *action*
  fluent fexpr
  ...
  fluent fexpr `)`

The fluents listed are those that are considered to be changed by the action. The value of the *fluent* after the action will be the value of the corresponding *fexpr* before the action. (All changes are considered to be done in parallel.)

The *action* in the definition can be a symbol or a list of symbols (*name var* ... *var*) for an action with parameters..

In addition, the *fluent* can be the special symbol `#:prereq` or `#:sensing`, in which case, the corresponding *fexpr* defines the prerequisite or sensing result of the action.

The `define-action` expression has the effect of defining the action itself as a global variable whose value is the action symbol (or a list of the action symbol and its arguments).

ERGO Programs

Each of the following expressions evaluates to an *ERGO* program:

`:nil`
: The program that always succeeds.

`:fail`
: The program that always fails.

`(:test` *fexpr*`)`
: Succeed or fail according to whether or not the current value of *fexpr* is true.

`(:act` *action*`)`
: Fail if the action has a prerequisite that evaluates to false, but succeed otherwise, and move to a new state where the fluents are updated as per its `define-action` (see above).

`(:begin` *pgm* ... *pgm*`)`
: Sequentially perform all of the programs.

`(:choose` *pgm* ... *pgm*`)`
: Nondeterministically perform one of the programs.

`(:if` *fexpr* pgm_1 pgm_2`)`
: Behave like pgm_1 if the current value of *fexpr* is true, but like pgm_2 otherwise.

`(:when` *fexpr pgm* ... *pgm*`)`
: Behave like `(:if` *fexpr* `(:begin` *pgm*...*pgm*`) :nil)`

`(:unless` *fexpr pgm* ... *pgm*`)`
: Behave like `(:when (not` *fexpr*`)` *pgm*...*pgm*`)`.

`(:until` *fexpr pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` repeatedly until the value of *fexpr* becomes true.

`(:while` *fexpr pgm* ... *pgm*`)`
: Behave like `(:until (not` *fexpr*`)` *pgm*...*pgm*`)`.

`(:star` *pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` repeatedly for some nondeterministically chosen number of times.

`(:for-all` *var list pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` repeatedly for all values of the variable *var* taken from the list *list*.

`(:for-some` *var list pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` for some value of *var* from the *list*, chosen nondeterministically.

`(:conc` *pgm* ... *pgm*`)`
: Concurrently perform all of the programs, nondeterministically interleaving any single steps.

`(:atomic` *pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` but with no interleaving from concurrent programs.

`(:monitor` pgm_1 pgm_2 ... pgm_n`)`
: Perform pgm_1 before every step of pgm_2, which is performed before every step of pgm_3, and so on.

`(:>>` *fexpr* ... *fexpr*`)`
: Succeed after evaluating the expressions (for effect).

`(:<<` *fexpr* ... *fexpr*`)`
: Like `:>>` except that evaluation only happens on failure.

`(:let ((`*var fexpr*`) ... (`*var fexpr*`))` *pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` in an environment where each variable *var* has the value of *fexpr*.

`(:wait)`
: Succeed after the next exogenous action happens.

`(:search` *pgm* ... *pgm*`)`
: Perform `(:begin` *pgm*...*pgm*`)` online, but using lookahead for any nondeterminism to guard against failure.

Note that obtaining the value of these expressions is not the same as executing them. Execution is what is done by `ergo-do` alone.

Robotic Interface

A robotic interface is defined by a set of expressions of the form:

`(define-interface 'out` *body*`)`
: The body should evaluate to a function of one argument (like `displayln`) that will send an endogenous action to an outside target, blocking until the target is ready.

`(define-interface 'in` *body*`)`
: The body should evaluate to a function of no arguments (like `read`) that will return the next exogenous action received from an outside source, blocking when none.

The bodies can perform whatever initialization is needed for the functions they return to work properly (such as opening files, or making TCP connections). More than one in and out interface can be defined. The functions `write-endogenous` and `read-exogenous` can be used as default bodies.