Lilo Language

Lilo is a formal specification language designed for describing, verifying and monitoring the behavior of complex, time-dependent systems.

Lilo allows you to:

- Write expressions using a familiar syntax with powerful temporal operators for defining properties over time.
- Define data structures using records to model your system's data.
- Structure your specifications using systems.

Types and Expressions

Lilo is an expression-based language. This means that most constructs, from simple arithmetic to complex temporal properties, are expressions that evaluate to a time-series value. This section details the fundamental building blocks of Lilo expressions.

Comments

Comment *blocks* start with `/*` and end with `*/`. Everything between these markers is ignored.

A *line comment* start with `//`, and indicates that the rest of the line is a comment.

Docstrings start with `///` instead of `//`, and attach documentation to various language elements.

Primitive types

Lilo is a typed language. The primitive types are:

- `Bool`: Boolean values. These are written `true` and `false`.
- `Int`: Integer values, e.g. `42`.
- `Float` Floating point values, e.g. `42.3`.
- `String`: Text strings, written between double-quotes, e.g. `"hello world"`.

Type errors are signaled to the user like this:

```
def x: Float = 1.02

def n: Int = 42

def example = x + n
```

The code blocks in this documentation page can be edited, for example try changing the type of `n` to `Float` to fix the type error.

Operators

Lilo uses the following operators, listed in order of precedence (from highest to lowest).

- Prefix negation: `-x` is the additive inverse of `x`, and `!x` is the negation of `x`.
- Multiplication and division: `x * y`, `x / y`.
- Addition and subtraction: `x + y` and `x - y`.
- Numeric comparisons:
  - `==`: equality
  - `!=`: non-equality
  - `>=`: greater than or equals
  - `<=`: less than or equals
  - `>`: greater than (strict)
  - `<`: less than (strict) Comparisons can be chained, in a consistent direction. E.g. `0 < x <= 10` means the same thing as `0 < x && x <= 10`.
- Temporal operators
  - `always phi`: `phi` is true at all times in the future.
  - `eventually phi`: `phi` is true at some point in the future.
  - `past phi`: `phi` was true at some time in the past.
  - `historically phi`: `phi` was true at all times in the past.
  - `will_change phi`: `phi` changes value at some point in the future.
  - `did_change phi`: `phi` changed value at some point in the past.
  - `phi since psi`: `phi` is true at all points in the past, from some point where `psi` was true.
  - `phi until psi`: `phi` is true at all points in the future until `psi` becomes true.

  Temporal operators can be qualified with intervals:

  - `always [a,b] phi`: `phi` is true at all times between `a` and `b` time units in the future.
  - `eventually [a,b] phi`: `phi` is true at some point between `a` and `b` time units in the future.
  - `phi until [a, b] psi`: `phi` is true at all points between now and some point between `a` and `b` time units in the future until `psi` becomes true.
  - Similar interval qualifications apply to other temporal operators.
  - One can use `infinity` in intervals: `[0, infinity]`.
- Conjunction: `x && y`, both `x` and `y` are true.
- Disjunction: `x || y`, one of `y` or `y` is true.

- Impliciation and equivalence:
  - `x => y`: if `x` is true, then `y` must also be true.
  - `x <=> y`: `x` is true if and only `y` is true.

Built-in functions

There are built-in functions:

- `float` will produce a `Float` from an `Int`:

  ```
  def n: Int = 42

  def x: Float = float(n)
  ```

- `time` will return the current time of the signal.

Conditional Expressions

Conditional expressions allow a specification to evaluate to different values based on a boolean condition. They use the `if-then-else` syntax.

```
if x > 0 then "positive" else "non-positive"
```

A key feature of Lilo is that `if`/`then`/`else` is an **expression**, not a statement. This means it always evaluates to a value, and thus the `else` branch is mandatory.

The expression in the `if` clause must evaluate to a `Bool`. The `then` and `else` branches must produce values of a compatible type. For example, if the `then` branch evaluates to an `Int`, the `else` branch must also evaluate to an `Int`.

Conditionals can be used anywhere an expression is expected, and can be nested to handle more complex logic.

```
// Avoid division by zero
def safe_ratio(numerator: Float, denominator: Float): Float =
  if denominator != 0.0 then
    numerator / denominator
  else
    0.0 // Return a default value

// Nested conditional
def describe_temp(temp: Float): String =
  if temp > 30.0
    then "hot"
  else if temp < 10.0
    then "cold"
  else
    "moderate"
```

Note that `if _ then _ else _` is *pointwise*, meaning that the condition applies to all points in time, independently.

Records

Records are composite data types that group together named values, called fields. They are essential for modeling structured data within your specifications.

The Lilo language supports anonymous, structurally typed, extensible records.

Construction and Type You can construct a record value by providing a comma-separated list of `field = value` pairs enclosed in curly braces. The type of the record is inferred from the field names and the types of their corresponding values.

For example, the following expression creates a record with two fields: `foo` of type `Int` and `bar` of type `String`.

```
{ foo = 42, bar = "hello" }
```

The resulting value has the structural type `{ foo: Int, bar: String }`. The order of fields in a constructor does not matter.

You can also declare a named record type using a `type` declaration, which is highly recommended for clarity and reuse.

```
/// Represents a point in a 2D coordinate system.
type Point = { x: Float, y: Float }

// Construct a value of type Point
def origin: Point = { x = 0.0, y = 0.0 }
```

Field punning When you already have a name in scope that should be copied into a record, you can *pun* the field by omitting the explicit assignment. A pun such as `{ foo }` is shorthand for `{ foo = foo }`.

```
def foo: Int = 42
def bar: String = "hello"

def record_with_puns = { foo, bar }
```

Punning works anywhere record fields are listed, including in record literals and updates. Each pun expands to a regular `field = value` pair during typechecking.

Path field construction Nested records can be created or extended in one step by assigning to a dotted path. Each segment before the final field refers to an enclosing record, and the compiler will merge the pieces together.

```
type Engine = { status: { throttle: Int, fault: Bool } }

def default_engine: Engine =
  { status.throttle = 0, status.fault = false }
```

The order of path assignments does not matter; the paths are merged into the final record. A dotted path cannot be combined with punning; write `{ status.throttle = throttle }` instead of `{ status.throttle }` when you need the path form.

Record updates with `with` Use `{ base with fields }` to copy an existing record and override specific fields. Updates respect the same syntax rules as record construction: you can mix regular assignments, puns, and dotted paths.

```
type Engine = { status: { throttle: Int, fault: Bool } }

def base: Engine =
  { status.throttle = 0, status.fault = false }

def warmed_up: Engine =
  { base with status.throttle = 70 }

def acknowledged: Engine =
  { warmed_up with status.fault = false }
```

All updated fields must already exist in the base record. Path updates let you rewrite deeply nested pieces without rebuilding the entire structure.

Projection To access the value of a field within a record, you use the dot (`.`) syntax. If `p` is a record that has a field named `x`, then `p.x` is the expression that accesses this value.

```
type Point = { x: Float, y: Float }

def is_on_x_axis(p: Point): Bool =
  p.y == 0.0
```

Records can be nested, and projection can be chained.

```
type Point = { x: Float, y: Float }
type Circle = { center: Point, radius: Float }

def is_unit_circle_at_origin(c: Circle): Bool =
  c.center.x == 0.0 && c.center.y == 0.0 && c.radius == 1.0
```

Local Bindings

Local bindings allow you to assign a name to an expression, which can then be used in a subsequent expression. This is accomplished using the `let` keyword

and is invaluable for improving the clarity, structure, and efficiency of your specifications.

A local binding takes the form `let name = expression1; expression2`. This binds the result of `expression1` to `name`. The binding `name` is only visible within `expression2`, which is the scope of the binding.

The primary purposes of `let` bindings are:

1. **Readability**: Breaking down a complex expression into smaller, named parts makes the logic easier to follow.
2. **Re-use**: If a sub-expression is used multiple times, binding it to a name avoids repetition and potential re-computation.

Consider the following formula for calculating the area of a triangle's circumcircle from its side lengths `a`, `b`, and `c`:

```
def circumcircle(a: Float, b: Float, c: Float): Float =
  (a * b * c) / sqrt((a + b +c) * (b + c - a) * (c + a - b) * (a + b - c))
```

Using `let` bindings makes the logic much clearer:

```
def circumcircle(a: Float, b: Float, c: Float): Float =
  let pi = 3.14;
  let s = (a + b + c) / 2.0;
  let area = sqrt(s * (s - a) * (s - b) * (s - c));
  let circumradius = (a * b * c) / (4.0 * area);
  circumradius * circumradius * pi
```

The type of the bound variable (`s`, `area`, `circumradius`) is automatically inferred from the expression it is assigned. You can also chain multiple `let` bindings to build up a computation step-by-step.

Systems

Signals, definitions and specifications are organized into *systems*. A system file should start with a system declaration:

```
system Engine
```

The name of the system should match the file name.

Type declarations

A new type is declared with the `type` keyword. To define a new record type `Point`:

```
type Point = { x: Float, y: Float }
```

We can then use `Point` as a type anywhere else in the file.

Signals

The time varying values of the system are called *signals*. They are declared with the `signal` keyword. E.g.:

```
signal x: Float
signal y: Float
signal speed: Float
signal rain_sensor: Bool
signal wipers_on: Bool
```

The definitions and specifications of a system can freely refer to the system's signals.

A signal can be of any type, i.e. a combination of primitive types and records.

System Parameters

Variables of a system which are constant over time are called *system parameters*. They are declared with the `param` keyword. E.g.:

```
param temp_threshold: Float
param max_errors: Int
```

The definitions and specifications of a system can freely refer to the system's parameters. Note that system parameters must be provided upfront before monitoring can begin. For exemplification, system parameters are optional. That is, they can be provided, in which case the example must conform to them, or otherwise the exemplification process will try to find values that work.

Definitions

A definition is declared with the `def` keyword:

```
def foo: Int = 42
```

A definition can depend on parameters:

```
def foo(x: Float) = x + 42
```

One can also specify the return type of a definition:

```
def foo(x: Float): Float = x + 42
```

The type annotations on parameters and the return type are both optional, if they are not provided they are inferred. It is recommended to always specify these types as a form of documentation.

The parameters of a definition can also be be record types, for instance:

```
type S = { x: Float, y: Float }

def foo(s: S) = eventually [0,1] s.x > s.y
```

Definitions can be used in other definitions, e.g.:

```
type S = { x: Float, y: Float }

def more_x_than_y(s: S) = s.x > s.y

def foo(s: S) = eventually [0,1] more_x_than_y(s)
```

Definitions can be specified in any order, as long as this doesn't create any circular dependencies.

Definitions can freely use any of the signals of the system, without having to declare them as parameters.

Specifications

A `spec` says something that should be true of the system. They can use all the `signal`s and `def`s of the system. They are declared using the `spec` keyword. They are much like `def`s except:

- The return type is always `Bool` (and doesn't need to be specified)
- They cannot have parameters.

Example:

```
signal speed: Float

def above_min = 0 <= speed

def below_max = speed <= 100

spec valid_speed =
  always (above_min && below_max)
```

Consistency Checking Specs are checked for consistency. A warning is produced if specs may be unsatisfiable:

```
signal x: Float

spec main = always (x > 0 && x < 0)
```

This means that the specification is problematic, because it is impossible that any system satisfies this specification.

Inconsistencies between specs are also reported to the user:

```
signal x: Float

spec always_positive = always (x > 0)
spec always_negative = always (x < 0)
```

In this case each of the specs are satisfiable on their own, but taken together they cannot be satisfied by any system.

Redundancy Checking If one spec is redundant, because implied by other specs of the system, this is also detected:

```
signal x: Float

spec positive_becomes_negative = always (x > 0 => eventually x < 0)

spec sometimes_positive = eventually x > 0

spec sometimes_negative = eventually x < 0
```

In this case we warn the user that the spec `sometimes_negative` is redundant, because this property is already implied by the combination of `positive_becomes_negative` and `sometimes_positive`. Indeed `sometimes_positive` implies that there is some point in time where `x > 0`, and using `positivie_becomes_negative` we conclude that therefore there must be some point in time after than when `x < 0`.

Modules

Lilo language supports modules. A module starts with a module declaration, and contains definitions (much like a system):

```
module Util

def add(x: Float, y: Float) = x + y

pub def calc(x: Float) = add(x, x)
```

A module can only contain `def`s and `type`s.

Those definitions which should be accessible from other modules should be parked as `pub`, which means "public".

To use a module, one needs to import it, e.g. `import Util`. The `pub`lic definitions from Util are then available to be used, with qualified names, e.g.:

```
import Util

def foo(x: Float) = Util.calc(x) + 42
```

One can import a module qualified with an alias, for example:

```
import Util as U

def foo(x: Float) = U.calc(x) + 42
```

To use symbols without a qualifier, use the `use` keyword:

```
import Util use { calc }

def foo(x: Float) = calc(x) + 42
```