Logical Student Language

Version 8.15

Andrey Piterkin Luke Jianu

February 15, 2025

1 Purpose

The Logical Student Language is a simple extension to ISL+ that allows students to practice writing formal specifications.

In HtDP, students write informal specifications for their programs. Students describe new forms of data by writing data definitions, and describe the behavior of functions with signatures and purpose statements, as comments.

In contrast, a formal specification is written as code, and therefore is formal in the sense that it's unambiguous. LSL supports writing data definitions, signatures, and purpose statements in code. We call these bits of code *contracts*. In addition, students can determine whether their code satisfies these formal specifications (contracts) using a testing technique called *property-based testing*.

This project is a re-implementation of LSL. The goal of the rewrite is to simplify the implementation, as well as improving the static checks.

2 Concepts

There are two core concepts in LSL: contracts and property-based testing (PBT).

A contract is a claim about a value or its behavior. A contract is a piece of data, in the sense that users can create new kinds of contracts. A contract could be simply represented by a Racket predicate.

PBT is a testing technique where a user writes *properties* about how their programs should behave; the property is then checked over a large number of randomly generated inputs. For example, a user might want to express the commutative property of arithmetic addition. They could do so by writing a function `addition-property` that accepts two numbers `x`, `y` and checks that `(+ x y)` is the same as `(+ y x)`. The PBT library could then be invoked on `addition-property`, checking the property holds for many numbers `x`, `y`. PBT is typically represented by a special function that inititiates PBT on a particular property with some configuration (number of inputs to generate, how complex the inputs should be).

3 Examples

Data Definitions as Code

```
(define-struct leaf [value])
(define-struct node [left right])
(define-contract (Leaf X) (Struct leaf [X]))
(define-contract (Node X Y) (Struct node [X Y]))
(define-contract (Tree X) (OneOf (Leaf X) (Node (Tree X) (Tree X))))
```

Signatures as Code

```
(: height (-> IntTree Natural))
(define (height t)
  (cond [(leaf? t) 0]
        [(node? t) (add1 (max (height (node-
left t)) (height (node-right t))))]))
```

Purpose Statements as Code

```
(: subsets (Function (arguments [n Natural])
                     (result (AllOf (List (List Natural))
                                    (lambda (l) (= (length l) (expt 2 n)))))))
(define (subsets n) ...)
```

Property-based Testing

```
(: height-rec-prop (-> IntTree True))
(define (height-rec-prop t)
  (and (equal? (add1 (height t))
               (height (make-node t (make-leaf 0))))
       (equal? (add1 (height t))
               (height (make-node (make-leaf 0) t)))))

(check-contract height-rec-prop)
```

4 Grammars and Signatures

LSL defines the following core forms on top of those provided by ISL+.

```
;; Binds a contract to the given id
(define-contract <id> <contract>)

<contract> := (Immediate <immediate-clause> ...)
            | (Function <args> <result> <raises>)
| (List <contract>)
            | (Tuple <contract> ...+) | (OneOf <contract> ...+)
            | (AllOf <contract> ...+) | (Struct <id> [<contract> ...])
            | (All (<id> ...) <contract>) | (Exists (<id> ...) <contract>)
            | <expr> ;; must evaluate to a predi-
cate   <immediate-clause> := (check <expr>) ;; the
expr in check position must ;; evaluate to a predicate
| (generate <expr>)         ;; must evaluate to a function
                                                    ;; (-
> Natural X) where X satisfies
                                                    ;; the contract
                    | (feature <string> <expr>) ;; expr must eval-
uate ;; to a function accepting ;; X where X satisfies the con-
tract | (shrink <expr>)           ;; expr must evaluate to
                                                    ;; a function (-
> Natural X X)

<args> := (arguments [id contract] ...) ;; cyclic dependencies are not allowed
<result> := <contract> ;; all id values bound in <args> are available in <result>
<raises> := <exn-id>

;; Annotates an id with a contract
(: <id> <contract>)

;; Attempts to break the contract placed on <id>
(check-contract <id> <natural> <size-expr>)
;; size-expr should evaluate to a fuel number, or a
;; function from the iteration number to fuel

;; Generates a value that satisfies the given contract using the supplied fuel.
(contract-generate <contract> <maybe-fuel>)
<maybe-fuel> := | | <natural>
```

These core forms provide the capability to define and compose contracts, to annotate values with contracts, and to check contracts with property-based testing.

In addition to core forms, LSL provides atomic contracts and various derived forms, which reduce the burden on students to implement these contracts themselves.

```
 <contract> := ...
             | Any | True
             | False | Boolean
             | Natural | Integer
             | Real | String
             | Symbol | (Constant <expr>)  ;; equal? <expr>
             | (Maybe <contract>) ;; #f or <contract>
 | (-> <contract> ... <contract>)
```

5 Implementation Timeline

To implement the re-write of LSL, we plan on working bottom-up:

1. Define specification for the language

2. Decide on the contract interface

3. Implement the contract runtime for core forms

4. Implement derived forms

5. Provide ordinary functionality from ISL+

6. Implement syntax transformations and static checking.

Given that LSL-v1 is already implemented, we will be extracting the core features, interfaces, and specification from the existing project. Once we are confident we have understood the existing design, we will re-implement the language, first at the runtime level, then using syntax-spec.