Linear Promises: Towards Safer Concurrent Programming – Artifact

Ohad Rau ohad@gatech.edu
Caleb Voss cvoss@gatech.edu
Vivek Sarkar vsarkar@gatech.edu

April 2021

1 Abstract

2 Getting Started

2.1 Installing the Compiler

The easiest way to get started with the compiler is using the provided Docker image. If you don't yet have Docker installed, you can follow the official instructions to install it. Once Docker is installed, you can run the following commands to download and then run the image:

```
$ docker pull ohadrau/linear-promises:latest
$ docker run -it ohadrau/linear-promises
```

After running this command, you should be loaded into a shell where you can interact with the compiler (`linear-promises.compiler --help` to see options for the compiler).

2.2 Running an Example Program

There are several example programs already available in the `examples/` and `stackoverflow/` directories in the Docker image. Whenever a program is compiled with `linear-promises.compiler`, it will generate a Java source file. This file utilizes the language runtime and must be compiled with the runtime to produce runnable code. The simplest way to do this is simply placing the Java code in the same directory as the runtime library and compiling it, though you can also precompile the library and link against it.

To compile and run `examples/long.txt`:

```
# -o sets the output file path
$ linear-promises.compiler -o runtime/src/Long.java examples/long.txt
$ cd runtime/src
# Compile the Java source alongside the runtime
```

```
$ javac Long.java
# Run the compiled Java program
$ java Long
```

3 Language Guide

```
-- Types
Unit -- this is the nothing type, like void in some languages
Int
Bool
Promise(Type) -- read-handle for a promise of a value
Promise*(Type) -- write-handle for a promise of a value

-- Values
() : Unit
1 : Int
-5 : Int
true : Bool
false : Bool

-- Functions
func <name>(<params>): <return> begin
  <code>
end

-- e.g.

func chooseFirstOne(a: Int, b: Int): Int begin
  a -- Functions automatically return the last value
end

-- Main function
func main(): Unit begin
  -- ...
end

-- Calling functions
<name>(<args>)

-- e.g.
f(1, true, p)

-- Asynchronous execution
async <expr>
```

```
-- e.g.
async f(1, true, p)

-- Let bindings
let <name>: <type> = <value> in
<code>

-- e.g.
let p: Bool = true in
p

-- Sequences
<code>; -- Sequence code with semi-colon
<code>  -- No semi-colon after last expression

-- Promises
-- Create a promise of an int (pRead is the read-handle, pWrite is the write-han
promise pRead, pWrite: Int in
pWrite <- 6; -- Write value to promise (only works with owned promises)
-- After writing to pWrite, the variable no longer exists in scope
pRead <~ 7; -- We can still force a write using the unsafe write syntax (<~) on
?pRead -- Await a value from promise's read-handle

-- Conditionals
if <boolean expr> then
  <expr>
else
  <expr>
end

-- Loops
while <boolean expr> begin
  <expr>
end

for <name> = <initial value> to <final value> begin
  <expr>
end

-- User-defined types

-- Records
record <name> begin
  <field>: <type>;
  <field>: <type>...
```

```
end

-- E.g.
record Vector2d begin
  x: Int;
  y: Int
end

-- Construct a record
Vector2d { x=5, y=10 }

-- Access fields directly
func manhattanDistance(v: Vector2d): Int begin
  v.x + v.y
end

-- Or with pattern matching
func swap(v: Vector2d): Vector2d begin
  match v begin
    { x=a, y=b } -> Vector2d { x=b, y=a }
  end
end

-- Unions
union <name> begin
  <case>[<param types>];
  <case>[<param types>]...
end

-- E.g.
union List begin
  Nil[];
  Cons[Int, List]
end

-- Construct a union
Cons[1, Cons[2, Cons[3, Nil[]]]]

-- Deconstruct using pattern matching
func length(list: List): Int begin
  match list begin
    Nil[] -> 0
    Cons[x, rest] -> 1 + length(rest)
  end
end
```

4 Code Breakdown

The compiler is written in OCaml with minimal dependencies (only the OCaml standard distribution and the Menhir parser generator). The code for the compiler is all contained within `src/` in the Docker image. This directory includes the following files:

- `compiler.ml`: the driver for the compiler; parses command-line arguments and runs the compilation pipeline.
- `contexts.ml`: implements the data structures used to provide typing contexts (specifically, typing environments & sets of variables).
- `ident.ml`: utilities for working with identifiers. This provides a way to generate unique identifiers, as well as a preprocessor to give each distinct variable a unique name (e.g. `let x = 0 in let x = 1 in x` becomes `let x0 = 0 in let x1 = 1 in x1`).
- `javaCG.ml`: implements Java code-generation by performing a translation from our language's AST to source-level Java. This code also inserts several built-in methods into the static scope of the program.
- `lang.ml`: defines the AST for the language, as well as string formatters for the AST.
- `lexer.mll`: implements the tokenization for the language syntax, utilizing the OCamllex lexer generator.
- `parser.mly`: implements parsing for the language syntax, utilizing the Menhir parser generator.
- `typecheck.ml`: the type-checking code for the language. This is the bulk of the compiler and is described in pseudo-code in the paper (section 4.2).