While$^+$ Denotational Semantics Interpreter

Alberto Schiabel

University of Padova
alberto.schiabel@gmail.com

March 28, 2021

Abstract

In this report, we describe the implementation of an interpreter for the denotational semantics of the **While**$^+$ language. We refer to the Kleene-Knaster-Tarski fixpoint iteration theorem for evaluating *while* loop statements. Non-termination is modelled with a never-ending loop, rather than giving rise to a runtime error or exception. We used the functional language Haskell for the implementation of this interpreter. The Github repository for the project is publicly available at https://github.com/jkomyno/whileplus.

I. Introduction

The purpose of this project is writing a program I such that:

- I takes as input any $S \in$ **While**$^+$and some representation of $s \in$ State.
- **While**$^+$ is **While** plus some syntactic sugar for arithmetic and boolean expressions and statements, including *for* and *repeat until* loops.
- It must always happen that $I(S, s) = \mathcal{S}_{ds}[\![S]\!]s$. Thus, $\mathcal{S}_{ds}[\![S]\!]s = undef$ if and only if the execution of $I(S, s)$ does not terminate, i.e. $I(S, s)$ keeps running forever without any runtime error or exception.
- I relies on Kleene-Knaster-Tarski fixpoint iteration sequence for evaluating the while statements. This means that if F : (**State** $\hookrightarrow$ **State**) $\to$ (**State** $\hookrightarrow$ **State**) is the functional induced by the loop program *while b do S*, then $I(\textit{while b do S}, s)$ must be implemented as: $\sqcup_{n \geq 0} F^n(\perp)s$. This means that $I(\textit{while b do S}, s)$ should look for the least $k \geq 0$ such that $F^k(\perp)s = s'$ for some s' such that $I(\textit{while b do S}, s)$ outputs s', whereas if for all $n \geq 0$ $F^n(\perp)s = undef$, then $I(\textit{while b do S}, s)$ does not terminate.

We choose Haskell as the programming language for implementing the **While**$^+$interpreter. Some of the most important reasons why this typed functional programming language was chosen are:

1. its syntax is particularly handy for describing function compositions with implicit parameters, keeping the code very close to the original mathematical notation[1];
2. the pattern matching capabilities of the language are especially useful for describing the different branches of an interpreter's evaluation while keeping the code tidy and reasonably readable;
3. the author has already had some experience implementing language parsers with the same language for another University course;
4. it was one of the languages explicitly suggested by the professor.

We also choose Stack for developing and building the Haskell program in an isolated environment. The Stack commands are abstracted by a Makefile.

II. Language Syntax

In this section we first specify the syntax of the **While** language, and then we integrate with **While**$^+$ syntax constructs. The syntactic notation we use is based on BNF.

i. Syntactic Categories

We first list the syntactic categories and give a meta-variable that will be used to range over *constructs* of each category:

- n will range over numerals, **Num**
- x will range over variables, **Var**
- a will range over arithmetic expressions, **Aexp**
- b will range over boolean expressions, **Bexp**
- S will range over statements, **Stm**

The meta-variables can be primed or subscripted. For example, n, n', n_1, n_2 all stand for numerals.

ii. **While** syntax

The syntax of the other **While** constructs is:

$$\begin{array}{rcll} a & ::= & n & \textit{numeral} \\ & | & x & \textit{variable} \\ & | & a_1 + b_2 & \textit{sum} \\ & | & a_1 - b_2 & \textit{subtraction} \\ & | & a_1 * b_2 & \textit{multiplication} \end{array}$$

$$\begin{array}{rcll} b & ::= & \texttt{true} & \textit{literal tt} \\ & | & \texttt{false} & \textit{literal ff} \\ & | & a_1 = a_2 & \textit{equal} \\ & | & a_1 \leq a_2 & \textit{less than or equal} \\ & | & \neg b & \textit{logic negation} \\ & | & b_1 \wedge b_2 & \textit{logic conjunction} \end{array}$$

$$\begin{array}{rcll} S & ::= & x \texttt{ := } a & \textit{assignment} \\ & | & \texttt{skip} & \textit{no-op} \\ & | & S_1 \texttt{ ; } S_2 & \textit{composition} \\ & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \textit{conditional} \\ & | & \texttt{while } b \texttt{ do } S & \textit{while} \end{array}$$

iii. **While$^+$** syntax

While$^+$ is built upon the previous **While**'s definition, but it defines the following additional syntactic sugar:

$$\begin{array}{rcll} b' & ::= & a_1 \neq a_2 & \textit{not equal} \\ & | & a_1 < a_2 & \textit{less than} \\ & | & a_1 > a_2 & \textit{greater than} \\ & | & a_1 \geq a_2 & \textit{greather than or equal} \\ & | & b_1 \vee b_2 & \textit{logic disjunction} \end{array}$$

$$\begin{array}{rcll} S' & ::= & x \mathrel{+}= a & \\ & | & x \mathrel{-}= a & \\ & | & x \mathrel{*}= a & \\ & | & \texttt{repeat } S \texttt{ until } b & \textit{repeat until} \\ & | & \texttt{for } x \texttt{ := } a_1 \texttt{ to } a_2 \texttt{ do } S & \textit{for} \end{array}$$

Moreover, we decided to enrich **While$^+$** with two additional statements:

$$\begin{array}{rcll} S' & ::= & x_1, x_2 \texttt{ := } a_1, a_2 & \textit{pair assignment} \\ & | & \texttt{repeat' } S \texttt{ until } b & \textit{repeat until (native)} \end{array}$$

The **pair assignment** is inspired by the Python language. Notice that **repeat'** (with the prime symbol) is to be implemented natively, whereas **repeat** will be converted in a special form and will rely on **while**'s evaluation. Different constructs in a **While$^+$** program are separated by the ";" (semicolon) symbol.

III. Math Preliminaries

Before proceeding further with the discussion about the implementation details, we will state some Theorems and Lemmas that provide a theoretical justification of why our interpreter works. Proofs are omitted, as they can be found in [4].
First, we must introduce some auxiliary semantic functions.

State We represent a state as a function from variables to values: $State : Var \to \mathbb{Z}$.

Cond The conditional clause is a higher-order function that, when supplied with an argument, will select either the second or the third parameter and that supply that parameter with the same argument. It accepts a boolean predicate function, two partial functions on $State$ g_1 and g_2, and it returns a partial function on $State$.

$$cond(p, g_1, g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \texttt{tt} \\ g_2\ s & \text{if } p\ s = \texttt{ff} \end{cases}$$

Definition 1 (Partially Ordered Set). A poset $(D, \sqsubseteq)$ is a set D with an orderering $\sqsubseteq$ that is:

- reflexive: $d \sqsubseteq d$;
- transitive: $d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_3 \Rightarrow d_1 \sqsubseteq d_3$;
- anti-symmetric: $d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_1 \Rightarrow d_1 = d_2$

d is a least element of $(D, \sqsubseteq)$ if $d \sqsubseteq d'\ \forall\ d' \in D$. Such d, if it exists, is unique.

Definition 2 (Chain). A subset Y of D is called a chain if, for any 2 elements d_1 and d_2 in Y, one of these two properties hold:

- $d_1 \sqsubseteq d_2$ or
- $d_2 \sqsubseteq d_1$

Definition 3 (Chain Complete Partially Ordered Set). $(D, \sqsubseteq)$ is a ccpo if every chain of D has a least upper bound.

Definition 4 (Monotonic Functions). Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ccpos and consider a function $f : D \to$

D'. f is monotone if, whenever $d_1 \sqsubseteq d_2$, it holds that $f\ d_1 \sqsubseteq' f\ d_2$. In other words, a monotonic function is an order-preserving function.

Definition 5 (Continuous Functions)**.** Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be ccpos and consider a function $f : D \to D'$. f is continuous if:

- f is monotone, and
- $\sqcup'\{f\ d \mid d \in Y\} = f(\sqcup\ Y)$ for all non-empty chains $Y \in D$

Lemma 4.13. $(\textbf{State} \hookrightarrow \textbf{State}, \sqsubseteq)$ is a partially ordered set. The partial function $\perp : \textbf{State} \hookrightarrow \textbf{State}$ defined by

$$\perp s = \text{undef } \forall s$$

is the least element of **State** $\hookrightarrow$ **State**.

Lemma 4.25. $(\textbf{State} \hookrightarrow \textbf{State}, \sqsubseteq)$ is a chain-complete ordered set. The least upper bound (lub) $\sqcup Y$ of a chain Y is given by:

$$(\sqcup Y)\ s = \begin{cases} g\ s & \text{if } g\ s \neq \text{undef for some } g \in Y \\ \text{undef} & \text{if } g\ s = \text{undef } \forall g \in Y \end{cases}$$

Lemma 4.35. If $f : D \to D'$ and $f' : D' \to D''$ are continuous functions, then $f' \circ f$ is a continuous function.

Theorem 4.37. Let $f : D \to D$ be a continuous function on the ccpo $(D, \sqsubseteq)$ with least element $\perp$. Then

$$\text{FIX } f = \sqcup\{f^n \perp \mid n \geq 0\}$$

defines an element of D, and this element is the least fixed point of f. We have used the fact that

$$f^0 = \text{id, and}$$
$$f^{n+1} = f \circ f^n \text{ for } n \geq 0$$

Lemma 4.43. Let $g_0 : \textbf{State} \hookrightarrow \textbf{State}$, $f : \textbf{State} \to T$ and define $F\ g = \text{cond}(p, g, g_0)$. Then F is continuous.

Lemma 4.44. Let $g_0 : \textbf{State} \hookrightarrow \textbf{State}$, $f : \textbf{State} \to T$ and define $F\ g = \text{cond}(p, g_0, g)$. Then F is continuous.

Lemma 4.45. Let $g_0 : \textbf{State} \hookrightarrow \textbf{State}$ and define $F\ g = g \circ g_0$. Then F is continuous.

Proposition 4.47. The semantics equations in Section IV define a total function $\mathcal{S}_{ds} : Stm \to (\textbf{State} \hookrightarrow \textbf{State})$.

IV. Denotational Semantics

In denotational semantics we are interested is in the effect of executing a program, where the effect is an association between initial and final states. Thus, we define a *semantic function* for each *syntactic category*, which maps each syntactic construct to a function that describes the execution of the construct.

The execution of a statement S possibly changes the state. We can then define its meaning as a partial function on states:

$$\mathcal{S}_{ds} : Stm \to (State \hookrightarrow State)$$

We can thus define the denotational semantics for the **While**$^+$ language statements as follows:

$$\mathcal{S}_{ds}[\![x := a]\!]s = s[x \mapsto \mathcal{A}_{ds}[\![a]\!]s]$$
$$\mathcal{S}_{ds}[\![skip]\!] = id$$
$$\mathcal{S}_{ds}[\![S_1; S_2]\!] = \mathcal{S}_{ds}[\![S_2]\!] \circ \mathcal{S}_{ds}[\![S_1]\!]$$
$$\mathcal{S}_{ds}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!] = cond(\mathcal{B}[\![b]\!], \mathcal{S}_{ds}[\![S_1]\!], \mathcal{S}_{ds}[\![S_2]\!])$$
$$\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] = \text{FIX } F$$
$$\text{where } F\ g = cond(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], id)$$
$$\mathcal{S}_{ds}[\![\text{repeat' } S \text{ until } b]\!] = \text{FIX } F'$$
$$\text{where } F'\ g = cond(\mathcal{B}[\![b]\!], id, g) \circ \mathcal{S}_{ds}[\![S]\!]$$

The pair assignment statement $\mathcal{S}_{ds}[\![x_1, x_2 := a_1, a_2]\!]s$ is defined as follows.
Let n_1 be $\mathcal{A}_{ds}[\![a_1]\!]s$, and let n_2 be $\mathcal{A}_{ds}[\![a_2]\!]s$. Then, $\mathcal{S}_{ds}[\![x_1, x_2 := a_1, a_2]\!]s = s[x_1 \mapsto n_1, x_2 \mapsto n_2]$.
The semantic definitions the syntactic sugar terms are given in Section VII.

V. Interpreter Flow

The **While**$^+$ interpreter flow is shown in Figure 1.

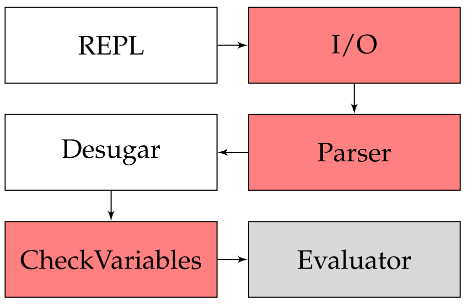

Figure 1: *While*$^+$ *interpreter flow. The action blocks drawn in red might be susceptible to errors, whereas the action block drawn in gray might never terminate.*

Initially, the REPL (**R**ead-**E**val-**P**rint-**L**oop) of the language is started. From there, the user can either import a file. If the user attempts to read a file that doesn't exist or that isn't readable, an IO error is

thrown and the application exits. If, however, the file is readable, its content is fed to the Parser.
The source of the **While**$^+$ program is then tokenized and transformed in AST form. If the source code doesn't follow the language syntax rules, a Parse error is thrown. The AST is then desugared. The process of removing syntactic sugar by rewriting a construct into another construct of the language is known as "desugaring". After a construct has been desugared, it won't appear in any expression.
The next step is checking that the variables referenced in the desugared AST are compatible with the order of that definition. This is a syntactical process, i.e. no runtime evaluation is performed for this check. In other words, the interpreter guarantees that the evaluation will only start when it's sure that every variable is bound to a value. If this process fails, the UnboundVariable error is thrown.

Finally, the desugared AST is evaluated one statement at a time. The computation state s is initially empty. Each statement execution might update the state or leave it as it is. If the interpreter enters in the Evaluator phase, no error will occur. In particular, there will be no StackOverflow, OutOfMemory, or UnboundVariable errors. It's possible, however, that the intepreter never terminates. This will only happen if the boolean guards of the `while` and `repeat until` loops are defined in such a way that those loop won't ever break.

VI. Language Parser

```
-- src/Parser.hs
statements :: Parser [Statement]
statements = statement `sepEndBy1` semi

statement :: Parser Statement
statement = composition
        <|> skip
        <|> conditional
        <|> while
        <|> repeat'
        <|> repeat
        <|> for
        <|> try assignment
        <|> try pairAssignment
        <|> opAssignment
```

Listing 1: *Detail of the **While**$^+$ statement parser implemented via a sequence of alternative subparsers.*

i. AST

Our Abstract Syntax Tree (AST) has the following core types of nodes, defined using recursive data constructors.

- `Statement` is the data-type corresponding to statements S;
- `AExpr` is the data-type corresponding to arithmetic expressions a. It embeds the arithmetic binary operations `ArithBinOp`, i.e. $a_1 + a_2$, $a_1 - a_2$, $a_1 * a_2$;
- `BExpr` is the data-type corresponding to boolean expressions b. It embeds boolean unary operators `BoolUnOp` ($\neg b$), boolean binary operators `BoolBinOp` ($b_1 \wedge b_2$, $b_1 \vee b_2$), arithmetic binary relations with boolean results `ArithRelation` ($a_1 = a_2$, $a_1 \neq a_2$, $a_1 < a_2$, $a_1 \leq a_2$, $a_1 > a_2$, $a_1 \geq a_2$).

For representing numerals, we use Haskell's native type `Integer`, which is an arbitrary precision integer that spans over the $\mathbb{Z}$ set.

ii. Parser Combinator

Rather than relying on a parser generator like Bison, Yacc or JavaCC, we preferred using a monadic parser combinator approach via the popular Parsec Haskell library, which is based on the work of Daan Leijen[3]. We first defined language token style, which tells Parsec:

1. how comments are delimited: we used the familiar C comment tokens, i.e. // for line comments and /* */ for block comments;

2. the valid characters for defining identifiers:

3. the reserved keywords and operators.

We then defined a number of parsers for every **While**$^+$ construct. The precedence rules for the boolean arithmetic operators are the usual ones. Moreover, every operator is infix and left-associative, with the exception the negation ($\neg b$) which is prefix and not associative.
Finally, we defined a top-level statement parser that traverses a string containing a **While**$^+$ program and tries to create an AST from it, using the core types introduced previously.

The parsing process proceeds via alternatives. The sequence of alternatives is defined via the <|> operator, as shown in Listing 1. The first parser combinator is executed, and if it doesn't work it falls back to the

second one, and so on. If no parser in the list of alternatives is able to create an AST from the given string, a Parser error is thrown.
For simplicity, the composition statement is not limited to just 2 statements, but it can contain a list of any number of statement greater than 0. If two or more statements are explicitly wrapped in parenthesis, they are automatically wrapped into a composition statement.

```
-- src/Parser.hs
while :: Parser Statement
while = do
  reserved "while"
  b <- bExpr
  reserved "do"
  s <- statement
  return $ While b s
```

Listing 2: *Detail of the **While**$^+$ while loop statement parser.*

VII. Desugaring

We briefly summarise the how some of the additional **While**$^+$ constructs are desugared before being evaluated.

i. Boolean Expressions

$$\begin{array}{lcr} (b \vee b') & \Rightarrow & \neg(\neg b \wedge \neg b') \\ (a \neq a') & \Rightarrow & \neg(a = a') \\ (a \geq a') & \Rightarrow & (a' \leq a) \\ (a > a') & \Rightarrow & \neg(a \leq a') \\ (a < a') & \Rightarrow & \neg(a' \leq a) \end{array}$$

ii. Statements

$$\begin{aligned} &\mathcal{S}_{ds}[\![x \mathrel{+}= a]\!]s = \mathcal{S}_{ds}[\![x := x + a]\!]s \\ &\mathcal{S}_{ds}[\![x \mathrel{-}= a]\!]s = \mathcal{S}_{ds}[\![x := x - a]\!]s \\ &\mathcal{S}_{ds}[\![x \mathrel{*}= a]\!]s = \mathcal{S}_{ds}[\![x := x * a]\!]s \\ &\mathcal{S}_{ds}[\![\text{repeat } S \text{ until } b]\!] = \\ &\qquad \mathcal{S}_{ds}[\![S; \text{while } \neg b \text{ do } S]\!] \\ &\mathcal{S}_{ds}[\![\text{for } x := a_1 \text{ to } a_2 \text{ do } S]\!] = \\ &\qquad \mathcal{S}_{ds}[\![x := a_1; \text{while } x < a_2 \text{ do } (S; x := x + 1)]\!] \end{aligned}$$

VIII. Evaluation

i. Considerations about State

```
-- src/State/State.hs
import AbstractState (Stateable (..))
import qualified Data.Map as Map

-- State assigns each variable to a
-- numeral
type State = S String Integer

-- State is defined as a key-value map
instance Stateable k v where
  data S k v = S (Map.Map k v)
  -- ...
```

Listing 3: *Detail of the **While**$^+$ state definition.*

The state of the computation s is implemented as a map that associates a variable name to an integer number. Arithmetical expressions are in fact fully evaluated before being assigned to a variable. We want to limit errors as much as possible, so proper state initialization is crucial. We mainly have two possibilities:

1. gathering all identifiers referenced in the AST an initialize them in the state with a default value, for instance 0;

2. leave the state empty at the beginning, ensuring that every variable declaration won't generate an error at execution time.

We preferred the second option, as it is cleaner and more intuitive.

ii. Evaluation

The top-level `eval` function takes a desugared AST and runs the `evalStmt` function for every statement in top-down order. Given a statement S and an initial state s, evalStmt evaluates the statement S and returns the new state s'. In case of a loop, it's possible that the function never terminates.
The `Assignment` first evaluates the arithmetic expression with the given state, then writes the numeral variable in the state map and returns the updated state. If the variable already existed, it is overridden. `Skip` is implemented via the identity function. The `Composition` construct is implemented very similarly to the top-level program: each statement in the composition is executed sequentially and the state updates are propagated top-down. The `Conditional` first evaluates the boolean value of its guard `b`, and then recursively calls `evalStmt` on either the first or the second branch.

```haskell
-- src/Eval/Evaluator.hs
-- Evaluate the entire desugared program
-- and return the final state
eval :: DesugaredProgram -> State -> State
eval (DesugaredProgram xs) state = foldl'
  (flip evalStmt) state xs

-- Evaluate the given statement and return
-- the updated state
evalStmt :: Statement -> State -> State
evalStmt (Assignment x a) =
  evalAssignment x a
evalStmt Skip = id
evalStmt (Conditional b t f) =
  evalConditional b t f
evalStmt (While b stmt) =
  evalWhile b stmt
evalStmt (Repeat' stmt b) =
  evalRepeat' stmt b
evalStmt (Composition ss) =
  evalComposition ss
```

Listing 4: *Detail of* `eval` *and* `evalStmt` *function declarations.*

The evaluation of the `While` and `Repeat'` loops deserve special attention. Recall the semantic definition of `While` and `Repeat'`:

$$\mathcal{S}_{ds}[\![\text{while } b \text{ do } S]\!] = \text{FIX } F$$
$$\text{where } F\ g = cond(\mathcal{B}[\![b]\!], g \circ \mathcal{S}_{ds}[\![S]\!], id)$$

$$\mathcal{S}_{ds}[\![\text{repeat}' \ S \text{ until } b]\!] = \text{FIX } F'$$
$$\text{where } F'\ g = cond(\mathcal{B}[\![b]\!], id, g) \circ \mathcal{S}_{ds}[\![S]\!]$$

The Haskell implementation of the evaluation of the `While` and `Repeat'` constructs is shown in Listing 5 and 6. The implementation of `fix`, `cond` and `id'` are discussed in detail in Subsection iii.

```haskell
-- src/Eval/Evaluator.hs
evalWhile :: BExpr -> Statement
          -> State -> State
evalWhile b stmt = let
  f g = cond (evalBExpr b,
              g . evalStmt stmt,
              id')
  in fix f
```

Listing 5: *Detail of* `evalWhile` *function declaration.*

```haskell
-- src/Eval/Evaluator.hs
evalRepeat' :: Statement -> BExpr
            -> State -> State
evalRepeat' stmt b = let
  f' g = cond (evalBExpr b, id', g) .
          evalStmt stmt
  in fix f'
```

Listing 6: *Detail of* `evalRepeat'` *function declaration.*

iii. Fixpoint Iteration

In order to decouple the fixpoint operation from a concrete state implementation, we didn't refer to the concrete `State` type directly, but we referred to a generic `state` type.
Functions between states are written as `state -> state`. We used the Maybe monad to model partiality, so a partial function between states is written as `state -> Maybe state`. Maybe is a data type defined in the Haskell standard library[5] as:

```haskell
data Maybe a = Just a | Nothing
```

where `Nothing` denotes undef.
The bottom ($\perp$) element of the ccpo of partial state functions ($\textbf{state} \hookrightarrow \textbf{state}, \sqsubseteq$) has type `bottom :: state -> Maybe state` (Lemma 4.13), and it is defined as a lambda function that always returns `Nothing`. As a matter of convenience, we define the type alias `Partial state` to represent functions of the type ($state \hookrightarrow state$):

```haskell
type Partial state = state -> Maybe state
```

Recall the cond utility function introduced in III:

$$cond(p, g_1, g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \texttt{tt} \\ g_2\ s & \text{if } p\ s = \texttt{ff} \end{cases}$$

p has type `state -> Bool`, g_1 and g_2 have type `Partial state`. For a convenient similarity with the mathematical notation, we wrapped the first three arguments of the `cond` function in a tuple. The trivial `cond` definition is shown in Listing 7.

```haskell
-- src/FixPoint.hs
cond :: (state -> Bool, Partial state, Partial state)
        -> Partial state
cond (p, g, g') state = if p state
    then g state else g' state
```

Listing 7: *Implementation of the* `cond` *utility function.*

Note that the `id` function, which has type signature `state -> state`, cannot be used directly for representing g or g' in the `cond` definition because their type don't match. We defined `id' :: Partial state` to solve this problem while mantaining the semantic meaning of identity:

```
id' = Just
```

Theorem 4.37 defines the notion of power of functions. We implemented it with the function `f'n`, which takes a functional f, a power number $n \geq 0$ and a partial state function g, and returns the nth application of fg. The function is defined in Listing 8.

```
-- src/FixPoint.hs
type F state = Partial state -> Partial state

-- power of functional f (Theorem 4.37)
f'n :: F state             -- functional
      -> Int               -- power of the functional
      -> Partial state     -- input of the functional
      -> Partial state
f'n _ 0 = id
f'n f n = f . f'n f (n - 1)
```

Listing 8: *Detail of the power of functions* `f'n`*.*

Theorem 4.37 also shows how to compute the least fixed point of a continuous function f on a ccpo with least element $\perp$:

$$\text{FIX } f = \sqcup\{f^n \perp \mid n \geq 0\}$$

Assuming that the least upper bound $\sqcup Y$ is calculated by a function called `lub`, the implementation of `fix` is shown in Listing 9.

```
-- src/FixPoint.hs
type Fix state = F state -> state -> state

-- Knaster-Tarski fixed-point theorem
-- application (Theorem 4.37)
fix :: Fix state
fix f = lub [f'n f n bottom | n <- [0..]]
```

Listing 9: *Detail of* `fix` *function declaration.*

The Haskell declaration of `fix` is extremely similar to the original mathematical notation. The expression `[f'n f n bottom | n <- [0..]]` is a list comprehension that, $\forall n \in \mathcal{N}$, computes $f^n\perp$. Using infinite lists in Haskell is a common pattern, and it works because lists are evaluated lazily.

`lub` then receives a potentially infinite list of partial state functions and a state s as arguments, and computes the least upper bound using the definition in Lemma 4.25. That potentially infinite list is a chain because it holds that:

$$f^0\perp \sqsubseteq f^1\perp \sqsubseteq \cdots \sqsubseteq f^n\perp$$

Lemma 4.25 describes how to implement a least upper bound of a chain. The Haskell implementation is shown in Listing 10. `(g:gs)` is the chain Y (`g` is the first element, `gs` is the tail of the list) and `s` is a state. There are some things to note here:

- The chain `(g:gs)` is never empty, as it contains at least the singleton $\{f^0\perp\}$;
- If $f^0\perp s$ = undef, the `lub` function evaluates $f^1\perp$, and so on;
- If $f^k\perp s \neq$ undef for some $k \geq 0$, it returns $f^k\perp s$;
- If $f^n\perp s$ = undef $\forall n \in \mathcal{N}$, the `lub` function will never terminate.

```
-- src/FixPoint.hs
lub :: [Partial state] -> state -> state
lub (g:gs) s = case g s of
  -- look for some g in gs
  Nothing   -> lub gs s

  -- g s is defined
  (Just s') -> s'
```

Listing 10: *Detail of* `lub` *function declaration.*

iv. Tail Recursion and Optimization

A recursive function is said to be *tail recursive* if the final result of the recursive call is the final result of the function itself. We can notice that `lub` is a tail resursive function. Some compilers can leverage this type of functions to perform *tail-call optimization*. This optimization allows the re-use of the same stack frame used by the first recursive function invocation, which effectively keeps the stack memory usage constant and avoids StackOverflow errors.

Haskell supports such optimization out-of-the-box. From [2]:

> In many programming languages, calling a function uses stack space, so a function that is tail recursive can build up a large stack of calls to itself, which wastes memory. Since in a tail call, the containing function is about

to return, its environment can actually be discarded and the recursive call can be entered without creating a new stack frame. This trick is called tail call elimination or tail call optimisation and allows tail-recursive functions to recur indefinitely.

IX. REPL Commands

The **While**$^+$ interpreter exposes a number of features via REPL commands. Each of this commands start with the colon (":") symbol. The user doesn't necessarily have to write the command in long form, as the REPL is able to understand the given command as long as it is a valid prefix of a supported command. As an example, if the user wants to quit the application, the user can write `:q`, `:qu`, or `:quit`.

The REPL is able to persist an execution context in memory. For example, the user can import a **While**$^+$ source file, which is referenced to as `FILE` in the commands description in Subsection i.

i. Available Commands

Here we describe the commands available.

:load `FILE` Use this command to load a **While**$^+$ source code from `FILE` to the program's memory. After importing a **While**$^+$ source code, the user can view the AST of the source, desugar it, check if the variable declarations are valid, and of course interpret the source. The **load** operation is only successful if `FILE` is readable and its content is parsed without errors.

:interpret `[LINE]` Use this command to interpret the given `LINE` of **While**$^+$ code. If no `LINE` is given, it interprets the latest loaded `FILE`. If no `FILE` is loaded, an error is returned. The user can interrupt the interpreter process with the `CTRL + C` keyboard combination.

:ast `[LINE]` Use this command to show the AST of the given `LINE` of **While**$^+$ code. If no `LINE` is given, it shows the AST of the latest loaded `FILE`. If no `FILE` is loaded, an error is returned.

:desugar `[LINE]` Use this command to show the desugared AST of the given `LINE` of **While**$^+$ code. If no `LINE` is given, it shows the desugared AST of the latest loaded `FILE`. If no `FILE` is loaded, an error is returned.

:check `[LINE]` Use this command to check the validity of variable declarations and references in the given `LINE` of **While**$^+$ code. If no `LINE` is given, it checks the variables of the latest loaded `FILE`. If no `FILE` is loaded, an error is returned.

:reset Use this command to reset the state of the computations performed by the `:interpret` command.

:state Use this command to show the content of the state of the computations performed by the `:interpret` command.

:verbose Use this command to toggle verbose mode on or off. If verbose mode is on, the desugared AST and the final state of computation is automatically shown when running the `:interpret` command. By default, verbose mode is off.

:quit Use this command to quit the **While**$^+$ language REPL.

:help Use this command to show the help message.

X. Conclusion

i. Our contributions

Our contributions with this project are multifold: (1) we created a **While**$^+$ interpreter in Haskell that relies on Kleeni-Knaster-Tarski fixpoint iteration sequence for evaluating loop statements; (2) we ensured the absence of runtime exceptions. If the user tries to load a source file that isn't readable, the REPL doesn't crash; instead, it offers the opportunity to retry. The interpreter never causes StackOverflow errors; (3) we embedded such interpreter in a REPL environment that exposes commands to interact with the state of the computations and to observe the AST of the parsed **While**$^+$ code.

ii. What we learned

Completing this project and writing this report helped us realize the challenges of writing an interpreter pipeline for an untyped language. Moreover, it made us interiorize better the theory we studied at the beginning of the Software Verification course, as this interpreter project relies on those theories. As a pleasant side-effect, this experience made us feel more comfortable with the Haskell ecosystem and functional programming in general.

iii. Project sources

The project code and this report are publicly available at https://github.com/jkomyno/whileplus. Please refer to the README file on Github for instructions on how to build and run the **While**$^+$ interpreter REPL.

Appendices

A. Proofs

Proof of Proposition 4.47. The proof is given by structural induction on the statement S. We take for granted the part of the proof given in [4].

The case $\mathcal{S}_{ds}[\![x_1, x_2 := a_1, a_2]\!]$: we know that $\mathcal{A}_{ds}[\![a_1]\!]$ and $\mathcal{A}_{ds}[\![a_2]\!]$ are well-defined. Let n_1 be $\mathcal{A}_{ds}[\![a_1]\!]s$, and let n_2 be $\mathcal{A}_{ds}[\![a_2]\!]s$. Clearly the function that maps a state s into $s[x_1 \mapsto n_1, x_2 \mapsto n_2]$ is well-defined.

The case $\mathcal{S}_{ds}[\![\text{repeat } S \text{ until } b]\!]$: $\mathcal{S}_{ds}[\![S]\!]$ is well defined, by inductive hypothesis. The functions F_1 and F_2, defined as

$$F_1\ g = cond(\mathcal{B}_{ds}[\![b]\!], id, g)$$
$$F_2\ g = g \circ \mathcal{S}_{ds}[\![g]\!],$$

are continuous because of Lemmas 4.43, 4.44, 4.45. Thus, Lemma 4.35 gives us that $Fg = F_2(F_1 g)$ is continuous. For Theorem 4.37, we have that $FIX\ F$ is well defined and therefore $\mathcal{S}_{ds}[\![\text{repeat } S \text{ until } b]\!]$ is well-defined. □

References

[1] HaskellWiki. *Haskell and mathematics — HaskellWiki,* [Online; accessed 16-November-2020]. 2009. URL: `https://wiki.haskell.org/index.php?title=Haskell_and_mathematics&oldid=26212`.

[2] HaskellWiki. *Tail recursion — HaskellWiki,* [Online; accessed 10-November-2020]. 2019. URL: `https://wiki.haskell.org/index.php?title=Tail_recursion&oldid=62916`.

[3] Daan Leijen. "Parsec, a fast combinator parser". In: 2001.

[4] Hanne Nielson. *Semantics with applications : a formal introduction*. Chichester New York: J. Wiley, 1992. ISBN: 978-0471929802.

[5] Wikibooks. *Haskell/Libraries/Maybe — Wikibooks, The Free Textbook Project*. [Online; accessed 10-November-2020]. 2020. URL: `https://en.wikibooks.org/w/index.php?title=Haskell/Libraries/Maybe&oldid=3676215`.