The While language

Cláudio Vasconcelos António Ravara

NOVA-LINCS and Dep. de Informática, FCT.
Universidade NOVA de Lisboa, Portugal

April 13, 2016

Abstract

This article presents a formalisation of a simple imperative programming language. The objective is to study and develop "hands-on" a formal specification of a programming language, namely its syntax, operational semantics and type system. To have an executable version of the language, we implemented in Racket its operational semantics and type system.

1 Introduction

This article consists on the presentation of the While language described in a book by Hanne Riis and Flemming Nielson [7]. We follow the definitions in the book, namely to set up the syntax and operational semantics of the language, and devise a type system, which is thus original.

This is an initial step to understand at the same time: (1) the fundamentals of imperative programming and its features, such as state changes, order of execution and control flow expressions [5]; and (2) how to have an executable version of the formalisation, allow to automaticaly build derivations of the reduction semantics and of type-checking. We decided to do this with a simple imperative programming language specification before we start studying and modifying more complex imperative languages, namely languages that include object-oriented features.

The While language presented in this article is a small imperative language that allows non-deterministic and parallel execution of statements and also the use of blocks with local variable and procedure declarations. The syntax, presented in Section 2, is based on the same syntax of While presented in chapters two and three of [7], although we changed some existing constructs and extended it with a runtime syntax, including a evaluation context.

Section 3 presents the operational semantics, based on the one presented in chapter two of [7]. In addition to those rules, we defined the reduction rules for non-deterministic, parallel and block constructs using structural operational semantics because, although they are presented in chapter three of [7], the authors presented them using only natural semantics. We choose to use structural operational semantics since small-step reduction allows us to specify in detail the behaviour of the language in a concurrent context, which it is not possible using natural semantics.

Section 4 presents the type system we created for While. In a similar way to the operational semantics, the typing rules use two different environments, one for declared variables and one for declared procedures. In the type system, variables are represented by its type but procedures are represented by a variable environment that contains all of the variables declared in the procedure. The typing rules for the expressions receive both environments as input, while typing rules for statements receive both as input and returns those environments, possibly modified, as output, so that the next rule is aware of the changes done to the program state.

Section 5 presents the Racket language [3], a programming language that supports other programming languages. Racket offers PLT Redex [2], a domain-specific language embedded in Racket that allows programmers to formalize and debug programming languages. We implemented the While as formalized in this article using PLT Redex and tested it using small programs defined by us. The code of the implementation, along with the programs we used to test it, is available at `https://bitbucket.org/cvasconcelos/thesis/src/876fc254db76aca1bb058b7e6ef069ee23c6c237/While/while.rkt`.

2 Syntax

For presentation sake, the syntax of While is divided into three parts.

2.1 Basic syntax

The basic syntax of While, defined in Figure 1, is based on the syntax presented in [7, p. 7], which contains basic arithmetic and boolean expressions and statements. Let n stand for an natural value and consider a set of variables symbols ranged over by x. In addition to that, we add some primitive types, a new type of value for statements (void) and instead of including the variable assignment expression we created two new statements: One for variable declaration and one for variable update. Figures 4, 5 and 6 show examples of programs in the abstract syntax of the While language.

2.2 Extended syntax

The extended syntax of While, defined in Figure 2, is based on some of the advanced constructs presented in [7, p. 47 - 56]. Consider a set of procedure names ranged over by p. Blocks and procedures are added to the syntax, allowing to specify a block inside a program. The While language has dynamic scope for variables and procedures, meaning that each block has its own scope.

This extension also specifies par, a construct for parallel execution of two statements in a interleaved way, and protect, a construct for atomic execution of a statement.

Figures 7 and 8 show examples of programs in the abstract syntax of the While language that use these new features.

2.3 Runtime syntax

The runtime syntax of the While language is composed by a set of constructs necessary during runtime, i.e., by the reduction and/or the typing rules, and

Basic Syntax

$$
\begin{array}{rrcl}
\text{(Arithmetic expressions)} & a & ::= & n \mid x \\
 & & & \mid a + a \mid a - a \mid a * a \\
\text{(Boolean expressions)} & b & ::= & \textsf{true} \mid \textsf{false} \mid x \\
 & & & \mid a = a \mid a \leq a \mid b \wedge b \\
 & & & \mid \neg b \\
\text{(Values)} & val & ::= & n \mid \textsf{true} \mid \textsf{false} \mid \textsf{void} \\
\text{(Types)} & t & ::= & \textsf{Nat} \mid \textsf{Bool} \mid \textsf{Cmd} \\
\text{(Statements)} & S & ::= & S;S \text{ [illegible]} \\
 & & & \mid \textsf{if } b \textsf{ t}\text{[illegible]} \textsf{else } S \mid \textsf{while } b \textsf{ do } S \\
 & & & \mid \textsf{var } t\ x\ := a \mid \textsf{var } t\ x\ := b \\
 & & & \mid x := a \mid x := b
\end{array}
$$

Figure 1: Basic syntax

Extended Syntax

$$
\begin{array}{rrcl}
\text{(Statements)} & S & ::= & \ldots \mid \textsf{begin } D_v\ D_p\ S \textsf{ end} \mid \textsf{call } p \\
 & & & \mid S \textsf{ par } S \mid \textsf{protect } S \textsf{ end} \\
\text{(Variable declarations)} & D_v & ::= & \varepsilon \mid D_v;D_v \\
 & & & \mid \textsf{var } t\ x\ := b \mid \textsf{var } t\ x\ := a \\
\text{(Procedure declarations)} & D_p & ::= & \varepsilon \mid D_p;D_p \mid \textsf{proc } p \textsf{ is } S
\end{array}
$$

Figure 2: Extended syntax

they are not available to the user. The original While language presented in [7] does not have any runtime exclusive constructs so we define the necessary constructs, which are presented in Figure 3.

A new construct, protected, is added to the statement set and is used in the operational semantics to help indicating that a statement must be executed as an atomic entity. This idea is presented in [1]. There is also two new constructs in the statement set, beginscope and endscope, that are used by the operational semantics for scope management. Arithmetic and boolean expressions and values are also added to the statement set because during runtime we need to consider them statements for evaluation purposes.

Finally, the contexts of the While language are defined. These contexts specify how each expression must be evaluated, more specifically the order of execution of each expression.

Runtime Syntax

$$\begin{array}{rrcl}
\text{(Statements)} & S & ::= & \ldots \mid a \mid b \mid \mathit{val} \mid \mathsf{beginscope} \mid \mathsf{endscope} \\
 & & & \mid \mathsf{protected}\ S\ \mathsf{end} \\
\text{(evaluation context)} & \mathcal{E} & ::= & n + \mathcal{E} \mid \mathcal{E} + a \mid n - \mathcal{E} \mid \mathcal{E} - a \\
 & & & \mid n * \mathcal{E} \mid \mathcal{E} * a \\
 & & & \mid n = \mathcal{E} \mid \mathcal{E} = a \mid n \leq \mathcal{E} \mid \mathcal{E} \leq a \\
 & & & \mid \mathsf{true} \wedge \mathcal{E} \mid \mathsf{false} \wedge \mathcal{E} \mid \mathcal{E} \wedge b \mid \neg\mathcal{E} \\
 & & & \mid \mathcal{E}; S \\
 & & & \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{var}\ t\ x\ :=\ \mathcal{E} \mid x\ :=\ \mathcal{E} \\
 & & & \mid \mathcal{E}\ \mathsf{par}\ S \mid S\ \mathsf{par}\ \mathcal{E} \mid \mathsf{protected}\ \mathcal{E}\ \mathsf{end}
\end{array}$$

Figure 3: Runtime syntax

var Nat y := 4; y := y + 1

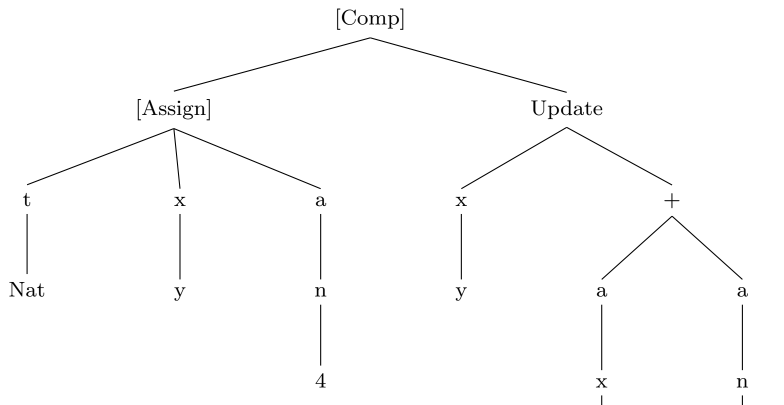


Figure 4: Abstract syntax example 1

var Bool y := false; if ¬ y then var z := 1 else var Nat z := 3

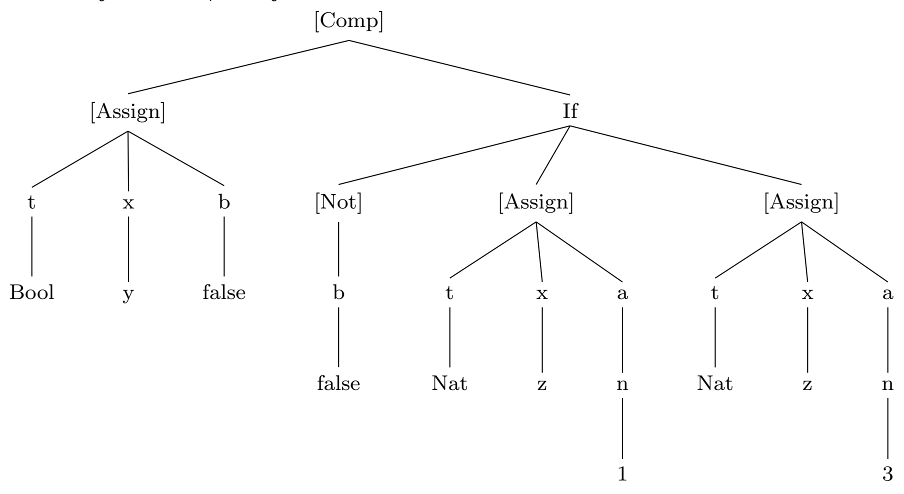


Figure 5: Abstract syntax example 2

3 Operational semantics

The operational semantics of While uses a structural operational semantics approach to specify the behavior of any program in While. The relation is rigorously defined by a set of reduction rules. Most of these rules, more specifically the ones for the basic syntax related expressions, are based on the reduction rules presented in [7, p. 33 - 35]. The form of the rules (judgments) is

$$\frac{P_1 \qquad \ldots \qquad P_n}{\sigma\ \rho\ \vdash\ S \longrightarrow S\ \dashv\ \sigma\ \rho}\ n \geq 0$$

An element of the relation is a pair of triples (σ, ρ, S) where σ and ρ are two environments and S is a statement (c.p. figure 1). Here we consider an environment to be a sequence of maps. In σ each variable x is mapped to a value *val*, while in ρ each procedure p is mapped to a statement *S*. Both of this environments consist of several "levels", each one represented by a map with all of the variables or procedures that belong to a program block it represents. Each environment starts with one level, which is always the global scope, while the "levels" created afterwards represent the scope of program blocks, with the earliest "level" representing the first program block, the "level" after representing the program block inside the first program block, and so on. Each time we get inside a program block a new "level" is added to both environments, and this "level" will be removed when we get out of that same program block. Section 3.1 shows a reduction of a simple program with a program block where this is shown.

The relation is inductively defined by the rules in Figures 9, 10 and 11. The semantics for arithmetic and boolean expressions is omitted because it is identical to the one presented in [7, p. 13 - 15].

Figure 9 presents the reductions rules for basic statements. Rules Assign and Update are both for variable assignment, with the first one being for a

var Nat y := 0; while y = 0 do y := y + 1

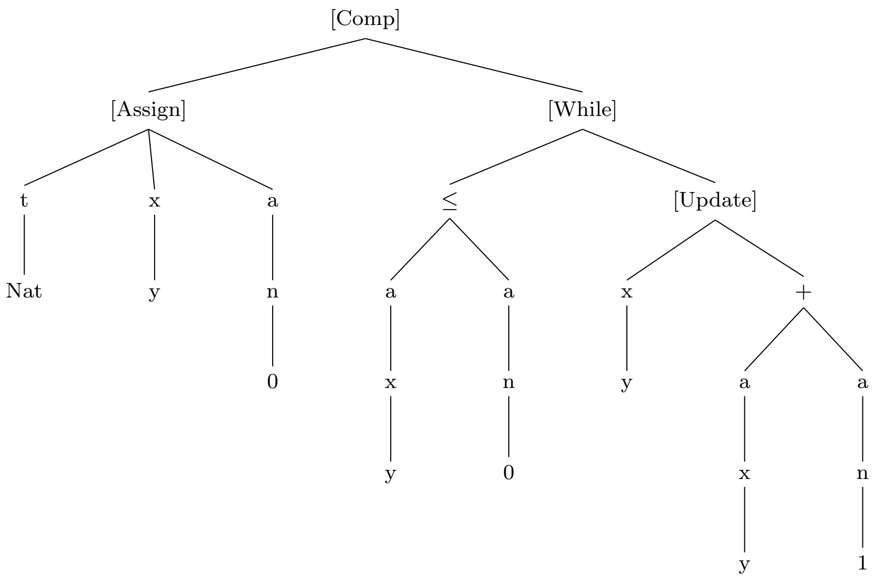


Figure 6: Abstract syntax example 3

begin var w := 2; proc z is var Nat r = 4; call z; w := r end

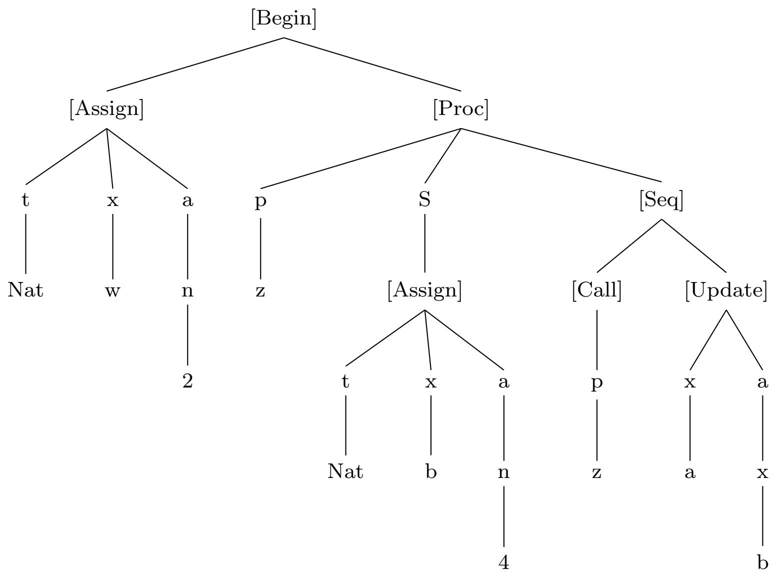


Figure 7: Abstract syntax example 4

var Nat x := 0; protect x := 2; x := 4 end par x := 6

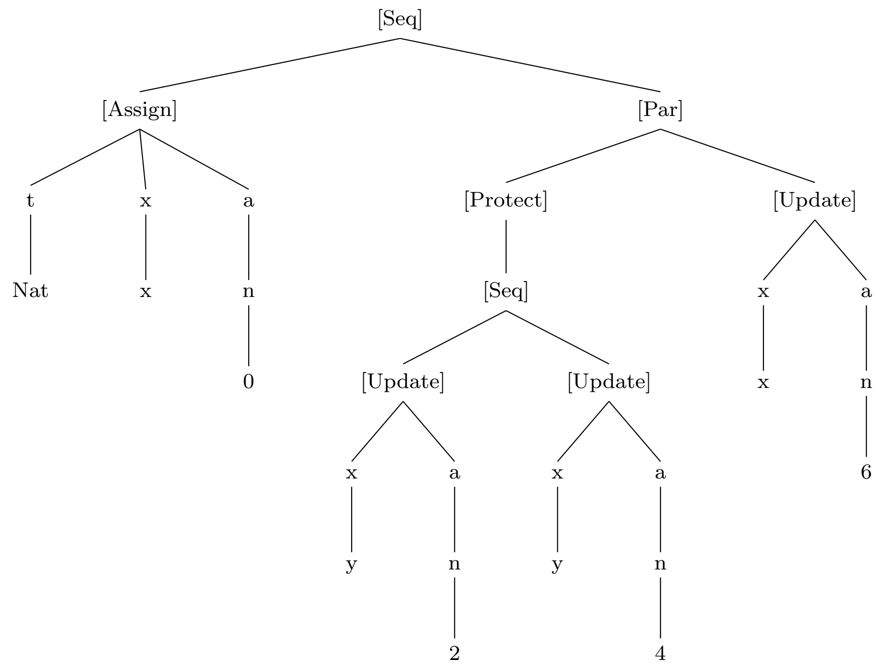


Figure 8: Abstract syntax example 5

new variable declaration, which maps a new variable x to value val in σ , and the second one being for variable update, which updates the value of x to the new value val_2. Rules Seq1 and Seq2 are for sequential compositions.

Since the operational semantics of While is based on a structural operational semantics approach, in a sequential composition the statement S_1 does not necessarily terminate on one computational step. The rule Seq1 expresses this situation, while rule Seq2 expresses a situation where S_1 completely terminates.

Rules If-True and If-False are for $\mathsf{if} - \mathsf{then} - \mathsf{else}$ expressions, reducing them to one of its branches depending on the boolean value that serves as the condition. The condition is evaluated using the semantics of boolean expressions in [7, p. 15]. Axiom While unfolds $\mathsf{while} - \mathsf{do}$ expressions into $\mathsf{if} - \mathsf{then} - \mathsf{else}$ expressions, with the first branch being the execution of statement S and then the execution of the same while expression again, and the second one being a empty branch.

Figure 10 presents the reduction rules for block and procedure related statements. Rule Begin creates a sequence of statements that, in a new scope, will assign variables and procedures and execute the body of the block. Axioms BeginScope and EndScope are for scope management, with the first being used to create a new "level" on the top of both variable environments and the second one to destroy those same "levels".

Proc maps a new variable x to a statement S in ρ while Call reduces a variable x to a statement S to which it is mapped in ρ.

Figure 11 shows the reduction rules for the parallelism and concurrency related statements Par1, Par2, Par3 and Par4 are for parallel execution of

$$\textsc{Assign}\quad \frac{|\sigma'| = 1 \wedge x \notin \mathrm{dom}(\sigma')}{(\sigma, \sigma')\ \rho\ \vdash\ \mathsf{var}\ t\ x := val \longrightarrow \mathsf{void}\ \dashv\ (\sigma, \sigma' \cup \{x \mapsto val\})\ \rho}$$

$$\textsc{Update}\quad \frac{x \in \mathrm{dom}(\sigma)}{\sigma\{x \mapsto val_1\}\ \rho\ \vdash\ x := val_2 \longrightarrow \mathsf{void}\ \dashv\ \sigma\{x \mapsto val_2\}\ \rho}$$

$$\textsc{Seq1}\quad \frac{\sigma\ \rho\ \vdash\ S_1 \longrightarrow S_1'\ \dashv\ \sigma'\ \rho'}{\sigma\ \rho\ \vdash\ S_1; S_2 \longrightarrow S_1'; S_2\ \dashv\ \sigma'\ \rho'} \qquad \textsc{Seq2}\quad \frac{\sigma\ \rho\ \vdash\ S_1 \longrightarrow \mathsf{void}\ \dashv\ \sigma'\ \rho'}{\sigma\ \rho\ \vdash\ S_1; S_2 \longrightarrow S_2\ \dashv\ \sigma'\ \rho'}$$

$$\textsc{If-True}\quad \frac{b = \mathsf{true}}{\sigma\ \rho\ \vdash\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \longrightarrow S_1 \dashv\ \sigma\ \rho}$$

$$\textsc{If-False}\quad \frac{b = \mathsf{false}}{\sigma\ \rho\ \vdash\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \longrightarrow S_2 \dashv\ \sigma\ \rho}$$

$$\textsc{While}\quad \sigma\ \rho\ \vdash\ \mathsf{while}\ b\ \mathsf{do}\ S \longrightarrow\ \mathsf{if}\ b\ \mathsf{then}\ S;\ \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{else\ void} \dashv\ \sigma\ \rho$$

Figure 9: Reduction rules for basic statements

$$\textsc{Begin}\quad \sigma\ \rho\ \vdash\ \mathsf{begin}\ D_v\ D_p\ S \longrightarrow\ \mathsf{beginscope}; D_v; D_p; S; \mathsf{endscope} \dashv\ \sigma'\ \rho$$

$$\textsc{BeginScope}\quad \sigma\ \rho\ \vdash\ \mathsf{beginscope} \longrightarrow \mathsf{void} \dashv\ (\sigma', \sigma)\ (\rho', \rho)$$

$$\textsc{EndScope}\quad (\sigma', \sigma)\ (\rho', \rho)\ \vdash\ \mathsf{endscope} \longrightarrow \mathsf{void} \dashv\ \sigma\ \rho$$

$$\textsc{Proc}\quad \frac{|\rho'| = 1 \wedge p \notin \mathrm{dom}(\rho')}{\sigma\ (\rho, \rho')\ \vdash\ \mathsf{proc}\ p\ \mathsf{is}\ S \longrightarrow void \dashv\ \sigma\ (\rho, \rho' \cup \{p \mapsto S\})}$$

$$\textsc{Call}\quad \frac{p \in \mathrm{dom}(\rho')}{\sigma\ \rho\{p \mapsto S\}\ \vdash\ \mathsf{call}\ p \longrightarrow S \dashv\ \sigma'\ \rho\{p \mapsto S\}}$$

Figure 10: Reduction rules for blocks and procedures statements

statements and they reflect the non deterministic and interleaved execution of the statements. The predicate *protected* used in this four rules is presented in [1]. Rule PROTECT works similar to a lock mechanism, where a statement S obtains a lock if available so it can execute as an atomic entity. Rule PROTECTED allows that same statement to release the lock.

$$\textsc{Par1}\quad \frac{\sigma\ \rho\ \vdash\ S_1 \longrightarrow S_1'\ \dashv\ \sigma'\ \rho' \qquad \neg\mathrm{protected}(S_1)}{\sigma\ \rho\ \vdash\ S_1\ \mathsf{par}\ S_2 \longrightarrow S_1'\ \mathit{par}\ S_2\ \dashv\ \sigma'\ \rho'}$$

$$\textsc{Par2}\quad \frac{\sigma\ \rho\ \vdash\ S_1 \longrightarrow \mathit{void}\ \dashv\ \sigma'\ \rho' \qquad \neg\mathrm{protected}(S_1)}{\sigma\ \rho\ \vdash\ S_1\ \mathsf{par}\ S_2 \longrightarrow S_2\ \dashv\ \sigma'\ \rho'}$$

$$\textsc{Par3}\quad \frac{\sigma\ \rho\ \vdash\ S_2 \longrightarrow S_2'\ \dashv\ \sigma'\ \rho' \qquad \neg\mathrm{protected}(S_2)}{\sigma\ \rho\ \vdash\ S_1\ \mathsf{par}\ S_2 \longrightarrow S_1\ \mathit{par}\ S_2'\ \dashv\ \sigma'\ \rho'}$$

$$\textsc{Par4}\quad \frac{\sigma\ \rho\ \vdash\ S_2 \longrightarrow S_2'\ \dashv\ \sigma'\ \rho' \qquad \neg\mathrm{protected}(S_2)}{\sigma\ \rho\ \vdash\ S_1\ \mathsf{par}\ S_2 \longrightarrow S_1\ \dashv\ \sigma'\ \rho'}$$

$$\textsc{Protect}\quad \sigma\ \rho\ \vdash\ \mathsf{protect}\ S\ \mathsf{end} \longrightarrow \mathsf{protected}\ S\ \mathsf{end}\ \dashv\ \sigma\ \rho$$

$$\textsc{Protected}\quad \sigma\ \rho\ \vdash\ \mathsf{protected}\ \mathit{val}\ \mathsf{end} \longrightarrow \mathsf{void}\ \dashv\ \sigma\ \rho$$

Figure 11: Reduction rules for parallelism and concurrency statements

$$\mathit{protected}(S) \overset{\text{def}}{=} \begin{cases} \mathit{tt} & \text{if } S \text{ is } \mathsf{protect}\ S\ \mathsf{end} \\ \mathit{protected}(S_1) & \text{if } S \text{ is } S_1; S_2 \\ \mathit{protected}(S_1) \vee \mathit{protected}(S_2) & \text{if } S\ \mathit{is}\ S_1\ \mathsf{par}\ S_2 \\ \mathit{ff} & \text{otherwise} \end{cases}$$

Figure 12: Protected predicate

3.1 Reduction example

Consider the program

$$\textsf{begin var Nat } a := 4;\ b := 2$$

and the environments $\sigma = (\{a \mapsto 3, b \mapsto 5\})$ and $\rho = (\{\ \})$. First, we apply the axiom Begin and get

$$\sigma\ \rho\ \vdash\ \textsf{begin var Nat } a := 4;\ b := 2 \longrightarrow S \dashv\ \sigma\ \rho$$

where $S = \textsf{beginscope};\ \textsf{var Nat } a := 4;\ b := 2;\ \textsf{endscope}$. Using the rule Seq1 and the axiom BeginScope we get

$$\frac{\sigma\ \rho\ \vdash\ \textsf{beginscope} \longrightarrow \textsf{void} \dashv\ \sigma^1\ \rho^1}{\sigma\ \rho\ \vdash\ S \longrightarrow S' \dashv\ \sigma^1\ \rho^1}$$

where $\sigma^1 = (\{a \mapsto 3, b \mapsto 5\}, \{\ \})$, $\rho = (\{\ \}, \{\ \})$ and $S' = \textsf{var Nat } a := 4;\ b := 2;\ \textsf{endscope}$. Notice that both environments now have a new "level". Using the rules Seq1 and Assign we get

$$\frac{\dfrac{x \notin \sigma^1}{\sigma^1\ \rho^1\ \vdash\ \textsf{var Nat } a := 4 \longrightarrow \textsf{void} \dashv\ \sigma^2\ \rho^1}}{\sigma^1\ \rho^1\ \vdash\ S' \longrightarrow b := 2;\ \textsf{endscope} \dashv\ \sigma^2\ \rho^1}$$

where$\sigma^2 = (\{a \mapsto 3, b \mapsto 5\}, \{a \mapsto 4\})$. Although a exists in σ^2, the rule Assign checks if a exists in the deepest "level" of σ^1 which is the case. So, a will be added to the deepest "level" and will be mapped to value 4. Continuing the reduction process, we now use the rules Seq1 and Update to get

$$\frac{\dfrac{x \in \sigma'}{\sigma^2\ \rho^1\ \vdash\ b := 2 \longrightarrow \textsf{void} \dashv\ \sigma^3\ \rho^1}}{\sigma^2\ \rho^1\ \vdash\ b := 2;\ \textsf{endscope} \longrightarrow \textsf{endscope} \dashv\ \sigma^3\ \rho^1}$$

where $\sigma^3 = (\{a \mapsto 3, b \mapsto 2\}, \{a \mapsto 4\})$. The rule Update updated the value of the most recent mapping of b, which is the only one in the first "level". Finally, we use the rule EndScope and get

$$\sigma^3\ \rho\ \vdash\ \textsf{endscope} \longrightarrow \textsf{void} \dashv\ \sigma^4\ \rho$$

where $\sigma^4 = (\{a \mapsto 3, b \mapsto 2\})$. Notice that both the environments now had their last level, both corresponding to the scope of the block we just terminated, removed, while the changes made in other "levels" inside the block still remain. We reached the end of the program.

4 Type system

Since [7] does not present a type system for While, we define one from scratch. This type system, similarly to the operational semantics, uses two typing environments: Γ for variables and Δ for procedures. In Γ each variable x is mapped to a type t, and in Δ each procedure p is mapped to a typing environment Γ with all the variables declared inside that procedure. In this context we consider an environment to be just one map instead of several maps.

The typing rules of this type system for expressions have the following form:

$$\frac{P_1 \qquad \ldots \qquad P_n}{\Gamma\ \Delta\ \vdash\ S : t}\ n \geq 0$$

The typing rules for commands have a similar form to the reduction rules, with input and output environments:

$$\frac{P_1 \qquad \ldots \qquad P_n}{\Gamma\ \Delta\ \vdash\ S : t\ \dashv\ \Gamma\ \Delta}\ n \geq 0$$

Figures 13 to 16 show the typing rules of While.

Figure 13 shows the typing axioms for values. Axioms T-True, T-False, T-Nat and T-Void just evaluate simple values, while axiom T-Var evaluates a variable x based on its most recent mapping in Γ.

Figure 14 shows the typing rules for all arithmetic and boolean expressions of While. In each of this rules the type checker evaluates the expression, checking if each operand has the correct type for the expression.

Figure 15 has the typing rules for simple statements of While. Rule T-Assign evaluates first a statement S and then maps a variable x to the type t of S. Rule T-Update just checks if the statement S has the same type has the variable to be updated.

Rule T-Seq evaluates the first statement and then evaluates the second statement taking in consideration all the changes caused by the first statement.

Rule T-If checks if the expressions that serves has the condition is of type Bool, evaluates both branches in the same conditions (same typing environments as input) and returns a new Γ which is the union between both Γ returned by each branch. Rule T-While also checks the condition first but it specifies that S must not change any of the typing environments (so it does not allow variable assignment in S).

Figure 16 shows the typing rules for blocks and procedures statements: Rule T-Begin evaluates D_v, D_p and S in a similar to rule T-Seq, with the environments returned by each statement to be used as the input typing environments for the next statement. In the end, this rule returns the same typing environments used as input since every change done to these environments can only be visible inside the block.

For procedures, rule T-Proc evaluates a statement S and maps a procedure p to the typing environment Γ returned by S in Δ, and rule T-Call returns as Γ the union of the Γ given as input and the Γ which p is mapped to.

Figure 17 shows the typing rules for the concurrent and parallel statements T-Par and T-Protect. Both just evaluate their statements S.

$$\textsc{T-Nat}\quad \Gamma\ \Delta\ \vdash\ n : \mathsf{Nat} \qquad\qquad \textsc{T-Var}\quad \Gamma\{x \mapsto t\}\ \Delta\ \vdash\ x : t$$

$$\textsc{T-True}\quad \Gamma\ \Delta\ \vdash\ \mathsf{true} : \mathsf{Bool} \qquad\qquad \textsc{T-False}\quad \Gamma\ \Delta\ \vdash\ \mathsf{false} : \mathsf{Bool}$$

$$\textsc{T-Empty}\quad \Gamma\ \Delta\ \vdash\ \epsilon : \mathsf{Cmd}$$

Figure 13: Typing rules for values

$$\textsc{T-Add}\quad \frac{\Gamma\ \Delta\ \vdash\ a_1 : \mathsf{Nat} \qquad \Gamma\ \Delta\ \vdash\ a_2 : \mathsf{Nat}}{\Gamma\ \Delta\ \vdash\ a_1 + a_2 : \mathsf{Nat}}$$

$$\textsc{T-Sub}\quad \frac{\Gamma\ \Delta\ \vdash\ a_1 : \mathsf{Nat} \qquad \Gamma\ \Delta\ \vdash\ a_2 : \mathsf{Nat}}{\Gamma\ \Delta\ \vdash\ a_1 - a_2 : \mathsf{Nat}}$$

$$\textsc{T-Mult}\quad \frac{\Gamma\ \Delta\ \vdash\ a_1 : Nat \qquad \Gamma\ \Delta\ \vdash\ a_2 : \mathsf{Nat}}{\Gamma\ \Delta\ \vdash\ a_1 * a_2 : \mathsf{Nat}}$$

$$\textsc{T-Equal}\quad \frac{\Gamma\ \Delta\ \vdash\ a_1 : \mathsf{Nat} \qquad \Gamma\ \Delta\ \vdash\ a_2 : \mathsf{Nat}}{\Gamma\ \Delta\ \vdash\ a_1 = a_2 : \mathsf{Bool}}$$

$$\textsc{T-LEqual}\quad \frac{\Gamma\ \Delta\ \vdash\ a_1 : \mathsf{Nat} \qquad \Gamma\ \Delta\ \vdash\ a_2 : \mathsf{Nat}}{\Gamma\ \Delta\ \vdash\ a_1 \leq a_2 : \mathsf{Bool}\ \dashv\ \Gamma\ \Delta}$$

$$\textsc{T-LEqual}\quad \frac{\Gamma\ \Delta\ \vdash\ b_1 : \mathsf{Bool} \qquad \Gamma\ \Delta\ \vdash\ b_2 : \mathsf{Bool}}{\Gamma\ \Delta\ \vdash\ b_1 \wedge b_2 : \mathsf{Bool}} \qquad\qquad \textsc{T-Not}\quad \frac{\Gamma\ \Delta\ \vdash\ b : \mathsf{Bool}}{\Gamma\ \Delta\ \vdash\ \neg\, b : \mathsf{Bool}}$$

Figure 14: Typing rules for arithmetic and boolean expressions

$$\textsc{T-Assign}\quad \frac{\Gamma\ \Delta\ \vdash\ S : t\ \dashv\ \Gamma\ \Delta}{\Gamma\ \Delta\ \vdash\ \mathsf{var}\ t\ x := e : \mathsf{Cmd}\ \dashv\ \Gamma \cup \{x \mapsto t\}\ \Delta}$$

$$\textsc{T-Update}\quad \frac{\Gamma\ \Delta\ \vdash\ S : t\ \dashv\ \Gamma\ \Delta}{\Gamma\{x \mapsto t\}\ \Delta\ \vdash\ x := S : \mathsf{Cmd}\ \dashv\ \Gamma\{x \mapsto t\}\ \Delta}$$

$$\textsc{T-Seq}\quad \frac{\Gamma\ \Delta\ \vdash\ S_1 : t_1\ \dashv\ \Gamma'\ \Delta \qquad \Gamma'\ \Delta\ \vdash\ S_2 : t_2\ \dashv\ \Gamma''\ \Delta}{\Gamma\ \Delta\ \vdash\ S_1; S_2 : t_2\ \dashv\ \Gamma''\ \Delta}$$

$$\textsc{T-If}\quad \frac{\Gamma\ \Delta\ \vdash\ b : \mathsf{Bool} \qquad \Gamma\ \Delta\ \vdash\ S_1 : t\ \dashv\ \Gamma'\ \Delta \qquad \Gamma\ \Delta\ \vdash\ S_2 : t\ \dashv\ \Gamma''\ \Delta}{\Gamma\ \Delta\ \vdash\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 : t\ \dashv\ (\Gamma'' \cup \Gamma'')\ \Delta}$$

$$\textsc{T-While}\quad \frac{\Gamma\ \Delta\ \vdash\ b : \mathsf{Bool} \qquad \Gamma\ \Delta\ \vdash\ S : \mathsf{Cmd}\ \dashv\ \Gamma\ \Delta}{\Gamma\ \Delta\ \vdash\ \mathsf{while}\ b\ \mathsf{do}\ S : \mathsf{Cmd}\ \dashv\ \Gamma\ \Delta}$$

Figure 15: Typing rules for simple statements

$$\textsc{T-Begin}\quad \frac{\begin{array}{c}\Gamma\ \Delta\ \vdash\ D_v : \mathsf{Cmd}\ \dashv\ \Gamma'\ \Delta \\ \Gamma'\ \Delta\ \vdash\ D_p : \mathsf{Cmd}\ \dashv\ \Gamma'\ \Delta' \qquad \Gamma'\ \Delta'\ \vdash\ S : \mathsf{Cmd}\ \dashv\ \Gamma''\ \Delta\end{array}}{\Gamma\ \Delta\ \vdash\ \mathsf{begin}\ D_v\ D_p\ S\ \mathsf{end} : \mathsf{Cmd}\ \dashv\ \Gamma\ \Delta}$$

$$\textsc{T-Call}\quad \Gamma\ \Delta\{p \mapsto \Gamma'\}\ \vdash\ \mathsf{call}\ p : \mathsf{Cmd}\ \dashv\ (\Gamma \cup \Gamma')\ \Delta\{p \mapsto \Gamma'\}$$

$$\textsc{T-Proc}\quad \frac{\Gamma\ \Delta\ \vdash\ S : \mathsf{Cmd}\ \dashv\ \Gamma'\ \Delta}{\Gamma\ \Delta\ \vdash\ \mathsf{proc}\ p\ \mathsf{is}\ S : \mathsf{Cmd}\ \dashv\ \Gamma\ \Delta \cup \{p \mapsto \Gamma' \backslash \Gamma\}}$$

Figure 16: Typing rules for block statements

$$\textsc{T-Par}\quad \frac{\Gamma\ \Delta\ \vdash\ S_1 : t_1\ \dashv\ \Gamma'\ \Delta \qquad \Gamma\ \Delta\ \vdash\ S_2 : t_2\ \dashv\ \Gamma''\ \Delta}{\Gamma\ \Delta\ \vdash\ S_1\ \mathsf{par}\ S_2 : \mathsf{Cmd}\ \dashv\ (\Gamma' \cup \Gamma'')\ \Delta}$$

$$\textsc{T-Protect}\quad \frac{\Gamma\ \Delta\ \vdash\ S : t\ \dashv\ \Gamma'\ \Delta}{\Gamma\ \Delta\ \vdash\ \mathsf{protect}\ S\ \mathsf{end} : \mathsf{Cmd}\ \dashv\ \Gamma'\ \Delta}$$

Figure 17: Typing rules for concurrent statements

Figures 18, 19 and 20 show examples of derivations of correct programs in While using the type system defined. Figure 21 shows an example of a badly constructed program that the type checker fails to evaluate due to the scope defined for While. In this example, when applying the rule T-Add, the type checker expects y to be of type Nat and while there is one y of type Nat, it is inside a block, so the current y is of type Bool.

$$\dfrac{\dfrac{\dfrac{}{\Gamma\ \Delta\ \vdash\ \mathsf{true} : \mathsf{Bool}\ \dashv\ \Gamma\ \Delta}\ \textsc{T-True}}{\Gamma\ \Delta\ \vdash\ \neg\ \mathsf{true} : t\ \dashv\ \Gamma\ \Delta}\ \textsc{T-Not} \qquad \mathrm{T1} \qquad \mathrm{T2}}{\Gamma\ \Delta\ \vdash\ \mathsf{if}\ \neg\mathsf{true}\ \mathsf{then}\ \mathsf{var}\ \mathsf{Nat}\ y := 2\ \mathsf{else}\ \mathsf{var}\ \mathsf{Nat}\ z := 4 : t\ \dashv\ \Gamma\{y \mapsto \mathsf{Nat}, z \mapsto \mathsf{Nat}\}\ \Delta}\ \textsc{T-If}$$

$$\mathrm{T1} \quad \dfrac{\dfrac{}{\Gamma\ \Delta\ \vdash\ 2 : \mathsf{Nat}\ \dashv\ \Gamma\ \Delta}\ \textsc{T-Nat}}{\Gamma\ \Delta\ \vdash\ \mathsf{var}\ \mathsf{Nat}\ y := 2 : \mathsf{Cmd}\ \dashv\ \Gamma\{y \mapsto \mathsf{Nat}\}\ \Delta}\ \textsc{T-Assign}$$

$$\mathrm{T2} \quad \dfrac{\dfrac{}{\Gamma\ \Delta\ \vdash\ 4 : \mathsf{Nat}\ \dashv\ \Gamma\ \Delta}\ \textsc{T-Nat}}{\Gamma\ \Delta\ \vdash\ \mathsf{var}\ \mathsf{Nat} z := 4 : \mathsf{Cmd}\ \dashv\ \Gamma\{z \mapsto \mathsf{Nat}\}\ \Delta}\ \textsc{T-Assign}$$

$\Gamma = \varnothing$
$\Gamma^1 = \{y \mapsto \mathsf{Nat}\}$
$\Gamma^2 = \{z \mapsto \mathsf{Nat}\}$
$\Gamma^3 = \{y \mapsto \mathsf{Nat}, z \mapsto \mathsf{Nat}\}$
$\Delta = \varnothing$

Figure 18: Typing example 1

$$\dfrac{\dfrac{\dfrac{}{\Gamma\ \Delta\ \vdash\ 1 : \mathsf{Nat}\ \dashv\ \Gamma\ \Delta}\ \textsc{T-Nat}}{\Gamma\ \Delta\ \vdash\ \mathsf{var}\ \mathsf{Nat}\ x := 4 : \mathsf{Cmd}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Assign} \qquad T1}{\Gamma\ \Delta\ \vdash\ \mathsf{var}\ \mathsf{Nat}\ x := 1; \mathsf{while}\ x \leq 4\ \mathsf{do}\ x := x + 1 : \mathsf{Cmd}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Seq}$$

$$\mathrm{T1} \quad \dfrac{\dfrac{\dfrac{}{\Gamma^1\ \Delta\ \vdash x : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Var} \qquad \dfrac{}{\Gamma^1\ \Delta\ \vdash 4 : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Nat}}{\Gamma^1\ \Delta\ \vdash\ x \leq 4 : \mathsf{Bool}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-LEqual} \qquad \mathrm{T2}}{\Gamma\ \Delta\ \vdash\ \mathsf{while}\ x \leq 4\ \mathsf{do}\ x := x + 1 : \mathsf{Cmd}\ \dashv\ \Gamma\{z \mapsto \mathsf{Nat}\}\ \Delta}\ \textsc{T-While}$$

$$\mathrm{T2} \quad \dfrac{\dfrac{}{\Gamma^1\ \Delta\ \vdash x : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Var} \qquad \mathrm{T3}}{\Gamma^1\ \Delta\ \vdash x := x + 1 : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Update}$$

$$\mathrm{T3} \quad \dfrac{\dfrac{}{\Gamma^1\ \Delta\ \vdash x : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Var} \qquad \dfrac{}{\Gamma^1\ \Delta\ \vdash 1 : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Nat}}{\Gamma^1\ \Delta\ \vdash x + 1 : \mathsf{Nat}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Var}$$

$\Gamma = \varnothing$
$\Gamma^1 = \{x \mapsto \mathsf{Nat}\}$
$\Delta = \varnothing$

Figure 19: Typing example 2

$$\dfrac{\mathrm{T1} \qquad \mathrm{T2} \qquad \mathrm{T3}}{\Gamma\ \Delta\ \vdash\ \mathsf{begin}\ Dv\ Dp\ S\ \mathsf{end} : \mathsf{Cmd}\ \dashv\ \Gamma\ \Delta}\ \textsc{T-Begin}$$

$$\mathrm{T1} \quad \dfrac{\dfrac{\dfrac{}{\Gamma\ \Delta\ \vdash\ 2 : \mathsf{Nat}}\ \textsc{T-Nat}}{\Gamma\ \Delta\ \vdash\ \mathsf{var\ Nat}\ x\ :=\ 2 : \mathsf{Cmd}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Assign} \qquad \mathrm{T4}}{\Gamma\ \Delta\ \vdash\ \mathsf{var\ Nat}\ x\ :=\ 2; \mathsf{var\ Bool}\ y\ :=\ \mathsf{true} : \mathsf{Cmd}\ \dashv\ \Gamma^2\ \Delta}\ \textsc{T-Seq}$$

$$\mathrm{T4} \quad \dfrac{\dfrac{}{\Gamma^1\ \Delta\ \vdash\ \mathsf{true} : \mathsf{Bool}}\ \textsc{T-True}}{\Gamma^1\ \Delta\ \vdash\ \mathsf{var\ Bool}\ y\ :=\ \mathsf{true} : \mathsf{Cmd}\ \dashv\ \Gamma^2\ \Delta}\ \textsc{T-Assign}$$

$$\mathrm{T2} \quad \dfrac{\dfrac{\dfrac{}{\Gamma^2\ \Delta\ \vdash\ 1 : \mathsf{Nat}}\ \textsc{T-Nat}}{\Gamma^2\ \Delta\ \vdash\ \mathsf{var\ Nat}\ y\ :=\ 1 : \mathsf{Cmd}\ \dashv\ \Gamma^3\ \Delta}\ \textsc{T-Assign}}{\Gamma^2\ \Delta\ \vdash\ \mathsf{proc}\ q\ \mathsf{is\ var\ Nat}\ y\ :=\ 1 : \mathsf{Cmd}\ \dashv\ \Gamma^2\ \Delta^1}\ \textsc{T-Proc}$$

$$\mathrm{T3} \quad \dfrac{\dfrac{}{\Gamma^2\ \Delta^1\ \vdash\ \mathsf{call}\ q : \mathsf{Cmd}\ \dashv\ \Gamma^4\ \Delta^1}\ \textsc{T-Call} \qquad \mathrm{T5}}{\Gamma^2\ \Delta^1\ \vdash\ \mathsf{call}\ q; x\ :=\ y : \mathsf{Cmd}\ \dashv\ \Gamma^4\ \Delta^1}\ \textsc{T-Seq}$$

$$\mathrm{T5} \quad \dfrac{\dfrac{}{\Gamma^4\ \Delta\ \vdash\ x : \mathsf{Nat}}\ \textsc{T-Var} \qquad \dfrac{}{\Gamma^4\ \Delta\ \vdash\ y : \mathsf{Nat}}\ \textsc{T-Var}}{\Gamma^4\ \Delta\ \vdash\ x\ :=\ y : \mathsf{Cmd}\ \dashv\ \Gamma^4\ \Delta}\ \textsc{T-Update}$$

$D_v = \mathsf{var\ Nat}\ x := 2; \mathsf{var\ Bool}\ y := \mathsf{true}$
$D_v = \mathsf{proc}\ p\ \mathsf{is\ var\ Nat}\ y := 1$
$S = \mathsf{call}\ p; x + y$

$\Gamma = \varnothing$
$\Gamma^1 = \{x \mapsto \mathsf{Nat}\}$
$\Gamma^2 = \{x \mapsto \mathsf{Nat}, y \mapsto \mathsf{Bool}\}$
$\Gamma^3 = \{y \mapsto \mathsf{Nat}\}$
$\Gamma^4 = \{x \mapsto \mathsf{Nat}, y \mapsto \mathsf{Bool}, y \mapsto \mathsf{Nat}\}$
$\Delta = \varnothing\ \Delta^1 = \{p \mapsto \Gamma^3\}$

Figure 20: Typing example 3

$$\dfrac{\dfrac{\dfrac{}{\Gamma\ \Delta\ \vdash\ 1 : \mathsf{Nat}}\ \textsc{T-Nat}}{\Gamma\ \Delta\ \vdash\ \mathsf{var\ Nat}\ y\ :=\ 1 : \mathsf{Cmd}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Assign} \qquad \mathrm{T1}}{\Gamma\ \Delta\ \vdash\ \mathsf{var\ Nat}\ y\ :=\ 1; \mathsf{begin}\ D_v\ D_p\ S : \mathsf{Cmd}\ \dashv\ \Gamma^1\ \Delta}\ \textsc{T-Seq}$$

$$\mathrm{T1} \quad \dfrac{\mathrm{T2} \qquad \dfrac{}{\Gamma\ \Delta\ \vdash\ \epsilon : \mathsf{Cmd}}\ \textsc{T-Empty} \qquad \mathrm{T3}}{\Gamma^1\ \Delta\ \vdash\ \mathsf{begin}\ Dv\ D_p\ S\ \mathsf{end} : \mathsf{Cmd}\ \dashv\ \Gamma^3\ \Delta}\ \textsc{T-Begin}$$

$$\mathrm{T2} \quad \dfrac{\dfrac{\dfrac{}{\Gamma^1\ \Delta\ \vdash\ 2 : \mathsf{Nat}}\ \textsc{T-Nat}}{\Gamma^1\ \Delta\ \vdash\ \mathsf{var\ Nat}\ x\ :=\ 2 : \mathsf{Cmd}\ \dashv\ \Gamma^2\ \Delta}\ \textsc{T-Assign} \qquad \mathrm{T4}}{\Gamma^1\ \Delta\ \vdash\ \mathsf{var\ Nat}\ x\ :=\ 2; \mathsf{var\ Bool}\ y\ :=\ \mathsf{true} : \mathsf{Cmd}\ \dashv\ \Gamma^3\ \Delta}\ \textsc{T-Seq}$$

$$\mathrm{T4} \quad \dfrac{\dfrac{}{\Gamma^2\ \Delta\ \vdash\ \mathsf{true} : \mathsf{Bool}}\ \textsc{T-True}}{\Gamma^2\ \Delta\ \vdash\ \mathsf{var\ Bool}\ y\ :=\ \mathsf{true} : \mathsf{Cmd}\ \dashv\ \Gamma^3\ \Delta}\ \textsc{T-Assign}$$

$$\mathrm{T3} \quad \dfrac{\dfrac{}{\Gamma^3\ \Delta\ \vdash\ x : \mathsf{Nat}}\ \textsc{T-Var} \qquad \mathrm{T5}}{\Gamma^3\ \Delta\ \vdash\ x\ :=\ x\ +\ y : \mathsf{Cmd}\ \dashv\ \Gamma^3\ \Delta}\ \textsc{T-Update}$$

$$\mathrm{T5} \quad \dfrac{\dfrac{}{\Gamma^3\ \Delta\ \vdash\ x : \mathsf{Nat}}\ \textsc{T-Var} \qquad \Gamma^3\ \Delta\ \vdash\ y : \mathsf{Bool}}{\Gamma^3\ \Delta\ \vdash\ x\ +\ y : \mathsf{Nat}\ \dashv\ \Gamma^3\ \Delta}\ \textsc{T-Add}$$

$\Gamma = \varnothing$
$\Gamma^1 = \{y \mapsto \mathsf{Nat}\}$
$\Gamma^2 = \{y \mapsto \mathsf{Nat}, x \mapsto \mathsf{Nat}\}$
$\Gamma^3 = \{y \mapsto \mathsf{Nat}, x \mapsto \mathsf{Nat}, y \mapsto \mathsf{Bool}\}$
$\Delta = \varnothing$

Figure 21: Typing example 4

5 Testing the While formalization

Formally defining a programming language is important since such definition can help detect design errors in the language and interpret and evaluate programs. Since producing derivations of executions or of typing is tedious and error-prone, implementing the reduction rules and the type system is crucial to avoid the above mentioned difficulties but may be very time consuming.

In this section we introduce the Racket language, a programming language that supports other programming languages.

5.1 Racket

Racket is a programming language in the Lisp family, meaning that while it can be used to create solutions like any conventional programming language, it also allows a language-oriented programming, i.e., allows creating new programming languages. To support this feature, Racket provides building blocks for protection mechanisms, which allows the programmers to protect individual components of the language from their clients, and the internalization of extra-linguistic mechanisms, such as project contexts and the delegation of program execution and inspection to external agents, by converting them into linguistic constructs, preventing programmers to resort to mechanism outside Racket [3].

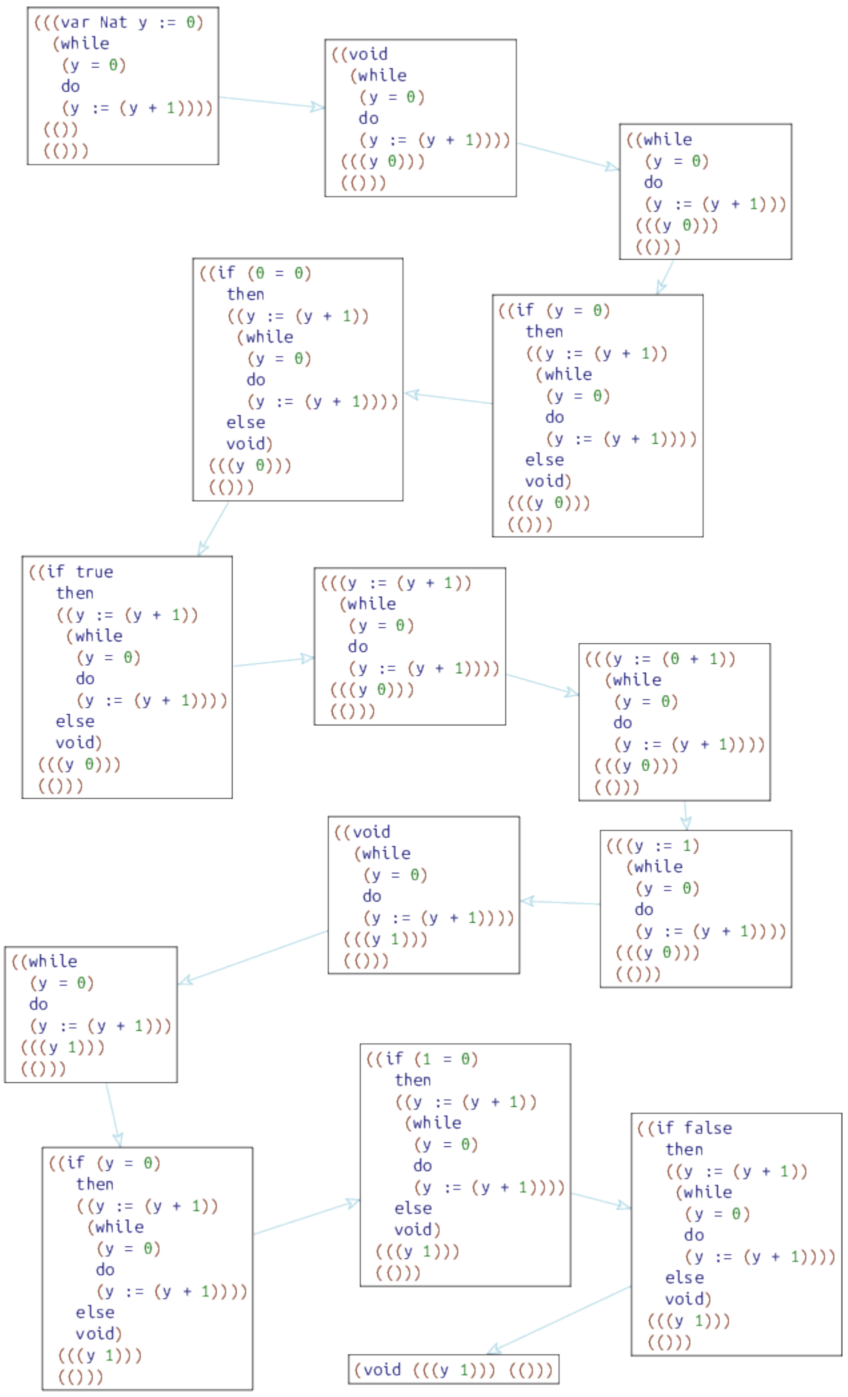
5.2 PLT Redex

PLT Redex is a domain-specific language embedded in Racket that allows programmers to formalize and debug programming languages. The modeling of a programming language in Redex is done by writing down the grammar, reductions of the language along with necessary metafunctions. Since Redex is embedded in Racket, programming in Redex is just like programming in Racket, with all of the features and tools available for Racket being also available for Redex, including DrRacket, a integrated development environment for Racket. One of the most interesting advantages of using DrRacket is the automatically generated reduction graphs that allows programmers to visualize reductions step by step. Redex also has other methods of testing, such as pattern matcher (for grammar testing) and judgment-form evaluation (which we use to test the type system) [2, 6].

PLT Redex is the tool we choose to help us certify our work. To understand it better, we implemented the While language as formalized in this article using PLT Redex [1]. We recommend using DrRacket while trying this and other implementations we provide since it is necessary to visualize the generated reduction graphs.

Figures 22, 23 and 24 show the reduction graphs for simple program examples for the While language.

[1]Available at `https://bitbucket.org/cvasconcelos/thesis/src/876fc254db76aca1bb058b7e6ef069ee23c6c237/While/while.rkt`



```
(((var Nat y := 0)
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (())
 (()))
```

```
((void
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
((while
  (y = 0)
  do
  (y := (y + 1)))
 (((y 0)))
 (()))
```

```
((if (y = 0)
   then
   ((y := (y + 1))
    (while
     (y = 0)
     do
     (y := (y + 1))))
   else
   void)
 (((y 0)))
 (()))
```

```
((if (0 = 0)
   then
   ((y := (y + 1))
    (while
     (y = 0)
     do
     (y := (y + 1))))
   else
   void)
 (((y 0)))
 (()))
```

```
((if true
   then
   ((y := (y + 1))
    (while
     (y = 0)
     do
     (y := (y + 1))))
   else
   void)
 (((y 0)))
 (()))
```

```
(((y := (y + 1))
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
(((y := (0 + 1))
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
(((y := 1)
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 0)))
 (()))
```

```
((void
  (while
   (y = 0)
   do
   (y := (y + 1))))
 (((y 1)))
 (()))
```

```
((while
  (y = 0)
  do
  (y := (y + 1)))
 (((y 1)))
 (()))
```

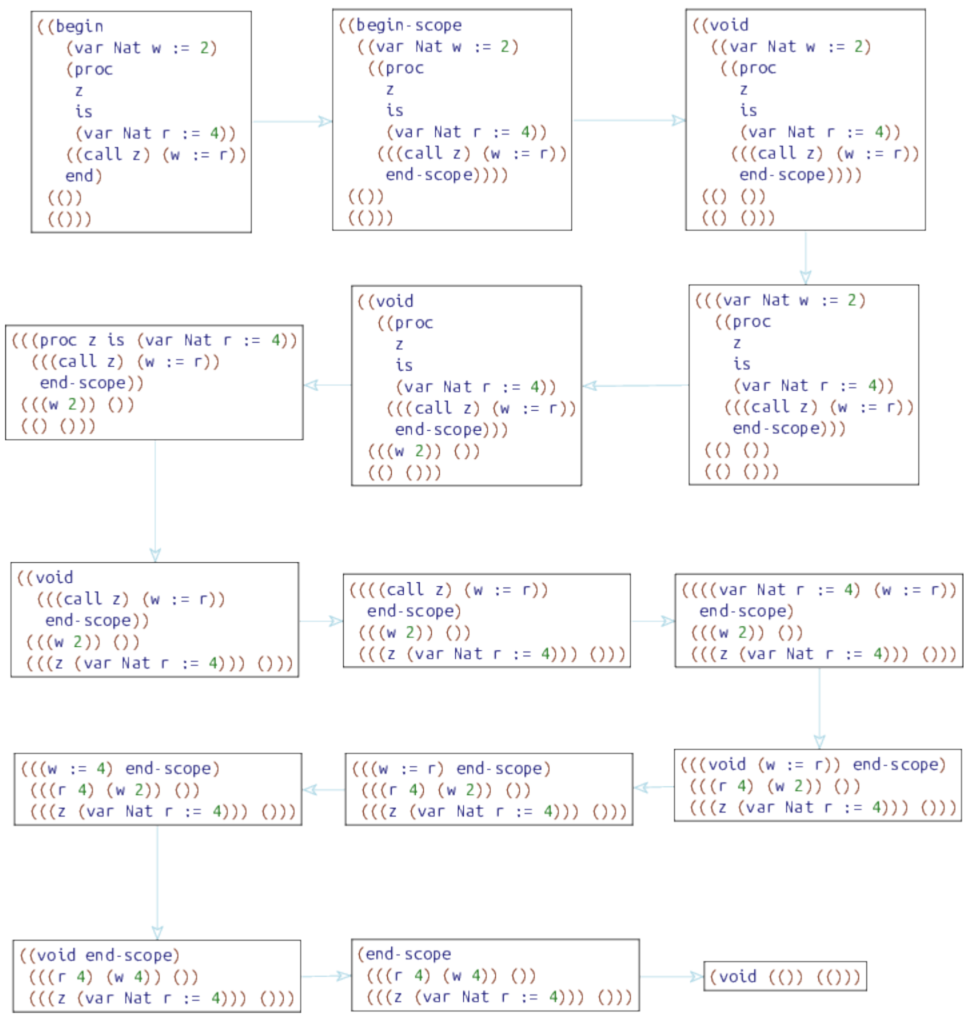
```
((if (y = 0)
   then
   ((y := (y + 1))
    (while
     (y = 0)
     do
     (y := (y + 1))))
   else
   void)
 (((y 1)))
 (()))
```

```
((if (1 = 0)
   then
   ((y := (y + 1))
    (while
     (y = 0)
     do
     (y := (y + 1))))
   else
   void)
 (((y 1)))
 (()))
```

```
((if false
   then
   ((y := (y + 1))
    (while
     (y = 0)
     do
     (y := (y + 1))))
   else
   void)
 (((y 1)))
 (()))
```

```
(void (((y 1))) (()))
```

Figure 22: PLT-Redex reduction graph example 1



```
((begin
   (var Nat w := 2)
   (proc
    z
    is
    (var Nat r := 4))
   ((call z) (w := r))
   end)
 (())
 (()))
```

```
((begin-scope
  ((var Nat w := 2)
   ((proc
     z
     is
     (var Nat r := 4))
    (((call z) (w := r))
     end-scope))))
 (())
 (()))
```

```
((void
  ((var Nat w := 2)
   ((proc
     z
     is
     (var Nat r := 4))
    (((call z) (w := r))
     end-scope))))
 (() ())
 (() ()))
```

```
(((var Nat w := 2)
  ((proc
    z
    is
    (var Nat r := 4))
   (((call z) (w := r))
    end-scope)))
 (() ())
 (() ()))
```

```
((void
  ((proc
    z
    is
    (var Nat r := 4))
   (((call z) (w := r))
    end-scope)))
 (((w 2)) ())
 (() ()))
```

```
(((proc z is (var Nat r := 4))
  (((call z) (w := r))
   end-scope))
 (((w 2)) ())
 (() ()))
```

```
((void
  (((call z) (w := r))
   end-scope))
 (((w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
((((call z) (w := r))
  end-scope)
 (((w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
((((var Nat r := 4) (w := r))
  end-scope)
 (((w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
(((void (w := r)) end-scope)
 (((r 4) (w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
(((w := r) end-scope)
 (((r 4) (w 2)) ())
 (((z (var Nat r := 4))) ()))
```

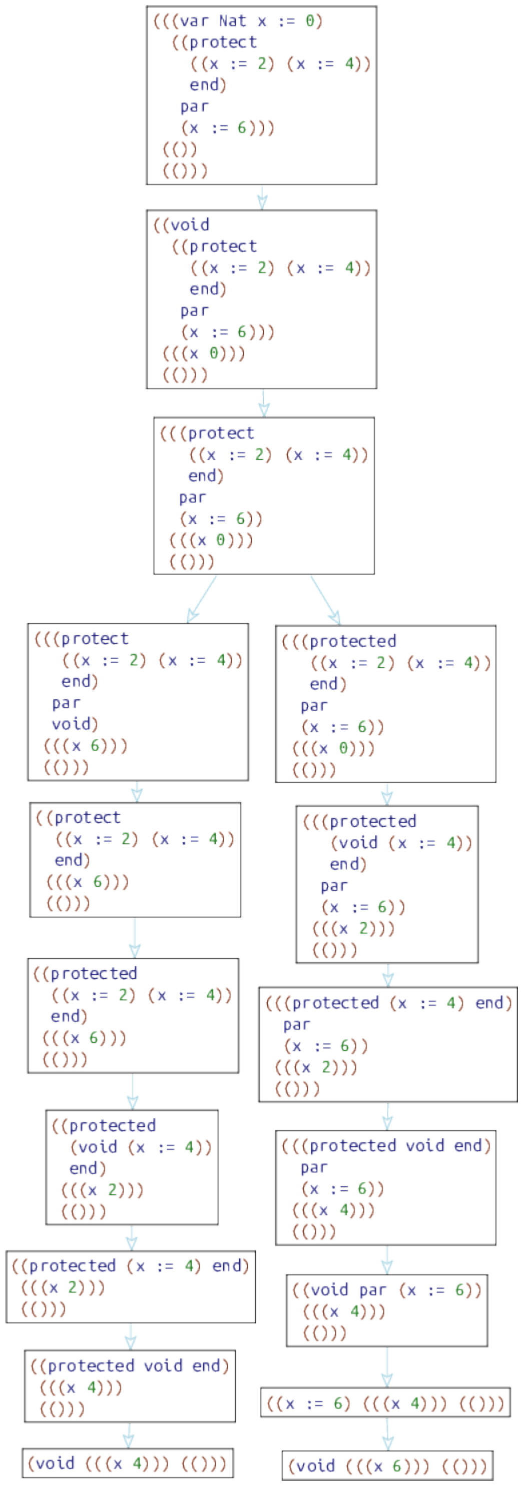
```
(((w := 4) end-scope)
 (((r 4) (w 2)) ())
 (((z (var Nat r := 4))) ()))
```

```
((void end-scope)
 (((r 4) (w 4)) ())
 (((z (var Nat r := 4))) ()))
```

```
(end-scope
 (((r 4) (w 4)) ())
 (((z (var Nat r := 4))) ()))
```

```
(void (()) (()))
```

Figure 23: PLT-Redex reduction graph example 2



```
(((var Nat x := 0)
  ((protect
    ((x := 2) (x := 4))
    end)
   par
   (x := 6)))
 (())
 (()))
```

```
((void
  ((protect
    ((x := 2) (x := 4))
    end)
   par
   (x := 6)))
 (((x 0)))
 (()))
```

```
(((protect
   ((x := 2) (x := 4))
   end)
  par
  (x := 6))
 (((x 0)))
 (()))
```

```
(((protect
   ((x := 2) (x := 4))
   end)
  par
  void)
 (((x 6)))
 (()))
```

```
((protect
  ((x := 2) (x := 4))
  end)
 (((x 6)))
 (()))
```

```
((protected
  ((x := 2) (x := 4))
  end)
 (((x 6)))
 (()))
```

```
((protected
  (void (x := 4))
  end)
 (((x 2)))
 (()))
```

```
((protected (x := 4) end)
 (((x 2)))
 (()))
```

```
((protected void end)
 (((x 4)))
 (()))
```

```
(void (((x 4))) (()))
```

```
(((protected
   ((x := 2) (x := 4))
   end)
  par
  (x := 6))
 (((x 0)))
 (()))
```

```
(((protected
   (void (x := 4))
   end)
  par
  (x := 6))
 (((x 2)))
 (()))
```

```
(((protected (x := 4) end)
  par
  (x := 6))
 (((x 2)))
 (()))
```

```
(((protected void end)
  par
  (x := 6))
 (((x 4)))
 (()))
```

```
((void par (x := 6))
 (((x 4)))
 (()))
```

```
((x := 6) (((x 4))) (()))
```

```
(void (((x 6))) (()))
```

Figure 24: PLT-Redex reduction graph example 3

6 Conclusions and further work

We present and offer an implementation in Racket [3], a programming language that supports other programming languages, of the language While described in a book by Hanne Riis and Flemming Nielson [7]. This implementation directly represents the original syntax and operational semantics of While, faithfully following the definitions presented in the book. One can now automatically build derivations of the possible reductions of any program, observing its step-by-step execution.

Moreover, we define an original type system for the language While, which we also implemented in Racket to provide an automatic type-checking engine.

Future work include stating and proving properties like subject reduction and type safety, in a system like Why3 [4].

References

[1] *Exercise 3.5.* URL: `https://studentportalen.uu.se/uusp-filearea-tool/download.action?nodeId=1110576&toolAttachmentId=207590` (visited on 11/01/2015).

[2] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* 1st. The MIT Press, 2009. ISBN: 0262062755, 9780262062756.

[3] Matthias Felleisen et al. "The Racket Manifesto." In: *SNAPL.* Ed. by Thomas Ball et al. Vol. 32. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128. ISBN: 978-3-939897-80-4. URL: `http://dblp.uni-trier.de/db/conf/snapl/snapl2015.html#FelleisenFFKBMT15`.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 - Where Programs Meet Provers". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP, 2013. Proceedings.* Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. DOI: `10.1007/978-3-642-37036-6_8`.

[5] *Functional Programming vs. Imperative Programming.* URL: `https://msdn.microsoft.com/en-us/library/bb669144.aspx` (visited on 02/03/2016).

[6] Casey Klein et al. "Run Your Research: On the Effectiveness of Lightweight Mechanization". In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 285–296. ISSN: 0362-1340. DOI: `10.1145/2103621.2103691`. URL: `http://doi.acm.org/10.1145/2103621.2103691`.

[7] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 1846286913.