A semantics for While in π -calculus

Patricia Peratto
psperatto@adinet.com.uy
Uruguay

November 2007

Abstract

This work presents an implementation in π -calculus of a subset of a **C-like** language called *While*.

We codify booleans, integers and the statements of while : *assignment*, *composition*, **if**, **skip** and **while**.

We study the relationship between precongruences in *While* and π -calculus.

1 Introduction

π -calculus is a calculus for communicating systems in which one can express processes which have changing structure. Mobility is achieved by allowing links to be communicated. There is no distinction between link names, variables and ordinary data values, we call them all names. We have also agents, that are of the following kind: *summation*, *input prefix*, *output prefix*, *silent prefix*, *composition*, *restriction*, *match*, and *defined agents* [1].

While is a very simple imperative language whose semantics was presented in [5]. As basic types we have booleans and integers. To codify the constructors of these types, we designate particular **names** as constants. We also define codifications for operations as by example **or** over booleans, **add** over integers, **greater** from integers to booleans.

Each sentence S of *While* is encoded as $[S]$, a map from names to agents.

The structural operational semantics of *While* give us transitions for pairs $Statement \times State$. We define a precongruence for such pairs in *While*

and we compare it with that induced by the encoding of *While* in *pi*-calculus.

The outline of the paper is as follows: section 2 presents notational conventions of the version of *pi*-calculus we use for the codification. In section 3 we present the semantics of integers, in section 4 the semantics of boolean expressions, in section 5 the codification of statements. In section 6 we prove our results about relations between precongruences in *While* and π -calculus. In section 7 we present conclusions and further work.

2 Notational Conventions

1. When a communication need to carry no parameter, (the parameter does not matter, is not used after), we write:

$\bar{x}.P$ in place of $\bar{x}y.P$

$x.P$ in place of $x(y).P$

2. We shall often omit $.0$ in an agent.
3. We shall often wish to allow input names to determine the course of computation. Thus we write:

$$x : [v_1(u_1) \dots (u_{n_1}) \Rightarrow P_1, \dots, v_m(w_1) \dots (w_{n_m}) \Rightarrow P_m]$$

for

$$x(v) : [[v = v_1] \Rightarrow x(u_1) \dots (u_{n_1})P_1 +, \dots, + \\ [v = v_m] \Rightarrow x(w_1) \dots (w_{n_m})P_m]$$

3 Semantics of Integers

3.1 Syntax

We use a syntactic notation based on BNF. Parenthesis can be used (not indicated in our BNF) to solve ambiguities and uniquely determine the corresponding parse tree.

Definition 3.1. *Abstract syntax for Arithmetic Expressions*

We have the following Syntactic Categories and meta-variables ranging over them

a will range over arithmetic expressions, **AExp**

x will range over arithmetic variables, **AVar**

The meta-variables can be primed or subscripted for example a, a', a_1, a_2 all stand for arithmetic expressions.

$$a ::= 0 \mid succ(a) \mid pred(a) \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$

3.2 Codification of Arithmetic Expressions

We designate three names: *zero*, *succ* and *pred* as constants. Let the set of variables X , be an infinite and co-infinite subset of *Names*. We shall let x, y, z range over X , and u, v, w range over $Names - X$.

We present below the codification of integers and operations over them. We prove in each case the process we define computes the corresponding operation.

Definition 3.2. *zero, succ, pred and the codification of variables*

Any integer n , is represented by the pointed value $[n] \ u \ v$, this is an agent that emits n piecemeal along the link u . In the presence of $[n] \ u \ v$, an agent which possesses or receives the link u , possesses or receives the power to explore the structure of the integer piecemeal by following pointers.

Along the paper we will distinguish two cases: the first name to which is applied the codification is a variable or is a name. When a definition applies in both cases we use as names (and variables) c, d possible primed or subscripted.

$Consume(y, u)$ is used to waste away an integer pointed by y .

$$[0] \ u \ v = \bar{u} \ zero \mid \bar{v}$$

$$[0] \ x \ v = (w)(w.\bar{x} \ zero \mid \bar{v} \mid Consume(x, w))$$

$$[succ(n)] \ u \ v = (w)(\bar{u} \ succ \ \bar{u} \ w \mid [n] \ w \ v)$$

$$[succ(n)] \ x \ v = (w_1, w_2)(w_2.(\bar{x} \ succ \ \bar{x} \ w_1) \mid [n] \ w_1 \ v \mid Consume(x, w_2))$$

$$[pred(n)] \ u \ v = (w)(\bar{u} \ pred \ \bar{u} \ w \mid [n] \ w \ v)$$

$$[pred(n)] \ x \ v = (w_1, w_2)(w_2.(\bar{x} \ pred \ \bar{x} \ w_1) \mid [n] \ w_1 \ v \mid Consume(x, w_2))$$

Variables that occurs in programs always have a value (the value is output by the name corresponding to the variable). All the variables used in a program must be declared, and when we process the declaration we initialize them with value zero, see 5. So we need to waste away the values pointed by a variable before pointing a new value.

When we use a name that does not correspond to a variable, we don't need to consume because has not a previous value.

$$[x] \ x \ u = \mathbf{0}$$

$$[x] \ y \ u = (w)(Consume(y, w) \mid w.Duplicate(x, y, x, u))$$

$$[x] \ v \ u = Duplicate(x, v, x, u)$$

The meaning of a variable in an expression is to output in the name corresponding to the variable the value of the variable in the program.

When we assign to a variable an expression that contains other variable, we copy the value corresponding to the variable in the expression to the variable assigned and we output also this value in the name corresponding to the variable in the expression (is done by Duplicate).

The idea is that when we "read" the value corresponding to a variable (we input in the variable its value), this value is wasted away, but we can have another expression with the variable, so we need its value again, so always that we use a variable we "duplicate" its value.

$$\begin{aligned} \text{Consume}(y, u) = y : [& \text{zero} \Rightarrow \bar{u}, \\ & \text{succ}(y') \Rightarrow \text{Consume}(y', u), \\ & \text{pred}(y') \Rightarrow \text{Consume}(y', u)] \end{aligned}$$

$$\begin{aligned} \text{Duplicate}(c_1, c, c_2, u) = (u_1, d_1, d_2)c_1 : [& \text{zero} \Rightarrow \bar{c}\text{zero} | \bar{c}_2\text{zero} | \bar{u} \\ & \text{succ}(v) \Rightarrow u_1.(\bar{c}\text{succ} \bar{c}d_1 | \\ & \quad \bar{c}_2\text{succ} \bar{c}_2d_2 | \\ & \quad \text{Duplicate}(v, d_1, d_2, u_1), \\ & \text{pred}(v) \Rightarrow u_1.(\bar{c}\text{pred} \bar{c}d_1 | \\ & \quad \bar{c}_2\text{pred} \bar{c}_2d_2 | \\ & \quad \text{Duplicate}(v, d_1, d_2, u_1))] \end{aligned}$$

Theorem 3.3. *Consume(y, u) wastes away the value pointed by y .*

Proof. *Straightforward.*

Theorem 3.4. *Duplicate(c_1, c, y, v) outputs in c and in y the value pointed by c_1 .*

Proof If c_1 points to zero, by the definition c and y output zero, so is proved. To the inductive step, assume $\text{Duplicate}(v, d_1, d_2, u_1)$ outputs in d_1 and in d_2 the value pointed by v . If c_1 is $\text{succ}(v)$ we output in c and in y succ of d_1 and d_2 respectively. If c_1 is $\text{pred}(v)$ we output in c and in y pred of d_1 and d_2 respectively.

Definition 3.5. *Sum, product and subtraction*

First we present the auxiliary function ACopy that copies the value of an integer to a pointer:

$$\begin{aligned} \text{ACopy}(c, d, u) = (u_1, c_2)c : [& \text{zero} \Rightarrow \bar{d}\text{zero} | \bar{u}, \\ & \text{succ}(c_1) \Rightarrow u_1.\bar{d}\text{succ} \bar{d}c_2 | \text{ACopy}(c_1, c_2, u_1) \\ & \text{pred}(c_1) \Rightarrow u_1.\bar{d}\text{pred} \bar{d}c_2 | \text{ACopy}(c_1, c_2, u_1)] \end{aligned}$$

Theorem 3.6. *ACopy(c, d, u) outputs the value pointed by c in d (wasting away the value pointed by c).*

Proof. If the value pointed by c is zero is straightforward. Assume $\text{ACopy}(c_1, c_2, u_1)$ outputs the value pointed by c_1 in c_2 . If c outputs $\text{succ}(c_1)$ we output in d succ of c_2 . Similar if c is $\text{pred}(c_1)$.

below we present the codification of sum, product, subtraction.

$$\begin{aligned}
[a_1 + a_2] xu = & \\
& (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 | \\
& \quad v_1. [a_2] u_2 v_2 | \\
& \quad v_2. Consume(x, v_3) | \\
& \quad v_3. Sum(u_1, u_2, c, v_4) | \\
& \quad v_4. \bar{u})
\end{aligned}$$

$$\begin{aligned}
[a_1 + a_2] vu = & \\
& (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 | \\
& \quad v_1. [a_2] u_2 v_2 | \\
& \quad v_2. Sum(u_1, u_2, c, v_4) | \\
& \quad v_3. \bar{u})
\end{aligned}$$

$$\begin{aligned}
Sum(u_1, u_2, c, v) = (d_2)u_1 : [zero \Rightarrow ACopy(u_2, c, v), \\
\quad succ(d_1) \Rightarrow \bar{c}succ \bar{c}d_2 | Sum(d_1, u_2, d_2, v), \\
\quad pred(d_1) \Rightarrow \bar{c}pred \bar{c}d_2 | Sum(d_1, u_2, d_2, v)]
\end{aligned}$$

Theorem 3.7. *Sum(u_1, u_2, c, v) outputs in c the sum of the values pointed by u_1 and u_2*

Proof *If u_1 points to zero, by the definition of $ACopy$, c will point to u_2 . By structural induction in the other two cases, we assume $Sum(d_1, u_2, d_2, v)$ outputs in d_2 the sum of the values d_1 and u_2 , where u_1 points to $succ(d_1)$ or to $pred(d_1)$. In the first case we output in c $succ$ and d_2 , in the second $pred$ and d_2 .*

$$\begin{aligned}
[a_1 - a_2] xu = & \\
& (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 | \\
& \quad v_1. [a_2] u_2 v_2 | \\
& \quad v_2. Consume(x, v_3) | \\
& \quad v_3. Subtract(u_1, u_2, c, v_4) | \\
& \quad v_4. \bar{u})
\end{aligned}$$

$$\begin{aligned}
[a_1 - a_2] vu = & \\
& (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 | \\
& \quad v_1. [a_2] u_2 v_2 | \\
& \quad v_2. Subtract(u_1, u_2, c, v_4) | \\
& \quad v_3. \bar{u})
\end{aligned}$$

$$\begin{aligned} \text{Subtract}(u_1, u_2, c, v) = (d_2)u_2 : [& \text{zero} \Rightarrow \text{ACopy}(u_1, c, v), \\ & \text{succ}(d_1) \Rightarrow \bar{c}\text{pred } \bar{c}d_2 \mid \text{Subtract}(u_1, d_1, d_2, v), \\ & \text{pred}(d_1) \Rightarrow \bar{c}\text{succ } \bar{c}d_2 \mid \text{Subtract}(u_1, d_1, d_2, v)] \end{aligned}$$

Theorem 3.8. *Subtract(u_1, u_2, c, v) outputs in c the difference of the values pointed by c_1 and c_2*

Proof We assume Subtract(u_1, d_1, d_2, v) outputs in d_2 the difference of u_1 and d_1 . Suppose u_2 is succ(d_1), we apply $u_1 - \text{succ}(d_1) = \text{pred}(u_1 - d_1)$. Suppose u_2 is pred(d_1), we apply $u_1 - \text{pred}(d_1) = \text{succ}(u_1 - d_1)$.

$$\begin{aligned} [a_1 * a_2] xu = \\ (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 \mid \\ v_1. [a_2] u_2 v_2 \mid \\ v_2. \text{Consume}(x, v_3) \mid \\ v_3. \text{Product}(u_1, u_2, c, v_4) \mid \\ v_4. \bar{u}) \end{aligned}$$

$$\begin{aligned} [a_1 * a_2] vu = \\ (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 \mid \\ v_1. [a_2] u_2 v_2 \mid \\ v_2. \text{Product}(u_1, u_2, c, v_4) \mid \\ v_3. \bar{u}) \end{aligned}$$

$$\begin{aligned} \text{Product}(u_1, u_2, c, v) = u_1 : [& \text{zero} \Rightarrow \bar{c}\text{zero} \mid \bar{v}, \\ & \text{succ}(d_1) \Rightarrow \text{Product}(d_1, u_2, d_2, u) \mid u. \text{Sum}(d_1, d_2, c, v), \\ & \text{pred}(d_1) \Rightarrow \text{Product}(d_1, u_2, d_2, u) \mid u. \text{Subtract}(d_2, u_2, c, v)] \end{aligned}$$

Theorem 3.9. *Product(u_1, u_2, c, v) outputs in c the product of the values pointed by u_1 and u_2*

Proof We assume Product(d_1, u_2, d_2, u) outputs in d_2 the product of the values d_1 and u_2 . If u_1 points to succ(d_1) as $\text{succ}(d_1) * u_2 = d_1 + d_1 * u_2$ the proof follows from theorem 3.7. If u_1 points to pred(d_1) as $\text{pred}(d_1) * u_2 = d_1 * u_2 - u_2$ the proof follows from theorem 3.8.

4 Semantics of Booleans

Definition 4.1. Abstract syntax for Boolean Expressions

We have the following Syntactic Categories and meta-variables ranging over them

b will range over boolean expressions, **BExp**

$$b ::= \mathbf{true} \mid \mathbf{false} \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 \geq a_2$$

true and **false** stand for constant truth values. b, b', b_1, b_2 all stand for boolean expressions.

Definition 4.2. true and false

We designate two names *true* and *false* as constants. We think of the agents $[true] \ u$ and $[false] \ u$ as pointed values, with u playing the role of pointer.

$$[true] \ u = \bar{u} \ true$$

$$[false] \ u = \bar{u} \ false$$

Definition 4.3. not, or, and

Similar to *ACopy*, we have *BCopy* to copy booleans.

$$BCopy(u, v) = u : [true \Rightarrow \bar{v}true, \\ false \Rightarrow \bar{v}false]$$

Theorem 4.4. $BCopy(u, v)$ "copies" the value from u to v .

Proof. Straightforward.

$$Not(x, y) = x : [true \Rightarrow \bar{y}false, \\ false \Rightarrow \bar{y}true]$$

$$Or(x, y, z) = x : [true \Rightarrow \bar{z}true, \\ false \Rightarrow BCopy(y, z)]$$

$$And(x, y, z) = x : [true \Rightarrow BCopy(y, z), \\ false \Rightarrow \bar{z}false]$$

$$[\neg b] w = (u)([b] u \mid Not(u, w))$$

$$[b_1 \vee b_2] w = (u, v)([b_1] u \mid [b_2] v \mid Or(u, v, w))$$

$$[b_1 \wedge b_2] w = (u, v)([b_1] u \mid [b_2] v \mid And(u, v, w))$$

Theorem 4.5. $\neg, \vee$ and $\wedge$ compute the corresponding boolean operations over the codification of booleans in π -calculus.

Proof. Straightforward.

Definition 4.6. *Equal, Greater, Greater or equal*

We need the following auxiliary functions to define *Equal*

1. *NumberOfSuccPred*(x, x_1, x_2): outputs in x_1 and x_2 two natural numbers that equal the number of applications of *succ* and *pred* in x .
2. *Iszero*(y, z) output *true* if y is equal zero, or in other case, if the number of applications of *succ* and *pred* in y are equal.
3. *Equal2*(x, y, z) computes equality applied to canonical forms (see below).
4. *SuccEqual*(x, y, z).
5. *PredEqual*(x, y, z).
6. *Reduce*(x, z) reduces the integer pointed by x eliminating applications of *succ* followed by applications of *pred*, or applications of *pred* followed by applications of *succ*.
7. *CForm* calls *Reduce*(x, z) so many times as applications of *succ* and *pred* appear in x . This gives as result an integer defined as zero, or with applications only of successor if is positive, or with applications only of predecessor if is negative.
8. *CanonicalForm*(x, z) calls *CForm* with x and a copy of x . One is reduced, the other controls the number of applications of *Reduce* that are applied.

The definitions are the following:

$$\begin{aligned}
 \text{NumberOfSuccPred}(u_1, u_2, u_3, u) = \\
 u_1 : [\text{zero} \Rightarrow \bar{u}_2 \text{zero} | \bar{u}_3 \text{zero} | \bar{u} \\
 \text{succ}(v) \Rightarrow (w_1, w_2, w_3) | \\
 \text{NumberOfSuccPred}(v, w_1, w_2, w_3) | \\
 w_3. \bar{u}_2 \text{succ} \bar{u}_2 w_1 | \\
 \text{ACopy}(w_2, u_3, u), \\
 \text{pred}(v) \Rightarrow (w_1, w_2, w_3) \\
 \text{NumberOfSuccPred}(v, w_1, w_2, w_3) | \\
 w_3. \bar{u}_3 \text{succ} \bar{u}_3 w_2 | \\
 \text{ACopy}(w_1, u_2, u)]
 \end{aligned}$$

Theorem 4.7. *NumberOfSuccPred(u_1, u_2, u_3, u) outputs in u_2 the number of applications of succ and in u_3 the number of applications of pred that occur in the integer pointed by u_1 .*

Proof. *By structural induction.*

If u_1 is zero we output zero both in u_2 and u_3 . Assume by inductive hypothesis *NumberOfSuccPred*(v, w_1, w_2, w_3) outputs in w_1 the number of applications of succ and in w_2 the number of applications of pred that occur in the integer pointed by v . If u_1 is *succ*(v) we output in u_2 *succ* and w_1 (one occurrence more of succ than in v) and we copy w_1 to u_2 (equal number of occurrences of pred). The case when u_1 is *pred*(v) is similar.

$$\begin{aligned}
 \text{Iszero}(v, w) = v : [\text{zero} \Rightarrow \bar{w} \text{true}, \\
 \text{succ}(w_1) \Rightarrow (v_1, v_2, u_1) (\bar{v} \text{succ} \bar{v} w_1 | \\
 \text{NumberOfSuccPred}(v, v_1, v_2, u_1) | \\
 u_1. \text{Equal}(v_1, v_2, w)), \\
 \text{pred}(w_1) \Rightarrow (v_1, v_2, u_1) (\bar{v} \text{pred} \bar{v} w_1 | \\
 \text{NumberOfSuccPred}(v, v_1, v_2, u_1) | \\
 u_1. \text{Equal}(v_1, v_2, w))]
 \end{aligned}$$

An alternative definition of *Iszero* is applying reduction to canonical form (see below):

$$\begin{aligned}
 \text{Iszero}(v, w) = \text{CanonicalForm}(v, u, u_1) | u : [\text{zero} \Rightarrow \bar{w} \text{true}, \\
 \text{succ}(w_1) \Rightarrow \bar{w} \text{false}, \\
 \text{pred}(w_1) \Rightarrow \bar{w} \text{false}]
 \end{aligned}$$

Theorem 4.8. *Iszero(v, w) outputs in w true if v is equal to zero and false otherwise.*

Proof. *If the number of applications of succ and pred in v is the same, v is equal to zero.*

$$\begin{aligned} \text{Equal2}(u, v, w) = u : [& \text{zero} \Rightarrow \text{Iszero}(v, w), \\ & \text{succ}(w_1) \Rightarrow \text{SuccEqual}(w_1, v, w), \\ & \text{pred}(w_1) \Rightarrow \text{PredEqual}(w_1, v, w)] \end{aligned}$$

Theorem 4.9. *Equal2(u, v, w) outputs true in w if the integers pointed by u and v are equal. Is applied to integers in canonical form (are equal to zero or have the constructor succ applied a finite number of times and no application of pred or have the constructor pred applied a finite number of times and no application of succ).*

Proof. *straightforward.*

$$\begin{aligned} \text{SuccEqual}(u, v, w) = v : [& \text{zero} \Rightarrow \bar{w}\text{false}, \\ & \text{pred}(w_1) \Rightarrow \bar{z}\text{false}, \\ & \text{succ}(w_1) \Rightarrow \text{Equal2}(u, w_1, z)] \end{aligned}$$

Theorem 4.10. *SuccEqual(u, v, w) outputs in w true if the number of occurrences of succ in v is one more than in u .*

Proof. *Straightforward.*

$$\begin{aligned} \text{PredEqual}(u, v, w) = v : [& \text{zero} \Rightarrow \bar{w}\text{false}, \\ & \text{succ}(w_1) \Rightarrow \bar{w}\text{false}, \\ & \text{pred}(w_1) \Rightarrow \text{Equal2}(u, w_1, w)] \end{aligned}$$

Theorem 4.11. *PredEqual(u, v, w) outputs in w true if the number of occurrences of pred in v is one more than in u .*

Proof. *Straightforward.*

$$\begin{aligned} \text{Reduce}(u, v, u_1) = u : [& \text{zero} \Rightarrow \bar{v}\text{zero} | \bar{u}_1, \\ & \text{succ}(w) \Rightarrow w : [\text{zero} \Rightarrow \bar{v}\text{succ } \bar{v}\text{zero}, \\ & \quad \text{succ}(w_1) \Rightarrow \bar{w}\text{succ } \bar{w}w_1 | \text{Reduce}(w, v, u_1), \\ & \quad \text{pred}(w_1) \Rightarrow \text{Reduce}(w_1, v, u_1)] \\ & \text{pred}(w) \Rightarrow w : [\text{zero} \Rightarrow \bar{v}\text{succ } \bar{v}\text{zero}, \end{aligned}$$

$$\begin{aligned} pred(w_1) &\Rightarrow \overline{w}pred \overline{w}w_1 \mid Reduce(w, v, u_1), \\ succ(w_1) &\Rightarrow Reduce(w_1, v, u_1) \mid \end{aligned}$$

Theorem 4.12. *Reduce(u, v, u_1) deletes from the integer pointed by u occurrences of $pred$ followed by occurrences of $succ$ and occurrences of $succ$ followed by occurrences of $pred$.*

Proof *Straightforward.*

$$\begin{aligned} CForm(u_1, u_2, u_3, u_4) = u_2 : [&zero \Rightarrow ACopy(u_1, u_3, u_4), \\ &succ(w) \Rightarrow (v_1)(Reduce(u_1, v, v_1) \mid v_1.CForm(v, w, u_3, u_4)) \\ &pred(w) \Rightarrow (v_1)(Reduce(u_1, v, v_1) \mid v_1.CForm(v, w, u_3, u_4))] \end{aligned}$$

Theorem 4.13. *CForm(u_1, u_2, u_3, u_4) outputs in u_3 the canonical form of u_1 . To this end, applies Reduce to u_1 so many times as applications of $succ$ and $pred$ there are in u_2 (is called with u_2 a copy of u_1 , this number of applications is greater than the necessary).*

Proof *Straightforward.*

$$CanonicalForm(u, v, w) = (y, u_1)(Duplicate(u, u_1, u, u_2) \mid u_2.CForm(u, u_1, v, w))$$

Theorem 4.14. *CanonicalForm(u, v, w) outputs in v the canonical form of u . To this end, applies CForm to u .*

Proof *Straightforward.*

$$\begin{aligned} Equal(u, v, w) = &(w_1, w_2, w_3, w_4, u_1, u_2, u_3, u_4, u_5, u_6, v_1, v_2, v_3, v_4) \\ &CanonicalForm(u, v_1, u_1) \mid \\ &u_1.CanonicalForm(v, v_2, u_2) \mid \\ &u_2.Equal2(v_1, v_2, w) \end{aligned}$$

Theorem 4.15. *Equal(u, v, w) outputs in w true if the canonical forms of u and v are equal.*

Proof *Straightforward.*

$$\begin{aligned} [a_1 = a_2] v = & \\ &(u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 \mid \\ &v_1.[a_2] u_2 v_2 \mid \\ &v_2.Equal(u_1, u_2, v) \end{aligned}$$

We need the following auxiliary functions to define Greater

1. $Greater2(x, y, z)$.
2. $CanonicalForm(x, z)$.

$$\begin{aligned}
 Greater2(w_1, w_2, w_3) = w_1 : [& zero \Rightarrow w_2 : [zero \Rightarrow \overline{w_3} false, \\
 & succ(u) \Rightarrow \overline{w_3} false, \\
 & pred(u) \Rightarrow \overline{w_3} true,] \\
 succ(u_1) \Rightarrow w_2 : [& zero \Rightarrow \overline{w_3} true, \\
 & succ(u_2 \Rightarrow Greater2(u_1, u_2, w_3), \\
 & pred(u_2 \Rightarrow \overline{w_3} true)] \\
 pred(u_1) \Rightarrow w_2 : [& zero \Rightarrow \overline{w_3} false, \\
 & succ(u_2 \Rightarrow \overline{w_3} false, \\
 & pred(u_2 \Rightarrow Greater2(u_1, u_2, w_3),]
 \end{aligned}$$

Theorem 4.16. $Greater2(w_1, w_2, w_3)$ outputs true in w_3 if the integer pointed by w_1 is greater than the pointed by w_2 . Is applied to integers in canonical form.

Proof. by cases.

$$\begin{aligned}
 Greater(u, v, w) = & (w_1, w_2, w_3, w_4, u_1, u_2, u_3, u_4, u_5, u_6, v_1, v_2, v_3, v_4) \\
 & CanonicalForm(u, v_1, u_1) | \\
 & u_1.CanonicalForm(v, v_2, u_2) | \\
 & u_2.Greater2(v_1, v_2, w)
 \end{aligned}$$

Theorem 4.17. $Greater(u, v, w)$ outputs in w true if the canonical form of u is smaller than the one of v .

Proof Straightforward.

$$\begin{aligned}
 [a_1 > a_2] v = & \\
 (u_1, u_2, v_1, v_2, v_3, v_4) & ([a_1] u_1 v_1 | \\
 & v_1. [a_2] u_2 v_2 | \\
 & v_2.Greater(u_1, u_2, v)
 \end{aligned}$$

To define greater or equal, we use : greater, equal and or.

$$\begin{aligned}
& [a_1 \geq a_2] v = \\
& (u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4, w_1, w_2) \\
& ([a_1] u_1 v_1 \mid \\
& v_1. [a_2] u_2 v_2 \mid \\
& v_2. Duplicate(u_1, u_3, u_1, v_3) \mid \\
& v_3. Duplicate(u_2, u_4, u_2, v_4) \mid \\
& v_2. (Greater(u_1, u_2, w_1) \mid \\
& Equal(u_3, u_4, w_2) \mid \\
& Or(w_1, w_2, v))
\end{aligned}$$

5 Codification of Statements

Definition 5.1. *Abstract syntax for statements, declarations and programs*

We have the following Syntactic Categories and meta-variables ranging over them

*S will range over statements, **Stm***

*D will range over declarations, **Decl***

*P will range over programs, **Prog***

$$S ::= x := a \mid \mathbf{skip} \mid S_1; S_2 \mid \mathbf{if} \ b \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \mid \mathbf{while} \ b \ \mathbf{do} \ S$$

$$D ::= \mathbf{var} \ x; D \mid \epsilon$$

ϵ stands for the empty declaration.

$$P ::= D; S$$

a program is a declaration and an statement.

To simplify our semantics we impose as constraint for a program to have declarations for all the variables it uses.

Definition 5.2. *Codification*

1. declarations

$$[\mathbf{var} \ x, D] = \bar{x}zero \mid [D]$$

$$[\epsilon] = \mathbf{0}$$

2. programs

$$[D; S] \ u = [D] \mid [S] \ u$$

3. statements

(a) skip

$$[skip] \ u = \bar{u}.0$$

(b) assignation

$$[x := a] \ u = [a] \ x \ u$$

(c) if

$$[if \ b \ then \ S_1 \ else \ S_2] \ u = (w)([b] \ w \mid \\ w : [true \Rightarrow [S_1] \ u \\ false \Rightarrow [S_2] \ u])$$

(d) while

$$[while \ b \ do \ S] \ u = (w)([b] \ w \mid \\ w : [true \Rightarrow [S; while \ b \ do \ S] \ u \\ false \Rightarrow [skip] \ u])$$

(e) composition

$$[S_1; S_2] \ u = ([S_1] \ u \mid (v)u. [S_2] \ v)$$

The codification is straightforward, what deserves comment is assignation and for this we have to see the semantics of variables. When in an expression appears a variable as for example in $x := y$ we get the expression $[y] \ x \ v$ that reduces to $(w)(Consume(x, w) \mid Duplicate(y, x, y, v))$. With *Consume* we waste away any value pointed by x , with *Duplicate* we point with x and y the integer that we have had in y . The reason to duplicate is that the variable y can appear again in an arithmetic expression, the same or other, and we need to have its value pointed by it. Always that we will return in a variable a value, we consume before the value it has had, (see the codifications of sum, product and subtraction).

6 Observation precongruences

Definition 6.1. The relation $P \rightarrow Q$ is identical to $\rightarrow^\tau$ and this last relation is the defined in [2] table 2. P is *r-determinate* if whenever $P \rightarrow^* Q$ and also $Q \rightarrow Q_1$ and $Q \rightarrow Q_2$ with $Q_1 \neq Q_2$, exists Q_3 such that $Q_1 \rightarrow Q_3$ and $Q_2 \rightarrow Q_3$. Also, P converges to Q , written $P \downarrow Q$ if $P \rightarrow^* Q \rightarrow$. We write $P \downarrow$ to mean P converges to some Q , and $P \uparrow$ otherwise.

Definition 6.2. The relation $\langle S, s \rangle \Rightarrow \langle S', s' \rangle$ or $\langle S, s \rangle \Rightarrow s'$ is the given by the structural operational semantics of While [5].

Definition 6.3. Let the relation $R \subseteq (Stm \times State \times Proc) \cup (State \times Proc)$ contain all the pairs (L, P) such that:

1. $L \equiv \langle S, s \rangle$ and $P \equiv (u)([S] u \mid [s])$
2. $L \equiv s$ and $P \equiv (u)(\bar{u} \mid [s])$

where

$$[s(x := A(a)s)] = ([s] \mid [a] x)$$

$$[a] x = (u)([a] x u \mid u) \text{ \textbf{Note:} } [s] \downarrow$$

Lemma 6.4. For any $(L, P) \in R$, P is *r-determinate* and the following condition hold:

for some $(L', P') \in R$ $L \Rightarrow L'$ and $P \rightarrow^* P'$

Proof. by structural induction on L .

1. $L \equiv \langle [x := a], s \rangle$, $L' \equiv s(x := A(a)s)$, $P \equiv (u)([x := a] u \mid [s]) \equiv (u)([a] x u \mid [s]) \equiv (u)([a] x \bar{u} \mid [s]) \equiv (u)(s[x := a] \mid \bar{u}) \equiv P'$
2. $L \equiv \langle skip, s \rangle$, $L' \equiv s$, $L \Rightarrow L'$ $P \equiv (u)([skip] u \mid [s]) \equiv (u)(\bar{u} \mid [s]) \equiv P'$
3. $L \equiv \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle$ $B[b, s] = \text{true}$. $L' \equiv \langle S_1, s \rangle$, $P' \equiv (v)([S_1] u, [u. [s] v])$.
4. $L \equiv \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle$ $B[b, s] = \text{false}$. Similar to 3.
5. $L \equiv \langle \text{while } b \text{ do } S, s \rangle$ follows from composition, if and skip.

6. $L \equiv < S_1; S_2, s >, P \equiv (u)([S_1; S_2] u | [s]) \equiv (u)([S_1] u | (v)(u.[S_2] v | [s])$
- (a) assume $< S_1, s > \Rightarrow < S'_1, s' >$. Let $L_1 \equiv < S_1, s >, P_1 \equiv (u)([S_1] u | [s])$, $L'_1 \equiv < S'_1, s' >, P'_1 \equiv (u)([S'_1] u | [s'])$ by hypothesis, $L_1 \Rightarrow L'_1$ $P_1 \rightarrow^* P'_1$ then $P \rightarrow (u)([S'_1] u | (v)(u.[S_2] v | [s'])) \equiv (u)([S'_1; S_2] u | [s'])$.
- (b) assume $< S_1, s > \Rightarrow s'$. Let $L_1 \equiv < S_1, s >, P_1 \equiv (u)([S_1] u | [s])$, $L'_1 \equiv s'$, $P'_1 \equiv (u)(\bar{u} | [s'])$ by hypothesis, $L_1 \Rightarrow L'_1$ $P_1 \rightarrow^* P'_1$ then $P \rightarrow (u)([S_1; S_2] u | [s]) \equiv (u)([S_1] u | (v)(u.[S_2] v | [s])) \rightarrow (u, v) [S_2] v | [s'] \equiv (v) [S_2] v | [s']$.

Theorem 6.5. For all $L \in \text{Stm} \times \text{State}$, $L \equiv < S, s > P \equiv (u)([S] u | [s])$ is r -determinate and one of the following conditions hold:

1. $L \downarrow s'$ and $P \rightarrow^* (u)(\bar{u} | [s'])$.
2. $L \uparrow$ and $P \uparrow$.

Proof. Immediate by iterating the preceding lemma.

Definition 6.6. P is determinate if is r -determinate and also, whenever $P \rightarrow^* Q \rightarrow^\alpha$, for $\alpha \neq \tau$, then $Q \nrightarrow$.

Lemma 6.7. $P \equiv (u)([S] u | [s])$ is determinate.

Proof. By inspection of the proof of 6.4

Definition 6.8. The action relations $\rightarrow^\alpha$ are the smallest that satisfy the rules on [2] table 2.

$\Rightarrow^\alpha$ is by definition $\rightarrow^* \rightarrow^\alpha$.

The weakest reasonable preorder and precongruence are as follows:

Definition 6.9. $P \sqsubseteq Q$ if, for all $\alpha \neq \tau$, $P \Rightarrow^\alpha$ implies $Q \Rightarrow^\alpha$.

$P \sqsubseteq Q$ if, for all contexts $C[_]$, $C[P] \sqsubseteq C[Q]$.

Definition 6.10. $< S, s >$ converges to s' (notation $< S, s > \downarrow s'$) if $< S, s > \Rightarrow s'$. $< S, s > \downarrow$ means $< S, s >$ converges for some s' .

$S_1 \sqsubseteq S_2$ if for all states s $< S_1, s > \downarrow$ implies $< S_2, s > \downarrow$.

Theorem 6.11. Let $P_i = (u)([S_i] u | [s])$. Then $P_1 \sqsubseteq P_2$ implies $S_1 \sqsubseteq S_2$.

Proof. Assume $< S_1, s > \downarrow s'$, let $P_1 = C([S_1] u)$. By theorem 6.5 $C([S_1] u) \rightarrow^* (u)(\bar{u} | [s'])$ and $[s'] \downarrow$.

By hypothesis $P_1 \equiv C([S_1] u) \Rightarrow^\alpha$ implies $P_2 \equiv C([S_2] u) \Rightarrow^\alpha$.

Since P_2 is determinate $P_2 \rightarrow^* P \rightarrow^\alpha$, $\alpha \neq \tau$ implies $P \nrightarrow$. So, $P_2 \downarrow$ and then exists an state s'' such that $< S_2, s > \downarrow s''$.

Theorem 6.12. *Let $P_i = (v)([S_i]u|[s]v)$. Then $S_1 \sqsubseteq S_2$ does not imply $P_1 \sqsubseteq P_2$.*

Proof. *Consider $S_1 \equiv (x := 0)$ and $S_2 \equiv (x := 1)$. Then $P_1 \Rightarrow^{\bar{x}zero}$ and $P_2 \Rightarrow^{\bar{x}succ} 0$.*

7 Conclusions

We have codified the language *While* in pi-calculus. As can be seen the codifications of integers and booleans are not trivial.

In the codifications we have defined orders of execution between agents by means of communication without parameters.

Our idea of codifying *While* follows from the codification of lambda calculus in π -calculus presented in [3]. Nevertheless in what agents concerns we are based on the presentation of π -calculus of [1, 2] and not in the one of [3].

We hope that the results on this paper throw some light on the semantics of the language *While*.

As further work can result interesting to codify other languages as by example the logic programming language Prolog in π -calculus, or the language of logic.

In what respects to the last idea, we are working on a tautology checker in *pi*-calculus.

References

- [1] Robin Milner, Joachim Parrow, David Walker. A Calculus of Mobile Processes, Part I, Report ECS-LFCS-89-85, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.
- [2] Robin Milner, Joachim Parrow, David Walker. A Calculus of Mobile Processes, Part II, Report ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.
- [3] Robin Milner, *Functions as Processes*, Rapports de Recherche Number 1154, INRIA, February 1990.

- [4] Robin Milner, *The Polyadic π -Calculus: a Tutorial*, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, October 1991.
- [5] H.R.Nielson and F.Nielson. *Semantics with Applications, A Formal Introduction for Computer Science*, John Wiley & Sons, 1992.
- [6] Gordon Plotkin. *A structural Approach to operational semantics*, Technical Report DAIMI FN-19, Aarhus University, 1981.