

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 .

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

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

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

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

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

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

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.

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

$$[x] \ y \ u = (w)(\text{Consume}(y, w) | w.\text{Duplicate}(x, y, x, u))$$

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

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(c₁, c, y, v) outputs in c and in y the value pointed by c₁.*

Proof If c₁ points to zero, by the definition c and y output zero, so is proved. To the inductive step, assume Duplicate(v, d₁, d₂, u₁) outputs in d₁ and in d₂ the value pointed by v. If c₁ is succ(v) we output in c and in y succ of d₁ and d₂ respectively. If c₁ is pred(v) we output in c and in y pred of d₁ and d₂ 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 ACopy(c₁, c₂, u₁) outputs the value pointed by c₁ in c₂. If c outputs succ(c₁) we output in d succ of c₂. Similar if c is pred(c₁).

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

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

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

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

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 $\text{Sum}(d_1, u_2, d_2, v)$ outputs in d_2 the sum of the values d_1 and u_2 , where u_1 points to $\text{succ}(d_1)$ or to $\text{pred}(d_1)$. In the first case we output in c succ and d_2 , in the second pred and d_2 .*

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

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

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

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 the values u_1 and d_1 . Suppose u_2 is $\text{succ}(d_1)$, we apply $u_1 - \text{succ}(d_1) = \text{pred}(u_1 - d_1)$. Suppose u_2 is $\text{pred}(d_1)$, we apply $u_1 - \text{pred}(d_1) = \text{succ}(u_1 - d_1)$.

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

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

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

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 $\text{succ}(d_1)$ 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 $\text{pred}(d_1)$ 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.

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

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

Definition 4.3. *not, or, and*

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

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

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

Proof. *Straightforward.*

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

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

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

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

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

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

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:

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

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.

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

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.*

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

