A semantics for While in $\pi$-calculus

Patricia Peratto
psperatto@adinet.com.uy
Uruguay
November 2007

Abstract

This work presents an implementation in $\pi$-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 $\pi$-calculus.

1 Introduction

$\pi$-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

1
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 $\pi$-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:

$$\pi.P \text{ in place of } \pi y.P$$

$$x.P \text{ in place of } x(y).P$$

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:

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

for

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

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

\[ a \text{ will range over arithmetic expressions, } \text{AExp} \]

\[ x \text{ will range over arithmetic variables, } \text{AVar} \]

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

\[ a ::= 0 \mid \text{succ}(a) \mid \text{pred}(a) \mid x \mid a_1 + a_2 \mid a_1 \ast a_2 \mid a_1 - a_2 \]

### 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 \(\text{Name}\). We shall let \(x, y, z\) range over \(X\), and \(u, v, w\) range over \(\text{Name} - X\).

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.

\(\text{Consume}(y, u)\) is used to waste away an integer pointed by \(y\).
\[ 0 | u \, v = \overline{u} \text{ zero} | \overline{v} \]

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

\[ \text{succ}(n) | u \, v = (w)(\overline{u} \text{ succ} \, \overline{w} | [n] \, w \, v) \]

\[ \text{pred}(n) | x \, v = (w_1, w_2)(w_2, (\overline{x} \text{ succ} \, \overline{x} \, w_1) | [n] \, w_1 \, v | \text{Consume}(x, w_2)) \]

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

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.

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

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

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

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.
Consume\((y, u) = y : [\text{zero} \Rightarrow \pi, \text{succ}(y') \Rightarrow \text{Consume}(y', u), \text{pred}(y') \Rightarrow \text{Consume}(y', u)]\)

Duplicate\((c_1, c, c_2, u) = (u_1, d_1, d_2) : [\text{zero} \Rightarrow \pi \text{zero} \mid \pi, \text{succ}(v) \Rightarrow u_1, \text{pred}(v) \Rightarrow u_1]\)

\[\text{Duplicate}(v, d_1, d_2) \Rightarrow \text{Consume}(v, d_1, d_2)]\]

\[\text{Duplicate}(v, d_1, d_2) \Rightarrow \text{Consume}(v, d_1, d_2)]\]

**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 Duplicate\((v, d_1, d_2, u_1)\) outputs in \(d_1\) and \(d_2\) the value pointed by \(v\). If \(c_1\) is succ\((v)\) we output in \(c\) and in \(y\) succ of \(d_1\) and \(d_2\) respectively. If \(c_1\) is 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:

ACopy\((c, d, u) = (u_1, c_2) : [\text{zero} \Rightarrow \pi \text{zero} \mid \pi, \text{succ}(c_1) \Rightarrow u_1, \text{pred}(c_1) \Rightarrow u_1]\)

\[\text{ACopy}(c_1, c_2, u_1) \Rightarrow \text{Consume}(c_1, c_2, u_1)]\]

\[\text{ACopy}(c_1, c_2, u_1) \Rightarrow \text{Consume}(c_1, c_2, u_1)]\]

**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\((c_1, c_2, u_1)\) outputs the value pointed by \(c_1\) in \(c_2\). If \(c\) outputs succ\((c_1)\) we output in \(d\) succ of \(c_2\). Similar if \(c\) is pred\((c_1)\).

below we present the codification of sum, product, subtraction.
\[ [a_1 + a_2 ] xu = (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 | \]
\[ v_1. [a_2] u_2 v_2 | \]
\[ v_2. \text{Consume}(x, v_3) | \]
\[ v_3. \text{Sum}(u_1, u_2, c, v_4) | \]
\[ v_4. \tau) \]

\[ [a_1 + a_2 ] vu = (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 | \]
\[ v_1. [a_2] u_2 v_2 | \]
\[ v_2. \text{Sum}(u_1, u_2, c, v_4) | \]
\[ v_3. \tau) \]

\[ \text{Sum}(u_1, u_2, c, v) = (d_2)u_1 : \{ \text{zero} \Rightarrow A\text{Copy}(u_2, c, v), \]
\[ \text{succ}(d_1) \Rightarrow \tau \text{succ } \tau d_2 | \text{Sum}(d_1, u_2, d_2, v), \]
\[ \text{pred}(d_1) \Rightarrow \tau \text{pred } \tau d_2 | \text{Sum}(d_1, u_2, d_2, v) \} \]

**Theorem 3.7.** \( \text{Sum}(u_1, u_2, c, v) \) 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 \( A\text{Copy} \), \( c \) 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 \( c \) \( \text{succ} \) and \( d_2 \), in the second \( \text{pred} \) and \( d_2 \).

\[ [a_1 - a_2 ] xu = (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 | \]
\[ v_1. [a_2] u_2 v_2 | \]
\[ v_2. \text{Consume}(x, v_3) | \]
\[ v_3. \text{Subtract}(u_1, u_2, c, v_4) | \]
\[ v_4. \tau) \]

\[ [a_1 - a_2 ] vu = (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 | \]
\[ v_1. [a_2] u_2 v_2 | \]
\[ v_2. \text{Subtract}(u_1, u_2, c, v_4) | \]
\[ v_3. \tau) \]
Suppose \( u \) and \( u_1 \). Subtract \( u \) and \( u_1 \). We assume 

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

**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 \( \text{succ}(d_1) \), we apply \( u_1 - \text{succ}(d_1) = \text{pred}(u_1 - d_1) \). Suppose \( u_2 \) is \( \text{pred}(d_1) \), we apply \( u_1 - \text{pred}(d_1) = \text{succ}(u_1 - d_1) \).

\[
\begin{align*}
[a_1 * a_2] xu &= (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 | \\
&v_1, [a_2] u_2 v_2 | \\
&v_2.\text{Consume}(x, v_3) | \\
&v_3.\text{Product}(u_1, u_2, c, v_4) | \\
&v_4, \overline{\pi})
\end{align*}
\]

\[
\begin{align*}
[a_1 * a_2] vu &= (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 | \\
&v_1, [a_2] u_2 v_2 | \\
&v_2.\text{Product}(u_1, u_2, c, v_4) | \\
&v_3, \overline{\pi})
\end{align*}
\]

**Product** \( (u_1, u_2, c, v) = u_1 : [ \text{zero} \Rightarrow \overline{\text{zero}} \overline{\pi}, \\
\text{succ}(d_1) \Rightarrow \text{Product}(d_1, u_2, d_2, u) | u.\text{Sum}(d_1, d_2, c, v), \\
\text{pred}(d_1) \Rightarrow \text{Product}(d_1, u_2, d_2, u) | u.\text{Subtract}(d_2, u_2, c, v) ]
\]

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

**Proof** We assume \( \text{Product}(d_1, u_2, d_2, u) \) 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.
Semantics of Boolean Expressions

**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, $\mathbf{BExp}$

\[
b ::= \text{true} \mid \text{false} \mid \neg b \mid b_1 \lor b_2 \mid b_1 \land b_2 \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 \geq a_2
\]

**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 $[\text{true}] \, u$ and $[\text{false}] \, u$ as pointed values, with $u$ playing the role of pointer.

\[
[\text{true}] \, u = !u \text{true}
\]

\[
[\text{false}] \, u = !u \text{false}
\]

**Definition 4.3. not, or, and**

Similar to $\text{ACopy}$, we have $\text{BCopy}$ to copy booleans.

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

**Theorem 4.4.** $\text{BCopy}(u, v)$ "copies" the value from $u$ to $v$.

**Proof.** Straightforward.

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

\[
\text{Or}(x, y, z) = x : [\text{true} \Rightarrow z \text{true}, \text{false} \Rightarrow \text{BCopy}(y, z)]
\]
\[ And(x, y, z) = x : \begin{cases} \text{true} \Rightarrow BCopy(y, z), \\ \text{false} \Rightarrow zfalse \end{cases} \]

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

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

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

**Theorem 4.5.** \(\neg, \lor\) and \(\land\) compute the corresponding boolean operations over the codification of booleans in \(\pi\)-calculus.

**Proof.** Straightforward.

**Definition 4.6.** Equal, Greater, Greater or equal
We need the following auxiliary functions to define Equal

1. \(\text{NumberOfSuccPred}(x, x_1, x_2)\): outputs in \(x_1\) and \(x_2\) two natural numbers that equal the number of applications of succ and pred in \(x\).
2. \(\text{Iszero}(y, z)\) 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. \(\text{Equal2}(x, y, z)\) computes equality applied to canonical forms (see below).
4. \(\text{SuccEqual}(x, y, z)\).
5. \(\text{PredEqual}(x, y, z)\).
6. \(\text{Reduce}(x, z)\) 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. \(\text{CForm}\) calls \(\text{Reduce}(x, z)\) 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. \(\text{CanonicalForm}(x, z)\) calls \(\text{CForm}\) with \(x\) and a copy of \(x\). One is reduced, the other controls the number of applications of \(\text{Reduce}\) that are applied.
The definitions are the following:

\[
\text{NumberofSuccPred}(u_1, u_2, u_3, u) =
\begin{align*}
u_1 : & \begin{cases}
zero & \Rightarrow w_1, w_2, w_3 \\
succ(v) & \Rightarrow (v_1, v_2, w_3) \\
\text{NumberofSuccPred}(v, w_1, w_2, w_3) \\
w_3.w_2.w_1 \\
\text{ACopy}(w_2, u_3, u), \\
pred(v) & \Rightarrow (v_1, v_2, w_3) \\
\text{NumberofSuccPred}(v, w_1, w_2, w_3) \\
w_3.w_2.w_1 \\
\text{ACopy}(w_1, u_2, u)
\end{cases}
\end{align*}
\]

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

**Proof.** By structural induction.

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

\[
\text{Iszero}(v, w) = v : \begin{cases}
\text{zero} & \Rightarrow \text{true}, \\
succ(w_1) & \Rightarrow (v_1, v_2, u_1)(v_1.w_1) \\
\text{NumberofSuccPred}(v, v_1, v_2, u_1) \\
u_1.\text{Equal}(v_1, v_2, w)), \\
pred(w_1) & \Rightarrow (v_1, v_2, u_1)(v_1.w_1) \\
\text{NumberofSuccPred}(v, v_1, v_2, u_1) \\
u_1.\text{Equal}(v_1, v_2, w))
\end{cases}
\]

An alternative definition of \(\text{Iszero}\) is applying reduction to canonical form (see below):

\[
\text{Iszero}(v, w) = \text{CanonicalForm}(v, u, u_1) \mid u : \begin{cases}
\text{zero} & \Rightarrow \text{true}, \\
succ(w_1) & \Rightarrow \text{false}, \\
pred(w_1) & \Rightarrow \text{false}
\end{cases}
\]
Theorem 4.8. $\text{Iszero}(v, w)$ outputs in $w$ true if $v$ is equal to zero and false otherwise.

Proof. If the number of applications of $\text{succ}$ and $\text{pred}$ in $v$ is the same, $v$ is equal to zero.

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

Theorem 4.9. $\text{Equal2}(u, v, w)$ 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 $\text{succ}$ applied a finite number of times and no application of $\text{pred}$ or have the constructor $\text{pred}$ applied a finite number of times and no application of $\text{succ}$).

Proof. Straightforward.

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

Theorem 4.10. $\text{SuccEqual}(u, v, w)$ outputs in $w$ true if the number of occurrences of $\text{succ}$ in $v$ is one more than in $u$.

Proof. Straightforward.

$\text{PredEqual}(u, v, w) = v : [\text{zero} \Rightarrow \overline{w} \text{false},$
$succ(w_1) \Rightarrow \overline{w} \text{false},$
$pred(w_1) \Rightarrow \text{Equal2}(u, w_1, w) ]$

Theorem 4.11. $\text{PredEqual}(u, v, w)$ outputs in $w$ true if the number of occurrences of $\text{pred}$ in $v$ is one more than in $u$.

Proof. Straightforward.

$\text{Reduce}(u, v, u_1) = u : [\text{zero} \Rightarrow \overline{v} \text{zero},$
$succ(w) \Rightarrow w : [\text{zero} \Rightarrow \overline{v} \text{succ} \overline{v} \text{zero},$
$succ(w_1) \Rightarrow \overline{w} \text{succ} \overline{w} w_1 | \text{Reduce}(w, v, u_1),$
$pred(w_1) \Rightarrow \text{Reduce}(w_1, v, u_1) ]$
$pred(w) \Rightarrow w : [\text{zero} \Rightarrow \overline{v} \text{succ} \overline{v} \text{zero},$
$\ldots$]
\[ \text{pred}(w_1) \Rightarrow \text{pred} \; w \; v_1 \; | \; \text{Reduce}(w, v, u_1), \]
\[ \text{succ}(w_1) \Rightarrow \text{Reduce}(w_1, v, u_1) \] 

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

**Proof** Straightforward.

\[ CForm(u_1, u_2, u_3, u_4) = u_2 : [ \text{zero} \Rightarrow \text{ACopy}(u_1, u_3, u_4), \]
\[ \text{succ}(w) \Rightarrow (v_1)(\text{Reduce}(u_1, v, v_1) \; | \; v_1.CForm(v, w, u_3, u_4)) \]
\[ \text{pred}(w) \Rightarrow (v_1)(\text{Reduce}(u_1, v, v_1) \; | \; v_1.CForm(v, w, u_3, u_4)) \]

**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 \text{Reduce} to \(u_1\) so many times as applications of \(\text{succ}\) and \(\text{pred}\) 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.

\[ \text{CanonicalForm}(u, v, w) = (y, u_1)(\text{Duplicate}(u, u_1, u_2) \; | \; u_2.CForm(u, u_1, v, w)) \]

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

**Proof** Straightforward.

\[ \text{Equal}(u, v, w) = (w_1, w_2, w_3, w_4, u_1, u_2, u_3, u_4, u_5, u_6, v_1, v_2, v_3, v_4) \]
\[ \text{CanonicalForm}(u, v_1, u_4) \mid u_1.\text{CanonicalForm}(v, v_2, u_2) \mid u_2.\text{Equal}(v_1, v_2, w) \]

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

**Proof** Straightforward.

\[ [a_1 = a_2] \; v = \]
\[ (u_1, u_2, v_1, v_2, v_3)([a_1] u_1 v_1 | v_1. [a_2] u_2 v_2 | v_2. \text{Equal}(u_1, u_2, v)) \]

12
We need the following auxiliary functions to define Greater

1. Greater2(x, y, z).

2. CanonicalForm(x, z).

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

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.

Greater(u, v, w) = (w_1, w_2, w_3, u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4)
   CanonicalForm(u, v_1, u_1) | u_1.CannoticalForm(v, v_2, u_2) | u_2.Greater2(v_1, v_2, w))

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.

\[ [a_1 > a_2] v = (u_1, u_2, v_1, v_2, v_3, v_4)([a_1] u_1 v_1 | v_1, [a_2] u_2 v_2 | v_2, \text{Greater}(u_1, u_2, v)) \]

To define greater or equal, we use : greater, equal and or.
\[ a_1 \geq a_2 \mid v = (u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4, w_1, w_2) \]
\[ (v_1, v_2.Duplicate(u_1, u_3, u_1, v_3) \mid v_3.Duplicate(u_2, u_4, u_2, v_4) \mid v_2.(Greater(u_1, u_2, w_1) \mid Equal(u_3, u_4, w_2) \mid Or(w_1, w_2, v)) \]

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, \( \text{Stm} \)
\( D \) will range over declarations, \( \text{Decl} \)
\( P \) will range over programs, \( \text{Prog} \)

\( S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \)

\( D ::= \text{var } x; D \mid \epsilon \)
\( \epsilon \) 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

\[ [\text{var } x, D] = \pi \text{zero} \mid [D] \]
\[ [\epsilon] = 0 \]
2. programs

\[ [D; S] \ u = [D] \ | \ [S] \ u \]

3. statements

(a) skip

\[ [\text{skip}] \ u = \mathbb{I}.0 \]

(b) assignment

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

(c) if

\[ [\text{if } b \text{ then } S_1 \text{ else } S_2] \ u = (w)([b] \ w | \ w : \text{true} \Rightarrow [S_1] \ u | \ w : \text{false} \Rightarrow [S_2] \ u) \]

(d) while

\[ [\text{while } b \text{ do } S] \ u = (w)([b] \ w | \ w : \text{true} \Rightarrow [S; \text{while } b \text{ do } S] \ u | \ w : \text{false} \Rightarrow [\text{skip}] \ u) \]

(e) composition

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

The codification is straightforward, what deserves comment is assignment 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 \((w)(\text{Consume}(x, w) | \text{Duplicate}(y, x, y, v))\). 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 $\rightarrow^*$ 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 $P \rightarrow^* Q \not\rightarrow$. We write $P \downarrow$ to mean $P$ converges to some $Q$, and $P \uparrow$ otherwise.

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

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

1. $L \equiv< S, s >$ and $P \equiv (u)([S] u \mid [s])$
2. $L \equiv s$ and $P \equiv (u)(\pi \mid [s])$

where

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

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

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 $P \rightarrow^* P'$

\textbf{Proof.} by structural induction on $L$.

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

Lemma 6.7. The action relations \( \rightarrow \) are the smallest that satisfy the rules on \([2]\) table 2.

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 \( \rightarrow^\alpha \rightarrow^\alpha \).

The weakest reasonable preorder and precongruence are as follows:

Definition 6.9. \( P \sqsubseteq Q \text{ if, for all } \alpha \neq \tau, P \Rightarrow^\alpha \text{ implies } Q \Rightarrow^\alpha \).

\( P \sqsubseteq Q \text{ if, for all contexts } C[\_], C[P] \subseteq C[Q] \).

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

\( S_1 \sqsubseteq S_2 \) if for all states \( s < S_1, s > \uparrow \) implies \( < S_2, s > \uparrow \).

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

Proof. Assume \( < S_1, s > \uparrow \) \( s' \). Let \( P_1 = C([S_1] u) \). By theorem 6.5 \( C([S_1] u) \rightarrow^{\alpha} (u)([\alpha] [s']) \) and \([s'] \downarrow \).

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

Since \( P_2 \) is determinate \( \rightarrow^\alpha P \rightarrow^\alpha, \alpha \neq \tau \) implies \( P \not\Rightarrow \). So, \( P_2 \downarrow \) and then exists an state \( s'' \) such that \( < S_2, s > \uparrow s'' \).
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 \( P_1 \Rightarrow \pi_{\text{zero}} \) and \( P_2 \Rightarrow \pi_{\text{suc}0} \).

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 \( \pi \)-calculus presented in [3]. Nevertheless in what agents concerns we are based on the presentation of \( \pi \)-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 \( \pi \)-calculus, or the language of logic.

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

References


