Version 1.0 February 13, 2016

The Type System of ProcessJ

by Matt B. Pedersen

1 Introduction

This document defines the type system of ProcessJ and specifies the type checker as well. First, let us define the operators we need in the following:

Two types T_1 and T_2 are said to be **equal** if they are the same type, and we write

$$T_1 =_{\mathcal{T}} T_2$$

Given two types T_1 and T_2, we say that they are **type equivalent** if any value of type T_1 can be assign ed to a variable of type T_2 and any value of type T_2 can be assigned to a variable of T_1, and we write

$$T_1 \sim_{\mathcal{T}} T_2$$

Note, if $T_1 =_{\mathcal{T}} T_2 \Rightarrow T_1 \sim_{\mathcal{T}} T_2$, that is, if the two types are the *same* type, then they are trivially type equivalent.

Consider an assignment of type form

$$v = e$$

where v represents a variable and e an expression. If we assume that the type of v is T_v and the type of e is T_e, and if a value of type T_e can be assigned to variable of type T_v, then we say that T_e is **assignment compatible** with T_v and we write

$$T_v :=_{\mathcal{T}} T_e$$

It should be noted, that $T_v :=_{\mathcal{T}} T_e$ and $T_v =_{\mathcal{T}} T_e$ is **not** the same. The former means that a value of type T_e can be assigned to a value of type T_v and the latter that the two types are the same. Consequently, the relation defined by $:=_{\mathcal{T}}$ is not symmetric; that is, $T_1 :=_{\mathcal{T}} T_2$ does not imply that $T_2 :=_{\mathcal{T}} T_1$.

2 Types in ProcessJ

Primitive Types	
Type	Representation
byte	*byte*
short	*short*
char	*char*
integer	*integer*
long	*long*
float	*float*
double	*double*
boolean	*boolean*
string	*string*
barrier	*barrier*
timer	*timer*
Constructed Types	
Type	Representation
array	$Array(t, size)$
record	$Record(n, \{(n_1\ t_1), \ldots, (n_m\ t_m)\})$
protocol	$Protocol(n, \{(tag_1, \{(n_{1,1}\ t_{1,1}), \ldots, (n_{1,m_1}\ t_{1,m_1})\}),$ $\ldots,$ $(tag_k, \{(n_{k,1}\ t_{k,1}), \ldots, (n_{k,m_k}\ t_{k,m_k})\})\})$
channel	$Channel(t)$
channel end	$ChannelEnd(t, access)$
procedure	$Procedure(n, (t_1, \ldots, t_m), t)$

t, t_i, and $t_{i,j}$ are types; n and n_i are names; $access \in \{\texttt{read}, \texttt{write}\}$, and tag_i is also a name.

2.1 Type Predicates

There is a type predicate for each type, for example, $byte_?, short_?, \ldots, Array_?, \ldots, NamedType_?$. In addition it is useful with the following additional predicates:

- $Integral_?(t) = (t \in \{byte, short, char, int, long\}$
- $Numeric_?(t) = (Integral_?(t) \vee t \in \{float, double\})$.
- $Primitive_?(t) = (Numeric?(t) \vee t \in \{boolean, string, barrier, timer\})$

Naturally, something like $t \in \{float, double\}$ could have been written as $(float_?(t) \vee double_?(t))$.

2.2 Type Equality ($=_{\mathcal{T}}$)

The $=_{\mathcal{T}}$ operator is used to determine *type equality*, that is, it evaluates to true, if and only if two types are the same.

2.2.1 Primitive Types

A primitive type t_1 is only ever equal to another primitive type t_2 if they are both the same type. That is

$$t_1 =_{\mathcal{T}} t_2 \Leftrightarrow Primitive_?(t_1) \wedge Primitive_?(t_2) \wedge (t_1 = t_2)$$

2.2.2 Arrays

Two array types $a_1 = Array(t_1, s_1)$ and $a_2 = Array(t_2, s_2)$ are equal if and only if they both have the same base type and their sizes are the same or both undefined ($\perp$), that is:

$$a_1 =_{\mathcal{T}} a_2 \Leftrightarrow Array_?(a_1) \wedge Array_?(a_2) \wedge (t_1 =_{\mathcal{T}} t_2) \wedge (s_1 = s_2)$$

2.2.3 Records and Protocols

ProcessJ uses *name equivalence* for records and protocols, so for two such types to be the same they have to have the same type name. Therefore, we get for two record types $r_1 = Record(n_1, \{\ldots\})$ and $r_2 = Record(n_2, \{\ldots\})$:

$$r_1 =_{\mathcal{T}} r_2 \Leftrightarrow Record_?(r_1) \wedge Record_?(r_2) \wedge (n_1 = n_2)$$

and for two protocol types $p_1 = Protocol(n_1, \{\ldots\})$ and $p_2 = Protocol(n_2, \{\ldots\})$:

$$r_1 =_{\mathcal{T}} r_2 \Leftrightarrow Protocol?(r_1) \wedge Protocol?(r_2) \wedge (n_1 = n_2)$$

2.3 Channels and Channel Ends

For channel types $c_1 = Channel(t_1)$ and $c_2 = Channel(t_2)$ to be equal they must carry the same type:

$$c_1 =_{\mathcal{T}} c_2 \Leftrightarrow Channel_?(t_1) \wedge Channel_?(t_2) \wedge (t_1 =_{\mathcal{T}} t_2)$$

For two channel ends $e_1 = ChannelEnd(t_1, a_1)$ and $t_2 = ChannelEnd(t_2, a_2)$ we further require that the access (read or write) is the same:

$$e_1 =_{\mathcal{T}} e_2 \Leftrightarrow ChannelEnd_?(e_1) \wedge ChannelEnd_?(e_2) \wedge (t_1 =_{\mathcal{T}} t_2) \wedge (a_1 = a_2)$$

2.4 Procedures

Most languages do not consider procedures a type, but ProcessJ has mobile procedures so we need a procedure type.

For two procedure types to be equal we require they have the same name and the same signature – thus, a procedure type is only ever equal to itself. For $p_1 = Procedure(n_1, (t_{1,1}, \ldots, t_{1,m_1}), t_1)$ and $p_2 = Procedure(n_2, (t_{2,1}, \ldots, t_{2,m_2}), 2_1)$ we have:

$$p_1 =_{\mathcal{T}} p_2 \Leftrightarrow (n_1 = n_2) \wedge (m_1 = m_2) \wedge (t_1 = t_2) \wedge (\bigwedge_{i=0}^{m_1} (t_{1,i} =_{\mathcal{T}} t_{2,i}))$$

Note, we could have added the names of the parameters in the type if we wanted and then required that they also match, but most languages – ProcessJ included – does not allow the same procedure to be re-implemented with the same name and type signature (parameters and return type).

2.4.1 Named Types

ProcessJ does not have real named types, that is, there is no type definition operator that can introduce aliasing for types. The *NamedType* type is used temporarily until resolved. Consider the following ProcessJ code:

```
record R {
  int id;
  string name;
}

...
proc void foo() {
  R myR;
  ...
}
```

The type R in the declaration of myR is created as a named type. All named types must resolve to real types during name resolution, thus, the type checker should never encounter a named type.

3 Type Equivalence ($\sim_{\mathcal{T}}$)

In ProcessJ, type equivalence is equal to type equality, so therefore for two types t_1 and t_2 we have:

$$(t_1 \sim_{\mathcal{T}} t_2) \Leftrightarrow (t_1 =_{\mathcal{T}} t_2)$$

4 Assignment Compatability ($:=_{\mathcal{T}}$)

Because of coercion of numeric types and because of inheritance in protocols assignment compatability is a little more than just type equivalence.

4.1 The Ordering Operator ($\leq_{\mathcal{T}}$)

The ordering operator is needed for both cases, that is, we need to define it for both primitives (actually, numeric primitive types only) and for protocols.

it is worth noting that the following implication follows trivially:

$$t_1 =_{\mathcal{T}} t_2 \Rightarrow ((t_1 \leq_{\mathcal{T}} t_2) \wedge (t_2 \leq_{\mathcal{T}} t_1))$$

4.1.1 Ordering of Primitive Numeric Types

In ProcessJ we have chosen to use the same rule as in Java and C/C++; that is, for an assignment line `a = b` where both `a` and `b` are numeric, we say that it is ok if `b` value can be coerced into a value of the type of `a` without loss of precision. This is not the case for integer values stored in floats and longs in doubles, but we disregard that little issue to be compatible with Java. So we get the following:

$$byte \leq_{\mathcal{T}} short \leq_{\mathcal{T}} char \leq_{\mathcal{T}} int \leq_{\mathcal{T}} long \leq_{\mathcal{T}} double$$

$$int \leq_{\mathcal{T}} float \leq_{\mathcal{T}} double$$

and we can define a ceiling function in the following way:

$$\lceil t_1, t_2 \rceil = \begin{cases} t_1 & \text{If } t_2 \leq_{\mathcal{T}} t_1 \\ t_2 & \text{If } t_1 \leq_{\mathcal{T}} t_2 \\ \bot & \text{Otherwise} \end{cases}$$

4.1.2 Ordering of Protocol Types

Protocol type ordering is based on inheritance, but it is important to understand the difference between object inheritance in a language like Java and protocol inheritance in ProcessJ. Let us briefly consider object inheritance first. Suppose we have the following code:

```
class A {
  int a;
}

class B extends A {
  int b;
}
```

then a variable of type `A` can hold a reference to an object of type `B`, not a variable of type `B` cannot hold a reference to an object of type `A`. To understand why assume that a variable of type `A` could hold a reference to an object of type `A` and consider the following code:

```
...
B myB = new A();

myB.b = 100;
```

Clearly, the `b` field does not exist in the `A` object. Now consider the following ProcessJ declarations:

```
protocol A {
  a1: { ... }
  a2: { ... }
}

protocol B extends A {
  b1: { ... }
  b2: { ... }
  b3: { ... }
}
```

If we are not careful, we run into the same problem as we did with objects. For example if the following code was legal we would have a problem:

```
chan<A> c;
B b = c.read();
...  b.b3 ...
```

The problem here consists of the unchecked access to the variant cases of `b`. In ProcessJ **all** access to variables must happen in a tag-guarded switch statement. That way the compiler can assure that the wrong fields are never accessed. The proper code would look like this:

```
chan<A> c;
B b = c.read();
switch (b) {
  case a1: // access to b.a1 is legal here
  case a2: // access to b.a2 is legal here
  case b1: // access to b.b1 is legal here
  case b2: // access to b.b2 is legal here
  case b3: // access to b.b2 is legal here
}
```

Naturally, since `c` carries only values of type `A`, all the tag cases from `B` will never be executed in the code above. So now we can say that one protocol type is less than another if its cases are a subset of the other – here we do take both names and types into consideration. So for two types
$p_1 = Protocol(n_1, \{(tag_{1,1}, \{(n_{1,1,1}\ t_{1,1,1}), \ldots, (n_{1,1,m_{1,1}}\ t_{1,1,m_{1,1}})\}),$
$(tag_{1,2}, \{(n_{1,2,1}\ t_{1,2,1}), \ldots, (n_{1,2,m_{1,2}}\ t_{1,2,m_{1,2}})\}),$
$\ldots$
$(tag_{1,k_1}, \{(n_{1,k_1,1}\ t_{1,k_1,1}), \ldots, (n_{1,k_1,m_{1,k_1}} t_{1,k_1,m_{1,k_1}})\})\})$ and
$p_2 = Protocol(n_2, \{(tag_{2,1}, \{(n_{2,1,1}\ t_{2,1,1}), \ldots, (n_{2,1,m_{2,1}}\ t_{2,1,m_{2,1}})\}),$
$(tag_{2,2}, \{(n_{2,2,1}\ t_{2,2,1}), \ldots, (n_{2,2,m_{2,2}}\ t_{2,2,m_{2,2}})\}),$
$\ldots$
$(tag_{2,k_2}, \{(n_{2,k_2,1}\ t_{2,k_2,1}), \ldots, (n_{2,k_2,m_{2,k_2}} t_{2,k_2,m_{2,k_2}})\})\})$ we define

$(p_1 \leq_{\mathcal{T}} p_2) \Leftrightarrow (\forall i : (1 \leq i \leq k_1) : tag_{1,i}$ exists as a tag with index $2, j$ in $p_2) \wedge$

$$(m_{1,i} = m_{2,j}) \wedge (\bigwedge_{k=1}^{m_{1,i}} ((n_{1,i,k} = n_{2,j,k}) \wedge (n_{1,i,k} \sim_{\mathcal{T}} n_{2,j,k}))$$

If needed, the ceiling function can be applied to protocol types as well.

4.2 Assignment Compatability

We can not continue with the definition of assignment compatability. The assignment compatability operator ($:=_{\mathcal{T}}$) determines if an assignment like:

$$v = e$$

where t_v is the type of the left hand side variable v and t_e is the type of the right hand side expression is legal, that is if $t_v :=_{\mathcal{T}} t_e$ is true.

4.2.1 Assignment Compatability for Primitives

For primitive types we utilize the $\leq_{\mathcal{T}}$ operator from the previous subsection. So for two primitive type t_1 and t_2 we have:

$$(t_1 :=_{\mathcal{T}} t_2) \Leftrightarrow \mathit{Primitive}_?(t_1) \wedge \mathit{Primitive}_?(t_2) \wedge (t_2 \leq_{\mathcal{T}} t_1)$$

4.2.2 Assignment Compatability for Arrays

For arrays we have to be a little careful; An array of integers cannot be assigned to an array type of doubles even if $\mathit{int} \leq_{\mathcal{T}} \mathit{double}$, but if the base type is a protocol, then we can make such an assignment as long as the ordering of protocol types is obeyed. So we have the following for two array types $a_1 = \mathit{Array}(t_1, d_1)$ and $t_2 = \mathit{Array}(t_2, d_2)$:

$$(t_1 :=_{\mathcal{T}} t_2) \Leftrightarrow \mathit{Array}_?(t_1) \wedge \mathit{Array}_?(t_2) \wedge$$

$$(\mathit{Protocol}_?(t_1) \wedge \mathit{Protocol}_?(t_2) \wedge (t_2 \leq_{\mathcal{T}} t_1)) \vee$$

$$(\neg \mathit{Protocol}_?(t_1) \wedge \neg \mathit{Protocol}_?(t_2) \wedge (t_1 \sim_{\mathcal{T}} t_2)))$$

4.2.3 Assignment Compatability for Records

Records are straightforward. For two record types $r_1 = \mathit{Record}(n_1, \{\ldots\})$ and $r_2 = \mathit{Record}(n_2, \{\ldots\})$ we have

$$(r_1 :=_{\mathcal{T}} r_2) \Leftrightarrow \mathit{Record}_?(r_1) \wedge \mathit{Record}_?(r_2) \wedge (r_1 \sim_{\mathcal{T}} r_2)$$

4.2.4 Assignment Compatability for Channels and Channel Ends

Channels cannot be assigned using the = operator, and neither channel ends. They are immutable. Channels cannot be passed as parameters either, only channel ends can.

Therefore we only need to define assignment compatability for channel ends. For two channel end types $e_1 = ChannelEnd(t_1, a_1)$ and $e_2 = ChannelEnd(t_2, a_2)$ we have:

$$(e_1 :=_{\mathcal{T}} e_2) \Leftrightarrow ChannelEnd_?(e_1) \wedge ChannelEnd_?(e_2) \wedge (e_1 \sim_{\mathcal{T}} e_2)$$

4.2.5 Assignment Compatability for Procedures

Since procedures can be mobile, and thus travel on channels, we need to define assignment compatability for procedures. It should be noted that the following code in ProcessJ is valid (or should be!):

```
mobile proc void foo() ;

mobile proc void bar() implements foo { }

proc void main() {
  foo f = new f;
  chan<foo> c;

  c.write(f);
}
```

So assignment compatability is as simple as requiring all parameters be type equal. For two procedure types $p_1 = (n_1, (t_{1,1}, \ldots, t_{1,k_1}), t_1)$ and $p_2 = (n_2, (t_{2,1}, \ldots, t_{2,k_2}), t_2)$ we have:

$$(p_1 :=_{\mathcal{T}} p_2) \Leftrightarrow Procedure?(p_1) \wedge Procedure_?(p_2) \wedge (k_1 = k_2) \wedge$$
$$(t_1 =_{\mathcal{T}} t_2) \wedge \bigwedge_{i=0}^{k_1} (t_{1,i} =_{\mathcal{T}} t_{2,i})$$

4.2.6 Assignment Compatability for Named Types

Since named types do not figure in the type checking phase we do not need to consider them here.

5 Type Checking

6 Notes

The compiler crashes on the mobile stuff at the moment - and the new keyword isn't implemented yet .... the type checker doesn't like passing

procedure types as it is right now ;-) it dies.