Sequential program composition in UNITY

Tanja Vos and **Doaitse Swierstra**
Utrecht University, Department of Computer science
e-mail: {tanja, doaitse}@cs.uu.nl

February 28, 2000

1 Introduction

Large distributed applications are composed of basic blocks, by using composition operators. In an ideal situation, one should be able to develop and verify each of these basic components by itself, using compositionality theorems of the respective composition operators stating that properties of a composite program can be proved by proving properties of its components.

Generally, two forms of distributed program composition can be distinguished: parallel composition and sequential composition. Parallel composition is standard in UNITY [CM89], and is used when two distributed component-programs need to cooperate in one way or another. Compositionality theorems of parallel composition on general progress properties are extensively studied in [CM89, Sin89a, Pra95]. Sequential composition of UNITY programs is not part of core UNITY [CM89]. It can however be very useful when we want a program to work with the results of another program. For example, for the Propogation of Information with Feedback (PIF) protocol [Seg83]:

elect a leader $\fatsemi$ let the leader be the starter of the PIF protocol

In [Mis90b], a brief and intuitive characterisation of sequential composition is given. In this technical report, we shall formally define and model sequential program composition within the HOL-UNITY embedding described in [Pra95, Vos00] In order to do so, we introduce a new type of UNITY programs called UNITY$^+$ programs which consist of sequentially composed UNITY programs. The semantics of a UNITY$^+$ programs is then defined in terms of a UNITY program that models the desired behaviour of the sequential composition. Finally, safety and progress operators are defined for these UNITY$^+$ programs, and compositionality theorems are derived. For those readers not familiar with UNITY and its embedding in HOL, Appendix A contains a brief overview of it. For those readers that are familiar with UNITY, we have compiled an extensive index that should enable the reader to start reading this technical report, looking up desired definitions in a demand-driven way.

2 Semantics of sequential program composition

In [Mis90b], sequential composition of programs $P \fatsemi Q$ is defined intuitively in operational terms as follows. Program P's execution is started. If a fixed-point state of P is reached, the execution of Q is started from that state. In this technical report, we generalise this by parametrising $\fatsemi$ with some state-predicate, and interpreting $P \fatsemi_r Q$ as follows. Program P's execution is started. If a predicate r holds in some state during the execution of P, the execution of Q is started from that state. Consequently, if r is a fixed-point of P (i.e. $_P\vdash \mathsf{FP}.r$), then our operational intuition of $\fatsemi$ corresponds to that of [Mis90b].

In order to formalise the $\fatsemi_r$-operator, we have to find a way to enforce that:

- the execution of P is stopped when r holds

- the execution of Q is started in exactly that state where r started to hold in P

In [Mis90b], r is assumed to be a fixed-point of P, and, as a consequence, if r holds, then the execution of P has effectively stopped. Moreover, in [Mis90b] it is implicitly assumed that the execution of a UNITY program is started only when its initial condition is satisfied, and since only those programs Q that have P's fixed-point as their initial condition are considered it is ensured that once P stops, Q can start executing. In our case, however, it is not as simple as this. First, we decided to generalise our $\fatsemi$ operator by parametrising it with some state-predicate which is not necessarily P's fixed-point. Second, in our HOL-UNITY embedding, where a program's progress properties are proved independently from its initial condition [Pra95], we cannot use this approach from [Mis90b] to ensure that Q is started in exactly that state where r started to hold in P. Consequently, we have to deal with these two aspects explicitly when formally defining the semantics of our $\fatsemi$-operator. In order to explain the approach we have taken, we use the two UNITY programs P and Q from below.

prog	P	prog	Q
read	$\{x\}$	read	$\{x\}$
write	$\{x\}$	write	$\{y\}$
init	$x = 0$	init	true
assign	if $x \geq 0$ then $x := x + 1$	assign	$y := x$

Obviously, we need to define the semantics of $P \fatsemi_r Q$, such that:

- as soon as predicate r holds, all guards of P its actions are disabled and remain disabled forever (i.e. we transform P such that r becomes a fixed-point of the transformation).
- when r is not yet satisfied during the execution of P, all guards of Q are disabled, and as soon as r becomes true the guards of Q's actions are enabled as far as this is allowed by Q itself.

In order to achieve this, we introduce a fresh variable pc, the value of which indicates which program (i.e. P or Q) is allowed to execute. We make sure that once r becomes true the value of the pc is adjusted as to ensure that the execution of P is stopped and that of Q is started. Subsequently, we transform P and Q by strengthening the guards of all their actions such that these are only enabled when the value of the pc allows them. Moreover, we strengthen the guards of all actions in the programs P with $\neg r$ such that these actions immediately become disabled when r becomes true (i.e. r becomes a fixed-point). Finally, we compose these transformations using parallel composition. For programs P and Q, this results in the following semantics of $P \fatsemi_r Q$:

prog	$P \fatsemi_r Q$
read	$\{x\}$
write	$\{x, y\}$
init	$(x = 0) \wedge (\mathsf{pc} = 0)$
assign	if $(x \geq 0) \wedge (\mathsf{pc} = 0) \wedge \neg r$ then $x := x + 1$
$[]$	if $(\mathsf{pc} = 0) \wedge r$ then $\mathsf{pc} := \mathsf{pc} + 1$
$[]$	if $(\mathsf{pc} = 1)$ then $y := x$

This evidently gives the desired effect. If r becomes true, the guard of P's action is immediately disabled since it contains $(\neg r)$. Eventually the pc will be incremented and become 1. As a result, Q can start executing, and since the pc will never change again P has stopped.

Now, we have laid the foundations of how we will define the semantics of $\fatsemi_r$. Subsequently, we have to determine how we handle sequences of sequential compositions:

$$P \fatsemi_r Q \fatsemi_s H$$

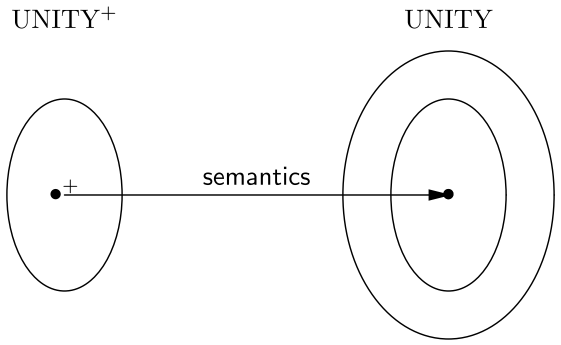

Figure 1: Semantics of a UNITY$^+$ program

A property following naturally from the intuitive interpretation of sequential composition, is that of associativity. Consequently, we want to define the semantics of $\fatsemi_r$ such that:

$$P \fatsemi_r Q \fatsemi_s H \;=\; (P \fatsemi_r Q) \fatsemi_s H \;=\; P \fatsemi_r (Q \fatsemi_s H)$$

From the discussion above, we can derive that the following program captures the intended meaning of these sequences:

```
prog    $P \fatsemi_r Q \fatsemi_s H$
read    $\mathbf{r}P \cup \mathbf{r}Q \cup \mathbf{r}H$
write   $\mathbf{w}P \cup \mathbf{w}Q \cup \mathbf{w}H$
init    $\mathbf{ini}P \wedge \mathbf{ini}Q \wedge \mathbf{ini}H \wedge (\mathsf{pc} = 0)$
assign  if $(\mathsf{pc} = 0) \wedge \neg r$ then $P$                    (☼)
[]      if $(\mathsf{pc} = 0) \wedge r$ then $\mathsf{pc} := \mathsf{pc} + 1$
[]      if $(\mathsf{pc} = 1) \wedge \neg s$ then $Q$
[]      if $(\mathsf{pc} = 1) \wedge s$ then $\mathsf{pc} := \mathsf{pc} + 1$
[]      if $(\mathsf{pc} = 2)$ then $H$
```

When defining the semantics of $\fatsemi_r$ such that the latter is associative, we need to make sure that it is defined in such a way that the semantics of $(P \fatsemi_r Q) \fatsemi_s H$ as well as $P \fatsemi_r (Q \fatsemi_s H)$ are equal to the program presented above. Consequently, since we allow sequential composition of any (finite) number of programs, a shallow embedding of $\fatsemi$ is inadequate to ensure that the right values of pc are used when strengthening the guards. Therefore, we define the $\fatsemi$ operator using a deep embedding[1]. Since, a sequence of sequential compositions consists of "simple" UNITY programs (i.e. those of type `Uprog` (see page 10)), composed with the $\fatsemi$ operator, we define the abstract syntax of UNITY$^+$ programs by the following recursive data type:

Definition 2.1 TYPE DEFINITION *UNITY_plus*

$$\begin{array}{lcl} \mathsf{Uprog}^+ & = & \mathsf{Simple.Uprog} \\ & | & \mathsf{Uprog}^+ \fatsemi_{\mathtt{Expr}} \mathsf{Uprog}^+ \end{array}$$

Now we can define the semantics and other properties of sequential composition as recursive functions over this data type. For example, the write variables of a UNITY$^+$ program can be obtained as follows (we overload the **w** destructor from Appendix A.2):

[1]In a deep embedding, the abstract syntax of the language is defined as a type in the HOL logic, and the semantics is defined as recursive functions over this type.

Definition 2.2 Write variables of a UNITY$^+$ program *WRITE_plus*

$$\begin{aligned}\mathbf{w}(\mathsf{Simple}.P) &= \mathbf{w}P\\ \mathbf{w}(P^+ \,{}_{9}^{\circ}{}_r\, Q^+) &= \mathbf{w}P^+ \cup \mathbf{w}Q^+\end{aligned}$$

◀

Continuing with the semantics, suppose we need to define the semantics of $P^+ \,{}_{9}^{\circ}{}_r\, Q^+$, where P^+ and Q^+ can consist of arbitrary sequential compositions. The first value of the pc used in the semantics of Q^+ has to be the successor of the maximal pc used in the semantics of P^+. Moreover, only the guard of the last "simple" UNITY program in P^+'s sequence has to be strengthened with the negation of the state-predicate r. In order to achieve this we define a transformation function tr that given an arbitrary Uprog$^+$ program U^+, and some start value n for the pc (i.e. if the pc is n, then the first Simple UNITY program in the sequence U^+ is allowed to execute) returns a tuple like:

$$(U \in \texttt{Expr}{\rightarrow}\texttt{Uprog},\ m \in \texttt{num})$$

such that given a state-predicate r, $U.r$ will execute until r holds and when r holds the value of the pc will be m. Note that, $U.r$ denotes the semantics of $U^+{}_{9}^{\circ}{}_r$. Inspecting the examples given earlier, when U^+ is a simple Uprog program P, this transformation shall return:

$$((\lambda r.\ \mathsf{if}\ \mathsf{pc} = n \wedge \neg r\ \mathsf{then}\ P\ \ [\!]\ \ \mathsf{if}\ (\mathsf{pc} = n) \wedge r\ \mathsf{then}\ \mathsf{pc} := \mathsf{pc} + 1), n + 1)$$

Now, suppose that U^+ is a composite Uprog$^+$ program $P^+ \,{}_{9}^{\circ}{}_r\, Q^+$. If transforming P^+ with n as the start value of the pc results in $(P,\, m)$, we can transform Q^+ with m as the start value of the pc since then we know that the first value of the pc used in the semantics of Q^+ is the successor of the maximal pc used in the semantics of P^+. If this transformation results in $(Q,\, k)$, then we can ensure that only the guard of the last "simple" UNITY program in P^+'s sequence is strengthened with the negation of the state-predicate r, by defining the transformation of $P^+ \,{}_{9}^{\circ}{}_r\, Q^+$ with n as the start value of the pc to result in:

$$((\lambda q.\ (P.r\ \ [\!]\ Q.q)), k)$$

The formal definition of this transformation function is stated below. We use restricted union superposition (Definition A.21$_{13}$) to transform Simple UNITY$^+$ programs.

Definition 2.3 *tr_DEF*

$$\begin{aligned}\mathsf{tr}.(\mathsf{Simple}.P).\mathsf{pc}.n = {}& ((\lambda r.\ \mathsf{RU\_S}.(\mathsf{strengthen\_guards}.(\mathsf{pc} = n \wedge \neg r).P)\\ &\qquad\qquad .(\mathsf{if}\ (\mathsf{pc} = n) \wedge r\ \mathsf{then}\ \mathsf{pc} := \mathsf{pc} + 1)\\ &\qquad\qquad .\mathsf{true})\\ &, n + 1)\end{aligned}$$

$$\begin{aligned}\mathsf{tr}.(P^+ \,{}_{9}^{\circ}{}_r\, Q^+).\mathsf{pc}.n = {}& \mathsf{let}\ (P, m) = \mathsf{tr}.P^+.\mathsf{pc}.n \wedge (Q, k) = \mathsf{tr}.Q^+.\mathsf{pc}.m\\ & \mathsf{in}\ ((\lambda q.\ P.r\ [\!]\ Q.q),\ k)\end{aligned}$$

◀

Now the semantics of a UNITY$^+$ program is defined by the following function:

Definition 2.4 *semantics*

$$\mathsf{semantics}.U^+.\mathsf{pc}.n.q\ =\ \mathsf{add\_to\_initial\_cond}.(pc = n).(\mathsf{FST}.(\mathsf{tr}.U^+.\mathsf{pc}.n.q))$$

◀

```
prog    $P \fatsemi_r Q \fatsemi_s H$
read    $\mathbf{r}P \cup \mathbf{r}Q \cup \mathbf{r}H$
write   $\mathbf{w}P \cup \mathbf{w}Q \cup \mathbf{w}H$
init    $\mathbf{ini}P \wedge \mathbf{ini}Q \wedge \mathbf{ini}H \wedge (\mathsf{pc} = 0)$
assign  if $(\mathsf{pc} = 0) \wedge \neg r$ then $P$
[]      if $(\mathsf{pc} = 0) \wedge r$ then $\mathsf{pc} := \mathsf{pc} + 1$
[]      if $(\mathsf{pc} = 1) \wedge \neg s$ then $Q$
[]      if $(\mathsf{pc} = 1) \wedge s$ then $\mathsf{pc} := \mathsf{pc} + 1$
[]      if $(\mathsf{pc} = 2) \wedge \boxed{\mathsf{true}}$ then $H$
[]      $\boxed{\text{if } (\mathsf{pc} = 2) \wedge \mathsf{false} \text{ then } \mathsf{pc} := \mathsf{pc} + 1}$
```

Figure 2: The result of $\mathsf{semantics}.(P \fatsemi_r Q \fatsemi_s H).\mathsf{pc}.0.\mathsf{false}$

where n is the start value of the pc, and q is a state-predicate indicating when the last simple program in the composition is allowed to stop executing (i.e. its exit condition). If q is false, this indicates that the actions of the last simple UNITY program in a sequential composition may be enabled indefinitely and that the pc will not be incremented anymore. Note that using this definition, the semantics of $P \fatsemi_r Q \fatsemi_s H$ is slightly different from that presented on page 3 (at the ☼). More specific, $\mathsf{semantics}.(P \fatsemi_r Q \fatsemi_s H).\mathsf{pc}.0.\mathsf{false}$ results in the UNITY program depicted in Figure 2_5. However, it is easily recognised that the semantics are the same, since true is the identity element of $\wedge$, and the last action will always be a skip action because false is never satisfied. Proving that $\fatsemi_r$ is associative is now straightforward, since ($[\!]$) is associative:

Theorem 2.5 Associativity of $\fatsemi$ *SEQ_ASSOC*
For arbitrary UNITY$^+$ programs P^+, Q^+, and H^+, and state-predicates $r, s \in \texttt{Expr}$:

$$\mathsf{semantics}.((P^+ \fatsemi_r Q^+) \fatsemi_s H^+) = \mathsf{semantics}.(P^+ \fatsemi_r (Q^+ \fatsemi_s H^+))$$

The formalisation of the semantics of $\fatsemi_r$ is now almost completed. There is, however, a snag in it somewhere. We still have to ensure that the variable pc, that is used to define the semantics of the UNITY$^+$ programs, is a *fresh* variable. That is, all actions of the sequentially composed programs, and all predicates attached to the $\fatsemi$ must ignore this variable pc:

Definition 2.6 variables ignored by UNITY$^+$ program *IG_BY_Uplus_DEF*

$$V \overset{+}{\not\Leftarrow} (\mathsf{Simple}.P) = V \not\Leftarrow P$$
$$V \overset{+}{\not\Leftarrow} (P^+ \fatsemi_r Q^+) = (V \overset{+}{\not\Leftarrow} P^+) \wedge (V \overset{+}{\not\Leftarrow} Q^+) \wedge (V^{\mathsf{c}}\ \mathcal{C}\ r)$$

Consequently, the function $\mathsf{semantics}.U^+.\mathsf{pc}$ only defines the desired semantics for U^+, when $\{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+$.

We end this section by stating some properties of the semantics of $\fatsemi$. The maximal value the pc can reach in the program $\mathsf{semantics}.U^+.\mathsf{pc}.n.q$ is defined by:

Definition 2.7 *max_pc*

$$\mathsf{max\_pc}.U^+.\mathsf{pc}.n \;=\; \mathsf{SND}.(\mathsf{tr}.U^+.\mathsf{pc}.n)$$

From the definition of tr (2.3_4), it is straightforward to deduce:

Theorem 2.8 *pc_INCR_thm*

$$\forall U^+ \ \mathsf{pc}\ n \ :: \ n \ < \ (\mathsf{max\_pc}.U^+.\mathsf{pc}.n)$$

◀

When the value of the pc is less that n, or greater than or equal to $\mathsf{max\_pc}.U^+.\mathsf{pc}.n$, all actions in the program $\mathsf{semantics}.U^+.\mathsf{pc}.n.q$ (for arbitrary q) are disabled. Consequently, these are fixed-points of this program:

Theorem 2.9 *FP_when_pc_too_small, FP_when_pc_too_big*

$$\forall U^+ \ \mathsf{pc}\ n\ q :: \frac{U = \mathsf{semantics}.U^+.\mathsf{pc}.n.q \ \wedge \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+}{(\ {}_U\vdash \mathsf{FP}.(\mathsf{pc} < n)) \quad \wedge \quad (\ {}_U\vdash \mathsf{FP}.(\mathsf{pc} \geq (\mathsf{max\_pc}.U^+.\mathsf{pc}.n)))}$$

◀

The value of the pc shall never decrease during the execution of $\mathsf{semantics}.U^+.\mathsf{pc}.n.q$ (for arbitrary q), so:

Theorem 2.10 *pc_GE_start_value_IS_STABLEe*

$$\forall U^+ \ \mathsf{pc}\ n\ q :: \frac{U = \mathsf{semantics}.U^+.\mathsf{pc}.n.q \ \wedge \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+}{{}_U\vdash \circlearrowleft (\mathsf{pc} > n)}$$

◀

Finally, during the execution of $\mathsf{semantics}.U^+.\mathsf{pc}.n.q$ (for arbitrary q), the value of the pc will be less than $\mathsf{max\_pc}.U^+.\mathsf{pc}.n$ until q holds and the pc is incremented such that it gets its maximum value:

Theorem 2.11 *pc_LT_max_UNTIL_pc_is_max_AND_pred*

$$\forall U^+ \ \mathsf{pc}\ n\ q :: \frac{U = \mathsf{semantics}.U^+.\mathsf{pc}.n.q \ \ \wedge \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+ \ \wedge \ q\ \mathcal{C}\ \{\mathsf{pc}\}^{\mathsf{c}}}{{}_U\vdash (\mathsf{pc} < \mathsf{max\_pc}.U^+.\mathsf{pc}.n) \ \mathsf{unless}\ ((\mathsf{pc} = \mathsf{max\_pc}.U^+.\mathsf{pc}.n) \wedge q)}$$

◀

3 Proving properties of program sequencing

When working with UNITY$^+$ programs, the semantics underlying the $;_r$ operator should be hidden, and the user should be able to prove properties of sequentially composed programs by proving properties of their component programs. In order to establish this, we first define safety and progress operators for UNITY$^+$ programs in terms of the standard safety and progress operators for UNITY programs. Subsequently we derive theorems that state how safety and progress properties of UNITY$^+$ programs can be proved by reducing them to standard UNITY properties of the component programs.

To express safety properties of UNITY$^+$ programs, we introduce two operators unless^+ and $\circlearrowleft^+$. Since, the semantics of a UNITY$^+$ program requires a state-predicate that indicates the

Theorem 3.3 *UNLESS_IMP_Simple_UNLESS_plus*

$$\frac{{}_{P}\vdash p \;\mathsf{unless}\; q \wedge (\forall \mathsf{pc} : \{\mathsf{pc}\} \not\Leftarrow (\mathsf{Simple}.P) : p \;\mathcal{C}\; \mathsf{pc}^{\mathsf{c}})}{\forall r : {}_{\mathsf{Simple}.P}\vdash \; p \;\mathsf{unless}^{+}_{r}\; q}$$

Theorem 3.4 *UNLESS_SEQ_UNLESS*

$$\frac{q \;\mathcal{C}\; \mathbf{w}P^{+} \wedge \;{}_{P^{+}}\vdash \; p \;\mathsf{unless}^{+}_{q}\; r \;\wedge\; {}_{Q^{+}}\vdash \; p \;\mathsf{unless}^{+}_{s}\; r}{J \;{}_{P^{+}\fatsemi_{q} Q^{+}}\vdash \; p \;\mathsf{unless}^{+}_{s}\; r}$$

Figure 3: Proving safety for UNITY^{+} programs

exit condition of the last simple UNITY program in the sequence of sequential compositions, these operators have an additional parameter stating this condition. Intuitively, for a UNITY^{+} program U^{+}: ${}_{U^{+}}\vdash \; p \;\mathsf{unless}^{+}_{r}\; q$ holds, if, for all numbers n and fresh variables pc, $p \;\mathsf{unless}\; q$ holds in $\mathsf{semantics}.U^{+}.\mathsf{pc}.n.r$. More formally this comes down to:

Definition 3.1 *UNLESS_plus_DEF*

$${}_{U^{+}}\vdash \; p \;\mathsf{unless}^{+}_{r}\; q$$
$$= \forall \mathsf{pc}\; n\; U : \;\{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^{+} \wedge U = \mathsf{semantics}.U^{+}.\mathsf{pc}.n.r \;:\; {}_{U}\vdash p \;\mathsf{unless}\; q$$

Definition 3.2 *STABLE_plus_DEF*

$${}_{U^{+}}\vdash \circlearrowleft^{+}_{r} J$$
$$= \forall \mathsf{pc}\; n\; U : \;\{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^{+} \wedge U = \mathsf{semantics}.U^{+}.\mathsf{pc}.n.r \;:\; {}_{U}\vdash \circlearrowleft J$$

Compositionality theorems of unless^{+} are stated in Figure 3$_{7}$. Similar properties hold for the $\circlearrowleft^{+}$ operator.

To express progress properties of UNITY^{+} programs, we introduce two operators $\overset{+}{\rightarrowtail}$, and $\overset{+}{\rightsquigarrow}$. Again, intuitively for a UNITY^{+} program U^{+}: the validity of $J \;{}_{U^{+}}\vdash \; p \;\overset{+}{\rightarrowtail}\; q$ shall imply that, during the execution of the semantics of U^{+}, when p holds then eventually q holds. However, we have to be more specific about what we mean here. Consider again the following sequential composition:

$$U^{+} \;=\; P \fatsemi_{r} Q \fatsemi_{s} H$$

Suppose, we are at a specific point in the execution of the semantics of U^{+} where the pc is such that it is P's turn to execute and p holds. Now, do we want $J \;{}_{U^{+}}\vdash \; p \;\overset{+}{\rightarrowtail}\; q$ to be valid if, from this specific point eventually q holds in P while actions of Q and H have not yet been executed? In order to answer this question, we have to consider what we are aiming at. As previously indicated, we want to derive theorems stating how progress properties of UNITY^{+} programs can be proved by reducing them to standard UNITY properties of the component programs. More specific, for the case of U^{+} above, these theorems shall state something like:

$$\frac{J \;{}_{P\fatsemi_{r}Q}\vdash \; p \;\overset{+}{\rightarrowtail}\; s \;\wedge\; J \;{}_{H}\vdash \; s \;\overset{+}{\rightarrowtail}\; r}{J \;{}_{P\fatsemi_{r}Q\fatsemi_{s}H}\vdash \; p \;\overset{+}{\rightarrowtail}\; r}$$

Suppose we know that:

$$J\ {}_{H}\!\vdash\ (x = 10)\ \overset{+}{\rightarrowtail}\ \textit{something beautiful}$$

Let P, and Q be the following programs:

prog	P	prog	Q
read	$\emptyset$	read	$\emptyset$
write	$\{x\}$	write	$\{x\}$
init	*some initial condition*	init	$(x = 9)$
assign	$x := 10$	assign	$x := 10$

Moreover, for the sake of the argument suppose that the answer to the previous question would be yes, and we define:

$$J\ {}_{U^+}\!\vdash\ p\ \overset{+}{\rightarrowtail}\ q = \forall \mathsf{pc}\ n\ U :\ \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+ \wedge U = \mathsf{semantics}.U^+.\mathsf{pc}.n.q\ :\ J\ {}_{U}\!\vdash p\ \rightarrowtail q$$

Therefore, we are able to prove that:

$$J\ {}_{P\,\fatsemi_{(x=9)}\,Q}\!\vdash\ \textit{some initial condition}\ \overset{+}{\rightarrowtail}\ (x = 10)$$

However, since the x will never be 9, the pc in the semantics of $P\ \fatsemi_{(x=9)}\ Q$ will never be incremented, and consequently, we cannot prove that

$$J\ {}_{P\,\fatsemi_{(x=9)}\,Q\,\fatsemi_{(x=10)}\,H}\!\vdash\ \textit{some initial condition}\ \overset{+}{\rightarrowtail}\ \textit{something beautiful}$$

since, H will never get a chance to execute. Thus, defining $\overset{+}{\rightarrowtail}$ in this way does not enable us to prove the theorems we are aiming at, and the answer to the question posed above is no. From this discussion we can derive that we only want $J\ {}_{U^+}\!\vdash\ p\ \overset{+}{\rightarrowtail}\ q$ to be valid if, q eventually holds during the execution of the last program in the $\fatsemi$-sequence U^+. This can be established by letting q be the exit condition of the last program in the $\fatsemi$-sequence U^+, and requiring that the value of the pc in the semantics of this sequence eventually reaches its maximum value. Consequently, progress properties for UNITY$^+$ programs are defined as follows:

Definition 3.5 *REACH_plus_DEF*

$$J\ {}_{U^+}\!\vdash\ p\ \overset{+}{\rightarrowtail}\ q$$

$$= \forall \mathsf{pc}\ n\ m\ U :\ \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+ \wedge U = \mathsf{semantics}.U^+.\mathsf{pc}.n.q \wedge m = \mathsf{max\_pc}.U^+.\mathsf{pc}.n\ :\ J\ {}_{U}\!\vdash (p \wedge (\mathsf{pc} = n)) \rightarrowtail (\mathsf{pc} = m)$$

Definition 3.6 *CONV_plus_DEF*

$$J\ {}_{P}\!\vdash\ p\ \overset{+}{\rightsquigarrow}\ q$$

$$= \forall \mathsf{pc}\ n\ m\ U :\ \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} U^+ \wedge U = \mathsf{semantics}.U^+.\mathsf{pc}.n.q \wedge m = \mathsf{max\_pc}.U^+.\mathsf{pc}.n\ :\ J\ {}_{U}\!\vdash (p \wedge (\mathsf{pc} = n)) \rightsquigarrow (\mathsf{pc} = m)$$

◄

Compositionality theorems for $\overset{+}{\rightarrowtail}$, are presented below. For $\overset{+}{\rightsquigarrow}$ similar properties hold.

Theorem 3.7 *REACH_IMP_Simple_REACH_plus*

$$\frac{J\ {}_{P}\vdash\ p\ \rightarrowtail\ q \land (\forall \mathsf{pc} : \{\mathsf{pc}\} \not\Leftarrow (\mathsf{Simple}.P) : J\ \mathcal{C}\ \mathsf{pc}^{\mathsf{c}})}{J\ {}_{\mathsf{Simple}.P}\vdash\ p\ \overset{+}{\rightarrowtail}\ q}$$

Theorem 3.8 *REACH_SEQ_REACH*

$$\frac{q\ \mathcal{C}\ \mathbf{w}P^{+} \land J\ {}_{P^{+}}\vdash\ p\ \overset{+}{\rightarrowtail}\ q\ \land\ J\ {}_{Q^{+}}\vdash\ q\ \overset{+}{\rightarrowtail}\ s}{J\ {}_{P^{+}\,{}_{\fatsemi q}\,Q^{+}}\vdash\ p\ \overset{+}{\rightarrowtail}\ s}$$

◄

4 Concluding remarks

In this technical report we have presented a formalisation of sequential program composition in UNITY. We have been brief and have not decribed any application of the developed theory in detail. We think we have obtained a nice formalisation of the semantics of $\fatsemi$. Once the transformation function (tr) was defined, the definitions of the safety and progress operators followed naturally, and the compositionality results were proved smoothly. Moreover, we find that the formalisation illustrates the possibly unexpected complications that can appear when formally defining allegedly simple concepts (like $\fatsemi$) of which the definition and properties are intuitively clear.

References

[CM89] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Austin, Texas, May 1989.

[Mis90a] J. Misra. More on strengthening the guard. *Notes on UNITY*, 19-90, 1990. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Mis90b] J. Misra. Proving progress for program sequencing. *Notes on UNITY*, 16-90, 1990. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Pra95] W. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* PhD thesis, Utrecht University, Oct 1995.

[Seg83] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, 29:23–35, 1983.

[Sin89a] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Sin89b] A.K. Singh. On strengthening the guard. *Notes on UNITY*, 07-89, 1989. `http://www.cs.utexas.edu/users/psp/notesunity.html`.

[Vos00] T.E.J. Vos. *UNITY in Diversity, a stratified approach to the verification of distributed algorithms.* PhD thesis, Utrecht University, Jan 2000.

Appendices

A UNITY

In this section we shall give an overview of the UNITY theory and Prasetya's extensions. We shall concentrate on those concepts that are needed in the rest of this report. For a more thorough treatment the reader is referred to [CM89, Pra95, Vos00].

A.1 Variables, values, states, expressions and actions

The state of a program is represented as a function from a universe `Var` of *all* program variables to a universe `Val` of all values these variables may take. The set of all program states will be denoted by `State`.

A *state-expression* is a function of type `State`$\rightarrow\alpha$, where α is an arbitrary type. The set of all state-expressions will be denoted by `Expr`.

A *state-predicate* is a state-expression where type α is `bool`.
A state-expression f is *confined* by a set of variables V, denoted by $f\ \mathcal{C}\ V$, if f does not restrict the value of any variable outside V:

Definition A.1 State-expression Confinement *CONF_DEF*
For all $f \in (\sigma_1\rightarrow\sigma_2)\rightarrow\alpha$, and $V \subseteq \sigma_1$,

$$f\ \mathcal{C}\ V \ = \ (\forall s,t :: (s\restriction V = t\restriction V) \Rightarrow (f.s = f.t))$$

The confinement operator is monotonic in its second argument.

Theorem A.2 $\mathcal{C}$ Monotonicity *CONF_MONO*

$$V \subseteq W \wedge (f\ \mathcal{C}\ V) \Rightarrow (f\ \mathcal{C}\ W)$$

The actions of a UNITY program can be multiple assignments or guarded multiple assignments. The universe of actions will be denoted by `ACTION`.

A set of variables is V *ignored-by* an action A, denoted by $V \not\leftarrow A$, if executing A's executable in any state does not change the values of these variables. Variables in V^{c} *may* however be written by A.

A.2 UNITY programs

A UNITY program consists of declarations of read variables, write variables, a specification of their initial values, and a set of actions.

An execution of a UNITY program starts in a state satisfying the initial condition and is an infinite and interleaved execution of its actions. In each step of the execution some action is selected and executed atomically. The selection of actions is weakly fair, i.e. non-deterministic selection constrained by the following fairness rule:

> *Each action is scheduled for execution infinitely often, and hence cannot be ignored forever.*

A UNITY program P is modelled by a quadruple (A, J, V_r, V_w) where $A \subseteq$ `ACTION` is a set consisting of P's actions, $J \in$ `Expr` is a state-predicate describing the possible initial states of P, and $V_r, V_w \subseteq$ `Var` are sets containing P's read and write variables respectively. The set of all possible quadruples (A, J, V_r, V_w) shall be denoted by `Uprog`. To access each component of such an `Uprog` object, the destructors **a**, **ini**, **r**, and **w** are introduced. They satisfy the following property:

Theorem A.3 `Uprog` Destructors

$$P \in \texttt{Uprog} \;=\; (P \;=\; (\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P))$$

◄

The operators on actions can now be lifted to the program level as follows:

Definition A.4 Variables Ignored-by Program *dIG_BY_Pr*

$$V \not\Leftarrow P = \forall a : a \in \mathbf{a}P : V \not\leftarrow a$$

◄

Due to the absence of ordering in the execution of a UNITY program, parallel composition of two programs can be modelled by simply merging the variables and actions of both programs. In UNITY parallel composition is denoted by $[]$. In [CM89] the operator is also called *program union.*

Definition A.5 Parallel Composition *dPAR*

$$P \,[]\, Q \;=\; (\mathbf{a}P \cup \mathbf{a}Q, \mathbf{ini}P \wedge \mathbf{ini}Q, \mathbf{r}P \cup \mathbf{r}Q, \mathbf{w}P \cup \mathbf{w}Q)$$

◄

Parallel composition is reflexive, commutative, associative, and has the identity element $(\emptyset, \mathsf{true}, \emptyset, \emptyset)$. We can strengthen the initial condition of a UNITY program using the folliwng function:

Definition A.6 Strengthen the initial condition *add_to_initial_cond*

$$\mathsf{a}dd\_to\_initial\_cond.I.P \;=\; (\mathbf{a}P, \mathbf{ini}P \wedge I, \mathbf{r}P, \mathbf{w}P)$$

◄

A.3 UNITY specification and proof logic

UNITY logic is used to specify the correctness expectations or properties of a UNITY program. UNITY specifications, and program properties are built from state-predicates and relations on them. Traditionally, two kinds of program properties are distinguished:

- *Safety properties* stating that some undesirable behaviour does not occur;
- *Progress properties* stating that some desirable behaviour is eventually realised.

Consequently, the UNITY logic contains two basic relations on state-predicates corresponding to these properties. For a UNITY program P and state-predicates $p, q \in$ `Expr`, these are defined by:

Definition A.7 Unless (Safety Property) *UNLESSe*

$${}_P\vdash p \;\mathsf{unless}\; q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\; a\; \{p \vee q\})$$

Definition A.8 Ensures (Progress Property) *ENSURESe*

$${}_P\vdash p \;\mathsf{ensures}\; q \;=\; ({}_P\vdash p \;\mathsf{unless}\; q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\; a\; \{q\})$$

◄

Safety properties are described by the unless relation (definition A.7). Intuitively, ${}_P\vdash p \;\mathsf{unless}\; q$ implies that once p holds during an execution of P, it remains to hold at least until q holds. Note that this interpretation gives no information whatsoever about what $p \;\mathsf{unless}\; q$ means if p never holds during an execution.

Progress properties are described by the $\mathsf{ensures}$ relation. As can be seen from Definition A.8, ${}_P\vdash p \ \mathsf{ensures}\ q$ encompasses $p\ \mathsf{unless}\ q$. Furthermore, it ensures that there exists an action that can – and, as a result of the weakly fair execution of UNITY programs, will – establish q.

A state-predicate p is a *stable* predicate in program P, if, once p holds during any execution of P, it will remain to hold forever.

Definition A.9 Stable Predicate *STABLEe*

$$ {}_P\vdash \circlearrowleft p \ = \ {}_P\vdash p\ \mathsf{unless\ false} $$
◀

A state-predicate p is a *fixed-point* of program P, if, once predicate p holds during the execution of P, the program can no longer make any progress. In other words, once p holds during the execution of P, the program will subsequently behave as skip. If p is a fixed point of program P we denote this by ${}_P\vdash \mathsf{FP}.p$.

To specify general progress properties in UNITY, the *leads-to* operator is used. It is denoted by $\mapsto$, and defined as the smallest transitive and disjunctive closure of $\mathsf{ensures}$. The precise definition and properties of $\mapsto$ can be found in [CM89]. In this technical report, we shall use Prasetya's [Pra95] variant of $\mapsto$ to specify progress properties. Prasetya's operator, called *reach*, is denoted by $\rightarrowtail$, and is defined (without overloading) as follows:

Definition A.10 Reach Operator *REACHe*
$(\lambda p, q.\ J\ {}_P\vdash p \rightarrowtail q)$ is defined as the smallest relation R satisfying:

$$(\boldsymbol{i}).\qquad \frac{p\,\mathcal{C}\,\mathbf{w}P \ \wedge\ q\,\mathcal{C}\,\mathbf{w}P \ \wedge\ ({}_P\vdash\circlearrowleft J) \ \wedge\ ({}_P\vdash J\wedge p\ \mathsf{ensures}\ q)}{R.p.q}$$

$$(\boldsymbol{ii}).\qquad \frac{R.p.q \wedge R.q.r}{R.p.r}$$

$$(\boldsymbol{iii}).\qquad \frac{(\forall i : \mathsf{evalb}.(W.i) : R.(p_i).q)}{R.(\forall i : W.i : p_i).q}$$

where W characterises a non-empty set.
◀

Intuitively, $J\ {}_P\vdash p \rightarrowtail q$ means that J is a stable predicate in P and that P can progress from $J \wedge p$ to q. Note that:

- $p \rightarrowtail q$ describes progress made through the *writable part* of program P (viz. p and q are confined by the write variables of P). However, since a program can only make progress on its write variables, this should not be a hindrance [Pra95].
- the predicate J can be used to specify the non-writable part of the program, e.g. assumptions on the environment in which the program operates.

Some properties of the UNITY operators can be found in Figure 4.

In [Pra95], Prasetya also introduces an operator to specify the more restricted form of self-stabilisation, called convergence, that allows a program to recover only from certain failures. The convergence operator is denoted by $\rightsquigarrow$ and defined in terms of $\rightarrowtail$ as follows:

Definition A.18 Convergence *CONe*

$$ J\ {}_P\vdash p \rightsquigarrow q \ \triangleq\ q\,\mathcal{C}\,\mathbf{w}P \ \wedge\ (\exists q' :: (J\ {}_P\vdash p \rightarrowtail q' \wedge q) \ \wedge\ ({}_P\vdash\circlearrowleft(J \wedge q' \wedge q))) $$
◀

A program P *converges* from p to q under the stability of J (i.e. $J\ {}_P\vdash p \rightsquigarrow q$), if, given that ${}_P\vdash\circlearrowleft J$, the program P started in p will eventually find itself in a situation where q holds and will remain to hold. Intuitively, a program P for which this holds can recover from failures which

Theorem A.11 unless COMPOSITIONALITY *UNLESSe_PAR_i*

$$({}_{P}\vdash p \ \mathsf{unless}\ q) \ \wedge \ ({}_{Q}\vdash p \ \mathsf{unless}\ q) \quad = \quad ({}_{P\,[]\,Q}\vdash p \ \mathsf{unless}\ q)$$

Theorem A.12 *dIG_BY_and_CONF_IMP_STABLEe*

$$\frac{(V \not\Leftarrow P) \wedge (p\,\mathcal{C}\,V)}{{}_{P}\vdash \circlearrowleft p}$$

Theorem A.13 $\rightarrowtail$ INTRODUCTION *REACHe_ENS_LIFT,REACHe_IMP_LIFT*

$$P, J: \ \frac{p, q\,\mathcal{C}\,\mathbf{w}P \ \wedge \ (\circlearrowleft J) \ \wedge \ ([J \wedge p \Rightarrow q] \ \vee \ (J \wedge p \ \mathsf{ensures}\ q))}{p \rightarrowtail q}$$

Theorem A.14 $\rightarrowtail$ TRANSITIVITY *REACHe_TRANS*

$$P, J: \ \frac{(p \rightarrowtail q) \ \wedge \ (q \rightarrowtail r)}{p \rightarrowtail r}$$

Theorem A.15 *REACHe_PAR_SKIPe_IMP_REACHe*

$$\frac{J \ {}_{P_1}\vdash p \rightarrowtail q \wedge \ {}_{P_2}\vdash \mathsf{FP}.(\neg q) \wedge \ {}_{P_2}\vdash \circlearrowleft J}{J \ {}_{P_1\,[]\,P_2}\vdash p \rightarrowtail q}$$

Theorem A.16 *REACHe_WHILE_r_PAR_SKIPe_r_IMP_REACHe_PAR*

$$\frac{J \ {}_{P_1}\vdash p \rightarrowtail q \wedge \ {}_{P_2}\vdash \mathsf{FP}.r \wedge \ {}_{P_2}\vdash \circlearrowleft J \wedge \ {}_{P_1}\vdash r \ \mathsf{unless}\ q \wedge (p \Rightarrow r) \wedge (r\,\mathcal{C}\,\mathbf{w}(P_1 \ [] \ P_2))}{J \ {}_{P_1\,[]\,P_2}\vdash p \rightarrowtail q}$$

Theorem A.17 *REACHe_and_STABLEe_r_PAR_SKIPe_r_IMP_REACHe_PAR*

$$\frac{J \ {}_{P_2}\vdash p \rightarrowtail q \wedge \ {}_{P_1}\vdash \mathsf{FP}.r \wedge \ {}_{P_2}\vdash \circlearrowleft r \wedge \ {}_{P_1}\vdash \circlearrowleft J \wedge \ {}_{P_1}\vdash \circlearrowleft r \wedge (p \Rightarrow r) \wedge (r\,\mathcal{C}\,\mathbf{w}(P_1 \ [] \ P_2))}{J \ {}_{P_1\,[]\,P_2}\vdash p \rightarrowtail q}$$

Figure 4: Some properties of the UNITY operators.

◀

preserve the validity of p and the stability of J. The necessity of the predicate q' in the definition of $\rightsquigarrow$ is explained in [Pra95].

Most properties of $\rightsquigarrow$ are analogous to those of $\rightarrowtail$. Since, in this technical report, we do not need these theorems directly, the reader is referred to [Pra95] for their exact characterisation.

A.4 Restricted union superposition

In [CM89], the *restricted union superposition* rule states that an action A may be added to an underlying program provided that A does not assign to the underlying variables. Here we split this into two parts:(1) defining the actual transformation of the program; (2) proving under which conditions this transformation preserves the properties of the underlying program.
Let A be an action from the universe `ACTION`, and let iA be a state-predicate describing the initial values of the superposed variables, then a program P can be refined by restricted union superposition using the transformation formally defined by:

Definition A.21 RESTRICTED UNION SUPERPOSITION *RU_superpose_DEF*

Let $A \in$ `ACTION`, $iA \in$ `Expr`, and $P \in$ `Uprog`.

$$\mathsf{RU\_S}.P.A.iA \quad = \quad P \ [] \ (\{A\}, iA, (\mathsf{assign\_vars}.A), (\mathsf{assign\_vars}.A))$$

◀

Let $P \in \texttt{Uprog}$, $A \in \texttt{ACTION}$, and $p, q, J \in \texttt{Expr}$.

Theorem A.19 PRESERVATION OF unless AND $\mathsf{ensures}$ *RU_Superpose_PRESERVES_UNLESS* *RU_Superpose_PRESERVES_ENSURES*

$$\frac{p\,\mathcal{C}\,\mathbf{w}P \;\wedge\; q\,\mathcal{C}\,\mathbf{w}P \;\wedge\; \mathbf{w}P \not\leftarrow A}{\begin{array}{l}(\;_P\vdash p\;\;\mathsf{unless}\;\;q \;\;\Rightarrow\;\; {}_{\mathsf{RU\_S}.P.A.iA}\vdash p\;\;\mathsf{unless}\;\;q)\\ (\;_P\vdash p\;\;\mathsf{ensures}\;\;q \;\;\Rightarrow\;\; {}_{\mathsf{RU\_S}.P.A.iA}\vdash p\;\;\mathsf{ensures}\;\;q)\end{array}}$$

Theorem A.20 PRESERVATION OF $\rightarrowtail$ AND $\rightsquigarrow$ *RU_Superpose_PRESERVES_REACH* *RU_Superpose_PRESERVES_CON*

$$\frac{J\,\mathcal{C}\,\mathbf{w}P \;\wedge\; \mathbf{w}P \not\leftarrow A}{\begin{array}{l}(J\;_P\vdash p\;\rightarrowtail\;q \;\;\Rightarrow\;\; J\;\;{}_{\mathsf{RU\_S}.P.A.iA}\vdash p\;\rightarrowtail\;q)\\ (J\;_P\vdash p\;\rightsquigarrow\;q \;\;\Rightarrow\;\; J\;\;{}_{\mathsf{RU\_S}.P.A.iA}\vdash p\;\rightsquigarrow\;q)\end{array}}$$

Figure 5: Restricted Union Superposition preserves properties

Theorem A.22 STRENGTHEN GUARD SYMMETRY *strengthen_guards_COMP*

$$\mathsf{strengthen\_guards}.(g_1 \wedge_* g_2).P \;=\; \mathsf{strengthen\_guards}.g_1.(\mathsf{strengthen\_guards}.g_2.P)$$

Theorem A.23 PRESERVATION OF $\rightarrowtail$ *strengthen_Pr_guards_PRESERVES_REACHe*

$$\frac{g\,\mathcal{C}\,\mathbf{w}P \;\wedge\; (\neg q \Rightarrow g)}{J\;_P\vdash p\;\rightarrowtail\;q \;\;\Rightarrow\;\; J\;\;{}_{\mathsf{strengthen\_guards}.g.P}\vdash p\;\rightarrowtail\;q}$$

Theorem A.24 *strengthen_Pr_guards_with_STABLE_PRESERVES_REACHe*

$$\frac{g\,\mathcal{C}\,\mathbf{w}P \;\wedge\; {}_P\vdash\circlearrowleft g}{J\;_P\vdash p\;\rightarrowtail\;q \;\;\Rightarrow\;\; J\;\;{}_{\mathsf{strengthen\_guards}.g.P}\vdash (p \wedge g)\;\rightarrowtail\;(q \wedge g)}$$

Figure 6: Properties of strengthening program guards

Where the function $\mathsf{assign\_vars}$, given an action A, returns the set of variables that are assigned by this action. Theorems stating that properties are preserved under restricted union superposition are listed in Figure 5. Note that instead of requiring that the superposed action A does not write to the underlying variables, it is sufficient to require that the write variables of the underlying program are ignored by the action A.

A.5 Strengthening guards

Another program transformation that preserves safety properties and, under some conditions, progress properties of the underlying program is that of strengthening guards [Sin89b, Mis90a]. Below, we define the transformation for the case where all guards of the program are strengthened with the same guard.

Definition A.25 Strengthening program guards *strengthen_guards*

$\mathsf{strengthen\_guards}.g.P = (\{\mathsf{strengthen\_guard}.g.A \mid A \in \mathbf{a}P\}, \mathbf{ini}P, \mathbf{w}P, \mathbf{r}P)$

◄

Some properties of this transformation are listed in Figure 6 below.

B Proofs of the ⨾-compositionality theorems

In this section we shall briefly discuss the verification of the compositionality theorems. Theorems 3.3 in Figure 3_7 can be proved using Restricted Union unless Preservation A.19_{14}. Theorems 3.4 in Figure 3_7 can be proved using unless Compositionality A.11_{13}. The verification of the theorems 3.7_9 and 3.8_9 will be described in the sections below.

B.1 Proof of Theorem 3.7_9

Theorem 3.7 *REACH_IMP_Simple_REACH_plus*

$$\frac{J\ {}_P{\vdash}\ p \rightarrowtail q \wedge (\forall \mathsf{pc} : \{\mathsf{pc}\} \not\Leftarrow (\mathsf{Simple}.P) : J\ \mathcal{C}\ \mathsf{pc}^{\mathsf{c}})}{J\ {}_{\mathsf{Simple}.P}{\vdash}\ p \overset{+}{\rightarrowtail} q}$$

◄

Assume the following:

$\mathbf{A}_1$: $J\ {}_P{\vdash}\ p \rightarrowtail q$
$\mathbf{A}_2$: $\forall \mathsf{pc} : \{\mathsf{pc}\} \not\Leftarrow (\mathsf{Simple}.P) : J\ \mathcal{C}\ \mathsf{pc}^{\mathsf{c}}$

we have to prove that: $J\ {}_{\mathsf{Simple}.P}{\vdash}\ p \overset{+}{\rightarrowtail} q$.

using Definition 3.5_8 this comes down to proving that:

$$\begin{array}{ll} \forall \mathsf{pc}\ n\ m\ U : & \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} (\mathsf{Simple}.P) \\ & \wedge\ U = \mathsf{semantics}.(\mathsf{Simple}.P).\mathsf{pc}.n.q \\ & \wedge\ m = \mathsf{max\_pc}.(\mathsf{Simple}.P).\mathsf{pc}.n \\ & \Rightarrow \\ & J\ {}_U{\vdash}\ (p \wedge (\mathsf{pc} = n)) \rightarrowtail (\mathsf{pc} = m) \end{array}$$

Assuming:

$\mathbf{A}_3$: $\{\mathsf{pc}\} \not\Leftarrow (\mathsf{Simple}.P)$
$\mathbf{A}_4$: $U = \mathsf{semantics}.(\mathsf{Simple}.P).\mathsf{pc}.n.q$
A_5: $m = \mathsf{max\_pc}.(\mathsf{Simple}.P).\mathsf{pc}.n$

we have to prove that: $J\ {}_U{\vdash}\ (p \wedge (\mathsf{pc} = n)) \rightarrowtail (\mathsf{pc} = m)$

rewriting $\mathbf{A}_4$ with Definitions A.21_{13}, 2.3_4 and 2.4_4 we can deduce:

$\mathbf{A}_6$: $U = U_P\ \ []\ \ U_{\mathsf{pc}}$, such that
$\mathbf{A}_7$: $U_P = \mathsf{strengthen\_guards}.(\mathsf{pc} = n \wedge \neg q).(\mathbf{a}P, \mathbf{ini}P \wedge (\mathsf{pc} = n), \mathbf{w}P \cup \{\mathsf{pc}\}, \mathbf{r}P \cup \{\mathsf{pc}\})$
$\mathbf{A}_8$: $U_{\mathsf{pc}} = (\mathsf{if}\ \ (\mathsf{pc} = n \wedge q)\ \mathsf{then}\ \mathsf{pc} := \mathsf{pc} + 1, (\mathsf{pc} = n), \{\mathsf{pc}\}, \{\mathsf{pc}\})$
$\mathbf{A}_9$: $m = (n + 1)$

Now we have to prove that: $J\ _{U_P\ \talloblong\ U_{\mathsf{pc}}}\!\vdash (p \wedge \mathsf{pc} = n) \rightarrowtail \mathsf{pc} = (n+1)$

$\Leftarrow$($\rightarrowtail$ Transitivity (A.14$_{13}$))

$$J\ _{U_P\ \talloblong\ U_{\mathsf{pc}}}\!\vdash (p \wedge \mathsf{pc} = n) \rightarrowtail (q \wedge \mathsf{pc} = n)$$
$$\wedge$$
$$J\ _{U_P\ \talloblong\ U_{\mathsf{pc}}}\!\vdash (q \wedge \mathsf{pc} = n) \rightarrowtail \mathsf{pc} = (n+1)$$

The second conjunct can be proved by $\rightarrowtail$ Introduction (A.13$_{13}$), since U_{pc} (from $\mathbf{A}_8$) ensures the required progress.

The first conjunct is decomposed as follows:

$\Leftarrow$(Theorem A.15$_{13}$)

$$J\ _{U_P}\!\vdash (p \wedge \mathsf{pc} = n) \rightarrowtail (q \wedge \mathsf{pc} = n)$$
$$\wedge$$
$$_{U_{\mathsf{pc}}}\!\vdash \mathsf{FP}.(\neg(q \wedge \mathsf{pc} = n))$$
$$\wedge$$
$$_{U_{\mathsf{pc}}}\!\vdash \circlearrowleft J$$

Since the guard of the only action of program U_{pc} (from $\mathbf{A}_8$) is $(\mathsf{pc} = n \wedge q)$, it is not hard to see that $\neg(q \wedge \mathsf{pc} = n)$ is a fixed point of U_{pc}

Using assumptions $\mathbf{A}_2$ and $\mathbf{A}_3$, we can infer that J does not depend on the variable pc (i.e. $J\ \mathcal{C}\ \{\mathsf{pc}\}^{\mathsf{c}}$). Moreover, since U_{pc} only writes to the variable pc it is straightforward to prove that U_{pc} ignores all other variables (i.e. $\{\mathsf{pc}\}^{\mathsf{c}} \not\Leftarrow U_{\mathsf{pc}}$). Consequently, we can use Theorem A.12$_{13}$ to prove that J is stable in U_{pc}.

Consequently, we are left with the proof obligation: $J\ _{U_P}\!\vdash (p \wedge \mathsf{pc} = n) \rightarrowtail (q \wedge \mathsf{pc} = n)$

Using A.22$_{14}$, we can rewrite assumption $\mathbf{A}_7$ into

$$U_P = \mathsf{strengthen\_guards}.(\mathsf{pc} = n).U'_P, \text{ where}$$

$$U'_P = \mathsf{strengthen\_guards}.(\neg q).(\mathbf{a}P, \mathbf{ini}P \wedge (\mathsf{pc} = n), \mathbf{w}P \cup \{\mathsf{pc}\}, \mathbf{r}P \cup \{\mathsf{pc}\})$$

now we proceed as follows:

$$J\ _{U_P}\!\vdash (p \wedge \mathsf{pc} = n) \rightarrowtail (q \wedge \mathsf{pc} = n)$$

$\Leftarrow$(Theorem Strengthen guards with stable predicate A.24$_{14}$)

$$J\ _{U'_P}\!\vdash p \rightarrowtail q$$
$$\wedge$$
$$(\mathsf{pc} = n)\ \mathcal{C}\ \mathbf{w}U'_P$$
$$\wedge$$
$$_{U'_P}\!\vdash \circlearrowleft (\mathsf{pc} = n)$$

Because adding variables and initial conditions to a program trivially preserves its progress properties, Theorem $\mathsf{strengthen\_guards}$ preservation of $\rightarrowtail$ (A.23$_{14}$) and assumption $\mathbf{A}_1$ can be used to establish the first conjunct. Since $\{\mathsf{pc}\} \subseteq \mathbf{w}U'_P$, Theorem $\mathcal{C}$ Monotonicity A.2$_{10}$ proves the second conjunct. Finally, Theorem A.12$_{13}$ and assumption $\mathbf{A}_3$ proves the last conjunct.

B.2 Proof of Theorem 3.8$_9$

Theorem 3.8 *REACH_SEQ_REACH*

$$\frac{q\,\mathcal{C}\,\mathbf{w}P^{+} \wedge J\ {}_{P^{+}}\vdash\ p \overset{+}{\rightarrowtail} q \ \wedge\ J\ {}_{Q^{+}}\vdash\ q \overset{+}{\rightarrowtail} s}{J\ {}_{P^{+}\,{}_{9}^{\circ}{}_{q}\,Q^{+}}\vdash\ p \overset{+}{\rightarrowtail} s}$$

◀

Assume the following:

A$_1$: $q\,\mathcal{C}\,\mathbf{w}P$
A$_2$: $J\ {}_{P}\vdash\ p \overset{+}{\rightarrowtail} q$
A$_3$: $J\ {}_{Q}\vdash\ q \overset{+}{\rightarrowtail} s$

we have to prove that: $J\ {}_{P\,{}_{9}^{\circ}{}_{q}\,Q}\vdash\ p \overset{+}{\rightarrowtail} s$.

Using Definition 3.5$_8$ this comes down to proving that:

$$\begin{array}{ll}\forall \mathsf{pc}\ n\ m\ U: & \{\mathsf{pc}\} \overset{+}{\not\Leftarrow} (P \,{}_{9}^{\circ}{}_{q}\, Q)\\ & \wedge\ U = \mathsf{semantics}.(P \,{}_{9}^{\circ}{}_{q}\, Q).\mathsf{pc}.n.s\\ & \wedge\ m = \mathsf{max\_pc}.(P \,{}_{9}^{\circ}{}_{q}\, Q).\mathsf{pc}.n\\ & \Rightarrow\\ & J\ {}_{U}\vdash\ (p \wedge (\mathsf{pc} = n)) \rightarrowtail (\mathsf{pc} = m)\end{array}$$

Assuming:

A$_4$: $\{\mathsf{pc}\} \overset{+}{\not\Leftarrow} (P \,{}_{9}^{\circ}{}_{q}\, Q)$
A$_5$: $U = \mathsf{semantics}.(P \,{}_{9}^{\circ}{}_{q}\, Q).\mathsf{pc}.n.s$
A$_6$: $m = \mathsf{max\_pc}.(P \,{}_{9}^{\circ}{}_{q}\, Q).\mathsf{pc}.n$

using Definitions 2.3$_4$ and 2.4$_4$ we can deduce that there exist U_P, U_Q, m, and k, for which:

A$_7$: $U_P = \mathsf{semantics}.P.\mathsf{pc}.n.q$
A$_8$: $k = \mathsf{max\_pc}.P.\mathsf{pc}.n$
A$_9$: $U_Q = \mathsf{semantics}.Q.\mathsf{pc}.m.s$
A$_{10}$: $m = \mathsf{max\_pc}.Q.\mathsf{pc}.k$

such that:

A$_{11}$: $U = U_P\ \,[\!]\ \, U_Q$

Now we have to prove that: $J\ {}_{U_P\,[\!]\,U_Q}\vdash\ (p \wedge \mathsf{pc} = n) \rightarrowtail \mathsf{pc} = k$

Using $\rightarrowtail$ Transitivity (A.14$_{13}$) this proof obligation can be decomposed into two proof obligations stating the progress that is established by U_P, and the progress that is established by U_Q, as follows:

$\Leftarrow$($\rightarrowtail$ Transitivity (A.14$_{13}$))

$$J\ {}_{U_P\,[\!]\,U_Q}\vdash\ (p \wedge \mathsf{pc} = n) \rightarrowtail (q \wedge \mathsf{pc} = m)$$
$\wedge$

$$J_{\ U_P \,[\!]\, U_Q} \vdash (q \wedge \mathsf{pc} = m) \rightarrowtail \mathsf{pc} = k$$

These two conjunct are proved using Theorem A.16$_{13}$ (take $r = (\mathsf{pc} < m)$) and Theorem A.17$_{13}$ (take $r = (\mathsf{pc} \geq m)$) respectively. Using Theorems 2.8$_6$, 2.10$_6$, 2.9$_6$, and 2.11$_6$, these proofs are straightforward.

Index