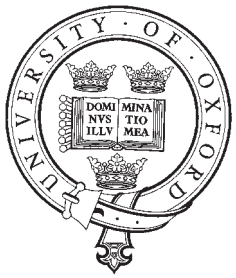
TYGER: A TOOL FOR AUTOMATICALLY SIMULATING CSP-LIKE LANGUAGES IN CSP



Thomas Gibson-Robinson
St. Catherine's College
University of Oxford

A dissertation submitted in partial satisfaction
of the requirements of the degree in

Master of Computer Science

Trinity 2010

Abstract. In [Ros08b] Roscoe outlines a class of languages, termed CSP-like languages, that can be simulated within CSP. Furthermore, Roscoe provides a construction that, given the operational semantics of an operator, gives a CSP simulation of the operator that is strongly bisimilar to the original operator. However, the construction is difficult to use, both for specifying the operational semantics and the processes that the user wishes to simulate. Furthermore, the construction is unfortunately infinite state even when simple recursive processes are used and therefore the construction is unable to be compiled by the CSP model checker, FDR.

In this Thesis we aim to solve both of these problems by, firstly, giving an adaptation to the construction that enables recursion to be successfully compiled by FDR. We then give many optimisations to the simulation to enable it to run at a reasonable speed through FDR. Lastly, we introduce Tyger, a Haskell program that is able to automate the construction of the simulation given a specification of the operational semantics of a language. Furthermore Tyger allows a user to supply process definitions using custom infix operators meaning that process definitions can be input and read easily. Lastly, Tyger also implements a type checker for the functional language that it uses meaning that many errors that would result in esoteric runtime errors in FDR can be caught at compile time.

Contents

1. Introduction

In recent years a number of languages and formalisms, such as [Ros01, Low96], have been complied into CSP [Ros97, Hoa85] by hand, a time consuming and difficult process. Furthermore, many new semantic models, such as those in [Low09, Low10, Ros09], have been proposed for CSP. Whilst FDR [Ros94, For97] has been extended to support some of these models doing so often takes long periods of time and thus denies the researcher an opportunity to fully experiment with the new model's, or languages's expressiveness.

In light of the above it would be desirable to automate the translation from the language definition, or the semantic model definition, into CSP. In [Ros08b] Roscoe provides a method that, given the *operational semantics* of the language or model, builds a simulation that is *strongly bisimilar* to the original definition. Therefore, we can simulate arbitrary languages within CSP and, furthermore, we can simulate new semantic models by simulating their operational semantics over the existing models available in FDR (for examples see Section 4).

Unfortunately, specifying the operational semantics and the processes the user wishes to compile in the required format is both time consuming and difficult. Furthermore, the simulation has the unfortunate effect of being infinite state, and thus unable to be compiled by FDR, when given recursive process definitions. We aim to solve both of these problems in this Thesis. We firstly extend the simulation to allow recursive processes to be compiled by FDR. We then describe Tyger, a tool that is able to automatically generate a corresponding CSP simulation given the operational semantics and the user processes. Furthermore, the input formats of both the operational semantics and the user processes are easy to use.

For the rest of this Thesis we assume familiarity with chapters 1–7 of [Ros97]; in particular we assume familiarity with the standard operational semantics of CSP. We further assume some knowledge of FDR, including the functional language CSP_M and some intuition as to what processes may be successfully compiled using it. Lastly, we assume basic knowledge of type systems and knowledge of Haskell.

1.1. Contributions of this Thesis.

We start in Section 2 by describing the CSP simulation that will be used; in particular we firstly describe Roscoe's machine representation of the operational semantics of operators for use within CSP_M before giving Roscoe's operator simulation. We then discuss why the simulation is infinite state when simulating recursive processes before giving a solution to the problem that works by automatically refactoring the users script into one that will be finite state (but is equivalent to the original script). Then, we describe many optimisations to the CSP model that enable it to be relatively efficiently evaluated by FDR.

In Section 3 we discuss Tyger and describe its features, the methods that it uses and the output files that it creates. In particular, in Section 3.1 we describe the input file format of the operational semantics before describing the type system that is used to type check the operational semantics. Then, in Section 3.2 we describe the input format for the users' process definitions. We then describe a type checker for CSP_M in Section 3.2.2 that is able to

type check the users' definitions. Lastly, in Section 3.2.3 we describe how the automated refactorings that were proposed for making recursion finite state are implemented within Tyger.

In Section 4 we consider a couple of examples of languages that can be simulated using Tyger and give example input files. Lastly, we discuss what we achieved in this Thesis and future research in Section 5.

2. CSP Model

In this section we firstly consider what class of languages we are able to simulate within CSP. Then, we give Roscoe's representation of both operational semantics (in Section 2.3) and the process that simulates an operator using this representation (in Section 2.4). In Section 2.5 we detail the subset of CSP-like languages that can be simulated by FDR and give adaptations necessary for a basic implementation of Roscoe's simulation to be compiled by FDR. In Section 2.5.3 we discuss recursion and give an automated technique for simulating recursion in a form that FDR can compile. Then, in Section 2.6 we give many optimisations that enable the model to be run efficiently by FDR.

Throughout this section we will need to enlarge the alphabet of events that the environment makes available. Therefore, for the remainder of the Thesis we will use:

- Σ (or *UserEvents* is CSP$_M$ scripts) to denote the set of events that the user wishes to use in their script;
- $\Sigma_0 \supset \Sigma$ (or *SystemEvents* in CSP$_M$ scripts) to be the set of all user events and special language events (such as *tick* to indicate termination in the case of CSP, or in the case of Lowe's Availability model, as defined in Section 4.1 $\{|\mathit{offer}.a \mid a \in \Sigma|\}$);
- $\Sigma_1 \supset \Sigma_0$ (or *Events* in CSP$_M$ scripts) to be the set of all events used by the simulation.

2.1. CSP-Like Languages.

Throughout this section we consider *simulation* to mean strong bisimulation.

The simulation that Roscoe defines in [Ros08b] only allows translation of some languages that he terms *CSP-like languages*. In order to define this we first introduce a definition that distinguishes different types of process arguments of operators (i.e. arguments of an operator that are processes).

Definition 2.1 ([Ros08b])**.** *A process argument P of an operator Op is said to be* **off** *iff it can perform no event (either a tau or otherwise) and* **on** *otherwise.*

Example 2.2. In the standard operational semantics of CSP in $P \Box Q$ both P and Q are **on** since both are capable of performing both tau's and visible events. This contrasts with $a \rightarrow P$ where P is **off**, since it is unable to perform any event.

Using the above definition we are then able to define *CSP-like* as follows.

Definition 2.3 ([Ros08b])**.** *An operator Op is said to be CSP-like iff all of the following conditions hold:*

(1) *The operator is positive in that the absence of one event cannot cause another event to happen. For example the following rule is not admissible:*

$$\frac{\neg(P \xrightarrow{a} P')}{Op(P) \xrightarrow{a} Op(P').}$$

(2) *No precondition of any rule depends on any argument performing multiple events (either in sequence of in parallel). For example the following rule is prohibited:*

$$\frac{P \xrightarrow{a} P' \wedge P' \xrightarrow{a} P''}{Op(P) \xrightarrow{a} Op(P'').}$$

(3) *There are tau promotions rules for all* ***on*** *arguments of Op and no tau promotion rules for any* ***off*** *arguments. For example, the following rule must be present for every* ***on*** *argument P of Op:*

$$\frac{P \xrightarrow{\tau} P'}{Op(P) \xrightarrow{\tau} Op(P')}$$

but the following rule must not be defined for any ***off*** *argument Q of Op:*

$$\frac{Q \xrightarrow{\tau} Q'}{Op(Q) \xrightarrow{\tau} Op(Q').}$$

(4) *If the precondition of a rule contains $P \xrightarrow{a} P'$ then in the resulting state of Op P will be in state P' (if present). For example, the following rule is prohibited since P changes state to P' but P' is not included in the resulting operator:*

$$\frac{P \xrightarrow{a} P'}{Op(P) \xrightarrow{a} Op(P).}$$

(5) *No* ***on*** *argument may be turned* ***off*** *(but it may be discarded). As an example the following rule is allowed for any* ***on*** *argument P:*

$$\frac{P \xrightarrow{a} P'}{Op(P, Q) \xrightarrow{a} Op'(Q)}$$

but the following rule is not allowed (supposing that Op' takes one argument that is ***off****):*

$$\frac{P \xrightarrow{a} P'}{Op(P) \xrightarrow{a} Op'(P').}$$

(6) *Every* ***off*** *argument that is an argument of the resulting state of Op is in its initial state. As an example the following rule is prohibited, assuming Q is an* ***off*** *argument of Op:*

$$\frac{Q \xrightarrow{a} Q'}{Op(Q) \xrightarrow{a} Op(Q').}$$

(7) *No* ***on*** *argument may be cloned. For example the following rule would be prohibited:*

$$\frac{}{Op(Q) \xrightarrow{\tau} Op'(Q', Q').}$$

A CSP-like language is therefore defined as a language in which every operator is CSP-like. The definition of CSP-like languages includes many existing languages including CCS [Mil82] and the π-calculus [Mil99][1].

2.2. **CSP+.** Unfortunately, it is the case that CSP is not complete in the sense that it is unable to simulate every CSP-like operator. In particular, there is no facility for an **on** process to perform a visible event and then immediately discard itself. To fix this problem an exception operator, $P\ \Theta_A\ Q$ was introduced that initially behaves as P but after having performed an event $a \in A$ behaves as Q. This can be defined as follows.

Definition 2.4 (From [Ros08b][2])**.** *The operational semantics of the exception operator are defined by the following inductive rules:*

$$\frac{P \xrightarrow{a} P'}{P\ \Theta_A\ Q \xrightarrow{a} Q}\ a \in A \qquad \frac{P \xrightarrow{b} P'}{P\ \Theta_A\ Q \xrightarrow{b} P'\ \Theta_A\ Q}\ b \notin A$$

Whilst this exception operator can be simulated up to traces and failures equivalence by interrupt, renaming and parallel[3] there is no combination that results in a strong bisimulation between the operator and the simulated operator. This was observed in [Ros08a] by noting that if any CSP operator performs an event a as a result of a process P changing state to P' then P' is always in the resulting state of the operator. Therefore, since P' is still **on**, if P' can diverge then the resulting operator can. Clearly, the exception operator does not fit this pattern since $a \rightarrow \mathbf{div}\ \Theta_{\{a\}}\ Q$ is strongly bisimilar to $a \rightarrow Q$. Hence, for the rest of this Thesis we make use of CSP+, which is defined to be CSP but with the exception operator included.

2.3. **Representation of Operational Semantics.** In this section we describe an encoding, defined by Roscoe in [Ros08b], that enables operational semantics rules such as:

$$\frac{P \,\Box\, Q \xrightarrow{a} P'}{P \xrightarrow{a} P'}\ a \in \Sigma$$

to be written in a form that can be processed automatically. Therefore, given an operator α we define $n(\alpha)$ as the number of **on** arguments and $I(\alpha)$ as the number of **off** arguments. We also make the decision to consider operators such as $\underset{A}{\|}$ as a *family* of operators with one for each $A \subseteq \Sigma$. This

[1] In fact CCS is not CSP-like as in the expression $P + Q$ if either P or Q performs a *tau* then it resolves the choice. However, in [Ros08b] Roscoe shows how to use the simulation provided in spite of this.

[2] This was originally defined in [Ros08a] but was generalised to the form we present here in [Ros08b].

[3] For example, consider the following simulation, $Exception(P, A, Q)$:

$$Exception(P, A, Q) \mathrel{\widehat{=}} \left(P \triangle Run(Prime(\Sigma)) \underset{\Sigma \cup Prime(\Sigma)}{\|} R\right)[[UnPrime]]$$

$$R \mathrel{\widehat{=}} \Box_{x \in A}\, x \rightarrow Q[[Prime]] \,\Box\, \Box_{x \notin A}\, x \rightarrow R.$$

where $Prime$ is a function that renames every event $a \in \Sigma$ to a' and $UnPrime$ is its inverse.

has the convenient consequence that the only arguments of an operator are the **on** and **off** processes and therefore we assume that each operator α has a sequence of **on** processes and a sequence of **off** processes as its arguments. Furthermore, we index the **on** arguments by the set $\{1 \ldots\}$ and **off** arguments by $\{\ldots -1\}$ to make it explicitly clear whether an argument is **on** or **off**.

We can now define the operational semantics of an operator α to be represented via a set of tuples of the form $(\phi, x, \beta, f, \psi, \chi, B)$ where:

- ϕ: is a map from $\{1 \ldots n(\alpha)\}$ to events, in particular if $\phi(x) = b$ it means that the x^{th} **on** process should perform the event b;
- x: is the resulting event that α performs;
- β: is the resulting operator;
- f: is a map from $\{1 \ldots k\}$ (where k is equal to the number of processes turned **on** by this rule) to $\{-I(\alpha) \ldots -1\}$ where if $f(x) = y$ then the x^{th} newly turned **on** process is a copy of the y^{th} **off** process;
- ψ: is a map from $\{1 \ldots n(\beta)\}$ to $\{1 \ldots n(\alpha) + k\}$ where $\psi(y) = x$ indicates that the y^{th} argument of β is either, if $x \leq n(\alpha)$ then the x^{th} **on** argument of the current operator, or otherwise the $x - n(\alpha)^{th}$ newly turned **on** operator;
- χ: is a map from $\{-I(\beta) \ldots -1\}$ to $\{-I(\alpha) \ldots -1\}$where $\chi(y) = x$ means that the y^{th} **off** argument of β is the x^{th} **off** argument of α;
- B: is the set of **on** processes that are discarded[4].

We note that all the above maps are partial functions and therefore they can be represented by sets of pairs.

As this representation is not necessarily easy to understand we now define a function *Rules* that, given a CSP operator returns the set of rules according to the standard CSP operational semantics (for a subset of the operators). In the following we assume that there exists a special event *tau* that is contained in Σ_1 (note that *tau* is not equal to τ since τ is in the semantics of the language whereas *tau* is in the syntax). Given this we can define the function *Rules* as follows:

$$
\begin{aligned}
&\mathit{Rules}(\mathit{Identity}) \mathrel{\widehat{=}} \\
&\quad \{(\{(1, a)\}, a, \mathit{Identity}, \{\}, \{(1, 1)\}, \{\}, \{\}) \mid a \in \Sigma_0\} \\
&\mathit{Rules}(\mathit{Prefix}.a) \mathrel{\widehat{=}} \\
&\quad \{(\{\}, a, \mathit{Identity}, \{(1, -1)\}, \{(1, 1)\}, \{\}, \{\})\} \\
&\mathit{Rules}(\mathit{ExtChoice}) \mathrel{\widehat{=}} \\
&\quad \{(\{(1, a)\}, a, \mathit{Identity}, \{\}, \{(1, 1)\}, \{\}, \{2\}) \mid a \in \Sigma\} \\
&\quad \cup \{(\{(2, a)\}, a, \mathit{Identity}, \{\}, \{(1, 2)\}, \{\}, \{1\}) \mid a \in \Sigma\} \\
&\mathit{Rules}(\mathit{IntChoice}) \mathrel{\widehat{=}} \\
&\quad \{(\{\}, \mathit{tau}, \mathit{Identity}, \{(0, -1)\}, \{(1, 1)\}, \{\}, \{\}) \mid a \in \Sigma\} \\
&\quad \cup \{(\{\}, \mathit{tau}, \mathit{Identity}, \{(0, -2)\}, \{(1, 1)\}, \{\}, \{\}) \mid a \in \Sigma\}
\end{aligned}
$$

[4]This was not in Roscoe's original model but is added for speed; this is discussed further in Section 2.6.1.

$$\begin{aligned}
&Rules(Parallel.A) \mathrel{\widehat{=}} \\
&\quad \{(\{(1,a),(2,a)\}, a, Parallel.A, \{\}, \{(1,1),(2,2)\}, \{\}, \{\}) \mid a \in A\} \\
&\quad \cup \{(\{(1,a)\}, a, Parallel.A, \{\}, \{(1,1),(2,2)\}, \{\}, \{\}) \mid a \notin A\} \\
&\quad \cup \{(\{(2,a)\}, a, Parallel.A, \{\}, \{(1,1),(2,2)\}, \{\}, \{\}) \mid a \notin A\} \\
&Rules(Hide.A) \mathrel{\widehat{=}} \\
&\{(\{(1,a)\}, tau, Hide.A, \{\}, \{(1,1)\}, \{\}, \{\}) \mid a \in A\} \\
&\quad \cup \{(\{(1,a)\}, a, Hide.A, \{\}, \{(1,1)\}, \{\}, \{\}) \mid a \notin A\}.
\end{aligned}$$

Note the inclusion of an identity operator above; this operator has a special status in terms of recursion (see Definition 2.6) and is therefore always included. For the rest of this section we assume that a function *Rules* has been defined for the language that we are attempting to model.

2.4. **Basic Model.** In this section we define the process $Operator(\alpha, onProcs, offProcs)$, that Roscoe developed in [Ros08b], that simulates, up to strong bisimulation, the operational semantics of α. The general form that the process will take is[5]:

$$\begin{aligned}
&Operator(\alpha, onProcs, offProcs) \mathrel{\widehat{=}} \\
&\quad \Big(\|_{(n,P) \in onProcs} (Harness(P,n), AlphaProc(n)) \\
&\qquad \| \\
&\qquad \cup_{n \in \{1 \ldots card(onProcs)\}} AlphaProc(n) \\
&\qquad Reg(\alpha, card(onProcs), id_{\{1 \ldots card(onProcs)\}}, id_{\{-card(offProcs)..-1\}}) \\
&\quad \Big)[[Rename]] \setminus \{tau\}.
\end{aligned}$$

It should be clear from the above that the left hand side of the parallel does not depend on the current operator, only the **on** processes. Therefore, only the regulator, *Reg*, differs between $Operator(ExternalChoice, \langle P, Q \rangle, \langle\rangle)$ and $Operator(Parallel.\Sigma, \langle P, Q \rangle, \langle\rangle)$. Thus, in order to ensure that we can differentiate between the two operators we need to rename the events of each **on** process to allow the processes to synchronise together in any possible way. The regulator could then only allow those events that correspond to rules of the current operator to occur. For example, in the case of the external choice it would allow either process *0* or *1* to perform an event after which the opposite process would turn **off**. However, in the case of parallel (synchronising on Σ) processes *0* and *1* would have to synchronise together to perform the a. Therefore, the general event form that will be used in the operator simulation will be a tuple (ϕ, x, B) where:

- ϕ: is a map from **on** process to events; in particular if $\phi(x) = y$ the x^{th} process should perform the event y;
- x: is the resulting event;
- B: is the set of **on** processes that should be discarded.[6]

[5]For a derivation of this see [Ros08b] where a step-by-step construction is made with each iteration being able to simulate more operators than the previous version.

[6] Roscoe's original construction renamed events to a 5-tuple with two extra components m and f where:

Therefore, we can simply define *Rename* as the function that renames (ϕ, x, B) to x. Furthermore, $AlphaProc(n)$ can be defined to be the set of all (ϕ, x, B) such that n is either in $dom(\phi)$ or B.

We now consider the definition of $Harness(P, n)$; the harness needs to run the process P as though it was the n^{th} **on** process, renaming every event that it performs into one of the above form. It also needs to turn this process **off** when an event of the form (ϕ, x, B) where $n \in B$ is performed. Hence, we can define $Harness(P, n)$ as follows:

$$Harness(P, n) \mathrel{\widehat{=}} ((P[[Prime]] \; \Theta_{\{x' | x \in \Sigma_0\}} \; Stop) \\ \triangle \; off \rightarrow Stop)[[HarnessRename]]$$

where *off* is a fresh event and *Prime* maps every event $x \in \Sigma_0$ to both x and x'. *Rename* is therefore the relation defined by:

$$a \; Rename \; (\phi, x, B) \Leftrightarrow \phi(n) = a \wedge n \notin B$$
$$a' \; Rename \; (\phi, x, B) \Leftrightarrow \phi(n) = a \wedge n \in B$$
$$off \; Rename \; (\phi, x, B) \Leftrightarrow n \notin dom(\phi) \wedge n \in B.$$

Hence, the event *off* is used when the process is discarded by an event performed by another process (e.g. external choice); the primed event a' is used when the process performs the event and discards itself in the process (e.g. the exception operator); the event a is used only when the process performs an event and stays **on** (e.g. parallel).

We now consider the definition of the regulator. In our simulation the regulator has three principle responsibilities: it must ensure that only the events that are allowed by the current operator can be performed by the processes; it must track which process on the left hand side corresponds to which process of the current operator; lastly it must turn **on** processes according to the executed rule. Hence, the regulator must take several parameters as follows:

- λ: is the current operator;
- m: is the current number of processes that have been started;
- Ψ: is a map from $\{1 \ldots n(\lambda)\}$ to $\{1 \ldots m\}$; in particular if $\Psi(x) = y$ it means that the the x^{th} **on** argument of λ is the y^{th} **on** argument;
- χ: is a map from $\{1 \ldots I(\lambda)\}$ to $\{1 \ldots I(\alpha)\}$ where α is the starting operator (i.e. the operator for which $Operator(\alpha, \ldots)$ was called); in particular if $\chi(x) = y$ it means that the x^{th} **off** argument of λ is the y^{th} **off** argument of α.

– m: is the current count of **on** processes;
– f: is a map from $\{1 \ldots k\}$ to **off** processes and if $f(x) = y$ then the x^{th} turned **on** process should be the y^{th} **off** process.

Roscoe required these two components since he defined an extra process that would use m and f to start up new processes. However, in our simulation we make the regulator perform this role, as shall be seen later.

We can then define the regulator, $Reg(\lambda, m, \Psi, \chi)$ as follows:

$$Reg(\lambda, m, \Psi, \chi) \mathrel{\widehat{=}} \Box_{(\phi,x,\beta,f,\psi,\chi',B)\in Rules(\lambda)} (\phi \circ \Psi^{-1}, x, \{\Psi(x) | x \in B\}) \rightarrow \left(\begin{array}{l} \|_{n\in\{m\ldots m+card(f)\}} \\ \qquad (Harness(offProcs(\chi(f(n-m)))), AlphaProc(n)) \\ \qquad \bigcup_{n\in\{1\ldots m+card(f)\}} AlphaProc(n) \| \bigcup_{n\in\{m+card(f)\ldots\}} AlphaProc(n) \\ \qquad Reg(\beta, m+card(f), (\Psi \cup id_{m+1,\ldots,m+card(f)}) \circ \psi, \chi \circ \chi') \end{array} \right).$$

Therefore it follows that the state of *Operator* having performed the event (ϕ, x, B) is equivalent to:

$$\left[\left| \right|_{(n,P)\in onProcs} (Harness'(P,n), AlphaProc(n)) \right. \\ \| \\ \bigcup_{n\in\{1\ldots m\}} AlphaProc(n) \\ \left(\begin{array}{l} \|_{n\in\{m\ldots m+card(f)\}} \\ \qquad (Harness(offProcs(\chi(f(n-m)))), AlphaProc(n)) \\ \qquad \bigcup_{n\in\{1\ldots m+card(f)\}} AlphaProc(n) \| \bigcup_{n\in\{m+card(f)\ldots\}} AlphaProc(n) \\ \qquad Reg(\beta, m+card(f), (\Psi \cup id_{m+1,\ldots,m+card(f)}) \circ \psi, \chi \circ \chi') \end{array} \right) \\ \left. \right] [[Rename]] \setminus \{tau\}.$$

where m is equal to $card(onProcs)$ and $Harness'(P, n)$ is the state of *Harness* having performed (ϕ, x, B). Thus it follows that the above is equivalent to $Operator(\beta, \ldots)$.

2.5. **Implementation in Machine-Readable CSP.** In this section we firstly consider what subset of CSP-like operators we can simulate within FDR. We then, in Section 2.5.2, consider adaptations that are required in order to express the simulation in machine-readable CSP before giving a basic implementation. In Section 2.5.3 we consider adaptations to the simulation that ensure that the simulation is finite state where possible, thus enabling recursive processes to be simulated. Lastly, in Section 2.6 we consider many optimisations to the basic model that enable the simulation to be run in an acceptable time.

2.5.1. *Operators that can be simulated within FDR.* It should be clear that the major problem with the above construction is that the alphabets involved are infinite; in particular the synchronisation alphabet of the regulator $AlphaProc(n)$ is infinite. This is because it is possible that an infinite number of processes can be turned **on** by an operator, or by a collection of mutually recursive operators. Therefore, since the **on** processes have to be able to be synchronised in any possible way $AlphaProc(n)$ is necessarily infinite. As an example, consider the following operator, $Farm(P)$:

$$\frac{}{Farm(P) \xrightarrow{start} P \;|||\; Farm(P)}$$

This operator, as defined above, will turn **on** an infinite number of processes. However, it is (assuming the standard CSP semantics) equivalent to the following CSP process:

$$Farm(P) = start \rightarrow (P \;|||\; Farm(P))$$

which is infinite state. Therefore, it follows that the operator *Farm* would result in infinite state processes that could not be compiled by FDR. Thus, we make the restriction that operators such as $Farm(P)$ are not allowed. Formally, we disallow any operators that can possibly turn **on** an infinite number of processes. (Note that this is not checked automatically, but it is assumed.)

2.5.2. *The Implementation.* We now consider the actual implementation of *Operator* in machine-readable CSP. The first thing that is done is to declare a channel, *renamed* along which the (ϕ, x, B) events are sent (this is necessary since all events must be sent along channels in FDR). Thanks to the above restriction it is possible to bound the maximum number of processes turned **on** to some constant N that can be statically computed. Therefore, assuming that we represent partial functions by sets of pairs, the type of the channel can be calculated as follows:

$$\begin{aligned}\{(\phi, x, B) \mid \phi \subseteq \{(x, e) \mid x \in \{0 \ldots N-1\}, e \in \Sigma_0\},\\ x \in \Sigma_0 \cup \{tau\}, B \subseteq \{0 \ldots N-1\},\\ card(dom(\phi)) = card(\phi)\}.\end{aligned}$$

The last line of the set comprehension ensures that a process is expected to perform at most one event. We note that whilst this set may be very large it is certainly computable. After this has been done the rest of the translation is purely mechanical and so we give the machine-readable CSP for an unoptimised version in Listing 2.5.

Listing 2.5. Here we define a basic machine-readable CSP version of Roscoe's simulation, as defined above. However, as this is the simplest possible implementation possible it is very slow. We discuss this more in Section 2.6.

```
N = ...
-- In this simulation there are no special events and thus we
-- omit SystemEvents
UserEvents = ...

-- Functional components
zip(<>, _) = <>
zip(_, <>) = <>
zip(<x>^xs, <y>^ys) = <(x,y)>^zip(xs, ys)

-- Partial Funcitons
domain(f) = {x | (x,_) <- f}
identity(lb,ub) = {(x,x) | x <- {lb..ub}}
inverse(f) = {(a,b) | (b,a) <- f}
-- equal to (fs1 o fs2) (also copes with fs2 being not defined on all
-- elements of fs1's image).
```

```
compose(fs1, fs2) =
  {(a, apply(fs1, b)) | (a, b) ← fs2, member(b,domain(fs1))}
apply(f, x) =
  let extract({x}) = x
  within extract({a | (x', a) ← f, x == x'})
applySeq(f, x) =
  let extract(<x>) = x
  within extract(<a | (x', a) ← f, x == x'>)

-- Operator Simulation
channel tau
channel off
channel renamed :
  {(phi,x,B) | phi ← Set({(x,e) | x ← {0..N-1}, e ←
UserEvents}),
      x ← union(UserEvents, {tau}), B ← Set({0..N-1}),
      card(domain(phi)) == card(phi)}
channel prime : UserEvents

datatype Operators =
  Op_Prefix.UserEvents

Rules(Op_Prefix.x) =
  { ({}, x, Op_Identity, {(0,-1)}, {(0,0)}, {}, {}) }
...

Operator(alpha, onProcs, offProcs) =
  let
    onProcMap = zip(<0..>, onProcs)
    offProcMap = zip(<(-length(offProcs))..-1>, offProcs)

    Harness(P, n) =
      ((P⟦x ← x, x ← prime.x | x ← UserEvents⟧
        [| {| prime |} ▷ STOP)
       △ off → STOP)
       ⟦x ← y | (x,y) ← RenamingsForProc(n)⟧
    RenamingsForProc(n) =
      {(x,renamed.y) | x ← union(UserEvents, {| prime, off |}),
            renamed.y ← AlphaProc(n),
            Rename(n, x, y)}
    Rename(n, x, (phi, _, B)) =
      if member(n, domain(phi)) then
        if member(n, B) then (x == (prime.apply(phi,n)))
        else (x == apply(phi,n))
      else if member(n, B) then (x == off) else false

    AlphaProc(n) =
      { renamed.(phi, x, B) | renamed.(phi, x, B) ← {| renamed |},
                  member(n, domain(phi)) or member(n, B)}
    AlphaProcs(n,m) = Union({AlphaProc(i) | i ← {n..m}})

    Reg(alpha, m, Psi, chi) =
```

```
      □ (phi, x, beta, f, psi, chi', B) : Rules(alpha) •
        renamed.(compose(phi,inverse(Psi)), x,
            {apply(Psi,x) | x ← B}) →
          ( ||| n : {m..m+card(f)-1} • [AlphaProc(n)]
            Harness(applySeq(offProcMap, apply(chi, apply(f, n-m))), n)
          [| AlphaProcs(0,m+card(f)-1) |]
          Reg(beta, m+card(f),
            compose(union(Psi, identity(m,m+card(f)-1)), psi),
            compose(chi,chi')))
  within
    (( ||| n : {0..length(onProcMap)-1} • [AlphaProc(n)]
      Harness(applySeq(onProcMap, n), n))
    [| AlphaProcs(0, length(onProcMap)-1) |]
    Reg(alpha, length(onProcMap),
      identity(0, length(onProcMap)-1),
      identity(-length(offProcMap),-1)))
    ⟦renamed.(phi,x,B) ← x | renamed.(phi,x,B) ← {|renamed|}⟧
    \ {tau}

Prefix(x, P) = Operator(Op_Prefix.x, <>, <P>)
...
```

2.5.3. *Recursion.* As mentioned previously recursion within the simulation does not work within FDR; more precisely it causes FDR to loop forever attempting to compile the simulated process. To see why this is the case consider simulating (using the standard CSP operational semantics) $P = a \rightarrow P$. This would be simulated by the process $P = Operator(Prefix.a, \langle\rangle, \langle P \rangle)$. We calculate the definition of this as FDR would below, writing HR for the renaming done by the *Harness* and R for the renaming done by *Operator*. Furthermore, we note that since neither *Prefix* nor *Identity* discards any arguments we can simplify $Harness(P, n)$ to $P[[HR]]$.

$$\begin{aligned}
P &= Operator(Prefix.a, \langle\rangle, \langle P \rangle) \\
&= Reg(Prefix, ...)[[R]] \setminus \{tau\} \\
&= ((\{\}, a, \{\}) \rightarrow (Harness(P, 0) \underset{...}{\parallel} Reg(Identity, ...)))[[R]] \setminus \{tau\} \\
&= a \rightarrow (Harness(P, 0) \underset{...}{\parallel} Reg(Identity, ...))[[R]] \setminus \{tau\} \\
&= a \rightarrow (P[[HR]] \underset{...}{\parallel} Reg(Identity, ...))[[R]] \setminus \{tau\}
\end{aligned}$$

Thus, P_i, the process that is P unwrapped i times, is equivalent (by the unique fixed point principle) to:

$$\begin{aligned}
P_0 &= P \\
P_{i+1} &= a \rightarrow ((P_i[[HR]] \underset{..}{\parallel} Reg(Identity, ...))[[R]] \setminus \{tau\})
\end{aligned}$$

Clearly FDR will be unable to compile the limit of the above sequence of processes since an infinite number of copies of $Reg(Identity, \ldots)$ would be put in parallel.

A solution to the problem described above would be to periodically *throw away* the collection of identity operator regulators that have been spawned.

This would be done in such a way to ensure that we eventually have a transition to P which would allow FDR to recognise the recursion. To see how this works consider the following simulation of $P = a \to P$:

$$\begin{aligned} PSim &\mathrel{\widehat{=}} Loop(Operator(Prefix.a, \langle\rangle, \langle CallPSim\rangle)) \\ CallPSim &\mathrel{\widehat{=}} callPSim \to Stop \\ Loop(Q) &\mathrel{\widehat{=}} Q \ \Theta_{\{callPSim\}} \ PSim. \end{aligned}$$

We then calculate the value of $PSim$ as follows:

$$\begin{aligned} PSim &= Loop(Operator(Prefix.a, \langle\rangle, \langle CallPSim\rangle)) \\ &= Operator(Prefix.a, \langle\rangle, \langle CallPSim\rangle) \ \Theta_{\{callPSim\}} \ PSim \\ &= ((\{\}, a, \{\}) \to Harness(CallPSim, 0) \underset{\ldots}{\parallel} Reg(Identity, \ldots))[[R]] \setminus \{tau\} \\ &\qquad \Theta_{\{callPSim\}} \ PSim \\ &= a \to \Big[(Harness(CallPSim, 0) \underset{\ldots}{\parallel} Reg(Identity, \ldots))[[R]] \setminus \{tau\} \\ &\qquad \Theta_{\{callPSim\}} \ PSim\Big] \\ &= a \to \Big[(CallPSim[[HR]] \underset{\ldots}{\parallel} Reg(Identity, \ldots))[[R]] \setminus \{tau\} \\ &\qquad \Theta_{\{callPSim\}} \ PSim\Big] \\ &= a \to \Big[((callPSim \to Stop)[[HR]] \underset{\ldots}{\parallel} Reg(Identity, \ldots))[[R]] \setminus \{tau\} \\ &\qquad \Theta_{\{callPSim\}} \ PSim\Big] \\ &= a \to [((\{(0, callPSim)\}, callPSim, \{\}) \to \\ &\qquad Stop[[HR]] \underset{\ldots}{\parallel} Reg(Identity, \ldots))[[R]] \setminus \{tau\} \ \Theta_{\{callPSim\}} \ PSim\Big] \\ &= a \to \Big[callPSim \to ((Stop[[HR]] \underset{\ldots}{\parallel} Reg(Identity, \ldots))[[R]] \setminus \{tau\}) \\ &\qquad \Theta_{\{callPSim\}} \ PSim\Big] \\ &= a \to callPSim \to PSim \end{aligned}$$

Hence, FDR is able to compile $PSim$; furthermore, $PSim \setminus \{callPSim\} \equiv_{FD} P$ (in fact they are strongly bisimilar except for the extra τ that is introduced when recursing).

We now consider how to generalise this to arbitrary processes and arbitrary operational semantics. The first refactoring will be as above; namely every call inside a recursive process to another recursive process P will be replaced with a *callProc*.P event. For the remainder of this section we make the assumption that processes can be transmitted over channels (in Section 3.2.3 we discuss how to actually simulate this). In order to define the generalisation we define a process WrapThread, analogous to *Loop* above, as follows:

```
WrapThread(proc) =
  (proc [| {| callProc |} ▷ callProc?proc' → WrapThread(proc'))
    \ {| callproc |}.
```

Unfortunately, at this time, FDR does not support such a generalised exception operator. However, we can simulate one as follows:

```
channel callProc, startProc : Proc

Call(proc) = callProc.proc → STOP
-- The following makes it easier to redefine this in Section 3.2.3
GetProc(proc) = proc

WrapThread(proc) =
  let
    RecursionRegulator =
      callProc?p → startProc!p → RecursionRegulator
    □ □ e : SystemEvents • e → RecursionRegulator
    Thread(proc) =
      GetProc(proc)
      [| {| callProc |} ▷
      startProc?proc' → Thread(proc')
  within
    diamond[7](Thread(proc)
      [| union(SystemEvents,{| callProc, startProc |}) |]
    RecursionRegulator)
    \ {| startProc, callProc |}
```

For the remainder of this section we refer to a process as *wrapped* iff it is of the form $WrapThread(Q)$ for some process Q. Furthermore, note that no *callProc* events can propagate out of a wrapped process. As an example of how the simulation can be used note that P above is simulated by the process $PSim = WrapThread(a \rightarrow CallProc(PSim))$.

Note that since the *callProc* events are not in *UserEvents* it follows that we will have to add new clauses to the operational semantics of the operators to allow them to propagate. However, it is easy to see that in all but the case of the identity operator that if a *callProc* event were to propagate then information would be lost. For example consider the following processes:

```
P = a → P □ R
R = b → R.
```

[7]The use of the compression function diamond here yields excellent results in terms of performance. Interestingly, using sbisim does not yield any further improvement, as is normally the case when using diamond.

Here, since the call to R on the right hand side of the external choice is replaced by a *callProc* event it follows that the recursion would resolve the external choice which is clearly incorrect. Hence, only the identity operator has the following rule added:

$$\frac{P \xrightarrow{callProc.p} P'}{Identity(P) \xrightarrow{callProc.p} Identity(P')}\ p \in Proc$$

However, this too introduces a problem with operators such as external choice. For example, consider the simulation of the processes P and R as defined above. If we were to simulate them as suggested above the right hand side of the external choice would be replaced by a *callProc* event. Since these are not allowed to propagate through the external choice it follows that only the choice of the left hand side would be presented which is clearly incorrect. The only solution to this is to *inline* the definition of R as follows:

```
P = a → P □ b → R
R = b → R.
```

We term an argument of an operator *finalised* iff the first event may not be a *callProc* event. Note that this can be generalised to *n-finalised* where none of the first n events may be *callProc* events. For example, consider the following operator, $RenameN(N, R, P)$ that renames the first N events of P:

$$\frac{P \xrightarrow{a} P'}{RenameN(N, R, P) \xrightarrow{b} RenameN(N-1, R, P')}\ (a, b) \in R \wedge N > 0$$

$$\frac{}{RenameN(0, R, P) \xrightarrow{\tau} P}$$

and the following processes:

```
P = Rename(N, {(a,b)}, Q) □ c → P
Q = a → Q
```

Since both P and Q are recursive the call to Q from P would be replaced by a *callProc* event. However, since these may not propagate through the $RenameN$ operator it follows that the simulation of P would deadlock. Again, the only solution would be to inline the definition of Q; in particular the first N events would need to be inlined as follows:

```
P = RenameN(N, {(a,b)}, a → a → ... → a → Q) □ c → P
Q = a → Q
```

Therefore, we re-write all process definitions so that the correct number of events are inlined[8].

The last re-writing that we need to consider concerns operators that are infinitely recursive, such as CSP's parallel operator. What we need to be able to do is to calculate which arguments of an operator may perform infinitely many events without ever coming in scope of an identity operator. We term such arguments *infinitely recursive* and all other arguments *finitely recursive*. For example:

- Both arguments of parallel are infinitely recursive;

[8]In Section 3.2.3 we discuss how this is implemented in Tyger

- Both arguments of external choice are 1-finalised and thus finitely recursive;
- The left argument of interrupt is infinitely recursive but the right argument is 1-finalised and thus finitely recursive;
- The left argument of exception is infinitely recursive but the right is finitely recursive (and 0-finalised).

Clearly, if an argument P of an operator Op may perform an infinite number of events without Op evolving into an identity operator (with P as its argument) then we must ensure that P cannot perform *callProc* events since these cannot propagate through any operator other than the identity operator. We can formalise the above as follows.

Definition 2.6. *An argument P of an operator Op is* infinitely recursive *iff there exists an infinite chain of the form:*

$$Op(\ldots, P, \ldots) \xrightarrow{a_1} Op_1(\ldots, P_1, \ldots) \xrightarrow{a_2} Op_2(\ldots, P_2, \ldots)\ldots$$

where for infinitely many i, $a_i \in \Sigma$, none of the Op_i is the identity operator and where P eventually performs infinitely many events (i.e. for infinitely many i, $P_i \neq P_{i+1}$). An argument P of an operator is finitely recursive *iff there exists no chain of the above form (i.e. the process always eventually ends up in scope of an identity operator, or, in any such chain P is* ***off****). Note that an operator is n-finalised for some n iff it is finitely recursive.*

Therefore, following our above intuition we ensure that any argument of an operator that is infinitely recursive is wrapped. For example, consider simulating the following process P:

```
P = Q [| Events |] RUN(Events)
Q = a → Q.
```

Furthermore, consider the simulation in which the simulated Q is not wrapped:

```
PSim = QSim [| Events |] RUN(Events)
QSim = Operator(Op_Prefix.a, <>, <CallProc(QSim)>).
```

Note that QSim is equivalent (by the unique fixed point principle) to the process QSim' = a → callProc.QSim → QSim'. Hence, since *callProc* events cannot propagate through the parallel composition it follows that PSim would be equivalent to a → STOP. Therefore, we ensure that every infinitely recursive argument of an operator is wrapped since this ensures that no *callProc* events can propagate.

Thus, in summary the re-writing of process definitions that must be done is as follows:

(1) Every call from a recursive process to another recursive process is replaced by the corresponding *callProc* event;
(2) Every call from a non-recursive process to a recursive process calls the wrapped version;
(3) Every infinitely recursive argument of an operator is wrapped;
(4) Every finitely recursive argument is inlined by the correct amount.

In Section 3.2.3 we discuss how these are automated within Tyger.

It is worth noting at this point that the simulation we provide now is not strongly bisimilar to the original as there are two taus created by a

recursive call, rather than the standard one. However, this is not an issue when we consider CSP since all the standard models equate the processes Q and $\cdot \xrightarrow{\tau} Q$.

2.6. **Optimisations.** In this section we consider a number of optimisations that were made to the CSP model in order to make it run efficiently within FDR.

As a running example throughout this section we consider the runtime of a simulated variant of the Dining Philosophers in the standard CSP operational semantics. All runtimes that we refer to concern the amount of time that it took to verify the two assertions in the following code, where $N + 1$ is the number of Philosophers.

```
channel sitdown, getup, eat : {0..N}
channel pickup, putdown : {0..N}.{0..N}
UserEvents = {| sitdown, getup, eat, pickup, putdown |}

mplus(i, j) = (i + j) % (N+1)

MyFork(id) =
  ExternalChoice(
    Prefix(pickup.id.id, Prefix(putdown.id.id, Call(P_Fork.id))),
    Prefix(pickup.mplus(id,1).id,
      Prefix(putdown.mplus(id,1).id, Call(P_Fork.id)))
  )
MyPhilosopher(id) =
  Prefix(sitdown.id,
    Prefix(pickup.id.id,
    Prefix(pickup.id.mplus(id,1),
    Prefix(eat.id,
    Prefix(putdown.id.mplus(id,1),
    Prefix(putdown.id.id,
    Prefix(getup.id, Call(P_Philosopher.id))))))))
Main =
  let
    phils =
      ReplicatedInterleave(<WrapThread(P_Philosopher.id>)
                  | id ← <0..N>>)
    forks =
      ReplicatedAlphaParallel(<WrapThread(P_Fork.i) | i ←
<0..N>>,
                  <AlphaFork(i) | i ← <0..N>>)
  within
    Parallel(phils, forks, {| pickup,putdown |})

-- Original process
Philosopher(id) =
  sitdown.id → pickup.id.id → pickup.id.mplus(id,1) → eat.id →
  putdown.id.mplus(id,1) → putdown.id.id → getup.id →
Philosopher(id)
Fork(id) =
  pickup?phil : {id, mplus(id,1)} !id → putdown!phil!id →
Fork(id)
```

```
AlphaFork(id) =
  {pickup.id.id, putdown.id.id, pickup.mplus(id,1).id,
    putdown.mplus(id,1).id}
System =
  (||| id : {0..N} • Philosopher(id))
  [| {| putdown, pickup |} |]
  (|| id : {0..N} • [AlphaFork(id)] Fork(id))

assert Main ⊑FD System
assert System ⊑FD Main
```

All timing tests were carried out on a 3Ghz Dual-core Linux machine with 2GB of RAM using a pre-release version of FDR 2.91 (that contains the exception operator and one other special feature that we discuss later).

2.6.1. *Precomputation of Discardable Args.* As mentioned in Section 2.3 we explicitly include the processes that are discarded in our representation of the operational semantics. This is because if we were to define a function $discard(rule, op)$ that computed which arguments should be discarded by the current operator we would have to traverse every rule in the operator. Since there are a polynomial number of rules (in terms of Σ_0) for each operator and since this function would have to be called many times by the regulator this would result in a significant slow down. Hence, since it is easy to statically compute which processes are discarded the discarded process are included in the rules.

2.6.2. *Sets – Computing the Transitive Closure of the Events.* The basic implementation that we consider is as described in Listing 2.5. Unfortunately, it is so inefficient that even with only 2 philosophers FDR consumes over 2GB of memory. Therefore, no figures are available for its performance. The major factor contributing to this is the size of the channel *renamings*; it is of size $O(card(\Sigma_0) \times 2^N \times 2^{N \times card(\Sigma_0)})$. Therefore, computing the functions $RenamingsForProc(n)$ and $AlphaProc(n)$ will be very slow.

Whilst *renamings* is very large, in the case of most operators the majority of the events in the *renamings* channel are never required. For example, in the case of CSP's parallel operator over a set A only events of the form $(\{(0,a),(1,a)\}, a, \{\}), (\{(0,b)\}, b, \{\}), (\{(1,b)\}, b, \{\})$ are needed, where a ranges over A and b ranges over $\Sigma \setminus A$. In fact, all standard CSP operators only use (at most) $O(card(\Sigma_0)^2)$ events. Therefore a significant gain could be achieved by calculating $RenamingsForProc(n)$ and $AlphaProc(n)$ using the subset of *renamings* that corresponds to the events that could be used either by the current operator or any resulting operator. We refer to this technique as computing the transitive closure of the events over the operators. This yields substantially better results which we summarise as follows:

N	2	3	4	5
Runtime (s)	23	126	440	1335

2.6.3. *Sequences Using Transitive Closure of Events.* The next optimisation follows from a surprising result; namely that the implementation of sets within FDR is extremely slow. For example, the Haskell program (using the

implementation of sets from `Data.Set`) `size (fromList [0..3000])` executes instantly whereas the equivalent CSP program, card(set(<0..3000>)) takes around 7 seconds to execute. Therefore, since there are few places where duplicates are created, we instead represent all the rules, partial functions and associated data structures using sequences and eliminate duplicates explicitly[9]. This yields a big improvement on the runtime as follows:

N	2	3	4	5	6
Runtime (s)	8	20	44	94	Did not finish[10]

2.6.4. *Using Integers Rather than Tuples.* The next optimisation follows from the observation that the average tuple on the *renamed* channel is complicated, thus making equality comparisons slow. Therefore, to solve this every tuple was represented by an integer and the type of *renamed* was then declared to be $\{0..M\}$ for some M sufficiently large. This on its own would yield results that were no better, since many equality comparisons would still have to be done in order to lookup the integer for a given tuple. Therefore, the computation of the transitive closure of the events over the operators was altered so that it returned a partial function that allowed the regulator to get the set of events that it should offer, along with the resulting state it should go to. This yielded a reasonable improvement in the runtime as follows:

N	2	3	4	5	6
Runtime (s)	5	13	29	88	Did not finish

2.6.5. *Only Using the Full Harness When Necessary.* The last optimisation that we consider is by far the simplest. Recall that the harness is defined as follows:

$$Harness(P, n) \mathrel{\widehat{=}} ((P[[Prime]] \; \Theta_{\{x' | x \in \Sigma\}} \; Stop) \\ \triangle \; off \rightarrow Stop)[[Rename]].$$

Clearly, if it is not possible for the process n to be turned **off** (as is the case with CSP's parallel operator, for example) then $Harness(P, n)$ would be equivalent to $P[[Rename]]$. Therefore, we adapt the computation of the transitive closure of events to also calculate which processes may be discarded by any future operator, and then we only use the full harness when the process may be discarded. This yields a considerable improvement in the asymptotic behaviour of the runtime as follows:

N	2	3	4	5	6	7	8
Runtime (s)	5	12	25	49	91	163	319

[9]Unfortunately this required converting a set to a sequence (since there is no sequence equivalent to {|e |}); therefore a function seq was implemented in a special release of FDR that returns a sequence (non-deterministically) that is equivalent to the set.

[10]Unfortunately, due to a bug in the pre-release version of FDR the author has access to, one of the intermediate processes has to be explicated resulting in a huge blow up in the memory requirement meaning that some checks could not be completed.

2.6.6. *Summary.* The runtime for FDR on the non-simulated version where there are 8 philosophers is only 2 seconds indicating that our simulation is still substantially slower than the non-simulated version. However, this is not a massive problem since the main point of this construction is to allow a user to experiment with their language or model on reasonable-sized problems. It is not designed or intended that it implements it in the most efficient way possible. Hence, in Tyger we make use of the simulation as described in Section 2.6.5. The full source code for this simulation given the CSP operational semantics can be found in Listing B.2.

3. Tyger

Tyger is a Haskell program that, given two files, one specifying the operational semantics of a language and the other containing process definitions, will produce a CSP file containing simulations of the processes that can be compiled by the model checker FDR. In particular, the user may specify the syntax of their operators and then use this syntax in the process definitions. For example, the user could specify that the string `!!` should be recognised as a binary operator `Foo` and thus the definition `R = P !! Q` would be parsed as `R = Foo(P,Q)`.

For the rest of this section we assume that *script* refers to the script containing the users' process definition and that *operational semantics definition* refers to the file containing the users' description of the operational semantics. In Section 3.1 we describe the operational semantics side of Tyger including the input file formats and the type checker. In Section 3.2 we describe how the script is processed, including the type checking that is performed and how recursion is handled. Lastly, in Section 3.3 we give an example of a run through in Tyger detailing which Haskell functions are called when.

3.1. **Operational Semantics.**

3.1.1. *Input Format.* The operational semantics of a language are specified by a file containing several operator descriptions. As an example of an operator description consider the following snippet that describes the CSP parallel operator (ignoring sequential composition):

```
Operator Parallel(P : InfRec, Q : InfRec, A)
  Syntax Binary "[| $3 |]" 12 AssocNone

  Rule
    P =a=> P'
    Q =a=> Q'
    ------------------------------  a <- A
    P [| A |] Q =a=> P' [| A |] Q'
  EndRule

  Rule
    P =a=> P'
    ------------------------------  a <- diff(Sigma, A)
    P [| A |] Q =a=> P' [| A |] Q
```

```
  EndRule

  Rule
    Q =a=> Q'
    ------------------------------  a <- diff(Sigma, A)
    P [| A |] Q =a=> P [| A |] Q'
  EndRule
EndOperator
```

We describe the meaning of each section in turn. The declaration `Parallel(P : InfRec, Q : InfRec, A)` declares that this operator is named Parallel and takes three arguments, P, Q and A where P and Q are infinitely recursive (as defined in Definition 2.6). The line `Syntax Binary "[| $3 |]" 12 AssocNone` is used for parsing both the subsequent rules and the script and indicates:

- that the parallel operator is a binary operator;
- the operator is recognised by any string of the form `[| e |]` where `e` is an expression, and that the third argument (namely A) is `e`;
- the operator is not associative;
- the operator has a precedence of 12 (the lower the number the tighter the operator binds); the pre-defined operators' precedences are given by:

Operator	Precedence
`e(...)` (function application)	0
`-` (unary)	1
`/`, `%`, `+`, `-`, `*`	2
`.`, `#`, `^`	3
`>`, `>=`, `<`, `<=`, `!=`, `==`	4
`and`, `or`, `not`	6

Each rule of the operator consists of zero or more premises (where the relation $\xrightarrow{a}$ is written as `=a=>`) separated by a string of `-` (that may be of any length) from the rule indicating how the operator evolves. The side condition is written next to the `-`. It may include clauses that bind free variables (written as `p <- e` for `p` a pattern and `e` an expression that evaluates to a set) and clauses that are propositional formulas (where the basic predicates are `e1 == e2`, `member(p,e1)` and `e1 <= e2` (i.e. subset) for `e1`, `e2` expressions and `p` a pattern). Note that tau promotion rules are not required since the operational semantics for CSP ensures that taus of any **on** process are promoted.

Support is also available for replicated operators. As an example we give the input corresponding to the alphabetised parallel operator of CSP:

```
Operator AlphaParallel(P, Q, A, B)
  Syntax Binary "[ $3 || $4 ]" 12 AssocNone

  Rule
    P =a=> P'
    Q =a=> Q'
    ------------------------------    a <- inter(A,B)
    P [A || B] Q =a=> P' [A || B] Q'
```

```
  EndRule

  Rule
    P =a=> P'
    ------------------------------    a <- diff(A, B)
    P [A || B] Q =a=> P' [A || B] Q
  EndRule

  Rule
    Q =a=> Q'
    ------------------------------    a <- diff(B, A)
    P [A || B] Q =a=> P [A || B] Q'
  EndRule

  Replicated(P, A)
    Syntax Prefix "|| $0 @ [$2]" 9

    BaseCase({}, {})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps, As)
      P [A || Union(As)] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator
```

We describe each element of the new `Replicated` section in turn. The declaration `Replicated(P,A)` states that the current operator has a replicated version where every item in the list has arguments `P` and `A`. The `Syntax` statement is mostly as before, except `$0` is a special indicator meaning the *generators* that a replicated operator takes (i.e. a comma separated list of expressions of the form $p : e$ and e for e an expression and p a pattern). The `BaseCase({}, {})` statement indicates that the base case of the replicated operator corresponds to having an empty set of arguments and is equal to `CSPSTOP`. `InductiveCase(Ps,As)` gives the recursive case; in particular `P` and `A` are in scope and bound to the current item and `Ps` and `As` are bound to the remaining set of processes and the remaining set of alphabets respectively. Inside the inductive case `InductiveCase` gives the process that results from the recursive call.

Note that in in some models it may be required to extend Σ (see Section 4 for examples). We support this by allowing channels to be declared by including the following section as the first section in the file:

```
Channels
        tick
        offer : Sigma
        offerSet : Set(Sigma)
EndChannels
```

where `Set` is the powerset constructor. A complete sample input file for the standard CSP operators may be found in Listing B.1. For a formal grammar of the input file see Appendix A.

3.1.2. *Type Checker.* Tyger contains a type checker for the operational semantics that not only checks for the consistency of the rules but also checks most of the rules for CSP-likeness. In particular it checks rules 2—7 as defined in Definition 2.3. Rule 1 is implicitly checked since it is not possible to express the absence of an event being performed as a precondition.

We now consider the type system that is used; assuming a set of type variables V the set of types T can be defined inductively by:

$$\frac{v \in V}{v \in T} \qquad \frac{}{TEvent \in T} \qquad \frac{}{TOnProcess \in T} \qquad \frac{}{TOffProcess \in T}$$

$$\frac{}{TProcArg^{11} \in T} \qquad \frac{t \in T}{\{t\} \in T} \qquad \frac{t_1 \in T, \ldots, t_n \in T}{\langle t_1, \ldots, t_n \rangle \in T}$$

$$\frac{t_1 \in T, \ldots, t_n \in T}{TChannel\ \langle t_1, \ldots, t_n \rangle \in T} \qquad \frac{t_1 \in T, \ldots, t_n \in T}{TOperator\ \langle t_1, \ldots, t_n \rangle \in T}$$

where $TChannel\ \langle t_1, \ldots, t_n \rangle$ is the type of a channel that takes n components each of type t_i, and $TOperator\ \langle t_1, \ldots, t_n \rangle$ is the type of an operator that takes n arguments each of type t_i. The rules for type checking the operational semantics are unsurprising and are thus omitted; they can however be viewed in `OpSemTypeChecker.hs` (Appendix C.2.3). The type checker is also able to output GHC-style (Glasgow Haskell Compiler[12]) error messages that can help pinpoint the error. For example, given the operator definition:

```
Operator Rename(P : InfRec, A)
  Rule
    P =a=> P'
    ---------------------------- (a,b) <- A
    Rename(P,A) =b=> Rename(P',A)
  EndRule

  Rule
    P =a=> P'
    ---------------------------- a <- diff(Sigma, {b | b <- A})
    Rename(P,A) =a=> Rename(P',A)
  EndRule
EndOperator
```

the following error is produced:

```
Could not match the types:
  "Event"
and
  "(Event, Event)"
```

[11]This type is relevant to the implementation of recursion in Tyger. In particular, it is used to define the semantics of the identity operator and is never deduced as a type for any user operator.

[12]Available from: `http://www.haskell.org/ghc/`.

```
in the expression:
  diff(Sigma, {b | b <- A})
in the rule:
  P =a=> P'
  ------------------------------ a <- diff(Sigma, {b | b <- A})
  Rename(P, A) =a=> Rename(P', A)
in the operator "Rename".
```

In the current version of Tyger mutual recursion amongst the operators is not supported by the type checker. Therefore, the user is required to specify the operators in an order such that if Op precedes Op' then Op may not call Op'.

3.1.3. *Recursion.* As mentioned in Section 2.5.3 recursion requires extra support. In particular, in order to support the automated refactorings we need to be able to deduce from the operational semantics which arguments of which operators are finitely recursive, infinitely recursive and finalised. We note that most languages (CSP included) contain only infinitely recursive operators and 1-finalised operators. Therefore, we require the user to annotate any arguments that are infinitely recursive and assume otherwise that any argument is finitely recursive. This can be specified as follows:

```
Operator Parallel(P : InfRec, Q : InfRec, A)
...
EndOperator
```

This ensures that the argument will be identified as infinitely recursive rather than finitely recursive which affects the automated refactorings that are discussed in Section 3.2.3.

3.1.4. *Output Format.* Tyger outputs a file (that is intended to be included by the file created by the functional portion of Tyger) containing one module, `Operator_M`, that contains the simulation of the operational semantics along with operator shortcuts such as `ExternalChoice(P,Q) = Operator(Op_ExternalChoice, <P,Q>, <>)`. Most of the code that is output does not change with the operational semantics (and is contained in the file `ConstantCode.hs` (Appendix C.2.6)); the only portions that are dynamically generated are the function *Rules*, as defined in Section 2.3 along with the operator shortcuts. The only other point of note is that every variable name has a `0` appended to its name, except the operator shortcuts that have a prime appended to their names. This is to ensure that there are no variable name clashes between the generated code and the pre-defined code. As an example Listing B.2 gives the CSP produced when given the operational semantics in Listing B.1.

3.2. **Functional Language.**

3.2.1. *Input.* In this section we describe the programming language that the user may give their process definitions in. It is essentially the functional portion of CSP_M together with support for the users' custom operators. For example, Listing 3.1 gives a file defining the Dining Philosophers problem that could be input to Tyger along with the standard CSP operational semantics (as defined in Listing B.1).

Listing 3.1. CSPM input script for the Dining Philosophers using the standard CSP operational semantics as defined in Listing B.1. This differs only from the version that would normally be given to FDR in that bounds are specified for the recursive processes (see Section 3.2.3) for more details).

```
N = 2
ID = {0..N}
channel pickup, putdown : ID.ID
channel sitdown, eat, getup : ID

mplus(x, y) = (x + y) % N

-- Original process
Philosopher :: (ID) → Proc
Philosopher(id) =
  sitdown.id → pickup.id.id → pickup.id.mplus(id,1) → eat.id →
  putdown.id.mplus(id,1) → putdown.id.id → getup.id →
Philosopher(id)

Fork :: (ID) → Proc
Fork(id) =
   □ phil : {id, mplus(id,1)} • pickup.phil.id →
putdown.phil.id → Fork(id)
AlphaFork(id) =
  {pickup.id.id, putdown.id.id, pickup.mplus(id,1).id,
    putdown.mplus(id,1).id}

System =
  ( ||| id : {0..N} • Philosopher(id))
  [| {| putdown, pickup |} |]
  ( || id : {0..N} • [AlphaFork(id)] Fork(id))
```

We do omit several notable features that are usually admissable in CSP$_M$ scripts as follows. The most notable is the lack of support for prefixing, namely expressions of the form x?y. This was omitted mostly due to the difficulty in supporting this; in particular the user would have to specify what prefixing means (e.g. in CSP prefixing is equivalent to the replicated external choice over an appropriate set). There is no reason why this could not be supported but time did not allow.

Another notable feature that is missing is support for CSP's module system. The reason for omitting this is principally because the semantics are ill-defined. For example, FDR does not emit an error when given the program:

```
module A
        f = 0
exports
endmodule

g = A::f
```

which it should do since `f` should not be accessible outside of `A` as it is not exported.

We also do not allow the use of the dot operator in arbitrary contexts; in particular the left hand side of a dot must either be a Datatype or a Channel. This is done to ensure that type checking is decidable, in particular it means that the definition `x = 0 . <x>` is not admitted. This is useful since the the type of x is infinite and thus no type can be inferred for it. Also, this restriction has no practical impact since we omit prefixing which is the only place where this is commonly used (for pattern matching over multiple components of a channel simultaneously).

The last notable feature that we omit is not one that the author realised existed until attempting to write a type checker for CSP_M, namely, support for functions that pattern match on different types such as:

```
gen(true) = false
gen(1) = 0
```

A decision was made not to support these sort of expressions since they muddy the semantics of the language. Furthermore, more elegant solutions (such as Haskell's *type classes*) could be used to solve this. A related item that we omit is pattern matching on channels. For example, we do not admit the definition:

```
channel p : A
channel q : B
gen(p) = ...
gen(q) = ...
```

for analogous reasons to the above. Again, this is not a well-used feature and therefore its omission, in practice, does not cause issues.

3.2.2. *Type Checker.* In order to support the automated refactoring required for recursion it was necessary to be able to get all process definitions from within a file (i.e. all definitions that have type *TProc*). Therefore a type checker was written that takes as input an abstract syntax tree (henceforth AST) and produces as output either an AST annotated with type information or an error. The errors are similar to ones emitted by GHC; as an example when type checking the program:

```
f(x) = x + 1
g = f(true)
```

the following error is emitted:

```
Could not match the types:
  Int
and
  Bool
in the expression at <stdin>:2:6:
  f(true)
in the declaration of g.
```

In this section we describe the type system and type inference algorithm, both of which are extensions of the standard Hindley-Milner type inference

algorithm [Hin69, Mil78] to support the extra features of machine-readble CSP.

The type system used by the type checker is *rank-1 polymorphic* meaning that the type $\forall\, a \cdot (a \to a)$ is valid but $(\forall\, a.a \to a) \to a$ is not (the former is the type of the identity function whereas the latter is the type of a function that takes a polymorphic function and returns a). It also admits basic *type constraints* of the form *Eq* and *Ord*. For example, the function cmp(x,y) = x == y is typed as $\forall\, Eq\ a \cdot (a, a) \to Bool$. This means that the type system is a *bounded rank-1 polymorphic* type system (see e.g. [CW85]). We now describe the set of valid *monomorphic* (i.e. types that are not quantified) types T, assuming a set V of variables and N a set of datatype names. We firstly give inductive rules referring to the basic types:

$$\frac{v \in V}{TVar\ v \in T} \qquad \frac{}{Int \in T} \qquad \frac{}{Bool \in T} \qquad \frac{}{Proc \in T}$$

$$\frac{t \in T}{\{t\} \in T} \qquad \frac{t \in T}{\langle t \rangle \in T} \qquad \frac{t_1, \ldots, t_m \in T}{(t_1, \ldots, t_m) \in T} \qquad \frac{n \in N \quad t_1, \ldots, t_n \in T}{TDataType\ n\ \langle t_1, \ldots, t_n \rangle \in T}$$

$$\frac{t \in T \quad t_1, \ldots, t_n \in T}{(t_1, \ldots, t_n) \to t \in T}$$

$$\frac{t_1 \in T \quad t_2 \in T}{TDotable\ t_1\ t_2 \in T}$$

where $TDotable\ t_1\ t_2$ means that something of this type can be dotted (i.e. placed on the left hand side of a '.') with a t_1 to yield the type t_2, and $TDataType\ n\ \langle t_1, \ldots, t_n \rangle$ means a datatype clause that is in the datatype n that takes $t_1, \ldots, t_n$ as its components. For example, given the datatype definition **datatype** T = A.0 | B.0.0, A has type $TDataType\ T\ \langle Int \rangle$ and B has type $TDataType\ T\ \langle Int, Int \rangle$. Also, you could write A's type as $TDotable\ Int\ (TDataType\ T\ \langle\rangle)$ (but the former is its most general type).

We now consider what types need to be added in order to support channels. Firstly, we consider the type of g when defined by g(x,y) = {|x.y|}. Suppose we let y have type b; it follows that we have to allow x to be any channel that takes a b as its first component, but that it may have arbitrarily many components after that. Hence, x must have type $TChannel\ y$ for any sequence of types y that begins with b. In order to express this we define the following types:

$$\frac{v \in V}{TPolyList\ v \in T} \qquad \frac{t_1 \in T \quad t_2 \in T}{TList\ t_1\ t_2 \in T} \qquad \frac{}{TListEnd \in T}$$

$$\frac{t \in T}{TChannel\ t \in T}$$

Therefore, we define *type list* to mean a structure that contains *TPolyList*, *TList* and *TListEnd* components. It should be noted that the inner type in a *TChannel* is always a type list (and type lists only appear in the contexts of channels) but, in order to simplify the implementation only one set of types is defined. Thus, the type $TChannel\ (TList\ t_1\ TListEnd)$ indicates that the channel has one component, t_1. The type $TChannel\ (TList\ t_1\ (TPolyList\ v))$ indicates that

the channel has at least one component, t_1 but then may have arbitrarily many components after. Lastly, we note that an event is the same as a channel that has no components. Thus the type of `g` is $\forall\, a, b \cdot (TChannel\ (TList\ b\ (TPolyList\ a)), b) \to \{TChannel\ TListEnd\}$.

We can now define the set of polymorphic types TS that we admit, assuming C is a set of constraints that contains a constraint that admits all types, as follows:

$$\frac{t \in T \quad v_1, \ldots, v_n \in V \quad c_1, \ldots, c_n \in C}{\forall\, c_1\ v_1, \ldots, c_n\ v_n \cdot t \in TS}$$

The process of going from a monomorphic type to a polymorphic type is known as *generalisation*.

Before describing the type checking algorithm we make a few definitions.

Definition 3.2. *A strongly connected component (SCC) in a directed graph G is a set of vertices C where there is a path between any $v_1, v_2 \in C$.*

Definition 3.3. *A topological sort of a sequence of strongly connected components is an ordering on the components such that c precedes d iff there is no edge from c to d.*

We now describe the general type checking algorithm. The input is an AST consisting of a number of definitions. The type checker firstly computes the dependency graph amongst the definitions (i.e. the graph where there is an edge from d_1 to d_2 if d_1 uses a variable bound by d_2) and then identifies the strongly connected components within it (using a built in Haskell function from the `Data.Graph` module that internally uses Tarjan's Strongly Connected Components Algorithm [Tar72]). Then, it topologically sorts the strongly connected components so that the first group of definitions that are type checked depends on no other group. It then performs type inference on each definition in the group before generalising the types. It then stores the polymorphic types so that types can be correctly inferred for the subsequent groups. The source code for the type checker is contained in Appendix C.4

To see why we must ensure that we type check only mutually recursive functions together consider the following example:

```
id(x) = x
g = id(true)
h = id(0).
```

Note that if we were to type check all the definitions together before generalising an error would be encountered. This is because lines 1 and 2 would infer a type for `id` of $Bool \to Bool$ which would thus mean that the call to `id` on line 3 would not be correctly typed. However, if we perform the type checking as described above the correct types will be inferred as the type for `id` would be generalised to a polymorphic type before type checking `g` or `h`. Unfortunately, this does not solve every problem. For example, consider the program:

```
fst((x,y)) = x
f(x) = (x,m)
m = fst(f(true))
```

and note that the most general type of m is *Bool* and thus f is of type $\forall\, a \cdot a \to (a, Bool)$. However, the algorithm described above will infer that f will be of type $Bool \to (Bool, Bool)$ since the type of f is generalised only after it has been restricted to being a boolean. Interestingly, GHC has the same problem when presented with the equivalent Haskell program and relies on the user manually annotating f with the correct type in order to infer the most general type. However, the technique that GHC uses to do this (as described in [VWPJ06]) is exceedingly complicated and (reportedly) unused and has therefore been scheduled for removal in a subsequent release. As a result of this we make no attempt to support the above case.

We now compare our type checker to the two existing publicly announced CSP_M type checkers that have been developed. The first we consider is that of Gao and Esser as described in [GE01]. Whilst their type system is able to accommodate most of the peculiarities of CSP_M, the type system they present is unable to type the function g(x) = {|x|} as it contains no way to state that x is a channel with arbitrary components. This is because they have no equivalent to our *TPolyList* v constructor above.

The other CSP type checker that is available is that of Formal Systems Europe [For01] Ltd. It is, in some senses, more complete than the one presented above as it admits arbitrary usage of dot (and thus does not terminate in some cases) along with union types (which the author thinks should not be part of the language). However, it too is unable to correctly type the above function g(x) = {|x|} and thus, when presented with the following, well-typed, program emits an error erroneously:

```
datatype B = A
channel a : {0}.{0}.{0}
channel b : B
f(x,y) = {| x.y |}
p = f(a,0)
q = f(b,A)
```

The reason why this fails to type check is unknown as there is insufficient documentation available on the type system. However, it would appear that it suffers from the same problem as described above: namely the type system is unable to express the fact that in the above example x is a channel that takes y as its first component but then has arbitrarily many components after that.

3.2.3. *Recursion.* As stated in Section 2.5.3 recursion, unfortunately, requires extra support. In this section we discuss the implementation of the automated refactorings that were discussed in Section 2.5.3 within Tyger.

Recall that in our implementation of recursion in Section 2.5.3 we assumed that processes may be transmitted over channels. Since FDR does not allow this we must instead use a different technique of transmitting which process should be called. Instead, we chose to create a datatype ProcArgs that contains one entry for each process that may be called recursively (with components according to the type of each argument). However, this does introduce one issue; since channels must be finite[13] we must be able to infer finite bounds on all arguments of recursive processes. However, this is not trivial as the following example shows:

```
Count(0) = up → Count(1)
Count(n) = n < N & up → Count(n+1) □ down → Count(n−1).
```

Clearly the bound on the value of the argument n not only depends on the starting value but it can also only be deduced by evaluating Count(i) where i is the initial value until the whole state space has been explored. Whilst this is theoretically possible it would require implementing an evaluator for CSP_M, something that is beyond the scope of this Thesis. Hence, an alternative solution is to require the user to manually bound arguments by giving the set of arguments that are allowed as follows:

```
Count :: ({0..N}) → Proc
Count(0) = up → Count(1)
Count(n) = n < N & up → Count(n+1) □ down → Count(n−1).
```

If no bounds are given for a process then it is assumed that the process only recurses a finite number of times. For example, no bounds are required on the process:

```
Spawn(P, 0) = STOP
Spawn(P, N) = P ||| Spawn(P, N−1)
```

since although it is recursive it can only recurse N times.

In order to simplify the refactorings that are done automatically a number of restrictions are placed on the script. In particular, no recursive process definitions may be made in let expressions and lambda expressions with a return type of *TProc*; recursive functions must be of type *TProc* (and not a tuple for example); further recursive curried functions are prohibited. Also, we assume that the user has inlined all finalised arguments appropriately. Given this we can summarise the algorithm as follows:

- A graph of processes is constructed (where there is an edge from P to P' iff P calls P'), and every process that is within a strongly connected component is identified as recursive (unless no bounds have been placed on its arguments);
- The ProcArgs datatype is then computed, with an entry for each recursive process;
- The function GetProc(proc) is defined as GetProc(Proc_P...) = P_UNWRAPPED(...) for every recursive P;
- The AST of the script is then transformed as follows:

[13]FDR does allow infinite channels providing only a finite portion of it is used. However, we cannot make use of this since if we did the definition of WrapThread would use an infinite set as the alphabet of the exception operator.

- Two copies of every recursive process P are made, namely P and P_UNWRAPPED where P is defined to be CallProc(Proc_P...) and P_UNWRAPPED is defined as P was originally but with every call to a recursive process replaced by an appropriate Call(Proc...) call;
- Each infinitely recursive argument of an operator is wrapped by ensuring that every call to another process calls the wrapped version.

As an example of the above refactorings Listing 3.4 shows the output of Tyger when given the standard CSP operational semantics as defined in Listing B.1 and the input file for the Dining Philosophers as defined in Listing 3.1.

Listing 3.4. The output of Tyger when given the standard CSP operational semantics file in Listing B.1 and the Dining Philosophers input file in Listing 3.1.

```
include "CSP.csp"

UserEvents = {|pickup', putdown', sitdown', eat', getup'|}

datatype ProcArgs = Proc_Philosopher'.ID'| Proc_Fork'.ID'|
      Proc_PhilCanEatSpec'.ID'

GetProc(Proc_Philosopher'.arg_1') =
  Philosopher_UNWRAPPED'(arg_1')
GetProc(Proc_Fork'.arg_1') = Fork_UNWRAPPED'(arg_1')
GetProc(Proc_PhilCanEatSpec'.arg_1') =
  PhilCanEatSpec_UNWRAPPED'(arg_1')

N' = 2

ID' = {0..N'}

channel pickup', putdown' : ID'.ID'

channel sitdown', eat', getup' : ID'

mplus'(x', y') = (x' + y') % N'

Philosopher_UNWRAPPED'(id') =
  Prefix'(sitdown'.id',
    Prefix'(pickup'.id'.id',
      Prefix'(pickup'.id'.mplus'(id', 1),
        Prefix'(eat'.id',
          Prefix'(putdown'.id'.mplus'(id', 1),
            Prefix'(putdown'.id'.id',
              Prefix'(getup'.id',
                CallProc(Proc_Philosopher'.id'))))))))

Philosopher'(arg_1') = WrapThread(Proc_Philosopher'.arg_1')

Fork_UNWRAPPED'(id') =
  ReplicatedExternalChoice'(<Prefix'(pickup'.phil'.id',
```

```
            Prefix'(putdown'.phil'.id',
              CallProc(Proc_Fork'.id')))  | phil'
                  ← seq({id',
                     mplus'(id',
                       1)})>)

Fork'(arg_1') = WrapThread(Proc_Fork'.arg_1')

AlphaFork'(id') =
  {pickup'.id'.id', putdown'.id'.id', pickup'.mplus'(id', 1).id',
  putdown'.mplus'(id', 1).id'}

System' =
  Parallel'(ReplicatedInterleave'(<Philosopher'(id') | id'
              ← seq({0..N'})>),
    ReplicatedAlphaParallel'(<Fork'(id') | id' ← seq({0..N'})>,
      <AlphaFork'(id') | id' ← seq({0..N'})>),
        {|putdown', pickup'|})

PhilCanEatSpec_UNWRAPPED'(n') =
  Prefix'(eat'.n', CallProc(Proc_PhilCanEatSpec'.n'))

PhilCanEatSpec'(arg_1') = WrapThread(Proc_PhilCanEatSpec'.arg_1')

assert PhilCanEatSpec'(0) ⊑F Hide'(System',
        {|pickup', putdown', sitdown', getup', eat'.1, eat'.2|})
```

3.2.4. *Output Format.* Tyger outputs a file that uses a include statement to include the file that was created by the operational semantics portion and contains all the definitions in the input script (having been refactored as described above) along with several functions that are assumed to exist by the operational semantics simulation. In particular UserEvents is defined as the set of all events that the user uses; the function GetProc and datatype ProcArgs are defined as described in Section 3.2.3. Also, as in the case of the operational semantics output, every variable name has a prime appended to it to ensure that no variable names clash with names used in the internal simulation.

3.3. **Anatomy of a Run Through Tyger.** In this Section we describe in more detail the Haskell implementation of Tyger by showing how two input files are transformed. Suppose the input to Tyger is two files, `Semantics.opsem` giving the operational semantics definitions and `Example.csp` giving the process definitions. These files are processed by Tyger as follows:

(1) Firstly, `Semantics.opsem` is parsed in `OpSemParser.hs` (Appendix C.2.2). This happens in two passes through the file. On the first pass all the syntax components of every operator are extracted. Then, in the second pass these syntax components are used to parse the operator definitions. The output of this stage is either an AST (of type

`InputOpSemDefinition` as defined in `OpSemDataStructures.hs` (Appendix C.2.1)) or an error message. The parser itself is written using the monadic parser combinator library, Parsec[14].

(2) The operational semantics are then typechecked in `OpSemTypeChecker.hs` (Appendix C.2.3); this is where the special identity operator is injected into the operational semantics. The type checking is as described above in Section 3.1.2. The output of this stage is either an AST where every operator is annotated with its type (of type `OpSemDefinition` as defined in `OpSemDataStructures.hs` (Appendix C.2.1)) or an error message.

(3) `Example.csp` is then parsed in `CSPMParser.hs` (Appendix C.3.2); this makes use of the output of the previous stage in order to know the syntax of the users' custom operators. Again, the parser is written using Parsec and the output for this stage is either an AST (of type `[PModule]`, as defined in `CSPMDataStructures.hs` (Appendix C.3.1)), annotated with source locations, or an error message detailing a syntax error.

(4) The next stage is to then typecheck the functional language (the source code for this section spans multiple files in Section C.4). This takes as input both the operational semantics definitions (so that the types of the users' operators are known) and the AST created in the previous stage. The algorithm then proceeds as outlined in Section 3.2.2 and outputs either an AST annotated with type information (of type `[TCModule]`) or an error message indicating a type error (these make use of the source locations that the AST is annotated with).

(5) Having done this the users' definitions are then refactored by the code in `CSPMRecursionRefactorings.hs` (Appendix C.3.4) as described in Section 3.2.3 in order to support recursion. The input to this stage is the typechecked AST (so that type information is available) and the output is either the transformed AST or an error message indicating that the user has done something that is unsupported by the automated refactorings. Furthermore, the output contains two new definitions, GetProc and the datatype ProcArgs, both as defined in Section 3.2.3.

(6) At this point we know that the users' input is valid and therefore start producing output. The first thing to do is to convert the users' operational semantics definitions from the format they were input in (i.e. standard inductive rules) to the format described in Section 2.3. This is done in `OpSemRules.hs` (Appendix C.2.5) and takes as input the operational semantics AST, annotated with argument types and produces as output a list of `CompiledOp` (as defined in `OpSemDataStructures.hs` (Appendix C.2.1)). This is then pretty printed (by `OpSemRules.hs` (Appendix C.2.5)) to produce the function Rules, the operator shortcuts, the Operator datatype, the user channel definitions and the SystemEvents definition. These definitions are then spliced into the appropriate places in the code contained in `ConstantCode.hs` (Appendix C.2.6).

[14]Available from `http://hackage.haskell.org/package/parsec`.

(7) Lastly, the users' process definitions, as transformed in stage (5), are pretty printed in `CSPMPrettyPrinter.hs` (Appendix C.3.3) along with the set UserEvents.

What we describe above is essentially what is implemented in `Main.hs` (Appendix C.1.2).

4. Examples

In this section we present a number of input files that are able to simulate a number of languages available in the literature.

4.1. **Lowe's Availability Model.** In [Low10] Lowe defines a new model for CSP that records what events a process makes available in addition to the events that it actually performs. Furthermore, he provides *congruent* CSP-like operational semantics meaning that we can use Tyger to simulate the model. He gives several different variants of this model of which we now discuss two.

4.1.1. *Singleton Model.* The simplest model is known as the singleton availability model and enables processes to indicate that they can perform a particular event. The operational semantics of this model can be deduced from the standard operational semantics of CSP as follows:

$$P \xrightarrow{a} P' \Leftrightarrow P \overset{a}{\Longrightarrow} P' \qquad P \xrightarrow{a} \cdot \Leftrightarrow P \overset{offer\ a}{\Longrightarrow} P$$

where $\xrightarrow{a}$ specifies CSP's operational semantics and $\overset{a}{\Longrightarrow}$ specifies the semantics of the singleton availability model. We can implement these semantics directly in the format required by Tyger without any difficulty. We firstly have to specify the channel *offer* as follows:

```
Channels
  offer : Sigma
EndChannels
```

In Listing 4.1 we give the definition of prefixing and external choice. A more complete version can be seen in Listing B.3.

Listing 4.1. The operational semantics of the prefixing and external choice operators in the singleton availability model can be defined as follows:

```
Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    -------------------------
    a -> P =offer.a=> a -> P
  EndRule
  Rule
    -------------
    a -> P =a=> P
  EndRule
EndOperator

Operator ExternalChoice(P, Q)
```

```
  Syntax Binary "[]" 10 AssocLeft
  Rule
    P =a=> P'
    ------------------   a <- Sigma
    P [] Q =a=> P'
  EndRule
  Rule
    Q =a=> Q'
    --------------      a <- Sigma
    P [] Q =a=> Q'
  EndRule
  Rule
    P =offer.a=> P'
    -------------------------   a <- Sigma
    P [] Q =offer.a=> P' [] Q
  EndRule
  Rule
    Q =offer.a=> Q'
    -------------------------   a <- Sigma
    P [] Q =offer.a=> P [] Q'
  EndRule
  Replicated(P)
    Syntax Prefix "[] $0 @" 9

    BaseCase({})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps)
      P [] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator
```

4.1.2. *Set Model.* Lowe also defines a version of his availability model that gives a set of events that the process offers concurrently. The operational semantics of this can, again, be defined in terms of the operational semantics of CSP by:

$$P \xrightarrow{a} P' \Leftrightarrow P \overset{a}{\Longrightarrow} P' \qquad P \xrightarrow{offer\ A} P \Leftrightarrow \forall\, a \in A \cdot P \xrightarrow{a}$$

In order to produce a simulation using Tyger we firstly have to specify the channel *offer* as follows:

```
Channels
  offer : Set(Sigma)
EndChannels
```

In Listing 4.2 we give the definition of prefixing and alphabetised parallel in this model. A more complete version can be seen in Listing B.4. Using the two simulations above we can then show that the that the set availability model is more coarse than the singleton availability model by noting that the assertions in the following script both hold in the singleton availability model but the second fails in the set availability model.

```
channel a, b

P = a → CSPSTOP □ b → CSPSTOP
Q = a → CSPSTOP ⊓ b → CSPSTOP

R = a → CSPSTOP

assert P ⊑T Q
assert Q ⊑T P
```

Listing 4.2. The operational semantics of the prefixing and alphabetised parallel operators in the set availability model can be defined as follows:

```
Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    -------------------------
    a -> P =offer.{a}=> a -> P
  EndRule
  Rule
    -------------------------
    a -> P =offer.{}=> a -> P
  EndRule
  Rule
    -------------
    a -> P =a=> P
  EndRule
EndOperator

Operator AlphaParallel(P, Q, A, B)
  Syntax Binary "[ $3 || $4 ]" 12  AssocNone

  Rule
    P =offer.X=> P'
    Q =offer.Y=> Q'
    ------------------------------------------------
      X <- Set(Sigma), Y <- Set(Sigma), X <= A, Y <= B,
      inter(X,A) == inter(Y,B)
    P [ A || B ] Q =offer.union(A,B)=> P' [ A || B ] Q'
  EndRule

  Rule
    P =a=> P'
```

```
    Q =a=> Q'
    ---------------------------------- a <- inter(A,B)
    P [ A || B ] Q =a=> P' [ A || B ] Q'
  EndRule
  Rule
    P =a=> P'
    ---------------------------------- a <- diff(A,B)
    P [ A || B ] Q =a=> P' [ A || B ] Q
  EndRule
  Rule
    Q =a=> Q'
    ---------------------------------- a <- diff(B,A)
    P [ A || B ] Q =a=> P [ A || B ] Q'
  EndRule
EndOperator
```

4.2. **Lowe's Readyness Testing Model.** In [Low09] Lowe considers an extension to CSP that allows processes to test whether an event is available. He does this by adding an operator, *if ready a then P else Q* that, if a is offered by the environment behaves like P and otherwise behaves like Q. In this section we show how to simulate the Readiness-Testing Traces Model within FDR using the operational semantics given in Appendix A of [Low09].

The operational semantics of Lowe's readyiness-testing model are based upon the standard CSP operational semantics with some additional events. In particular, there are $offer.a$, $notOffer.a$, $ready.a$, $notReady.a$ events that indicate respectively that: the process is offering an a; the process is not offering an a; the process is testing if an a is available; the process is testing if an a is not available. In order to model this with Tyger we declare the following channels:

```
Channels
  offer: Sigma
  notOffer : Sigma
  ready : Sigma
  notReady : Sigma
EndChannels
```

Using these definitions we can then define the operational semantics of the language. In Listing 4.3 we present the operational semantics of prefixing and of the ready testing operator. The full semantics of the language are given in Listing B.5.

In [Low09] Lowe gives a solution to the readers and writers problem [CHP71] that is *fair* to the writers in that collectively the writers cannot forever be denied access to the resource. Lowe then gives a (necessarily) complicated simulation of the solution for use within FDR. However, using the operational semantics that we defined above we are able to express this much more succinctly in Tyger's input format as follows:

```
MaxReaders = 1
MaxWriters = 1
Readers = {0..MaxReaders}
```

```
Writers = {0..MaxWriters}

channel startWrite, startRead, endWrite, endRead

Guard :: (Readers, Writers) → Proc
Guard(r, w) =
  if w == 0 and r < MaxReaders then
    if ready startWrite then STOPT
    else startRead → Guard(r+1,w)
  else STOPT
  □ if r > 0 then endRead → Guard(r−1, w) else STOPT
  □ if r == 0 and w == 0 then startWrite →
Guard(r, w+1) else STOPT
  □ if w > 0 then endWrite → Guard(r, w−1) else STOPT

Reader = startRead → endRead → Reader
Writer = startWrite → endWrite → Writer

ReadersWriters =
  (||| id : {1..MaxReaders} • Reader)
  ||| ||| id : {1..MaxWriters} • Writer

System =
  let
    alphaGuard =
      {| startWrite, startRead, endWrite, endRead |}
    alphaReadersWriters =
      {| startWrite, startRead, endWrite, endRead |}
  within
    ReadersWriters [ alphaReadersWriters || alphaGuard ]
Guard(0,0)
```

The reason why the simulation is fair to the writers is that if a writer is waiting to write then it will make the startWrite event available. Therefore, the guard process detects this using the *if ready a then P else Q* construct and does not allow any more readers to enter until a writer has done so. This ensures that if a writer wants to enter then no more readers will be allowed to enter until a writer does.

Listing 4.3. The operational semantics of the prefixing and ready testing operators in Lowe's Readyiness-Testing traces model can be defined as follows:

```
Operator PrefixHat(a, P)
  Rule
    ----------------------
    PrefixHat(a, P) =a=> P
  EndRule
  Rule
    ---------------------------------------
    PrefixHat(a,P) =offer.a=> PrefixHat(a,P)
  EndRule
  Rule
```

```
    ------------------------------------------ b <- Sigma, b != a
    PrefixHat(a,P) =notOffer.b=> PrefixHat(a,P)
  EndRule
EndOperator

Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    --------------------------   b <- Sigma
    a -> P =notOffer.b=> a -> P
  EndRule

  Rule
    --------------------------
    a -> P =tau=> PrefixHat(a,P)
  EndRule
EndOperator

Operator TestReady(Q, P, a)
  Syntax Prefix "if ready $3 then $2 else" 12

  Rule
    ----------------------------
    TestReady(Q,P,a) =ready.a=> P
  EndRule

  Rule
    --------------------------------
    TestReady(Q,P,a) =notReady.a=> Q
  EndRule

  Rule
    ------------------------------------------- b <- Sigma
    TestReady(Q,P,a) =notOffer.b=> TestReady(Q,P,a)
  EndRule
EndOperator
```

5. Conclusions

In this Thesis we have adapted Roscoe's model from [Ros08b] to allow it to be relatively efficiently evaluated within FDR. Furthermore, we extended the simulation to allow it to support recursive processes in a way that FDR is able to compile. We then described Tyger, a tool that enables a simulation of a language and process definitions using the language to be specified easily. Lastly we presented a number of simulations that were automatically constructed by Tyger of CSP models from the literature. This was intended to demonstrate that Tyger is exceptionally easy to use and that it enables models of languages to be built with comparative ease.

The biggest issue with Tyger in its current form is the performance of the simulation. Whilst the simulation performs reasonably on small examples its performance on combinatorial problems is not as good as the performance of FDR. This can mostly be put down to FDR's two level compilation strategy: namely that some operators are fully evaluated and have their LTS explicitly constructed whilst others are compiled using *supercombinators* and do not have their LTS fully constructed. Unfortunately, the current definition of *Harness* appears to cause FDR to explicitly construct the LTS of the process that is in the harness. It would therefore be interesting to consider methods of alleviating this problem. The author suspects that the best solutions to this would involve using special simplified operator simulations for operators that, for example, never turn arguments off or on.

Beyond what we have described in this Thesis there are many tasks that Tyger could be put to, some of which would require more alterations than others. One possible use of it would be to build interpreters for *domain specific languages* that allow a particular problem to be expressed succinctly. This could be done by defining CSP-like operators with convenient syntaxes that could manipulate the language. It is worth noting that this would probably require a richer type system for the operational semantics than the one currently implemented; however, this poses no practical problem.

An example of a domain specific language would be that used in Roscoe and Hopkins's shared variable analyser (SVA), as described in [RH07]. SVA takes as input a program that uses shared variables written in a simple sequential language and produces a CSP simulation of the program. It can then check that the program accesses the shared variables properly. The author conjectures that it would be possible to express each of the constructs of the sequential language as a CSP-like operator that could then be simulated using Tyger. This would enable the simulation to be built automatically rather than manually.

There are other possible uses of Tyger that extend beyond its original purpose; for example, the CSP_M type checker could be used as a stand alone program that could be used to typecheck standard CSP_M programs before running them through FDR. The author has, in fact, done this himself several times now and has found it extremely useful; with a bit of work on the error messages it would be ready for general consumption.

Lastly, one slightly more ambitious task would be to make Tyger directly construct the labelled transition system (LTS) and verify it itself rather than outputting CSP for FDR to verify. This would have the potential advantage of reducing the considerable performance overhead that the simulation has. It is likely to be extremely difficult to reach performance that is on par with FDR given the number of years that FDR has had to mature. However, given that Tyger is written in Haskell and should therefore be relatively accessible to most students it may be of pedagogical interest.

Acknowledgements

I would like to thank Prof. Gavin Lowe for both suggesting the project and then guiding me through it. I would especially like to thank him for proof-reading drafts of this Thesis.

References

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS*, 17(4):471–522, 1985.

[For97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2 User Manual*, 1997.

[For01] Formal Systems (Europe) Ltd. *CSP Typechecker*, 2001.

[GE01] Ping Gao and Robert Esser. Polymorphic CSP type checking. *Australasian Computer Science Conference*, 0:156, 2001.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Low96] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[Low09] Gavin Lowe. Extending CSP with tests for availability. *Procedings of Communicating Process Architectures (CPA 2009)*, 2009.

[Low10] Gavin Lowe. Models for CSP with availability information. 2010. draft.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[RH07] A. W. Roscoe and David Hopkins. SVA, a tool for analysing shared-variable programs. In *Proceedings of AVoCS 2007*, pages 177–183, 2007. to appear.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[Ros01] A. W. Roscoe. Compiling Shared Variable Programs into CSP. In *Proceedings of PROGRESS workshop 2001*, 2001.

[Ros08a] A. W. Roscoe. The three platonic models of divergence-strict CSP. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 23–49, Berlin, Heidelberg, 2008. Springer-Verlag.

[Ros08b] A.W. Roscoe. On the expressiveness of CSP. Draft of October 23, 2008, 2008.

[Ros09] A.W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *Journal of Logic and Algebraic Programming*, 78(3):163 – 190, 2009.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[VWPJ06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. *SIGPLAN Not.*, 41(9):251–262, 2006.

Appendix A. Operational Semantics Input Format

In this section we give the input file format of the operational semantics in EBNF form. Due to the fact that the user may give custom syntax for their operators we use `operator application` to denote a valid string corresponding to a valid application of a user operator (e.g. "P [] Q" given the CSP syntax).

```
name ::= letter, { letter | digit  | "'" | "_" } ;
symbol ::= "[" | "]" | "|" | "/" | "\" | "-" | "<" | ">"
  |  "{" | "}" | "~" | "=" | "@"
number ::= digit { digit };
event ::= "tau" | name { ".", name };
pat ::= "(", pat, { pat }, ")" | "{", pat, { ",", pat }, "}"
  | name;
exp ::= "Sigma" | "InductiveCase" | name
  | ? operator application ? | "{", { exp }, "}"
  | "diff(", exp, ",", exp, ")" | "union(", exp, ",", exp, ")"
  | "inter(", exp, ",", exp, ")" | "Set(", exp, ")"
  | "Union(", exp, ")" | "diff(", exp, ",", exp, ")"
  | "{", exp, { ",", exp },
    "|", sidecondition { ",", sidecondition }, "}";
sidecondition ::= pat, "<-", exp | propositionalformula;
propositionalformula ::=
  "member(", exp, ",", exp, ")" | exp, "==", exp
  | exp, "<=", exp | "not", formula
  | formula, "and", formula | formula, "or", formula
  | "false" | "true";
processrelation ::= exp, "=", event, "=>", exp;
inductiverule ::=
  { processrelation }, "-", { "-" }, { sidecondition },
  processrelation;
syntaxDecl ::= "Syntax", ("Binary" | "Prefix" | "Postfix"),
  "\"", { ((letter | digit | symbol) | ("$", number) },"\"",
  number, ("AssocNone" | "AssocLeft"  | "AssocRight");
channel ::= name, [ ":" exp { "." exp } ];
channelsection ::= "Channels", { channel }, "EndChannels";
ruleSection ::= "Rule", inductiveRule, "EndRule";
replicatedOpSection ::=
  "Replicated", ["(", name, { "," name }, ")"],
    "BaseCase(", { pat }, ")", exp, "EndBaseBase",
    "InductiveCase(", { name }, ")", exp, "EndInductiveCase",
  "EndReplicated";
operatorSection ::=
  "Operator", name, ["(", name [":InfRec"],
      { "," name [":InfRec"] } ")"]
    [ syntaxDecl ]
    { ruleSection }, [ replicatedOpSection ],
```

```
  "EndOperator";
```

Also, comments may be included in a operational semantics file as follows:

```
/*
  Multi-line comment
*/
// Single line comment
```

Appendix B. CSP Code

B.1. **CSP Operational Semantics.**

Listing B.1. The standard CSP+ operational semantics (ignoring sequential composition) can be specified in Tyger's input format as follows:

```
Operator CSPSTOP
EndOperator

Operator ExternalChoice(P, Q)
  Syntax Binary "[]" 10 AssocLeft

  Rule
    P =a=> P'
    ------------------   a <- Sigma
    P [] Q =a=> P'
  EndRule

  Rule
    Q =a=> Q'
    -------------      a <- Sigma
    P [] Q =a=> Q'
  EndRule

  Replicated(P)
    Syntax Prefix "[] $0 @" 9

    BaseCase({})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps)
      P [] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    ------------
    a -> P =a=> P
  EndRule
EndOperator

Operator InternalChoice(P, Q)
  Syntax Binary "|~|" 10 AssocLeft

  Rule
    ---------------
    P |~| Q =tau=> P
  EndRule

  Rule
    ---------------
    P |~| Q =tau=> Q
  EndRule

  Replicated(P)
    Syntax Prefix "\$ $0 @" 9

    BaseCase({P})
      P
    EndBaseCase
    InductiveCase(Ps)
      P |~| InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator Hide(P : InfRec, A)
  Syntax Binary "\" 14 AssocLeft

  Rule
    P =a=> P'
    ----------------   a <- diff(Sigma, A)
    P \ A =a=> P' \ A
  EndRule

  Rule
    P =a=> P'
    ----------------   a <- A
    P \ A =tau=> P' \ A
  EndRule
EndOperator

Operator Rename(P : InfRec, A)
  Rule
    P =a=> P'
    ---------------------------- (a,b) <- A
    Rename(P,A) =b=> Rename(P',A)
  EndRule

  Rule
    P =a=> P'
    ---------------------------- a <- diff(Sigma, {a | (a,b) <- A})
    Rename(P,A) =a=> Rename(P',A)
  EndRule
EndOperator

Operator Parallel(P : InfRec, Q : InfRec, A)
  Syntax Binary "[| $3 |]" 12 AssocNone

  Rule
    P =a=> P'
    Q =a=> Q'
    ------------------------------   a <- A
    P [| A |] Q =a=> P' [| A |] Q'
  EndRule

  Rule
    P =a=> P'
    ------------------------------   a <- diff(Sigma, A)
    P [| A |] Q =a=> P' [| A |] Q
  EndRule

  Rule
    Q =a=> Q'
    ------------------------------   a <- diff(Sigma, A)
    P [| A |] Q =a=> P [| A |] Q'
  EndRule
EndOperator

Operator Interleave(P : InfRec, Q : InfRec)
  Syntax Binary "|||" 12 AssocNone

  Rule
    P =a=> P'
    ---------------------- a <- Sigma
    P ||| Q =a=> P' ||| Q
  EndRule

  Rule
    Q =a=> Q'
    ---------------------- a <- Sigma
    P ||| Q =a=> P ||| Q'
  EndRule

  Replicated(P)
    Syntax Prefix "||| $0 @" 9

    BaseCase({})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps)
      P ||| InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator AlphaParallel(P : InfRec, Q : InfRec, A, B)
  Syntax Binary "[ $3 || $4 ]" 12 AssocNone

  Rule
    P =a=> P'
    Q =a=> Q'
    ------------------------------   a <- inter(A,B)
    P [A || B] Q =a=> P' [A || B] Q'
  EndRule

  Rule
    P =a=> P'
    ------------------------------   a <- diff(A, B)
    P [A || B] Q =a=> P' [A || B] Q
  EndRule

  Rule
    Q =a=> Q'
    ------------------------------   a <- diff(B, A)
    P [A || B] Q =a=> P [A || B] Q'
  EndRule

  Replicated(P, A)
    Syntax Prefix "|| $0 @ [$2]" 9

    BaseCase({}, {})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps, As)
      P [A || Union(As)] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator Interrupt(P : InfRec, Q)
  Syntax Binary "/\" 10 AssocNone

  Rule
    P =a=> P'
    ---------------------        a <- Sigma
    P /\ Q =a=> P' /\ Q
  EndRule

  Rule
    Q =a=> Q'
    ----------------          a <- Sigma
    P /\ Q =a=> Q'
  EndRule
EndOperator

Operator Exception(P : InfRec, Q, A)
  Syntax Binary "[| $3 |>" 12 AssocNone

  Rule
    P =a=> P'
```

```
    ----------------------------   a <- diff(Sigma, A)
    P [| A |> Q =a=> P' [| A |> Q
  EndRule

  Rule
    P =a=> P'
    ------------------          a <- A
    P [| A |> Q =a=> Q
  EndRule
EndOperator

Operator Timeout(P, Q)
  Syntax Binary "[>" 10 AssocLeft

  Rule
    P =a=> P'
    -------------  a <- Sigma
    P [> Q =a=> P'
  EndRule

  Rule
    ---------------
    P [> Q =tau=> Q
  EndRule
EndOperator
```

B.2. **Compiled Tyger Script.**

Listing B.2. When B.1 is compiled using Tyger the following file is output:

```
module Operator_M
  -- Gives the set of all sequences of type t of length ≤ length
  FinSeq(t, length) =
    let
      Gen(0) = {<>}
      Gen(n) = {<x>^xs, xs | x ←t, xs ← Gen(n−1)}
    within
      Gen(length)
  FinSeq'(t, length) =
    let
      Gen(0) = <<>>
      Gen(n) = concat(< <<x>^xs,xs> | xs ← Gen(n−1), x ← t>)
    within
      Gen(length)

  powerSeq(<>) = <<>>
  powerSeq(<x>^xs) = <<x>^ys, ys | ys ← powerSeq(xs)>

  zip(<>, _) = <>
  zip(_, <>) = <>
  zip(<x>^xs, <y>^ys) = <(x,y)>^zip(xs, ys)

  flatmap(f,<>) = <>
  flatmap(f,<x>^xs) = f(x)^flatmap(f,xs)

  remdups(x) =
    let
      iter(<>,X) = <>
      iter(<x>^xs,X) =
        if member(x,X) then iter(xs,X)
        else <x>^iter(xs,union(X,{x}))
    within
      iter(x, {})

  foldr(f, e, <>) = e
  foldr(f, e, <x>^xs) = f(x, foldr(f, e, xs))

  foldl(f, e, <>) = e
  foldl(f, e, <x>^xs) = foldl(f, f(e, x), xs)

  --                                  *
  -- Partial functions
  --                                  *

  functionDomain(f) = {x | (x,_) ← f}
  functionDomainSeq(f) = <x | (x,_) ← f>
  functionImage(f) = {x | (_,x) ← f}
  functionImageSeq(f) = <x | (_,x) ← f>
  identityFunction(domain) = {(x,x) | x ← domain}
  identityFunctionSeq(domain) = <(x,x) | x ← domain>
  invert(f) = {(a,b) | (b,a) ← f}
  invertSeq(f) = <(a,b) | (b,a) ← f>

  composeFunctions(fs1, fs2) = {(a, apply(fs1, b)) | (a, b) ← fs2}
  composeFunctionsSeq(fs1, fs2) = <(a,applySeq(fs1,b)) | (a,b) ← fs2>

  apply(f, x) =
    extract({a | (x', a) ← f, x == x'})

  applySeq(f, x) =
    let extract(<x>) = x
    within
      extract(<a | (x', a) ← f, x == x'>)

  mapOverSet(f, X) =
    {apply(f, x) | x ← X}
  mapOverSeq(f, <>) = <>
  mapOverSeq(f, <x>^xs) = <applySeq(f,x)>^mapOverSeq(f,xs)

  seqDiff(xs, ys) = <x | x ← xs, not elem(x,ys)>
  seqInter(xs, ys) = <x | x ← xs, elem(x, ys)>
  seqUnion(xs, ys) = remdups(xs^ys)

  --                                  *
  -- Semantics Calculation
  --                                  *

  -- Returns a partial function from (op, onProcMap, procCount, offProcMap) to
  -- the possible internal events
  InternalEventsFromOperator(op, onProcMap, procCount, offProcMap, nextId, doneCalls) =
    let
      -- The sequence of all possible events that each rule gives
      (possibleEvents, nextId') =
        let
          process((events, nextId),
              rule @@ (phi, x, mu, f, xi, chi, discards)) =
            let
              procsToDiscard =
                mapOverSeq(onProcMap, discards)
              procEvents =
                <(applySeq(onProcMap, p), e) | (p, e) ← phi>
              newProcs = composeFunctionsSeq(offProcMap, f)
              thisEvent =
                (rule, procEvents, x, procsToDiscard, procCount, newProcs)
            within
              (<(nextId,thisEvent)>^events, nextId+1)
        within
          foldl(process, (<>, nextId), Rules(op))

      -- The sequence of recursive calls to this function to make,
      -- it contains no duplicates and no items in doneCalls.
      recursiveCallsToMake =
        seqDiff(
          remdups(<
            let
              newOnProcMap =
                composeFunctionsSeq(
                  concat(<onProcMap,
                      identityFunctionSeq(
                        <procCount..procCount+newProcCount−1>)
                      >),
                      xi)
              newProcCount = procCount + length(f)
              newOffProcMap =
                composeFunctionsSeq(offProcMap, chi)
            within
              (mu, newOnProcMap, newProcCount, newOffProcMap)
            | (phi, x, mu, f, xi, chi, discards) ← Rules(op), op ≠ mu>),
          doneCalls)

      doneCalls' = doneCalls^recursiveCallsToMake

      (recursiveEvents, recursiveDiscardableArgs, nextId'') =
        let
          process((events, discardableArgs, nextId), (mu, xi, m, chi)) =
            let
              (events', discardableArgs', nextId') =
                InternalEventsFromOperator(mu, xi, m, chi, nextId, doneCalls')
            within
              (events'^events, remdups(discardableArgs'^discardableArgs), nextId')
        within
          foldl(process, (<>, <>, nextId'), recursiveCallsToMake)
    within
      (<((op, onProcMap, procCount, offProcMap), possibleEvents)>^recursiveEvents,
        remdups(mapOverSeq(onProcMap, DiscardableArgs(op))^recursiveDiscardableArgs),
        nextId'')
  Rules(Op_Identity) =
      concat(<<(<(0, a0)>, a0, Op_Identity, <>, <(0, 0)>, <>, <>)
                | (a0) ← seq ({(a0) | a0 ← SystemEvents})>,
             <(<(0, callProc.p0)>, callProc.p0, Op_Identity, <>, <(0, 0)>, <>,
               <>)
              | (p0) ← seq ({(p0) | p0 ← ProcArgs})>>)
  Rules(Op_CSPSTOP) = concat(<>)
  Rules(Op_ExternalChoice) =
      concat(<<(<(0, a0)>, a0, Op_Identity, <>, <(0, 0)>, <>, <1>)
```

```
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>,
          <(<(1, a0)>, a0, Op_Identity, <>, <(0, 1)>, <>, <0>)
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>>)
Rules(Op_Prefix.a0) =
    concat(<<(<>, a0, Op_Identity, <(0, -1)>, <(0, 0)>, <>, <>)>>)
Rules(Op_InternalChoice) =
    concat(<<(<>, tau, Op_Identity, <(0, -2)>, <(0, 0)>, <>, <>)>,
           <(<>, tau, Op_Identity, <(0, -1)>, <(0, 0)>, <>, <>)>>)
Rules(Op_Hide.A0) =
    concat(<<(<(0, a0)>, a0, Op_Hide.A0, <>, <(0, 0)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← diff(UserEvents, A0)})>,
           <(<(0, a0)>, tau, Op_Hide.A0, <>, <(0, 0)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← A0})>>)
Rules(Op_Rename.A0) =
    concat(<<(<(0, a0)>, b0, Op_Rename.A0, <>, <(0, 0)>, <>, <>)
            | (a0, b0) ← seq ({(a0, b0) | (a0, b0) ← A0})>,
           <(<(0, a0)>, a0, Op_Rename.A0, <>, <(0, 0)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← diff(UserEvents, {a0 | (a0,
                                                        b0) ← A0})})>>)
Rules(Op_Parallel.A0) =
    concat(<<(<(0, a0), (1, a0)>, a0, Op_Parallel.A0, <>,
              <(0, 0), (1, 1)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← A0})>,
           <(<(0, a0)>, a0, Op_Parallel.A0, <>, <(0, 0), (1, 1)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← diff(UserEvents, A0)})>,
           <(<(1, a0)>, a0, Op_Parallel.A0, <>, <(0, 0), (1, 1)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← diff(UserEvents, A0)})>>)
Rules(Op_Interleave) =
    concat(<<(<(0, a0)>, a0, Op_Interleave, <>, <(0, 0), (1, 1)>, <>,
              <>)
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>,
           <(<(1, a0)>, a0, Op_Interleave, <>, <(0, 0), (1, 1)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>>)
Rules(Op_AlphaParallel.A0.B0) =
    concat(<<(<(0, a0), (1, a0)>, a0, Op_AlphaParallel.A0.B0, <>,
              <(0, 0), (1, 1)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← inter(A0, B0)})>,
           <(<(0, a0)>, a0, Op_AlphaParallel.A0.B0, <>, <(0, 0), (1, 1)>, <>,
             <>)
            | (a0) ← seq ({(a0) | a0 ← diff(A0, B0)})>,
           <(<(1, a0)>, a0, Op_AlphaParallel.A0.B0, <>, <(0, 0), (1, 1)>, <>,
             <>)
            | (a0) ← seq ({(a0) | a0 ← diff(B0, A0)})>>)
Rules(Op_Interrupt) =
    concat(<<(<(0, a0)>, a0, Op_Interrupt, <>, <(0, 0), (1, 1)>, <>,
              <>)
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>,
           <(<(1, a0)>, a0, Op_Identity, <>, <(0, 1)>, <>, <0>)
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>>)
Rules(Op_Exception.A0) =
    concat(<<(<(0, a0)>, a0, Op_Exception.A0, <>, <(0, 0)>, <(-1, -1)>,
              <>)
            | (a0) ← seq ({(a0) | a0 ← diff(UserEvents, A0)})>,
           <(<(0, a0)>, a0, Op_Identity, <(0, -1)>, <(0, 0)>, <>, <0>)
            | (a0) ← seq ({(a0) | a0 ← A0})>>)
Rules(Op_Timeout) =
    concat(<<(<(0, a0)>, a0, Op_Identity, <>, <(0, 0)>, <>, <>)
            | (a0) ← seq ({(a0) | a0 ← UserEvents})>,
           <(<>, tau, Op_Identity, <(0, -1)>, <(0, 0)>, <>, <0>)>>)
DiscardableArgs(Op_Identity) = <>
DiscardableArgs(Op_CSPSTOP) = <>
DiscardableArgs(Op_ExternalChoice) = <1, 0>
DiscardableArgs(Op_Prefix.a0) = <>
DiscardableArgs(Op_InternalChoice) = <>
DiscardableArgs(Op_Hide.A0) = <>
DiscardableArgs(Op_Rename.A0) = <>
DiscardableArgs(Op_Parallel.A0) = <>
DiscardableArgs(Op_Interleave) = <>
DiscardableArgs(Op_AlphaParallel.A0.B0) = <>
DiscardableArgs(Op_Interrupt) = <0>
DiscardableArgs(Op_Exception.A0) = <0>
DiscardableArgs(Op_Timeout) = <0>
exports
  SystemEvents = union(UserEvents, {| |})

  -- Allows the rules to specify a tau transition
  channel tau
  -- Let a process be turned off
  channel off
  -- Represents a primed event; used to let processes turn themselves off
  channel prime : SystemEvents
  channel renamed : {0..20000}

  --                                  *
  -- Main Simulator Function
  --                                  *

  -- startOperator operator the process starts in
  -- onProcesses   sequence of processes that are initially on
  -- offProcesses  sequence of processes that are initially off
  Operator(startOperator, onProcesses, offProcesses) =
    let
      OnProcessCount = length(onProcesses)
      OnProcesses =
        zip(<0..OnProcessCount - 1>, onProcesses)
      OffProcessCount = length(offProcesses)
      OffProcesses =
        zip(<(-OffProcessCount)..-1>, offProcesses)

      InternalEvents =
        concat(<es | (id, es) ← InternalEventsByOperator>)

      (InternalEventsByOperator, discardableProcs, _) =
        InternalEventsFromOperator(startOperator,
          identityFunctionSeq(<0..OnProcessCount-1>), OnProcessCount,
          identityFunctionSeq(<(-OffProcessCount)..-1>),
          0, <>)

      RenamingsForProc(id) =
        let
          calc((rid, (rule, procs,b,discards,m,f))) =
            if elem(id, functionDomainSeq(procs)) then
              if elem(id,discards) then
                <(prime.applySeq(procs, id), rid)>
              else
                <(applySeq(procs, id), rid)>
            else
              if elem(id,discards) then
                <(off, rid)>
              else <>
        within
          flatmap(calc,InternalEvents)

      Process(proc, id) =
        (if elem(id, discardableProcs) then
          explicate((proc⟦a ← prime.a, a ← a | a ← SystemEvents⟧
          [| {| prime |} ▷ STOP)
          △ off → STOP)
        else
          proc)
        ⟦a ← renamed.b | (a,b) ← set(RenamingsForProc(id))⟧

      -- onProcMap  Function from current process id to actual process id
      -- procCount  Current number of on processes
      -- offProcMap Function from current off process id to actual off
      --            process ids
      Reg(currentOperator, onProcMap, procCount, offProcMap) =

        □(rid, (rule @@ (phi, x, mu, f, xi, chi, discards), _, _, _, _, _)) :

          set(applySeq(InternalEventsByOperator,
                (currentOperator, onProcMap, procCount, offProcMap))) •
          let
            procsToDiscard =
              mapOverSeq(onProcMap, discards)
            procEvents =
              <(applySeq(onProcMap, p), e) | (p, e) ← phi>
            newOnProcMap =
              composeFunctionsSeq(concat(<onProcMap,
                          identityFunctionSeq(<procCount..procCount+newProcCount-1>)>)
                        , xi)
            newProcCount = procCount + length(newProcs)
            newProcs = composeFunctionsSeq(offProcMap, f)
            newOffProcMap =
              composeFunctionsSeq(offProcMap, chi)
          within
            renamed.rid →
              if length(newProcs) == 0 then
                Reg(mu, newOnProcMap, newProcCount, newOffProcMap)
              else
                ((|| id : {procCount..newProcCount-1}
                • [AlphaProcess(id)]
                  Process(applySeq(OffProcesses,
                      applySeq(newProcs, id-procCount)), id))
                [| AlphaProcesses(newProcCount) |]
                Reg(mu, newOnProcMap, newProcCount, newOffProcMap))

      AlphaProcess(id) =
        set(<renamed.b | (a,b) ← RenamingsForProc(id)>)
      -- Important: /= Renamings beacuse there could be events that
      -- happen because of no processes events (cf internal choice).
      AlphaProcesses(maxId) = Union({AlphaProcess(id) | id ← {0..maxId-1}})

      H = {tau}
```

```
    within
      ((( || id : {0..OnProcessCount-1} @
        [AlphaProcess(id)] Process(applySeq(OnProcesses, id), id))
      )
      [| AlphaProcesses(OnProcessCount) |]
      Reg(startOperator, identityFunctionSeq(<0..OnProcessCount-1>),
        OnProcessCount, identityFunctionSeq(<(-OffProcessCount)..-1>))
      )[[renamed.r <- b
        | (r,b) <-
          set(<(r,b) | (r,(_,_,b,_,_,_)) <- InternalEvents>)]]
      \ H
endmodule
--                                      *
-- User Callable Functions
--                                      *

transparent explicate, diamond

--                                      *
-- Recursion control procedures
--                                      *
channel callProc, startProc : ProcArgs

CallProc(proc) = callProc.proc -> STOP

WrapThread(proc) =
  let
    RecursionRegulator =
      callProc?p -> startProc!p -> RecursionRegulator

      []□ e : Operator_M::SystemEvents @ e -> RecursionRegulator

    Thread(proc) =
      GetProc(proc)
      [| {| callProc |} |>
      startProc?proc' -> Thread(proc')
  within
    -- diamond removes the tau's from the resulting LTS (but never
    -- increases the size of the resulting LTS, cf. normalize)
    -- This removes the problem of the new events being introduced.
    diamond(
      (Thread(proc) [| union(Operator_M::SystemEvents, {| callProc, startProc |}) |] RecursionRegulator)
      \ {| startProc, callProc |}
    )
--                                      *
-- Operators
--                                      *
datatype Operators =
    Op_Identity
    | Op_CSPSTOP
    | Op_ExternalChoice
    | Op_Prefix.UserEvents
    | Op_InternalChoice
    | Op_Hide.Set(UserEvents)
    | Op_Rename.Set({(a, b) | a <- UserEvents, b <- UserEvents})
    | Op_Parallel.Set(UserEvents)
    | Op_Interleave
    | Op_AlphaParallel.Set(UserEvents).Set(UserEvents)
    | Op_Interrupt
    | Op_Exception.Set(UserEvents)
    | Op_Timeout
Identity'(P0) = Operator_M::Operator(Op_Identity, <P0>, <>)
CSPSTOP' = Operator_M::Operator(Op_CSPSTOP, <>, <>)
ExternalChoice'(P0, Q0) =
    Operator_M::Operator(Op_ExternalChoice, <P0, Q0>, <>)
Prefix'(a0, P0) = Operator_M::Operator(Op_Prefix.a0, <>, <P0>)
InternalChoice'(P0, Q0) =
    Operator_M::Operator(Op_InternalChoice, <>, <P0, Q0>)
Hide'(P0, A0) = Operator_M::Operator(Op_Hide.A0, <P0>, <>)
Rename'(P0, A0) = Operator_M::Operator(Op_Rename.A0, <P0>, <>)
Parallel'(P0, Q0, A0) =
    Operator_M::Operator(Op_Parallel.A0, <P0, Q0>, <>)
Interleave'(P0, Q0) =
    Operator_M::Operator(Op_Interleave, <P0, Q0>, <>)
AlphaParallel'(P0, Q0, A0, B0) =
    Operator_M::Operator(Op_AlphaParallel.A0.B0, <P0, Q0>, <>)
Interrupt'(P0, Q0) =
    Operator_M::Operator(Op_Interrupt, <P0, Q0>, <>)
Exception'(P0, Q0, A0) =
    Operator_M::Operator(Op_Exception.A0, <P0>, <Q0>)
Timeout'(P0, Q0) = Operator_M::Operator(Op_Timeout, <P0>, <Q0>)
ReplicatedExternalChoice'(<>) = CSPSTOP'
ReplicatedExternalChoice'(<P0>^Ps0) =
    ExternalChoice'(P0, ReplicatedExternalChoice'(Ps0))
ReplicatedInternalChoice'(<P>) = P0
ReplicatedInternalChoice'(<P0>^Ps0) =
    InternalChoice'(P0, ReplicatedInternalChoice'(Ps0))
ReplicatedInterleave'(<>) = CSPSTOP'
ReplicatedInterleave'(<P0>^Ps0) =
    Interleave'(P0, ReplicatedInterleave'(Ps0))
ReplicatedAlphaParallel'(<>, <>) = CSPSTOP'
ReplicatedAlphaParallel'(<P0>^Ps0, <A0>^As0) =
    AlphaParallel'(P0, ReplicatedAlphaParallel'(Ps0, As0), A0,
                   Union(set(As0)))
```

B.3. **Singleton Availability Operational Semantics.**

Listing B.3. The operational semantics of Lowe's singleton availability model as defined in [Low10] can be specified in Tyger's input format as follows:

```
Channels
  offer : Sigma
EndChannels

Operator CSPSTOP
EndOperator

Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    -------------------------
    a -> P =offer.a=> a -> P
  EndRule
  Rule
    -------------
    a -> P =a=> P
  EndRule
EndOperator

Operator ExternalChoice(P, Q)
  Syntax Binary "[]" 10 AssocLeft

  Rule
    P =a=> P'
    --------------------   a <- Sigma
    P [] Q =a=> P'
  EndRule

  Rule
    Q =a=> Q'
    --------------       a <- Sigma
    P [] Q =a=> Q'
  EndRule

  Rule
    P =offer.a=> P'
    -------------------------   a <- Sigma
    P [] Q =offer.a=> P' [] Q
  EndRule
  Rule
    Q =offer.a=> Q'
    -------------------------   a <- Sigma
    P [] Q =offer.a=> P [] Q'
  EndRule

  Replicated(P)
    Syntax Prefix "[] $0 @" 9

    BaseCase({})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps)
      P [] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator InternalChoice(P, Q)
  Syntax Binary "|~|" 10 AssocLeft

  Rule
    ----------------
    P |~| Q =tau=> P
  EndRule

  Rule
    ----------------
    P |~| Q =tau=> Q
  EndRule
EndOperator

Operator Hide(P : InfRec, A)
  Syntax Binary "\" 14 AssocLeft

  Rule
    P =a=> P'
    -----------------   a <- diff(Sigma, A)
    P \ A =a=> P' \ A
  EndRule

  Rule
    P =a=> P'
    -----------------   a <- A
    P \ A =tau=> P' \ A
  EndRule
```

```
EndOperator

Operator Parallel(P : InfRec, Q : InfRec, A)
  Syntax Binary "[| $3 |]" 12 AssocNone

  Rule
    P =a=> P'
    Q =a=> Q'
    ------------------------------    a <- A
    P [| A |] Q =a=> P' [| A |] Q'
  EndRule
  Rule
    P =offer.a=> P'
    Q =offer.a=> Q'
    ------------------------------    a <- A
    P [| A |] Q =offer.a=> P' [| A |] Q'
  EndRule

  Rule
    P =a=> P'
    ------------------------------    a <- diff(Sigma, A)
    P [| A |] Q =a=> P' [| A |] Q
  EndRule
  Rule
    P =offer.a=> P'
    ------------------------------    a <- diff(Sigma, A)
    P [| A |] Q =offer.a=> P' [| A |] Q
  EndRule

  Rule
    Q =a=> Q'
    ------------------------------    a <- diff(Sigma, A)
    P [| A |] Q =a=> P [| A |] Q'
  EndRule
  Rule
    Q =offer.a=> Q'
    ------------------------------    a <- diff(Sigma, A)
    P [| A |] Q =offer.a=> P [| A |] Q'
  EndRule
EndOperator

Operator Interleave(P : InfRec, Q : InfRec)
  Syntax Binary "|||" 12 AssocNone

  Rule
    P =a=> P'
    ---------------------- a <- Sigma
    P ||| Q =a=> P' ||| Q
  EndRule
  Rule
    P =offer.a=> P'
    ---------------------- a <- Sigma
    P ||| Q =offer.a=> P' ||| Q
  EndRule

  Rule
    Q =a=> Q'
    ---------------------- a <- Sigma
    P ||| Q =a=> P ||| Q'
  EndRule
  Rule
    Q =offer.a=> Q'
    ---------------------- a <- Sigma
    P ||| Q =offer.a=> P ||| Q'
  EndRule

  Replicated(P)
    Syntax Prefix "||| $0 @" 9

    BaseCase({})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps)
      P ||| InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator
```

B.4. **Set Availability Operational Semantics.**

Listing B.4. The operational semantics of Lowe's set availability model as defined in [Low10] can be specified in Tyger's input format as follows:

```
Channels
  offer : Set(Sigma)
EndChannels

Operator CSPSTOP
  Rule
    --------------------------
    CSPSTOP =offer.{}=> CSPSTOP
  EndRule
EndOperator

Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    --------------------------
    a -> P =offer.{a}=> a -> P
  EndRule
  Rule
    --------------------------
    a -> P =offer.{}=> a -> P
  EndRule
  Rule
    -------------
    a -> P =a=> P
  EndRule
EndOperator

Operator ExternalChoice(P, Q)
  Syntax Binary "[]" 10 AssocLeft

  Rule
    P =a=> P'
    -------------------   a <- Sigma
    P [] Q =a=> P'
  EndRule

  Rule
    Q =a=> Q'
    ---------------       a <- Sigma
    P [] Q =a=> Q'
  EndRule

  Rule
    P =offer.A=> P'
    Q =offer.B=> Q'
    ------------------------------------  A <- Set(Sigma), B <- Set(Sigma)
    P [] Q =offer.union(A,B)=> P' [] Q'
  EndRule

  Replicated(P)
    Syntax Prefix "[] $0 @" 9

    BaseCase({})
      CSPSTOP
    EndBaseCase
    InductiveCase(Ps)
      P [] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator InternalChoice(P, Q)
  Syntax Binary "|~|" 10 AssocLeft

  Rule
    ----------------
    P |~| Q =tau=> P
  EndRule

  Rule
    ----------------
    P |~| Q =tau=> Q
  EndRule
EndOperator

Operator AlphaParallel(P, Q, A, B)
  Syntax Binary "[ $3 || $4 ]" 12  AssocNone

  Rule
    P =offer.X=> P'
    Q =offer.Y=> Q'
    ------------------------------------------------
      X <- Set(Sigma), Y <- Set(Sigma), X <= A, Y <= B,
      inter(X,A) == inter(Y,B)
    P [ A || B ] Q =offer.union(A,B)=> P' [ A || B ] Q'
  EndRule

  Rule
    P =a=> P'
    Q =a=> Q'
    ------------------------------------ a <- inter(A,B)
    P [ A || B ] Q =a=> P' [ A || B ] Q'
  EndRule
  Rule
    P =a=> P'
    ------------------------------------ a <- diff(A,B)
    P [ A || B ] Q =a=> P' [ A || B ] Q
  EndRule
  Rule
    Q =a=> Q'
    ------------------------------------ a <- diff(B,A)
    P [ A || B ] Q =a=> P [ A || B ] Q'
  EndRule
EndOperator
```

B.5. **Readyness Testing Operational Semantics.**

Listing B.5. The operational semantics of Lowe's Readyiness-Testing model as defined in [Low09] can be specified in Tyger's input format as follows:

```
Channels
  ready : Sigma
  notReady : Sigma
  offer : Sigma
  notOffer : Sigma
```

```
EndChannels

Operator STOPT
  Rule
    ----------------------- a <- Sigma
    STOPT =notOffer.a=> STOPT
  EndRule
EndOperator

Operator PrefixHat(a, P)
  Rule
    ----------------------
    PrefixHat(a, P) =a=> P
  EndRule
  Rule
    ----------------------------------------
    PrefixHat(a,P) =offer.a=> PrefixHat(a,P)
  EndRule
  Rule
    ---------------------------------------- b <- Sigma, b != a
    PrefixHat(a,P) =notOffer.b=> PrefixHat(a,P)
  EndRule
EndOperator

Operator Prefix(a, P)
  Syntax Binary "->" 8 AssocRight

  Rule
    --------------------------   b <- Sigma
    a -> P =notOffer.b=> a -> P
  EndRule

  Rule
    --------------------------
    a -> P =tau=> PrefixHat(a,P)
  EndRule
EndOperator


Operator InternalChoice(P, Q)
  Syntax Binary "|~|" 10 AssocLeft

  Rule
    ----------------
    P |~| Q =tau=> P
  EndRule
  Rule
    ----------------
    P |~| Q =tau=> Q
  EndRule
EndOperator

Operator ExternalChoice(P, Q)
  Syntax Binary "[]" 10 AssocLeft

  Rule
    P =a=> P'
    -------------------   a <- Sigma
    P [] Q =a=> P'
  EndRule

  Rule
    Q =a=> Q'
    --------------      a <- Sigma
    P [] Q =a=> Q'
  EndRule

  Rule
    P =offer.a=> P'
    -------------------------   a <- Sigma
    P [] Q =offer.a=> P' [] Q
  EndRule
  Rule
    P =ready.a=> P'
    -------------------------   a <- Sigma
    P [] Q =ready.a=> P' [] Q
  EndRule
  Rule
    P =notReady.a=> P'
    --------------------------  a <- Sigma
    P [] Q =notReady.a=> P' [] Q
  EndRule

  Rule
    Q =offer.a=> Q'
    -------------------------   a <- Sigma
    P [] Q =offer.a=> P [] Q'
  EndRule
  Rule
    Q =ready.a=> Q'
    -------------------------   a <- Sigma
    P [] Q =ready.a=> P [] Q'
  EndRule
  Rule
    Q =notReady.a=> Q'
    --------------------------  a <- Sigma
    P [] Q =notReady.a=> P [] Q'
  EndRule

  Rule
    P =notOffer.a=> P'
    Q =notOffer.a=> Q'
    ----------------------------  a <- Sigma
    P [] Q =notOffer.a=> P' [] Q'
  EndRule

  Replicated(P)
    Syntax Prefix "[] $0 @" 9

    BaseCase({})
      STOPT
    EndBaseCase
    InductiveCase(Ps)
      P [] InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator TestReady(Q, P, a)
  Syntax Prefix "if ready $3 then $2 else" 12

  Rule
    --------------------------------
    TestReady(Q,P,a) =ready.a=> P
  EndRule

  Rule
    --------------------------------
    TestReady(Q,P,a) =notReady.a=> Q
  EndRule

  Rule
    ------------------------------------------------   b <- Sigma
    TestReady(Q,P,a) =notOffer.b=> TestReady(Q,P,a)
  EndRule
EndOperator

Operator Interleave(P : InfRec, Q : InfRec)
  Syntax Binary "|||" 12 AssocNone

  Rule
    P =a=> P'
    --------------------- a <- Sigma
    P ||| Q =a=> P' ||| Q
  EndRule
  Rule
    Q =a=> Q'
    --------------------- a <- Sigma
    P ||| Q =a=> P ||| Q'
  EndRule
  Rule
    P =offer.a=> P'
    ---------------------------- a <- Sigma
    P ||| Q =offer.a=> P' ||| Q
  EndRule
  Rule
    Q =offer.a=> Q'
    ---------------------------- a <- Sigma
    P ||| Q =offer.a=> P ||| Q'
  EndRule
  Rule
    P =notOffer.a=> P'
    Q =notOffer.a=> Q'
    ------------------------------- a <- Sigma
    P ||| Q =notOffer.a=> P' ||| Q'
  EndRule

  Rule
    P =ready.a=> P'
    ------------------------------- a <- Sigma
    P ||| Q =ready.a=> P' ||| Q
  EndRule
  Rule
    Q =ready.a=> Q'
    ------------------------------- a <- Sigma
    P ||| Q =ready.a=> P ||| Q'
  EndRule
  Rule
    P =notReady.a=> P'
    ------------------------------- a <- Sigma
    P ||| Q =notReady.a=> P' ||| Q
  EndRule
  Rule
    Q =notReady.a=> Q'
    ------------------------------- a <- Sigma
    P ||| Q =notReady.a=> P ||| Q'
  EndRule

  Replicated(P)
    Syntax Prefix "||| $0 @" 9

    BaseCase({})
      STOPT
    EndBaseCase
    InductiveCase(Ps)
      P ||| InductiveCase
    EndInductiveCase
  EndReplicated
EndOperator

Operator AlphaParallel(P : InfRec, Q : InfRec, A, B)
  Syntax Binary "[ $3 || $4 ]" 12 AssocNone

  Rule
    P =a=> P'
    Q =a=> Q'
    -------------------------------    a <- inter(A,B)
    P [ A || B ] Q =a=> P' [ A || B ] Q'
  EndRule
```

```
Rule
  P =offer.a=> P'
  Q =offer.a=> Q'
  ------------------------------    a <- inter(A,B)
  P [ A || B ] Q =a=> P' [ A || B ] Q'
EndRule

Rule
  P =ready.a=> P'
  Q =offer.a=> Q'
  ------------------------------------    a <- A
  P [ A || B ] Q =ready.a=> P' [ A || B ] Q'
EndRule

Rule
  P =a=> P'
  ------------------------------    a <- diff(A, B)
  P [ A || B ] Q =a=> P' [ A || B ] Q
EndRule
Rule
  P =offer.a=> P'
  ----------------------------------   a <- diff(A, B)
  P [ A || B ] Q =offer.a=> P' [ A || B ] Q
EndRule
Rule
  P =notOffer.a=> P'
  ---------------------------------------    a <- diff(A, B)
  P [ A || B ] Q =notOffer.a=> P' [ A || B ] Q
EndRule
Rule
  P =ready.b=> P'
  ----------------------------------   b <- diff(Sigma, B)
  P [ A || B ] Q =ready.b=> P' [ A || B ] Q
EndRule
Rule
  P =notReady.b=> P'
  --------------------------------------  b <- diff(Sigma, B)
  P [ A || B ] Q =notReady.b=> P' [ A || B ] Q
EndRule

Rule
  Q =a=> Q'
  ------------------------------    a <- diff(B, A)
  P [ A || B ] Q =a=> P [ A || B ] Q'
EndRule
Rule
  Q =offer.a=> Q'
  ----------------------------------   a <- diff(B, A)
  P [ A || B ] Q =offer.a=> P [ A || B ] Q'
EndRule
Rule
  Q =notOffer.a=> Q'
  ---------------------------------------    a <- diff(B, A)
  P [ A || B ] Q =notOffer.a=> P [ A || B ] Q'
EndRule
Rule
  Q =ready.b=> Q'
  ----------------------------------   b <- diff(Sigma, A)
  P [ A || B ] Q =ready.b=> P [ A || B ] Q'
EndRule
Rule
  Q =notReady.b=> Q'
  --------------------------------------  b <- diff(Sigma, A)
  P [ A || B ] Q =notReady.b=> P [ A || B ] Q'
EndRule

Rule
  P =ready.b=> P'
  Q =offer.b=> Q'
  ------------------------------------------ b <- B
  P [ A || B ] Q =ready.b=> P' [ A || B ] Q'
EndRule
Rule
  P =offer.b=> P'
  Q =ready.b=> Q'
  ------------------------------------------ b <- A
  P [ A || B ] Q =ready.b=> P' [ A || B ] Q'
EndRule

  Rule
    P =notReady.b=> P'
    Q =notOffer.b=> Q'
    ------------------------------------------ b <- B
    P [ A || B ] Q =tau=> P' [ A || B ] Q'
  EndRule
  Rule
    P =notOffer.b=> P'
    Q =notReady.b=> Q'
    ------------------------------------------ b <- A
    P [ A || B ] Q =tau=> P' [ A || B ] Q'
  EndRule

  Rule
    P =notReady.b=> P'
    Q =offer.b=> Q'
    ------------------------------------------ b <- B
    P [ A || B ] Q =notReady.b=> P' [ A || B ] Q'
  EndRule
  Rule
    P =offer.b=> P'
    Q =notReady.b=> Q'
    ------------------------------------------ b <- A
    P [ A || B ] Q =notReady.b=> P' [ A || B ] Q'
  EndRule

  Rule
    ------------------------------------------ d <- diff(Sigma, union(A,B))
    P [ A || B ] Q =notOffer.d=> P [ A || B ] Q
  EndRule
EndOperator

Operator Hide(P : InfRec, A)
  Syntax Binary "\" 14 AssocLeft

  Rule
    P =a=> P'
    ------------------   a <- diff(Sigma, A)
    P \ A =a=> P' \ A
  EndRule
  Rule
    P =ready.a=> P'
    ------------------   a <- diff(Sigma, A)
    P \ A =ready.a=> P' \ A
  EndRule
  Rule
    P =notReady.a=> P'
    ------------------   a <- diff(Sigma, A)
    P \ A =notReady.a=> P' \ A
  EndRule
  Rule
    P =notOffer.a=> P'
    ------------------   a <- diff(Sigma, A)
    P \ A =notOffer.a=> P' \ A
  EndRule
  Rule
    P =offer.a=> P'
    ------------------   a <- diff(Sigma, A)
    P \ A =offer.a=> P' \ A
  EndRule

  Rule
    P =a=> P'
    ------------------   a <- A
    P \ A =tau=> P' \ A
  EndRule
  Rule
    P =ready.a=> P'
    ------------------   a <- A
    P \ A =tau=> P' \ A
  EndRule

  Rule
    ------------------------- b <- A
    P \ A =notOffer.b=> P \ A
  EndRule
EndOperator
```

Appendix C. Tyger Source Code

C.1. **Utility Files.**

C.1.1. *OperatorParsers.hs.*

```
module OperatorParsers where

import OpSemDataStructures

import Char
import Control.Monad.Identity
import Control.Monad(liftM, sequence)
import List
import Text.Parsec
import Text.Parsec.Char
import qualified Text.Parsec.Expr as E

constructParseTable :: (Monad m) =>
  [(Integer, E.Operator String u m a)] ->
  [(Name, Maybe OperatorSyntax)] ->
  [(Name, Maybe OperatorSyntax)] ->
  ParsecT String u m (Name -> [a] -> a) ->
  ParsecT String u m (Name -> [b] -> [a] -> a) ->
  ParsecT String u m a ->
  ParsecT String u m b ->
  ParsecT String u m a
constructParseTable builtInOps ops replicatedOps semAction replicatedSemAction
        expParser generatorParser =
  let
    constructParser comps =
      let
        parser (Argument n) =
          do
            e <- (infixParser <|> expParser)
            return [(n, e)]
        parser (String s) =
          do
            string s
            skipMany (satisfy isSpace)
            return []
        processResults =
          map snd . sortBy (\(n1,_) (n2,_) -> compare n1 n2) . concat
      in liftM processResults (sequence . map parser $ comps)

    constructReplicatedParser comps =
      let
        parser (Argument 0) =
          do
            gens <- sepBy generatorParser (string ",")
            return (gens, [])
        parser (Argument n) =
          do
            e <- (infixParser <|> expParser)
            return ([], [(n, e)])
        parser (String s) =
          do
            string s
            skipMany (satisfy isSpace)
            return ([], [])
        processResults =
          map snd . sortBy (\(n1,_) (n2,_) -> compare n1 n2) . concat
      in do
        (gens, args) <- liftM unzip (sequence . map parser $ comps)
        return (concat gens, processResults args)

    convAssociativity AssocLeft = E.AssocLeft
    convAssociativity AssocRight = E.AssocRight
    convAssociativity AssocNone = E.AssocNone

    getPriority (InfixOp _ n _) = n
    getPriority (PrefixOp _ n) = n
    getPriority (PostfixOp _ n) = n

    operatorParser1 n pat =
      do
        args <- constructParser pat
        res <- semAction
        return (\ e1 -> res n (e1:args))
    operatorParser2 n pat =
      do
        args <- constructParser pat
        res <- semAction
        return (\ e1 e2 -> res n (e1:e2:args))

    replicatedOperatorParser n pat =
      do
        (gens, args) <- constructReplicatedParser pat
        res <- replicatedSemAction
        return (\ e1 -> res n gens (e1:args))

    convOperator (n, InfixOp pat _ assoc) =
      E.Infix (try (operatorParser2 n pat)) (convAssociativity assoc)
    convOperator (n, PrefixOp pat _) =
      E.Prefix (try (operatorParser1 n pat))
    convOperator (n, PostfixOp pat _) =
      E.Postfix (try (operatorParser1 n pat))

    convReplicatedOperator (n, PrefixOp pat _) =
      E.Prefix (try (replicatedOperatorParser n pat))
    convReplicatedOperator (n, PostfixOp pat _) =
      E.Postfix (try (replicatedOperatorParser n pat))

    allOperators =
      [(getPriority s, convReplicatedOperator (n, s))
        | (n, Just s) <- replicatedOps]
      ++[(getPriority s, convOperator (n, s)) |  (n, Just s) <- ops]
      ++builtInOps

    operatorTable = (map (map snd)
        . sortBy (\ ((e1, _):_) ((e2, _):_) -> compare e1 e2)
        . groupBy (\ (e1,_) (e2,_) -> e1 == e2)) allOperators

    infixParser = E.buildExpressionParser operatorTable expParser
  in
    infixParser
```

C.1.2. *Main.hs.*

```
module Main where

import Control.Monad.Trans
import System.Directory
import System.FilePath
import System.Exit
import Text.PrettyPrint.HughesPJ

import ConstantCode
import CSPMDataStructures
import CSPMParser
import CSPMPrettyPrinter
import CSPMRecursionRefactorings
import CSPMTypeChecker.TCModule
import CSPMTypeChecker.TCMonad
import OpSemRules
import OpSemParser
import OpSemTypeChecker
import Util

main :: IO ()
main =
  do
    res <- runTyger (tygerMain "../Examples/SingletonAvailabilityTesting.opsem"
                    "../Examples/SingletonAvailabilityTestingExample.csp")
    case res of
      Left err  -> putStrLn (show err) >> exitFailure
      Right _   -> exitSuccess

interactiveMain :: FilePath -> FilePath -> IO ()
interactiveMain opSemFile cspmFile =
  do
    res <- runTyger (tygerMain opSemFile cspmFile)
    case res of
      Left err  -> putStrLn (show err)
      Right _   -> putStrLn ("Done")

tygerMain :: FilePath -> FilePath -> Tyger ()
tygerMain opsemFile cspmFile =
    do
      inputOpDefn <- parseOpSemFile opsemFile
      opSemDefn <- typeCheckOperators inputOpDefn

      let compiledOps = compileOperators opSemDefn
      cspmModules <- parseCSPMFile cspmFile opSemDefn
```

```
  if length cspmModules > 1 then
      panic "Modules are not currently supported"
    else return ()

  transformedModules <- runTypeChecker (do
    typeCheckedModules <- typeCheckModules opSemDefn cspmModules
    runTransformMonad $ transformModules opSemDefn typeCheckedModules)

  let operatorsFile =
    "module Operator_M"
    ++ indentEveryLine operatorModuleNotExported
    ++ (indentEveryLine . show) (rulesFunctionToCSP compiledOps)
    ++ ((indentEveryLine . show)
        (discardableArgsFunctionToCSP compiledOps))
    ++ "exports\n"
    ++ (indentEveryLine . show) (channelsToCSP opSemDefn)
    ++ indentEveryLine operatorModuleExported
    ++ "endmodule\n"
    ++ makeHeading "User Callable Functions"
    ++ globalModule
    ++ makeHeading "Operators"
    ++ show (operatorDatatypeToCSP compiledOps)++"\n"
    ++ show (operatorShortcutsToCSP compiledOps)++"\n"
    ++ show (replicatedOperatorsToCSP opSemDefn)

  let outputOpSemFile = replaceExtension opsemFile ".csp"
  let outputCSPMFile =
    replaceFileName cspmFile
    (dropExtension (takeFileName cspmFile)++"_Compiled.csp")

  let [Annotated _ _ (GlobalModule decls)] = cspmModules
  let channels = concat [n | Channel n _ <- map removeAnnotation decls]
  absoluteCSPMFilePath <- liftIO $ canonicalizePath outputCSPMFile
  absoluteOpSemFilePath <- liftIO $ canonicalizePath outputOpSemFile
  let cspmFile =
    "include \""++makeRelative (takeDirectory absoluteCSPMFilePath)
                  absoluteOpSemFilePath
          ++"\"\n\n"
    ++"UserEvents = {|"
      ++ (show . sep . punctuate comma . map prettyPrint) channels
      ++"|}\n\n"
    ++show (prettyPrint (head transformedModules))

  liftIO $ writeFile outputOpSemFile operatorsFile
  liftIO $ writeFile outputCSPMFile cspmFile

  return ()
```

C.1.3. *Util.hs.*

```
{-# LANGUAGE QuasiQuotes, GeneralizedNewtypeDeriving,
      FlexibleInstances, MultiParamTypeClasses #-}
module Util where

import Control.Monad
import Control.Monad.Error
import Control.Monad.Trans
import Control.Monad.State

import Language.Haskell.TH.Quote
import Language.Haskell.TH.Syntax hiding (lift)
import Language.Haskell.TH.Lib
import List (nub)

-- See ConstantCode.hs for examples of how to use
multilineLiteral :: QuasiQuoter
multilineLiteral = QuasiQuoter (litE . stringL) (litP . stringL)

indentEveryLine :: String -> String
indentEveryLine = unlines . map (\l -> '\t':l) . lines

makeHeading :: String -> String
makeHeading s =
  "--********************************************************************\n"++
  "--"++s++"\n"++
  "--********************************************************************\n"

-- ****************************************************************
-- Partial functions
-- ****************************************************************
type PartialFunction a b = [(a,b)] -- From a to b

functionDomain :: Eq a => PartialFunction a b -> [a]
functionDomain f = map fst f

functionImage :: Eq a => PartialFunction a b -> [b]
functionImage f = map snd f

identityFunction :: Eq a => [a] -> PartialFunction a a
identityFunction xs = [(x,x) | x <- xs]

invert :: (Eq a, Eq b) => PartialFunction a b -> PartialFunction b a
invert f = [(a,b) | (b,a) <- f]

apply :: Eq a => PartialFunction a b -> a -> b
apply f x =
  let
    pos = [b | (a,b) <- f, a == x]
  in
    if length pos == 0 then
      error ("Partial_function_applied_to_value_outside_of_domain")
    else head pos

applyRelation :: Eq a => PartialFunction a b -> a -> [b]
applyRelation f x = [b | (a,b) <- f, a == x]

safeApply :: Eq a => PartialFunction a b -> a -> Maybe b
safeApply f x =
  let
    pos = [b | (a,b) <- f, a == x]
  in
    if length pos == 0 then Nothing
    else Just (head pos)

composeFunctions ::
  (Eq a, Eq b) => PartialFunction b c -> PartialFunction a b ->
        PartialFunction a c
composeFunctions f g =
  [(a, apply f b) | (a,b) <- g]

mapPF :: Eq a => PartialFunction a b -> [a] -> [b]
mapPF f xs = [apply f x | x <- xs]

safeMapPF :: Eq a => PartialFunction a b -> [a] -> [b]
safeMapPF f xs = [x | Just x <- [safeApply f x | x <- xs]]

updatePF :: Eq a => PartialFunction a b -> a -> b -> PartialFunction a b
updatePF f a b = (a,b):[(c,d) | (c,d) <- f, c /= a]

removeEntry :: Eq a => PartialFunction a b -> a -> PartialFunction a b
removeEntry f a = [(c,d) | (c,d) <- f, c /= a]



noDups :: Eq a => [a] -> Bool
noDups xs = nub xs == xs

-- ****************************************************************
-- Monads
-- ****************************************************************
concatMapM :: (Monad m) => (a -> m [b]) -> [a] -> m [b]
concatMapM f xs = liftM concat (mapM f xs)

data TygerError =
  CSPMTypeCheckError String
  | CSPMParseError String
  | OpSemParseError String
  | OpSemTypeCheckError String
  | TygerUnknownError String
  | Panic String

instance Show TygerError where
  show (CSPMTypeCheckError s) = s
  show (CSPMParseError s) = s
  show (OpSemParseError s) = s
  show (OpSemTypeCheckError s) = s
  show (TygerUnknownError s) = "An unknown error occured. More information: "++s
  show (Panic s) = "Internal inconsistency error! More information: "++s

instance Error TygerError where
  strMsg = TygerUnknownError

-- Main monad used
newtype Tyger a = Tyg {
  runTyg :: ErrorT TygerError IO a
} deriving (Monad, MonadIO, MonadError TygerError)

class Monad m => MonadTyger m where
  panic :: String -> m a
  debugOutput :: String -> m ()
instance MonadTyger Tyger where
```

```
  panic = throwError . Panic
  debugOutput = liftIO . putStrLn
instance MonadTyger m => MonadTyger (StateT s m) where
  panic = lift . panic
  debugOutput = lift . debugOutput
instance (MonadTyger m, Error e) => MonadTyger (ErrorT e m) where
  panic = lift . panic
  debugOutput = lift . debugOutput

runTyger :: Tyger a -> IO (Either TygerError a)
runTyger = runErrorT . runTyg
```

C.2. **Operational Semantics.**

C.2.1. *OpSemDataStructures.hs.*

```
module OpSemDataStructures where

import Data.IORef
import Util
import Text.PrettyPrint.HughesPJ

-- ************************************************************************
-- Rule input data types
-- ************************************************************************

newtype Name = Name String deriving (Eq)

instance Show Name where
  show (Name n) = show n

data Event =
  Event Name
  | ChanEvent Name [Exp]     -- The channel must carry events only
  | Tau                -- and thus its type must be UserEvents.UserEvents...
  deriving (Eq, Show)

data Exp =
  OperatorApp Name [Exp]
  | InductiveCase
  | Tuple [Exp]
  | Var Name

  | SigmaPrime  -- SigmaPrime = SystemEvents
  | Sigma       -- i.e. UserEvents
  | ProcArgs
  | SetComprehension [Exp] [SideCondition]
  | Set [Exp]
  | SetMinus Exp Exp
  | Union Exp Exp
  | Intersection Exp Exp
  | Powerset Exp

  | ReplicatedUnion Exp
  deriving (Eq, Show)

data ProcessRelation =
  Performs Exp Event Exp
  deriving Show

data Pattern =
  PVar Name
  | PTuple [Pattern]
  | PSet [Pattern]
  deriving (Eq, Show)

-- A side condition should always be equal to SCGenerator p e1 e2
data SideCondition =
  SCGenerator Pattern Exp
  | Formula PropositionalFormula
  deriving (Eq, Show)

data PropositionalFormula =
  Member Pattern Exp
  | Equals Exp Exp
  | Subset Exp Exp
  | Not PropositionalFormula
  | And PropositionalFormula PropositionalFormula
  | Or PropositionalFormula PropositionalFormula
  | PFalse
  | PTrue
  deriving (Eq, Show)

data InductiveRule =
  InductiveRule [ProcessRelation] ProcessRelation [SideCondition]
  deriving Show

-- Always omit tau promotion rules
data InputOperator =
  InputOperator {
    iopFriendlyName :: Name,
    iopArgs :: [(Name, ProcessSubtype)],
    iopRules :: [InductiveRule],
    iopReplicatedOperator :: Maybe InputReplicatedOperator,
    iopParsingInformation :: Maybe OperatorSyntax
  }
  deriving Show

data InputReplicatedOperator =
  InputReplicatedOperator {
    irepOpArgs :: [Name],
    irepOpBaseCase :: ([Pattern], Exp),
    -- List of vars for this case, list of vars for recursive case
    -- lengths should all be equal to the args length
    irepOpInductiveCase :: ([Name], Exp),
    irepOpParsingInformation :: Maybe OperatorSyntax
  }
  deriving Show

data InputOpSemDefinition =
  InputOpSemDefinition {
    inputOperators :: [InputOperator],
    inputChannels :: [Channel]
  }
  deriving Show

data Assoc =
  AssocLeft | AssocRight | AssocNone
  deriving Show

data OperatorSyntax =
  InfixOp [ParseComponent] Integer Assoc  -- Int refers to precedence
  | PrefixOp [ParseComponent] Integer
  | PostfixOp [ParseComponent] Integer
  deriving Show

data ParseComponent =
  String String
  | Argument Integer
  deriving Show

-- ************************************************************************
-- Intermediate Rule Data Types (post type inference)
-- ************************************************************************
newtype TypeVar = TypeVar Int deriving (Eq, Show)

-- We use typerefs so that when we return a type we don't have to worry about
-- passing substitutions around
data TypeVarRef =
  TypeVarRef TypeVar (IORef (Maybe Type))
  deriving Eq

instance Show TypeVarRef where
  show (TypeVarRef tv _) = "TypeVarRef_"++show tv

data ProcessSubtype =
  InfinitelyRecursive
  | FinitelyRecursive
  | NotRecursive

  | Unknown
  deriving (Eq, Show)

data Type =
  TVar TypeVarRef
  | TEvent
  | TProcArg
  | TOnProcess ProcessSubtype
  | TOffProcess ProcessSubtype
  | TSet Type      -- Only sets of events are supported currently
  | TChannel [Type]
  | TTuple [Type]
  | TOperator [Type]
  deriving (Eq)

instance Show Type where
  show (TVar (TypeVarRef tv _)) = show tv
  show TEvent = "Event"
  show TProcArg = "ProcArg"
  show (TOnProcess st) = "OnProc_"++show st
  show (TOffProcess st) = "OffProcess_"++show st
```

```
  show (TSet t) = "{"++show t++"}"
  show (TChannel ts) = "Channel_"++show ts
  show (TTuple ts) =
    show (parens (hsep (punctuate comma (map (text . show) ts))))
  show (TOperator ts) = "Operator_"++show ts

typeIsProc :: Type -> Bool
typeIsProc (TOnProcess _) = True
typeIsProc (TOffProcess _) = True
typeIsProc _ = False

-- Always omit tau promotion rules
data Operator =
  Operator {
    opFriendlyName :: Name,
    opArgs :: [(Name, Type)],
    opRules :: [InductiveRule],
    opParsingInformation :: Maybe OperatorSyntax
  }
  | ReplicatedOperator {
    opFriendlyName :: Name,
    opArgs :: [(Name, Type)],
    repOpBaseCase :: ([Pattern], Exp),
    -- List of vars for this case, list of vars for recursive case
    -- lengths should all be equal to the args length
    repOpInductiveCase :: ([Name], Exp),
    opParsingInformation :: Maybe OperatorSyntax
  }
  deriving Show

data Channel = Channel Name [Exp]
  deriving Show

data OpSemDefinition = OpSemDefinition {
    operators :: [Operator],
    channels :: [Channel]
  }
  deriving Show

-- ************************************************************************
-- Rule Output Data Types (post compilation)
-- ************************************************************************
type ProcId = Int

data CompiledOp =
  CompiledOp {
    copFriendlyName :: String,
    copArgs :: [(Name, Type)],
    copRules :: [CompiledRule],
    copDiscards :: [ProcId]
  }
  deriving Show

data CompiledRule =
  CompiledRule (PartialFunction ProcId Event) Event (String, [Exp])
    (PartialFunction ProcId ProcId) (PartialFunction ProcId ProcId)
    (PartialFunction ProcId ProcId) [ProcId] [Name] [Stmt]
  deriving Show

data Stmt =
  Generator Pattern Exp
  | PropFormula PropositionalFormula
  deriving (Show)
```

C.2.2. *OpSemParser.hs.*

```
module OpSemParser(parseOpSemFile) where

import OpSemDataStructures
import OperatorParsers
import Util

import Control.Monad(liftM, sequence)
import Control.Monad.Error
import List
import qualified Text.Parsec.Expr as E
import Text.Parsec.Language
import Text.Parsec
import qualified Text.Parsec.Token as PT

type Parser = Parsec String (PartialFunction Name (Maybe OperatorSyntax))

dotSep p = sepBy p dot
braces = between (symbol "{") (symbol "}")

opSemLanguage = PT.LanguageDef {
    PT.commentStart = "/*",
    PT.commentEnd = "*/",
    PT.commentLine = "//",
    PT.nestedComments = False,
    PT.identStart = letter,
    PT.identLetter = alphaNum <|> char '\'' <|> char '_',
    PT.opStart = oneOf "[]|/\\-<>{}~=@",
    PT.opLetter = oneOf "[]|/\\-<>{}~=@",
    PT.reservedOpNames = ["⇒", "==", "!=", "<="],
    PT.reservedNames = [
              -- Section delimeters
              "Rule", "EndRule", "Operator", "EndOperator",
              -- Syntax rules
              "Syntax", "Binary",
              -- Expressions
              "Union",
              "Sigma", "union", "diff", "inter", "Set",
              -- Side conditions
              "member", "true", "false",
              -- Events
              "tau",
              "Identity",
              "Replicated", "EndReplicated",
              "BaseCase", "EndBaseCase",
              "InductiveCase", "EndInductiveCase",
              "Channels", "EndChannels"],
    PT.caseSensitive = True
  }
PT.TokenParser{ PT.parens = parens,
          PT.identifier = identifier,
      PT.natural = natural,
      PT.charLiteral = charLiteral,
      PT.stringLiteral = stringLiteral,
      PT.lexeme = lexeme,
          PT.reservedOp = reservedOp,
      PT.operator = operator,
          PT.reserved = reserved,
      PT.whiteSpace = whiteSpace,
      PT.symbol = symbol,
      PT.comma = comma,
      PT.dot = dot,
      PT.commaSep = commaSep,
      PT.commaSep1 = commaSep1} = PT.makeTokenParser opSemLanguage

parseOpSemFile :: String -> Tyger InputOpSemDefinition
parseOpSemFile fname =
  let
    pass1 =
      do
        whiteSpace
        option [] channelSectionParser
        whiteSpace
        operatorParseInfo <- many operatorPhase1Parser
        eof
        return operatorParseInfo
    pass2 =
      do
        whiteSpace
        chans <- option [] channelSectionParser
        whiteSpace
        ops <- many operatorParser
        eof
        return $ InputOpSemDefinition ops chans
  in
    do
      input <- liftIO $ readFile fname
      case runP pass1 [] fname input of
        Left err      -> throwError $ OpSemParseError (show err)
        Right opParseInfo ->
          case runP pass2 opParseInfo fname input of
            Left err -> throwError $ OpSemParseError (show err)
            Right ops -> return ops

channelSectionParser :: Parser [Channel]
channelSectionParser =
  do
    reserved "Channels"
    chans <- many (do
            name <- identifier
            typ <- option [] (do
                  lexeme (string ":")
                  components <- dotSep expressionParser
                  return components)
            return $ Channel (Name name) typ
      )
    reserved "EndChannels"
    return chans
```

```
operatorPhase1Parser :: Parser (Name, Maybe OperatorSyntax)
operatorPhase1Parser =
  do
    reserved "Operator"
    friendlyName <- identifier
    args <- option [] (parens (commaSep nameSubtypeParser))
    syntax <- option Nothing (liftM Just syntaxParser)
    manyTill anyChar (reserved "EndOperator")
    return (Name friendlyName, syntax)

operatorParser :: Parser InputOperator
operatorParser =
  do
    reserved "Operator"
    friendlyName <- identifier
    args <- option [] (parens (commaSep nameSubtypeParser))
    syntax <- option Nothing (liftM Just syntaxParser)
    rules <- many ruleParser
    replicatedOp <- option Nothing (liftM Just replicatedOpParser)
    reserved "EndOperator"
    return $ InputOperator (Name friendlyName)
                  args
                  rules
                  replicatedOp
                  syntax

nameSubtypeParser :: Parser (Name, ProcessSubtype)
nameSubtypeParser =
  do
    id <- identifier
    st <- ((lexeme (string ":") >>
        choice [
          lexeme (string "FinRec") >> return FinitelyRecursive,
          lexeme (string "InfRec") >> return InfinitelyRecursive])
        <|> return Unknown)
    return (Name id, st)

replicatedOpParser :: Parser InputReplicatedOperator
replicatedOpParser =
  do
    reserved "Replicated"
    args <- option [] (parens (commaSep identifier))
    syntax <- option Nothing (liftM Just syntaxParser)

    reserved "BaseCase"
    basePats <- parens (commaSep patternParser)
    baseCase <- expressionParser
    reserved "EndBaseCase"
    reserved "InductiveCase"
    recursiveArgs <- parens (commaSep identifier)
    inductiveCase <- expressionParser
    reserved "EndInductiveCase"

    reserved "EndReplicated"

    return $ InputReplicatedOperator (map Name args) (basePats, baseCase)
          (map Name recursiveArgs, inductiveCase) syntax

ruleParser :: Parser InductiveRule
ruleParser =
  do
    reserved "Rule"
    pres <- many (processRelationParser False)
    lexeme (skipMany1 (char '-'))
    scs <- commaSep (try sideConditionParser)
    whiteSpace
    post <- processRelationParser True
    reserved "EndRule"
    return (InductiveRule pres post scs)

syntaxParser :: Parser OperatorSyntax
syntaxParser =
  let
    patternParser = between (lexeme (char '"'))_(lexeme_(char_'"'))
              (many parseComponentParser)
  in
    reserved "Syntax" >>
    choice [
      do
        reserved "Binary"
        pattern <- patternParser
        precedence <- natural
        string "Assoc"
        associativity <-
          ((lexeme (string "Left") >> return AssocLeft)
            <|> (lexeme (string "Right") >> return AssocRight)
            <|> (lexeme (string "None") >> return AssocNone)
            <?> "associativity_specification")
        return $ InfixOp pattern precedence associativity,
      do
        reserved "Prefix"
        pattern <- patternParser
        precedence <- natural
        return $ PrefixOp pattern precedence,
      do
        reserved "Postfix"
        pattern <- patternParser
        precedence <- natural
        return $ PostfixOp pattern precedence]

parseComponentParser :: Parser ParseComponent
parseComponentParser =
  (try (lexeme (string "\\$" >> return (String "$"))))
  <|> (do
    char '$'
    id <- natural
    return (Argument id))
  <|> (operator >>= (\ n -> return (String n)))
  <|> (identifier >>= (\ n -> return (String n)))

processRelationParser :: Bool -> Parser ProcessRelation
processRelationParser isPost =
  do
    e1 <- expressionParser
    lexeme (string "=")
    ev <- eventParser
    reservedOp "=>"
    e2 <- expressionParser
    case (isPost,e2) of
      (True, Var n) -> return (Performs e1 ev
                  (OperatorApp (Name "Identity") [e2]))
      _       -> return (Performs e1 ev e2)

expressionParser :: Parser Exp
expressionParser =
  do
    st <- getState
    (let
      parseFunctionCall1 s e =
        do
          reserved s
          e1 <- parens (expressionParser)
          return (e e1)
      parseFunctionCall2 s e =
        do
          reserved s
          [e1,e2] <- parens (commaSep expressionParser)
          return (e e1 e2)

      normalOperatorNames = [s | (s, _) <- st]
      exprParser = constructParseTable [] st []
              (return (\ n es -> OperatorApp n es))
              (error "generator_sem_called")
              term
              (error "generator_called")

      term = choice [
          try (liftM Tuple (parens (commaSep expressionParser))),
          parens expressionParser,
          braces (
            do
              exps <- commaSep expressionParser
              option (Set exps) (do
                symbol "|"
                scs <- commaSep1 sideConditionParser
                return $ SetComprehension exps scs)
          ),
          try (
            do
              opName <- liftM Name identifier
              args <- option [] (parens (commaSep expressionParser))
              if (elem opName normalOperatorNames) then
                  return (OperatorApp opName args)
                else (unexpected ("operator_"++show opName++
                        "_not_recognized"))
          ),
        -- identifier uses try, hence we can use it here and it will
        -- not consume any input
        liftM (Var . Name) identifier,
        (reserved "Sigma" >> return Sigma),
        (reserved "InductiveCase" >> return InductiveCase),
        parseFunctionCall1 "Set" Powerset,
        parseFunctionCall1 "Union" ReplicatedUnion,
        parseFunctionCall2 "union" Union,
        parseFunctionCall2 "inter" Intersection,
```

```
          parseFunctionCall2 "diff" SetMinus]
          <?> "expression"
      in
        try exprParser
        <|> term)


sideConditionParser :: Parser SideCondition
sideConditionParser =
  let
    table = [
      [E.Prefix (prefix "!" Not)],
      [E.Infix (binary "&&" And) E.AssocLeft,
       E.Infix (binary "||" Or) E.AssocLeft]
      ]
    prefix s e = lexeme (string s) >> return e
    binary s e = lexeme (string s) >> return e

    booleanParser = E.buildExpressionParser table term
    term =
      parens booleanParser
      <|> try (reserved "true" >> return PTrue)
      <|> try (reserved "false" >> return PFalse)
      <|> try (do
          reserved "member"
          parens (
            do
              p <- patternParser
              comma
              e <- expressionParser
              return (Member p e)))
      <|> try (do
          e1 <- expressionParser
          lexeme (reservedOp "==")
          e2 <- expressionParser
          return (Equals e1 e2))
      <|> try (do
          e1 <- expressionParser
          lexeme (reservedOp "!=")
          e2 <- expressionParser
          return (Not (Equals e1 e2)))
      <|> try (do
          e1 <- expressionParser
          lexeme (reservedOp "<=")
          e2 <- expressionParser
          return (Subset e1 e2))
      <?> "boolean_term"
  in
    try (do
      pat <- patternParser
      lexeme (string "<-")
      exp <- expressionParser
      return (SCGenerator pat exp))
    <|>
      liftM Formula booleanParser

patternParser :: Parser Pattern
patternParser =
  liftM PTuple (parens (commaSep patternParser))
  <|> liftM PSet (braces (commaSep patternParser))
  <|> liftM (PVar . Name) identifier

eventParser :: Parser Event
eventParser =
  try (reserved "tau" >> return Tau)
  <|> do
      chanName <- identifier
      option (Event (Name chanName)) (do
        dot
        dataComponents <- dotSep expressionParser
        return $ ChanEvent (Name chanName) dataComponents)
```

C.2.3. *OpSemTypeChecker.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
module OpSemTypeChecker (typeCheckOperators, varsBound) where

import Data.IORef
import Control.Monad.Error
import Control.Monad.State
import List
import Util
import Text.PrettyPrint.HughesPJ
import OpSemPrettyPrinter
import OpSemDataStructures

identityRule =
  -- We specify identity's argument as not recursive since it is special
  InputOperator (Name "Identity") [(Name "P", NotRecursive)] [
    InductiveRule
      [Performs (Var (Name "P")) (Event (Name "a")) (Var (Name "P'"))]
      (Performs
        (OperatorApp (Name "Identity") [Var (Name "P")])
        (Event (Name "a"))
        (OperatorApp (Name "Identity") [Var (Name "P'")]))
      [SCGenerator (PVar (Name "a")) SigmaPrime],
    InductiveRule
      [Performs (Var (Name "P")) (ChanEvent (Name "callProc") [Var (Name "p")])
          (Var (Name "P'"))]
      (Performs
        (OperatorApp (Name "Identity") [Var (Name "P")])
        (ChanEvent (Name "callProc") [Var (Name "p")])
        (OperatorApp (Name "Identity") [Var (Name "P'")]))
      [SCGenerator (PVar (Name "p")) ProcArgs]
  ] Nothing Nothing

data TypeCheckError =
  ErrorWithOperator Name TypeCheckError
  | ErrorWithRule InductiveRule TypeCheckError
  | ErrorWithExpression Exp TypeCheckError

  -- ** Specific Errors (i.e. errors that indicate their cause)
  -- Name appears multiple times in some context
  | DuplicatedNameError [Name]
  -- e.g. P -> P
  | ArgumentsUsedOnRHSOfProcessRelation [Name]
  | OnProcessClonedError [Name]
  -- E.g. P -tau-> P' => op(P) -a-> op(P')
  | TauPromotedError
  | ResultingProcessDiscarded [Name]

  -- ** General Errors
  -- Error whilst unifying the two types
  | UnificationError Type Type
  -- Name is an unknown variable
  | VariableNotInScope Name
  | InfiniteUnificationError TypeVar Type

  | UnknownError String

prettyPrintType = show

showList' :: Show a => [a] -> String
showList' xs = show $ hsep (punctuate comma (map text (map show xs)))

instance Show TypeCheckError where
  show (ErrorWithOperator n err) =
    show err++
    "in_the_operator_"++show n++"."
  show (ErrorWithRule rule err) =
    show err++
    "in_the_rule:\n"++
    indentEveryLine (show (prettyPrint rule))
  show (ErrorWithExpression exp err) =
    show err++
    "in_the_expression:\n"++
    indentEveryLine (show (prettyPrint exp))
  show (DuplicatedNameError ns) =
    "The_names_"++showList' ns++"_are_duplicated.\n"
  show (ArgumentsUsedOnRHSOfProcessRelation ns) =
    "The_arguments_"++showList' ns++"_are_used_on_the_right_hand_side_of"++
    "a_process_relation.\n"
  show (OnProcessClonedError ns) =
    "The_on_processes_"++showList' ns++"_are_cloned.\n"
  show (TauPromotedError) =
    "A_tau_was_promoted_to_a_non-tau_event.\n"
  show (ResultingProcessDiscarded ns) =
    "The_processes_resulting_from_"++showList' ns++"_are_discarded.\n"
  show (UnificationError t1 t2) =
    "Could_not_match_the_types:\n"++
    "\t"++show (prettyPrintType t1)++
    "\nand\n"++
    "\t"++show (prettyPrintType t2)++"\n"
  show (VariableNotInScope n) = "Name_"++show n++"_is_not_in_scope.\n"
  show (InfiniteUnificationError tv t) =
    "Cannot_construct_the_infinite_type:_"++
    show tv++"_=_"++show (prettyPrintType t)++"\n"
  show (UnknownError s) =
    "An_unknown_error_occured:_"++s++"\n"

instance Error TypeCheckError where
  strMsg = UnknownError
```

```
addOperatorToError :: Name -> TypeCheckMonad a -> TypeCheckMonad a
addOperatorToError n m =
  m `catchError` (\ e -> throwError $ ErrorWithOperator n e)

addRuleToError :: InductiveRule -> TypeCheckMonad a -> TypeCheckMonad a
addRuleToError n m =
  m `catchError` (\ e -> throwError $ ErrorWithRule n e)

addExpToError :: Exp -> TypeCheckMonad a -> TypeCheckMonad a
addExpToError exp m =
  m `catchError` (\ e -> throwError $ ErrorWithExpression exp e)

-- ***************************************************************************
-- Type Check Monad
-- ***************************************************************************
data TypeInferenceState = TypeInferenceState {
    -- map from names to types
    environment :: [PartialFunction Name Type],
    -- Next TypeVar to be allocated
    nextTypeId :: Int,
    -- If true then
    -- unify (TOnProcess) (TOffProcess) = TOnProcess
    -- NB: this is not symmetric and is used only in operator application
    allowGeneralUnification :: Bool
  }
  deriving Show

type TypeCheckMonad =
  ErrorT TypeCheckError (StateT TypeInferenceState Tyger)

errorIfFalse :: Bool -> TypeCheckError -> TypeCheckMonad ()
errorIfFalse True e = return ()
errorIfFalse False e = throwError e

errorIfFalseM :: TypeCheckMonad Bool -> TypeCheckError -> TypeCheckMonad ()
errorIfFalseM m e =
  do
    res <- m
    errorIfFalse res e

unifyTypeWithName :: Name -> Type -> TypeCheckMonad ()
unifyTypeWithName n t =
  do
    typ <- getType n
    unify t typ
    return ()


readTypeRef :: TypeVarRef -> TypeCheckMonad (Either TypeVar Type)
readTypeRef (TypeVarRef tv ioref) =
  do
    mtyp <- liftIO $ readIORef ioref
    case mtyp of
      Just t  -> return (Right t)
      Nothing -> return (Left tv)

writeTypeRef :: TypeVarRef -> Type -> TypeCheckMonad ()
writeTypeRef (TypeVarRef tv ioref) t = liftIO $ writeIORef ioref (Just t)

freshTypeVar :: TypeCheckMonad Type
freshTypeVar =
  do
    nextId <- gets nextTypeId
    modify (\s -> s { nextTypeId = nextId+1 })
    ioRef <- liftIO $ newIORef Nothing
    return $ TVar (TypeVarRef (TypeVar nextId) ioRef)


safeGetType_ :: [PartialFunction Name Type] -> Name -> Maybe Type
safeGetType_ [] n = Nothing
safeGetType_ (pf:pfs) n =
  case safeApply pf n of
    Just t  -> Just t
    Nothing -> safeGetType_ pfs n

getType :: Name -> TypeCheckMonad Type
getType name =
  do
    envs <- gets environment
    case safeGetType_ envs name of
      Just t  -> return t
      Nothing -> throwError $ VariableNotInScope name

safeGetType :: Name -> TypeCheckMonad (Maybe Type)
safeGetType n =
  do
    envs <- gets environment
    return $ safeGetType_ envs n
```

```
-- Sets the type of n to be t in the current scope only. No unification is
-- performed.
setType :: Name -> Type -> TypeCheckMonad ()
setType n t =
  do
    res <- safeGetType n
    (env:envs) <- gets environment
    let env' = updatePF env n t
    modify (\ s -> s { environment = env':envs })

local :: [Name] -> TypeCheckMonad a -> TypeCheckMonad a
local ns m =
  do
    env <- gets environment
    nextId <- gets nextTypeId
    newArgs <- replicateM (length ns) freshTypeVar
    modify (\s -> s { environment = (zip ns newArgs):env })

    res <- m

    env <- gets environment
    modify (\ s -> s { environment = tail env })

    return res

generalUnificationAllowed :: TypeCheckMonad a -> TypeCheckMonad a
generalUnificationAllowed m =
  do
    r <- gets allowGeneralUnification
    modify (\ s -> s { allowGeneralUnification = True})
    res <- m
    modify (\ s -> s { allowGeneralUnification = r})
    return res

compress :: Type -> TypeCheckMonad Type
compress (tr @ (TVar typeRef)) =
  do
    res <- readTypeRef typeRef
    case res of
      Left tv -> return tr
      Right t -> compress t
compress (TSet t) =
  do
    t' <- compress t
    return $ TSet t'
compress (TTuple ts) =
  do
    ts' <- mapM compress ts
    return $ TTuple ts'
compress (TChannel ts) =
  do
    ts' <- mapM compress ts
    return $ TChannel ts'
compress (TOperator ts) =
  do
    ts' <- mapM compress ts
    return $ TOperator ts'
compress t = return t

-- ***************************************************************************
-- Dependency Analyisis
-- ***************************************************************************
class VarsBound a where
  -- There should never be any duplicates
  varsBound :: a -> [Name]
instance VarsBound a => VarsBound [a] where
  varsBound = concatMap varsBound
instance VarsBound Pattern where
  varsBound (PVar n) = [n]
  varsBound (PTuple ps) = varsBound ps
  varsBound (PSet ps) = varsBound ps
instance VarsBound SideCondition where
  varsBound (SCGenerator p e) = varsBound p
  varsBound (Formula p) = []

class FreeVars a where
  freeVars :: a -> [Name]
  freeVars = nub . freeVars'
  freeVars' :: a -> [Name]
instance FreeVars a => FreeVars [a] where
  freeVars' = concatMap freeVars
instance FreeVars Exp where
  freeVars' (Tuple es) = freeVars' es
  freeVars' (Var n) = [n]
  freeVars' (Sigma) = []
  freeVars' (SigmaPrime) = []
  freeVars' (Powerset e) = freeVars' e
```

```
  freeVars' (ProcArgs) = []
  freeVars' (SetMinus e1 e2) = freeVars' e1 ++ freeVars' e2
  freeVars' (Set es) = freeVars' es
  freeVars' (Union e1 e2) = freeVars' e1 ++ freeVars' e2
  freeVars' (Intersection e1 e2) = freeVars' e1 ++ freeVars' e2
  freeVars' (OperatorApp n es) = freeVars' es
  freeVars' (InductiveCase) = []
  freeVars' (ReplicatedUnion e) = freeVars' e
  freeVars' (SetComprehension es scs) =
    (freeVars' es++freeVars' scs) \\ varsBound scs
instance FreeVars SideCondition where
  freeVars' (SCGenerator p e) = freeVars' e
  freeVars' (Formula p) = freeVars p
instance FreeVars PropositionalFormula where
  freeVars' (Subset e1 e2) = freeVars' [e1,e2]
  freeVars' (Member p e) = freeVars' e++freeVars' p
  freeVars' (Equals e1 e2) = freeVars' [e1,e2]
  freeVars' (And e1 e2) = freeVars' [e1,e2]
  freeVars' (Or e1 e2) = freeVars' [e1,e2]
  freeVars' (Not e1) = freeVars' e1
  freeVars' PFalse = []
  freeVars' PTrue = []
instance FreeVars Pattern where
  freeVars' (PVar n) = [n]
  freeVars' (PTuple ns) = freeVars' ns
  freeVars' (PSet ps) = freeVars' ps
instance FreeVars Event where
  freeVars' (Tau) = []
  freeVars' (Event n) = [n]
  freeVars' (ChanEvent n es) = n:(concatMap freeVars' es)

class TypeCheckable a b | a -> b where
  errorConstructor :: a -> (TypeCheckError -> TypeCheckError)
  typeCheck :: a -> TypeCheckMonad b
  typeCheck e =
    typeCheck' e `catchError` (throwError . errorConstructor e)
  typeCheck' :: a -> TypeCheckMonad b

instance TypeCheckable SideCondition () where
  errorConstructor x = id
  typeCheck' (SCGenerator p generator) =
    do
      tgen <- typeCheck generator
      tpat <- typeCheck p
      unify tgen (TSet tpat)
      return ()
  typeCheck' (Formula f) = typeCheck f
instance TypeCheckable PropositionalFormula () where
  errorConstructor x = id
  typeCheck' (Subset e1 e2) =
    do
      t1 <- typeCheck e1
      t2 <- typeCheck e2
      ensureIsSet t1
      ensureIsSet t2
      unify t1 t2
      return ()
  typeCheck' (Member p e) =
    do
      t1 <- typeCheck p
      t2 <- typeCheck e
      unify (TSet t1) t2
      return ()
  typeCheck' (Equals e1 e2) =
    do
      t1 <- typeCheck e1
      t2 <- typeCheck e2
      unify t1 t2
      return ()
  typeCheck' (Not sc) = typeCheck sc
  typeCheck' (And sc1 sc2) = typeCheck sc1 >> typeCheck sc2
  typeCheck' (Or sc1 sc2) = typeCheck sc1 >> typeCheck sc2
  typeCheck' PTrue = return ()
  typeCheck' PFalse = return ()
instance TypeCheckable Exp Type where
  errorConstructor = ErrorWithExpression
  typeCheck' Sigma = return $ TSet TEvent
  typeCheck' SigmaPrime = return $ TSet TEvent
  typeCheck' ProcArgs = return $ TSet TProcArg
  typeCheck' (Powerset e) =
    do
      t <- typeCheck e
      ensureIsSet t
      return $ TSet t
  typeCheck' (Tuple es) =
    do
      ts <- mapM typeCheck es
      return $ TTuple ts
  typeCheck' (Var n) = getType n
  typeCheck' (Set es) =
    do
      ts <- mapM typeCheck es
      t <- unifyAll ts
      return $ TSet t
  typeCheck' (SetMinus s1 s2) =
    do
      t1 <- typeCheck s1
      t2 <- typeCheck s2
      ensureIsSet t1
      ensureIsSet t2
      unify t1 t2
  typeCheck' (Union s1 s2) =
    do
      t1 <- typeCheck s1
      t2 <- typeCheck s2
      ensureIsSet t1
      ensureIsSet t2
      unify t1 t2
  typeCheck' (Intersection s1 s2) =
    do
      t1 <- typeCheck s1
      t2 <- typeCheck s2
      ensureIsSet t1
      ensureIsSet t2
      unify t1 t2
  typeCheck' (InductiveCase) =
    -- We return TOnProcess here - this is fine since we assume everything
    -- in the inductive case is On.
    return $ TOffProcess Unknown
  typeCheck' (SetComprehension es scs) =
    do
      let varsToBind = varsBound scs
      errorIfFalse (noDups varsToBind) (DuplicatedNameError varsToBind)
      local varsToBind (
        do
          mapM typeCheck scs
          ts <- mapM typeCheck es
          t <- unifyAll ts
          return $ TSet t)
  typeCheck' (ReplicatedUnion e) =
    do
      t <- typeCheck e
      fv <- freshTypeVar
      unify t (TSet (TSet fv))
      return $ TSet fv
  typeCheck' (OperatorApp n es) =
    do
      ts <- mapM typeCheck es
      opType <- getType n
      -- IMPORTANT: order is important here as the unification may not
      -- be symmetric (see allowGeneralUnification)
      t <- unify opType (TOperator ts)
      return $ TOnProcess Unknown

instance TypeCheckable Pattern Type where
  errorConstructor x = id
  typeCheck' (PVar n) = getType n
  typeCheck' (PTuple ps) =
    do
      ts <- mapM typeCheck ps
      return $ TTuple ts
  typeCheck' (PSet ps) =
    do
      ts <- mapM typeCheck ps
      t <- unifyAll ts
      return $ TSet t

typeCheckOperators :: InputOpSemDefinition -> Tyger OpSemDefinition
typeCheckOperators (InputOpSemDefinition ops chans) =
  let
    chanNames = [n | Channel n _ <- chans]
    typeCheckOps =
      do
        errorIfFalse (noDups chanNames) (DuplicatedNameError chanNames)
        setType (Name "callProc") (TChannel [TSet TProcArg])
        mapM injectChannelType chans
        ops <- concatMapM typeCheckOperator (identityRule:ops)
        return $ OpSemDefinition ops chans
    injectChannelType (Channel n es) =
      do
        ts <- mapM typeCheck es
        setType n (TChannel ts)
  in do
    (lh, st) <-
      runStateT (runErrorT typeCheckOps) (TypeInferenceState [[]] 0 False)
```

```
    case lh of
      Left err -> throwError $ OpSemTypeCheckError (show err)
      Right ops -> return ops

typeCheckOperator :: InputOperator -> TypeCheckMonad [Operator]
typeCheckOperator (InputOperator n argsSt rules replicatedOp syntax) =
  let
    args = map fst argsSt
    onArgs = nub [n | InductiveRule pres _ _ <- rules,
              Performs (Var n) _ _<- pres]
    isOnArg arg = arg `elem` onArgs
    mergeType (name, st) (TOnProcess _) = TOnProcess st
    mergeType (name, st) (TOffProcess _) = TOffProcess st
    mergeType (name, Unknown) t = t
  in
    addOperatorToError n (do
      errorIfFalse (noDups args) (DuplicatedNameError args)

      operatorTypeArgs <- replicateM (length args) freshTypeVar
      setType n (TOperator operatorTypeArgs)

      -- Type check all the rules: returns monads that will type check
      -- the operator calls each rule makes
      local args (mapM_ (typeCheckInductiveRule args) rules)
      ts' <- mapM compress operatorTypeArgs
      let ts = map (\ t -> case t of
                TVar _       -> TOffProcess Unknown
                _        -> t) ts'

      let argsTypes =
        zipWith (\ (arg, st) t ->
          case t of
            -- Check that it actually is an on process
            -- e.g it might infer that it is on because of a
            -- recursive call that makes it on
            TOnProcess st' ->
              if isOnArg arg then mergeType (arg, st) t
              else mergeType (arg, st) (TOffProcess st')
            t       -> mergeType (arg, st) t) argsSt ts
      setType n (TOperator argsTypes)
      let op = Operator n (zip args argsTypes) rules syntax
      case replicatedOp of
          Just repOp ->
            do
              repOp <- typeCheckReplicatedOperator n repOp
              return [op, repOp]
          Nothing   -> return [op])

typeCheckReplicatedOperator
  :: Name -> InputReplicatedOperator -> TypeCheckMonad Operator
typeCheckReplicatedOperator (Name n) (InputReplicatedOperator args
    (basePats, baseCase)
    (inductiveVars, inductiveCase) syntax) =
  do
    (tBaseCase, tPats) <- local (freeVars basePats) (do
      tPats <- mapM typeCheck basePats
      mapM ensureIsSet tPats
      t <- typeCheck baseCase
      ensureIsProc t
      return (t, tPats))

    errorIfFalse (noDups (args++inductiveVars))
          (DuplicatedNameError (args++inductiveVars))

    operatorArgTypes <- local (args++inductiveVars) (do
      t <- generalUnificationAllowed (typeCheck inductiveCase)
      ensureIsProc t
      tArgs <- mapM getType args
      generalUnificationAllowed (zipWithM unify tPats (map TSet tArgs))
      tArgs' <- mapM getType args
      return tArgs')

    ts <- mapM compress operatorArgTypes

    return $ ReplicatedOperator (Name ("Replicated"++n))
          (zip args ts) (basePats, baseCase)
          (inductiveVars, inductiveCase) syntax

-- Returns a type inference monad that, when called, will typecheck the
-- operator calls
typeCheckInductiveRule
  :: [Name] -> InductiveRule -> TypeCheckMonad ()
typeCheckInductiveRule args
    (rule @ (InductiveRule pres (post @ (Performs opApp1 ev opApp2)) sc)) =
  addRuleToError rule (
  let
    onPreProcs = [(n, TOnProcess Unknown) | Performs (Var n) _ _ <- pres]
    onPostProcs = [(n, TOnProcess Unknown) | Performs _ _ (Var n) <- pres]
    argPostProcs = intersect args (map (\ (n, t) -> n) onPostProcs)
    -- We use freeVars' here as it does not remove duplicates
    procsCloned = [head xs | xs <- group (freeVars' opApp2), length xs > 1]
    isOnProc (TOnProcess _) = True
    isOnProc _ = False
  in do
    errorIfFalse (length argPostProcs == 0)
      (ArgumentsUsedOnRHSOfProcessRelation argPostProcs)
    errorIfFalse (length [r | r @ (Performs _ Tau _) <- pres] == 0)
      TauPromotedError

    let varsToBind = varsBound sc
    errorIfFalse (noDups varsToBind) (DuplicatedNameError varsToBind)
    local varsToBind (do
      mapM_ (\ ev ->
        case ev of
          Event n      -> unifyTypeWithName n TEvent
          ChanEvent n es  ->  do
            t <- getType n
            tsArgs <- mapM typeCheck es
            unify t (TChannel $ map TSet tsArgs)
            return ())
        [ev | Performs _ ev _ <- pres]

      mapM_ typeCheck sc

      case ev of
        Event n          -> unifyTypeWithName n TEvent
        -- See comment in DataTypes: a channel may only have event
        -- components
        ChanEvent chanName es -> do
            t <- getType chanName
            tsArgs <- mapM typeCheck es
            unify t (TChannel $ map TSet tsArgs)
            return ()
        Tau             -> return ()

      mapM_ (uncurry unifyTypeWithName) onPreProcs
      typeCheck opApp1

      let discardedProcResults =
        intersect (freeVars opApp2) (map fst onPreProcs)
      errorIfFalse (length discardedProcResults == 0)
        (ResultingProcessDiscarded discardedProcResults)
      -- We use set type as they cannot be in scope
      mapM_ (uncurry setType) onPostProcs
      generalUnificationAllowed (typeCheck opApp2)

      onProcsCloned <-
        filterM (\ n -> getType n >>= (\ t -> return $ isOnProc t))
        procsCloned
      errorIfFalse (length onProcsCloned == 0)
        (OnProcessClonedError onProcsCloned)))


-- ***************************************************************************
-- Type Unification
-- ***************************************************************************
ensureIsSet :: Type -> TypeCheckMonad ()
ensureIsSet t =
  do
    fv <- freshTypeVar
    unify t (TSet fv)
    return ()

ensureIsProc :: Type -> TypeCheckMonad ()
ensureIsProc t =
  do
    t' <- compress t
    case t of
      TOnProcess _  -> return ()
      TOffProcess _ -> return ()
      _         -> unify t (TOnProcess Unknown) >> return ()

ensureIsChannel :: Name -> [Exp] -> TypeCheckMonad ()
ensureIsChannel n es =
  do
    freshVars <- replicateM (length es) freshTypeVar
    unifyTypeWithName n (TChannel (map TSet freshVars))
    return ()

unifyAll :: [Type] -> TypeCheckMonad Type
unifyAll [] = freshTypeVar
unifyAll [t] = return t
unifyAll (t1:ts) =
  do
```

```
    t2 <- unifyAll ts
    unify t1 t2

-- Unifies the two types, updating all types in the name map and returns the
-- unified type
-- Important: unification may not necessarily be symmetric -
-- see allowGeneralUnification
unify :: Type -> Type -> TypeCheckMonad Type
unify (TVar t1) (TVar t2) | t1 == t2 =
  return (TVar t1)
unify (TVar t1) (TVar t2) =
  do
    res1 <- readTypeRef t1
    res2 <- readTypeRef t2
    case (res1, res2) of
      (Left tv1, Left tv2)  -> applySubstitution t1 (TVar t2)
      (Left _, Right t)     -> unify (TVar t1) t
      (Right t, Left _)     -> unify t (TVar t2)
      (Right t1, Right t2)  -> unify t1 t2
unify (TVar a) b =
  do
    res <- readTypeRef a
    case res of
      Left tva  -> applySubstitution a b
      Right t   -> unify t b
unify b (TVar a) =
  do
    res <- readTypeRef a
    case res of
      Left tva  -> applySubstitution a b
      Right t   -> unify b t
unify (TSet a) (TSet b) =
  do
    tr <- unify a b
    return (TSet tr)
unify (TTuple ts1) (TTuple ts2) | length ts1 == length ts2 =
  do
    ts' <- zipWithM unify ts1 ts2
    return $ TTuple ts'
unify (TChannel ts1) (TChannel ts2) | length ts1 == length ts2 =
  do
    ts' <- zipWithM unify ts1 ts2
    return $ TChannel ts'
unify (TOperator ts1) (TOperator ts2) | length ts1 == length ts2 =
  do
    ts' <- zipWithM unify ts1 ts2
    return $ TOperator ts'
unify TEvent TEvent = return TEvent
unify TProcArg TProcArg = return TProcArg
unify (TOnProcess _) (TOnProcess _) = return $ TOnProcess Unknown
unify (TOffProcess _) (TOffProcess _) = return $ TOffProcess Unknown
unify (TOnProcess st1) (TOffProcess st2) =
  do
    r <- gets allowGeneralUnification
    if r then return $ TOnProcess Unknown else
      throwError $ UnificationError (TOnProcess st1) (TOffProcess st2)
unify t1 t2 =
  do
    t1' <- compress t1
    t2' <- compress t2
    throwError $ UnificationError t1' t2'


-- Returns the type that we substitute for
applySubstitution :: TypeVarRef -> Type -> TypeCheckMonad Type
applySubstitution (tvref @ (TypeVarRef tv _)) typ =
  do
    t' <- compress typ
    errorIfFalseM (liftM not (occurs tv typ)) (InfiniteUnificationError tv t')
    writeTypeRef tvref typ
    return typ

-- Apply the substitution type for tv to the type
{-substituteType :: (TypeVar, Type) -> Type -> TypeCheckMonad Type
substituteType (tv, t) (TVar a) = if a == tv then t else TVar a
substituteType _ (TEvent) = return TEvent
substituteType _ (TOnProcess) = return TOnProcess
substituteType _ (TOffProcess) = return TOffProcess
substituteType (tv, t) (TSet t1) = TSet (substituteType (tv,t) t1)
substituteType (tv, t) (TTuple ts) =
  TTuple (map (substituteType (tv,t)) ts)
-}

occurs :: TypeVar -> Type -> TypeCheckMonad Bool
occurs a (TVar (tvref @ (TypeVarRef tv _))) =
  do
    res <- readTypeRef tvref
    case res of
      Left tv -> return $ a == tv
      Right t -> occurs a t
occurs a (TSet t) = occurs a t
occurs a (TTuple ts) = liftM or (mapM (occurs a) ts)
occurs a (TChannel ts) = liftM or (mapM (occurs a) ts)
occurs a TEvent = return False
occurs a (TOnProcess _) = return False
occurs a (TOffProcess _) = return False
occurs a TProcArg = return False
```

C.2.4. *OpSemPrettyPrinter.hs.*

```
module OpSemPrettyPrinter where
import Text.PrettyPrint.HughesPJ

import OpSemDataStructures

-- ************************************************************************
-- Pretty Printer Extensions
-- ************************************************************************
angles :: Doc -> Doc
angles d = char '<' <> d <> char '>'

list :: [Doc] -> Doc
list docs =
  fsep (punctuate (text ",") docs)

tabHang :: Doc -> Doc -> Doc
tabHang d1 d2 = hang d1 4 d2


-- ************************************************************************
-- Basic instances
-- ************************************************************************
class PrettyPrintable a where
  prettyPrint :: a -> Doc

instance PrettyPrintable Int where
  prettyPrint n = int n
instance (PrettyPrintable a, PrettyPrintable b) =>
    PrettyPrintable (a,b) where
  prettyPrint (a,b) = parens (list [prettyPrint a, prettyPrint b])
instance PrettyPrintable a => PrettyPrintable [a] where
  prettyPrint xs = angles (list (map prettyPrint xs))

instance PrettyPrintable Name where
  prettyPrint (Name s)  = text s

instance PrettyPrintable Event where
  prettyPrint Tau = text "tau"
  prettyPrint (Event s) = prettyPrint s
  prettyPrint (ChanEvent n ns) =
    hcat $ punctuate (char '.') (prettyPrint n:map prettyPrint ns)

instance PrettyPrintable Exp where
  prettyPrint (OperatorApp n es) =
    prettyPrint n <> parens (list (map prettyPrint es))
  prettyPrint (Tuple es) =
    (parens . hsep . punctuate comma . map prettyPrint) es
  prettyPrint (Var n) = prettyPrint n

  prettyPrint (SetComprehension es scs) =
    braces (list (map prettyPrint es)
        <+> text "|" <+> list (map prettyPrint scs))

  prettyPrint (SigmaPrime) = text "SigmaPrime"
  prettyPrint (Sigma) = text "Sigma"
  prettyPrint (ProcArgs) = text "ProcArgs"
  prettyPrint (InductiveCase) = text "InductiveCase"
  prettyPrint (Powerset e) = text "Set" <> parens (prettyPrint e)
  prettyPrint (SetMinus s1 s2) =
    text "diff" <> parens (prettyPrint s1 <> comma <+> prettyPrint s2)
  prettyPrint (Union s1 s2) =
    text "union" <> parens (prettyPrint s1 <> comma <+> prettyPrint s2)
  prettyPrint (Intersection s1 s2) =
    text "inter" <> parens (prettyPrint s1 <> comma <+> prettyPrint s2)
  prettyPrint (ReplicatedUnion e) =
    text "Union" <> parens (prettyPrint e)
  prettyPrint (Set es) =
    braces (list (map prettyPrint es))
instance PrettyPrintable ProcessRelation where
  prettyPrint (Performs e1 ev e2) =
    prettyPrint e1 <+> text "=">> prettyPrint ev <> text"=>" <+> prettyPrint e2
```

```
instance PrettyPrintable Pattern where
  prettyPrint (PVar n) = prettyPrint n
  prettyPrint (PTuple ps) =
    (parens . hsep . punctuate comma . map prettyPrint) ps
  prettyPrint (PSet ps) =
    (braces . hsep . punctuate comma . map prettyPrint) ps

instance PrettyPrintable SideCondition where
  prettyPrint (SCGenerator pat exp) =
    prettyPrint pat <+> text "<-" <+> prettyPrint exp
  prettyPrint (Formula f) = prettyPrint f

instance PrettyPrintable PropositionalFormula where
  prettyPrint (Member pat exp) =
    text "member" <> parens (prettyPrint pat <> comma <+> prettyPrint exp)
  prettyPrint (Equals e1 e2) = prettyPrint e1 <+> text "==" <+> prettyPrint e2
  prettyPrint (Not f) = text "not" <+> prettyPrint f
  prettyPrint (And f1 f2) =
    prettyPrint f1 <+> text "and" <+> prettyPrint f2
  prettyPrint (Or f1 f2) =
    prettyPrint f1 <+> text "or" <+> prettyPrint f2
  prettyPrint (PFalse) = text "false"
  prettyPrint (PTrue) = text "true"

instance PrettyPrintable InductiveRule where
  prettyPrint (InductiveRule pres post scs) =
    let
      presString = show (vcat (map prettyPrint pres))
      postString = show (prettyPrint post)
      dashes 0 = empty
      dashes n = char '-' <> dashes (n-1)
    in
      text presString
      $+$ (dashes (max (length presString) (length postString))
        <+> list (map prettyPrint scs))
      $+$ text postString
```

C.2.5. *OpSemRules.hs.*

```
module OpSemRules (
  compileOperators, rulesFunctionToCSP,
  discardableArgsFunctionToCSP, operatorShortcutsToCSP,
  operatorDatatypeToCSP,
  replicatedOperatorsToCSP,
  channelsToCSP) where
import List
import Text.PrettyPrint.HughesPJ

import OpSemDataStructures
import OpSemTypeChecker
import Util

-- ************************************************************************
-- Transformer
-- ************************************************************************
compileOperators :: OpSemDefinition -> [CompiledOp]
compileOperators opSemDefn =
  let
    ops = operators opSemDefn
    transformOperator (op @ (Operator name args rules _)) =
      let
        Name cname = name
        cargs = args
        crules = map (transformInductiveRule ops op) rules
      in
        CompiledOp cname cargs crules (discardableArguments crules)
  in
    map transformOperator [op | op @ (Operator _ _ _ _) <- ops]

transformInductiveRule :: [Operator] -> Operator -> InductiveRule -> CompiledRule
transformInductiveRule ops op (InductiveRule pre post conditions) =
  let
    (Performs (OperatorApp name currentOperatorArgs) resultingEvent
              (OperatorApp newOperator newArgs)) = post
    -- All arguments of the operator of type proc
    procArgs = [pr | Var pr <- currentOperatorArgs,
                typeIsProc (getArgType op pr)]
    -- Map from proc |-> proc; if resultingProcs p' = p then it means
    -- p -> p' by some rule
    resultingProcs =
      [(pr', pr) | Performs (Var pr) _ (Var pr') <- pre]
    -- Map from proc to proc; resulting proc is an argument of the current
    -- operator (and thus has an index)
    procMap = resultingProcs
      ++ (identityFunction (procArgs \\ functionImage resultingProcs))
    currentOnProcs = map (getId op) (onProcs op)
    currentOffProcs = map (getId op) (offProcs op)

    resultingOperator @ (Operator _ resultingOperatorArgs _ _) =
      getOp ops newOperator
    resultingOperatorName = (name,
        [exp | (exp, (_, t)) <- zip newArgs resultingOperatorArgs,
            not (typeIsProc t)])
      where
        Name name = newOperator
    resultingOperatorProcessArgs =
      zip [0..] [arg | (Var arg, (_, TOnProcess _))
                  <- zip newArgs resultingOperatorArgs]
      ++ zip [(-length (offProcs resultingOperator))..(-1)]
             [arg | (Var arg, (_, TOffProcess _))
               <- zip newArgs resultingOperatorArgs]
    -- PartialFunction from arg id of resulting op to argument value
    resultingOnProcs =
      map (\ pid -> (pid, apply resultingOperatorProcessArgs pid))
        (map (getId resultingOperator) (onProcs resultingOperator))
    -- PartialFunction from arg id of resulting op to argument value
    resultingOffProcs =
      map (\ pid -> (pid, apply resultingOperatorProcessArgs pid))
        (map (getId resultingOperator) (offProcs resultingOperator))

    -- If procMap pid = pid' then it means the pid of the new operator
    -- is pid' of the old process
    onProcMap =
      [(pid, getId op (apply procMap pr')) | (pid,pr') <- resultingOnProcs]
    -- Map from pid to [0..]; if newProcMap pid = n it means that the
    -- nth turned on argument of the resulting operator is the pid off
    -- argument of the current operator
    newProcMap =
      zip [pid' | (pid, pid') <- onProcMap, elem pid' currentOffProcs] [0..]

    phi = [(getId op n, ev) | Performs (Var n) ev (Var n') <- pre]
    psi = [(pid, if pid' < 0 then apply newProcMap pid' else pid')
        | (pid, pid') <- onProcMap]
    f = invert newProcMap
    chi =
      [(pid, getId op (apply procMap pr')) | (pid,pr') <- resultingOffProcs]
    discards =
      currentOnProcs
      \\ (map (getId op) (mapPF procMap (functionImage resultingOnProcs)))
    generators = map transformSideCondition conditions
    boundVars = varsBound conditions
  in
{-    error (
      show resultingOperator ++"\n\n"
      ++ show resultingOperatorProcessArgs ++"\n\n"
      ++ show resultingOnProcs ++"\n\n"
      ++ show resultingOffProcs ++"\n\n"
      ++ show procMap++"\n\n"
      ++ show phi ++"\n\n"
      ++ show psi ++"\n\n"

      ++show resultingOperatorName++"\n\n"
      ++show resultingEvent++"\n\n"
      ++show f++"\n\n"
      ++show chi++"\n\n"
      ++show discards++"\n\n"
      ++show generators++"\n\n"
    )
-}
    CompiledRule phi resultingEvent resultingOperatorName f psi
      chi discards boundVars generators

transformSideCondition :: SideCondition -> Stmt
transformSideCondition (SCGenerator p e) = Generator p e
transformSideCondition (Formula f) = PropFormula f

getOp :: [Operator] -> Name -> Operator
getOp ops n =
  head [op | (op @ (Operator n' _ _ _)) <- ops, n==n']

getArgType :: Operator -> Name -> Type
getArgType (Operator _ args _ _) n =
  head [t | (n', t) <- args, n==n']

-- A process is off iff there exists no rule that allows it to perform any
-- visible event
offProcs :: Operator -> [Name]
offProcs (op @ (Operator n args rules _)) =
  [Name pr | (Name pr, TOffProcess _) <- args]

onProcs :: Operator -> [Name]
```

```
onProcs (op @ (Operator _ args rules _)) =
  [Name pr | (Name pr, TOnProcess _) <- args]

getId :: Operator -> Name -> ProcId
getId op n =
  head ([id | (n', id) <- zip (onProcs op) [0..], n==n']
       ++[id | (n', id) <- zip (offProcs op) [-(length (offProcs op))..(-1)],
           n==n']
  )

-- ********************************************************************
-- Discardable Arguments
-- ********************************************************************
discardableArguments :: [CompiledRule] -> [ProcId]
discardableArguments rules =
  (nub . concat)
    [discards | CompiledRule _ _ _ _ _ _ discards _ _ <- rules]



-- ********************************************************************
-- Pretty Printer
-- ********************************************************************
opNameToCSP s args =
  hcat (punctuate (char '.') ((text "Op_" <> text s):map toCSP args))

channelsToCSP :: OpSemDefinition -> Doc
channelsToCSP opSemDefn =
  let
    chans = channels opSemDefn
    ppChan (Channel n exps) =
      text "channel" <+> toCSP n <+>
        if length exps == 0 then empty
        else text ":" <+> hcat (punctuate (char '.') (map setExpr exps))
    systemEvents =
      text "SystemEvents_=_union(UserEvents,_{|"
        <+> list (map (\ (Channel n _) -> toCSP n) chans) <+> text "|})"
    setExpr e = toCSP e
  in
    vcat (systemEvents:map ppChan chans)

rulesFunctionToCSP :: [CompiledOp] -> Doc
rulesFunctionToCSP ops =
  let
    ppOp (CompiledOp name args rules discards) =
      tabHang (text "Rules" <> parens (opNameToCSP name nonProcArgs)
            <+> equals)
          (text "concat" <> (parens . angles . list . map toCSP) rules)
        where
          nonProcArgs = [n | (n, t) <- args, not (typeIsProc t)]
  in
    vcat (map ppOp ops)

discardableArgsFunctionToCSP :: [CompiledOp] -> Doc
discardableArgsFunctionToCSP ops =
  let
    ppOp (CompiledOp name args rules discards) =
      tabHang (text "DiscardableArgs" <> parens (opNameToCSP name nonProcArgs)
          <+> equals)
          (toCSP discards)
        where
          nonProcArgs = [n | (n, t) <- args, not (typeIsProc t)]
  in
    vcat (map ppOp ops)

operatorShortcutsToCSP :: [CompiledOp] -> Doc
operatorShortcutsToCSP ops =
  let
    ppOp(CompiledOp name args rules discards) =
      tabHang (text (name++"'") <>
          (if length args == 0 then empty
          else parens (list (map toCSP [n | (n,t) <- args])))
          <+> equals)
          (text "Operator_M::Operator"
          <> parens (opName <> comma <+>
                toCSP [n | (n, TOnProcess _) <- args] <> comma <+>
                toCSP [n | (n, TOffProcess _) <- args]))
        where
          opName = hcat (punctuate (char '.')
            ((text "Op_" <> text name):map (toCSP . fst) nonProcArgs))
          nonProcArgs = [(n,t) | (n, t) <- args, not (typeIsProc t)]
  in
    vcat (map ppOp ops)

operatorDatatypeToCSP :: [CompiledOp] -> Doc
operatorDatatypeToCSP ops =
  let
    ppOp (CompiledOp name args rules discards) =
        hcat (punctuate (char '.')
          ((text "Op_" <> text name):map constructor nonProcTypes))
      where
        nonProcTypes = [t | (n, t) <- args, not (typeIsProc t)]
        constructor TEvent = text "UserEvents"
        constructor (TSet a) = text "Set" <> parens (constructor a)
        constructor (TTuple ts) =
          let
            names = zip ts ['a'..'z']
            generator (t, n) =
              char n <+> text "<-" <+> constructor t
          in
            braces (
              (parens (list (map (char . snd) names))
              <+> char '|' <+>
              (sep . punctuate comma . map generator) names
              ))
        constructor' t = error (show t)
  in
    text "datatype_Operators_=_"
    $$ nest 4 (vcat ((\(x:xs) -> x:(map ((<+>) (char '|')) xs)) (map ppOp ops)))

replicatedOperatorsToCSP :: OpSemDefinition -> Doc
replicatedOperatorsToCSP opSemDefn =
  vcat (map replicatedOperatorToCSP
      [repOp | repOp @ (ReplicatedOperator _ _ _ _ _) <- operators opSemDefn])

replicatedOperatorToCSP :: Operator -> Doc
replicatedOperatorToCSP
  (ReplicatedOperator (Name n) args (basePats, baseExp)
            (inductiveVars, inductiveCase) _) =

  tabHang (
    text (n++"'")
    <> (parens . hsep . punctuate comma $
      map basePat basePats)
    <+> char '=')
    (setExpr baseExp)
  $$
  tabHang (
    text (n++"'")
    <> (parens . hsep . punctuate comma $
      zipWith (\ current inductive ->
            angles (toCSP current) <> char '^' <> toCSP inductive)
          (map fst args) inductiveVars)
    <+> char '=')
    (setExpr inductiveCase)

  where
    basePat (PVar (Name s)) = text s
    basePat (PSet ps) = angles (list $ map basePat ps)

    setExpr (OperatorApp (Name n) es) =
      text (n++"'") <>
        if length es == 0 then empty else parens (list (map setExpr es))
    setExpr (Tuple es) =
      (parens . hsep . punctuate comma . map setExpr) es
    setExpr (Var n) =
      if n `elem` inductiveVars then text "set" <> parens (toCSP n)
      else toCSP n
    setExpr SigmaPrime = text "SystemEvents"
    setExpr Sigma = text "UserEvents"
    setExpr ProcArgs = text "ProcArgs"
    setExpr (Powerset e) = text "Set" <> parens (setExpr e)
    setExpr (SetMinus s1 s2) =
      text "diff" <> parens (setExpr s1 <> comma <+> setExpr s2)
    setExpr (Union s1 s2) =
      text "union" <> parens (setExpr s1 <> comma <+> setExpr s2)
    setExpr (Intersection s1 s2) =
      text "inter" <> parens (setExpr s1 <> comma <+> setExpr s2)
    setExpr (SetComprehension exps stmts) =
      braces ((hsep . punctuate comma . map setExpr $ exps)
          <+> char '|' <+> (hsep . punctuate comma . map toCSP $ stmts))
    setExpr (InductiveCase) =
      text (n++"'")
      <> parens (hsep (punctuate comma (map toCSP inductiveVars)))
    setExpr (ReplicatedUnion s) =
      text "Union" <> (parens (setExpr s))



-- ********************************************************************
-- Data Types
-- ********************************************************************
class ToCSP a where
  toCSP :: a -> Doc
```

```
-- ********************************************************************
-- Pretty Printer Extensions
-- ********************************************************************
list :: [Doc] -> Doc
list docs =
  fsep (punctuate (text ",") docs)

tabHang :: Doc -> Doc -> Doc
tabHang d1 d2 = hang d1 4 d2

angles :: Doc -> Doc
angles d = char '<' <> d <> char '>'

-- ********************************************************************
-- Basic instances
-- ********************************************************************

instance ToCSP Int where
  toCSP n = int n
instance (ToCSP a, ToCSP b) =>
    ToCSP (a,b) where
  toCSP (a,b) = parens (list [toCSP a, toCSP b])
instance ToCSP a => ToCSP [a] where
  toCSP xs = angles (list (map toCSP xs))

instance ToCSP Name where
  toCSP (Name "callProc") = text "callProc"
  toCSP (Name s) = text (s++"0")

instance ToCSP Event where
  toCSP Tau = text "tau"
  toCSP (Event s) = toCSP s
  toCSP (ChanEvent n ns) =
    hcat (punctuate (char '.') (toCSP n:map toCSP ns))

instance ToCSP Exp where
  toCSP (OperatorApp n es) = toCSP n <> parens (list (map toCSP es))
  toCSP (Tuple es) =
    (parens . hsep . punctuate comma . map toCSP) es
  toCSP (Var n) = toCSP n

  toCSP (Powerset e) = text "Set" <> parens (toCSP e)
  toCSP (ProcArgs) = text "ProcArgs"
  toCSP (Sigma) = text "UserEvents"
  toCSP (SigmaPrime) = text "SystemEvents"
  toCSP (SetMinus s1 s2) =
    text "diff" <> parens (toCSP s1 <> comma <+> toCSP s2)
  toCSP (Union s1 s2) =
    text "union" <> parens (toCSP s1 <> comma <+> toCSP s2)
  toCSP (Intersection s1 s2) =
    text "inter" <> parens (toCSP s1 <> comma <+> toCSP s2)
  toCSP (SetComprehension exps stmts) =
    braces ((hsep . punctuate comma . map toCSP $ exps)
        <+> char '|' <+> (hsep . punctuate comma . map toCSP $ stmts))
  toCSP (Set exps) = braces (hsep . punctuate comma . map toCSP $ exps)
instance ToCSP SideCondition where
  -- SETS
  toCSP (SCGenerator p e) =
    toCSP p <+> text "<-" <+> toCSP e
instance ToCSP PropositionalFormula where
  toCSP (Member p e) = text "member" <> parens (list [toCSP p, toCSP e])
  toCSP (Subset e1 e2) = toCSP e1 <+> text "<=" <+> toCSP e2
  toCSP (Equals e1 e2) = toCSP e1 <+> text "==" <+> toCSP e2
  toCSP (Not f) = text "not" <+> toCSP f
  toCSP (And f1 f2) = toCSP f1 <+> text "and" <+> toCSP f2
  toCSP (Or f1 f2) = toCSP f1 <+> text "or" <+> toCSP f2
  toCSP (PFalse) = text "false"
  toCSP (PTrue) = text "true"
instance ToCSP Pattern where
  toCSP (PVar n) = toCSP n
  toCSP (PTuple ps) = parens (list (map toCSP ps))
  toCSP (PSet ps) = braces (list (map toCSP ps))

instance ToCSP Stmt where
  toCSP (Generator exp set) =
    toCSP exp <+> text "<-" <+> toCSP set
  toCSP (PropFormula f) = toCSP f

instance ToCSP CompiledRule where
  toCSP (CompiledRule phi resultingEvent (n, args) f psi chi discards
            namesBound generators) =
    angles (parens (list [
      toCSP phi, toCSP resultingEvent,
      opNameToCSP n args, toCSP f, toCSP psi,
      toCSP chi, toCSP discards
    ]) $$
      if length generators == 0 then empty
      else char '|' <+>
        parens (list $ map toCSP namesBound) <+> text "<-"
            <+> text "seq" <+> parens (braces (
              parens (list $ map toCSP namesBound) <+> char '|' <+>
              list (map toCSP generators))))
```

C.2.6. *ConstantCode.hs.*

```
{-# LANGUAGE QuasiQuotes #-}
module ConstantCode where
import Util

-- ********************************************************************************
-- ********************************************************************************
operatorModuleNotExported = [$multilineLiteral|
-- Gives the set of all sequences of type t of length <= length
FinSeq(t, length) =
  let
    Gen(0) = {<>}
    Gen(n) = {<x>^xs, xs | x <-t, xs <- Gen(n-1)}
  within
    Gen(length)
FinSeq'(t, length) =
  let
    Gen(0) = <<>>
    Gen(n) = concat(< <<x>^xs,xs> | xs <- Gen(n-1), x <- t>)
  within
    Gen(length)

powerSeq(<>) = <<>>
powerSeq(<x>^xs) = <<x>^ys, ys | ys <- powerSeq(xs)>

zip(<>, _) = <>
zip(_, <>) = <>
zip(<x>^xs, <y>^ys) = <(x,y)>^zip(xs, ys)

flatmap(f,<>) = <>
flatmap(f,<x>^xs) = f(x)^flatmap(f,xs)

remdups(x) =
  let
    iter(<>,X) = <>
    iter(<x>^xs,X) =
      if member(x,X) then iter(xs,X)
      else <x>^iter(xs,union(X,{x}))
  within
    iter(x, {})

foldr(f, e, <>) = e
foldr(f, e, <x>^xs) = f(x, foldr(f, e, xs))

foldl(f, e, <>) = e
foldl(f, e, <x>^xs) = foldl(f, f(e, x), xs)

-- ********************************************************************
-- Partial functions
-- ********************************************************************

functionDomain(f) = {x | (x,_) <- f}
functionDomainSeq(f) = <x | (x,_) <- f>
functionImage(f) = {x | (_,x) <- f}
functionImageSeq(f) = <x | (_,x) <- f>
identityFunction(domain) = {(x,x) | x <- domain}
identityFunctionSeq(domain) = <(x,x) | x <- domain>
invert(f) = {(a,b) | (b,a) <- f}
invertSeq(f) = <(a,b) | (b,a) <- f>

composeFunctions(fs1, fs2) = {(a, apply(fs1, b)) | (a, b) <- fs2}
composeFunctionsSeq(fs1, fs2) = <(a,applySeq(fs1,b)) | (a,b) <- fs2>

apply(f, x) =
  extract({a | (x', a) <- f, x == x'})

applySeq(f, x) =
  let extract(<x>) = x
  within
    extract(<a | (x', a) <- f, x == x'>)

mapOverSet(f, X) =
  {apply(f, x) | x <- X}
mapOverSeq(f, <>) = <>
mapOverSeq(f, <x>^xs) = <applySeq(f,x)>^mapOverSeq(f,xs)

seqDiff(xs, ys) = <x | x <- xs, not elem(x,ys)>
```

```
seqInter(xs, ys) = <x | x <- xs, elem(x, ys)>
seqUnion(xs, ys) = remdups(xs^ys)

-- ********************************************************************
-- Semantics Calculation
-- ********************************************************************

-- Returns a partial function from (op, onProcMap, procCount, offProcMap) to
-- the possible internal events
InternalEventsFromOperator(op, onProcMap, procCount, offProcMap, nextId, doneCalls) =
  let
    -- The sequence of all possible events that each rule gives
    (possibleEvents, nextId') =
      let
        process((events, nextId),
            rule @@ (phi, x, mu, f, xi, chi, discards)) =
          let
            procsToDiscard =
              mapOverSeq(onProcMap, discards)
            procEvents =
              <(applySeq(onProcMap, p), e) | (p, e) <- phi>
            newProcs = composeFunctionsSeq(offProcMap, f)
            thisEvent =
              (rule, procEvents, x, procsToDiscard, procCount, newProcs)
          within
            (<(nextId,thisEvent)>^events, nextId+1)
      within
        foldl(process, (<>, nextId), Rules(op))

    -- The sequence of recursive calls to this function to make,
    -- it contains no duplicates and no items in doneCalls.
    recursiveCallsToMake =
      seqDiff(
        remdups(<
          let
            newOnProcMap =
              composeFunctionsSeq(
                concat(<onProcMap,
                     identityFunctionSeq(
                       <procCount..procCount+newProcCount-1>)
                     >),
                     xi)
            newProcCount = procCount + length(f)
            newOffProcMap =
              composeFunctionsSeq(offProcMap, chi)
          within
            (mu, newOnProcMap, newProcCount, newOffProcMap)
          | (phi, x, mu, f, xi, chi, discards) <- Rules(op), op != mu>),
        doneCalls)

    doneCalls' = doneCalls^recursiveCallsToMake

    (recursiveEvents, recursiveDiscardableArgs, nextId'') =
      let
        process((events, discardableArgs, nextId), (mu, xi, m, chi)) =
          let
            (events', discardableArgs', nextId') =
              InternalEventsFromOperator(mu, xi, m, chi, nextId, doneCalls')
          within
            (events'^events, remdups(discardableArgs'^discardableArgs), nextId')
      within
        foldl(process, (<>, <>, nextId'), recursiveCallsToMake)
  within
    (<((op, onProcMap, procCount, offProcMap), possibleEvents)>^recursiveEvents,
      remdups(mapOverSeq(onProcMap, DiscardableArgs(op))^recursiveDiscardableArgs),
      nextId'')
|]


-- ********************************************************************
-- ********************************************************************
operatorModuleExported = [$multilineLiteral|
-- Allows the rules to specify a tau transition
channel tau
-- Let a process be turned off
channel off
-- Represents a primed event; used to let processes turn themselves off
channel prime : SystemEvents
channel renamed : {0..20000}

-- ********************************************************************
-- Main Simulator Function
-- ********************************************************************
--  startOperator  operator the process starts in
--  onProcesses    sequence of processes that are initially on
--  offProcesses   sequence of processes that are initially off
Operator(startOperator, onProcesses, offProcesses) =
let
  OnProcessCount = length(onProcesses)
  OnProcesses =
    zip(<0..OnProcessCount - 1>, onProcesses)
  OffProcessCount = length(offProcesses)
  OffProcesses =
    zip(<(-OffProcessCount)..-1>, offProcesses)

  InternalEvents =
    concat(<es | (id, es) <- InternalEventsByOperator>)

  (InternalEventsByOperator, discardableProcs, _) =
    InternalEventsFromOperator(startOperator,
      identityFunctionSeq(<0..OnProcessCount-1>), OnProcessCount,
      identityFunctionSeq(<(-OffProcessCount)..-1>),
      0, <>)

  RenamingsForProc(id) =
    let
      calc((rid, (rule, procs,b,discards,m,f))) =
        if elem(id, functionDomainSeq(procs)) then
          if elem(id,discards) then
            <(prime.applySeq(procs, id), rid)>
          else
            <(applySeq(procs, id), rid)>
        else
          if elem(id,discards) then
            <(off, rid)>
          else <>
    within
      flatmap(calc,InternalEvents)

  Process(proc, id) =
    (if elem(id, discardableProcs) then
      explicate((proc[[a <- prime.a, a <- a | a <- SystemEvents]]
      [| {| prime |} |> STOP)
      /\ off -> STOP)
    else
      proc)
    [[a <- renamed.b | (a,b) <- set(RenamingsForProc(id))]]

  -- onProcMap  Function from current process id to actual process id
  -- procCount  Current number of on processes
  -- offProcMap Function from current off process id to actual off
  --          process ids
  Reg(currentOperator, onProcMap, procCount, offProcMap) =
    [] (rid, (rule @@ (phi, x, mu, f, xi, chi, discards), _, _, _, _, _)) :
      set(applySeq(InternalEventsByOperator,
            (currentOperator, onProcMap, procCount, offProcMap))) @
      let
        procsToDiscard =
          mapOverSeq(onProcMap, discards)
        procEvents =
          <(applySeq(onProcMap, p), e) | (p, e) <- phi>
        newOnProcMap =
          composeFunctionsSeq(concat(<onProcMap,
                        identityFunctionSeq(<procCount..procCount+newProcCount-1>)>)
                      , xi)
        newProcCount = procCount + length(newProcs)
        newProcs = composeFunctionsSeq(offProcMap, f)
        newOffProcMap =
          composeFunctionsSeq(offProcMap, chi)
      within
        renamed.rid ->
          if length(newProcs) == 0 then
            Reg(mu, newOnProcMap, newProcCount, newOffProcMap)
          else
            ((|| id : {procCount..newProcCount-1}
            @ [AlphaProcess(id)]
              Process(applySeq(OffProcesses,
                   applySeq(newProcs, id-procCount)), id))
            [| AlphaProcesses(newProcCount) \|]
            Reg(mu, newOnProcMap, newProcCount, newOffProcMap))

  AlphaProcess(id) =
    set(<renamed.b | (a,b) <- RenamingsForProc(id)>)
  -- Important: /= Renamings beacuse there could be events that
  -- happen because of no processes events (cf internal choice).
  AlphaProcesses(maxId) = Union({AlphaProcess(id) | id <- {0..maxId-1}})

  H = {tau}
within
  ((((|| id : {0..OnProcessCount-1} @
    [AlphaProcess(id)] Process(applySeq(OnProcesses, id), id))
  )
  [| AlphaProcesses(OnProcessCount) \|]
  Reg(startOperator, identityFunctionSeq(<0..OnProcessCount-1>),
```

```
      OnProcessCount, identityFunctionSeq(<(-OffProcessCount)..-1>))
    )[[renamed.r <- b
      | (r,b) <-
        set(<(r,b) | (r,(_,_,b,_,_,_)) <- InternalEvents>)]]
    \ H
|]


globalModule = [$multilineLiteral|
transparent explicate, diamond

-- ********************************************************************
-- Recursion control procedures
-- ********************************************************************
channel callProc, startProc : ProcArgs

CallProc(proc) = callProc.proc -> STOP

WrapThread(proc) =
  let
    RecursionRegulator =
      callProc?p -> startProc!p -> RecursionRegulator
      [] [] e : Operator_M::SystemEvents @ e -> RecursionRegulator
    Thread(proc) =
      GetProc(proc)
      [| {| callProc |} |>
      startProc?proc' -> Thread(proc')
  within
    -- diamond removes the tau's from the resulting LTS (but never
    -- increases the size of the resulting LTS, cf. normalize)
    -- This removes the problem of the new events being introduced.
    diamond(
      (Thread(proc)|]
  ++"_[|_union(Operator_M::SystemEvents,_{|_callProc,_startProc_|})_|]_"++
  [$multilineLiteral|RecursionRegulator)
      \ {| startProc, callProc |}
    )
|]
```

C.3. **CSPM.**

C.3.1. *CSPMDataStructures.hs.*

```
module CSPMDataStructures where

import Control.Monad.Trans
import Data.IORef
import Text.PrettyPrint.HughesPJ
import qualified OpSemDataStructures as OpSem
import Util

newtype ModuleName = ModuleName String deriving (Eq, Show)

newtype Name = Name String deriving (Eq, Ord, Show)

data QualifiedName =
  UnQual Name
  | Qual ModuleName Name
  deriving (Eq, Show)

data Literal =
  Int Integer
  | Bool Bool
  deriving (Eq, Show)

data Annotated typeType innerType =
  Annotated SrcLoc typeType innerType

instance Show b => Show (Annotated a b) where
  show (Annotated _ _ inner) = "("++show inner++")"

instance Eq b => Eq (Annotated a b) where
  (Annotated _ _ inner1) == (Annotated _ _ inner2) = inner1 == inner2

removeAnnotation :: Annotated t1 t2 -> t2
removeAnnotation (Annotated _ _ e) = e

-- P = post parsing, TC = post typechecking, An = annotated
type AnModule = Annotated () Module
-- Declarations may bind multiple names
type AnDecl = Annotated PSymbolTable Decl
type AnMatch = Annotated () Match
type AnPat = Annotated () Pat
type AnExp = Annotated PType Exp
type AnStmt = Annotated () Stmt
type AnDataTypeClause = Annotated () DataTypeClause

type PModule = AnModule
type PDecl = AnDecl
type PMatch = AnMatch
type PPat = AnPat
type PExp = AnExp
type PStmt = AnStmt
type PDataTypeClause = AnDataTypeClause

type TCModule = AnModule
type TCDecl = AnDecl
type TCMatch = AnMatch
type TCPat = AnPat
type TCExp = AnExp
type TCStmt = AnStmt
type TCDataTypeClause = AnDataTypeClause

data SrcLoc =
  SrcLoc String Int Int -- filename, line, column
  | SrcSpan String (Int, Int) (Int, Int)
  deriving (Eq)

instance Show SrcLoc where
  show (SrcLoc fname line column) = fname++":"++show line++":"++show column
  show (SrcSpan fname (line1, col1) (line2, col2)) =
    fname++":"++show line1++":"++show col1++"-"++show line2++":"++show col2

-- ********************************************************************
-- Modules
-- ********************************************************************
data Module =
  Module ModuleName [AnDecl] [AnDecl] -- internal, exported
  | GlobalModule [AnDecl]
  deriving (Eq, Show)

-- ********************************************************************
-- Expressions
-- ********************************************************************
data BinaryBooleanOp =
  And
  | Or
  | Equals
  | NotEquals
  | LessThan
  | GreaterThan
  | LessThanEq
  | GreaterThanEq
  deriving (Eq, Show)

data BooleanUnaryOp =
  Not
  deriving (Eq, Show)

data MathsOp =
  Divide | Minus | Mod | Plus | Times
  deriving (Eq, Show)

data Exp =
  App AnExp [AnExp]
  | BooleanBinaryOp BinaryBooleanOp AnExp AnExp
  | BooleanUnaryOp BooleanUnaryOp AnExp
  | Concat AnExp AnExp
  | DotApp AnExp AnExp
  | If AnExp AnExp AnExp
  | Lambda AnPat AnExp
  | Let [AnDecl] AnExp
  | Lit Literal
  | List [AnExp]
  | ListComp [AnExp] [AnStmt]
  | ListEnumFrom AnExp
  | ListEnumFromTo AnExp AnExp
  | ListLength AnExp
  | MathsBinaryOp MathsOp AnExp AnExp
  | NegApp AnExp
  | Paren AnExp
  | Set [AnExp]
  | SetComp [AnExp] [AnStmt]
  | SetEnum [AnExp]      -- {| |}
  | SetEnumComp [AnExp] [AnStmt]  -- {|c.x | x <- xs|}
  | SetEnumFrom AnExp
  | SetEnumFromTo AnExp AnExp
  | Tuple [AnExp]
  | Var QualifiedName

  -- Processes
```

```
  | ReplicatedUserOperator Name [AnExp] [AnStmt]
  | UserOperator Name [AnExp]

  {-
  | AlphaParallel AnExp AnExp AnExp
  | Event Name [Component]
  | Exception AnExp AnExp
  | ExtChoice AnExp AnExp
  | Interrupt AnExp AnExp
  | Prefix AnExp AnExp
  | RepAlphaParallel [Stmt] AnExp AnExp -- first exp is the alphabet
  | RepExtChoice [AnStmt a] AnExp
  | Rename AnExp [AnStmt a] [AnStmt a]  -- First one should only contain generators
  -}

  deriving (Eq, Show)

data Stmt =
  Generator AnPat AnExp
  | Qualifier AnExp
  deriving (Eq, Show)

{-
data Component =
  Input Name
  | Output Exp
  | InputRes Name Exp   -- ?x : {0..1}
  deriving (Eq, Show)
-}

-- ***************************************************************************
-- Declarations
-- ***************************************************************************
data Decl =
  -- Third argument is the annotated type
  FunBind Name [AnMatch] (Maybe [AnExp])
  | PatBind AnPat AnExp
  | Channel [Name] [AnExp]
  | Assert AnExp AnExp Model
  | DataType Name [AnDataTypeClause]
  | External [Name]
  | Transparent [Name]
  deriving (Eq, Show)

data Model =
  Traces | Failures | FailuresDivergences
  deriving (Eq, Show)

data DataTypeClause =
  DataTypeClause Name [AnExp]
  deriving (Eq, Show)

data Match =
  Match [[AnPat]] AnExp -- allows for decls like drop(xs,ys)(n)
  deriving (Eq, Show)

data Pat =
  PConcat AnPat AnPat
  | PDotApp AnPat AnPat
  | PDoublePattern AnPat AnPat
  | PList [AnPat]
  | PLit Literal
  | PParen AnPat              -- Need to add PSet
  | PSet [AnPat]
  | PTuple [AnPat]
  | PVar Name
  | PWildCard
  deriving (Eq, Show)


-- ***************************************************************************
-- Types
-- ***************************************************************************
newtype TypeVar = TypeVar Int deriving (Eq, Show)

data TypeScheme =
  ForAll [(TypeVar, [Constraint])] Type
  deriving (Eq, Show)

data Constraint =
  Eq | Ord
  deriving (Eq, Ord, Show)

data Type =
  TVar TypeVarRef
  | TInt
  | TFunction [Type] Type -- Arguments to result type
  | TSeq Type

  | TList Type Type   -- The second type should either be a TVar or TList
  | TListEnd      -- or a TEmptyList
  | TPolyList TypeVarRef  -- Polymorphic list thingy - should be a TypeVarRef

  | TBool
  | TSet Type
  | TTuple [Type]

  | TChannel Type      -- Should be a TList
  | TDotable Type Type  -- TDotted a b means that this type can be dotted
                -- with an a to yield a b
  | TDatatypeClause Name [Type]
    -- Op_Parallel.T1.T2.T3 :: TDatatypeClause "Operators" [T1,T2,T3]

  | TProc
  deriving (Eq, Show)

data TypeVarRef =
  TypeVarRef TypeVar [Constraint] PType
  deriving Eq

instance Show TypeVarRef where
  show (TypeVarRef tv cs _) = "TypeVarRef_"++show tv ++ show cs


opSemTypeToCSPMType :: OpSem.Type -> Type
opSemTypeToCSPMType (OpSem.TEvent) = TChannel TListEnd
opSemTypeToCSPMType (OpSem.TOnProcess _) = TProc
opSemTypeToCSPMType (OpSem.TOffProcess _) = TProc
opSemTypeToCSPMType (OpSem.TSet t) = TSet (opSemTypeToCSPMType t)
opSemTypeToCSPMType (OpSem.TTuple ts) = TTuple (map opSemTypeToCSPMType ts)

opSemOperatorNameToCSPMName (OpSem.Name s) = Name s


type SymbolTable = PartialFunction Name TypeScheme
type PType = IORef (Maybe Type)
type PTypeScheme = IORef (Maybe TypeScheme)
type PSymbolTable = IORef SymbolTable

readPType :: (MonadIO m) => PType -> m (Maybe Type)
readPType ioref =
  do
    t <- liftIO $ readIORef ioref
    return t
setPType :: (MonadIO m) => PType -> Type -> m ()
setPType ioref t = liftIO $ writeIORef ioref (Just t)
freshPType :: (MonadIO m) => m PType
freshPType = liftIO $ newIORef Nothing

readPSymbolTable :: (MonadIO m) => PSymbolTable -> m SymbolTable
readPSymbolTable ioref =
  do
    t <- liftIO $ readIORef ioref
    return t
setPSymbolTable :: (MonadIO m) => PSymbolTable -> SymbolTable -> m ()
setPSymbolTable ioref t = liftIO $ writeIORef ioref t
freshPSymbolTable :: (MonadIO m) => m PSymbolTable
freshPSymbolTable = liftIO $ newIORef []


readPTypeScheme :: (MonadIO m) => PTypeScheme -> m (Maybe TypeScheme)
readPTypeScheme ioref =
  do
    t <- liftIO $ readIORef ioref
    return t
setPTypeScheme :: (MonadIO m) => PTypeScheme -> TypeScheme -> m ()
setPTypeScheme ioref t = liftIO $ writeIORef ioref (Just t)
freshPTypeScheme :: (MonadIO m) => m PTypeScheme
freshPTypeScheme = liftIO $ newIORef Nothing



prettyPrintTypeScheme :: TypeScheme -> Doc
prettyPrintTypeScheme (ForAll ts t) =
  (if length ts > 0 then
    text "forall" <+> hsep (punctuate comma
              [parens (hsep (punctuate comma (map ppConstraint cs)) <+>
                char (apply vmap n)) | (TypeVar n, cs) <- ts])
    <+> text ":"
  else
    empty)
    <+> prettyPrintType vmap t
  where
```

```
    ppConstraint Eq = text "Eq"
    ppConstraint Ord = text "Ord"
    vmap = zip (map (\ (TypeVar n, _) -> n) ts) ['a'..'z']
prettyPrintType :: PartialFunction Int Char -> Type -> Doc
prettyPrintType vmap (TVar (TypeVarRef (TypeVar n) cs ioref)) =
  case safeApply vmap n of
    Just c  -> char c
    Nothing -> int n
prettyPrintType vmap (TFunction targs tr) =
  parens (hsep (punctuate comma (map (prettyPrintType vmap) targs)))
  <+> text "->" <+> prettyPrintType vmap tr
prettyPrintType vmap (TSeq t) =
  char '<' <> prettyPrintType vmap t <> char '>'
prettyPrintType vmap (TSet t) =
  char '{' <> prettyPrintType vmap t <> char '}'
prettyPrintType vmap (TTuple ts) =
  parens (hsep (punctuate comma (map (prettyPrintType vmap) ts)))
prettyPrintType vmap (TDotable t1 t2) =
  text "DotableWith" <+> prettyPrintType vmap t1 <+> text "yielding"
  <+> prettyPrintType vmap t2
prettyPrintType vmap (TChannel TListEnd) =
  text "Event"
prettyPrintType vmap (TChannel t1) =
  let
    chanType TListEnd = []
    chanType (TPolyList tv) = [TPolyList tv]
    chanType (TList t1 t2) = t1:chanType t2

    ts = chanType t1
  in
    text "Channel" <+>
      brackets (hsep (punctuate comma (map (prettyPrintType vmap) ts)))
prettyPrintType vmap (TListEnd) = text "Event"
prettyPrintType vmap (TList t1 t2) =
  prettyPrintType vmap t1 <> comma <+> prettyPrintType vmap t2
prettyPrintType vmap (TPolyList (TypeVarRef (TypeVar n) cs ioref)) =
  text "PolyList" <+> char (apply vmap n)
prettyPrintType vmap (TDatatypeClause (Name n) []) = text n
prettyPrintType vmap (TDatatypeClause (Name n) ts) =
  text "DataTypeClause" <+> text n <+>
  brackets (hsep (punctuate comma (map (prettyPrintType vmap) ts)))

prettyPrintType vmap (TBool) = text "Bool"
prettyPrintType vmap (TInt) = text "Int"
prettyPrintType vmap (TProc) = text "Proc"
```

C.3.2. *CSPMParser.hs.*

```
{-# LANGUAGE QuasiQuotes #-}
module CSPMParser (parseCSPMFile, stringParser) where

import Control.Monad(liftM, sequence)
import Control.Monad.Trans
import Control.Monad.Error
import List
import qualified Text.Parsec.Expr as E
import Text.Parsec.Language
import Text.Parsec
import qualified Text.Parsec.Token as PT

import qualified CSPMDataStructures as CSP
import OpSemDataStructures
import OperatorParsers
import Util

data CSPMParserState =
  CSPMParserState {
    operatorSyntax :: PartialFunction Name (Maybe OperatorSyntax),
    repOperatorSyntax :: PartialFunction Name (Maybe OperatorSyntax),
    canUseAngles :: Bool
  }
  deriving Show

gets :: (CSPMParserState -> b) -> Parser b
gets f = liftM f getState

type Parser = ParsecT String CSPMParserState IO

prohibitAngles :: Parser a -> Parser a
prohibitAngles p =
  do
    useAngles <- gets canUseAngles
    modifyState (\ s -> s { canUseAngles = False })
    v <- p
    modifyState (\ s -> s { canUseAngles = useAngles })
    return v

allowAngles :: Parser a -> Parser a
allowAngles p =
  do
    useAngles <- gets canUseAngles
    modifyState (\ s -> s { canUseAngles = True })
    v <- p
    modifyState (\ s -> s { canUseAngles = useAngles })
    return v

angles = between (symbol "<") (symbol ">")
braces = between (symbol "{") (symbol "}")
bars = between (symbol "|") (symbol "|")

dotSep1 = \ p -> sepBy1 p dot


cspmLanguage = PT.LanguageDef {
    PT.commentStart = "{-",
    PT.commentEnd = "-}",
    PT.commentLine = "--",
    PT.nestedComments = False,
    PT.identStart = letter,
    PT.identLetter = alphaNum <|> char '\'' <|> char '_',
    PT.opStart = oneOf "+-*/%.^#=!;<>|\\[]",
    PT.opLetter = oneOf "+-*/%.^#=!;<>|\\[]",
    PT.reservedOpNames = [
      -- lambda functions
      "@",
      -- arthmetic
      "+", "-", "*", "/", "%",
      -- sequences
      "^", "#",
      -- boolean statements
      "and", "or", "not", "==", "!=", "<", ">", "<=", ">=",
      -- declaration
      "=", "::"],
    PT.reservedNames =
      ["let", "within", "if", "then", "else",
        "external", "transparent", "channel",
        "true", "false", "assert", "datatype"],
    PT.caseSensitive = True
  }

PT.TokenParser{ PT.parens = parens,
        PT.identifier = identifier,
      PT.natural = natural,
      PT.charLiteral = charLiteral,
      PT.stringLiteral = stringLiteral,
      PT.lexeme = lexeme,
            PT.reservedOp = reservedOp,
      PT.operator = operator,
            PT.reserved = reserved,
      PT.whiteSpace = whiteSpace,
      PT.symbol = symbol,
      PT.comma = comma,
      PT.dot = dot,
      PT.commaSep = commaSep,
      PT.commaSep1 = commaSep1} = PT.makeTokenParser cspmLanguage

annotateTypeParser :: Parser a -> Parser (CSP.Annotated CSP.PType a)
annotateTypeParser = annotateParser CSP.freshPType

annotateTypeSchemeParser :: Parser a -> Parser (CSP.Annotated CSP.PTypeScheme a)
annotateTypeSchemeParser = annotateParser CSP.freshPTypeScheme

annotateNullParser :: Parser a -> Parser (CSP.Annotated () a)
annotateNullParser = annotateParser (return ())

annotateParser :: IO c -> Parser a -> Parser (CSP.Annotated c a)
annotateParser mtyp p =
  do
    typ <- lift mtyp
    startPos <- getPosition
    inner <- p
    endPos <- getPosition
    let srcloc = (CSP.SrcLoc (sourceName startPos)
            (sourceLine startPos) (sourceColumn startPos))
    return $ CSP.Annotated srcloc typ inner

annotateFunctionParser
  :: IO c -> Parser (a -> b) -> Parser (a -> CSP.Annotated c b)
annotateFunctionParser mtyp p =
  do
    typ <- lift mtyp
    startPos <- getPosition
    inner <- p
    endPos <- getPosition
    let srcloc = (CSP.SrcLoc (sourceName startPos)
```

```
            (sourceLine startPos) (sourceColumn startPos))
    return $ CSP.Annotated srcloc typ . inner
annotateFunctionParser2
  :: IO d -> Parser (a -> b -> c) -> Parser (a -> b -> CSP.Annotated d c)
annotateFunctionParser2 mtyp p =
  do
    typ <- lift mtyp
    startPos <- getPosition
    inner <- p
    endPos <- getPosition
    let srcloc = (CSP.SrcLoc (sourceName startPos)
            (sourceLine startPos) (sourceColumn startPos))
    return $ \ e1 e2 -> CSP.Annotated srcloc typ (inner e1 e2)
annotateFunctionParser3
  :: IO e -> Parser (a -> b -> c -> d) -> Parser (a -> b -> c -> CSP.Annotated e d)
annotateFunctionParser3 mtyp p =
  do
    typ <- lift mtyp
    startPos <- getPosition
    inner <- p
    endPos <- getPosition
    let srcloc = (CSP.SrcLoc (sourceName startPos)
            (sourceLine startPos) (sourceColumn startPos))
    return $ \ e1 e2 e3 -> CSP.Annotated srcloc typ (inner e1 e2 e3)

stringParser :: String -> [Operator] -> Tyger [CSP.PModule]
stringParser s ops =
  do
    let opMap = [(n, syntax) | Operator n _ _ syntax <- ops]
    let repOpMap = [(n, syntax) | ReplicatedOperator n _ _ _ syntax <- ops]
    res <- liftIO $ runParserT fileParser
                   (CSPMParserState opMap repOpMap True)
                   "<stdin>" s
    case res of
      Left err -> throwError $ CSPMParseError (show err)
      Right v -> return v

parseCSPMFile :: String -> OpSemDefinition -> Tyger [CSP.PModule]
parseCSPMFile fname opSemDef =
  do
    input <- liftIO $ readFile fname
    let opMap = [(n, syntax) | Operator n _ _ syntax <- operators opSemDef]
    let repOpMap =
      [(n, syntax) | ReplicatedOperator n _ _ _ syntax <- operators opSemDef]
    res <- liftIO $ runParserT fileParser (CSPMParserState opMap repOpMap True)
                   fname input
    case res of
      Left err -> throwError $ CSPMParseError (show err)
      Right v -> return v

fileParser :: Parser [CSP.PModule]
fileParser =
  do
    m <- annotateNullParser $
      do
        whiteSpace
        decls <- many declarationParser
        eof
        return $ CSP.GlobalModule decls
    return [m]

dotAppToList :: CSP.PExp -> [CSP.PExp]
dotAppToList (CSP.Annotated a b exp) =
  let
    dotAppToList' (CSP.DotApp e1 e2) = dotAppToList e1++dotAppToList e2
    dotAppToList' x = [CSP.Annotated a b x]
  in
    dotAppToList' exp

declarationParser :: Parser CSP.PDecl
declarationParser = annotateParser CSP.freshPSymbolTable (
  (try (reserved "external") >> liftM CSP.External (commaSep nameParser))
  <|> (try (reserved "transparent") >> liftM CSP.Transparent (commaSep nameParser))
  <|> do
      try (reserved "channel")
      names <- commaSep1 nameParser
      option (CSP.Channel names []) (
        do
          symbol ":"
          exp <- expressionParser
          return $ CSP.Channel names (dotAppToList exp))
  <|>
    do
      try (reserved "datatype")
      name <- nameParser
      lexeme (string "=")
      es <- sepBy (annotateNullParser $
            do
              name <- nameParser
              ts <- option [] (
                do
                  dot
                  exp <- expressionParser
                  return $ dotAppToList exp)
              return $ CSP.DataTypeClause name ts) (symbol "|")
      return $ CSP.DataType name es
  <|> do
      try (reserved "assert")
      e1 <- expressionParser
      m <- modelParser
      e2 <- expressionParser
      return $ CSP.Assert e1 e2 m
  <|> do
    pat <- try (
        do
          pat <- patternParser
          -- We don't use reservedOp here since if we did it would
          -- not parse p =#s properly since we have said =# is a
          -- proper operator.
          lexeme (string "=")
          return pat)
    exp <- expressionParser
    return (CSP.PatBind pat exp)
  <|> do
    name <- lookAhead identifier
    typ <- typeAnnotationParser name
    matches <- many1 (matchParser name)
    return $ CSP.FunBind (CSP.Name name) matches typ
  <?> "declaration")

typeAnnotationParser :: String -> Parser (Maybe [CSP.AnExp])
typeAnnotationParser n =
  (do
    try (lexeme (string n) >> lookAhead (string "::"))
    reserved "::"
    args <- parens (commaSep expressionParser)
    reserved "->"
    reserved "Proc"
    return $ Just args)
  <|> return Nothing

modelParser :: Parser CSP.Model
modelParser =
  do
    whiteSpace
    char '['
    model <- choice [
      string "T" >> return CSP.Traces,
      try (string "FD" >> return CSP.FailuresDivergences),
      string "F" >> return CSP.Failures]
    char '='
    whiteSpace
    return model

matchParser :: String -> Parser CSP.PMatch
matchParser fname = annotateNullParser $
  do
    -- Test to see if the function name is the same
    try ((lexeme . string $ fname) >> lookAhead (symbol "("))
    groups <-
      many1 (parens (option [] (commaSep patternParser)))
    lexeme (string "=")
    exp <- expressionParser
    return $ CSP.Match groups exp

patternParser :: Parser CSP.PPat
patternParser =
  let
    sequence = liftM CSP.PList . angles . commaSep $ patternParser
    set = liftM CSP.PSet . braces . commaSep $ patternParser
    tuple = liftM CSP.PTuple . parens . commaSep $ patternParser

    operatorTable =
      -- Top binds tightest
      [[E.Infix
        (annotateFunctionParser2 (return ())
          (reservedOp "." >> return CSP.PDotApp))
        E.AssocLeft],
       [E.Infix
        (annotateFunctionParser2 (return ())
          (reservedOp "^" >> return CSP.PConcat))
        E.AssocLeft],
       [E.Infix
        (annotateFunctionParser2 (return ())
```

```
        (reservedOp "@@" >> return CSP.PDoublePattern))
      E.AssocLeft]]

  infixParser = E.buildExpressionParser operatorTable term

  tupleOrPatternParser =
    do
      p1 <- patternParser
      option p1 (annotateNullParser $ do
        comma
        ps <- commaSep1 patternParser
        return $ CSP.PTuple (p1:ps))

  term =
    choice [
      annotateNullParser $ symbol "_" >> return CSP.PWildCard,
      annotateNullParser $ liftM CSP.PLit literalParser,
      annotateNullParser sequence,
      annotateNullParser set,
      parens tupleOrPatternParser,
      annotateNullParser $ liftM CSP.PVar nameParser]
in
  infixParser
  <|> term

nameParser :: Parser CSP.Name
nameParser = liftM CSP.Name identifier

qNameParser :: Parser CSP.QualifiedName
qNameParser = liftM (CSP.UnQual . CSP.Name) identifier

expressionParser :: Parser CSP.PExp
expressionParser =
  let
    sequence =
      angles (prohibitAngles (
        option (CSP.List []) (
          do
            e1 <- expressionParser
            choice [
              do
                lexeme (string "..")
                option (CSP.ListEnumFrom e1) (
                  do
                    ub <- expressionParser
                    return (CSP.ListEnumFromTo e1 ub)),
              do
                es <- option []
                    (comma >> commaSep expressionParser)
                option (CSP.List (e1:es)) (
                  do
                    lexeme (string "|")
                    stmts <- commaSep1 stmtParser
                    return (CSP.ListComp (e1:es) stmts))])))

    set =
      braces (
        bars (
          do
            es <- commaSep expressionParser
            option (CSP.SetEnum es) (try $
              do
                lexeme (string "|")
                stmts <- commaSep1 stmtParser
                return $ CSP.SetEnumComp es stmts))
        <|> option (CSP.Set []) (
          do
            e1 <- expressionParser
            choice [
              do
                lexeme (string "..")
                option (CSP.SetEnumFrom e1) (
                  do
                    ub <- expressionParser
                    return (CSP.SetEnumFromTo e1 ub)),
              do
                es <- option []
                    (comma >> commaSep expressionParser)
                option (CSP.Set (e1:es)) (
                  do
                    lexeme (string "|")
                    stmts <- commaSep1 stmtParser
                    return (CSP.SetComp (e1:es) stmts))]))


    letExp =
      do
        reserved "let"
        ds <- many1 declarationParser
        reserved "within"
        e <- expressionParser
        return $ CSP.Let ds e

    ifExp =
      do
        reserved "if"
        cond <- expressionParser
        reserved "then"
        e1 <- expressionParser
        reserved "else"
        e2 <- expressionParser
        return $ CSP.If cond e1 e2

    builtInOps useAngles =
      if useAngles then
        angleOps++normalOps
      else normalOps
      where
        angleOps =
          [booleanOp ">" CSP.GreaterThan E.AssocNone 4,
          booleanOp ">=" CSP.GreaterThanEq E.AssocNone 4]
        normalOps =
          [functionApp 0,
          binaryOp "." CSP.DotApp E.AssocLeft 3,
          mathsOp "/" CSP.Divide E.AssocLeft 2,
          mathsOp "%" CSP.Mod E.AssocLeft 2,
          mathsOp "*" CSP.Times E.AssocLeft 2,
          mathsOp "+" CSP.Plus E.AssocLeft 2,
          mathsOp "-" CSP.Minus E.AssocLeft 2,
          prefixOp "-" CSP.NegApp 1,
          booleanOp "<" CSP.LessThan E.AssocNone 4,
          booleanOp "<=" CSP.LessThanEq E.AssocNone 4,
          prefixOp "not" (CSP.BooleanUnaryOp CSP.Not) 1,
          booleanOp "and" CSP.And E.AssocNone 6,
          booleanOp "or" CSP.Or E.AssocNone 6,
          booleanOp "==" CSP.Equals E.AssocNone 4,
          booleanOp "!=" CSP.NotEquals E.AssocNone 4,
          binaryOp "^" CSP.Concat E.AssocNone 3,
          prefixOp "#" CSP.ListLength 3]

        mathsOp pat sem assoc int =
          binaryOp pat (CSP.MathsBinaryOp sem) assoc int
        booleanOp pat sem assoc int =
          binaryOp pat (CSP.BooleanBinaryOp sem) assoc int
        binaryOp pat sem assoc int =
          (int, E.Infix
            (annotateFunctionParser2 CSP.freshPType
              (reservedOp pat >> return sem)) assoc)
        prefixOp pat sem int =
          (int, E.Prefix (annotateFunctionParser CSP.freshPType
            (reservedOp pat >> return sem)))
        functionApp int =
          (int, E.Postfix (annotateFunctionParser CSP.freshPType $
            try (do
              args:rest <-
                many (parens(commaSep expressionParser))
              freshPTypes <-
                replicateM (length rest) CSP.freshPType
              -- The below allows us to differentiate between
              -- x = y(a,b) ... and
              --
              -- x = y
              -- (a,b) = ...
              notFollowedBy (reservedOp "=" >> return '=')

              let mkApplication
                    (func @ (CSP.Annotated srcloc _ _)) =
                  foldr (\ (typ, args) f ->
                        CSP.App
                          (CSP.Annotated srcloc typ f)
                          args)
                    (CSP.App func args) (zip freshPTypes rest)
              let mkApplicationOrOperator
                    (f @ (CSP.Annotated _ _ node)) =
                  case node of
                    CSP.UserOperator n [] ->
                      if length rest /= 0 then
                        error "Curried operator"
                      else
                        CSP.UserOperator n args
                    _ -> mkApplication f
              return mkApplication)))

infixOperatorParser =
  do
```

```
        useAngles <- gets canUseAngles
        ops <- gets operatorSyntax
        repOps <- gets repOperatorSyntax
        typ <- CSP.freshPType
        let srcloc = CSP.SrcLoc "" 0 0
        constructParseTable (builtInOps useAngles) ops repOps
            (annotateFunctionParser2 CSP.freshPType (
              return (\ (Name s) es ->
                    CSP.UserOperator (CSP.Name s) es)))
            (annotateFunctionParser3 CSP.freshPType (
              return (\ (Name s) gens args ->
                CSP.ReplicatedUserOperator (CSP.Name s) args gens)))
            term
            (annotateNullParser (
              try (do
                   pat <- patternParser
                   lexeme (string ":")
                   exp <- expressionParser
                   return (CSP.Generator pat exp)
              )
              <|> liftM CSP.Qualifier expressionParser))

    lambdaFunction =
      do
        symbol "\\"
        pat <- patternParser
        reservedOp "@"
        exp <- expressionParser
        return (CSP.Lambda pat exp)

    tupleOrExpressionParser :: Parser CSP.PExp
    tupleOrExpressionParser =
      do
        e1 <- expressionParser
        option e1 (annotateTypeParser $ do
          comma
          es <- commaSep1 expressionParser
          return $ CSP.Tuple (e1:es))

    variableParser =
      do
        ops <- gets operatorSyntax
        let opNames = map (\ (Name s, _) -> s) ops
        (n @ (CSP.UnQual (CSP.Name s))) <- qNameParser
        if s `elem` opNames then return $ CSP.UserOperator (CSP.Name s) []
          else return $ CSP.Var n

    term :: Parser CSP.PExp
    term =  choice [allowAngles $ parens tupleOrExpressionParser,
            annotateTypeParser letExp,
            annotateTypeParser ifExp,
            annotateTypeParser lambdaFunction,
            annotateTypeParser sequence,
            annotateTypeParser set,
            annotateTypeParser $ liftM CSP.Lit literalParser,
            annotateTypeParser variableParser]
    in
      try (infixOperatorParser) <|> term

stmtParser :: Parser CSP.PStmt
stmtParser = annotateNullParser (
  try (do
       pat <- patternParser
       lexeme (string "<-")
       exp <- expressionParser
       return (CSP.Generator pat exp)
  )
  <|> liftM CSP.Qualifier expressionParser)

literalParser :: Parser CSP.Literal
literalParser =
  liftM CSP.Int natural
  <|> (try (reserved "true") >> (return $ CSP.Bool True))
  <|> (try (reserved "false") >> (return $ CSP.Bool False))
```

C.3.3. *CSPMPrettyPrinter.hs.*

```
{-# LANGUAGE FlexibleInstances #-}
module CSPMPrettyPrinter where
import CSPMTypeChecker.TCBuiltInFunctions
import qualified CSPMDataStructures as CSPM
import Text.PrettyPrint.HughesPJ

class PrettyPrintable a where
  prettyPrint :: a -> Doc

-- ************************************************************************
-- Extensions
-- ************************************************************************
angles :: Doc -> Doc
angles d = char '<' <> d <> char '>'

bars :: Doc -> Doc
bars d = char '|' <> d <> char '|'

list :: [Doc] -> Doc
list docs =
  fsep (punctuate (text ",") docs)

tabWidth = 4
tabindent = nest tabWidth

instance (PrettyPrintable b) => PrettyPrintable (CSPM.Annotated a b) where
  prettyPrint (CSPM.Annotated loc typ inner) = prettyPrint inner

-- ************************************************************************
-- Names
-- ************************************************************************
instance PrettyPrintable CSPM.Name where
  prettyPrint (CSPM.Name "Events") = text "UserEvents"
  prettyPrint (CSPM.Name "WrapThread") = text "WrapThread"
  prettyPrint (CSPM.Name "ProcArgs") = text "ProcArgs"
  prettyPrint (CSPM.Name "GetProc") = text "GetProc"
  prettyPrint (CSPM.Name "CallProc") = text "CallProc"
  prettyPrint (CSPM.Name s) =
    if s `elem` builtInNames then text s else text (s++"'")

instance PrettyPrintable CSPM.QualifiedName where
  prettyPrint (CSPM.UnQual name) = prettyPrint name

-- ************************************************************************
-- Modules
-- ************************************************************************
instance PrettyPrintable [CSPM.Module] where
  prettyPrint = vcat . map prettyPrint

instance PrettyPrintable CSPM.Module where
  prettyPrint (CSPM.GlobalModule decls) =
    vcat (punctuate (char '\n') (map prettyPrint decls))

-- ************************************************************************
-- Declarations
-- ************************************************************************
instance PrettyPrintable CSPM.Decl where
  prettyPrint (CSPM.FunBind name matches _) =
      vcat (map (\ (CSPM.Annotated _ _ m) -> ppMatch m) matches)
    where
      ppGroup ps = parens (list (map prettyPrint ps))
      ppMatch (CSPM.Match groups exp) =
        hang ((prettyPrint name <> hcat (map ppGroup groups))
           <+> equals)
          tabWidth (prettyPrint exp)
  prettyPrint (CSPM.PatBind pat exp) =
    hang (prettyPrint pat <+> equals)
      tabWidth (prettyPrint exp)
  prettyPrint (CSPM.Channel ns es) =
    text "channel" <+> list (map prettyPrint ns)
    <+>
    if length es == 0 then empty
    else text ":" <+> fsep (punctuate (text ".") (map prettyPrint es))

  prettyPrint (CSPM.External ns) =
    text "external" <+> list (map prettyPrint ns)
  prettyPrint (CSPM.Transparent ns) =
    text "transparent" <+> list (map prettyPrint ns)
  prettyPrint (CSPM.DataType n dtcs) =
    text "datatype" <+> prettyPrint n <+> text "="
      <+> fsep (punctuate (text "|") (map prettyPrint dtcs))
  prettyPrint (CSPM.Assert e1 e2 m) =
    text "assert" <+> prettyPrint e1 <+> prettyPrint m <+> prettyPrint e2
instance PrettyPrintable CSPM.Model where
  prettyPrint (CSPM.Traces) = text "[T="
  prettyPrint (CSPM.Failures) = text "[F="
  prettyPrint (CSPM.FailuresDivergences) = text "[FD="

instance PrettyPrintable CSPM.DataTypeClause where
  prettyPrint (CSPM.DataTypeClause n es) =
    hcat (punctuate (text ".") (prettyPrint n:(map prettyPrint es)))

instance PrettyPrintable CSPM.Pat where
  prettyPrint (CSPM.PConcat p1 p2) =
    prettyPrint p1 <+> text "^" <+> prettyPrint p2
  prettyPrint (CSPM.PDotApp p1 p2) =
```

```
    prettyPrint p1 <> text "." <> prettyPrint p2
  prettyPrint (CSPM.PDoublePattern p1 p2) =
    prettyPrint p1 <+> text "@@" <+> prettyPrint p2
  prettyPrint (CSPM.PList patterns) =
    angles (list (map prettyPrint patterns))
  prettyPrint (CSPM.PLit lit) = prettyPrint lit
  prettyPrint (CSPM.PSet patterns) =
    braces (list (map prettyPrint patterns))
  prettyPrint (CSPM.PParen pattern) =
    parens (prettyPrint pattern)
  prettyPrint (CSPM.PTuple patterns) =
    parens (list (map prettyPrint patterns))
  prettyPrint (CSPM.PVar name) = prettyPrint name
  prettyPrint (CSPM.PWildCard) = char '_'

-- ************************************************************************
-- Expressions
-- ************************************************************************
instance PrettyPrintable CSPM.BinaryBooleanOp where
  prettyPrint CSPM.And = text "and"
  prettyPrint CSPM.Or = text "or"
  prettyPrint CSPM.Equals = text "=="
  prettyPrint CSPM.NotEquals = text "!="
  prettyPrint CSPM.GreaterThan = text ">"
  prettyPrint CSPM.LessThan = text "<"
  prettyPrint CSPM.LessThanEq = text "<="
  prettyPrint CSPM.GreaterThanEq = text ">="

instance PrettyPrintable CSPM.BooleanUnaryOp where
  prettyPrint CSPM.Not = text "not"

instance PrettyPrintable CSPM.MathsOp where
  prettyPrint CSPM.Divide = text "/"
  prettyPrint CSPM.Minus = text "-"
  prettyPrint CSPM.Mod = text "%"
  prettyPrint CSPM.Plus = text "+"
  prettyPrint CSPM.Times = text "*"

class Precedence a where
  precedence :: a -> Int
minPrecedence = -1

instance Precedence CSPM.BinaryBooleanOp where
  precedence CSPM.And = 6
  precedence CSPM.Or = 6
  precedence CSPM.Equals = 4
  precedence CSPM.NotEquals = 4
  precedence CSPM.GreaterThan = 4
  precedence CSPM.LessThan = 4
  precedence CSPM.LessThanEq = 4
  precedence CSPM.GreaterThanEq = 4
instance Precedence CSPM.BooleanUnaryOp where
  precedence CSPM.Not = 1
instance Precedence CSPM.MathsOp where
  precedence CSPM.Divide = 2
  precedence CSPM.Mod = 2
  precedence CSPM.Times = 2
  precedence CSPM.Plus = 3
  precedence CSPM.Minus = 3
instance Precedence CSPM.Exp where
  precedence (CSPM.App _ _) = 0
  precedence (CSPM.BooleanBinaryOp op _ _) = precedence op
  precedence (CSPM.BooleanUnaryOp op _) = precedence op
  precedence (CSPM.Concat _ _) = 3
--  precedence (CSPM.DotApp _ _) = 3
  precedence (CSPM.ListLength _) = 3
  precedence (CSPM.MathsBinaryOp op _ _) = precedence op
  precedence (CSPM.NegApp _) = 1
  precedence (CSPM.ReplicatedUserOperator _ _ _) = 0
  precedence (CSPM.UserOperator _ _) = 0
  precedence _ = minPrecedence

instance (Precedence b) => Precedence (CSPM.Annotated a b) where
  precedence (CSPM.Annotated _ _ inner) = precedence inner

prettyPrintPrec :: (Precedence a, Precedence b, PrettyPrintable b) => a -> b -> Doc
prettyPrintPrec old new =
  if precedence old < precedence new then parens (prettyPrint new)
  else prettyPrint new

instance PrettyPrintable CSPM.Exp where
  prettyPrint (CSPM.App e1 args) =
    prettyPrint e1 <> parens (list (map prettyPrint args))
  prettyPrint (CSPM.BooleanBinaryOp op e1 e2) =
    prettyPrintPrec op e1 <+> prettyPrint op <+> prettyPrintPrec op e2
  prettyPrint (CSPM.BooleanUnaryOp op e1) =
    prettyPrint op <+> prettyPrintPrec op e1
  prettyPrint (CSPM.Concat e1 e2) =
    prettyPrint e1 <> text "^" <> prettyPrint e2
  prettyPrint (CSPM.DotApp e1 e2) =
    prettyPrint e1 <> text "." <> prettyPrint e2
  prettyPrint (CSPM.If e1 e2 e3) =
    hang (text "if" <+> prettyPrint e1 <+> text "then")
      tabWidth (prettyPrint e2)
    $$
    hang (text "else")
      tabWidth (prettyPrint e3)
  prettyPrint (CSPM.Let decls exp) =
    text "let" $+$
    tabindent (vcat ((map prettyPrint decls))) $+$
    text "within"  $+$
    tabindent (prettyPrint exp)
  prettyPrint (CSPM.ListLength exp) =
    char '#' <> prettyPrint exp
  prettyPrint (CSPM.List exps) =
    angles (list (map prettyPrint exps))
  prettyPrint (CSPM.ListComp exps stmts) =
    angles (
      list (map prettyPrint exps)
    <+> char '|'
    <+> list (map prettyPrint stmts))
  prettyPrint (CSPM.ListEnumFrom lb) =
    angles (prettyPrint lb <> text "...")
  prettyPrint (CSPM.ListEnumFromTo lb ub) =
    angles (prettyPrint lb <> text "..." <> prettyPrint ub)
  prettyPrint (CSPM.Lit lit) = prettyPrint lit
  prettyPrint (CSPM.MathsBinaryOp op e1 e2) =
    prettyPrintPrec op e1 <+> prettyPrint op <+> prettyPrintPrec op e2
  prettyPrint (CSPM.NegApp exp) = char '-' <> prettyPrintPrec (CSPM.NegApp exp) exp
  prettyPrint (CSPM.Paren e) = parens (prettyPrint e)
  prettyPrint (CSPM.ReplicatedUserOperator n exps stmts) =
    prettyPrint n <> parens (list [angles (
        prettyPrint e
      <+> char '|'
      <+> list (map (\ (CSPM.Annotated _ _ stmt) -> ppStmt stmt) stmts))
        | e <- exps])
    where
      ppStmt (CSPM.Generator pat exp) =
        sep [prettyPrint pat,
             text "<-" <+> text "seq" <> parens (prettyPrint exp)]
      ppStmt stmt = prettyPrint stmt

  prettyPrint (CSPM.Set exps) =
    braces (list (map prettyPrint exps))
  prettyPrint (CSPM.SetComp exps stmts) =
    braces (
      list (map prettyPrint exps)
    <+> char '|'
    <+> list (map prettyPrint stmts))
  prettyPrint (CSPM.SetEnum es) =
    braces . bars . list . map prettyPrint $ es
  prettyPrint (CSPM.SetEnumComp es stmts) =
    braces (bars (
      list (map prettyPrint es)
    <+> char '|'
    <+> list (map prettyPrint stmts)))
  prettyPrint (CSPM.SetEnumFrom lb) =
    braces (prettyPrint lb <> text "..")
  prettyPrint (CSPM.SetEnumFromTo lb ub) =
    braces (prettyPrint lb <> text ".." <> prettyPrint ub)
  prettyPrint (CSPM.Tuple exps) = parens (list (map prettyPrint exps))
  prettyPrint (CSPM.UserOperator opName args) =
    prettyPrint opName <>
      if length args == 0 then empty
      else parens (list (map prettyPrint args))
  prettyPrint (CSPM.Var qname) = prettyPrint qname

instance PrettyPrintable CSPM.Stmt where
  prettyPrint (CSPM.Generator pat exp) =
    sep [prettyPrint pat, text "<-" <+> prettyPrint exp]
  prettyPrint (CSPM.Qualifier exp) =
    prettyPrint exp

instance PrettyPrintable CSPM.Literal where
  prettyPrint (CSPM.Int n) = integer n
  prettyPrint (CSPM.Bool True) = text "true"
  prettyPrint (CSPM.Bool False) = text "false"
```

C.3.4. *CSPMRecursionRefactorings.hs.*

```
{-# LANGUAGE QuasiQuotes #-}
```

```
module CSPMRecursionRefactorings where

import Control.Monad.State
import Data.Graph

import CSPMDataStructures
import CSPMTypeChecker.TCCommon
import CSPMTypeChecker.TCDependencies
import CSPMTypeChecker.TCModule
import CSPMTypeChecker.TCMonad
import CSPMParser
import CSPMPrettyPrinter
import List (intersect)
import qualified OpSemDataStructures as OpSem
import OpSemRules
import OpSemParser
import OpSemTypeChecker

import Util

isProcessTypeScheme :: TypeScheme -> Bool
isProcessTypeScheme (ForAll _ t) = isProcessType t
isProcessType :: Type -> Bool
isProcessType (TFunction _ b) = isProcessType b
isProcessType (TTuple ts) = or $ map isProcessType ts
isProcessType TProc = True
isProcessType _ = False

unWrappedName :: Name -> Name
unWrappedName (Name n) = (Name (n++"_UNWRAPPED"))

dataTypeMember :: Name -> AnExp
dataTypeMember (Name n) = an (Var (UnQual (Name ("Proc_"++n))))

an :: b -> Annotated a b
an = Annotated (error "inserted_srcloc") (error "inserted_annotation")

listToDotApp :: [AnExp] -> AnExp
listToDotApp (e1:[]) = e1
listToDotApp (e1:es) = an (DotApp e1 (listToDotApp es))

listToPatDotApp :: [AnPat] -> AnPat
listToPatDotApp (e1:[]) = e1
listToPatDotApp (e1:es) = an (PDotApp e1 (listToPatDotApp es))

declHasBounds :: TCDecl -> Bool
declHasBounds (Annotated _ _ (FunBind _ _ Nothing)) = False
declHasBounds _ = True

computeProcessGraph :: [TCDecl] -> TypeCheckMonad [Name]
computeProcessGraph decls =
  do
    let declMap = zip decls [0..]
    varToDeclIdMap <-
      concatMapM (\ decl ->
        do
          namesBound <- namesBoundByDecl decl
          return [(n, apply declMap decl) | n <- namesBound]) decls

    procNames <- concatMapM (\ decl @ (Annotated _ psymtable _) ->
      do
        symtable <- readPSymbolTable psymtable
        liftM (map fst) (filterM (\ (n,t) ->
          do
            tc <- compressTypeScheme t
            return $ isProcessTypeScheme tc) symtable)
      ) decls

    mapM_ prebindDataType
      [DataType n ms | DataType n ms <- map removeAnnotation decls]

    -- Map from decl id -> [decl id] meaning decl id depends on the list of
    -- ids
    declDeps <- mapM (\ decl ->
        do
          fvsd <- dependencies decl
          let fvs' = intersect fvsd procNames
          return (apply declMap decl, mapPF varToDeclIdMap fvs')
          ) decls

    -- Edge from n -> n' iff n uses n'
    let sccs = stronglyConnComp [(id, id, deps) | (id, deps) <- declDeps]
    let recursiveDeclIds =
      concatMap (\ scc ->
            case scc of
              AcyclicSCC _  -> []
              CyclicSCC vs  -> vs) sccs
    let boundedRecursiveDeclIds =
          [id | id <- recursiveDeclIds, declHasBounds (apply (invert declMap) id)]
    let recursiveNames =
      concatMap (applyRelation (invert varToDeclIdMap)) boundedRecursiveDeclIds
    debugOutput ("Recursive_processes:_"++show recursiveNames)

    return recursiveNames

data TransformMonadState = TransformMonadState {
    recursiveNames :: [[Name]],
    operatorTypeMap :: PartialFunction OpSem.Name [OpSem.Type],
    isRecursive :: Bool,
    recursiveCallsAllowed :: Bool -- rule 0 below
  }
type TransformMonad = StateT TransformMonadState TypeCheckMonad

runTransformMonad :: TransformMonad a -> TypeCheckMonad a
runTransformMonad m =
  liftM fst (runStateT m (TransformMonadState [] [] False True))

getOperatorType :: Name -> TransformMonad [OpSem.Type]
getOperatorType (Name n) =
  do
    m <- gets operatorTypeMap
    return $ apply m (OpSem.Name n)
setOperatorTypeMap :: PartialFunction OpSem.Name [OpSem.Type] -> TransformMonad ()
setOperatorTypeMap typeMap = modify (\ s -> s { operatorTypeMap = typeMap })

isRecursiveName :: Name -> TransformMonad Bool
isRecursiveName n =
  do
    names <- gets recursiveNames
    return $ or (map (elem n) names)
addRecursiveNames :: [Name] -> TransformMonad a -> TransformMonad a
addRecursiveNames ns m =
  do
    modify (\ s -> s { recursiveNames = ns:(recursiveNames s) })
    ret <- m
    modify (\ s -> s { recursiveNames = tail (recursiveNames s) })
    return ret
setIsRecursive :: Bool -> TransformMonad a -> TransformMonad a
setIsRecursive b m =
  do
    oldv <- isCurrentlyRecursive
    modify (\ s -> s { isRecursive = b })
    ret <- m
    modify (\ s -> s { isRecursive = oldv })
    return ret
isCurrentlyRecursive :: TransformMonad Bool
isCurrentlyRecursive = gets isRecursive

allowRecursiveCalls :: TransformMonad Bool
allowRecursiveCalls = gets recursiveCallsAllowed
setAllowRecursiveCalls :: Bool -> TransformMonad a -> TransformMonad a
setAllowRecursiveCalls b m =
  do
    oldv <- allowRecursiveCalls
    modify (\ s -> s { recursiveCallsAllowed = b })
    ret <- m
    modify (\ s -> s { recursiveCallsAllowed = oldv })
    return ret


class Transformable a where
  transform :: a -> TransformMonad a
instance Transformable a => Transformable [a] where
  transform = mapM transform
instance Transformable b => Transformable (Annotated a b) where
  transform (Annotated srcloc b inner) =
    do
      inner' <- transform inner
      return $ Annotated srcloc b inner'

transformModules :: OpSem.OpSemDefinition -> [TCModule] -> TransformMonad [TCModule]
transformModules opSemDef [Annotated b c (GlobalModule ds)] =
  do
    let ops = OpSem.operators opSemDef
    let operatorTypeMap =
      [(OpSem.opFriendlyName op, map snd (OpSem.opArgs op)) | op <- ops]
    setOperatorTypeMap operatorTypeMap
    ds' <- transformDecls ds
    return [Annotated b c (GlobalModule ds')]

transformDecls :: [TCDecl] -> TransformMonad [TCDecl]
transformDecls ds =
  do
    ns <- lift (computeProcessGraph ds)
    addRecursiveNames ns (do
        declRecTypes <- mapM (\ (Annotated n t d) ->
```

```
            do
              (ds', rt) <- transformDecl d
              return ([Annotated n t d' | d' <- ds'], rt)) ds
        let (decls', recTypes) = unzip declRecTypes
        let hasRecTypes = length [0 | Just _ <- recTypes] /= 0
        let procDataType =
          if hasRecTypes then
            an $ DataType (Name "ProcArgs")
                 [an $ DataTypeClause (Name ("Proc_"++s)) es
                   | Just (Name s, es, isPat) <- recTypes]
          else
            an $ PatBind (an $ PVar (Name "ProcArgs")) (an $ Set [])
        let getProc = an $
          FunBind (Name "GetProc")
            [let
              argNames = ["arg_"++show  i | i <- [1..(length es)]]
              argPats = [an $ PVar (Name i) | i <- argNames]
              argExps = [an $ Var (UnQual (Name i)) | i <- argNames]
              arg = (an $ (PVar (Name ("Proc_"++s)))):argPats
              exp = if isPat then Var (UnQual (Name $ s++"_UNWRAPPED"))
                    else App (an $ Var (UnQual
                               (Name $ s++"_UNWRAPPED")))
                             argExps
            in
              an $ Match [[listToPatDotApp arg]]
                     (an exp)
              | Just (Name s, es, isPat) <- recTypes]
            Nothing
        if hasRecTypes then
            return $ procDataType:getProc:(concat decls')
        else return $ procDataType:concat decls')

makeWrapThread :: Name -> [AnExp] -> AnExp
makeWrapThread n args =
  an (App (an (Var (UnQual (Name "WrapThread"))))
    [listToDotApp $ (dataTypeMember n):args])

transformDecl :: Decl -> TransformMonad ([Decl], Maybe (Name, [AnExp], Bool))
transformDecl (FunBind n ms annotTyp) =
  do
    b <- isRecursiveName n
    if b then setIsRecursive True (
      do
        ms' <- transform ms
        let args = head [ps | Match [ps] e <- map removeAnnotation ms]
        let argCount = length args
        let argList = map (\ s -> "arg_"++show s) [1..argCount]
        let decls = [FunBind (unWrappedName n) ms' annotTyp,
            FunBind n [an $
              Match
                [[an $ PVar (Name s) | s <- argList]]
                (makeWrapThread n
                  (map (\ s -> an (Var (UnQual (Name s))))
                    argList))] annotTyp]
        case annotTyp of
          Just es -> return (decls, Just (n, es, False))
          Nothing -> return (decls, Nothing))
      else setIsRecursive False (
        do
          ms' <- transform ms
          return ([FunBind n ms' annotTyp], Nothing))
transformDecl (PatBind (p @ (Annotated _ _ (PVar n))) e) =
  do
    b <- isRecursiveName n
    if b then setIsRecursive True (
      do
        e' <- transform e
        let decls = [PatBind (an $ PVar (unWrappedName n)) e',
              PatBind p (makeWrapThread n [])]
        return (decls, Just (n, [], True)))
      else setIsRecursive False (
        do
          e' <- transform e
          return ([PatBind p e'], Nothing))
transformDecl (PatBind pat e) = error ""
transformDecl (Channel ns es) = return ([Channel ns es], Nothing)
transformDecl (DataType n dtcs) = return ([DataType n dtcs], Nothing)
transformDecl (External ns) = return ([External ns], Nothing)
transformDecl (Transparent ns) = return ([Transparent ns], Nothing)
transformDecl (Assert e1 e2 m) = return ([Assert e1 e2 m], Nothing)

instance Transformable Match where
  transform (Match ps e) =
    do
      e' <- transform e
      return $ Match ps e'
instance Transformable Exp where
transform (App e es) =
  do
    es' <- transform es
    case removeAnnotation e of
      Var (UnQual n) -> do
        isRecursive <- isRecursiveName n
        isCurrentlyRecursive <- isCurrentlyRecursive
        return (
          if isRecursive && isCurrentlyRecursive then
            App (an (Var (UnQual (Name "CallProc"))))
              [listToDotApp $ (dataTypeMember n):es']
          else App (an (Var (UnQual n))) es')
transform (BooleanBinaryOp op e1 e2) =
  do
    e1' <- transform e1
    e2' <- transform e2
    return $ BooleanBinaryOp op e1' e2'
transform (BooleanUnaryOp op e) =
  do
    e' <- transform e
    return $ BooleanUnaryOp op e'
transform (Concat e1 e2) =
  do
    e1' <- transform e1
    e2' <- transform e2
    return $ Concat e1' e2'
transform (DotApp e1 e2) =
  do
    e1' <- transform e1
    e2' <- transform e2
    return $ DotApp e1' e2'
transform (If e1 e2 e3) =
  do
    e1' <- transform e1
    e2' <- transform e2
    e3' <- transform e3
    return $ If e1' e2' e3'
transform (Lambda p e) =
  do
    e' <- transform e
    return $ Lambda p e'
transform (Let ds e) =
  do
    namesBound <- lift (concatMapM namesBoundByDecl ds)
    recursiveNames <- lift (computeProcessGraph ds)
    if intersect namesBound recursiveNames == [] then (
      do
        e' <- transform e
        -- We don't transform ds as it does not contain any
        -- recursive terms.
        return $ Let ds e')
      else
        error "let_contains_recursive_names"
transform (Lit lit) = return $ Lit lit
transform (List es) =
  do
    es' <- transform es
    return $ List es'
transform (ListComp es stmts) =
  do
    es' <- transform es
    stmts' <- transform stmts
    return $ ListComp es' stmts'
transform (ListEnumFromTo e1 e2) =
  do
    e1' <- transform e1
    e2' <- transform e2
    return $ ListEnumFromTo e1' e2'
transform (ListLength e) =
  do
    e' <- transform e
    return $ ListLength e'
transform (MathsBinaryOp op e1 e2) =
  do
    e1' <- transform e1
    e2' <- transform e2
    return $ MathsBinaryOp op e1' e2'
transform (NegApp e) =
  do
    e' <- transform e
    return $ NegApp e
transform (Paren e) =
  do
    e' <- transform e
    return $ Paren e'
transform (ReplicatedUserOperator n es stmts) =
  do
```

```
      es' <- transform es
      stmts' <- transform stmts
      return $ ReplicatedUserOperator n es' stmts'
  transform (Set es) =
    do
      es' <- transform es
      return $ Set es'
  transform (SetComp es stmts) =
    do
      es' <- transform es
      stmts' <- transform stmts
      return $ SetComp es' stmts'
  transform (SetEnum es) =
    do
      es' <- transform es
      return $ SetEnum es'
  transform (SetEnumFromTo e1 e2) =
    do
      e1' <- transform e1
      e2' <- transform e2
      return $ SetEnumFromTo e1' e2'
  transform (Tuple es) =
    do
      es' <- transform es
      return $ Tuple es'
  transform (Var (UnQual n)) =
    do
      isRecursive <- isRecursiveName n
      isCurrentlyRecursive <- isCurrentlyRecursive
      return (
        if isRecursive && isCurrentlyRecursive then
          App (an (Var (UnQual (Name "CallProc")))) [dataTypeMember n]
        else Var (UnQual n))
  transform (UserOperator opname es) =
    do
      opType <- getOperatorType opname
      es' <- zipWithM transformOpArg opType es
      return $ UserOperator opname es'
    where
      transformOpArg (OpSem.TOnProcess OpSem.InfinitelyRecursive) e =
        setIsRecursive False (transform e)
      transformOpArg (OpSem.TOffProcess OpSem.InfinitelyRecursive) e =
        setIsRecursive False (transform e)
      transformOpArg _ e = transform e

instance Transformable Stmt where
  transform (Generator p e) =
    do
      e' <- transform e
      return $ Generator p e'
  transform (Qualifier e) =
    do
      e' <- transform e
      return $ Qualifier e'
```

C.4. **CSPM Type Checker.**

C.4.1. *CSPMTypeChecker/TCBuiltInFunctions.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
module CSPMTypeChecker.TCBuiltInFunctions(
  injectBuiltInFunctions, externalFunctions, transparentFunctions,
  builtInNames
) where

import CSPMDataStructures
import CSPMTypeChecker.TCMonad
import qualified OpSemDataStructures as OpSem
import Util

-- ***********************************************************************
-- Built in function types
-- ***********************************************************************
sets :: [Type -> (String, [Type], Type)]
sets =
  let
    cspm_union fv = ("union", [TSet fv, TSet fv], TSet fv)
    cspm_inter fv = ("inter", [TSet fv, TSet fv], TSet fv)
    cspm_diff fv = ("diff", [TSet fv, TSet fv], TSet fv)
    cspm_Union fv = ("Union", [TSet (TSet fv)], TSet fv)
    cspm_Inter fv = ("Inter", [TSet (TSet fv)], TSet fv)
    cspm_member fv = ("member", [fv, TSet fv], TBool)
    cspm_card fv = ("card", [TSet fv], TInt)
    cspm_empty fv = ("empty", [TSet fv], TBool)
    cspm_set fv = ("set", [TSeq fv], TSet fv)
    cspm_Set fv = ("Set", [TSet fv], TSet (TSet fv))
    cspm_Seq fv = ("Seq", [TSet fv], TSet (TSeq fv))
    cspm_seq fv = ("seq", [TSet fv], TSeq fv)
  in
    [cspm_union, cspm_inter, cspm_diff, cspm_Union, cspm_Inter,
        cspm_member, cspm_card, cspm_empty, cspm_set, cspm_Set,
        cspm_Seq, cspm_seq]
seqs :: [Type -> (String, [Type], Type)]
seqs =
  let
    cspm_length fv = ("length", [TSeq fv], TInt)
    cspm_null fv = ("null", [TSeq fv], TBool)
    cspm_head fv = ("head", [TSeq fv], fv)
    cspm_tail fv = ("tail", [TSeq fv], TSeq fv)
    cspm_concat fv = ("concat", [TSeq (TSeq fv)], TSeq fv)
    cspm_elem fv = ("elem", [fv, TSeq fv], TBool)
  in
    [cspm_length, cspm_null, cspm_head, cspm_tail, cspm_concat,
        cspm_elem]

typeConstructors :: [(String, Type)]
typeConstructors =
  let
    cspm_Int = ("Int", TSet TInt)
    cspm_Bool = ("Bool", TSet TBool)
    cspm_Proc = ("Proc", TSet TProc)
    cspm_Events = ("Events", TSet (TChannel TListEnd))
  in
    [cspm_Int, cspm_Bool, cspm_Proc, cspm_Events]

injectBuiltInFunctions :: TypeCheckMonad ()
injectBuiltInFunctions =
  let
    mkFuncType cs func =
      do
        fv @ (TVar (TypeVarRef tv _ _)) <- freshTypeVarWithConstraints cs
        let (n, args, ret) = func fv
        let t = ForAll [(tv, cs)] (TFunction args ret)
        setType (Name n) t
    mkPatternType func =
      do
        let (n, t) = func
        setType (Name n) (ForAll [] t)
  in do
    mapM_ (mkFuncType []) seqs
    mapM_ (mkFuncType [Eq]) sets
    mapM_ mkPatternType typeConstructors

externalFunctions :: PartialFunction String Type
externalFunctions = []

transparentFunctions :: PartialFunction String Type
transparentFunctions = [
  ("diamond", TFunction [TProc] TProc),
  ("normal", TFunction [TProc] TProc),
  ("sbisim", TFunction [TProc] TProc),
  ("tau_loop_factor", TFunction [TProc] TProc),
  ("model_compress", TFunction [TProc] TProc),
  ("explicate", TFunction [TProc] TProc)
  ]

builtInNames :: [String]
builtInNames =
    map fst externalFunctions
    ++ map fst transparentFunctions
    ++ map fst typeConstructors
    ++ map extract seqs
    ++ map extract sets
  where
    extract f =
      let (a,_,_) = f (error "")
      in a
```

C.4.2. *CSPMTypeChecker/TCCommon.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies #-}
module CSPMTypeChecker.TCCommon where

import CSPMDataStructures
import CSPMTypeChecker.TCBuiltInFunctions
import CSPMTypeChecker.TCMonad
import CSPMTypeChecker.TCUnification
import qualified OpSemDataStructures as OpSem
import Util
```

```
typeCheckWrapper :: TypeCheckable a b =>
  a -> PartialFunction OpSem.Name [OpSem.Type] -> TypeCheckMonad b
typeCheckWrapper tc ops =
    do
      setUserOperators convertedops
      injectBuiltInFunctions
      -- We add an extra scope layer here to allow the built in functions
      -- to be overloaded.
      local [] (typeCheck tc)
  where
    convertedops =
      [(opSemOperatorNameToCSPMName n, map opSemTypeToCSPMType ts)
        | (n,ts) <- ops]

-- ************************************************************************
-- Helper methods
-- ************************************************************************
ensureIsList :: Type -> TypeCheckMonad Type
ensureIsList typ =
  do
    fv <- freshTypeVar
    unify (TSeq fv) typ

ensureIsSet :: Type -> TypeCheckMonad Type
ensureIsSet typ =
  do
    fv <- freshTypeVar
    ensureHasConstraint Eq fv
    unify (TSet fv) typ

ensureIsBool :: Type -> TypeCheckMonad Type
ensureIsBool typ = unify (TBool) typ

ensureIsInt :: Type -> TypeCheckMonad Type
ensureIsInt typ = unify TInt typ

ensureIsChannel :: Type -> TypeCheckMonad Type
ensureIsChannel t =
  do
    (TVar fv) <- freshTypeVar
    unify t (TChannel (TPolyList fv))

ensureIsEvent :: Type -> TypeCheckMonad Type
ensureIsEvent t = unify t (TChannel TListEnd)

ensureIsDotable :: Type -> TypeCheckMonad Type
ensureIsDotable t =
  do
    fv1 <- freshTypeVar
    fv2 <- freshTypeVar
    unify t (TDotable fv1 fv2)

ensureHasConstraint :: Constraint -> Type -> TypeCheckMonad Type
ensureHasConstraint c t =
  do
    fv1 <- freshTypeVarWithConstraints [c]
    unify fv1 t

ensureIsProcess :: Type -> TypeCheckMonad Type
ensureIsProcess t =
  do
    unify TProc t

-- See http://www.haskell.org/haskellwiki/Functional_dependencies for more
-- descriptions on a -> b but it esentially says that a determins the type
-- b (i.e. we can't have two instances that match on a but differ on b)
class TypeCheckable a b | a -> b where
  typeCheck :: a -> TypeCheckMonad b
  typeCheck a = typeCheck' a
    `catchError` (\ e -> throwError $ errorConstructor a e)

  errorConstructor :: a -> (TypeCheckError -> TypeCheckError)
  typeCheck' :: a -> TypeCheckMonad b

instance TypeCheckable Literal Type where
  errorConstructor = error "No_error_should_ever_occur_in_a_literal"
  typeCheck' (Int n) = return TInt
  typeCheck' (Bool b) = return TBool
```

C.4.3. *CSPMTypeChecker/TCDecl.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, TypeSynonymInstances, FlexibleInstances #-}
module CSPMTypeChecker.TCDecl (typeCheckDecls) where

import Data.Graph
import List (nub, intersect)

import CSPMDataStructures
import CSPMTypeChecker.TCBuiltInFunctions
import CSPMTypeChecker.TCCommon
import CSPMTypeChecker.TCDependencies
import CSPMTypeChecker.TCExpr
import CSPMTypeChecker.TCMonad
import CSPMTypeChecker.TCPat
import CSPMTypeChecker.TCUnification

import Util

-- Type check a list of possibly mutually recursive functions
typeCheckDecls :: [PDecl] -> TypeCheckMonad ()
typeCheckDecls decls =
  do
    let declMap = zip decls [0..]
    boundVarToDeclIdMap <-
      concatMapM (\ decl ->
        do
          namesBound <- namesBoundByDecl decl
          return [(n, apply declMap decl) | n <- namesBound]) decls
    let fvs = map fst boundVarToDeclIdMap

    errorIfFalse (noDups fvs) (DuplicatedDefinitions fvs)

    -- Pre-analysis phase:
    -- We prebind the datatypes so that we can detect when something is
    -- a free variable in a pattern and when it is bound
    mapM_ prebindDataType [DataType n ms | DataType n ms <- map removeAnnotation decls]

    -- Map from decl id -> [decl id] meaning decl id depends on the list of
    -- ids
    declDeps <- mapM (\ decl ->
        do
          fvsd <- dependencies decl
          let fvs' = intersect fvsd fvs
          return (apply declMap decl, mapPF boundVarToDeclIdMap fvs')
          ) decls

    -- Edge from n -> n' iff n uses n'
    let sccs = map flattenSCC
          (stronglyConnComp [(id, id, deps) | (id, deps) <- declDeps])
    let typeInferenceGroups = map (mapPF (invert declMap)) sccs

    debugOutput ("Type_check_order:_"
        ++show (map (safeMapPF (invert boundVarToDeclIdMap)) sccs))

    mapM_ typeCheckMutualyRecursiveGroup typeInferenceGroups

typeCheckMutualyRecursiveGroup :: [PDecl] -> TypeCheckMonad ()
typeCheckMutualyRecursiveGroup ds =
  do
    -- We only create variables for non datatype declarations as the others
    -- were prebound in prebindDataType
    fvs <- liftM nub (concatMapM namesBoundByDecl
          (filter (not . isDataTypeDecl) ds))
    ftvs <- replicateM (length fvs) freshTypeVar
    zipWithM setType fvs (map (ForAll []) ftvs)

    -- The list of all variables bound by these declaration
    fvs <- liftM nub (concatMapM namesBoundByDecl ds)

    -- Type check each declaration then generalise the types
    nts <- generaliseGroup fvs (map typeCheck ds)
    -- Add the type of each declaration (if one exists to each declaration)
    zipWithM annotate nts ds

    -- Compress all the types we have inferred here
    -- (They should never be touched again)
    mapM_ (\ n ->
      do
        t <- getType n
        t' <- compressTypeScheme t
        setType n t') fvs
  where
    isDataTypeDecl (Annotated _ _ inner) = isDataTypeDecl' inner
    isDataTypeDecl' (DataType _ _) = True
    isDataTypeDecl' _ = False

    annotate nts (Annotated _ psymbtable _) =
      setPSymbolTable psymbtable nts

evaluateTypeExpression :: Type -> Type -> TypeCheckMonad ()
evaluateTypeExpression t fv =
  (do
    unify t (TSet fv)
```

```
    `catchError` (\ e ->
      case t of
        -- Could be a tuple of sets (TTuple [TSet t1 ,...] = TSet (TTuple [t1 ,...]))
        -- according to Bill's book P529
        TTuple ts ->
          do
            fvs <- replicateM (length ts) freshTypeVar
            zipWithM evaluateTypeExpression ts fvs
            unify fv (TSet (TTuple fvs))))
  >> return ()

instance TypeCheckable PDecl [(Name, Type)] where
  errorConstructor = ErrorWithDecl
  -- We perform type annotation in typeCheckMutualyRecursiveGroup since
  -- this is the first time we know the TypeScheme.
  typeCheck' = typeCheck' . removeAnnotation

instance TypeCheckable Decl [(Name, Type)] where
  errorConstructor = error "Decl_()_error_constructor_called"
  typeCheck' (FunBind n ms usertyp) =
    do
      ts <- mapM typeCheck ms
      ForAll [] t <- getType n
      (t' @ (TFunction tsargs _)) <- unifyAll (t:ts)
      case usertyp of
        Nothing -> return ()
        Just es -> do
              ts <- mapM typeCheck es
              mapM ensureIsSet ts
              zipWithM unify (map TSet tsargs) ts
              return ()
      return [(n, t')]
  typeCheck' (PatBind pat exp) =
    do
      tpat <- typeCheck pat
      fvs <- freeVars pat
      -- Allow the pattern to be recursive
      texp <- local fvs (typeCheck exp)
      unify texp tpat
      ns <- namesBoundByDecl' (PatBind pat exp)
      ts <- mapM getType ns
      return $ zip ns [t | ForAll _ t <- ts]
  typeCheck' (Channel ns es) =
    do
      ts <- mapM typeCheck es
      fvs <- replicateM (length ts) freshTypeVar
      -- Make sure everything is a set
      zipWithM evaluateTypeExpression ts fvs
      -- Channels require that each component is comparable
      mapM (ensureHasConstraint Eq) fvs
      let t = TChannel $ foldr TList TListEnd fvs
      mapM_ (\ n -> setType n (ForAll [] t)) ns
      return [(n,t) | n <- ns]
  -- This clause is relies on the fact that prebindDataType has been called first
  -- any changes to that will require a change here
  typeCheck' (DataType n clauses) =
    do
      nts <- mapM (\ clause ->
        do
          (n', ts) <- typeCheck clause
          -- We already have a variable for n' (introduced in prebindDataType)
          ForAll [] t <- getType n'
          unify t (TDatatypeClause n ts)
          return (n', t)
          ) clauses
      -- We have already set the type of n in prebindDataType
      ForAll [] t <- getType n
      return $ (n,t):nts
  typeCheck' (Assert e1 e2 m) =
    do
      t1 <- typeCheck e1
      ensureIsProcess t1
      t2 <- typeCheck e2
      ensureIsProcess t2
      return []
  typeCheck' (Transparent ns) =
    do
      let ts = map (\ (Name n) ->
                (Name n, ForAll [] (apply transparentFunctions n))) ns
      mapM_ (\ (n, t) -> setType n t) ts
      return []
  typeCheck' (External ns) =
    do
      let ts = map (\ (Name n) ->
                (Name n, ForAll [] (apply externalFunctions n))) ns
      mapM_ (\ (n, t) -> setType n t) ts
      return []

instance TypeCheckable PDataTypeClause (Name, [Type]) where
  errorConstructor = ErrorWithDataTypeClause
  typeCheck' (Annotated srcloc typ inner) =
    do
      t' <- typeCheck' inner
--      setPType typ t'
      return t'
instance TypeCheckable DataTypeClause (Name, [Type]) where
  errorConstructor = error "Error:_DataTypeClause_error_constructor_called"
  typeCheck' (DataTypeClause n' es) =
    do
      ts <- mapM typeCheck es
      fvs <- replicateM (length ts) freshTypeVar
      -- Make sure everything is a set
      zipWithM evaluateTypeExpression ts fvs
      return (n', fvs)

instance TypeCheckable PMatch Type where
  errorConstructor = ErrorWithMatch
  typeCheck' (Annotated srcloc typ inner) =
    do
      t' <- typeCheck' inner
      return t'
instance TypeCheckable Match Type where
  errorConstructor = error "Error:_match_error_constructor_called"
  typeCheck' (Match groups exp) =
    do
      fvs <- liftM concat (mapM freeVars groups)
      local fvs (
        do
          tr <- typeCheck exp
          tgroups <- mapM (\ pats -> mapM typeCheck pats) groups
          return $ foldr (\ targs tr -> TFunction targs tr) tr tgroups)
```

C.4.4. *CSPMTypeChecker/TCDecl.hs-boot.*

```
module CSPMTypeChecker.TCDecl where

import CSPMDataStructures
import CSPMTypeChecker.TCMonad

typeCheckDecls :: [PDecl] -> TypeCheckMonad ()
```

C.4.5. *CSPMTypeChecker/TCDependencies.hs.*

```
module CSPMTypeChecker.TCDependencies
  (Dependencies, dependencies, namesBoundByDecl, namesBoundByDecl',
  FreeVars, freeVars, prebindDataType) where

import List (nub, (\\))

import CSPMDataStructures
import CSPMTypeChecker.TCMonad
import Util


-- This method heavily affects the DataType clause of typeCheckDecl.
-- If any changes are made here changes will need to be made to typeCheckDecl
-- too
prebindDataType :: Decl -> TypeCheckMonad ()
prebindDataType (DataType n cs) =
  let
    prebindDataTypeClause (DataTypeClause n' es) =
      do
        fvs <- replicateM (length es) freshTypeVar
        setType n' (ForAll [] (TDatatypeClause n fvs))
    clauseNames = [n' | DataTypeClause n' es <- map removeAnnotation cs]
  in do
    errorIfFalse (noDups clauseNames) (DuplicatedDefinitions clauseNames)
    mapM_ (prebindDataTypeClause . removeAnnotation) cs
    -- n is the set of all TDatatypeClause
    setType n (ForAll [] (TSet (TDatatypeClause n [])))


class Dependencies a where
  dependencies :: a -> TypeCheckMonad [Name]
  dependencies xs = liftM nub (dependencies' xs)
  dependencies' :: a -> TypeCheckMonad [Name]

instance Dependencies a => Dependencies [a] where
  dependencies' xs = concatMapM dependencies xs
instance Dependencies a => Dependencies (Maybe a) where
  dependencies' (Just x) = dependencies' x
```

```
  dependencies' Nothing = return []
instance Dependencies a => Dependencies (Annotated b a) where
  dependencies' (Annotated _ _ inner) = dependencies inner

instance Dependencies Pat where
  dependencies' (PVar n) =
    do
      res <- safeGetType n
      case res of
        Just (ForAll _ t) ->
          case t of
            -- See typeCheck (PVar) for a discussion of why
            -- we only do this
            TDatatypeClause n ts  -> return [n]
            _                     -> return []
        Nothing -> return []  -- var is not bound
  dependencies' (PConcat p1 p2) =
    do
      fvs1 <- dependencies' p1
      fvs2 <- dependencies' p2
      return $ fvs1++fvs2
  dependencies' (PDotApp p1 p2) = dependencies' [p1,p2]
  dependencies' (PList ps) = dependencies' ps
  dependencies' (PWildCard) = return []
  dependencies' (PTuple ps) = dependencies' ps
  dependencies' (PSet ps) = dependencies' ps
  dependencies' (PParen p) = dependencies' p
  dependencies' (PLit l) = return []
  dependencies' (PDoublePattern p1 p2) =
    do
      fvs1 <- dependencies' p1
      fvs2 <- dependencies' p2
      return $ fvs1++fvs2

instance Dependencies Exp where
  dependencies' (App e es) = dependencies' (e:es)
  dependencies' (BooleanBinaryOp _ e1 e2) = dependencies' [e1,e2]
  dependencies' (BooleanUnaryOp _ e) = dependencies' e
  dependencies' (Concat e1 e2) = dependencies' [e1,e2]
  dependencies' (DotApp e1 e2) = dependencies' [e1,e2]
  dependencies' (If e1 e2 e3) = dependencies' [e1,e2, e3]
  dependencies' (Lambda p e) =
    do
      fvsp <- freeVars p
      depsp <- dependencies p
      fvse <- dependencies e
      return $ (fvse \\ fvsp)++depsp
  dependencies' (Let ds e) =
    do
      fvsd <- dependencies ds
      newBoundVars <- liftM nub (concatMapM namesBoundByDecl ds)
      fvse <- dependencies e
      return $ nub (fvse++fvsd) \\ newBoundVars
  dependencies' (Lit _) = return []
  dependencies' (List es) = dependencies es
  dependencies' (ListComp es stmts) =
    do
      fvStmts <- freeVars stmts
      depsStmts <- dependencies stmts
      fvses' <- dependencies es
      let fvse = nub (fvses'++depsStmts)
      return $ fvse \\ fvStmts
  dependencies' (ListEnumFrom e1) = dependencies' e1
  dependencies' (ListEnumFromTo e1 e2) = dependencies' [e1,e2]
  dependencies' (ListLength e) = dependencies' e
  dependencies' (MathsBinaryOp _ e1 e2) = dependencies' [e1,e2]
  dependencies' (NegApp e) = dependencies' e
  dependencies' (Paren e) = dependencies' e
  dependencies' (Set es) = dependencies es
  dependencies' (SetComp es stmts) =
    do
      fvStmts <- freeVars stmts
      depsStmts <- dependencies stmts
      fvses' <- dependencies es
      let fvse = nub (fvses'++depsStmts)
      return $ fvse \\ fvStmts
  dependencies' (SetEnumComp es stmts) =
    do
      fvStmts <- freeVars stmts
      depsStmts <- dependencies stmts
      fvses' <- dependencies es
      let fvse = nub (fvses'++depsStmts)
      return $ fvse \\ fvStmts
  dependencies' (SetEnumFrom e1) = dependencies' e1
  dependencies' (SetEnumFromTo e1 e2) = dependencies' [e1,e2]
  dependencies' (SetEnum es) = dependencies' es
  dependencies' (Tuple es) = dependencies' es
  dependencies' (Var (UnQual n)) = return [n]

  dependencies' (UserOperator n es) = dependencies' es
  dependencies' (ReplicatedUserOperator n es stmts) =
    do
      fvStmts <- freeVars stmts
      depsStmts <- dependencies stmts
      fvses' <- dependencies es
      let fvse = nub (fvses'++depsStmts)
      return $ fvse \\ fvStmts

instance Dependencies Stmt where
  dependencies' (Generator p e) =
    do
      ds1 <- dependencies p
      ds2 <- dependencies e
      return $ ds1++ds2
  dependencies' (Qualifier e) = dependencies e

instance Dependencies Decl where
  dependencies' (FunBind n ms t) =
    do
      fvsms <- dependencies ms
      fvst <- dependencies t
      return $ fvsms++fvst
  dependencies' (PatBind p e) =
    do
      depsp <- dependencies p
      fvsp <- freeVars p
      depse <- dependencies e
      return $ depsp++depse
  dependencies' (Channel ns es) = dependencies es
  dependencies' (Assert e1 e2 m) = dependencies [e1, e2]
  dependencies' (DataType n cs) = dependencies [cs]
  dependencies' (External ns) = return []
  dependencies' (Transparent ns) = return []

instance Dependencies Match where
  dependencies' (Match ps e) =
    do
      fvs1 <- freeVars ps
      depsPs <- dependencies ps
      fvs2 <- dependencies e
      return $ (fvs2 \\ fvs1) ++ depsPs

instance Dependencies DataTypeClause where
  dependencies' (DataTypeClause n es) = dependencies es

namesBoundByDecl :: AnDecl -> TypeCheckMonad [Name]
namesBoundByDecl =  namesBoundByDecl' . removeAnnotation
namesBoundByDecl' (FunBind n ms t) = return [n]
namesBoundByDecl' (PatBind p ms) = freeVars p
namesBoundByDecl' (Channel ns es) = return ns
namesBoundByDecl' (Assert e1 e2 m) = return []
namesBoundByDecl' (DataType n dcs) =
  let
    namesBoundByDtClause (DataTypeClause n _) = [n]
  in
    return $ n:concatMap (namesBoundByDtClause . removeAnnotation) dcs
namesBoundByDecl' (External ns) = return ns
namesBoundByDecl' (Transparent ns) = return ns


class FreeVars a where
  freeVars :: a -> TypeCheckMonad [Name]
  freeVars = liftM nub . freeVars'

  freeVars' :: a -> TypeCheckMonad [Name]

instance FreeVars a => FreeVars [a] where
  freeVars' xs = concatMapM freeVars' xs

instance FreeVars a => FreeVars (Annotated b a) where
  freeVars' (Annotated _ _ inner) = freeVars inner

instance FreeVars Pat where
  freeVars' (PVar n) =
    do
      res <- safeGetType n
      case res of
        Just (ForAll _ t) ->
          case t of
            -- See typeCheck (PVar) for a discussion of why
            -- we only do this
            TDatatypeClause n ts  -> return []
            _                     -> return [n]
        Nothing -> return [n] -- var is not bound
```

```
  freeVars' (PConcat p1 p2) =
    do
      fvs1 <- freeVars' p1
      fvs2 <- freeVars' p2
      return $ fvs1++fvs2
  freeVars' (PDotApp p1 p2) = freeVars' [p1,p2]
  freeVars' (PList ps) = freeVars' ps
  freeVars' (PWildCard) = return []
  freeVars' (PTuple ps) = freeVars' ps
  freeVars' (PSet ps) = freeVars' ps
  freeVars' (PParen p) = freeVars' p
  freeVars' (PLit l) = return []
  freeVars' (PDoublePattern p1 p2) =
    do
      fvs1 <- freeVars' p1
      fvs2 <- freeVars' p2
      return $ fvs1++fvs2

instance FreeVars Stmt where
  freeVars' (Qualifier e) = return []
  freeVars' (Generator p e) = freeVars p
```

C.4.6. *CSPMTypeChecker/TCExpr.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, TypeSynonymInstances #-}
module CSPMTypeChecker.TCExpr () where

import CSPMDataStructures
import CSPMTypeChecker.TCCommon
import {-# SOURCE #-} CSPMTypeChecker.TCDecl
import CSPMTypeChecker.TCDependencies
import CSPMTypeChecker.TCPat
import CSPMTypeChecker.TCMonad
import CSPMTypeChecker.TCUnification

import Util

instance TypeCheckable PExp Type where
  errorConstructor = ErrorWithExp
  -- Important: we use the typeCheck' version after removing the annotation
  -- so the error message contains this node
  typeCheck' (Annotated srcloc typ inner) =
    do
      t' <- typeCheck' inner
      setPType typ t'
      return t'
instance TypeCheckable Exp Type where
  errorConstructor = error "Error:_expression_error_constructor_called."
  typeCheck' (App f args) =
    do
      tFunc <- typeCheck f
      tr <- freshTypeVar
      tArgs <- replicateM (length args) freshTypeVar
      unify (TFunction tArgs tr) tFunc

      tArgs' <- mapM typeCheck args
      errorIfFalse (length tArgs == length tArgs')
        (IncorrectNumberOfArguments f (length tArgs))
      unifiedTArgs <- zipWithM unify tArgs tArgs'
      return tr
  typeCheck' (BooleanBinaryOp op e1 e2) =
    do
      t1 <- typeCheck e1
      t2 <- typeCheck e2
      t <- unify t1 t2
      case op of
        And       -> ensureIsBool t
        Or        -> ensureIsBool t
        Equals      -> ensureHasConstraint Eq t
        NotEquals   -> ensureHasConstraint Eq t
        LessThan    -> ensureHasConstraint Ord t
        LessThanEq    -> ensureHasConstraint Ord t
        GreaterThan   -> ensureHasConstraint Ord t
        GreaterThanEq -> ensureHasConstraint Ord t
      return TBool
  typeCheck' (BooleanUnaryOp op e1) =
    do
      t1 <- typeCheck e1
      ensureIsBool t1
      return TBool
  typeCheck' (Concat e1 e2) =
    do
      t1 <- typeCheck e1
      t2 <- typeCheck e2
      ensureIsList t1
      ensureIsList t2
      unify t1 t2
  typeCheck' (DotApp e1 e2) =
    do
      t1 <- typeCheck e1
      t2 <- typeCheck e2

      argt <- freshTypeVar
      rt <- freshTypeVar
      unify t1 (TDotable argt rt)
      unify t2 argt

      return rt
  typeCheck' (If e1 e2 e3) =
    do
      t1 <- typeCheck e1
      ensureIsBool t1
      t2 <- typeCheck e2
      t3 <- typeCheck e3
      unify t2 t3
  typeCheck' (Lambda p exp) =
    do
      fvs <- freeVars p
      local fvs (
        do
          tr <- typeCheck exp
          targ <- typeCheck p
          return $ TFunction [targ] tr)
  typeCheck' (Let decls exp) =
    local [] (  -- Add a new scope: typeCheckDecl will add vars into it
      do
        typeCheckDecls decls
        typeCheck exp)
  typeCheck' (Lit lit) = typeCheck lit
  typeCheck' (List es) =
    do
      ts <- mapM typeCheck es
      t <- unifyAll ts
      return $ TSeq t
  typeCheck' (ListComp es stmts) =
    do
      fvs <- concatMapM freeVars stmts
      errorIfFalse (noDups fvs) (DuplicatedDefinitions fvs)
      local fvs (
        do
          stmts <- mapM (typeCheckStmt TSeq) stmts
          ts <- mapM typeCheck es
          t <- unifyAll ts
          return $ TSeq t)
  typeCheck' (ListEnumFrom lb) =
    do
      t1 <- typeCheck lb
      ensureIsInt t1
      return $ TSeq TInt
  typeCheck' (ListEnumFromTo lb ub) =
    do
      t1 <- typeCheck lb
      ensureIsInt t1
      t2 <- typeCheck ub
      ensureIsInt t1
      return $ TSeq TInt
  typeCheck' (ListLength e) =
    do
      t1 <- typeCheck e
      ensureIsList t1
      return $ TInt
  typeCheck' (MathsBinaryOp op e1 e2) =
    do
      t1 <- typeCheck e1
      t2 <- typeCheck e2
      ensureIsInt t1
      ensureIsInt t2
      return TInt
  typeCheck' (NegApp e1) =
    do
      t1 <- typeCheck e1
      ensureIsInt t1
      return TInt
  typeCheck' (Paren e) = typeCheck e
  typeCheck' (Set es) =
    do
      ts <- mapM typeCheck es
      t <- unifyAll ts
      ensureHasConstraint Eq t
      return $ TSet t
  typeCheck' (SetComp es stmts) =
    do
      fvs <- concatMapM freeVars stmts
      errorIfFalse (noDups fvs) (DuplicatedDefinitions fvs)
```

```
      local fvs (
        do
          stmts <- mapM (typeCheckStmt TSet) stmts
          ts <- mapM typeCheck es
          t <- unifyAll ts
          ensureHasConstraint Eq t
          return $ TSet t)
  typeCheck' (SetEnum es) =
    do
      ts <- mapM typeCheck es
      mapM ensureIsChannel ts
      return $ TSet (TChannel TListEnd)
  typeCheck' (SetEnumComp es stmts) =
    do
      fvs <- concatMapM freeVars stmts
      errorIfFalse (noDups fvs) (DuplicatedDefinitions fvs)
      local fvs (
        do
          stmts <- mapM (typeCheckStmt TSet) stmts
          ts <- mapM typeCheck es
          mapM ensureIsChannel ts
          return $ TSet (TChannel TListEnd))
  typeCheck' (SetEnumFrom lb) =
    do
      t1 <- typeCheck lb
      ensureIsInt t1
      -- No need to check for Eq - Ints always are
      return $ TSet TInt
  typeCheck' (SetEnumFromTo lb ub) =
    do
      t1 <- typeCheck lb
      ensureIsInt t1
      t2 <- typeCheck ub
      ensureIsInt t2
      -- No need to check for Eq - Ints always are
      return $ TSet TInt
  typeCheck' (Tuple es) =
    do
      ts <- mapM typeCheck es
      return $ TTuple ts
  typeCheck' (ReplicatedUserOperator n es stmts) =
    do
      fvs <- concatMapM freeVars stmts
      errorIfFalse (noDups fvs) (DuplicatedDefinitions fvs)
      local fvs (
        do
          stmts <- mapM (typeCheckStmt TSet) stmts
          ts <- mapM typeCheck es
          opMap <- getUserOperators
          let opTypes = apply opMap n
          zipWithM unify ts opTypes
          return TProc)
  typeCheck' (UserOperator n es) =
    do
      ts <- mapM typeCheck es
      opMap <- getUserOperators
      let opTypes = apply opMap n
      zipWithM unify ts opTypes
      return TProc
  typeCheck' (Var (UnQual n)) =
    do
      t <- getType n
      instantiate t

typeCheckStmt :: (Type -> Type) -> PStmt -> TypeCheckMonad Type
typeCheckStmt typc = typeCheckStmt' typc . removeAnnotation
typeCheckStmt' typc (Qualifier e) =
  do
    t <- typeCheck e
    ensureIsBool t
typeCheckStmt' typc (Generator p exp) =
  do
    tpat <- typeCheck p
    texp <- typeCheck exp
    unify (typc tpat) texp
```

C.4.7. *CSPMTypeChecker/TCModule.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, TypeSynonymInstances, FlexibleInstances #-}
module CSPMTypeChecker.TCModule where

import CSPMDataStructures
import CSPMTypeChecker.TCCommon
import CSPMTypeChecker.TCDecl
import CSPMTypeChecker.TCMonad
import qualified OpSemDataStructures as OpSem
import Util

instance TypeCheckable [PModule] () where
  errorConstructor an = id
  typeCheck' ([m]) = typeCheck m

instance TypeCheckable PModule () where
  errorConstructor = ErrorWithModule
  typeCheck' (Annotated _ _ m) = typeCheck' m

instance TypeCheckable Module () where
  errorConstructor = error "Error:_Module_error_constructor_called"
  typeCheck' (GlobalModule ds) =
    typeCheckDecls ds


typeCheckModules :: OpSem.OpSemDefinition -> [PModule] -> TypeCheckMonad [TCModule]
typeCheckModules opSemDefn ms =
  do
    let operatorTypeMap =
      [(OpSem.opFriendlyName op, map snd (OpSem.opArgs op))
        | op <- OpSem.operators opSemDefn]
    typeCheckWrapper ms operatorTypeMap
    return ms
```

C.4.8. *CSPMTypeChecker/TCMonad.hs.*

```
module CSPMTypeChecker.TCMonad
  (module Control.Monad.Error,
  TypeCheckError(..), TypeCheckMonad, readTypeRef, writeTypeRef, freshTypeVar,
  freshTypeVarWithConstraints, getType, safeGetType, setType,
  local, getEnvironment, compress, compressTypeScheme, errorIfFalseM, errorIfFalse,
  getUserOperators,

  runTypeChecker, setUserOperators
  )
  where

import Control.Monad.Error
import Control.Monad.State

import List (nub, (\\), intersect, group, sort)
import Text.PrettyPrint.HughesPJ

import Data.Graph
import Data.IORef
import Prelude

import CSPMDataStructures
import {-# SOURCE #-} CSPMPrettyPrinter
import qualified OpSemDataStructures as OpSem
import Util

-- ************************************************************************
-- Type Checker Monad
-- ************************************************************************
data TypeCheckError =
  ErrorWithExp PExp TypeCheckError
  | ErrorWithPat PPat TypeCheckError
  | ErrorWithMatch PMatch TypeCheckError
  | ErrorWithDataTypeClause PDataTypeClause TypeCheckError
  | ErrorWithDecl PDecl TypeCheckError
  | ErrorWithModule PModule TypeCheckError
  | UnificationError (Name, Type) (Name, Type)
  | UnknownUnificationError Type Type
  | InfiniteUnificationError TypeVar Type
  | DuplicatedDefinitions [Name]  -- ns is not the duplicates - we calc these
  | IncorrectNumberOfArguments PExp Int
  | InvalidSetPattern [PPat]
  | UnknownError String

  | VariableNotInScope Name

instance Error TypeCheckError where
  strMsg = UnknownError

instance Show TypeCheckError where
  show (ErrorWithPat (Annotated srcloc _ pat) err) =
    show err++
    "in_pattern:\n"++
    indentEveryLine (show (prettyPrint pat))
  show (ErrorWithDecl (Annotated srcloc _ (FunBind (Name n) ms _)) err) =
    show err++
    "in_the_declaration_of_"++n++".\n"
  show (ErrorWithDecl (Annotated srcloc _ (PatBind p e)) err) =
    show err++
```

```
    "in the declaration of "++show (prettyPrint p)++".\n"
  show (ErrorWithMatch (Annotated srcloc _ m) err) = show err
  show (ErrorWithDataTypeClause (Annotated srcloc _ m) err) = show err
  show (ErrorWithDecl (Annotated srcloc _ d) err) =
    show err++
    "in the declaration of:"++
    show (prettyPrint d)
  show (ErrorWithExp (Annotated srcloc _ exp) err) =
    show err++
    "in the expression at "++show srcloc++":\n"++
    indentEveryLine (show (prettyPrint exp))
  show (ErrorWithModule (Annotated srcloc _ exp) err) = show err
  show (InfiniteUnificationError tv typ) =
    "Cannot construct the infinite type:"++
    show tv++" = "++show (prettyPrintType [] typ)++"."
  show (UnknownUnificationError t1 t2) =
    "Could not match the types:\n"++
    "\t"++show (prettyPrintType [] t1)++
    "\nand\n"++
    "\t"++show (prettyPrintType [] t2)++"\n"
  show (DuplicatedDefinitions ns) =
    "The variables:"++
      show (hsep (punctuate comma (map (\ (Name n) -> text n) dupedVars)))++
    " have multiple definitions."
    where
      dupedVars = (map head . filter (\ l -> length l > 1) . group . sort) ns
  show (IncorrectNumberOfArguments exp correct) =
    "An incorrect number (correct number: "++show correct++
    ") of arguments was supplied to:\n"++
    indentEveryLine (show (prettyPrint exp))
  show (InvalidSetPattern ps) =
    "You may only pattern match on set patterns of length 0 or 1."
  show (VariableNotInScope (Name n)) =
    "Name "++n++" is not in scope\n"
  show (UnknownError s) =
    "An unknown error occured: "++s

type Environment = [PartialFunction Name TypeScheme]
type UserOperators = PartialFunction Name [Type]

data TypeInferenceState = TypeInferenceState {
    -- map from names to arbitrary types
    environment :: Environment,
    -- Next TypeVar to be allocated
    nextTypeId :: Int,
    -- map from operator name to types of args
    operatorSymbolTable :: UserOperators
  }

type TypeCheckMonad =
  ErrorT TypeCheckError (StateT TypeInferenceState Tyger)

runTypeChecker :: TypeCheckMonad a -> Tyger a
runTypeChecker prog =
  do
    (errOrVal,state)<-
      runStateT (runErrorT prog) (TypeInferenceState [[]] 0 [])
    case errOrVal of
      Left err   -> throwError $ CSPMTypeCheckError (show err)
      Right val  -> return val

getEnvironment :: TypeCheckMonad Environment
getEnvironment = gets environment

getUserOperators :: TypeCheckMonad UserOperators
getUserOperators = gets operatorSymbolTable

setUserOperators :: UserOperators -> TypeCheckMonad ()
setUserOperators ops = modify (\ s -> s { operatorSymbolTable = ops})

errorIfFalse :: Bool -> TypeCheckError -> TypeCheckMonad ()
errorIfFalse True e = return ()
errorIfFalse False e = throwError e

errorIfFalseM :: TypeCheckMonad Bool -> TypeCheckError -> TypeCheckMonad ()
errorIfFalseM m e =
  do
    res <- m
    errorIfFalse res e

-- ************************************************************************
-- Type Operations
-- ************************************************************************
readTypeRef :: TypeVarRef -> TypeCheckMonad (Either (TypeVar, [Constraint]) Type)
readTypeRef (TypeVarRef tv cs ioref) =
  do
    mtyp <- readPType ioref
    case mtyp of
      Just t  -> return (Right t)
      Nothing -> return (Left (tv, cs))

writeTypeRef :: TypeVarRef -> Type -> TypeCheckMonad ()
writeTypeRef (TypeVarRef tv cs ioref) t = setPType ioref t

freshTypeVar :: TypeCheckMonad Type
freshTypeVar = freshTypeVarWithConstraints []

freshTypeVarWithConstraints :: [Constraint] -> TypeCheckMonad Type
freshTypeVarWithConstraints cs =
  do
    nextId <- gets nextTypeId
    modify (\s -> s { nextTypeId = nextId+1 })
    ioRef <- freshPType
    return $ TVar (TypeVarRef (TypeVar nextId) cs ioRef)


safeGetType_ :: [PartialFunction Name TypeScheme] -> Name -> Maybe TypeScheme
safeGetType_ [] n = Nothing
safeGetType_ (pf:pfs) n =
  case safeApply pf n of
    Just t  -> Just t
    Nothing -> safeGetType_ pfs n

getType :: Name -> TypeCheckMonad TypeScheme
getType name =
  do
    envs <- gets environment
    case safeGetType_ envs name of
      Just t  -> return t
      Nothing -> throwError $ VariableNotInScope name

safeGetType :: Name -> TypeCheckMonad (Maybe TypeScheme)
safeGetType n =
  do
    envs <- gets environment
    return $ safeGetType_ envs n

-- Sets the type of n to be t in the current scope only. No unification is
-- performed.
setType :: Name -> TypeScheme -> TypeCheckMonad ()
setType n t =
  do
    res <- safeGetType n
    (env:envs) <- gets environment
    let env' = updatePF env n t
    modify (\ s -> s { environment = env':envs })

local :: [Name] -> TypeCheckMonad a -> TypeCheckMonad a
local ns m =
  do
    env <- gets environment
    newArgs <- replicateM (length ns) freshTypeVar
    modify (\s -> s { environment = (zip ns (map (ForAll []) newArgs)):env })

    res <- m

    env <- gets environment
    modify (\ s -> s { environment = tail env })

    return res

compressTypeScheme :: TypeScheme -> TypeCheckMonad TypeScheme
compressTypeScheme (ForAll ts t) =
  do
    t' <- compress t
    return $ ForAll ts t'
compress :: Type -> TypeCheckMonad Type
compress (tr @ (TVar typeRef)) =
  do
    res <- readTypeRef typeRef
    case res of
      Left tv -> return tr
      Right t -> compress t
compress (TFunction targs tr) =
  do
    targs' <- mapM compress targs
    tr' <- compress tr
    return $ TFunction targs' tr'
compress (TSeq t) =
  do
    t' <- compress t
    return $ TSeq t'
compress (TSet t) =
  do
    t' <- compress t
```

```
    return $ TSet t'
compress (TTuple ts) =
  do
    ts' <- mapM compress ts
    return $ TTuple ts'
compress (TDotable t1 t2)=
  do
    t1' <- compress t1
    t2' <- compress t2
    return $ TDotable t1' t2'
compress (TDatatypeClause n ts) =
  do
    ts' <- mapM compress ts
    return $ TDatatypeClause n ts'
compress (TChannel t1) =
  do
    t1' <- compress t1
    return $ TChannel t1'
compress (TList t1 t2) =
  do
    t1 <- compress t1
    t2 <- compress t2
    return $ TList t1 t2
compress (tr @ (TPolyList typeref)) =
  do
    res <- readTypeRef typeref
    case res of
      Left tv -> return tr
      Right t -> compress t
compress t = return t
```

C.4.9. *CSPMTypeChecker/TCPat.hs.*

```
{-# LANGUAGE MultiParamTypeClasses, TypeSynonymInstances #-}
module CSPMTypeChecker.TCPat () where

import CSPMDataStructures
import CSPMTypeChecker.TCCommon
import CSPMTypeChecker.TCMonad
import CSPMTypeChecker.TCUnification

instance TypeCheckable PPat Type where
  errorConstructor = ErrorWithPat
  typeCheck' (Annotated srcloc typ inner) =
    do
      t' <- typeCheck' inner
      return t'
instance TypeCheckable Pat Type where
  errorConstructor = error "Error:_pattern_error_constructor_called"
  typeCheck' (PConcat p1 p2) =
    do
      t1 <- typeCheck p1
      t1 <- ensureIsList t1
      t2 <- typeCheck p2
      t2 <- ensureIsList t2
      unify t1 t2
  typeCheck' (PDoublePattern p1 p2) =
    do
      t1 <- typeCheck p1
      t2 <- typeCheck p2
      unify t1 t2
  typeCheck' (PDotApp p1 p2) =
    do
      t1 <- typeCheck p1
      t2 <- typeCheck p2
      argt <- freshTypeVar
      rt <- freshTypeVar
      unify t1 (TDotable argt rt)
      unify t2 argt
      return rt
  typeCheck' (PList ps) =
    do
      ts <- mapM typeCheck ps
      t <- unifyAll ts
      return $ TSeq t
  typeCheck' (PLit lit) = typeCheck lit
  typeCheck' (PParen p1) = typeCheck p1
  typeCheck' (PSet ps) =
      do
        errorIfFalse (length ps <= 1) (InvalidSetPattern ps)
        ts <- mapM typeCheck ps
        t <- unifyAll ts
        ensureHasConstraint Eq t
        return $ TSet t
  typeCheck' (PTuple ps) =
    do
      ts <- mapM typeCheck ps
      return $ TTuple ts
  typeCheck' (PWildCard) = freshTypeVar
  typeCheck' (PVar n) =
    do
      t @ (ForAll _ t') <- getType n
      -- All variables are already in scope hence we can just return the
      -- type (since we always typeCheck a pattern in between a
      -- local (freeVars pat)).
      return t'
```

C.4.10. *CSPMTypeChecker/TCUnification.hs.*

```
module CSPMTypeChecker.TCUnification
  (generaliseGroup, instantiate, unify, unifyAll) where

import List (nub, (\\), intersect, group, sort)

import Prelude

import CSPMDataStructures
import CSPMTypeChecker.TCMonad

import Util


class FreeTypeVars a where
  freeTypeVars :: a -> TypeCheckMonad [(TypeVar, [Constraint])]
  freeTypeVars = liftM nub . freeTypeVars'

  freeTypeVars' :: a -> TypeCheckMonad [(TypeVar, [Constraint])]

instance FreeTypeVars Type where
  freeTypeVars' (TVar tv) =
    do
      typ <- readTypeRef tv
      case typ of
        Left (tv, cs) -> return [(tv, cs)]
        Right t       -> freeTypeVars' t
  freeTypeVars' (TFunction targs tr) =
    liftM concat (mapM freeTypeVars' (tr:targs))
  freeTypeVars' (TSeq t) = freeTypeVars' t
  freeTypeVars' (TSet t) = freeTypeVars' t
  freeTypeVars' (TInt) = return []
  freeTypeVars' TBool = return []
  freeTypeVars' (TTuple ts) = liftM concat (mapM freeTypeVars' ts)
  freeTypeVars' (TChannel t1) = freeTypeVars t1
  freeTypeVars' (TDotable t1 t2) = liftM concat (mapM freeTypeVars' [t1,t2])
  freeTypeVars' (TDatatypeClause n1 ts) = liftM concat (mapM freeTypeVars' ts)
  freeTypeVars' (TList t1 t2) = liftM concat (mapM freeTypeVars' [t1,t2])
  freeTypeVars' (TListEnd) = return []
  freeTypeVars' (TPolyList tv) =
    do
      typ <- readTypeRef tv
      case typ of
        Left (tv, cs) -> return [(tv, cs)]
        Right t       -> freeTypeVars' t
  freeTypeVars' (TProc) = return []

-- ************************************************************************
-- Unification + Substitution
-- ************************************************************************
-- Name is a workaround for the problem as follows:
--  we convert a type T into forall vs T where vs = fvts (T) - fvts(Env)
--  where Env does not contain the function whose type we are generalizing
--  (this is because when we type a declaration we are really typing a
--  lambda function).
generaliseGroup :: [Name] -> [TypeCheckMonad [(Name, Type)]] ->
        TypeCheckMonad [[(Name, TypeScheme)]]
generaliseGroup names tsm =
  do
    ts <- sequence tsm
    envs <- getEnvironment
    envfvs <- liftM nub
      (concatMapM freeTypeVars
        [t | env <- envs, (n, (ForAll _ t)) <- env, not (n `elem` names)])
    mapM (\ nts -> mapM (\ (n,t) ->
                  do
                    deffvs <- freeTypeVars t
                    let unboundVars =
                      filter (\ (fv, cs) ->
                        not (fv `elem` map fst envfvs)) deffvs
                    let t' = ForAll unboundVars t
                    setType n t'
                    return $ (n, t')) nts) ts
```

```
instantiate :: TypeScheme -> TypeCheckMonad Type
instantiate (ForAll ts t) =
  do
    tvs <- mapM (freshTypeVarWithConstraints . snd) ts
    foldM (\ x y -> substituteType y x) t (zip (map fst ts) tvs)

occurs :: TypeVar -> Type -> TypeCheckMonad Bool
occurs a (TVar (tvref @ (TypeVarRef tv _ _))) =
  do
    res <- readTypeRef tvref
    case res of
      Left (tv,cs)-> return $ a == tv
      Right t   -> occurs a t
occurs a (TSet t) = occurs a t
occurs a (TSeq t) = occurs a t
occurs a (TTuple ts) = liftM or (mapM (occurs a) ts)
occurs a (TFunction ts t) = liftM or (mapM (occurs a) (t:ts))
occurs a (TList t1 t2) = liftM or (mapM (occurs a) [t1,t2])
occurs a (TChannel t) = occurs a t
occurs a (TDatatypeClause n ts) = liftM or (mapM (occurs a) ts)
occurs a (TDotable t1 t2) = liftM or (mapM (occurs a) [t1,t2])
occurs a (TPolyList (tvref @ (TypeVarRef tv _ _))) =
  do
    res <- readTypeRef tvref
    case res of
      Left (tv,cs)-> return $ a == tv
      Right t   -> occurs a t
occurs a TListEnd = return False
occurs a TInt = return False
occurs a TBool = return False
occurs a TProc = return False

unifyTypeSchemes :: TypeScheme -> TypeScheme -> TypeCheckMonad TypeScheme
unifyTypeSchemes (ForAll ts1 t1) (ForAll ts2 t2) =
  do
    t3 <- unify t1 t2
    return $ ForAll ts1 t3

unifyAll :: [Type] -> TypeCheckMonad Type
unifyAll [] = freshTypeVar
unifyAll [t] = return t
unifyAll (t1:ts) =
  do
    t2 <- unifyAll ts
    unify t1 t2

-- Takes a constraint and a type and returns True iff the type satisfies the
-- constraint, or can be made to satsify the constraint by appropriate type
-- substitutions.
unifyConstraint :: Constraint -> Type -> TypeCheckMonad Bool
unifyConstraint c (TVar v) =
  do
    res <- readTypeRef v
    case res of
      Left (tva, cs)  -> if c `elem` cs then return True else
                do
                  fv <- freshTypeVarWithConstraints (nub (cs++[c]))
                  applySubstitution v fv
                  return True
      Right t     -> unifyConstraint c t
unifyConstraint c TInt = return True
unifyConstraint Eq TBool = return True -- Bools are not orderable P524
unifyConstraint c (TSeq t) = unifyConstraint c t
unifyConstraint c (TSet t) = return True -- All set elements must support comparison
unifyConstraint c (TTuple ts) = liftM and (mapM (unifyConstraint c) ts)
unifyConstraint c (TList t1 t2) = liftM and (mapM (unifyConstraint c) [t1, t2])
unifyConstraint c (TListEnd) = return True
unifyConstraint c (TPolyList v) =
  do
    res <- readTypeRef v
    case res of
      Left (tva, cs)  -> if c `elem` cs then return True else
                do
                  (TVar nref) <-
                    freshTypeVarWithConstraints (nub (cs++[c]))
                  applySubstitution v (TPolyList nref)
                  return True
      Right t     -> unifyConstraint c t
unifyConstraint Eq (TChannel t1) = return True -- channels are comparable only
unifyConstraint Eq (TDotable a b) = -- channels and datatypes are only dotable things
  liftM and (mapM (unifyConstraint Eq) [a,b])
unifyConstraint Eq (TDatatypeClause n ts) =
  liftM and (mapM (unifyConstraint Eq) ts) -- User data types are not orderable P524
unifyConstraint c t = return False

unify :: Type -> Type -> TypeCheckMonad Type
unify (TVar t1) (TVar t2) | t1 == t2 =
  return (TVar t1)
unify (TVar t1) (TVar t2) =
  do
    res1 <- readTypeRef t1
    res2 <- readTypeRef t2
    case (res1, res2) of
      (Left (tv1, cs1), Left (tv2,cs2)) ->
        do
          fv <- freshTypeVarWithConstraints (nub (cs1 ++ cs2))
          applySubstitution t1 fv
          applySubstitution t2 fv
          return fv
      (Left _, Right t)   -> unify (TVar t1) t
      (Right t, Left _)   -> unify t (TVar t2)
      (Right t1, Right t2)  -> unify t1 t2
unify (TVar a) b =
  do
    res <- readTypeRef a
    case res of
      Left (tva, cs)  ->
        do
          res <- liftM and (mapM (\ c -> unifyConstraint c b) cs)
          if res then applySubstitution a b
            else do
                t1' <- compress (TVar a)
                t2' <- compress b
                throwError $ UnknownUnificationError t1' t2'
      Right t     -> unify t b
unify t (TVar b) = unify (TVar b) t

unify TInt TInt = return TInt
unify TBool TBool = return TBool
unify TProc TProc = return TProc

unify (TFunction ts1 rt1) (TFunction ts2 rt2) | length ts1 == length ts2 =
  do
    ts <- zipWithM unify ts1 ts2
    rt <- unify rt1 rt2
    return $ TFunction ts rt
unify (TSeq t1) (TSeq t2) =
  do
    tr <- unify t1 t2
    return $ TSeq tr
unify (TSet t1) (TSet t2) =
  do
    tr <- unify t1 t2
    return $ TSet tr
unify (TTuple ts1) (TTuple ts2) | length ts1 == length ts2 =
  do
    ts <- zipWithM unify ts1 ts2
    return $ TTuple ts
unify (TDotable t1 t2) (TDotable t1' t2') =
  do
    a <- unify t1 t1'
    b <- unify t2 t2'
    return $ TDotable a b
unify (TChannel t11) (TChannel t12) =
  do
    t1 <- unify t11 t12
    return $ TChannel t1
unify (TDatatypeClause n1 ts1) (TDatatypeClause n2 ts2)
  | n1 == n2 && length ts1 == length ts2 =
  do
    ts <- zipWithM unify ts1 ts2
    return $ TDatatypeClause n1 ts

unify (TList t1 t2) (TList t1' t2') =
  do
    a <- unify t1 t1'
    b <- unify t2 t2'
    return $ TList a b
unify (TPolyList t1) (TPolyList t2) | t1 == t2 = return $ TPolyList t1
unify (TPolyList t1) (TPolyList t2) =
  do
    res1 <- readTypeRef t1
    res2 <- readTypeRef t2
    case (res1, res2) of
      (Left (tv1, cs1), Left (tv2,cs2)) ->
        do
          (TVar fv) <- freshTypeVarWithConstraints (nub (cs1 ++ cs2))
          applySubstitution t1 (TPolyList fv)
          applySubstitution t2 (TPolyList fv)
          return (TPolyList fv)
      (Left _, Right t)   -> unify (TVar t1) t
      (Right t, Left _)   -> unify t (TVar t2)
      (Right t1, Right t2)  -> unify t1 t2
```

```
unify (TPolyList t1) t2 =
  do
    res <- readTypeRef t1
    case res of
      Left (tva, cs)  ->
        do
          res <- liftM and (mapM (\ c -> unifyConstraint c t2) cs)
          if res then applySubstitution t1 t2
            else do
                 t1' <- compress (TVar t1)
                 t2' <- compress t2
                 throwError $ UnknownUnificationError t1' t2'
      Right t'    -> unify t' t2
unify t (TPolyList t1) = unify (TPolyList t1) t
unify (TListEnd) (TListEnd) = return TListEnd

-- Non trivial cases
unify (TDotable t1 t2) (TChannel tl) =
  do
    fv1 <- freshTypeVar
    fv2 <- freshTypeVar
    unify tl (TList fv1 fv2)
    t' <- unify t1 fv1
    (TChannel tl') <- unify t2 (TChannel fv2)
    return $ TChannel (TList t' tl')
unify (TChannel tl) (TDotable t1 t2) = unify (TDotable t1 t2) (TChannel tl)
unify (TDotable t1 t2) (TDatatypeClause n1 (t1':ts)) =
  do
    t' <- unify t1 t1'
    (TDatatypeClause _ ts') <- unify t2 (TDatatypeClause n1 ts)
    return $ TDatatypeClause n1 (t':ts')
unify (TDatatypeClause n1 ts) (TDotable t1 t2) =
  unify (TDotable t1 t2) (TDatatypeClause n1 ts)
unify t1 t2 =
  do
    t1' <- compress t1
    t2' <- compress t2
    throwError $ UnknownUnificationError t1' t2'

-- Returns the type that we substitute for
-- NB: in a quantified type we do not apply the substitution to any
-- quantified variables
applySubstitution :: TypeVarRef -> Type -> TypeCheckMonad Type
applySubstitution (tvref @ (TypeVarRef tv _ _)) typ =
  do
    t' <- compress typ
    errorIfFalseM (liftM not (occurs tv typ)) (InfiniteUnificationError tv t')
    writeTypeRef tvref typ
    return typ

-- Applies a subtitution directly to the type. This is used in
-- type instantiation where we create a fresh type for each universal
-- variable
```

```
substituteType :: (TypeVar, Type) -> Type -> TypeCheckMonad Type
substituteType (tv, t) (b @ (TVar (a @ (TypeVarRef tv' cs ioref)))) =
  do
    res <- readTypeRef a
    case res of
      Left tva  -> if tv == tv' then return t else return b
      Right t'  -> substituteType (tv, t) t'
substituteType sub (TFunction targs tr) =
  do
    targs' <- mapM (substituteType sub) targs
    tr' <- substituteType sub tr
    return $ TFunction targs' tr'
substituteType sub (TSeq t) =
  do
    t' <- substituteType sub t
    return $ TSeq t'
substituteType sub (TSet t) =
  do
    t' <- substituteType sub t
    return $ TSet t'
substituteType sub (TChannel tl) =
  do
    tl' <- substituteType sub tl
    return $ TChannel tl'
substituteType sub (TList t1 t2) =
  do
    t1' <- substituteType sub t1
    t2' <- substituteType sub t2
    return $ TList t1' t2'
substituteType (tv,t) (b @ (TPolyList (a @ (TypeVarRef tv' cs ioref)))) =
  do
    res <- readTypeRef a
    case res of
      Left tva  -> if tv == tv' then return t else return b
      Right t'  -> substituteType (tv, t) t'
substituteType sub TListEnd = return TListEnd
substituteType sub (TDotable t1 t2) =
  do
    t1' <- substituteType sub t1
    t2' <- substituteType sub t2
    return $ TDotable t1' t2'
substituteType sub (TTuple ts) =
  do
    ts' <- mapM (substituteType sub) ts
    return $ TTuple ts'
substituteType sub (TDatatypeClause n1 ts)=
  do
    ts' <- mapM (substituteType sub) ts
    return $ TDatatypeClause n1 ts'
substituteType sub TInt = return TInt
substituteType sub TBool = return TBool
substituteType sub TProc = return TProc
```