L-Università ta' Malta

Department of Computer Science
Faculty of ICT

Final year project in computer science for the partial fulfilment for the award of
B.Sc. (Hons.) Mathematics and Computer Science

Synthesising Safety Runtime Enforcement Monitors for μHML

Luke Collins

supervised by

Prof. Adrian Francalanza

4^{th} June, 2019
Academic Year 2018/2019
ICT3004: APT in Computer Science
Version 1.1

Abstract

In this project, we consider a subset sHML of formulæ in the Hennessy-Milner Logic with recursion (μHML) which are enforcable through suppressions. A synthesis function is introduced, which converts safety properties in sHML to suppression enforcers through a formula normalisation process. This synthesis function assumes that different branches in the input formula are disjoint, and that every variable is guarded by modal necessity—such formulæ are said to be in normal form. It turns out that this restriction of input formulæ is only superficial: an algorithm which converts any given formula in sHML to an equivalent formula in normal form is implemented in the form of a Haskell program.

Table of Contents

1 Introduction

Runtime monitoring is the process of analysing the behaviour of a software system at runtime via *monitors*, software entities which compare the behaviour of a system against some correctness specification. Runtime enforcement (RE) is a specialised form of runtime monitoring which ensures that the behaviour of the system is always in agreement with the correctness specification. The role of the monitor in RE is to anticipate incorrect behaviour and take necessary measures to prevent it.

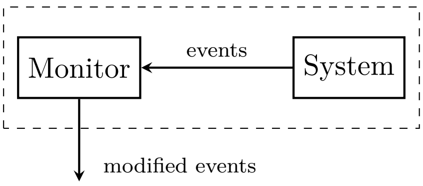

FIGURE 1.1: Runtime Enforcement

Typically the monitor is designed to act as an ostiary, wrapping itself around the system and analysing any external interactions (figure 1.1). This allows it to transform any incorrect actions by replacing them, suppressing them, or inserting other actions.

Software systems are becoming larger and more complex, so building an ad hoc monitor for a software system from scratch is seldom feasible, and might result in more room for error in development. Instead, the correctness specification of a system is expressed as a formula in some *logic* with precise formal semantics, and a program designed to interpret this logic synthesises the monitor automatically.

The expressiveness of the logic used for defining the correctness specification is an important consideration. Unfortunately the expressiveness of a logic is adverse to its enforceability, meaning that the more expressive a logic is, the more likely it is that certain formulæ in that logic cannot be synthesised into monitors.

This document is structured as follows. In the remainder of chapter 1, some preliminary notions are introduced, and the logics μHML, sHML and sHML_{nf} are discussed in view of their expressiveness and enforceability. The goal of the project is to realise a theoretical construction detailed in [1] which transforms formulæ from sHML into sHML_{nf} in the form of a Haskell program. How this is achieved is the subject of chapters 2 and 3. Possible future work is outlined in chapter 4. Finally, the code for the construction is presented in appendix A.

1.1 Preliminaries

1.1.1 Concrete Events and Patterns

The behaviour of a system is represented as a stream of observable operations called (*concrete*) *events*. Let $\textsc{Val}$, $\textsc{Prc}$ and $\textsc{Var}$ be pairwise disjoint sets whose members are to be called *values*, *process names*, and *free variables*; respectively. Moreover, let $\textsc{Pid} = \textsc{Prc} \cup \textsc{Var}$, and similarly $\textsc{Vid} = \textsc{Val} \cup \textsc{Var}$. If $i \in \textsc{Prc}$ and $\delta \in \textsc{Val}$, then $i\,?\,\delta$ denotes the event that a process with identifier i inputs δ, whereas $i\,!\,\delta$ denotes the event that a process with identifier i outputs δ. The set of such concrete events is denoted by $\textsc{Evt}$, i.e., we have $\textsc{Evt} = \textsc{Prc}\{?,!\}\textsc{Val}$.

A *pattern* is a syntactic object which represents possible concrete events. For example, if $x \in \textsc{Var}$ and $\delta \in \textsc{Val}$, then $x\,?\,\delta$ represents patterns which input the value δ to some unspecified process identifier. Variables in a pattern may either occur free, such as x in $x\,?\,\delta$, or as binders, which we denote by prepending a dollar sign: $\$x\,?\,\delta$. The set $\textsc{Patt}$ of patterns is defined in definition 1.1.

$$
\begin{array}{rlll}
p, q \in \textsc{Patt} ::= & \textsc{Pid}\,?\,\textsc{Vid} \quad \text{(input)} & \mid \textsc{Pid}\,!\,\textsc{Vid} \quad \text{(output)} \\
\mid & \textsc{Pid}\,?\,\$\textsc{Var} & \mid \textsc{Pid}\,!\,\$\textsc{Var} \\
\mid & \$\textsc{Var}\,?\,\textsc{Vid} & \mid \$\textsc{Var}\,!\,\textsc{Vid} \\
\mid & \$\textsc{Var}\,?\,\$\textsc{Var} & \mid \$\textsc{Var}\,!\,\$\textsc{Var}
\end{array}
$$

Definition 1.1: Patterns

The set of *free variables* in a pattern p, denoted $\text{fv}(p)$, contains the variables which appear unbounded in p; e.g. $\text{fv}(\$x\,?\,y) = \{y\}$. Similarly the *bound variables* in a pattern p, denoted $\text{bv}(p)$, contains the variables which appear bounded in p; e.g. $\text{bv}(\$x\,?\,y) = \{x\}$.

Pattern matching is the process of checking whether a concrete event conforms to a given pattern. For example, the concrete event $i\ ?\ \delta$ where $i \in \textsc{Prc}$ matches the pattern $x\ ?\ \delta$ from earlier, but $i\ !\ \delta$ or $i\ ?\ \vartheta$ where $\delta \neq \vartheta \in \textsc{Val}$ do not. The *pattern matching function* $\mathrm{mt}\colon \textsc{Patt} \times \textsc{Evt} \rightharpoonup (\textsc{Var} \rightharpoonup (\textsc{Prc} \cup \textsc{Val}))$ is a partial function which checks whether a given pattern and concrete event are compatible. If they are compatible, mt returns a *substitution* σ, that is, a partial map from the free variables which appear in the pattern to the respective values. For example, $\mathrm{mt}(x\ ?\ \delta, i\ ?\ \delta) = \{x \mapsto i\}$ and $\mathrm{mt}(i\ ?\ \delta, i\ ?\ \delta) = \emptyset$, whereas $\mathrm{mt}(x\ ?\ \delta, i\ !\ \delta)$ and $\mathrm{mt}(x\ ?\ \delta, i\ ?\ \vartheta)$ are not defined.

If $p \in \textsc{Patt}$ is a pattern and $\sigma\colon \mathrm{fv}(p) \to \textsc{Pid} \cup \textsc{Val}$ is a substitution, then the application of σ to p is denoted by $p\sigma$. Put differently, if $\mathrm{mt}(p, \alpha) = \sigma$, then $p\sigma = \alpha$.

Two patterns $p, q \in \textsc{Patt}$ are said to be equivalent or isomorphic, written $p \simeq q$, if they describe the same concrete events. In other words,

$$p \simeq q \Leftrightarrow \forall \alpha \in \textsc{Evt} \cdot \mathrm{mt}(p, \alpha) = \mathrm{mt}(q, \alpha).$$

The quotient set $\textsc{Patt}/\simeq$ is then the set of patterns which are unique up to isomorphism.

1.1.2 Symbolic Events

Let $\textsc{Cond}(V)$ be the set of decidable logical predicates involving the variables in the set $V \subseteq \textsc{Var}$. If $c \in \textsc{Cond}(V)$, let $\mathrm{fv}(c) \subseteq V$ denote the variables appearing in c. In other words, if $\mathrm{fv}(c) = \{v_1, v_2, \ldots, v_n\} \subseteq V$, then $c = c(v_1, v_2, \ldots, v_n)$.

A *closed* predicate is a predicate $c \in \textsc{Cond}(V)$ such that $\mathrm{fv}(c) = \emptyset$. Using the usual inference rules of predicate logic, we can evaluate closed predicates down to *true* or *false*. Symbolically, $\mathrm{fv}(c) = \emptyset \implies (c \Downarrow \mathit{true}) \vee (c \Downarrow \mathit{false})$.

We also have substitutions for predicates. If c is a predicate, then a substitution is a partial map $\sigma\colon \mathrm{fv}(c) \rightharpoonup \textsc{Pid} \cup \textsc{Val}$. For example, if c is the predicate $x \geqslant y$ and $\sigma = \{x \mapsto 3, y \mapsto 4\}$, then $c\sigma = 3 \geqslant 4 \Downarrow \mathit{false}$.

Now we can generalise the idea of concrete events to that of *symbolic events* (a.k.a. *symbolic actions*). The set $\textsc{SEvt}$ of symbolic events is defined by

$$\textsc{SEvt} = \{(p, c) \in \textsc{Patt} \times \textsc{Cond}(\textsc{Var}) \mid \mathrm{fv}(c) \subseteq \mathrm{bv}(p)\}.$$

In other words, $\textsc{SEvt}$ is the set of pairs of patterns and predicates, where the predicate says something about the variables in the pattern. We will denote symbolic events using the notation $\{p, c\}$ instead of (p, c).

What the symbolic event $\{p, c\}$ describes is the set of concrete events which conform to the pattern p, and, moreover, satisfy the condition c. This is similar to the idea of set comprehension, where $\{x \in A \mid \phi(x)\}$ denotes the set of objects x which satisfy the condition $\phi(x)$.

Definition 1.2 (Filter Set)**.** Given a symbolic event $\eta = \{\!|p, c|\!\}$, the *filter set* of η, denoted $\Phi(\eta)$, is the set

$$\Phi(\{\!|p, c|\!\}) = \{\alpha \in \textsc{Evt} \mid \mathrm{mt}(p, \alpha) = \sigma \ \wedge \ c\sigma \Downarrow \mathit{true}\},$$

i.e., the set of concrete events which conform to p and satisfy c.

Example 1.3. Suppose we have $\textsc{Val} = \{1, 2, 3, 4, 5\}$, $\textsc{Pid} = \{i, j, k\}$ and $\textsc{Var} = \{x, y, z\}$. Then

$$\begin{aligned}\Phi(\{\!|x\,?\,y, x \neq k \wedge y \geqslant 3|\!\}) &= \{i\,?\,3, i\,?\,4, i\,?\,5, j\,?\,3, j\,?\,4, j\,?\,5\}\\ \Phi(\{\!|x\,!\,y, y = 1|\!\}) &= \{i\,!\,1, j\,!\,1, k\,!\,1\}\end{aligned}$$

Two symbolic events η_1 and η_2 are said to be *disjoint* if their filter sets are disjoint, i.e. if $\Phi(\eta_1) \cap \Phi(\eta_2) = \emptyset$. For example, the events in example 1.3 are disjoint.

1.1.3 Labelled Transition Systems and μHML

A *labelled transition system* (LTS) is a triple $(\mathcal{S}, A \cup \{\tau\}, \rightarrow)$ where $\mathcal{S}$ is a set whose members are called *states*, A is a set of symbolic actions, $\tau \notin A$ denotes a distinguished *silent action*, and $\rightarrow$ is a subset of $\mathcal{S} \times (A \cup \{\tau\}) \times \mathcal{S}$, called the *transition relation* of the LTS. We call the elements of $\rightarrow$ *transitions* of the LTS, and write $s \xrightarrow{\nu} r$ instead of $(s, \nu, r) \in \rightarrow$.

If there are finite sequences $(s_1, \dots, s_n)$ and $(r_1, \dots, r_m)$ in $\mathcal{S}$ such that $s_i \xrightarrow{\tau} s_{i+1}$ for all $i \in \{1, \dots, n-1\}$, $s_n \xrightarrow{\alpha} r_1$, and $r_i \xrightarrow{\tau} r_{i+1}$ for all $i \in \{1, \dots, m-1\}$, then we write $s_1 \xRightarrow{\alpha} r_m$, which we call a *weak transition* of the LTS. Moreover, if (s_i) is a sequence of states and $\boldsymbol{\alpha} = (\alpha_i)$ is a sequence of actions such that $s_i \xRightarrow{\alpha_i} s_{i+1}$ for $i \in \{1, \dots, n-1\}$, we write $s_1 \xRightarrow{\boldsymbol{\alpha}} s_n$.

We consider a slightly generalised variant of the Hennessy-Milner logic with recursion (μHML) which is defined in definition 1.4. The definition assumes a countable set $\textsc{LVar}$ of logical variables ($X \in \textsc{LVar}$), and provides standard logical constructs such as truth, falsehood, conjunctions and disjunctions over finite indexing sets Γ, recursion using greatest/least fixed points, as well as necessity and possibility modal operators with symbolic events, where $\mathrm{bv}(p)$ binds free variables in c *and in* φ as well.

We interpret formulæ over the power set domain $\wp\mathcal{S}$ of the states in an LTS. The semantic definition of $[\![\varphi, \rho]\!]$ in definition 1.4 is given for both open and closed formulæ, employing a valuation $\rho\colon \textsc{LVar} \rightarrow \wp\mathcal{S}$ which permits an inductive definition of the structure of the formulæ.

Syntax

$$\begin{array}{rlllll}
\varphi, \psi \in \mu\mathrm{HML} ::= & \mathsf{tt} & \text{(truth)} & \mid & \mathsf{ff} & \text{(falsehood)} \\
\mid & \bigvee_{\gamma\in\Gamma} \varphi_\gamma & \text{(disjunction)} & \mid & \bigwedge_{\gamma\in\Gamma} \varphi_\gamma & \text{(conjunction)} \\
\mid & \langle\{p,c\}\rangle\, \varphi & \text{(possibility)} & \mid & [\{p,c\}]\, \varphi & \text{(necessity)} \\
\mid & \min X \,.\, \varphi & \text{(least f.p.)} & \mid & \max X \,.\, \varphi & \text{(greatest f.p.)} \\
\mid & X & \text{(f.p. variable)} & & &
\end{array}$$

Semantics

$$[\![\mathsf{tt}, \rho]\!] \stackrel{\mathsf{def}}{=} \mathcal{S} \qquad [\![\mathsf{ff}, \rho]\!] \stackrel{\mathsf{def}}{=} \emptyset \qquad [\![X, \rho]\!] \stackrel{\mathsf{def}}{=} \rho(X)$$

$$[\![\textstyle\bigvee_{\gamma\in\Gamma} \varphi_\gamma, \rho]\!] \stackrel{\mathsf{def}}{=} \bigcup_{\gamma\in\Gamma} [\![\varphi_\gamma, \rho]\!] \qquad [\![\textstyle\bigwedge_{\gamma\in\Gamma} \varphi_\gamma, \rho]\!] \stackrel{\mathsf{def}}{=} \bigcap_{\gamma\in\Gamma} [\![\varphi_\gamma, \rho]\!]$$

$$[\![\max X \,.\, \varphi, \rho]\!] \stackrel{\mathsf{def}}{=} \bigcup\{S \subseteq \mathcal{S} \mid S \subseteq [\![\varphi, \rho \cup \{X \mapsto S\}]\!]\}$$

$$[\![\min X \,.\, \varphi, \rho]\!] \stackrel{\mathsf{def}}{=} \bigcap\{S \subseteq \mathcal{S} \mid [\![\varphi, \rho \cup \{X \mapsto S\}]\!] \subseteq S\}$$

$$[\![\langle\{p,c\}\rangle\, \varphi, \rho]\!] \stackrel{\mathsf{def}}{=} \{s \in \mathcal{S} \mid \exists\, r \in \mathcal{S} \cdot \exists\, \alpha \in \Phi(\{p,c\}) \cdot (s \stackrel{\alpha}{\Longrightarrow} r \wedge r \in [\![\varphi\sigma, \rho]\!])\}$$

$$[\![[\{p,c\}]\, \varphi, \rho]\!] \stackrel{\mathsf{def}}{=} \{s \in \mathcal{S} \mid (\forall r \in \mathcal{S} \cdot \forall \alpha \in \Phi(\{p,c\}) \cdot s \stackrel{\alpha}{\Longrightarrow} r) \Rightarrow r \in [\![\varphi\sigma, \rho]\!]\}$$

Definition 1.4: The syntax and semantics for μHML

Symbolic actions of the form $\{p, \mathit{true}\}$ are relaxed notationally to p. In this case, we write $\langle p\rangle\varphi$ and $[p]\varphi$ for modal possibility and necessity respectively.

Generally we consider closed formulæ, and write $[\![\varphi]\!]$ instead of $[\![\varphi,\rho]\!]$, since the semantics of closed formulæ is independent of any valuation ρ. A system $s \in \mathcal{S}$ is said to *satisfy a formula* $\varphi \in \mu HML$ if $s \in [\![\varphi]\!]$. Conversely, a formula $\varphi \in \mu$HML is *satisfiable* if there exists a system $r \in \mathcal{S}$ such that $r[\![\varphi]\!]$.

1.2 Enforceability, sHML and Normal Form

1.2.1 The Enforceability of μHML

In [1], the authors describe the notion of a *transducer*, a device capable of *enforcing* formulæ in μHML. By "enforcing" we basically mean that the transducer m modifies the transitions of the system under scrutiny $s \in \mathcal{S}$ in the corresponding LTS to be in accordance with φ. This is done in such a way that $m[s]$ (the resulting monitored system) satisfies $m[s] \in [\![\varphi]\!]$ (*soundness*), but also without needlessly changing other systems which already satisfy φ (i.e. if $s \in [\![\varphi]\!]$, then $m[s] \sim s$.[1])

A transducer is also called an *enforcement monitor*.

Now we go to the notion of enforceability. A logic $\mathfrak{L}$ is said to be *enforceable* if for every formula $\varphi \in \mathfrak{L}$, there exists a transducer m such that m enforces φ.

For any reasonably expressive logic (such as μHML), one expects that not every formula is enforceable. Indeed, consider the formula

$$\varphi_{\mathrm{ns}} \stackrel{\mathsf{def}}{=} [i\,!\,v]\mathsf{ff} \vee [j\,!\,w]\mathsf{ff}.$$

A system satisfies φ_{ns}, either if it never produces the action $i\,!\,v$, or it never produces $j\,!\,w$. Now consider the systems

$$s_{\mathrm{ra}} \stackrel{\mathsf{def}}{=} i\,!\,v\,.\,\mathsf{nil} + j\,!\,w\,.\,\mathsf{nil} \qquad \text{and} \qquad s_{\mathrm{r}} \stackrel{\mathsf{def}}{=} i\,!\,v\,.\,\mathsf{nil}.$$

Clearly s_{ra} violates this property as it can produce both. This formula can only be enforced by suppressing or replacing either one of these actions. But doing so will needlessly suppress s_{r}'s actions, i.e., we would have $m[s_{\mathrm{r}}] \nsim s_{\mathrm{r}}$. Intuitively, the reason for this problem is that a monitor cannot "look into" the computation graph of a system, but is limited to the behaviour exhibited by a system at runtime.

[1]Here $\sim$ denotes some appropriate notion of equivalence, usually *bisimilarity*.[2]

$$\begin{array}{llllll}\varphi, \psi \in \mathrm{sHML} ::= \mathsf{tt} & (\text{truth}) & \mid \mathsf{ff} & (\text{falsehood}) \\ \qquad\qquad \mid \bigwedge_{\gamma\in\Gamma} \varphi_\gamma & (\text{conjunction}) & \mid [\{p, c\}]\, \varphi^\dagger & (\text{necessity}) \\ \qquad\qquad \mid \max X \,.\, \varphi & (\text{greatest f.p.}) & \mid X & (\text{f.p. variable})\end{array}$$

† If $\varphi = \mathsf{ff}$, then p must be an output pattern; i.e., $\mathrm{mt}(x\,!\,y, p)$ is defined.

Definition 1.5: The syntax for the safety fragment sHML

$$(\!|X|\!) \stackrel{\text{def}}{=} x \qquad (\!|\mathsf{tt}|\!) \stackrel{\text{def}}{=} (\!|\mathsf{ff}|\!) \stackrel{\text{def}}{=} \mathsf{id} \qquad (\!|\max X \,.\, \varphi|\!) \stackrel{\text{def}}{=} \mathsf{rec}\, x \,.\, (\!|\varphi|\!)$$

$$(\!|\textstyle\bigwedge_{\gamma\in\Gamma}[\{p_\gamma, c_\gamma\}]\varphi_\gamma|\!) \stackrel{\text{def}}{=} \mathsf{rec}\, y \,.\, \sum_{\gamma\in\Gamma} \begin{cases} \{p_\gamma, c_\gamma, \bullet\} & \text{if } \varphi_\gamma = \mathsf{ff} \\ \{p_\gamma, c_\gamma, \underline{p_\gamma}\}(\!|\varphi_\gamma|\!) & \text{otherwise} \end{cases}$$

Definition 1.6: Synthesis function for $\mathrm{sHML}_{\mathrm{nf}}$ formulæ.

1.2.2 The Safety Fragment and Normal Form

The safety fragment of μHML is a subset sHML $\subseteq \mu$HML which *is enforceable*. The definition of this restricted logic is given in definition 1.5.

Even though sHML is enforceable, complications still arise when attempting to define a synthesis function $(\!|\cdot|\!)\colon \mathrm{sHML} \to \textsc{Trn}$ which produces a transducer for any given sHML formula. This is discussed and exemplified in [1, sec. 5]. Although it is theoretically possible to define such a function directly, it is more straightforward to consider yet another subset, $\mathrm{sHML}_{\mathrm{nf}} \subseteq \mathrm{sHML}$ of formulæ in so-called *normal form*. This subset is only a superficial restriction of the logic. Indeed, any closed sHML formula φ can be transformed into an $\mathrm{sHML}_{\mathrm{nf}}$ formula φ' such that $[\![\varphi]\!] = [\![\varphi']\!]$. It is this process which we refer to as *normalisation*.

A formula $\varphi \in$ sHML is in normal form if:

(i) Branches in a conjunction are pairwise disjoint, i.e. in $\bigwedge_{\gamma\in\Gamma}[\{p_\gamma, c_\gamma\}]\varphi_\gamma$ we have $\Phi(\{p_{\gamma_1}, c_{\gamma_1}\}) \cap \Phi(\{p_{\gamma_2}, c_{\gamma_2}\}) = \emptyset$ for $\gamma_1 \neq \gamma_2$;

(ii) For every $\max X \,.\, \varphi$, we have $X \in \mathrm{fv}(\varphi)$;

(iii) Every logical variable is guarded by modal necessity.

If an sHML formula satisfies properties (i)–(iii), then it is in $\mathrm{sHML}_{\mathrm{nf}}$. An enforcement monitor for $\varphi \in \mathrm{sHML}_{\mathrm{nf}}$ can then be synthesised by the synthesis function defined in definition 1.6. More details about this function can be found in [1].

2 Parsing sHML in Haskell

A Haskell module `SHMLParser` was written to parse inputted sHML formulæ. This module made use of Haskell's `Parsec` combinators.

2.1 Parser Design

First, appropriate data structures were defined for sHML formulæ, which mirror definition 1.5, with the difference that conjunction is a purely binary operation. Next, a language structure for sHML was defined using the `LanguageDef` constructor. This assigns symbols to different tokens, e.g. `max`, `<=` and `==` are given special status when lexing.

Indeed, from the language constructor, the parsec package allows for the creation of "trivial" parsers, i.e. parsers which parse identifiers,[1] round brackets, square brackets, integers, special keywords from the language constructor, etc. These parsers can then be combined to form more sophisticated ones, e.g. to parse $\max X \,.\, \varphi$, the parser code is:

```haskell
maxFormula :: Parser Formula
maxFormula =
    do  keyword "max"
    x <- identifier
    op "."
    phi <- formulaTerm
    return $ Max x phi
```

This parser first reads the keyword "max", then an identifier stored in `x`, followed by the operator `.`, followed by something returned by the parser `formulaTerm`, defined in a similar way in terms of other parsers. Finally, the corresponding data structure is returned.

The parser is capable of parsing arithmetic and logic for symbolic actions such as $\{i \,?\, y, i \geqslant 4 \wedge y \neq 2+3\}$, but they have no defined semantics. In

[1] As usual, an identifier is a string matching `[a-Z]`$^+$(`[0-9]` | `[a-Z]` | `_`)*

general, binary operations associate to the left, so that `X&Y&Z` is parsed as $(X \wedge Y) \wedge Z$. Maximal fixed points take precedence over conjunction, so $\max X \,.\, \varphi \wedge \psi$ is interpreted as $(\max X \,.\, \varphi) \wedge \psi$. Whitespaces are ignored in formulæ.

2.2 Using the Parser

Here are some examples of formulæ and their syntactic equivalents in the parser language.

Formula	Syntax
$X \wedge Y \wedge Z$	`X&Y&Z` or `X & Y & Z`
$\max X \,.\, ([i?3]X \wedge [i\,!\,4]\mathsf{ff})$	`max X . ([i?3] X & [i!4]ff)`
$[\$i?\mathsf{req}][\{i\,!\,\mathsf{ans}, i < 3 \wedge i \neq 10\}]\mathsf{ff}$	`[$i?req][i!ans,i<3 & i!=10]ff`

To parse a formula, the function `parseF` :: `String` → `Formula` is used. For example, running `parseF` "`[i?3][i!4][i?5]max X . [i!6]ff`" will return the formula, displaying it using the defined instance of `Show`.

Another nice command is the `parseTree` :: `Formula` → `IO()` command (or for string input, `stringParseTree` :: `String` → `IO()`), which displays a visual parse tree of the formula data structure. For example, running `stringParseTree` on the string

"`[$i?3][$j?5, j>7 & j+1!=i]max X0 . ([i!6]ff & [j!2]X0)`"

produces the tree illustrated in figure 2.1.

```
Necessity
└─ Input
   └─ i (binding variable)
   └─ 3 (int const)
└─ True (bool const)
└─ Necessity
   └─ Input
      └─ j (binding variable)
      └─ 5 (int const)
   └─ ∧
      └─ >
         └─ j (free variable)
         └─ 7 (int const)
      └─ ≠
         └─ +
            └─ j (free variable)
            └─ 1 (int const)
         └─ i (free variable)
   └─ max X0 .
      └─ ∧
         └─ Necessity
            └─ Output
               └─ i (free variable)
               └─ 6 (int const)
            └─ True (bool const)
            └─ FF
         └─ Necessity
            └─ Output
               └─ j (free variable)
               └─ 2 (int const)
            └─ True (bool const)
            └─ X0 (logical variable)
```

Figure 2.1: Example of a `parseTree` output

3

The Normalisation Algorithm

The reduction of sHML formulæ to normal form is carried out in a series of six steps presented in [1], corresponding to each of the following sections.

§3.1. Preliminary Minimisation.
Well known logical equivalence rules are applied to simplify and reduce the size of the formula as much as possible. This includes rules such as $[\![\mathsf{tt} \wedge \varphi]\!] = [\![\varphi]\!]$ and $[\![\max X \,.\, X]\!] = [\![\mathsf{tt}]\!]$.

§3.2. Unguarded fixed point variable removal.
At this stage, the formula is modified to ensure that fixed point variables are all guarded.

§3.3. System of Equations.
The formula is reformulated into a system of equations to ease manipulation in further stages.

§3.4. Power set Construction.
The resultant system is restructured into an equivalent system that ensures that patterns in conjunctions are disjoint.

§3.5. Formula reconstruction.
The system of equations is converted back into an sHML formula with disjoint conjunctions, which may introduce redundant fixed points.

§3.6. Redundant fixed point removal.
Any redundant fixed points from the previous stage are removed, leaving us with the required sHML_{nf} formula.

3.1 Preliminary Minimisation

The function `simplify` :: `Formula` → `Formula` was written to carry out the preliminary minimisation of sHML formulæ.

The simplification of conjunctions required particular care. Indeed, when defining `simplify` case by case, one might naïvely do the following for the conjunction case:

$$\texttt{simplify}(a \wedge b) \stackrel{\text{def}}{=} \texttt{simplify}(a) \wedge \texttt{simplify}(b)$$

But this definition would simplify $(\mathsf{tt} \wedge \mathsf{tt}) \wedge (\mathsf{tt} \wedge \mathsf{tt})$ to $\mathsf{tt} \wedge \mathsf{tt}$, not to tt. The correct approach is to simplify the two children of the $\wedge$ node (picturing the formula as a parse tree), and then to use another function, `simplifyCon` :: `Formula` → `Formula` → `Formula`, which simplifies conjunctions, i.e. we define

$$\texttt{simplify}(a \wedge b) \stackrel{\text{def}}{=} \texttt{simplifyCon}(\texttt{simplify}(a))(\texttt{simplify}(b)),$$

and then

$$\begin{aligned}
\texttt{simplifyCon}(\mathsf{ff})(\varphi) &\stackrel{\text{def}}{=} \mathsf{ff} \\
\texttt{simplifyCon}(\varphi)(\mathsf{ff}) &\stackrel{\text{def}}{=} \mathsf{ff} \\
\texttt{simplifyCon}(\mathsf{tt})(\varphi) &\stackrel{\text{def}}{=} \varphi \\
\texttt{simplifyCon}(\varphi)(\mathsf{tt}) &\stackrel{\text{def}}{=} \varphi \\
\texttt{simplifyCon}(\varphi)(\psi) &\stackrel{\text{def}}{=} \begin{cases} \varphi & \text{if } \varphi = \psi \\ \varphi \wedge \psi & \text{otherwise.} \end{cases}
\end{aligned}$$

Similarly for maximum fixed points, we considered that $[\![\max X \,.\, X]\!] = [\![\mathsf{tt}]\!]$, so we first simplify the subtree and then do `simplifyMax`:

$$\texttt{simplify}(\max X \,.\, \varphi) \stackrel{\text{def}}{=} \texttt{simplifyMax}(X)(\texttt{simplify}(\varphi)),$$

where

$$\begin{aligned}
\texttt{simplifyMax}(X)(\mathsf{tt}) &\stackrel{\text{def}}{=} \mathsf{tt} \\
\texttt{simplifyMax}(X)(\mathsf{ff}) &\stackrel{\text{def}}{=} \mathsf{ff} \\
\texttt{simplifyMax}(X)(X) &\stackrel{\text{def}}{=} \mathsf{tt} \\
\texttt{simplifyMax}(X)(X \wedge \varphi) &\stackrel{\text{def}}{=} \texttt{simplify}(\max X \,.\, \varphi) \\
\texttt{simplifyMax}(X)(\varphi \wedge X) &\stackrel{\text{def}}{=} \texttt{simplify}(\max X \,.\, \varphi) \\
\texttt{simplifyMax}(X)(\varphi) &\stackrel{\text{def}}{=} \max X \,.\, \varphi.
\end{aligned}$$

If the simplifying of the subtree is not carried out first, things like $\max X \,.\, ((X \wedge X) \wedge (X \wedge X))$ do not simplify correctly.

Simplification of the remaining cases was straightforward.

3.2 Standard Form

An sHML formula is said to be in *standard form* if all free and unguarded recursion variables are at the top-most level, at every level. For example, the formula

$$\max Y \,.\, ([i\,?\,3]Y \wedge X) \wedge [i\,?\,3]\mathsf{ff}$$

is not in standard form, since X is unguarded but is not at the top most level. We can easily mitigate this by elevating X:

$$\max Y \,.\, [i\,?\,3]Y \wedge [i\,?\,3]\mathsf{ff} \wedge X.$$

In definition 3.1, we present the construction $\langle\!\langle \cdot \rangle\!\rangle_1 \colon \text{sHML} \to \text{sHML}$ which carries out this standardisation reasoning. This is a slightly modified version of the construction presented in [3, ch. 4] which is easier to implement in Haskell.

$$\begin{aligned}
\langle\!\langle \max X \,.\, \varphi \rangle\!\rangle_1 &\stackrel{\text{def}}{=} \mathfrak{Bg}(\varphi)[{}^{\max X \,.\, \mathfrak{Bg}(\varphi)}/X] \wedge \bigwedge (\mathfrak{Fu}(\varphi) \smallsetminus \{X\}) \\
\langle\!\langle \varphi \wedge \psi \rangle\!\rangle_1 &\stackrel{\text{def}}{=} \mathfrak{Bg}(\varphi) \wedge \mathfrak{Bg}(\psi) \wedge \bigwedge \mathfrak{Fu}(\varphi) \cup \mathfrak{Fu}(\psi) \\
\langle\!\langle [\{p, c\}]\varphi \rangle\!\rangle_1 &\stackrel{\text{def}}{=} [\{p, c\}]\langle\!\langle \varphi \rangle\!\rangle_1 \\
\langle\!\langle \varphi \rangle\!\rangle_1 &\stackrel{\text{def}}{=} \varphi
\end{aligned}$$

where $\mathfrak{Fu}(\varphi)$ denotes the set of free and unguarded logical variables in φ, i.e. $\mathfrak{Fu}(\varphi) \stackrel{\text{def}}{=} \{X \in \text{fv}(\varphi) \mid X \text{ is unguarded}\}$, and $\mathfrak{Bg}(\varphi)$ denotes the remaining bound and guarded part of a formula after $\langle\!\langle \cdot \rangle\!\rangle_1$ is applied; i.e. if $\langle\!\langle \varphi \rangle\!\rangle_1 = \psi \wedge \bigwedge \mathfrak{Fu}(\varphi)$, then $\mathfrak{Bg}(\varphi) = \psi$.

Definition 3.1: Standardisation of sHML formulæ.

Notice that in the case of maximum fixed points, definition 3.1 unfolds the bound logical variable X. This ensures that the resulting conjuncted branches are always guarded by a necessity operation. For example, applying definition 3.1 to the formula

$$\max Y \,.\, ([i\,?\,3]Y \wedge X) \wedge [i\,?\,3]\mathsf{ff},$$

noting that $\mathfrak{Bg}([i\,?\,3]Y \wedge X) = [i\,?\,3]Y$, yields

$$\begin{aligned}
&([i\,?\,3]Y)[{}^{\max Y \,.\, [i\,?\,3]Y}/Y] \wedge [i\,?\,3]\mathsf{ff} \wedge X \\
&= [i\,?\,3] \max Y \,.\, [i\,?\,3]Y \wedge [i\,?\,3]\mathsf{ff} \wedge X.
\end{aligned}$$

To implement this, first, a function `sub` $::$ `Formula` $\to$ `String` $\to$ `Formula` $\to$ `Formula` was implemented to carry out substitution of free logical variables. The substitution $\varphi[{}^{\psi}/X]$ is equivalent to `sub`$(\varphi)(X)(\psi)$. Next, a function `sf'` :: `Formula` $\to$ [`String`] $\to$ (`Formula`, [`String`]) was defined. This function "takes out" free variables out of a given formula by replacing them with tt in the manner illustrated below. The second argument is to keep track of bound variables when traversing subtrees, allowing for recursive definition of `sf'`.

Examples 3.2. The following few examples illustrate the behaviour of the function `sf'` :: `Formula` $\to$ [`String`] $\to$ (`Formula`, [`String`]).

$$\begin{aligned}
\texttt{sf1'}(X)([\,]) &= (\mathsf{tt}, [X]) \\
\texttt{sf1'}(X \wedge Y)([\,]) &= (\mathsf{tt} \wedge \mathsf{tt}, [X, Y]) \\
\texttt{sf1'}(\max X \,.\, (X \wedge Y))([\,]) &= (\max X \,.\, (X \wedge \mathsf{tt}) \wedge \mathsf{tt}, [Y]) \\
\texttt{sf1'}(\max X \,.\, (X \wedge [i \mathbin{?} 3]Y))([\,]) &= (\max X \,.\, (X \wedge [i \mathbin{?} 3]Y) \wedge [i \mathbin{?} 3]Y, [\,]) \\
\texttt{sf1'}(X \wedge (Y \wedge Z))([Y]) &= (\mathsf{tt} \wedge (Y \wedge \mathsf{tt}), [X, Z])
\end{aligned}$$

The last example illustrates the purpose of the second argument: if the expression $X \wedge (Y \wedge Z)$ appears in a subtree of a larger expression, it is possible that it is preceded by a binder (say $\max Y\,.\,$). In that case, Y should not be "taken out".

The actual implementation of the function is straightforward and faithfully mirrors definition 3.1—the reader is invited to glance at the code in appendix A. Now `sf'` itself does not give us a `Formula`, but a pair of type (`Formula`, [`String`]). So we define a function `sf` :: `Formula` $\to$ `Formula` which simply runs `sf'`$(\varphi)([\,])$, appends the variables in the list to the end of the resulting formula with conjunctions, and invokes `simplify` to remove all the redundant tt's.

A proof that the $\langle\!\langle \cdot \rangle\!\rangle_1$ preserves semantics, i.e. that for all $\varphi \in \mathrm{sHML}$, $[\![\langle\!\langle \varphi \rangle\!\rangle_1]\!] = [\![\varphi]\!]$, is given as lemma 8 in [4].

3.3 System of Equations

A *system of equations* is a triple $(\mathscr{E}, X, \mathscr{F})$ where X is the *principal logical variable* which defines the starting equation, $\mathscr{F}$ is a finite set of *free logical variables*, and $\mathscr{E}$ is an tuple of equations $(X_1 = \varphi_1, \ldots, X_n = \varphi_n)$ where $X_i \neq X_j$ for $i \neq j$, and $\varphi_i \in \mathrm{sHML}_{\mathrm{eq}}$ (see definition 3.3).

$$\varphi \in \mathrm{sHML}_{\mathrm{eq}} ::= \mathsf{ff} \quad | \quad \textstyle\bigwedge_{\gamma \in \Gamma} [\eta_\gamma] X_\gamma$$

where Γ is a finite indexing set such that for all $\gamma \in \Gamma$, $\eta_\gamma \in \textsc{Patt}$ and $X_\gamma \in \mathrm{LV}\textsc{ar}$.

Definition 3.3: The syntactic restriction for equations.

$$
\begin{aligned}
\langle\!\langle \mathsf{tt} \rangle\!\rangle_2 &\stackrel{\text{def}}{=} (\{X_i = \mathsf{tt}\}, X_i, \emptyset) \\
\langle\!\langle \mathsf{ff} \rangle\!\rangle_2 &\stackrel{\text{def}}{=} (\{X_i = \mathsf{ff}\}, X_i, \emptyset) \\
\langle\!\langle Y \rangle\!\rangle_2 &\stackrel{\text{def}}{=} (\{X_i = Y\}, X_i, \{Y\}) \\
\langle\!\langle \varphi \wedge \psi \rangle\!\rangle_2 &\stackrel{\text{def}}{=} (\mathscr{E}_\varphi \cup \mathscr{E}_\psi \cup \{X_i = \mathscr{E}_\varphi(X_\varphi) \cup \mathscr{E}_\psi(X_\psi)\}, X_i, \mathscr{F}_\varphi \cup \mathscr{F}_\psi) \\
\langle\!\langle [\eta]\varphi \rangle\!\rangle_2 &\stackrel{\text{def}}{=} (\mathscr{E}_\varphi \cup \{X_i = [\eta]X_\varphi\}, X_i, \mathscr{F}_\varphi) \\
\langle\!\langle \max Y \,.\, \varphi \rangle\!\rangle_2 &\stackrel{\text{def}}{=} (\mathscr{E}_{\varphi'} \cup \{X_i = \mathscr{E}_{\varphi'}(X_{\varphi'})\}, X_i, \mathscr{F}_{\varphi'} \setminus \{X_i\})
\end{aligned}
$$

where $\langle\!\langle \vartheta \rangle\!\rangle_2 = (\mathscr{E}_\vartheta, X_\vartheta, \mathscr{F}_\vartheta)$ for all ϑ, φ' denotes $\varphi[{}^{X_i}/_{Y}]$, and X_i is a fresh variable.

Definition 3.5: Conversion from sHML formula to a system of equations.

Through equations, maximal fixed points can be expressed by referring to previously defined variables. We abuse notation and use $\mathscr{E}$ as a map $\mathscr{E} \colon \text{LVar} \to \text{sHML}_{\text{eq}}$ so that if $(X_i = \varphi_i) \in \mathscr{E}$, then $\mathscr{E}(X_i) = \varphi_i$.

Example 3.4. The formula $\varphi = \max X \,.\, [i \,?\, 3]([i \,!\, 4]X \wedge [i \,!\, 5]\mathsf{ff})$ can be represented by the equations

$$
\begin{aligned}
X_0 &= [i \,?\, 3]X_1 \\
X_1 &= [i \,!\, 4]X_2 \wedge [i \,!\, 5]X_3 \\
X_2 &= [i \,?\, 3]X_1 \qquad (= X_0) \\
X_3 &= \mathsf{ff}
\end{aligned}
$$

where X_0 is the principal variable, and $\mathscr{F} = \emptyset$, as no variable in the equations is free.

The conversion into a system of equations is defined by the construction $\langle\!\langle \cdot \rangle\!\rangle_2 \colon \text{sHML} \to (\mathscr{E}, \text{Var}, \wp\text{Var})$ in definition 3.5. Again, this is a slightly modified version from [3, 1] which more Haskell-friendly.

Since variables are being introduced, we want to make sure that no capturing occurs. Thus a function `rename` :: `Formula` → (`Formula`, [(`Int`, `String`)]) was implemented to rename all variables to successive natural numbers, e.g.

$$
\begin{aligned}
&\mathtt{rename}(\max X \,.\, [i \,?\, 3](X \wedge Y) \wedge Z) \\
&= (\max(0 \,.\, [i?3]0 \wedge 1) \wedge 2, [(0, X), (1, Y), (2, Z)].
\end{aligned}
$$

Variable capturing is guaranteed not to happen during intermediate stages of `rename`'s execution, since the user is prohibited from using integers as variable names. The implementation of this function is straightforward.

The system of equations is generated is as follows. First, the type synonyms `Equation` $\stackrel{\text{def}}{=}$ (`String`, `Formula`) and `SoE` $\stackrel{\text{def}}{=}$ ([`Equation`], `String`, [`String`]) are introduced to simplify the code legibility, where $X = \varphi$ is encoded as the `Equation` (“X”, φ), and $(\mathscr{E}, X, \mathscr{F})$ is encoded naturally as an `SoE`. A function `SysEq'` :: `Int` $\to$ `Formula` $\to$ `SoE` is then defined to implement definition 3.5, where the variables are named `X0`, `X1`, ... The integer argument of `SysEq'` is the index of the first variable it is allowed to introduce. One of the simple cases is

$$\texttt{SysEq'}(n)(\texttt{tt}) = ([\texttt{Xn} = \texttt{tt}], \texttt{Xn}, [\,]).$$

One of the cases which required more care (mainly for variable indices) was the conjunction. This was defined as follows:

$$\texttt{SysEq'}(n)(\varphi \wedge \psi) = ([\texttt{Xn} = \mathscr{E}_1(\texttt{Xm}) \wedge \mathscr{E}_2(\texttt{Xt})] +\!\!+ \mathscr{E}_1 +\!\!+ \mathscr{E}_2, \texttt{Xn}, \mathscr{F}_1 +\!\!+ \mathscr{F}_2),$$

where $(\mathscr{E}_1, \texttt{Xm}, \texttt{F}_1) = \texttt{SysEq'}(n{+}1)(\varphi)$ and $(\mathscr{E}_2, \texttt{Xt}, \texttt{F}_2) = \texttt{SysEq'}(t)(\varphi)$, where t is one more than the index of the last variable in $\mathscr{E}_1$ (obtained in Haskell using various functions on lists, such as head, snd, etc.). The reasoning for other cases was similar.

Finally, a function `SysEq` :: `Formula` $\to$ (`SoE`, [`Int`, `String`]) was defined. This carries out `rename` followed by `SysEq'` starting from 0. The function then returns the system, together with the list of correspondences with the original variable names provided by `rename`.

As in the previous stage, a proof that the $\langle\!\langle\, \cdot \,\rangle\!\rangle_2$ preserves semantics, i.e. that for all $\varphi \in$ sHML, $[\![\langle\!\langle \varphi \rangle\!\rangle_2]\!] = [\![\varphi]\!]$, is given as lemma 10 in [4].

Example 3.6. Consider $\varphi = \max X \,.\, [i \,?\, \mathsf{req}]([i \,!\, \mathsf{ans}][i \,!\, \mathsf{ans}]\mathsf{ff} \wedge [i \,!\, \mathsf{ans}]X)$. Running (`sysEq . sf`) on φ produces the following output:

```
(([(“X0”, [i ? req]X1), (“X1”, [i ! ans]X3 & [i ? ans]X6),
   (“X2”, [i ! ans]X3), (“X3”, [i ! ans]X4), (“X4”, ff),
   (“X5”, [i ? ans]X6), (“X6”, [i ? req]X8), (“X7”, [i ? req]X8),
   (“X8”, [i ! ans]X10 & [i ? ans]X13), (“X9”, [i ! ans]X10),
   (“X10”, [i ! ans]X11), (“X11”, ff), (“X12”, [i ? ans]X13),
   (“X13”, [i ? req]X8)], “X0”, []), [(0,“X”)])
```

Or in a more legible typeface:

$$\begin{aligned}
X_0 &= [i\,?\,\mathsf{req}]X_1 & X_7 &= [i\,?\,\mathsf{req}]X_8 \\
X_1 &= [i\,!\,\mathsf{ans}]X_3 \wedge [i\,?\,\mathsf{ans}]X_6 & X_8 &= [i\,!\,\mathsf{ans}]X_{10} \wedge [i\,?\,\mathsf{ans}]X_{13} \\
X_2 &= [i\,!\,\mathsf{ans}]X_3 & X_9 &= [i\,!\,\mathsf{ans}]X_{10} \\
X_3 &= [i\,!\,\mathsf{ans}]X_4 & X_{10} &= [i\,!\,\mathsf{ans}]X_{11} \\
X_4 &= \mathsf{ff} & X_{11} &= \mathsf{ff} \\
X_5 &= [i\,?\,\mathsf{ans}]X_6 & X_{12} &= [i\,?\,\mathsf{ans}]X_{13} \\
X_6 &= [i\,?\,\mathsf{req}]X_8 & X_{13} &= [i\,?\,\mathsf{req}]X_8 \qquad (= X_6)
\end{aligned}$$

The greyed out formulæ are not reachable from X_0 and are hence redundant.

3.4 Power Set Construction

Next, we present the power set construction $\langle\!\langle \cdot \rangle\!\rangle_3$. Here the implementation does not mirror the theoretical construction so closely, unlike in the previous sections.

The previous section ensured that requirement (iii) in the definition of $\mathrm{sHML}_{\mathrm{nf}}$ (see section 1.2.2) is met. The goal here is to ensure the first property (i) is adhered to, i.e. that branches in conjunctions are pairwise disjoint.

Consider a system of equations $(\mathscr{E}, X, \mathscr{F})$ where $\mathscr{E}$ contains $n+1$ equations, i.e. $\mathscr{E} = \{X_0 = \varphi_0, \ldots, X_n = \varphi_n\}$. The idea of the construction is to introduce new variables $X_{\{0\}}, \ldots, X_{\{0,\ldots,n\}}$, indexed by the power set $\Gamma = \wp\{0, \ldots n\}$, such that for all $\gamma \in \Gamma$,

$$X_\gamma = \bigwedge_{i \in \gamma} \varphi_i,$$

where we identify any variables X_j appearing in φ_i with $X_{\{j\}}$. (Indeed, by this definition, $X_{\{j\}} = \varphi_j = X_j$.) After these equations are constructed, any common symbolic actions are factored out, e.g. if $X_{\{0,1\}} = [i\,?\,3]X_2 \wedge [i\,!\,3]X_3 \wedge [i\,?\,3]X_4$, then we instead take

$$X_{\{0,1\}} = [i\,?\,3]X_{\{2,4\}} \wedge [i\,!\,3]X_{\{3\}}.$$

This way, all the symbolic actions are (syntactically) disjoint.[1]

The way this construction is formally presented in [1, 3] mainly hinges on subsets of Γ. In definition 3.7, we present an equivalent definition of $\langle\!\langle \cdot \rangle\!\rangle_3$ which is more indicative of the Haskell implementation.

Indeed, first a few straightforward functions were implemented to aid with manipulation of subsets and variable indices. The first one is `nsubsets` :: `Eq a` $\Rightarrow$ `[a]` $\rightarrow$ `[[a]]`, which generates all non-empty sublists of a given list ℓ, such that the first $|\ell|$ members are the singletons, followed by the remaining sublists in lexicographical order. For example:

$$\begin{aligned}\texttt{nsubsets}([1,2,3,4]) = [&[1],[2],[3],[4],[1,2],[1,3],[2,3],[1,2,3],[4],\\ &[1,4],[2,4],[1,2,4],[3,4],[1,3,4],[2,3,4]].\end{aligned}$$

It is not important that the remaining sublists are in lexicographical order, this is simply a consequence of the inbuilt function `subsequences` which Haskell provides. It *is* important however that the singletons come first; this way, if $(\mathscr{E}, X, \mathscr{F})$ has $|\mathscr{E}| = n$ variables, then we associate X_i with $X_{\{i\}}$

[1] We assume for now that if $\eta_1 \neq \eta_2$, then $\Phi(\eta_1) \cap \Phi(\eta_2) = \emptyset$.

$$\langle\!\langle(\mathscr{E}, X_i, \mathscr{F})\rangle\!\rangle_3 \stackrel{\text{def}}{=} \langle\!\langle(\{X_\gamma = \textstyle\bigwedge_{\eta\in E(\gamma)}\big([\eta]\bigwedge f_\gamma(\eta)\big) \mid \gamma \in \wp|\mathscr{E}|\}, X_{\{i\}}, \mathscr{F})\rangle\!\rangle$$

where $E(\gamma)$ is the set of symbolic events appearing in the equations $X_j = \mathscr{E}(X_j)$ for $j \in \gamma$, i.e.

$$E(\gamma) \stackrel{\text{def}}{=} \bigcup_{j\in\gamma} \text{sas}\big(\mathscr{E}(X_j)\big),$$

$\text{sas}(\varphi) \subseteq \text{SE}\textsc{vt}$ is the set of symbolic actions appearing in φ, defined by

$$\begin{aligned} \text{sas}([\eta]\varphi) &\stackrel{\text{def}}{=} \{\eta\} \cup \text{sas}(\varphi) \\ \text{sas}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \text{sas}(\varphi) \cup \text{sas}(\psi) \\ \text{sas}(\varphi) &\stackrel{\text{def}}{=} \emptyset, \end{aligned}$$

$f_\gamma(\eta)$ is the set of all logical variables guarded by η in the equations $X_j = \mathscr{E}(X_j)$ for $j \in \gamma$, i.e.

$$f_\gamma(\eta) \stackrel{\text{def}}{=} \bigcup_{j\in\gamma} \text{savars}(\eta)\big(\mathscr{E}(X_j)\big),$$

and savars: $\text{SE}\textsc{vt} \to \text{sHML} \to \wp\,\text{LV}\textsc{ar}$ gives all the logical variables in a formula φ guarded by a particular symbolic event η, defined by

$$\begin{aligned} \text{savars}(\eta)([\nu]\varphi) &\stackrel{\text{def}}{=} \begin{cases} \{\eta\} \cup \text{savars}(\varphi) & \text{if } \eta = \nu \\ \quad\text{savars}(\varphi) & \text{otherwise} \end{cases} \\ \text{savars}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \text{savars}(\varphi) \cup \text{sas}(\psi) \\ \text{savars}(\varphi) &\stackrel{\text{def}}{=} \emptyset. \end{aligned}$$

Definition 3.7: The power set construction for systems of equations.

for $0 \leqslant i \leqslant n-1$, and X_i with X_{I_i}, where $I_i \subseteq \{0,\ldots,n-1\}$ is the corresponding ith sublist in `nsubsets`$([0,\ldots,n-1])$ for $i \geqslant n$.

The functions `subIdx` :: `Int` → `Int` → [`Int`] and `idxSub` :: `Int` → [`Int`] → `Int` give the corresponding subset I_i for given i of $\{0,\ldots,n-1\}$, and vice-versa. For example,

$$\texttt{subIdx}(5)(12) = [2,3] \qquad \text{and} \qquad \texttt{idxSub}(5)([2,3]) = 12.$$

These allowed us to switch back and forth between the variables indexed by subsets and by integral indices, which is what the resulting system of equations has.

Next the function `sas` :: `Formula` → [(`Patt`, `BExpr`)] was defined, which produces a list of pairs (p,c) corresponding to each symbolic event $\{p,c\}$ which occurs in a given formula. The implementation is straightforward by pattern matching, identical to $\mathrm{sas}(\varphi)$ in definition 3.7.

The important function is `factor` :: `Int` → `Equation` → `Equation`, which carries out the "factorisation" of common patterns in a given formula φ. Using list comprehension and `sas`, the list `saVarPairs` is constructed, consisting of pairs of type ((`Patt`, `BExpr`), [`String`]) where all variables guarded by the same pattern are placed in the list. This corresponds to the function savars in definition 3.7. For example, if

$$X_0 = [i\,?\,3]X_1 \wedge [\{i\,!\,k, k \geqslant 2\}]X_2 \wedge [i\,?\,3]X_3,$$

then `saVarPairs` would be $[((i?3,\mathtt{tt}),[X_1,X_3],((i!k,k\geqslant 2),[X_2]))]$. Followed by further manipulation and a left fold, this list is transformed into

$$[i\,?\,3](X_j) \wedge [\{i\,!\,k, k \geqslant 2\}]X_2,$$

where $j = \texttt{idxSub}(n)([1,3])$, the subscript corresponding to the variable identified with $X_{\{1,3\}}$ and n is the number of equations in the system where this equation resides, since this subscript depends on n (and this is why the first argument is an `Int`).

Finally, the function `norm` which carries out the normalisation itself first builds the corresponding new set of equations using `nsubsets` and a left fold with $\wedge$, and `zips` this with $\{X_0,\ldots,X_{2^n-2}\}$. Since the first subsets are $\{0\}$, $\ldots$, $\{n\}$, then the first n equations correctly correspond with the subscripts, and no labels subscripts need to be changed in the right-hand side of any of the equations. Then, the `factor` function is applied to each equation via `map`.

The preservation of semantics for the power set construction is given as lemma 11 in [4].

Example 3.8. Let φ be as in example 3.6, i.e.

$$\varphi = \max X \,.\, [i \mathbin{?} \mathsf{req}]([i \mathbin{!} \mathsf{ans}][i \mathbin{!} \mathsf{ans}]\mathsf{ff} \wedge [i \mathbin{!} \mathsf{ans}]X).$$

Running (`norm . sysEq`) produces a set of 254 equations, where the only reachable ones from X_0 are

$$\begin{aligned} X_0 &= [i \mathbin{?} \mathsf{req}]X_2 & X_2 &= [i \mathbin{?} \mathsf{ans}]X_{143} \\ X_5 &= \mathsf{ff} & X_{143} &= [i \mathbin{?} \mathsf{ans}]X_5 \wedge [i \mathbin{?} \mathsf{req}]X_2 \end{aligned}$$

Notice that all the necessity operations are disjoint, in particular thanks to the equation for X_2, which comes from $X_2 = [i \mathbin{?} \mathsf{ans}]X_4 \wedge [i \mathbin{?} \mathsf{ans}]X_7$ in the un-normalised system (i.e. if we do `sysEq` alone on φ). The index 143 corresponds to `idxSub`$(8)([4,7])$, where 8 is the number of equations in the un-normalised the system.

3.5 Formula Reconstruction

Now we reconstruct a single formula from the normalised set of equations. The idea is to recurse through the equations using maximal fixed points, until a term with no free variables is encountered.

$$\sigma_{\mathsf{shml}}(\varphi, \mathscr{E}) \overset{\text{def}}{=} \begin{cases} \varphi & \text{if } \mathrm{fv}(\varphi) = \emptyset \\ \sigma_{\mathsf{shml}}(\varphi\sigma, \mathscr{E}) & \text{otherwise,} \end{cases}$$

where $\sigma \overset{\text{def}}{=} \{ {}^{\max X_i \,\cdot\, \mathscr{E}(X_i)}/{X_i} \mid X_i \in \mathrm{fv}(\varphi) \}$.

Definition 3.9: Converting a system of equations into a single formula.

This is achieved through the map $\sigma_{\mathsf{shml}} \colon \mathrm{sHML} \rightharpoonup \mathrm{sHML}$ in definition 3.9. The construction $\langle\!\langle \cdot \rangle\!\rangle_4$ is then defined as $\langle\!\langle (\mathscr{E}, X, \mathscr{F}) \rangle\!\rangle_4 \overset{\text{def}}{=} \sigma_{\mathsf{shml}}(X, \mathscr{E})$. Thus σ_{shml} starts from the formula $\varphi = X$, which has $X \in \mathrm{fv}(\varphi)$, and thus looks up $\mathscr{E}(X)$ and then does $\sigma_{\mathsf{shml}}(X[{}^{\max X \,\cdot\, \mathscr{E}(X)}/X], \mathscr{E})$, and continues to recurse until a formula with $\mathrm{fv}(\varphi) = \emptyset$ is encountered.

Example 3.10. Consider the normalised system of equations

$$\begin{aligned} X_0 &= [i \mathbin{?} \mathsf{req}]X_2 & X_2 &= [i \mathbin{?} \mathsf{ans}]X_{143} \\ X_5 &= \mathsf{ff} & X_{143} &= [i \mathbin{?} \mathsf{ans}]X_5 \wedge [i \mathbin{?} \mathsf{req}]X_2 \end{aligned}$$

from example 3.8.

Applying the construction to this set of equations yields the formula

$$\max X_0 \cdot [i?\mathsf{req}](\max X_2 \,.\, [i!\mathsf{ans}](\max X_{143} \,.\, ([i!\mathsf{ans}](\max X_5 \,.\, \mathsf{ff}) \wedge [i?\mathsf{req}]X_2)))$$

$$\begin{aligned}
\langle\!\langle \max X\,.\,\varphi\rangle\!\rangle_5 &\overset{\text{def}}{=} \begin{cases} \max X\,.\,\langle\!\langle\varphi\rangle\!\rangle_5 & \text{if } X \in \mathrm{fv}(\varphi) \\ \langle\!\langle\varphi\rangle\!\rangle_5 & \text{otherwise}\end{cases} \\
\langle\!\langle \varphi\wedge\psi\rangle\!\rangle_5 &\overset{\text{def}}{=} \langle\!\langle\varphi\rangle\!\rangle_5 \wedge \langle\!\langle\psi\rangle\!\rangle_5 \\
\langle\!\langle [\eta]\varphi\rangle\!\rangle_5 &\overset{\text{def}}{=} [\eta]\langle\!\langle\varphi\rangle\!\rangle_5 \\
\langle\!\langle\varphi\rangle\!\rangle_5 &\overset{\text{def}}{=} \varphi
\end{aligned}$$

Definition 3.12: Removing redundant fixed points to obtain a formula in sHML$_{\mathrm{nf}}$.

The implementation `sigmaSHML` of σ_{shml} is straightforward, mirroring the definition. For substitutions, we use set comprehension and the function `sub` defined in section 3.2 to build a list of substitutions which is then folded with $\circ$, i.e. function composition.

The function `reconstruct` is then defined in terms of `sigmaSHML` as described previously. At this stage, any free variables which were renamed as integers in section 3.3 are given back their original names using the function `replace`.

The proof that $\langle\!\langle\cdot\rangle\!\rangle_4$ preserves semantics is given as lemma 12 in [4].

3.6 Redundant Fixed Point Removal

As seen in example 3.10, the reconstruction of a formula may give rise to redundant fixed points. This violates the requirement (ii) for sHML$_{\mathrm{nf}}$. Thus the final stage is simply to determine which fixed points are redundant and to remove them.

The definition of the construction $\langle\!\langle\cdot\rangle\!\rangle_5$ is intuitive, see definition 3.12. This is implemented as the function `redfix :: Formula → Formula`. The proof that $\langle\!\langle\cdot\rangle\!\rangle_5$ preserves semantics is given in appendix A.1 of [1].

Example 3.11. Take the resulting formula

$\max X_0 \cdot [i?\mathsf{req}](\max X_2\,.\,[i!\mathsf{ans}](\max X_{143}\,.\,([i!\mathsf{ans}](\max X_5\,.\,\mathsf{ff})\wedge[i?\mathsf{req}]X_2)))$

from example 3.10. Applying `redfix` to this formula yields

$$[i\,?\,\mathsf{req}](\max X_2\,.\,[i\,!\,\mathsf{ans}]([i\,!\,\mathsf{ans}]\mathsf{ff}\wedge[i\,?\,\mathsf{req}]X_2)) \in \text{sHML}_{\mathrm{nf}}.$$

4 Conclusion

The six stages outlined in the previous chapter convert an arbitrary closed sHML formula into one in sHML_{nf}. Indeed, the stages §3.4, §3.6 and §3.3 ensure that (i), (ii) and (iii) in section 1.2.2 hold respectively.

The last function in the Normaliser module is the function `nf` :: `Formula` → `Formula`, whose definition is done in one line:

```
nf = redfix . reconstruct . norm . sysEq . sf . simplify.
```

This function will carry out all the stages in order, giving a normalised version for any closed sHML formula.

4.1 Possible Future Work

There are two main practical issues yet to tackle. First of all, the assumption that any two syntactically disjoint symbolic actions are disjoint in section 3.4 is false in general. Indeed, one need not be creative to find an example: $\{i?3, i = 4\}$ and $\{i?3, i \geqslant 4\}$ are two symbolic actions which are clearly not disjoint. In subsection 5.4.1 of [1], the authors describe a way to manipulate symbolic actions so that their syntactic disjointness implies their semantic disjointness. This takes the form of two "additional" normalisation steps, §3.i and §3.ii.

Once this is taken care of, then the algorithm described in definition 1.6 can be implemented to actually synthesise sHML monitors.

A The Code

The code can be cloned from the Git repository at

https://github.com/drmenguin/shml-normaliser,

we also give it below for completeness.

A.1 The sHML Parser

```haskell
module SHMLParser where

import System.IO
import Control.Monad
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language
import qualified Text.ParserCombinators.Parsec.Token as Token

-- Data Structures
data Formula = LVar String
             | TT
             | FF
             | Con Formula Formula
             | Max String Formula
             | Nec Patt BExpr Formula
            deriving Eq

data Patt = Input Var AExpr
          | Output Var AExpr
         deriving Eq

data Var  = BVar String
          | FVar String
         deriving Eq

data AExpr = AVar Var
           | IntConst Integer
           | Neg AExpr
           | ABin ABinOp AExpr AExpr
          deriving Eq

data ABinOp = Add
            | Subtract
            | Multiply
            | Divide
           deriving Eq

data BExpr = BoolConst Bool
```

```
           | Not BExpr
           | And BExpr BExpr
           | RBin RBinOp AExpr AExpr
           deriving Eq

data RBinOp = Eq
            | Neq
            | Lt
            | Gt
            | LtEq
            | GtEq
            deriving Eq

-- Language Definition
lang :: LanguageDef st
lang =
    emptyDef{ Token.commentStart    = "/*"
            , Token.commentEnd      = "*/"
            , Token.commentLine     = "//"
            , Token.identStart      = letter
            , Token.identLetter     = alphaNum
            , Token.opStart         = oneOf "&~+-*/<>=!?$"
            , Token.opLetter        = oneOf "&~+-*/<>=!?$"
            , Token.reservedOpNames = ["&", "~", "+", "-", "*", "/", "<", ">",
                                       "<=", ">=", "==", "!=", ".", ",", "!",
                                       "?", "$"]
            , Token.reservedNames   = ["tt", "ff", "max"]
            }


-- Lexer for langauge
lexer =
    Token.makeTokenParser lang


-- Trivial Parsers
identifier     = Token.identifier lexer
keyword        = Token.reserved lexer
op             = Token.reservedOp lexer
integer        = Token.integer lexer
roundBrackets  = Token.parens lexer
squareBrackets = Token.brackets lexer
whiteSpace     = Token.whiteSpace lexer

-- Main Parser, takes care of trailing whitespaces
formulaParser :: Parser Formula
formulaParser = whiteSpace >> formula

-- Parsing Formulas
formula :: Parser Formula
formula = conFormula
        <|> formulaTerm

-- Conjunction
conFormula :: Parser Formula
conFormula =
    buildExpressionParser [[Infix (op "&" >> return Con) AssocLeft]] formulaTerm

-- Term in a Formula
formulaTerm :: Parser Formula
formulaTerm = roundBrackets formula
            <|> maxFormula
            <|> necFormula
            <|> ttFormula
            <|> ffFormula
            <|> lvFormula

-- Truth
ttFormula :: Parser Formula
ttFormula = keyword "tt" >> return TT

-- Falsehood
ffFormula :: Parser Formula
ffFormula = keyword "ff" >> return FF

-- Logical Variable
lvFormula :: Parser Formula
lvFormula =
    do  v <- identifier
        return $ LVar v

-- Least Fixed Point
```

```
maxFormula :: Parser Formula
maxFormula =
    do  keyword "max"
        x <- identifier
        op "."
        phi <- formulaTerm
        return $ Max x phi

-- Necessity
necFormula :: Parser Formula
necFormula = try condNecFormula
           <|> simpleNecFormula

-- Necessity with condition
condNecFormula :: Parser Formula
condNecFormula =
    do  (p,c) <- squareBrackets condpatt
        phi   <- formulaTerm
        return $ Nec p c phi

-- Inside of conditional pattern
condpatt :: Parser (Patt, BExpr)
condpatt =
    do  p <- pattern
        op ","
        c <- bExpression
        return (p,c)

-- Necessity without condition
simpleNecFormula :: Parser Formula
simpleNecFormula =
    do  p <- squareBrackets pattern
        phi <- formulaTerm
        return $ Nec p (BoolConst True) phi

-- Variable
var :: Parser Var
var = bvar <|> fvar

-- Free Variable
fvar :: Parser Var
fvar =
    do  v <- identifier
        return $ FVar v

-- Bound Variable
bvar :: Parser Var
bvar =
    do  op "$"
        v <- identifier
        return $ BVar v

-- Pattern
pattern :: Parser Patt
pattern = try inputPattern
        <|> outputPattern

-- Input pattern
inputPattern :: Parser Patt
inputPattern =
    do  v <- var
        op "?"
        a <- aExpression
        return $ Input v a

-- Output pattern
outputPattern :: Parser Patt
outputPattern =
    do  v <- var
        op "!"
        a <- aExpression
        return $ Output v a

-- Arithmetic Expressions
aExpression :: Parser AExpr
aExpression = buildExpressionParser aOperators aTerm

aOperators = [ [Prefix (op "-" >> return (Neg          ))          ]
             , [Infix  (op "*" >> return (ABin Multiply)) AssocLeft,
                Infix  (op "/" >> return (ABin Divide  )) AssocLeft]
             , [Infix  (op "+" >> return (ABin Add     )) AssocLeft,
                Infix  (op "-" >> return (ABin Subtract)) AssocLeft]
```

```
             ]

aTerm :: Parser AExpr
aTerm = roundBrackets aExpression
      <|> liftM AVar var
      <|> liftM IntConst integer


-- Boolean Expressions
bExpression :: Parser BExpr
bExpression = buildExpressionParser bOperators bTerm

bOperators = [ [ Prefix (op "~" >> return Not)          ]
             , [ Infix  (op "&" >> return And) AssocLeft]
             ]

bTerm :: Parser BExpr
bTerm = roundBrackets bTerm
      <|> (keyword "tt" >> return (BoolConst True))
      <|> (keyword "ff" >> return (BoolConst False))
      <|> rExpression


-- Relational Expressions
rExpression :: Parser BExpr
rExpression =
    do  a1 <- aExpression
        rel <- relation
        a2 <- aExpression
        return $ RBin rel a1 a2

relation :: Parser RBinOp
relation = (op "==" >> return Eq)
         <|> (op "!=" >> return Neq)
         <|> (op "<"  >> return Lt)
         <|> (op ">"  >> return Gt)
         <|> (op "<=" >> return LtEq)
         <|> (op ">=" >> return GtEq)


-- Parse String Input
parseF :: String -> Formula
parseF s =
    case ret of
        Left e -> LVar "ErrorParsing"
        Right f -> f
    where
        ret = parse formulaParser "" s


-- Pretty Outputs (Parse tree)
indent :: Int -> String
indent 0 = "  "
indent 1 = "  |-"
indent n = "   " ++ indent (n-1)

prettyf :: Formula -> Int -> String
prettyf f n = (indent n) ++ pf
    where
        pf =
            case f of
                LVar s -> s ++ " (logical variable)\n"
                TT -> "TT\n"
                FF -> "FF\n"
                Con phi psi -> "&\n" ++ prettyf phi (n+1)
                               ++ prettyf psi (n+1)
                Max x phi   -> "max " ++ x ++ " .\n"
                               ++ prettyf phi (n+1)
                Nec p c phi -> "Necessity\n"
                               ++ prettyp p (n+1)
                               ++ prettyb c (n+1)
                               ++ prettyf phi (n+1)

prettyp :: Patt -> Int -> String
prettyp p n =
    case p of
        Input v a   -> (indent n) ++ "Input\n"
                       ++ prettyv v (n+1) ++ "\n"
                       ++ prettya a (n+1)
        Output v a  -> (indent n) ++ "Output\n"
                       ++ prettyv v (n+1) ++ "\n"
                       ++ prettya a (n+1)
```

```haskell

prettyv :: Var -> Int -> String
prettyv v n =
    case v of
        BVar v -> (indent n) ++ v ++ " (binding variable)"
        FVar v -> (indent n) ++ v ++ " (free variable)"


prettya :: AExpr -> Int -> String
prettya a n =
            case a of
                AVar v -> prettyv v n ++ "\n"
                IntConst i -> (indent n) ++ (show i) ++ " (int const)\n"
                Neg a1 ->(indent n) ++ "Negation (-)\n"
                          ++ prettya a1 (n+1)
                ABin binop a1 a2 -> (indent n) ++ sbinop ++ "\n"
                                    ++ prettya a1 (n+1)
                                    ++ prettya a2 (n+1)
                    where
                        sbinop =
                            case binop of
                                Add -> "+"
                                Subtract -> "-"
                                Multiply -> "*"
                                Divide -> "/"

prettyb :: BExpr -> Int -> String
prettyb b n = (indent n) ++ pb
    where
        pb =
            case b of
                BoolConst bc -> (show bc) ++ " (bool const)\n"
                Not b1 -> "Negation (~)\n"
                          ++ prettyb b1 (n+1)
                And b1 b2 -> "&\n" ++ prettyb b1 (n+1)
                             ++ prettyb b2 (n+1)
                RBin rbinop a1 a2 -> sbinop ++ "\n"
                                    ++ prettya a1 (n+1)
                                    ++ prettya a2 (n+1)
                    where
                        sbinop =
                            case rbinop of
                                Eq -> "="
                                Neq -> "!="
                                Lt -> "<"
                                Gt -> ">"
                                LtEq -> "<="
                                GtEq -> ">="


-- Output Parse Tree of a given Formula
parseTree :: Formula -> IO ()
parseTree f = putStrLn (prettyf f 0)

-- String to Parse Tree
stringParseTree :: String -> IO ()
stringParseTree s =
    case ret of
        Left e ->  putStrLn $ "Error: " ++ (show e)
        Right f -> putStrLn $ "Interpreted as:\n" ++ (prettyf f 0)
    where
        ret = parse formulaParser "" s


-- Normal output (formula)
instance Show Formula where
    showsPrec _ TT = showString "tt"
    showsPrec _ FF = showString "ff"
    showsPrec _ (LVar v) = showString v
    showsPrec p (Con f1 f2) =
        showParen (p >= 2) $ (showsPrec 2 f1) . (" & " ++) . showsPrec 2 f2
    showsPrec p (Max x f) =
        showParen (p >= 3) $ (("max " ++ x ++ " . ") ++) . showsPrec 3 f
    showsPrec p (Nec pt c f) =
        case c of
            BoolConst True ->
                showParen (p >= 4) $ (("[" ++ show pt ++ "]") ++) . showsPrec 4 f
            _ ->
                showParen (p >= 4) $ (("[" ++ show pt ++"," ++ show c ++ "]") ++) .
      showsPrec 4 f

instance Show Patt where
```

```
    show (Input v a) = (show v) ++ " ? " ++ (show a)
    show (Output v a) = (show v) ++ " ! " ++ (show a)

instance Show Var where
    show (FVar v) = v
    show (BVar v) = "$" ++ v

instance Show AExpr where
    showsPrec _ (AVar v) = shows v
    showsPrec _ (IntConst i) = shows i
    showsPrec p (ABin op a1 a2) =
        case op of
            Add ->
                showParen (p >= 5) $ (showsPrec 5 a1) . (" + " ++) . showsPrec 5 a2
            Subtract ->
                showParen (p >= 5) $ (showsPrec 5 a1) . (" - " ++) . showsPrec 5 a2
            Multiply ->
                showParen (p >= 6) $ (showsPrec 6 a1) . (" * " ++) . showsPrec 6 a2
            Divide ->
                showParen (p >= 6) $ (showsPrec 6 a1) . (" / " ++) . showsPrec 6 a2

instance Show BExpr where
    showsPrec _ (BoolConst b) = shows b
    showsPrec _ (Not b) = ("~" ++) . (shows b)
    showsPrec p (And b1 b2) = (shows b1) . (" & " ++) . (shows b2)
    showsPrec p (RBin op b1 b2) =
        case op of
            Eq ->
                (shows b1) . (" = " ++) . (shows b2)
            Neq ->
                (shows b1) . (" != " ++) . (shows b2)
            Lt ->
                (shows b1) . (" < " ++) . (shows b2)
            Gt ->
                (shows b1) . (" > " ++) . (shows b2)
            LtEq ->
                (shows b1) . (" <= " ++) . (shows b2)
            GtEq ->
                (shows b1) . (" >= " ++) . (shows b2)
```

A.2 The Normalisation Algorithm

```
module SHMLNormaliser where

import Data.List
import Data.Char
import SHMLParser as Parser

-- Substitution of free variables
sub :: Formula -> String -> Formula -> Formula
sub phi v psi =
    case psi of
        LVar u
            | u == v     -> phi
            | otherwise -> psi
        Con f1 f2 -> Con (sub phi v f1) (sub phi v f2)
        Max u f
            | u == v     -> psi
            | otherwise -> Max u (sub phi v f)
        Nec p c f -> Nec p c (sub phi v f)
        _ -> psi


-- Replace free/bound variables of a formula
-- (Possibly introduces variable capture)
replace :: String -> String -> Formula -> Formula
replace x y phi =
    case phi of
        LVar u
            | u == x     -> LVar y
            | otherwise -> phi
        Con f1 f2 -> Con (replace x y f1) (replace x y f2)
        Max u f
            | u == x     -> Max y (replace x y f)
            | otherwise -> Max u (replace x y f)
```

```haskell
        Nec p c f -> Nec p c (replace x y f)
        _ -> phi


-- Basic Logical Simplifications (step 1)
simplify :: Formula -> Formula
simplify (Con phi psi) = simplifyCon (simplify phi) (simplify psi)
    where
        simplifyCon :: Formula -> Formula -> Formula
        simplifyCon FF _ = FF
        simplifyCon _ FF = FF
        simplifyCon TT b = b
        simplifyCon b TT = b
        simplifyCon a b
            | a == b      = a
            | otherwise   = (Con a b)
simplify (Max x psi) = simplifyMax x (simplify psi)
    where
        simplifyMax :: String -> Formula -> Formula
        simplifyMax x TT = TT
        simplifyMax x FF = FF
        simplifyMax x (LVar y)
            | x == y      = TT
            | otherwise   = Max x (LVar y)
        simplifyMax x (Con phi psi)
            | phi == LVar x = simplify (Max x psi)
            | psi == LVar x = simplify (Max x phi)
            | otherwise     = Max x (Con phi psi)
        simplifyMax x phi = Max x phi
simplify (Nec p c phi)
    | simpPhi == TT = TT
    | otherwise     = Nec p c simpPhi
    where
        simpPhi = simplify phi
simplify phi = phi


-- Standard form (step 2)
sf :: Formula -> Formula
sf f = simplify (conj (sf' f []))
    where
        conj :: (Formula, [String]) -> Formula
        conj (phi, [])  = phi
        conj (phi, v:vs) = Con phi (conj (LVar v, vs))

sf' :: Formula -> [String] -> (Formula, [String])
sf' (LVar x) bv
    | x `elem` bv  = (LVar x, [])
    | otherwise    = (TT, [x])
sf' (Con phi1 phi2) bv = (Con psi1 psi2, nub (vars1 ++ vars2))
    where
        (psi1, vars1) = sf' phi1 bv
        (psi2, vars2) = sf' phi2 bv
sf' (Max x phi) bv = (sub (Max x psi) x psi, delete x vars)
    where
        (psi, vars) = sf' phi (x:bv)
sf' (Nec p c phi) bv = (Nec p c (sf phi), [])
sf' phi _ = (phi, [])


-- All variables which appear in formula (free or bound)
variables :: Formula -> [String]
variables = nub . variables'

variables' :: Formula -> [String]
variables' (LVar x) = [x]
variables' (Con phi psi) = (variables' phi) ++ (variables' psi)
variables' (Max x phi) = [x] ++ (variables' phi)
variables' (Nec p c phi) = variables' phi
variables' _ = []


-- Rename the variables in a formula using integers
rename :: Formula -> (Formula, [(Int, String)])
rename phi = (psi, sigma)
    where
        sigma = zip [0..] (variables phi)

        listReplace :: [(Int, String)] -> Formula -> Formula
        listReplace (p:ps) =
            (listReplace ps).(replace (snd p) (show (fst p)))
        listReplace [] = id
```

```
        psi = listReplace sigma phi


-- Equation 'X = phi' encoded as (X, phi)
type Equation = (String, Formula)
type SoE      = ([Equation], String, [String])


-- System of Equations (step 3)
sysEq :: Formula -> (SoE, [(Int, String)])
sysEq phi = (sysEq' 0 phi', sigma)
    where
        (phi', sigma) = rename phi

sysEq' :: Int -> Formula -> SoE
sysEq' n TT = ([(x, TT)], x, [])
    where
        x = "X" ++ show n

sysEq' n FF = ([(x, FF)], x, [])
    where
        x = "X" ++ show n

sysEq' n (LVar y) = ([(x, LVar y)], x, [y])
    where
        x = "X" ++ show n

sysEq' n (Con f1 f2) = (eq, x, y1 ++ y2)
    where
        x = "X" ++ show n
        (eq1, x1, y1) = sysEq' (n+1) f1
        lastEq1 = read ((tail.fst.last) eq1) :: Int
        (eq2, x2, y2) = sysEq' (lastEq1+1) f2
        eq = [(x, Con (snd (head eq1)) (snd (head eq2)))] ++ eq1 ++ eq2

sysEq' n (Max u f) = (eq, x, y)
    where
        x = "X" ++ show n
        (eq1, x1, y1) = sysEq' (n+1) (replace u x f)

        expandX :: Equation -> Equation
        expandX (v, rhs)
            | rhs == LVar x = (v, snd(head eq1))
            | otherwise     = (v, rhs)

        eq = [(x, snd(head eq1))] ++ (map expandX eq1)
        y = filter (\v->v/=x) y1

sysEq' n (Nec p c f) = (eq, x, y)
    where
        x = "X" ++ show n
        (eq1, x1, y) = sysEq' (n+1) f
        eq = [(x, Nec p c (LVar x1))] ++ eq1


-- Normalisation of System of Equations (Power Set Construction, step 4)

-- The following functions are for subset/index manipulation
-- nsubsets (Non-empty subsets, with singletons first, then lexicographical)
nsubsets :: Eq a => [a] -> [[a]]
nsubsets s = [[i]|i<-s] ++ (subsequences s \\ ([]:[[i]|i<-s]))

-- Index (subscript) of a variable Xi
idx :: String -> Int
idx (x:xs) | x == 'X'  = read xs :: Int
           | otherwise = -1

-- Subset corresponding to given index
subIdx :: Int -> Int -> [Int]
subIdx n = (!!) $ nsubsets [0..n-1]

-- Index corresponding to given Subset
idxSub :: Int -> [Int] -> Int
idxSub n [k] | k < n      = k
             | otherwise  = error "Not a valid subset"
idxSub n s = binarysum (n-1) (reverse memberQSet) + n - 2 - maximum s
    where
        binarysum k []     = 0
        binarysum k (x:xs) = (2^k * x) + binarysum (k-1) xs
        btoi True  = 1
        btoi False = 0
```

```haskell
        memberQSet = [btoi (i `elem` s) | i <- [0..(n-1)]]


-- All symbolic actions in a formula
sas :: Formula -> [(Patt, BExpr)]
sas = nub . sas'

sas' :: Formula -> [(Patt, BExpr)]
sas' (Nec p c phi) = (p,c) : sas' phi
sas' (Con phi psi) = (sas' phi) ++ (sas' psi)
sas' (Max x phi) = sas' phi
sas' _ = []

-- Factor (i.e. normalise) a single equation in SoE with n equations
factor :: Int -> Equation -> Equation
factor n (v, FF) = (v, FF)
factor n (v, LVar x) = (v, LVar x)
factor n (v, rhs)
    = (v, bigWedge ((map saVarToFormula $ saVarPairs rhs) ++ (unguardedVars rhs)))
    where
        saVars (p,c) (Nec p' c' (LVar x))
            | p == p' && c == c' = [x]
            | otherwise          = []
        saVars (p,c) (Con phi psi) = saVars (p,c) phi ++ saVars (p,c) psi
        saVars (p,c) _ = []

        guardedVars phi = concat [saVars sa phi | sa <- sas phi]
        unguardedVars phi = map (\x -> LVar x) (variables phi \\ guardedVars phi)
        saVarPairs phi = [(sa, map idx $ saVars sa phi) | sa <- sas phi]

        saVarToFormula ((p,c), v) = Nec p c (LVar ("X" ++ show (idxSub n v)))

        bigWedge [] = FF
        bigWedge lst = foldl1 (\x y -> Con x y) lst

-- Normalisation of SoE's
norm :: (SoE, a) -> (SoE, a)
norm ((eq, x, y), sigma) = ((map (factor n) psEqs, x, y), sigma)
    where
        n = length eq
        conj = \x y -> Con x y
        lhs = ["X" ++ show i | i <- [0..2^n-2]]
        rhs = map (foldl1 conj) $ (nsubsets.snd.unzip) eq
        psEqs = zip lhs rhs


-- Formula Reconstruction (step 5)

-- Free variables
fv :: Formula -> [String]
fv (LVar x)      = [x]
fv (Con phi psi) = fv phi ++ fv psi
fv (Nec p c phi) = fv phi
fv (Max x phi)   = fv phi \\ [x]
fv _             = []

-- Compose a list of maps
compose :: [a -> a] -> (a -> a)
compose [] = id
compose (f:fs) = f . (compose fs)

-- Reconstruction
reconstruct :: (SoE, [(Int, String)]) -> Formula
reconstruct ((eq, x, y), sigma) = sigma' recon
    where
        recon = sigmaSHML (LVar x) (eq, x, y)
        sigma' = compose [replace (show u) v | (u,v) <- sigma]

-- Recursive SigmaSHML Map
sigmaSHML :: Formula -> SoE -> Formula
sigmaSHML phi (eq, x, y)
    | fv phi == []      = phi
    | fv phi `subset` y = phi
    | otherwise         = sigmaSHML ((compose subs) phi) (eq, x, y)
        where
            getEq v = case lookup v eq of
                Nothing  -> TT
                Just rhs -> rhs

            subs = [sub (Max x (getEq x)) x | x <- fv phi]

            subset (a:as) b = elem a b && subset as b
```

```
            subset [] b = True


-- Redundant fixed points (step 6)
redfix :: Formula -> Formula
redfix (Max x phi)
    | x `elem` (fv phi) = Max x (redfix phi)
    | otherwise         = redfix phi
redfix (Con phi psi) = Con (redfix phi) (redfix psi)
redfix (Nec p c phi) = Nec p c (redfix phi)
redfix phi = phi


-- Normal Form (all steps in order)
nf :: Formula -> Formula
nf = redfix . reconstruct . norm . sysEq . sf . simplify

-- Normal Form from string
nfs :: String -> Formula
nfs = nf . parseF
```

Bibliography

[1] L. Aceto, I. Cassar, A. Francalanza, and A. Ingólfsdóttir. On Runtime Enforcement via Suppressions. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory (CONCUR 2018)*, volume 118 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 1st edition, 2007.

[3] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. Developing theoretical foundations for runtime enforcement, 2018.

[4] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and S. Kjartansson. Determinizing monitors for HML with recursion. *CoRR*, abs/1611.10212, 2016.

Index