A Model Checker for CSP$_M$

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Marc Fontaine

Düsseldorf, Juli 2011

aus dem Institut für Informatik der Heinrich-Heine Universität Düsseldorf

gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Michael Leuschel
Koreferent: Prof. Dr. Heike Wehrheim

Tag der mündlichen Prüfung: 6. Oktober 2011

Abstract

This thesis presents a new tool for the animation and model checking of CSP$_M$. CSP$_M$ is the *machine readable* syntax for Hoare's *Communicating Sequential Processes*. It is a specification language used by several formal methods tools, for example FDR and ProB.

The main contribution of the thesis is the detailed and comprehensive discussion of a new CSP$_M$ tool. The most important design goals for my CSP$_M$ tool are correctness, modularity and reusability. I describe how the design goals are reflected in the source code and explain the basic design decisions that were taken. I also compare the features and performance of the new tool with ProB and FDR and I present some benchmarks for a multi-core version of my tool.

This thesis is also a case study for the use of the Haskell programming language for the implementation of a formal methods tool. I explain how Haskell has influenced the design and how Haskell helps to achieve the design goals of my software. The presented software can serve as a reference implementation of the CSP$_M$ semantics and as a building block for future CSP$_M$ tools.

Kurzreferat

Diese Arbeit präsentiert ein neues Softwarewerkzeug zur Animation und zum Modelchecking von CSP$_M$. CSP$_M$ ist eine Spezifikationssprache, aus dem Bereich der formalen Methoden, die von mehreren Werkzeugen unterstützt wird, z.B. FDR und ProB. Sie basiert auf der Prozessalgebra kommunizierender sequenzieller Prozesse (engl. communicating sequential processes, CSP) von C.A.R. Hoare.

Der Hauptbeitrag der Arbeit ist die detaillierte und umfassende Beschreibung eines neuen Werkzeugs für CSP$_M$. Die wichtigsten Designziele meines Werkzeugs sind Korrektheit, Modularität und Wiederverwendbarkeit. Ich beschreibe die Umsetzung der Designziele im Quellcode und erkläre die grundsätzlichen Designentscheidungen, die getroffen wurden. Außerdem vergleiche ich die Eigenschaften und die Leistungsfähigkeit meines neuen Werkzeugs mit ProB und FDR und zeige mehrere Laufzeitmessungen für eine Parallelrechnerversion meines Programms.

Diese Arbeit ist auch eine Fallstudie über die Verwendung von Haskell als Programmiersprache für Werkzeuge im Bereich der formalen Methoden. Ich beschreibe wie Haskell das Design meiner Software beeinflusst hat und wie Haskell dazu beiträgt, die Designziele zu erreichen. Die erstellte Software kann als eine Referenzimplementierung für die CSP$_M$ Semantik und als Baustein für zukünftige CSP$_M$ Werkzeuge dienen.

Vorwort

Die vorliegende Dissertationsschrift entstand zwischen Februar 2010 und Juli 2011. Meine Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Softwaretechnik und Programmiersprachen begann Mitte 2005 und circa Mitte 2006 kam ich zum ersten Mal mit CSP, dem späteren Thema meiner Doktorarbeit, in Kontakt. Ich danke allen, die in dieser Zeit dazu beigetragen haben, dass diese Doktorarbeit erfolgreich beendet wurde.

Insbesondere danke ich Herrn Prof. Dr. Michael Leuschel für seine freundliche, großzügige und geduldige Unterstützung. Ohne die finanzielle Absicherung als wissenschaftlicher Mitarbeiter wäre diese Doktorarbeit nicht möglich gewesen.

Speziell danke ich Herrn Janus Tomaschewski für seine große Hilfbereitschaft, die ich bei verschiedenen Gelegenheiten in Anspruch genommen habe. Herrn Ivaylo Dobrikov danke ich für die Zusammenarbeit in Rahmen seiner Masterarbeit.

Des Weitern danke ich Jens Bendisposto, Daniel Plagge, Michael Jastram, Carl Friedrich Bolz und allen Mitarbeitern und Mitarbeiterinnen des Lehrstuhls für Softwaretechnik und Programmiersprachen und des Fachbereiches Informatik Heinrich-Heine-Universität für die freundliche Zusammenarbeit.

Eine große Motivation für mich war die Kooperation über den Lehrstuhl hinaus. Ich bedanke mich bei allen, die meine Software ausprobiert, getestet und Rückmeldungen dazu gegeben haben. In dem Zusammenhang möchte ich Moritz Kleine, Diego Oliveira, Marc Dragon, Philip Armstrong, Edward Turner und Robert Colvin nennen.

Contents

Chapter 1

Introduction

1.1 Introduction

"*I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*" [22]

The main subject of this thesis is the software design and the implementation of an interpreter for CSP$_M$. CSP$_M$, i.e. machine readable CSP, is one incarnation of Communicating Sequential Processes, a formalism devised by C.A.R. Hoare to deal with the complexity of concurrent systems. The design of *correct* concurrent computer systems is a difficult and interesting problem and at the same time concurrent computer systems are becoming more and more ubiquitous.

In CSP concurrent systems are modeled as processes, which communicate by synchronizing on events. CSP is a process algebra—a mathematical formalism—which makes it possible to define processes and to reason about the behavior of processes in a precise mathematical manner. CSP also makes it possible to mechanically check interesting properties of processes like, for example, deadlock freedom or safety properties.

Generally, CSP belongs to the so-called *formal methods*, a branch of computer science that investigates the use of mathematical methods for the design and analysis of software and hardware systems. The goal of formal methods is a provable correct system.

A CSP specification is the definition of a process using the CSP formalism. Several software tools, so-called model checkers and refinement checkers, have been developed for analyzing and checking CSP specifications. Among them are FDR[20], ProB[33] and ProBE[19].

The above three are similar in that they are based on a particular variant of CSP called machine readable CSP (abbreviated as CSP$_M$). CSP$_M$ is widely accepted in the CSP community (c.f. Section 8.6) and it is the de-facto standard for machine readable CSP specifications. The model checker, described in this thesis, uses CSP$_M$ as input language.

Unfortunately, CSP$_M$ exhibits many complexities which are often completely unrelated to the original formalism as it was proposed by Hoare. For example CSP$_M$ includes an ad-hoc definition of a functional programming language. One of the challenges of this thesis is to deal with the complexities of CSP$_M$ in a clear, structured manner.

Currently, the CSP community uses the exiting tools as black boxes. One goal of this thesis is to design an understandable implementation which can help to narrow the gap between CSP$_M$ users and CSP$_M$ tool implementers. Another motivation is that having multiple compatible CSP$_M$ tools helps to improve the overall confidence in the formalism.

This thesis is also a case study for the use of the Haskell programming language for writing a formal methods tool. For an objective case study, it is crucial that different approaches really solve exactly the same problem. Here, this means that a top priority of my work was to be compatible with existing tools.

Contributions

The main contributions of my work are:

- A new CSP$_M$ animator and model checker that improves upon FDR and ProB.
- A structured, documented and reusable implementation.
- A critical review of the CSP$_M$ formalism and existing tools.
- A real world case study for the use of the Haskell programming language.

1.2 Outline

The rest of the thesis is structured as follows:

Chapter 2 contains a short introduction to CSP and CSP$_M$. I informally explain the underlying ideas of CSP and the extensions of the CSP core language that are included in CSP$_M$. I also define what we consider as the CSP core language in this thesis.

Chapter 3 makes some remarks about the Haskell programming language and the role of Haskell in this thesis.

Chapters 4, 5 and 6 are a detailed description of the implementation. The structure of these chapters follows the module-structure of the program. Chapter 4 describes the implementation of the core language, Chapter 5 the functional sub-language of CSP$_M$ and Chapter 6 the front-end, i.e. the parser and the syntax tree for CSP$_M$.

Chapter 7 describes a simple extension of my model checker, which allows a speeding up of computations using parallel processing and modern multi-core hardware. It also contains some preliminary benchmarks for speed-ups that were measured for existing specifications from the literature.

Chapter 8 contains a comparison of the different CSP$_M$ tools. It also contains some small sections that do not deserve an individual chapter, like the description of the command line interface, installation instructions for my tool, etc. Finally, I have a future-work section (Chapter 9) and a summary (Chapter 10).

The most important part of the appendices is probably (Appendix A) which contains a complete listing of the implemented firing rules. The appendix also contains the source code listings of the benchmark CSP$_M$ specifications and source code listings of the most important Haskell modules of my CSP$_M$ tool.

Chapter 2

CSP

2.1 Informal Introduction to CSP

This section contains a small informal introduction to Communicating Sequential Processes. I focus on those aspects of CSP that are relevant for this work. For a comprehensive discussion of CSP and the underlying theories, see for example [52, 54, 21].

CSP is a formalism for the specification and analysis of concurrent systems. Systems are defined in terms of processes and events. CSP is also a process algebra, where processes are defined using equations and process operators. $STOP$ is the most primitive process, which never performs any event and does not take part in any synchronization.

Prefix

The prefix operation, written as "$\rightarrow$" is the most basic operation for building more complex processes. For example

$$P = a \rightarrow STOP$$

defines the process which can perform the event a and after that behaves exactly like the $STOP$ process. $Q = a \rightarrow b \rightarrow STOP$ first performs a then b and after that becomes $STOP$. The left hand side of the "$\rightarrow$" operation is always an event and the right hand side is a process. To save parentheses, it is a convention that "$\rightarrow$" is right associative, i.e. $Q = a \rightarrow b \rightarrow STOP$ is read as $Q = a \rightarrow (b \rightarrow STOP)$. Process definitions can be recursive. The process

$$R = a \rightarrow b \rightarrow R$$

first performs event a then event b and then behaves exactly as R. In fact, R can perform an infinite sequence of events $a, b, a, b, \cdots$. A sequence of events is also called a trace. Traces use an angle-brackets notation, e.g. $\langle a, b, c\rangle$. The notation $traces(P)$ is used for the set of all traces of process P. By definition, $traces(P)$ is prefix-closed, i.e. if $traces(P)$ contains $\langle a, b, c\rangle$ it also contains $\langle a, b\rangle$, $\langle a\rangle$ and $\langle\rangle$.

Interleaving

Besides the prefix operation, which combines an event and a process, CSP uses a number of operations which combine two or more processes. For example several processes can operate independently of each other in an interleaving mode. The operator-symbol for interleaving is "$|||$". The process

$$P = (a \to b \to STOP) \; ||| \; (c \to d \to STOP)$$

can perform the following set of traces:

$$\{\langle a, b, c, d\rangle, \langle a, c, b, d\rangle, \langle a, c, d, b\rangle, \langle c, d, a, b\rangle, \langle c, a, b, d\rangle, \langle c, a, d, b\rangle, \ldots\}$$

For simplicity, only the traces of length four, i.e. the maximal length traces, are shown.

Synchronization

The parallel composition operator, written as "$\|$", enforces synchronization between several processes. For example in the definition

$$P = Q \parallel R$$

$$Q = a \to b \to STOP$$

$$R = a \to c \to STOP$$

P is a synchronized combination of Q and R. The combined process $Q \parallel R$ can only perform the single event a. Events are instantaneous and the synchronized sub-processes Q and R perform the event a at the same moment in time. After that the system becomes equivalent to $P' = (b \to STOP) \parallel (c \to STOP)$ and since the events b and c are distinct, they do not synchronize and P' cannot perform any more events.

The model of communication of CSP is also known as hand-shake synchronization. An event can only occur if all involved processes are willing to perform it immediately. There are no buffers or transmission delays built into CSP and there is no distinction between sender and receiver. An event can synchronize an arbitrary number of processes.

Choice

Another important concept of CSP is *choice*. CSP contains two operators for modeling a choice between a number of possible alternatives, namely external choice ($\Box$) and internal choice ($\sqcap$). For example the process

$$P = (a \to b \to STOP) \sqcap (c \to d \to STOP)$$

can chose non-deterministically between either performing trace $\langle a, b\rangle$ or trace $\langle c, d\rangle$. External choice is sometimes also called angelic choice, while internal choice can be seen and as daemonic choice. A precise semantics of external choice and internal choice is given later.

Sequential Composition

The process $P = Q\ ;\ R$ first behaves like process Q and after *successful termination* of Q it behaves like R. To signal successful termination the new primitive process *SKIP* is used. For example the process

$$P = (a \rightarrow b \rightarrow \mathit{SKIP})\ ;\ (c \rightarrow d \rightarrow \mathit{STOP})$$

can perform the trace $\langle a, b, c, d\rangle$.

Abstraction

Finally, CSP provides a mechanism for abstracting the behavior of a process by hiding its internal implementation. The hiding operator is written as "\\". The left-hand side of "\\" must be of type process and the right-hand side argument of "\\" is the set of hidden events. The hidden events of a process are not visible outside the hiding construct and do not take part in any external synchronizations. For example the process

$$P = (a \rightarrow b \rightarrow c \rightarrow d \rightarrow \mathit{STOP}) \setminus \{a, c\}$$

has the trace $\langle b, d\rangle$.

Variants of CSP

These are the most basic and most agreed on building blocks for modeling processes in CSP. However, there is no canonical and fixed definition of the core CSP algebra; instead, there are some possible alternatives on what to consider part of core CSP.

For example interleaving and parallel composition can be seen as a special case of the other operator called alphabetized parallel $({}_x\|_y)$ [52]. Another alternative is to define process operations that work on sets of processes instead of binary operations. For example the binary internal choice can be seen as a special case of a generalized internal choice. The binary operation would then be defined as $P \sqcap Q == \bigsqcap\{P, Q\}$. Roscoe [52] defines several other useful operations like *interrupt*, *timeout* and *renaming*.

CSP is a flexible formalism for modeling and analyzing complex concurrent systems. There has been a lot of research on applications of CSP, on the expressiveness of CSP, on how to apply CSP to model real world problems and on extensions of the CSP method (c.f. Section 8.6). To successfully apply CSP, it is necessary to have a good intuition about the CSP concepts; it takes some time to build this intuition and to get familiar with the syntax and the meaning of the various process operators.

On the other hand, the subject of this thesis is the implementation of a tool for CSP$_M$. My experience is that an intuition about CSP does not help much for tool development. Instead of using intuition, I strictly focus on the formal semantics of CSP. In other words, I do not try to explain the difference between the timeout-operator and the interrupt-operator, etc. in this thesis; instead I take the operational semantics of CSP from Roscoe and implement it as-is.

2.2 Semantics

To use CSP in a rigorous and mathematical manner, it is necessary to define a *formal semantics*. This section sketches the three main alternatives:

- An algebraic semantics
- A denotational semantics
- An operational semantics

Roscoe gives detailed definitions of these three semantics and argues that they all define the same thing. My CSP$_M$ tool implements the operational semantics. Therefore, the underlying ideas of the operational semantics are explained in a little more detail. The complete implemented operational semantics is listed in Appendix A. The sections about the algebraic semantics and the denotational semantics may be skipped.

2.2.1 Algebraic Semantics

The algebraic semantics of CSP consists of a set of equations which are considered the axioms of CSP. By rewriting equations and terms with the help of the axioms, new interesting theorems can be found. The algebraic semantics of CSP can also be used to prove that two processes are equivalent. If two processes can be rewritten according to the axioms and theorems to become syntactically equal, then they must also be semantically equal.

This list gives a flavour of some typical CSP axioms:

$$\begin{array}{rcl}
P \mathbin{\Box} P & = & P \\
P \mathbin{\Box} Q & = & Q \mathbin{\Box} P \\
(P \mathbin{\Box} Q) \mathbin{\Box} R & = & P \mathbin{\Box} (Q \mathbin{\Box} R) \\
STOP \mathbin{\Box} P & = & P \\
P \sqcap P & = & P \\
P \sqcap Q & = & Q \sqcap P \\
(P \sqcap Q) \sqcap R & = & P \sqcap (Q \sqcap R) \\
a \rightarrow (P \sqcap Q) & = & (a \rightarrow P) \sqcap (a \rightarrow Q) \\
P \mathbin{|||} Q & = & Q \mathbin{|||} P \\
(P \mathbin{|||} Q) \mathbin{|||} R & = & P \mathbin{|||} (Q \mathbin{|||} R) \\
P \mathbin{|||} (Q \sqcap R) & = & (P \mathbin{|||} Q) \sqcap (P \mathbin{|||} R)
\end{array}$$

As in other algebras, there are axioms about associativity, commutativity and idem-potency of the operators and all kinds of distribution rules. The algebraic semantics is actually the reason why CSP is called a process algebra.

2.2.2 Denotational Semantics

This section sketches some ideas of denotational semantics for CSP. The denotational semantics is not immediately relevant for the rest of this thesis and this section can be safely skipped.

A denotational semantics for CSP can be roughly seen as a function $S[\![.]\!]$ which maps a syntactical description of a process to a mathematical model of the process behavior. The mathematical model could, for example, be the set of all traces which a process can perform.

Requirements for the denotational function $S[\![.]\!]$ are that it should be *consistent* with the operational and algebraic semantics and it should be defined as a recursion on the syntactical structure of the process. In other words, it should be possible to define $S[\![P \oplus Q]\!]$ with the help of $S[\![P]\!]$ and $S[\![Q]\!]$ for all process operations $\oplus$. The denotational semantics of recursive process definitions is usually defined as a suitable fixed-point of the denotational function. Roscoe describes two possible semantics in full detail. One uses the traces model the other uses the divergences-failures model.

2.2.3 Operational Semantics and Firing Rules

The operational view of a CSP process is a labeled transition system (LTS). The nodes of the LTS are labeled with process expressions and the transitions are labeled with events. For every state of the system, the LTS lists all possible transitions, i.e. all possible events that the system can perform in the state and the corresponding successor states. An LTS is very similar to a non-deterministic finite automaton, with the exception that an LTS may consist of an infinite number of states.

Given the initial state and the transition relation, it is straightforward to explore an LTS in breadth first or depth first manner and to perform different kinds of model checking. The operational semantics of CSP defines the transition relation of the LTS with the help of rules, which are called *firing rules* or *inference rules.*

Inference Rules

The general form of an inference rule is:

$$\frac{\text{Premise}_1, \cdots, \text{Premise}_n}{\text{Conclusion}}(\text{Side Condition})$$

An inference rule can have zero premises, in which case it is called an axiom. To prove some conclusion with the help of a inference rule, one has to check the side condition and also recursively prove all premises. Proofs are often laid out in the form of a proof tree. For example consider the following rule

$$\frac{x \leq y, y \leq z}{x \leq z}$$

and the tree axioms

$$\frac{}{2 \leq 3} \qquad \frac{}{3 \leq 4} \qquad \frac{}{4 \leq 1}$$

One can prove that $2 \leq 1$ with the following proof tree:

$$\frac{\dfrac{\dfrac{}{2 \leq 3} \quad \dfrac{}{3 \leq 4}}{2 \leq 4} \qquad \dfrac{}{4 \leq 1}}{2 \leq 1}$$

Unless stated otherwise, inference rules work purely syntactically. This means that they do not take any interpretation of the syntax into account. In the example, "$\leq$",1,2,3 and 4 are just symbols with no inherent meaning. $2 \leq 1$ is a valid conclusion for the given axioms. Side conditions can be used to express additional non-syntactical properties and constraints. For example,

$$\frac{}{x < y}\text{(The value of } x \text{ is less then the value of } y\text{)}$$

is an inference rule with a side condition. Inference rules can contain place holders (e.g. x, y, z), which get substituted with concrete syntax in the proof tree.

In CSP, inference rules are used to prove that a certain transition is possible. In the inference rules P, P', Q,.. are place holders for processes expressions and e is a place holder for an event. $P \xrightarrow{e} P'$ is the the transition from P to P' while performing event e. For example, this is the inference rule[1] for the prefix operation:

$$\frac{}{(e \to P) \xrightarrow{e} P}$$

This rule is an axiom. One rule for the '$\|$' operator is:

$$\frac{P \xrightarrow{e} P' \qquad Q \xrightarrow{e} Q'}{P \parallel Q \xrightarrow{e} P' \parallel Q'}$$

With these two rules one can prove that process $(a \to \mathit{STOP}) \parallel (a \to \mathit{STOP})$ can perform event a and then becomes process $\mathit{STOP} \parallel \mathit{STOP}$. This is the corresponding proof tree:

$$\frac{\dfrac{}{(a \to \mathit{STOP}) \xrightarrow{a} \mathit{STOP}} \qquad \dfrac{}{(a \to \mathit{STOP}) \xrightarrow{a} \mathit{STOP}}}{(a \to \mathit{STOP}) \parallel (a \to \mathit{STOP}) \xrightarrow{a} \mathit{STOP} \parallel \mathit{STOP}}$$

τ Rules

The operational semantics describes a CSP process as an LTS. In other words, the process is characterized with the transitions it can make. Up to now, transitions were always of the form $P \xrightarrow{e} P'$ (P performs event e and becomes P'), i.e, they always involved an event e. It turns out however, that some processes are best described with transitions which do not involve an event. For example the process $P \sqcap Q$ can non-deterministically choose between performing the transitions $(P \sqcap Q) \rightsquigarrow P$ and $(P \sqcap Q) \rightsquigarrow Q$.

Instead of inventing a new transition symbol like "$\rightsquigarrow$", standard CSP literature uses the following trick: One simply introduces a pseudo-event, called τ. Transitions which do not perform a regular event use the notation $P \xrightarrow{\tau} P'$. The transition is called a τ transition. For example the inference rules for "$\sqcap$" are:

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P} \qquad \frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

The τ event allows a unified notation for transitions that perform an event and transitions that do not.

[1] This is a simplified rule ignoring event patterns and binding identifiers.

$\checkmark$ Rules

A similar trick is used for the termination of a processes. The transition that happens when a process terminates is called a $\checkmark$-transition.[2] For process termination, it is also useful to introduce a pseudo-process Ω, which designates a process which has just terminated. With $\checkmark$ and Ω, the two most important firing rules for termination can be written as:

$$\frac{}{\mathit{SKIP} \xrightarrow{\checkmark} \Omega} \qquad \frac{P \xrightarrow{\checkmark} \Omega}{P\,;Q \xrightarrow{\tau} Q}$$

The set of all events of a specification is called the alphabet Σ. $\Sigma^{\checkmark,\tau} = \Sigma \cup \{\tau, \checkmark\}$ is the set containing all regular events plus the special τ and $\checkmark$ event. With the help of τ and $\checkmark$ all conclusions and premises of the CSP inference rules are syntactically of the form $P \xrightarrow{x} P'$ with $x \in \Sigma^{\checkmark,\tau}$. This allows a unified treatment of the different variants of transitions.

Nevertheless, I keep a clean separation between the firing rules for τ transitions, $\checkmark$-transitions and regular transitions, because for my implementation, it was also beneficial to implement them separately. Appendix A contains the complete list of inference rules that have been implemented.

Non-Determinism and Search

My CSP$_M$ tool uses the inference rules to search for possible transitions of a process. In other words, it searches for proof trees with the conclusion $P \xrightarrow{e} P'$ for a fixed and known process P, but the event e and P' are unknown. It is possible that there are several alternatives of matching firing rules which can be used to build a proof tree. In this case the tool tries all alternatives.

In general, finding a proof tree for a set of firing rules and a given conclusion can be an expensive computation. In the operational semantics, non-determinism can be modeled by simply having multiple firing rules which can be used alternatively. When a set of firing rules contains multiple rules that can be applied alternatively, one has to perform some kind of search to compute the proof trees. Firing rules are a succinct method for specifying an operational semantics. On the other hand, the involved non-determinism can make a firing rule semantics difficult to implement efficiently.

2.3 Extensions of Core CSP

Section 2.1 describes the basic ideas and concepts of CSP. I call this basic form of the formalism *core CSP* or the *CSP core language*. Unfortunately, it turns out that core CSP is often not expressive enough for specifying real world systems.

This section sketches some extensions that are used to make CSP more expressive. These extensions are all included in CSP$_M$ and the FDR tool. I will freely refer to the combination of core CSP with the various extensions as CSP$_M$ although, strictly speaking, CSP$_M$ only stands for the machine readable syntax of CSP.

[2] $\checkmark$-transition is pronounced as *tick-transition*.

The presentation makes a strict separation between core CSP and the extensions. This approach differs from the standard CSP textbooks [52, 54, 21] which first liberally introduce *extended versions of CSP* and later explain why these extensions are not relevant for the formal treatment of the *core CSP* semantics.

I put a stronger focus on the separation between the *core CSP* and the extensions because I found that this separation helped me to structure my program. It is a design decision of where exactly to draw line between *core CSP* and the *extensions* but I think that the chosen split works well. The separation between *core CSP* and *extension* corresponds to the split between Chapter 4 and Chapter 5. See those chapters for more details.

The following list of extensions is further split up into separate sub-sections. This further split-up is not directly reflected in the module structure of my program, but it is rather how one could, ad hoc, break up CSP$_M$ into separate aspects.

2.3.1 Multi-field Events and Data

Core CSP has no means to manipulate data or events. In core CSP, events are just abstract members of the set Σ and can only be used in prefix operations and for hiding etc. For real world applications, it is convenient to allow structured events which can carry data, for example an elevator might use the alphabet:

$$\Sigma = \{\mathit{action}.\mathit{floor} \mid \mathit{action} \in \{\text{openDoor}, \text{pressButton}\}, \mathit{floor} \in \{1,2,3\}\}$$

The events of the elevator are pairs (2-tuples) of an *action* and a $floor$, e.g. openDoor.1, openDoor.2 and pressButton.1. The *action* describes what happens and the $floor$ is some attached data. In general n-tuples of any size n can be used as events. The parts of a tuple are called *event fields* in this thesis. In CSP$_M$ the fields of an event are joined together with the dot-operator.

A convention is that the first field of an event is called the *channel*. The channels of the elevator are *pressButton* and *openDoor*. CSP$_M$ requires explicit channel declarations, which ensures that Σ is well defined and enumerable. The *event closure operation* is a CSP$_M$ notation for subsets of Σ. Event closures are written as `{|...|}`. The event closure of a channel is the set of all events from Σ, where the first event field equals the channel. As Σ is enumerable, so are event closures.

2.3.2 Event Patterns

Event patterns allow access to the fields of an event in prefix operations. For example the process

$$P = \text{pressButton}?x \rightarrow Q$$

can communicate the events pressButton.1,pressButton.2 and pressButton.3. The event pattern "pressButton?x" binds a new identifier x which can be used in the right-hand side of the prefix operation. For example a process which always opens a door on the floor where a button has been pressed can be specified as:

$$P = \text{pressButton}?x \rightarrow \text{openDoor}!x \rightarrow P$$

The syntax "?*pattern*" is used for an event *pattern* (input field) and the syntax "!$field$" is used to stress that a field is an output field. The only importance of input fields is to bind a value to an identifier. There is no distinction between input and output fields concerning synchronization. In other words, several input fields, several output fields or any combination of input and output fields may synchronize.

The extended firing rule for the prefix operation with event patterns is defined in Appendix A Rule R-1. The semantics of the prefix operation with pattern binding can be defined similar to the β-reduction step of the lambda calculus [52]. During the prefix step, free occurrences of the identifier, which is bound in the pattern, get substituted with the matched value.

2.3.3 Data Processing

Another logical extension is to allow event fields to be computed by arbitrary functions. For example the elevator could, for some reason, open the door in the next floor above, or it could use some arbitrary function `doorFun` to compute which door to open:

$$\begin{aligned} P &= \text{pressButton}?x \rightarrow \text{openDoor}!(x+1) \rightarrow P \\ P &= \text{pressButton}?x \rightarrow \text{openDoor}!doorFun(x) \rightarrow P \end{aligned}$$

2.3.4 Mixing Input and Output Fields

In CSP there is no distinction between events that receive data and events that send data and therefore it is also possible to mix input and output event fields in one prefix operation. For example an adder can be defined as:

$$Adder = add?a?b!(a+b) \rightarrow Adder$$

Although the prefix in the definition of *Adder* contains two pattern matches (which bind a and b) and the computation of an arithmetic expressions $(a+b)$, the events that the *Adder*-process communicates are still atomic. The events should be seen as just 4-tuples (e.g. $add.3.4.7$) with no indication of input or output fields.

2.3.5 Parametrised Processes

A system composed of many similar processes can conveniently be specified using parametrised process definition. For example a server running several identical services on port 80, 8080 and 8888 could be specified as:

$$Server = Service(80) \mid\mid\mid Service(8080) \mid\mid\mid Service(8888)$$

To make it even more clear that the server is built of identical services, the replicated version of $\mid\mid\mid$ should be used.

$$Server = \Big|\Big|\Big|_{x \in \{80,8080,8888\}} Service(x)$$

The CSP$_M$ syntax for replicated operations is:

```
Server = |||x:{80,8080,8888}:Service(x)
```

2.3.6 Complete Functional Process Definition

In CSP$_M$, there is no distinction between processing of data, i.e functions on events, and functions that work on processes, like parametrized processes. Instead CSP$_M$ extends CSP with a full higher order functional programming language where processes and events are first class citizens.

In other words, processes and events can be arguments of functions as well as the return value of a function. For example, a process can be specified as:

```
P(x) = if x == 0 then STOP else out.x → P(x-1)
Main = P(10)
```

2.4 Formal Definition of CSP$_M$

The description of CSP$_M$ so far was very informal. Note that the presented extensions of the CSP core language are non-trivial. They add new possibilities for divergences and new side conditions on valid specifications. For a specification language for safety critical system, the following ingredients are desirable:

- A formal definition of the syntax of CSP$_M$.
- A formal semantics covering core CSP and the extensions.
- A type system for the functional language included in CSP$_M$.
- The side conditions that are not covered by the syntax and the type system.

In his PhD thesis [53] Scattergood provides these definitions to some extent. He provides a reference implementation and he also describes the CSP$_M$ parser, which is used in the FDR tool. However, there is gap between the formal definitions of Scattergood and what is actually implemented. For example, new syntax and new process operations have been recently added to FDR which are not covered by Scattergood and there is no formal proof that FDR is a correct implementation of the Scattergood semantics.

FDR is widely used and accepted as the de-facto standard CSP$_M$ tool and the current state is that the semantics of CSP$_M$ is implicitly defined by the reference implementation (namely the FDR tool). One goal of this thesis is to explore the edge cases of FDR and clarify CSP$_M$ by implementing the CSP$_M$ semantics in Haskell. The hope is that a Haskell program can help to fill in the gap between a pen-and-paper semantics like that of Scattergood and a black-box implementation like FDR.

Compared to a pen-and-paper semantics, a Haskell program has the advantage that it is executable and testable. It is easy to test that an executable semantics does not miss important pieces and behaves as one expects. One just has to run the program with interesting specifications and check the result. Compared to a black box like FDR, the hope is that a Haskell program can, at least to some extent, work as a declarative semantics.

To solve the discrepancies for good, one would have to define a formal semantics for CSP$_M$, write a CSP$_M$ tool and mechanically prove that the tool is indeed an implementation of the formal semantics. Unfortunately, this rigorous use of formal methods is beyond the scope of this thesis.

Chapter 3

Software Architecture of the CSP$_M$ Tool

3.1 The Haskell Programming Language

The web page `www.haskell.org` promotes Haskell as follows:
"Haskell is an advanced purely functional programming language. An open source product of more than twenty years of cutting edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, Haskell makes it easier to produce flexible, maintainable high-quality software. "

For this work the following features are important:

- Haskell supports algebraic data types (ATD) and pattern matching. ADTs and pattern matching are handy for all kinds of symbolic computations like parsers, syntax trees, compilers, theorem provers, etc.
- Haskell is a concise and expressive programming language.
- Haskell is purely functional, statically typed and there exists a semi-formal definition of the Haskell semantics. Those features help to reason about the correctness of Haskell programs. There are also tools which connect Haskell and theorem provers to generate formally verified programs.
- Haskell is a full featured general purpose programming language. A compiler (GHC) and a variety of libraries, for example parser generators, GUI tool kits, advanced data structures, etc. are available as open source.
- There is an active Haskell community. Useful infrastructure like building tools, profilers, etc. have been developed as community projects.
- GHC has built-in support for concurrency and parallelism. Parallel model checking is an interesting subject of research.
- The Glasgow Haskell compiler is highly optimizing. In special cases C-like performance is possible. My experience with the memory consumption of my CSP$_M$ tool was also overall positive.

The CSP$_M$ project is a good test case for the use of the Haskell programming language in the domain of formal methods tools. Implementing a CSP$_M$ interpreter is a clearly defined problem. The reference implementation (FDR) is implemented in C++ and a ProB-CSP extension has been implemented in SICStus-Prolog.

Haskell References

The definite reference for the Haskell programming language is the *Haskell Report*. The first version of the report defined a standard called Haskell-98 [48] while the second and latest version (Haskell 2010 Language Report [38]) basically consists of Haskell-98 plus some well established extensions.

The Haskell report is written in semi-formal style and is not suited for learning Haskell. Fortunately, there are several good Haskell tutorials online, for example Learn-you-a-haskell [36] and the Real World Haskell book [46]. I therefore do not try to come up with a new Haskell tutorial as part of this thesis.

The presented implementation uses some extensions of the Glasgow Haskell Compiler which go beyond the Haskell 2010 Language Report. Documentation for GHC can be found on the web-page `www.haskell.org/ghc`. Generally a good starting point for looking up Haskell-related information online is `www.haskell.org`.

3.2 Architecture Overview

I try to achieve some generally undisputed design goals. For example, the software should:

- have a modular design
- be easy to understand
- be amenable to testing
- consist of reusable components
- have good performance

Commonly agreed-on rules for good functional program design are:

- Use pure functions.
- Use total functions.
- Use small functions.
- Use strong types.
- Favor combinators (`map, fold`) over direct recursion.
- Favor statically checked properties (enforced by the type checker) over run-time case switches.

I also follow the concept that software architecture consists of the set of design decisions and try to explicitly state what the underlying design decisions of my software are.

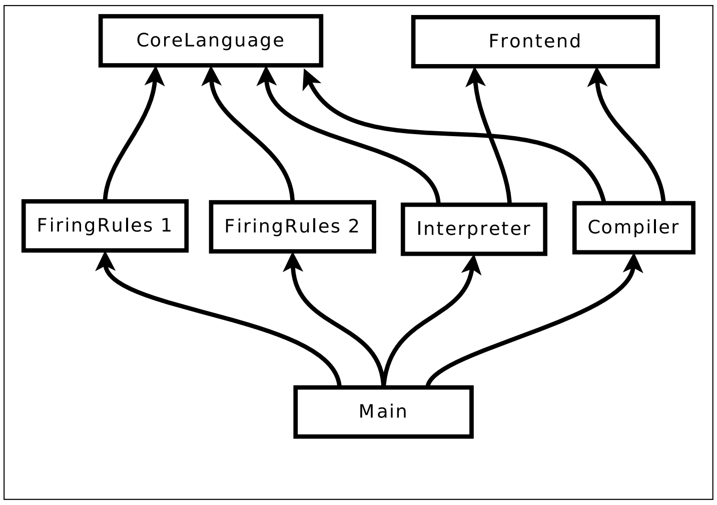


Figure 3.1: Component Structure

Component Structure

Figure 3.1 shows the dependency graph of of the CSP$_M$ tool components. The term *module* is sometimes used as a synonym for *component*, which should not be confused with a *Haskell module*. The components in this section basically correspond to *Haskell packages*. Each *Haskell package* itself may consist of several *Haskell modules*. It follows a brief overview of all components:

`CoreLanguage` models the core concepts of CSP, i.e. processes, events, event sets and process operations. It consists almost entirely of data type, type class, type family definitions and some small wrapper functions. `CoreLanguage` provides the interfaces but not the implementations.

`FiringRules 1` and `FiringRules 2` are two alternative implementations of the firing rule semantics of CSP, they only depend on `CoreLanguage`. The implementation of the firing rule semantics and the core language are described in Chapter 4.

`Interpreter` and `Compiler` are two alternative implementations for the functional sub-language of CSP$_M$. The interpreter is described in Chapter 5 while the compiler was the subject of Ivaylo Dobrikov's master thesis [12].

Both the `Interpreter` and `Compiler` depend on the `Frontend` which is described in Chapter 6. `Frontend` contains the CSP$_M$ parser and the definition of the syntax tree. Finally `Main` contains the `main` function which glues all the components together (see Chapter 8 for more details).

One design goal was to keep a strong separation between the core CSP concepts and the functional sub-language of CSP$_M$. The functional sub-language depends on `CoreLanguage` but not the other way around; at the same time the firing rule semantics is independent of the functional sub-language.

`CoreLanguage` is the central interface between core CSP and the functional sub-language. An explicitly defined interface makes it possible to have several alternative implementations of the firing rule semantics and the functional sub-language, which can be used in any combination and which can be tested independently. Concerning `CoreLanguage`, I think that there is some truth in the philosophy that the interface is more important than the implementation.

The interfaces of the components consist of the exported algebraic data types, type classes and the exported functions. All components except `Main` contain only pure functions (with some small exceptions). Most of the exported functions are also total, which means that the functions are easy to use and have a simple interface.

The result of a pure function is completely determined by the function arguments. One just has to call a function and it will always return the correct result. The type signatures tell what kind of arguments a function accepts and the type checker makes sure that a function can only be called with correct arguments. This makes it easy to reuse functions.

Most functions will not throw exceptions except in the case of internal errors of the implementation. Exceptions are only handled in `Main`.

Chapters 4, 5 and 6 can be read independently. In Chapter 6, only the definition of the abstract syntax tree is relevant for the other chapters, not the implementation of the parser itself.

Similarly, the interpreter can be seen as a black box which computes the function:

```
evalModule :: AST.Module INT → Env
```

In other words, the interpreter takes the abstract syntax tree of a module as input and computes the environment that is declared in the module. Most of the interpreter chapter can be understood independently of the other chapters.

Listings of the most important Haskell modules of each component are shown in the Appendix. Note that the development of the CSP tool is still going on and the source code in this document may be not identical with the the latest version of the packages in the HACKAGE online repository.

3.3 Source Code Included in the Thesis

This thesis contains relatively much Haskell source code. I have included source code for several reasons:

- The Haskell code is the most precise specification of what my program does.
- A piece of code is, hopefully, also a readable documentation of what the program does.
- Sometimes I want to show that an idea can be concisely expressed with a piece of source code.
- To argue whether a program is correct, one has to see the source code.

To read and understand the source code included in this document, it is necessary to have a basic proficiency in Haskell. On the other hand, to just get a feeling of how, for example, the firing rules relate to the implemented Haskell code, it may be possible to look at the source code and *just guess* what it means.

A reader with good proficiency in a functional programming language like ML, OCAML or F# will find it easier to read the Haskell source code. On the other hand Haskell is relatively different to imperative programming languages like C, JAVA or PYTHON.

The source code of the CSP$_M$ tool [14] is also browsable online on the HACKAGE package repository (*hackage.haskell.org*). The important interfaces are documented using the HADDOCK documentation tool, which is somewhat similar to JAVADOC. I did not use UML or similar approaches. UML seems to be best suited for modeling object-oriented software designs and Haskell is *not* an object-oriented programming language.

3.3.1 Hints for Understanding the Code

Here are some unsorted hints for understanding the Haskell code for readers who have never heard of Haskell before. These hints are not meant as a Haskell tutorial.

Haskell is not OOP

Haskell is not object-oriented programming.[1] Haskell emphasizes immutable values and pure functions, i.e. function without side effects. One should not pay too much attention to Haskell type classes. Type classes are somehow related to the classes of object-oriented programming, but in the end they are something different.

Types are the Key

The type signatures are a good documentation of the data flow of a program. The first step in understanding a function is understanding the types of the function arguments and the return type.

Algebraic Data Types and Case Switches

Algebraic data types and functions in Haskell roughly replace objects and methods in OOP. The constructors of an ADT replace sub-typing and inheritance. Functions that work on an ADT are often structured as a big case switch, which covers exactly the constructors of the ADT. Instead of dispatching to different methods of different objects, Haskell uses an explicit case switch.

In Haskell, a function covers all alternatives to provide some functionality, whereas in OOP, alternatives are modeled with sub-typing. In OOP, each method definition only provides the functionality for its subtype. If a Haskell function has an argument type of some ADT it makes sense to first understand the ADT and then the function.

Recursion

Recursive ADTs are often processed with structural recursive functions, i.e. the recursion follows the structure of the ADT. Mutually recursive functions are used for mutually recursive ADTs.

[1]Object-oriented programming is a very broad concept. It is possible to use concepts from OOP in Haskell, but OOP is not the underlying philosophy of Haskell and Haskell is not typically advertised as an OOP language.

Combinators and Operator Symbols

Haskell has only a small core syntax with few keywords and special purpose symbols.[2] On the other hand, the standard libraries define a huge number of combinators and operator symbols, for example `fmap`, `filter`, `mplus`, `$`, `>>=`, etc. Sometimes special purpose libraries define additional operator symbols or redefine operators (`<*>`, `<|>`, `<$>`, etc.). I have tried to write easy-to-read code, but reading Haskell is a also matter of practice. The author himself is still practicing and sometimes finds it difficult to read code which other people consider clear and easy to read. A very useful resource for looking up unknown functions and operators is `http://haskell.org/hoogle`.

Monads

There are two reasons why I do not want to include a monad tutorial in this thesis. First, there are several good monad tutorials available online[3] and second, in my opinion, the best approach for learning monads is to use them. Yet another monad tutorial would not be helpful. A good reference for monads and related concepts in Haskell is [64]. To understand the presented source code only a very basic knowledge of monads is needed. The main applications of monads in my CSP$_M$ tool are to model input/output, state, partial functions/exceptions, non-determinism, passing of an environment and a monadic parser.

A filter is a nice example[4] of the abstractions provided by monads. The non-monadic `filter` function has type:

```
filter :: (a → Bool) → [a] → [a]
```

The arguments of `filter` are a predicate and a list and it returns the sub-list containing all elements for which the predicate holds. For example:

```
Prelude> filter even [1,2,3,4,7,10]
[2,4,10]
```

The monadic version of a filter is:

```
filterM :: Monad m ⇒ (a → m Bool) → [a] → m [a]
```

`filterM` does roughly the same thing as `filter` except that it also provides the aspect of whatever monad `m` is used. The type of the monad `m` that is used for the predicate determines the type of the result. If the predicate involves input/output (`m` ≡ `IO`) then `filterM` will also perform input/output and if the predicate is a partial function (`m` ≡ `Maybe`) then the filter will also be a partial function.

```
fileExist :: FilePath → IO Bool
onlyFiles :: [FilePath] → IO [FilePath]
onlyFiles = filterM fileExist
```

One can also use `filterM` with the list monad for non-determinism and define the function `subseq`. `subseq` uses a predicate which ignores its argument and non-deterministically returns `False` or `True`.

[2]see `http://www.haskell.org/haskellwiki/Keywords`

[3] `http://www.haskell.org/haskellwiki/Monad_tutorials_timeline` lists about 33 monad tutorials.

[4]Thanks to Ivaylo Dobrikov for reminding me of this example.

```
subseq :: [b] → [[b]]
subseq = filterM (const [False,True])
```

subseq computes the sub-sequences of a list. Below follows a complete interaction with *ghci*, the REPL interface of GHC.

```
Prelude> import Control.Monad
Prelude Control.Monad> let subseq = filterM (const [False, True])
Prelude Control.Monad> subseq [1,2,3]
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
```

One abstraction for non-determinism in Haskell is the type class MonadPlus. (For more advanced abstractions see [27, 13]). The MonadPlus type class is defined as:

```
class Monad m ⇒ MonadPlus m where
   mzero :: m a
   mplus :: m a → m a → m a
```

mzero is a failed computation and mplus represents an alternative. The list instance of MonadPlus is defined as:

```
instance MonadPlus [] where
   mzero = []
   mplus = (++)
```

subseq2 is a function which works with any instance of MonadPlus:

```
subseq2 :: MonadPlus m ⇒ [b] → m [b]
subseq2 = filterM (const (return False ‘mplus‘ return True))
```

3.3.2 Type Families

The presented approach for modularizing the project makes heavy use of a relatively new extension of the Haskell type system called *type families*. Informally speaking, type families make it, to some extent, possible to define functions that work on types instead of data.

The Haskell-Wiki [8] introduction to type families is:

> Indexed type families, or type families for short, are a Haskell extension supporting ad hoc overloading of data types. Type families are parametric types that can be assigned specialized representations based on the type parameters they are instantiated with. They are the data type analogue of type classes: families are used to define overloaded data in the same way that classes are used to define overloaded functions. Type families are useful for generic programming, for creating highly parametrised library interfaces, and for creating interfaces with enhanced static information, much like dependent types.

Type families are an extension to the Haskell-2010 standard. Good references for type families are [45, 8], which address readers who are familiar with Haskell-2010.

Type families are only used for the interfaces between the building blocks of the project, i.e. the core language package and the functional sub-language package. The internal design of the packages is independent of type families and the packages themselves can be understood without type families.

Example

Suppose one wants to use some abstract data type `TF` for which one wants to allow several alternative implementations. This can be achieved with the following code:

```
type family TF i
type instance TF Int = Bool
type instance TF Integer = String
type instance TF Char = Bool
```

Type families are open, i.e. the lines of the above code can be spread over separate modules. `TF` is a mapping from index types to implementation types. In pseudo code it can be written as a function that works on types:

```
TF(Int)=Bool
TF(Integer)=String
TF(Char)=Bool
```

It is a function because every index type can be assigned at most one implementation type, however this function does not need to be injective. (`TF(Int)` and `TF(Char)` both have the implementation type `Bool`). This means that it is in general not possible to determine the value of `x` from `TF(x)`. A consequence is that values of type `TF(x)` can only be manipulated if the type of `x` is determined by the context.

I come back to the subject with a critique of type families in Section 3.5.

3.4 Role of Haskell

Choosing Haskell as implementation language for the CSP$_M$ tool is the single most important design decision of the project. The experience with the presented implementation shows that Haskell is well suited for the task and furthermore Haskell provides additional benefits which are often difficult to achieve with other languages. For example, Haskell helps to write correct programs, it helps to keep software small and declarative and also makes it easier to profit from multi-core parallelism.

This thesis has three aspects.

- I present my work on CSP$_M$ and a new CSP$_M$ tool.
- I present a case study for the use of the Haskell programming language.
- I work towards a reusable and understandable formal methods tool for CSP$_M$ as an alternative to black box implementations.

I think that Haskell can help to build reusable and understandable software. Therefore Haskell plays a central role in this thesis.

3.5 Haskell Critique

This sections informally, lists some points of critique against Haskell and the presented software design. The critique is related to my experiences during the work on the CSP$_M$ project, therefore it may make sense to read this section after the actual description of the implementation.

Modularization and Type Classes

A considerable effort was put into the modularization of the project. For example, I have separated the interface definitions from the implementations by using type classes and I have split the project into several packages. Compared to a monolithic implementation, this means that extra code has to be written and it also means that the functions have more complex types. For example in a monolithic implementation, one could define the following function to compare two CSP events for equality:

```
eventEq_monolithic :: Event → Event → Bool
eventEq_monolithic = ...
```

For the modularized version, which allows alternative implementations of `Event`, I define a type family and a class:

```
type family Event i
class BE i where eventEq :: i → Event i → Event i → Bool
```

and separate implementations:

```
data INT
type Event = [Field]
type instance Core.Event INT = Event
instance BE INT where eventEq _ty = ...
```

The first argument of `eventEq` is a phantom argument. Its only purpose is to give Haskell additional information about the types. It may be possible to circumvent the use of this phantom argument—which has other drawbacks, however.

The modularization of the CSP$_M$ project was one of the hardest problems of this work. Haskell's type classes are powerful, but they also belong to the more advanced features of the language. I would not claim to have fully understood type classes. Personally, I found them much more difficult than for example monads. Also, Haskell provides several alternative concepts for modularization and it is difficult to understand the design space and the pros and cons of the different approaches.

Of course the modularization overhead in the implementation is also related to the fact that the interfaces of the modules are statically typed. The modularization overhead could be reduced by giving up on static typing. Dynamic languages follow the philosophy that the trade-offs are in general against static typing. However my experience in the CSP$_M$ project was that the benefits of static typing are much more important than the drawbacks. Section 8.5 compares the size of the source code of my Haskell implementation with the size of a CSP$_M$ tool that is implemented in Prolog. This comparison shows that the Haskell source code is still concise, in spite of some overhead for modularization and static typing.

To summarize, I think that improving the current design with respect to modularization is an interesting subject of further research. The current design has some drawbacks, like the use of phantom-type arguments; on the other hand it is tested and works in practice.

Cutting Edge Research

Haskell extensions are a subject of cutting edge research. The Glasgow Haskell Compiler supports many extensions which go beyond the Haskell-2010 standard.

Extensions provide interesting features and can make Haskell more expressive. On the other hand, most of the extensions are restricted to GHC. The documentation of the extensions often only consists of research papers and is often difficult to understand.

Many extensions target the type system. If a program does not type-check, it could be that the programmer has not fully understood the extension, that there is some restriction in the type checker, or it could also be a plain bug in the compiler. At least, a program which does not type-check also does not compile and can therefore not contain run-time errors.

One of the extensions I use is *type families*. Although the type-family-based design works, it also has its drawbacks. In particular, the combination of *type families*, *type classes* and *phantom types* is complicated. It may be possible to improve the design, but this is future work and beyond the scope of this thesis.

Programming in the Large

Programming in the large in Haskell is difficult and requires learning new patterns and designs. Most Haskell tutorials only teach the language itself. There is a big gap between understanding the programming language and designing large Haskell programs. Haskell is a very expressive language, however that means that there are also many alternative ways to solve a problem and there many possibilities of choosing a bad design.

Chapter 4

Modeling of the CSP Core Language in Haskell

4.1 Overview

This section describes how I model the CSP core language. The CSP core language comprises processes and events but omits data processing and the functional sub-language of CSP$_M$. This chapter also describes how I model the firing rules that define the CSP operational semantics and proof trees which are built with the help of these firing rules.

Furthermore, I present two implementations of the actual operational semantics. The first implementation is based on a straightforward enumeration of events and consists of very succinct code while the second uses a constraint-based approach, which can improve the performance for some CSP specifications but which is also more elaborate.

One objective of my design was modularity. I wanted to keep the modeling of the core language well separated from other aspects of the CSP$_M$ specification language and in particular I wanted to make it possible to use the core language module with several different implementations of the functional sub-language. I also wanted to make it possible to test the implemented operational CSP semantic or at least to give some evidence for the correctness of the implementation.

Chapter Outline

The rest of Section 4.1 describes how I model the core concepts of CSP, namely processes and events. Section 4.2 explains my implementation of proof trees, the proof tree verifier and a naïve proof tree generator. Section 4.3 discusses an alternate, more complicated proof tree generator and Section 4.4 is dedicated to testing and the correctness of the code. The sections should be read in the above order. Section 4.3 is relatively technical and may be skipped.

4.1.1 Modeling of Processes

Processes are modeled as the algebraic data type `Process` (Listing 4.1).

Listing 4.1: The `Process` data type

```
data Process i
  = Prefix (Prefix i)
  | ExternalChoice (Process i) (Process i)
  | InternalChoice (Process i) (Process i)
  | Interleave  (Process i) (Process i)
  | Interrupt (Process i) (Process i)
  | Timeout (Process i) (Process i)
  | Sharing (Process i) (EventSet i) (Process i)
  | AParallel (EventSet i) (EventSet i) (Process i) (Process i)
  | RepAParallel [(EventSet i,Process i)]
  | Seq (Process i) (Process i)
  | Hide (EventSet i) (Process i)
  | Stop
  | Skip
  | Omega
  | Chaos (EventSet i)
  | AProcess Int
  | SwitchedOff  (ExtProcess i)
  | Renaming (RenamingRelation i) (Process i)
  | LinkParallel (RenamingRelation i) (Process i) (Process i)
```

`Process` contains a constructor for each CSP core operation, constructors for the primitive processes STOP, SKIP and some special constructors like e.g `Omega`. For performance reasons, binary alphabetized parallel operation and replicated (n-ary) alphabetized parallel operation are represented with two separate constructors (`AParallel` and `RepAParallel`). For the other replicated operations I use a simple translation to a nested binary operation. `AProcess` is only used for testing.

The module also defines two type families `Prefix`, `ExtProcess` and a type class `BL` [1] (see Listing 4.2).

The type families and the type class only declare an interface for which a user of the core language package has to provide the instantiation. One example for a user of the core language is the functional sub-language of CSP$_M$ (see Chapter 5).

Listing 4.2: The type class `BL`

```
type family Prefix i
type family ExtProcess i

class (BE i) ⇒ BL i where
  prefixNext :: Prefix i → Event i → Maybe (Process i)
  switchOn :: ExtProcess i → Process i
```

The type definitions in this module already indicate how the core language and the functional sub-language work together. Lets first explain the interplay of `SwitchedOff`, `switchOn` and `ExtProcess`. The constructor `SwitchedOff` is used to model processes that have not been evaluated yet, for example the right-hand side of a sequential composition $P\,;Q$. The constructor `SwitchedOff` contains data of type `ExtProcess i`. A user of the core language package has to instantiate

[1] `BL` stands for base language. In the presented implementation, this is the functional sub-language of CSP$_M$ but it could be something else.

the type family `ExtProcess` with a data type suitable for storing switched-off processes. The function

```
switchOn :: ExtProcess i → Process i
```

is used to switch on the process, i.e. to convert from `ExtProcess i` to `Process i`.

Prefix operations are modeled similarly to switched-off processes. The user of the core language module has to define its instance of the data family `Prefix` and instantiate the function `prefixNext`.

The core language uses the function `prefixNext` to perform events. `prefixNext` takes as arguments a prefix expression and an event and computes the representation of the new process, which is the result of the prefix performing the event. The new process is wrapped in a `Maybe` data type to make it possible to indicate that the process cannot perform the event (in which case `prefixNext` just returns `Nothing`).

The interface of the CSP core language is pure and all the involved data types are immutable. Class `BE` (`BE` stands for base event) is a super class of `BL`, which means that the functional sub-language also has to provide an instance of `BE`. `BE` is described in the next section. The `Process` data type is defined in module `Process`. The full source code is shown in Appendix B.1.1. The wrapper for replicated operations is defined in module `ProcessWrapper` (Appendix B.1.2).

Technical Remarks about Type Families

The types in this module are polymorphic with a type variable `i`. Technically speaking, `Process` is a type constructors of kind `* -> *` and `Prefix` and `ExtProcess` are type-indexed type families (See 3.3.2). Informally speaking, the type variable `i` makes it possible to use the core language package with more than one implementation of the functional sub-language. In the source code the process type often appears as `Process i`, but for brevity I sometimes omit the `i` and just write `Process`.

4.1.2 Modeling of Events

Module `Event` (Appendix B.1.3 / Listing 4.3) defines the interface of the core language package for atomic events. It defines three type families `Event`, `EventSet` and `RenamingRelation` and the class `BE` with basic functions that work on these types.

Listing 4.3: Type class BE

```
type family Event i
type family EventSet i
type family RenamingRelation i

class BE i where
  eventEq :: i → Event i → Event i → Bool
  member ::  i → Event i → EventSet i → Bool
  intersection :: i → EventSet i → EventSet i → EventSet i
  difference :: i → EventSet i → EventSet i → EventSet i
  union :: i → EventSet i → EventSet i → EventSet i
  null :: i → EventSet i → Bool
```

```
singleton :: i → Event i → EventSet i
insert :: i → Event i → EventSet i → EventSet i
delete :: i → Event i → EventSet i → EventSet i
eventSetToList :: i → EventSet i → [Event i]
allEvents :: i → EventSet i
isInRenaming :: i → RenamingRelation i → Event i → Event i → Bool
imageRenaming :: i → RenamingRelation i → Event i → [Event i]
preImageRenaming :: i → RenamingRelation i → Event i → [Event i]
isInRenamingDomain :: i → Event i → RenamingRelation i → Bool
isInRenamingRange :: i → Event i → RenamingRelation i → Bool
getRenamingDomain :: i → RenamingRelation i → [Event i]
getRenamingRange  :: i → RenamingRelation i → [Event i]
renamingFromList :: i → [(Event i, Event i)] → RenamingRelation i
renamingToList :: i → RenamingRelation i → [(Event i, Event i)]
singleEventToClosureSet :: i → Event i → EventSet i
```

In CSP$_M$, it is possible to define channels with additional data fields; however this module treats events as atomic, i.e. it hides the fact that an event may consist of several data fields. Section 4.3.6 describes a separate class `BF`, which also models the internal structure of events and allows to use data fields.

Note that the first argument of all functions in this module, the argument of type `i`, is used as a phantom-type argument. Its only purpose is to give the Haskell type checker additional information. Callers can simply pass `undefined` as first argument and instance-functions of class `BE` must ignore this argument.

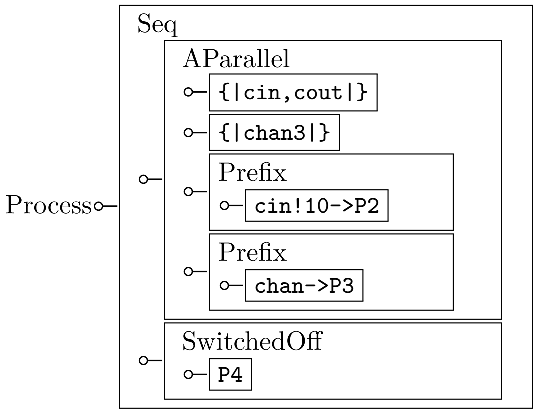
`undefined`, also written as $\perp$, is a bit like a NULL-pointer in C. Any attempt to "dereference" `undefined` causes a run-time exception. `undefined` is even more restrictive than NULL because it is (in pure code) not even possible to test if a value is equal to `undefined`.

4.1.3 Example of the Modeling of Processes

Let's consider the following process in CSP$_M$ syntax:

```
((cin!10→P2) [{|cin,cout|} || {|chan3|}] (chan2→P3)); P4
```

The core language view of the process is the following tree:



The relevant part of the process data type (Listing 4.1) is:

```
data Process i
```

```
  = Prefix (Prefix i)
  | AParallel (EventSet i) (EventSet i) (Process i) (Process i)
  | Seq (Process i) (Process i)
  | SwitchedOff  (ExtProcess i)
  ...
```

The core language module does not enforce any fixed implementation for:
`{|cin,cout|}`, `{|chan3|}`, `cin!10->P2`, `chan2-> P3` and `P4`.
The core language point of view is that these parts are represented using the type families `EventSet i`, `Prefix i` and `ExtProcess i`. Furthermore, these parts are only accessed via the interfaces defined in the type classes `BE` and `BL` (Listing 4.3 and Listing 4.2).

4.2 Implementation of the Operational Semantics

4.2.1 Section Outline

The previous section describes the data types and type classes which model processes and events. This section describes an implementation of the operational semantics of CSP based on the definitions of the previous section.

Basically, the operational semantics of CSP is defined by a set of firing rules, as described in Chapter 2. Firing rules are a concise, high level description of an operational semantics. They work as building blocks for proof trees, which show that a process can perform some transition. Given the firing rules and a proof tree, it is easy to check that a process can perform the corresponding transition. On the other hand, firing rules are implicitly non-deterministic and constructing a proof tree is in general much harder than verifying it (assuming that $N \not\equiv NP$).

This section describes an algorithm for constructing proof trees. The section is structured as follows: I first discuss the advantages of explicit proof trees and describe how I model them as a data structure. The presented data structure has some nice properties, for example it allows for a concise representation and some kind of built-in correctness (c.f. Section 4.2.2, Section 4.2.3). Along with this data structure I describe a concise function for verifying proof trees with respect to the firing rules. After that, I come to the core of the problem and describe a function which actually computes the proof trees (Section 4.2.4, Section 4.2.5)

In my tool, the operational semantics is used to perform model checking. More precisely, it is used to perform a breadth first search on the labeled transition system of the process. The model checker starts with an initial process, follows the transition relation in forward direction and checks all encountered processes for some property.

In this setting, the actual proof tree for a transition is not needed. It would, in principle, be sufficient to implement the operational semantics as a function `nextState :: Process` $\rightarrow$`[(Event,Process)]` that computes all possible transitions for a process. However, there are several reasons (c.f. 4.2.2) why it is still beneficial to compute explicit proof trees and therefore I split the computation of the transition relation into one function that computes the proof trees and one that extracts an actual transition from a proof tree.

The function for computing the proof trees has the type:

```
computeTransitions ::  BL i ⇒ Sigma i → Process i → [Rule i]
```

The two arguments of `computeTransitions` are the set Σ of all possible events and a process. `computeTransitions` computes a list of proof trees of possible transitions of that process. The data type `Rule i` is used to store a proof tree.[2]

The following function is used to examine proof trees:

```
viewRule :: BL i ⇒ Rule i → (Process i, TTE i, Process i)
viewRule r = case viewRuleMaybe r of
  Nothing → error "viewRule : internal error malformed Rule"
  Just v → v
```

This function verifies a proof tree and constructs an explicit representation of the transition that is actually proved. `TTE` is just a sum-type for τ, $\checkmark$ and regular events. In other words, the return type `(Process i, TTE i, Process i)` is a triple which simply encodes transitions ($P \xrightarrow{e} P'$).

The algorithms are constructed such that `viewRule` is always called with valid proof trees. If `viewRule` is ever called with a bad proof, there must be an error in the implementation and `viewRule` throws an exception in this case. `viewRule` is just a simple wrapper for `viewRuleMaybe`, which does the real work and which is described in Section 4.2.3.

4.2.2 Advantages of Explicit Proof Trees

Computing explicit proof trees has several advantages. First of all, it greatly helps to improve the confidence in the implementation. In particular, one can use QuickCheck [61] to randomly generate proof trees and automatically check properties of `viewRule` and `computeTransitions`. I describe this approach in detail in Section 4.4.

In general, proof trees are very useful for debugging. They can be used to give the user of the tool useful feedback about his specification. The proof tree immediately shows why a transition is possible, which is often not obvious with complicated specifications.

Finally, explicit proof trees have generally helped me to structure my implementation and they helped me to keep an overview of what firing rules have actually been implemented. Every firing rule has to be implemented in three places: in the `Rule` data type, in `viewRule` and in `computeTransitions`. The `Rule` data type clues the different parts together and `viewRule` is in some sense a test procedure for `computeTransitions`. It makes sense to first define the data constructor for a rule; after that, define the proof tree verifier for the rule and finally extend the proof tree generator.

The fact that one implements the same concept three times adds some redundancies. However, these redundancies do not add additional bugs. On the contrary, the proof tree verifier works like an assertion and the redundancy helps to catch errors.

Overall, the implementation is still far from being an elegant functional pearl. But, the presented architecture is not my first shot at CSP. An earlier attempt, which basically tried to compute the `nextState` function directly, worked to some extent but quickly became incomprehensible. In particular the earlier

[2] The misleading name `Rule` for the type of proof trees is used for historical reasons.

implementation was difficult to test and debug, which is exactly one issue that is addressed by using explicit proof trees.

Furthermore the run-time overhead for using explicit proof trees is small and I think that it is possible to apply a deforestation technique to the presented implementation and remove the overhead of explicit proof trees—if this should every be necessary.

It is interesting to compare the presented implementation with other designs. For example, there is often an elegant way to translate firing rules to Prolog clauses. In this approach, there is a one-to-one correspondence between a proof tree and a Prolog SDL derivation. The drawback is that the SDL tree is managed by the Prolog run-time system and there is no direct way to recover the proof tree from inside the Prolog program. Indirect methods like meta-programming can be used, however. The CSP$_M$ implementation of ProB [33] is Prolog based. Chapter 8.5 compares my implementation with ProB.

4.2.3 Modeling of Proof Trees

To show that a particular transition is consistent with the operational semantics one has to build the corresponding proof tree. The proof trees of interest are all similar in structure. The conclusion and the premises are always transitions of the form $P \xrightarrow{e} P'$, the premises are always proofed with a recursive proof tree and some firing rules might involve additional side conditions.

To verify a proof tree one has to check that the side conditions hold and also to ensure that the tree is "syntactically correct". With "syntactically correct", I mean that the tree is really constructed with valid instantiations of the firing rules. The identifiers in the firing rules work like logic variables. For example, if a firing rule contains two occurrences of P then both occurrences of P must be substituted with same syntactical expression in the proof tree.

I model proof trees as a Haskell data structure such that the proof trees are syntactically correct by construction, or in other words, such that it is impossible to construct syntactically invalid proof trees.

The idea can be explained with the help of an example. Let's consider the following proof tree:

$$\frac{\dfrac{}{(e \to Proc1) \xrightarrow{e} Proc1} \qquad \dfrac{}{(e \to Proc2) \xrightarrow{e} Proc2}}{((e \to Proc1)\ {}_A\|_B\ (e \to Proc2)) \xrightarrow{e} Proc1\ {}_A\|_B\ Proc2} e \in A \wedge e \in B$$

The tree is built with one application of the firing rule for parallel composition (R-13)

$$\frac{P \xrightarrow{e} P' \qquad Q \xrightarrow{e} Q'}{(P\ {}_X\|_Y\ Q) \xrightarrow{e} (P'\ {}_X\|_Y\ Q')} e \in X \wedge e \in Y$$

and two application of the rule for prefix (R-1).

$$\frac{}{(e \to P) \xrightarrow{e} P}$$

One can immediately see that the proof tree contains many occurrences of identical syntactical expressions ($Proc1$ and $Proc2$ each appear four times).

The simple idea is to store each expression that is part of a proof tree only once. Furthermore, the conclusion is mostly built of syntactical parts of the premises. Since each premise comes with its own proof tree, most of the syntactical parts of the conclusion also occur in the the proof trees of the premises. This means that one only has to store the event sets X and Y and the sub-proof trees for the two premises in the tree node for a parallel composition firing rule.

The data structure for the proof trees is constructed such that the missing parts of each inference step can be computed with a simple recursive traversal of the proof tree. At the same time, this traversal validates the proof tree.

The concrete code for the proof tree verifier for inference rule R-13 (alphabetized parallel) and R-1 (prefix operation) is the following:

Listing 4.4: Proof Tree Verifier

```
1  viewRuleEvent :: forall i. BL i
2    ⇒ RuleEvent i → Maybe (Process i, Event i, Process i)
3  viewRuleEvent rule = case rule of
4    AParallelBoth c1 c2 pp qq → do
5      (p, e2, p’) ← viewRuleEvent pp
6      (q, e1, q’) ← viewRuleEvent qq
7      guard $ eventEq ty e1 e2
8      in_Closure e1 c1
9      in_Closure e1 c2
10     return (AParallel c1 c2 p q, e1, AParallel c1 c2 p’ q’)
11   HPrefix e p → do
12     p’ ← prefixNext p e
13     return (Prefix p, e, p’)
```

`viewRuleEvent` validates a proof tree and (if valid) at the same time reconstructs the conclusion. `AParallelBoth` is the data constructor which stores a proof tree node for rule (R-13).

In detail, `viewRuleEvent` does the following:

Line 4 The pattern match for the alphabetized parallel rule binds `c1` to the left closure set, `c2` to the right closure set, `pp` to the left sub-proof tree and `qq` to the right sub-proof tree.

Lines 5,6 It reconstructs the two premises of the rule by recursively calling `viewRuleEvent` on the sub-proof trees (which also validates the sub-proofs). The premises are $P \xrightarrow{e2} P'$ and $Q \xrightarrow{e1} Q'$ (encoded as `(p, e2, p’)` and `(q, e1, q’)`).

Line 7 It checks that the event-part of both premises is equal ($e1 == e2$).

Lines 8,9 It checks that the event is in both synchronisation sets ($e1 \in c1$ and $e1 \in c2$).

Line 10 It returns the conclusion, i.e. the encoding of the transition.
`(AParallel c1 c2 p q, e1, AParallel c1 c2 p’ q’)`
$\equiv (P \;{}_{c1}\|_{c2}\; Q) \xrightarrow{e1} (P' \;{}_{c1}\|_{c2}\; Q')$

Note that `viewRuleEvent` is defined in terms of the `Maybe` monad. It can either return the transition of the proof tree or signal an invalid proof tree by returning

`Nothing`. Informally, in the `Maybe` monad, every line of a `do`-block corresponds to the check, whether the line has returned `Nothing`. In this case the hole `do`-block evaluates to `Nothing`. `viewRuleEvent` is structured as a big case switch over all firing rules. The above example only shows the cases for `AParallelBoth` and `HPrefix`.

`RuleEvent` is the data type which stores proof trees. It is implemented in module `Rules` (B.1.5). `AParallelBoth` is the constructor that is used for the alphabetized parallel firing rule. `AParallelBoth` has four fields: the left event closure set, the right event closure set, the left sub-proof tree and the right sub-proof tree. Here is the code for `RuleEvent` (showing only the constructors for rule R-1 and rule R-13):

```
data RuleEvent i =
  | ...
  | HPrefix (Event i) (Prefix i)
  | AParallelBoth (EventSet i) (EventSet i) (RuleEvent i) (RuleEvent i)
  | ...
```

Example using CSP$_M$ Syntax

Here is a slightly more elaborated example using CSP$_M$ syntax. As described in Section 4.1.1, the core language module and the proof tree verifier use an abstract view for those aspects that are handled by the functional sub-language of CSP$_M$.

Those parts are printed in $\boxed{\text{boxes}}$ and I simply repeat the concrete syntax CSP$_M$ $\boxed{\text{in the box}}$. The example process in CSP$_M$ syntax is:

```
(c?x→P(x)) [s1||s2] (c!3→ Q)
```

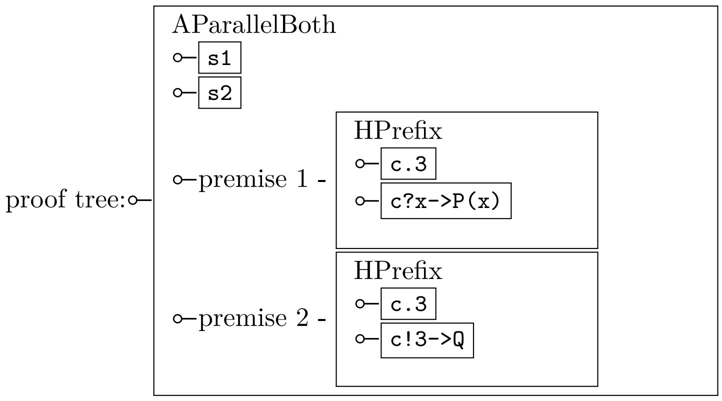
`s1`, `s2` are some event closure sets and we are again looking at a transition via the rules R-1 and R-13 (c.f. Appendix A). In particular the process should perform event `c.3` and become:

```
P(3) [s1||s2] Q
```

The proof tree for this transition is:

$$\frac{\dfrac{}{\boxed{\texttt{c?x->P(x)}} \xrightarrow{\boxed{\texttt{c.3}}} \boxed{\texttt{P(3)}}} \qquad \dfrac{}{\boxed{\texttt{c!3->Q}} \xrightarrow{\boxed{\texttt{c.3}}} \boxed{\texttt{Q}}}}{\left(\boxed{\texttt{c?x->P(x)}}\ {}_{\boxed{\texttt{s1}}}\|_{\boxed{\texttt{s2}}}\ \boxed{\texttt{c!3->Q}}\right) \xrightarrow{\boxed{\texttt{c.3}}} \boxed{\texttt{P(3)}}\ {}_{\boxed{\texttt{s1}}}\|_{\boxed{\texttt{s2}}}\ \boxed{\texttt{Q}}}\ \boxed{\texttt{c.3}} \in \boxed{\texttt{s1,s2}}$$

Apart from the structure itself, the proof tree consists entirely of boxes of components that are implemented by the functional sub-language. It can also be seen that the proof tree contains many identical boxes. The Haskell representation of this proof tree is:



In the Haskell representation every box is only stored once. The Haskell proof tree only stores the information about the premises and the side conditions. The conclusion of the proof tree is not stored explicitly. Instead, it is reconstructed on the fly, at the same time as when the proof tree is verified with `viewRuleEvent`.

Note that the proof tree for the prefix operation only stores the prefix expression $\boxed{\texttt{c?x->P(x)}}$ and the actual event $\boxed{\texttt{c.3}}$. The computation of the result of the prefix operation, i.e. performing the pattern match for $\boxed{\texttt{c?x}}$ and binding `x` to `3`, is carried out by `prefixNext` (Line 12 Listing 4.4). The implementation of `prefixNext` is provided by the functional sub-language.

Similarly the side conditions `c.3∈s1` and `c.3∈s2` can be checked via the abstract function `member` from class `BE` (Listing 4.3). `in_closure` (Line 8,9 Listing 4.4) is just a small wrapper for `member`. Since the two premises of `AParallelBoth` return two completely unrelated transitions, one also has to check that the events from both transitions are equal (Line 7 Listing 4.4).

The proof tree verifier, the `Rule` data type and the firing rules all express the same semantics. It may be helpful to look at Appendix A, which lists all firing rules and the corresponding proof tree verifiers.

The same approach works for all firing rules of CSP, since they are all similar in structure. The data type `RuleEvent` is used for firing rules with regular events. ✓-rules and τ-rules are represented with the data types `RuleTick` and `RuleTau` respectively. To verify ✓-and τ-rules, I use the following two functions:

```
viewRuleTau :: BL i ⇒ RuleTau i → Maybe (Process i, Process i)
viewRuleTick :: BL i ⇒ RuleTick i → Maybe (Process i)
```

Note that the conclusion of a τ-rule is always a transition with a τ event. Therefore there is no need to return an explicit event and `viewRuleTau` only returns a pair of two processes. Similarly, the conclusion of a ✓-rule is always in the form $P \xrightarrow{\checkmark} \Omega$ and it is sufficient to return P. For every rule, it is statically known if the premises are normal transitions, ✓- or τ- transitions.

The proof tree verifier is implemented in module `Verifier` (Listing in Appendix B.1.6). The code of the verifier is mostly self-explanatory, the only non-trivial rule is replicated alphabetized parallel. Altogether, I have implemented 24 regular firing rules, 32 τ-rules and 12 ✓-rules. Appendix A contains a table with all implemented inference rules.

To summarise, the presented approach allows for a compact representation of proof trees and, at the same time, guarantees that all proof trees are syntactically correct with respect to the firing rules. The proof tree verifier validates the side conditions and extracts the conclusion of the proof tree, i.e. the representation of the transition $P \xrightarrow{e} P'$. This can be done with a single traversal of the proof tree.

With the presented scheme, the Haskell implementation of a proof tree verifier for a firing rule becomes a merely syntactical translation of the firing rule. The proof tree verifier and the latex code for the firing rules have been written by hand, but in principle it would be possible to mechanically generate one representation from the other. The next sections explain an approach for generating proof trees.

4.2.4 Generation of Proof Trees for τ and ✓

This section describes the functions for generating τ and ✓ proof trees. Both functions are implemented in module `EnumerateEvents` (B.1.7). Since τ and ✓ rules are pure syntactic rules without side conditions on events, it is relatively easy to generate the proof trees.

Example

I will explain the generation of τ proof trees for the example of an alphabetized parallel process $Proc = P \;{}_{pc}\|_{qc}\; Q$. This process is encoded as:

```
AParallel pc qc p q
```

The following code generates the τ-rules for an alphabetizes parallel process.

Listing 4.5: Generator for τ proof trees for alphabetized parallel

```
1  tauTransitions :: forall i. BL i ⇒ Process i → Search (RuleTau i)
2  tauTransitions proc = case proc of
3    AParallel pc qc p q
4      →        (AParallelTauL pc qc <$> tauTransitions p <*> pure q)
5       ‘mplus‘ (AParallelTauR pc qc p <$> tauTransitions q)
6       ‘mplus‘ (AParallelTickL pc qc <$> tickTransitions p <*> pure q)
7       ‘mplus‘ (AParallelTickR pc qc p <$> tickTransitions q)
```

The above code can be read as follows: One makes a case switch on the structure of the process (Line 2). Lines 3 to 7 show the part that covers the alphabetized parallel operation. Other process operations are not shown.

The process can perform four different kinds of τ transitions defined by the following firing rules: R-56, R-57, R-58 and R-59. The corresponding constructors in the `RuleTau` data type are `AParallelTauL`, `AParallelTauR`, `AParallelTickL` and `AParallelTickR`. The proof trees for those firing rules are generated in Lines 4 to 7. The alternatives are joined with `mplus`.

For example, rule R-56 handles the propagation of a τ-event of P:

$$\frac{P \xrightarrow{\tau} P'}{P \;{}_X\|_Y\; Q \xrightarrow{\tau} P' \;{}_X\|_Y\; Q}$$

This rule is implemented with the following expression (Line 4):

```
AParallelTauL pc qc <$> tauTransitions p <*> pure q
```

Basically this means that one obtains a proof tree with this rule by applying the constructor `AParallelTauL` to four arguments:

1. pc ≡ the closure set *pc*
2. qc ≡ the closure set *qc*
3. tauTransitions p ≡ all possible τ-transition of P
4. pure q ≡ the single process Q

I use <$>, <*> and pure from the Applicative class and mplus from the MonadPlus class.[3] mplus models a non-deterministic choice between two alternatives. The Applicative class is a super class of Monad. In the presented code, <$>, <*> and mplus work a little bit like a mathematical cross-product.

I will not explain in detail what the MonadPlus class and the Applicative class are. A good overview of these concepts can be found in [64]. I hope that it is possible to get an intuition about how the above code works without understanding all the details.

Technically, the presented functions are defined in terms of a type constructor Search that determines the monad which actually implements the non-determinism (Line 1). Currently I use the Search monad from the tree-monad package [13]. It would also be possible to use the Logic monad [27] from the logict package or the plain old list nmonad.

```
type Search a = [a]
```

The choice of the monad can have some impact on the performance and it also determines whether the implementation uses backtracking or breadth first search and whether one can use parallel search strategies.

Translation to List Comprehensions

The code from Listing 4.5 works with any instance of MonadPlus, i.e. with any underlying implementation of non-determinism. Using mplus,<$> and <*> is the idiomatic Haskell implementation of the function.

However, the recent releases of GHC support an extension which allows one to also reuse the syntax of list comprehensions with any monad.[4] With this extension Listing 4.5 can be rewritten to list comprehensions which *may be* more readable.

Listing 4.6: Generator for τ proof trees with list comprehensions

```
tauTransitions proc = case proc of
    AParallel pc qc p q
      →       [AParallelTauL pc qc h q  | h ← tauTransitions p]
         ++   [AParallelTauR pc qc p h  | h ← tauTransitions q]
         ++   [AParallelTickL pc qc h q | h ← tickTransitions p]
         ++   [AParallelTickR pc qc p h | h ← tickTransitions q]
  where (++) = mplus
```

The function ++, which by default appends two lists, can also be redefined to be more generic. Listing 4.6 behaves exactly like Listing 4.5 and, in the case of the plain old list monad, it also behaves exactly like plain old list comprehensions.

[3]Later, I will also use mzero, which stands for a failed computation, or in other words a computation with zero alternatives.

[4]This is a recent extension. It was not used or not available at the time of writing most of the presented code.

Complete Code

For reference, this is the complete code for the construction of τ proof trees:

```
tauTransitions :: forall i. BL i ⇒ Process i → Search (RuleTau i)
tauTransitions proc = case proc of
  SwitchedOff p → tauTransitions $ switchOn p
  Prefix {} → mzero
  ExternalChoice p q
    →       (ExtChoiceTauL <$> tauTransitions p <*> pure q)
     `mplus` (ExtChoiceTauR p <$> tauTransitions q)
  InternalChoice p q
    →       (return $ InternalChoiceL p q)
     `mplus` (return $ InternalChoiceR p q)
  Interleave p q
    →       (InterleaveTauL <$> tauTransitions p <*> pure q)
     `mplus` (InterleaveTauR p <$> tauTransitions q)
     `mplus` (InterleaveTickL <$> tickTransitions p <*> pure q)
     `mplus` (InterleaveTickR p <$> tickTransitions q)
  Interrupt p q
    →       (InterruptTauL <$> tauTransitions p <*> pure q)
     `mplus` (InterruptTauR p <$> tauTransitions q)
  Timeout p q
    →       (TimeoutTauR <$> tauTransitions p <*> pure q)
     `mplus` (return $ TimeoutOccurs p q)
  Sharing p c q
    →       (ShareTauL c <$> tauTransitions p <*> pure q)
     `mplus` (ShareTauR c p <$> tauTransitions q)
     `mplus` (ShareTickL c <$> tickTransitions p <*> pure q)
     `mplus` (ShareTickR c p <$> tickTransitions q)
  AParallel pc qc p q
    →       (AParallelTauL pc qc <$> tauTransitions p <*> pure q)
     `mplus` (AParallelTauR pc qc p <$> tauTransitions q)
     `mplus` (AParallelTickL pc qc <$> tickTransitions p <*> pure q)
     `mplus` (AParallelTickR pc qc p <$> tickTransitions q)
  Seq p q
    →        (SeqTau <$> tauTransitions p <*> pure q)
      `mplus` (SeqTick <$> tickTransitions p <*> pure q)
  Hide hidden p → (do
    e ← anyEvent ty hidden
    rule ← buildRuleEvent e p
    return $ Hidden hidden rule)
   `mplus` (HideTau hidden <$> tauTransitions p)
  Stop → mzero
  Skip → mzero
  Omega → mzero
  AProcess _n → mzero
  RepAParallel l → mzero -- TODO ! tau for replicated AParallel
  Renaming rel p → RenamingTau rel <$> tauTransitions p
  Chaos c → return $ ChaosStop c
  LinkParallel rel p q
    →        (LinkTauL rel <$> tauTransitions p <*> pure q)
      `mplus` (LinkTauR rel p <$> tauTransitions q)
      `mplus` (LinkTickL rel <$> tickTransitions p <*> pure q)
      `mplus` (LinkTickR rel p <$> tickTransitions q)
      `mplus` mkLinkedRules rel p q
  where
    ty = (undefined :: i)
```

For most of the CSP operations, this code boils down to listing the corresponding firing rules and making the recursive calls.

The special cases are hiding and linked parallel composition (Rule R-37 and Rule R-68). Both hiding and linked parallel composition can turn a regular event into a τ event. Therefore, they both rely on calling the proof generators for regular events (`buildRuleEvent` which is described later). Currently, the function for linked parallel compositions does a naïve brute-force enumeration, which might represent an opportunity for future optimizations.

```
mkLinkedRules :: forall i. BL i
   ⇒ RenamingRelation i
   → Process i
   → Process i
   → Search (RuleTau i)
mkLinkedRules rel p q = do
  (e1, r1) ← rules1
  (e2, r2) ← rules2
  guard $ isInRenaming ty rel e1 e2
  return $ LinkLinked rel r1 r2
  where
    rules1 :: Search (Event i, RuleEvent i)
    rules1 = rules (getRenamingDomain ty rel) p
    rules2 = rules (getRenamingRange ty rel) q
    rules :: [Event i] → Process i → Search (Event i, RuleEvent i)
    rules s proc = do
      e ← s
      r ← buildRuleEvent e proc
      return (e,r)
    ty = (undefined :: i)
```

✓-Rules

The code for ✓-rules follows the same scheme:

```
tickTransitions :: BL i ⇒ Process i → Search (RuleTick i)
tickTransitions proc = case proc of
  SwitchedOff p → tickTransitions $ switchOn p
  Prefix {} → mzero
  ExternalChoice p q
    →        (ExtChoiceTickL <$> tickTransitions p <*> pure q)
     `mplus` (ExtChoiceTickR p <$> tickTransitions q)
  InternalChoice _p _q → mzero
  Interleave Omega Omega → return $ InterleaveOmega
  Interleave _ _ → mzero
  Interrupt p q → InterruptTick <$> tickTransitions p <*> pure q
  Timeout p q → TimeoutTick <$> tickTransitions p <*> pure q
  Sharing Omega c Omega → return $ ShareOmega c
  Sharing _ _ _ → mzero
  AParallel c1 c2 Omega Omega → return $ AParallelOmega c1 c2
  AParallel _ _ _ _ → mzero
```

```
17    Seq _p _q → mzero
18    Hide c p → HiddenTick c <$> tickTransitions p
19    Stop → mzero
20    Skip → return SkipTick
21    Omega → mzero
22    AProcess _n → mzero
23    RepAParallel l → if all (isOmega ∘ snd) l
24      then return $ RepAParallelOmega $ map fst l
25      else mzero
26    Renaming rel p → RenamingTick rel <$> tickTransitions p
27    Chaos _ → mzero
28    LinkParallel rel Omega Omega → return $ LinkParallelTick rel
29    LinkParallel _ _ _ → mzero
```

The special case is the introduction of the ✓-event (Rule R-25) in Line 20. Most CSP operations do not propagate ✓-events.[5] Instead they use a synchronized form of termination (See rules R-29, R-30, R-31, R-32 and R-36). To implement this concisely, I use two pattern matches for this process operation (see line 9/10, line 13/14 and line 15/16. For all other operations there is exactly one case per operation in the case–of switch.

4.2.5 Naïve Generation of Proof Trees for Regular Transitions

This section describes a naïve approach for generating the proof trees for regular events. In the previous sections, I have explained how to generate τ- and ✓-proof trees. The difference with regular transitions is that proof trees for regular transitions can contain additional side conditions on the event of the transition. A naïve approach is, to iterate over all elements of Σ and to compute the transitions for each event in Σ in turn.

In other words, the problem of generating all proof trees of a process is reduced to the problem of generating the proof trees of a process for a fixed event. With a fixed event, it is easy to check the side conditions at the same time when generating the proof trees. In CSP$_M$, channels must be explicitly declared and the set Σ of all events is always fixed. Therefore, it is always possible to enumerate Σ and to consider one event after the other. Of course, this approach may be inefficient, depending on the size of Σ and the exact structure of the process. I will address this problem in Section 4.3.

Iterating over all events is done in function `eventTransitions`.

```
eventTransitions :: forall i.
     BL i
  ⇒ Sigma i
  → Process i
  → Search (RuleEvent i)
eventTransitions sigma p = do
  e ← anyEvent ty sigma
  buildRuleEvent e p
  where
    ty = (undefined :: i)
```

[5]An exception is external choice (R-33 and R-34).

```
anyEvent :: forall i. BL i ⇒ i → EventSet i → Search (Event i)
anyEvent ty sigma
  = anyOf $ eventSetToList ty sigma
```

`eventTransitions` has two arguments: Σ and the given process. The function `eventTransitions` simply calls `buildRuleEvent` for each event in turn and passes this event as the first argument. `buildRuleEvent` does the actual work. The functions `event-Transitions` and `enumRuleEvent` are implemented in module `EnumerateEvents` (B.1.7).

The function `buildRuleEvent` generates the proof trees for a process and for one fixed event. It is structured similarly to `tickTransitions` and `tauTransitions` from the previous section. The function non-deterministically computes the rules for a CSP operation and recursively calls the proof tree generator for the premises. I abbreviate the recursive call to `buildRuleEvent` with `rp` (line 59).

```
1  buildRuleEvent :: forall i. BL i
2    ⇒ Event i
3    → Process i
4    → Search (RuleEvent i)
5  buildRuleEvent event proc = case proc of
6    SwitchedOff p → rp $ switchOn p
7    Prefix p  → case (prefixNext p event :: Maybe (Process i)) of
8      Nothing → mzero
9      Just _ → return $ HPrefix event p
10   ExternalChoice p q
11     →       (ExtChoiceL <$> rp p <*> pure q)
12      `mplus` (ExtChoiceR p <$> rp q)
13   InternalChoice _ _ → mzero
14   Interleave p q
15     →       (InterleaveL <$> rp p <*> pure q)
16      `mplus` (InterleaveR p <$> rp q)
17   Interrupt p q → (NoInterrupt <$> rp p <*> pure q)
18      `mplus` (InterruptOccurs p <$> rp q)
19   Timeout p q → TimeoutNo <$> rp p <*> pure q
20   Sharing p c q → if member ty event c
21       then Shared c <$> rp p <*> rp q
22       else (NotShareL c <$> rp p <*> pure q)
23            `mplus` (NotShareR c p <$> rp q)
24   Seq p q → SeqNormal <$> rp p <*> pure q
25   AParallel x y p q → case (member ty event x, member ty event y) of
26     (True, True) →  AParallelBoth x y <$> rp p <*> rp q
27     (True, False) → AParallelL x y <$> rp p <*> pure q
28     (False, True) → AParallelR x y p <$> rp q
29     (False,False) → mzero
30   RepAParallel l → buildRuleRepAParallel event l
31   Hide c p → if member ty event c
32       then mzero
33       else NotHidden c <$> rp p
34   Stop → mzero
35   Skip → mzero
36   Omega → mzero
37   AProcess _n → mzero
38   Renaming rel p → (do
39     e2 ← anyEvent ty (allEvents ty)
40      guard $ isInRenaming ty rel e2 event
41      rule ← buildRuleEvent e2 p
42      return $ Rename rel event rule
43      )
44      `mplus` (do
45         guard $ not $ isInRenamingDomain ty event rel
46         RenameNotInDomain rel <$> rp p
47         )
48    Chaos c → if member ty event c
49      then return $ ChaosEvent c event
50      else mzero
51    LinkParallel rel p q → (do
52        guard $ not $ isInRenamingDomain ty event rel
53        LinkEventL rel <$> rp p <*> pure q
54      ) `mplus` (do
55        guard $ not $ isInRenamingRange ty event rel
56        LinkEventR rel p <$> rp q
57      )
58    where
59      rp = buildRuleEvent event
60      ty = (undefined :: i)
```

Side Conditions

To make sure that only correct proof trees are generated, one also has to check the side conditions for some rules. For example, the rules for alphabetized parallel (R-13, R-11 and R-12) have the side conditions that the event must be in one of the two synchronization sets (or in both). Since `buildRuleEvent` is always called with a fixed event, checking the side conditions is easy. See for example lines 25 to 29. Side conditions are built with functions from the class `BL` for example `member`, `isInRenaming`, `isInRenamingDomain` and `isInRenamingRange`.

For prefix operations I call the function `prefixNext` which is implemented in the functional sub-language (lines 7 to 9). `prefixNext` checks whether an event synchronizes with a prefix operation and, if so, it directly computes a representation of the successor process.

The Renaming Operation

A special case is the implementation of renaming (R-18). The side condition for renaming is that the *external visible* event is in a renaming relation with the event *inside the renaming operation*. CSP$_M$ supports relational renaming which means that, in principle, renaming can introduce additional non-determinism. Even if the *external visible* event is fixed, there can be several alternatives for the *internal* event of the renaming operation. For simplicity, *internal events* are generated by brute-force enumeration. In detail the function does the following:

Line 39 It generates an arbitray event `e2`. This is a full enumeration of Σ.

Line 40 It checks that (`e2`, `event`) is a member of the renaming relation.

Line 41 It recursively generates the premise using `e2`.

Additionally, one has to cover the case that the external visible event is not in the domain of the renaming relation (lines 44 to 47).

The fact that I use brute-force enumeration of events inside the renaming operation can be fatal for the performance of the proof tree generator. The naïve proof tree generator also performs a brute-force enumeration of Σ; however this enumeration is an outer loop which is independent from the structure of the process under consideration. On the other hand, the implementation of the renaming operation enumerates Σ and then recurses on the process inside the renaming operation. This means that nested renaming operations cause a nested enumeration. In other words, if a specification contains a renaming operation, which itself contains a nested renaming operation, the slowdown will be $|\Sigma|^2$. If renamings are nested three times the slowdown will be $|\Sigma|^3$.

4.3 Constraint-Based Generation of Proof Trees

In the previous section, I have described a relatively straightforward way to generate the proof trees for ✓ and τ transitions and a naïve approach for enumerating regular transitions. This approach was based on enumerating the complete set Σ.

However, iterating over Σ can be relatively inefficient, in particular if the set Σ is big and the number of events that actually occur is small. M. Leuschel [34] shows examples of such specifications and describes how constraint programming and Prolog can be used to avoid the enumeration of Σ in first place. The algorithm, that is presented in this section has been inspired by the constraint-based Prolog approach.

The algorithm is based on four ideas:

1. It processes the fields of an event from left to right.
2. It uses a data type for abstract event fields.
3. It uses a data type for proof tree skeletons, i.e. proof trees with partial information.
4. It performs an abstract interpretation of proof tree skeletons to compute new information about the event.

I will first describe the main ideas and try to give an intuition of how my constraint-based algorithm works and after that I will describe the actual implementation on source code level.

Idea 1: Process the fields from left to right

The advantage of processing event fields from left to right is that one can deal with one field after the other. It is not necessary to enumerate the complete set Σ—in the worst case, one only has to enumerate all possibilities for one event field. Another reason for this heuristic is that the first field of an event is always the channel. Often CSP$_M$ specifications are structured such that the synchronization conditions can be decided by only looking at the channel.

Although events are by definition atomic, there are several reasons why an implementation might process the fields from left to right. One of them is the scoping rule for input fields. In CSP$_M$, an input field binds an identifier that is in scope in all the following event fields. For example, `ch?x?y!x+y->P` first binds identifier `x` then identifier `y` and then outputs `x+y` on the channel. Information is always propagated from left to right.[6]

The intuition behind my approach is that a prefix operation with a multi-field event has some similarities with an nested single-field prefix operation. For example $c.1.3 \to STOP$ is a little bit like $c \to (1 \to (3 \to STOP))$. The main difference is that $c.1.3$ is atomic while $c \to (1 \to (3 \to STOP))$ performs three separate events. The left-to-right order is a heuristic which computes a super-set of all possible events. Every atomic event can be split into the smaller steps which perform one event field after the other, but not every sequence of small steps also corresponds to a valid atomic event. Therefore, an extra step is needed to check the atomicity constraint.

Idea 2: Abstract event fields

The interface between the core language and the underlying functional programming language uses an abstract view of event fields. I use the following data type for abstract event fields:

```
data PrefixFieldView i
  = FieldOut (Field i)
  | FieldIn
  | FieldGuard (FieldSet i)
```

There are only three cases for an event field:

`FieldOut` The field is an output field which communicates one fixed value.

`FieldIn` The field is an unconstrained input field.

`FieldGuard` The field is an input field, where the value is restricted to a set of alternatives.

The implementation of the core language does not distinguish between, for example, $c?x!y * 2$ and $c?y!z$. Everything related to variable names and binding values to variables is handled by the functional sub-language.

The interface between the core language and the functional sub-language consists of two functions:

1. The core language can ask for the *current* event field and the functional sub-language returns a value of `PrefixFieldView i`.
2. The core language can tell the functional sub-language a value, which it has determined for the *current* field.

There is an implicit pointer for event fields and there is always exactly one *current* field at the time.[7] Telling the value of an event field automatically advances the current-field-pointer to the next position.

In parallel to my work on CSP$_M$, the enumerator-iteratee idiom [7, 26] became popular in Haskell. My approach is very similar an enumerator-iteratee.

[6] Another justification for the left-to-right order is the infamous generic buffer feature that is implemented in FDR [34].

[7] Unfortunately, one aspect of the implementation could be a little confusing. It sometimes uses identifiers that contain the word "next" where what is actually meant is the *current* field. The *current* field is always the *next* field that will get a fixed field value.

Idea 3: Proof tree skeletons

The implementation uses a data structure which I call proof tree skeleton. The structure of proof tree skeletons is almost identical to the structure of regular proof trees, except that the events in a proof tree skeleton are only partially determined.

I will explain proof tree skeletons with an example. Suppose one wants to compute the transitions of the following process:

$$(c!1!2 \to P) \underset{\{\!|c|\!\}}{\parallel} (c?x!2 * x \to Q)$$

A proof tree for event $c!1!2$, using inference rules R-13 and R-1, is:

$$\frac{(c!1!2 \to P) \xrightarrow{c!1!2} P' \qquad (c?x!2 * x \to Q) \xrightarrow{c!1!2} Q'}{((c!1!2 \to P) \underset{\{\!|c|\!\}}{\parallel} (c?x!2 * x \to Q)) \xrightarrow{c!1!2} (P' \underset{\{\!|c|\!\}}{\parallel} Q')}\, c!1!2 \in \{\!|c|\!\}$$

My algorithm generates several proof tree skeletons with partial information. These proof tree skeletons could be depicted as:

$$\frac{(\sqcup \to P) \xrightarrow{\sqcup} P' \qquad (\sqcup \to Q) \xrightarrow{\sqcup} Q'}{((\sqcup \to P) \underset{\{\!|c|\!\}}{\parallel} (\sqcup \to Q)) \xrightarrow{\sqcup} (P' \underset{\{\!|c|\!\}}{\parallel} Q')}\, \sqcup \in \{\!|c|\!\}$$

$$\frac{(c.\sqcup \to P) \xrightarrow{c.\sqcup} P' \qquad (c.\sqcup \to Q) \xrightarrow{c.\sqcup} Q'}{((c.\sqcup \to P) \underset{\{\!|c|\!\}}{\parallel} (c.\sqcup \to Q)) \xrightarrow{c.\sqcup} (P' \underset{\{\!|c|\!\}}{\parallel} Q')}\, c.\sqcup \in \{\!|c|\!\}$$

$$\frac{(c.1.\sqcup \to P) \xrightarrow{c.1.\sqcup} P' \qquad (c.1.\sqcup \to Q) \xrightarrow{c.1.\sqcup} Q'}{((c.1.\sqcup \to P) \underset{\{\!|c|\!\}}{\parallel} (c.1.\sqcup \to Q)) \xrightarrow{c.1.\sqcup} (P' \underset{\{\!|c|\!\}}{\parallel} Q')}\, c.1.\sqcup \in \{\!|c|\!\}$$

The algorithm starts by generating the proof tree skeletons which contains no information about the events and then derives more and more concrete skeletons. In each iteration it determines the value for one extra event field. If there are several valid values for the next event field it branches non-deterministically. If there is no possible next field, the proof tree skeleton is abandoned.

There is an important invariant for proof tree skeletons. A proof tree skeleton is related to exactly one partial event. For example, it cannot happen that a proof tree skeleton contains $c.\sqcup$ and $c.1.\sqcup$ at the same time.

The data type for proof tree skeletons is almost identical to the data type for regular proof trees that was described in Section 4.2.3. The information about the partial events is not explicitly stored in the proof tree skeletons—it is only implicit.

Idea 4: Abstract interpretation of proof tree skeletons

To determine the value of the next data field, I use a technique that has some similarities with abstract interpretation. The abstract interpretation/constraint propagation consist of a simple recursive traversal of the proof tree skeleton. The algorithm maintains an abstract field value, which is just the set of all possible event fields.

For each node in the proof tree, the side conditions of the firing rule are used to constrain the abstract field value. Basically this means that the algorithm computes the intersection of the current set of possible field values and the set of values that are consistent with the side conditions.

Constraint Propagation and Closure Sets

Most of the side conditions are of the form $e \in \{\!|x|\!\}$, i.e. the side condition is that an event is an element of an event closure set. To use this form of side conditions for constraint propagation, one has to implement closure sets such that they also work with partially defined events.

The idea is that I replace the simple membership test $e \in \{\!|x|\!\}$ with the functions `viewClosureState` and `viewClosureFields`, which can return more information than just `True` or `False`. In particular, `viewClosureState` returns a value of type:

```
data ClosureView = InClosure | NotInClosure | MaybeInClosure
```

and `viewClosureFields` returns the set of possible *current* field values which are consistent with the *current* closure set.

The *current* field values and the *current* closure set depend on the partial event, which has already been fixed. The partial event is only implicitly known when calling `viewClosureState` and `viewClosureFields`.

The efficiency of the constraint-based approach crucially depends on how accurate `viewClosureState` and `viewClosureFields` work. For example, `view-ClosureState` may always return `MaybeInClosure` but the search space will only be restricted if it returns the more precise values `InClosure` or `NotInClosure`.

The constraint-based/propagation algorithm requires an extended interface between the CSP core language and the underlying function programming language. This interface is defined in module `Field` (B.1.4). It will be described in Section 4.3.6 and Section 4.3.7.

There are some extra steps that need to be done when gluing everything together. For example, if the constraint propagation finally yields a set of possible alternatives, these alternatives have to be enumerated. After the last event field has been fixed, one has to double-check that the complete event, i.e. the concatenation of all event fields, is also valid with the side conditions.

This section has informally explained the underlying ideas of the algorithm. The next sections describe the source code for this algorithm in detail.

4.3.1 Generating the Initial Proof Tree Skeletons

The function for generating the initial proof tree skeletons is called `rulePattern`. It is similar to the function for generating the ✓ and τ rules. `rulePattern` is just a big case switch over all process operations. For every operation, it returns the firing rules that are relevant and the sub-trees for the premises are built by recursively calling `rulePattern` (the recursive call is abbreviated as `rp`). The initial proof tree skeletons contain no information about the event of the transition. Therefore, `rulePattern` does not have to deal with the events or side conditions of the firing rules.

The data type for proof tree skeletons is called `RuleField`. It is declared in module `FieldConstraintsSearch` (B.1.8) and closely resembles the type for regular proof trees (`RuleEvent`). The constructor names in `RuleField` are the same as those of `RuleEvent` except they are prefixed with the letter `F`. The main difference is that skeletons contain `ClosureState` and `PrefixState` instead of `EventSet` and `Prefix`. The data types `ClosureState` and `PrefixState` store information about the event fields that have been processed. They have to be initialized by calling `prefixStateInit` and `initClosure`.

```
rulePattern :: forall i.
  BF i ⇒ Event.EventSet i → Process i → Search (RuleField i)
rulePattern events proc = case proc of
  SwitchedOff p → rp $ switchOn p
  Prefix p → return $ FPrefix $ prefixStateInit ty p
  ExternalChoice p q
    → joinRepExtChoiceParts (initRepExtChoicePart events p)
                            (initRepExtChoicePart events q)
  InternalChoice _p _q → mzero
  Interleave p q
    →       (FInterleaveL <$> rp p <*> pure q)
     `mplus` (FInterleaveR p <$> rp q)
  Interrupt p q → (FNoInterrupt <$> rp p <*> pure q)
     `mplus` (FInterrupt p <$> rp q)
  Timeout p q → FTimeout <$> rp p <*> pure q
  Sharing p c q
    →       (FShared (initClosure c) <$> rp p <*> rp q)
     `mplus` (FNotShareL (initClosure c) <$> rp p <*> pure q)
     `mplus` (FNotShareR (initClosure c) p <$> rp q)
  AParallel pc qc p q
    →       (FAParallelL (initClosure pc) (initClosure qc)
                 <$> rp p <*> pure q)
     `mplus` (FAParallelR (initClosure pc) (initClosure qc)
                 <$> pure p <*> rp q)
     `mplus` (FAParallelBoth (initClosure pc) (initClosure qc)
                 <$> rp p <*> rp q)
  Seq p q → FSeqNormal <$> rp p <*> pure q
  Hide c p → FNotHidden (initClosure c) <$> rp p
  Stop → mzero
  Skip → mzero
  Omega → mzero
  AProcess _n → mzero
  RepAParallel l → return $ FRepAParallel $ initRepAParallel l
  Renaming rel p → return $ FRenaming rel p
  Chaos c → return $ FChaos $ initClosure c
  LinkParallel rel p q
    →         (FLinkEventL rel <$> rp p <*> pure q)
       `mplus` (FLinkEventR rel p <$> rp q)

  where
    ty = (undefined :: i)
    initClosure = closureStateInit ty
    rp = rulePattern events
```

4.3.2 Constraint Propagation

The recursive traversal, which carries out the constraint propagation for fields, is implemented in function `probField`. It uses the Monad `PropM` which maintains a state of type `FieldSet` (the abstract field value) and allows an early failure via the underlying `Maybe`.

```
type PropM i a = StateT (FieldSet i) Maybe a
```

The function is a big case switch on the constructors of `RuleField`. The full function has about 80 lines (see `FieldConstraintsSearch` Appendix B.1.8). I will only discuss the cases for prefix and alphabetised parallel since these are the most interesting rules.

```
1  propField :: forall i. BF i ⇒ RuleField i → PropM i ()
2  propField rule = case rule of
3    FPrefix p → case viewPrefixState ty p of
4      FieldOut f → fixField f
5      FieldIn → return ()
6      FieldGuard g → restrictField $ λe → intersection ty e g
7    FAParallelL c1 c2 r _ → case (closureState c1,closureState c2) of
8      (NotInClosure,_) → impossibleRule
9      (_,InClosure) → impossibleRule
10     _ → do
11       restrictField $ λe → intersection ty e (closureFields c1)
12       propField r
13   FAParallelR c1 c2 _ r → case (closureState c1,closureState c2) of
14     (_,NotInClosure) → impossibleRule
15     (InClosure,_) → impossibleRule
16     _ → do
17       restrictField $ λe → intersection ty e (closureFields c2)
18       propField r
19   FAParallelBoth c1 c2 r1 r2 → case (closureState c1,closureState c2) of
20     (NotInClosure,_) → impossibleRule
21     (_,NotInClosure) → impossibleRule
22     _ → do
23       restrictField $ λe → intersection ty e (closureFields c1)
24       restrictField $ λe → intersection ty e (closureFields c2)
25       propField r1
26       propField r2
```

For `FPrefix` (line 3), the functions makes a case distinction on the current event field. The field can be either an output field, an input field or a guarded input.

FieldOut/Line 4 The value of the field is immediately known.

FieldIn/Line 5 One cannot gain any information.

FieldGuarded/Line 6 The set of possible events is restricted according to the guard.

The next cases are for alphabetized parallel operations ($P\ {}_X\|_Y\ Q$). `FAParallelL` (line 7) is for an operation where only P has performed an event (R-11) and `FAParallelR` (line 13) is the analog case where only Q has taken part in the event (R-12). `FAParallelBoth` is for transitions where P and Q synchronize (R-13).

In detail, the function case for `FAParallelBoth` does the following:

Line 19 It checks whether the event fields that have been processed so far are still consistent with the closure sets X and Y. There are three possible cases.

Lines 20,21 / Case one and two If the event is not an element of any of the synchronization sets X or Y, one can abandon the proof skeleton.

Lines 22 to 26 / Case three Otherwise, the proof skeleton is still possible.

Lines 23 to 24 In this case information from the synchronization sets X and Y is propagated.

Lines 25,26 After that, the function recursively traverses on the sub-trees.

`probField` uses the following helper definitions:

```
restrictField :: (FieldSet i → FieldSet i) → PropM i ()
restrictField fkt = do
    possible ← get
    let restricted = fkt possible
    if Field.null ty restricted
        then impossibleRule
        else put restricted

fixField :: Field i → PropM i ()
fixField e = do
    possible ← get
    if member ty e possible
        then put $ singleton ty e
        else impossibleRule

impossibleRule :: PropM i ()
impossibleRule = mzero
closureState :: ClosureState i → ClosureView
closureState = viewClosureState ty
closureFields :: ClosureState i → FieldSet i
closureFields = viewClosureFields ty
ty = (undefined :: i)
```

An interesting observation is that all functions have return type `m ()`. In other words, this is a degenerated use-case of a monad. A news group article[8] suggests that in this case, it is better to use the Monoid idiom. However, I have not yet investigated if this makes further optimizations possible.

4.3.3 Fixing a Field Value in the Proof Tree Skeleton

The function `nextField` fixes a field value in the proof tree skeleton.

```
nextField :: forall i. BF i
  ⇒ RuleField i
  → Field i
  → Search (RuleField i)
```

[8]Unfortunately, the exact reference was lost.

It is basically a recursive traversal of the tree which calls `prefixStateNext` and `closureStateNext` on all values of type `PrefixState` and `ClosureState`. This also advances the implicit current event-field-pointer, which `PrefixState` and `ClosureState` refer to.

4.3.4 Converting a Proof Tree Skeleton to a Proof Tree

After the values for all event fields are fixed, I convert the proof tree skeleton to a regular proof tree. This is done with function `lastField`.

```
lastField :: forall i. BF i
  ⇒ RuleField i
  → Event.Event i
  → Search (RuleEvent i)
```

The second argument of `lastField` is a *multi-field* event, which is the concatenation of all event fields that have been computed before. In the previous steps, the atomicity of events was not taken into account; instead, the side conditions have only been checked field-wise. As a consequence these steps actually compute a super-set of all possible transitions. To fix this, `lastField` checks the side conditions again, on the event level, and it filters out any illegal transitions.

4.3.5 The Main Loop

This section describes the clue code that calls the constraint-based proof tree generator. In particular, it describes the inner loop that iterates over the fields of a prefix operation. I describe the functions, more or less, in a top-down manner. The functions are all defined in terms of the monad `Search a`, which handles enumeration and non-deterministic choice.

The external interface to the proof tree generator is the function `computeAllRules`, which just calls the generators for ✓, τ and regular transitions and returns the union of the results (lines 5 to 8). For regular transitions, `computeAllRules` calls `computeNext` (line 5).

```
1  computeAllRules ::  forall i. BF i
2    ⇒ Event.EventSet i
3    → Process i
4    → [Rule i]
5  computeAllRules events p
6    =  (liftM EventRule $ computeNext events p)
7           ‘mplus‘ (liftM TickRule $ buildRuleTick p)
8           ‘mplus‘ (liftM TauRule $ buildRuleTau p)
9
10 computeNext ::
11   BF i ⇒ Event.EventSet i → Process i → Search (RuleEvent i)
12 computeNext events proc = liftM snd $ computeNextE events proc
13
14 computeNextE :: BF i
15   ⇒ Event.EventSet i
16   → Process i
17   → Search (Event.Event i, RuleEvent i)
18 computeNextE events proc = rulePattern events proc ≫= runFields events
```

`computeNext` is just a small wrapper for `computeNextE` that throws away the event. `computeNextE` first calls `rulePattern` to generate the skeletons and then passes these to `runFields` (line 18). It returns a combination of proof trees and the corresponding transitions.

```
 1  runFields :: forall i. BF i ⇒
 2    Event.EventSet i → RuleField i → Search (Event.Event i, RuleEvent i)
 3  runFields events r = do
 4        let baseEvents = closureStateInit ty events
 5        (chan,next) ← enumField (viewClosureFields ty baseEvents ) r
 6        (e,final) ← loopFields
 7           (closureStateNext ty baseEvents chan)
 8           [chan]
 9           next
10           (channelLen ty chan -1)
11        let event = joinFields ty $ reverse e
12        rule ← lastField final event
13        return (event,rule)
14     where ty = (undefined :: i)
```

The function `runFields` has three main tasks:

Line 5 It computes the first field of the event by calling `enumField`. In CSP$_M$, the first field must always be a channel identifier. The channel is needed to compute the number of channel fields (line 10).

Lines 6 to 10 It calls `loopFields` to determine the rest of the event. It passes the number of fields in the event as an argument.

Line 12 After the last field has been fixed, it calls `lastField` to convert the proof tree skeleton to a regular proof tree.

`runFields` also has to do some plumbing with the event set Σ which is converted to a `ClosureState` with `closureStateInit` (line 4) and used to compute the initial set of field values for the constraint propagation (`viewClosureFields`, line 5).

```
 1  loopFields :: forall i. BF i ⇒
 2       ClosureState i
 3     → [Field i]
 4     → RuleField i
 5     → Int
 6     → Search ([Field i], RuleField i)
 7  loopFields _ eventAcc rule 0 = return (eventAcc, rule)
 8  loopFields closureState eventAcc rule n = do
 9        (f,next) ← enumField (viewClosureFields ty closureState) rule
10        loopFields
11          (closureStateNext ty closureState f)
12          (f:eventAcc)
13          next
14          (n-1)
15     where ty = (undefined :: i)
```

The arguments of `loopFiled are`:

Line 2 & Line 11 The `closureState` which represents Σ.

Line 3 & Line 12 An accumulator for the fields computed so far.

Line 4 & Line 13 The proof tree skeleton.

Line 5 & Line 14 The number of fields that are left to go.

`loopFields` is a simple tail-recursive loop:

Line 7 The loop runs until no fields are left.

Line 9 The body of the loop computes the next field by calling `enumField`.

Lines 10 to 14 The function recursively calls `loopFields` for the next field.

The first argument of `enumField` is the set of possible event fields, which is used as the initial value for the constraint propagation. It is computed with `(viewClosureFields ty closureState)` (line 9). The type `ClosureState` is used to represent event closure sets. The algorithm starts with a representation of Σ. The function `runFields` computes the value for the channel and restricts the `ClosureState` with that information (line 7 of `runFields`). In the function `loopFields f` is used to restrict the `ClosureState` further after the field value `f` has been determined (line 11).

```
enumField :: forall i. BF i
  ⇒ FieldSet i
  → RuleField i
  → Search (Field i, RuleField i)
enumField top r = case execStateT (propField r) top of
      Just s → do
        f ← fieldSetToList ty s
        nr ← nextField r f
        return (f ,nr )
      Nothing → mzero
  where ty = (undefined :: i)
```

`enumField` is the driver for `probField` and `nextField`. It executes the constraint propagation for the current proof tree skeleton, and iterates over the returned set of possible field values. For every possible value it calls `nextField`.

4.3.6 Modeling Multi-field Events

In CSP$_M$, it is possible to define channels that contain data fields, for example `channel c:{1..10}.{1..10}`. This section describes how I support events with data fields. Technically, the functions of this section belong to the type class `BF`, which is a sub-class of `BL`. It is defined in module `Field` (B.1.4).

I use the following functions to process the events of a field from left to right.

```
  prefixStateInit :: i→Prefix i→PrefixState i
  prefixStateNext :: i→PrefixState i→Field i→Maybe (PrefixState i)
  prefixStateFinalize :: i→PrefixState i→Maybe (Prefix i)
  viewPrefixState :: i→PrefixState i→PrefixFieldView i
```

The order in which these functions get called follows a simple protocol. `prefixStateInit` is always called first. This function converts a `Prefix` to a `PrefixState`. Next, `viewPrefixState` is called to examine the first event field. `viewPrefixState` returns a value of type `PrefixFieldView i` to characterize the current event field.

```
data PrefixFieldView i
  = FieldOut (Field i)
  | FieldIn
  | FieldGuard (FieldSet i)
```

Once the value of a field has been determined, `prefixStateNext` is called to proceed to the next field. One argument of `prefixStateNext` is the value of the field. This enables further processing of the value in the functional sub-language, in particular `prefixStateNext` checks if the value does synchronize and returns `Nothing` if it does not.

This sequence of `viewPrefixState` followed by `prefixStateNext` is repeated for each event field. `viewPrefixState` always returns information about the current event field and `prefixStateNext` advances to the next event field. After the last field has been processed, `prefixStateFinalize` is called to convert `PrefixState` back to `Prefix`.

The functional sub-language which implements these functions also defines the data structure `PrefixState` to maintain information about the intermediate state of a prefix.

4.3.7 Event Sets for Multi-field Events

The side conditions that have to be checked in the firing rules always test whether an event is an element of an event closure set. These tests can only succeed or fail. However, to generate the proof trees with the constraint-based approach, I need more possibilities to manipulate and query closure sets. For this purpose, the class `BF` defines the following function :

```
  closureStateInit :: i → EventSet i → ClosureState i
  closureStateNext :: i → ClosureState i → Field i → ClosureState i
  closureRestore   :: i → ClosureState i → EventSet i
  viewClosureState :: i → ClosureState i → ClosureView
  viewClosureFields :: i → ClosureState i → FieldSet i
  seenPrefixInClosure :: i → ClosureState i → Bool

data ClosureView
  = InClosure
  | NotInClosure
  | MaybeInClosure
  deriving (Show,Eq,Ord)
```

The type `ClosureState` is used to store an event closure set plus additional information about a fixed prefix of the fields of an event. `viewClosureState` queries a closure set while stepping through the event fields. In case the query returns `InClosure`, this means that any completion of the partial event seen so far is for sure in the event closure set. `NotInClosure` means that it is not possible to complete the fields seen so far to an event that is member of the closure set. If neither `InClosure` nor `NotInClosure` is valid, `viewClosureState` returns `MaybeInClosure`.

The functions `closureStateInit`, `closureStateNext` and `closureRestore` are used to step through an event closure set similar to the protocol described for prefix fields. `viewClosureFields` computes the projection of a `ClosureState` to the next undefined field, with respect to the part of the event, that has already been determined.

The interpreter implements event closure sets with a trie-like data set. As the algorithm processes the event fields from left to right the trie will be traversed from the root to the nodes.

Finally, the class `BF` defines the following functions that are used manipulate fields and sets of fields.

```
fieldEq :: i → Field i → Field i → Bool
member :: i → Field i → FieldSet i → Bool
intersection :: i → FieldSet i → FieldSet i → FieldSet i
difference :: i → FieldSet i → FieldSet i → FieldSet i
union :: i → FieldSet i → FieldSet i → FieldSet i
null :: i → FieldSet i → Bool
singleton :: i → Field i → FieldSet i
insert :: i → Field i → FieldSet i → FieldSet i
delete :: i → Field i → FieldSet i → FieldSet i
fieldSetToList :: i → FieldSet i → [Field i]
fieldSetFromList :: i → [Field i] → FieldSet i

joinFields :: i → [Field i] → Event i
splitFields :: i → Event i → [Field i]
channelLen :: i → Field i → Int
```

4.3.8 Critique

The constraint based-approach, which has been described in this section, was designed with one particular use-case in mind, namely to deal with large sets of Σ and complicated synchronizations. It is unclear if this special use-case justifies the efforts. Furthermore, the data structures, which where used for the underlying functional sub-language, do not completely avoid the enumeration of Σ yet and some specifications that should actually be fast are still slow.

Overall, the described approach seems to be convoluted and inefficient. To compute only one event, the algorithm needs to pass over the proof tree skeletons many times. An update of a proof tree skeleton effectively computes a new version, which means that the functions cause a high load on the garbage collector. I had to define four separate functions that deal with proof tree skeletons.

1. `rulePattern` (Section 4.3.1) to generate the proof tree skeleton.
2. `probField` (Section 4.3.2) for constraint propagation.
3. `nextField` (Section 4.3.3) to move to the next field.
4. `lastField` (Section 4.3.4) to convert the skeleton to a regular proof tree.

In principle, each supported firing rule has to be covered in each of the four functions. Plumbing together all the functions is cumbersome and finally the data flow is still relatively fixed compared to, for example, the Prolog implementation. An underlying idea of constraint programming is that the data flow should be as flexible as possible, whereas in the presented approach information is only propagated from a field on the left to field on the right.

The overall complexity of the implementation is relatively high compared to other parts of my CSP$_M$ animator. Most functions are relatively cleanly structured as one big case switch and one recursive traversal of a recursive data structure, but still there is also need for glue code. The complete module (`FieldConstraintsSearch`, B.1.8) is one of the biggest modules of the project with 592 lines of code.

Positive Critique

Implementing the described algorithm was interesting. I have gained experience and developed new ideas for other approaches. Replacing this algorithm with something better is interesting future work. It is surprising that although there are obvious inefficiencies, my implementation is never-the-less often faster than ProB (See Section 8.5.4).

My experience with Haskell was overall positive. The presented approach is complicated, but at least it was possible to keep the implementation concise. Most functions of the implementation consist of a big case switch over all supported firing rules. Often the case definition for one rule only consists of one line of code. Monads and higher order combinators where very useful in expressing non-trivial concepts, like for example non-determinism.

The code is complicated, but the complexity is hopefully still manageable. My personal experience was that I did not have to spend too much time on prolonged debugging sessions. Explicit proof trees and pure functions turned out to be helpful concepts for the implementation of the constraint-based proof tree generator.

4.4 Testing the Implemented Semantics with QuickCheck

In the previous sections, I have described two Haskell implementations. A naïve enumeration-based implementation and a constraint-based approach that uses several optimizations, e.g. to deal efficiently with multi-field events. The constraint-based approach is relatively complex and it is it is far from obvious if it is equivalent to the firing rules.

A standard technique to improve the confidence in an implementation is testing, typically in the form of unit tests or regression tests. This kind of testing has the drawback that one usually only tests for those cases which the programmer has thought of beforehand, and for those bugs that have already shown up. Manually writing exhaustive test cases for symbolic computations, like an CSP$_M$ animator, is difficult.

I have therefore decided to use an alternative approach which is known to Haskell community as QuickCheck [61]. The basic idea of QuickCheck is to test abstract properties on a set of automatically generated test cases.

For example, one could test that the reverse function is its own inverse with the following property:

```
prob_rev :: String → Bool
prob_rev x = (reverse $ reverse x) == x
```

To run the tests, one has to pass the property to the function

```
quickCheck :: Testable prop ⇒ prop → IO ().
```

A property can be any instance of type class `Testable`. For my application the following two instances are most important:
1) Boolean values are testable.
2) If one know hows to generate arbitrary arguments, then one can test functions that return testable values. In Haskell syntax:

```
(Arbitrary a, Show a, Testable prop) ⇒ Testable (a → prop)
```

`Arbitrary` is the class of types for which one can generate arbitrary values.

A nice application is to use QuickCheck for refinement checking. For example, one can check that an efficient sorting algorithm returns the same results as an alternative, less efficient but simpler algorithm with the following property:

```
prop_sort :: [Integer] → Bool
prop_sort l = mergeSort l == bubbleSort l
```

QuickCheck tests are not a formal proof. A test is only as good as the generated test cases. One approach to improve the confidence in the tests is to combine QuickCheck with a code coverage tool.

I use QuickCheck to test the completeness and soundness of my proof tree generator and also to test the equivalence of the naïve proof tree generator and the constraint base proof tree verifier. My implementation passes all those tests.

4.4.1 Proof Tree Verifier as a Specification of the Proof Tree Generator

In Section 4.2.3, I have described a proof tree verifier which tests that a proof tree is consistent with the firing rules. An important property of the proof tree verifier is that it is derived by a simple syntactic translation from the firing rules to Haskell syntax. *I will therefore assume that the proof tree verifier is a valid implementation of the firing rules.* In other words, I will use the source code of the proof tree verifier as a formal specification of the firing rules.

Under the assumption that the proof tree verifier is correct, QuickCheck can be used to gain confidence in the correctness of the proof tree generators. I use QuickCheck to show the following properties:

Soundness The proof tree generator returns only valid proof trees.

Completeness The proof tree generator returns all possible proof trees.

Checking Soundness

Semi-formally, if $generateProofTrees$ is the function that generates a set of proof trees and $realProofTrees$ is the real set of valid proof trees, then soundness can be expressed as:

$$\forall\, P. generateProofTrees(P) \subseteq realProofTrees(P)$$

To check for soundness it is in principle sufficient to generate random processes and check that the generated proof trees are correct. In other words, one just has to run the proof tree verifier on the output of the proof tree generator.

However, there is one catch. Just generating arbitrary processes will result in very poor code coverage. For example, the chance that a randomly generated process can perform a transition, involving a synchronised transition of several sub-processes is very low.

As a solution I use the following trick. I use an indirect method for generating the processes, that serve as test cases. Instead of arbitrary processes, I generate arbitrary proof trees. A proof tree justifies a transition $P \xrightarrow{e} P'$. I use the function `viewProcBefore` to extract the process P from the proof tree and if everything is correct, P is guaranteed to have at least one interesting transition (namely $P \xrightarrow{e} P'$).

Here are examples of the concrete QuickCheck properties:

```
sound_EnumRuleTick :: CSP1 i ⇒ RuleTick i → Bool
sound_EnumRuleTick r
  = all (checkRule proc ∘ TickRule) $ EnumNext.tickTransitions proc
  where proc = viewProcBefore $ TickRule r

sound_EnumRuleTau :: CSP1 i ⇒ RuleTau i → Bool
sound_EnumRuleTau r
  = all (checkRule proc ∘ TauRule) $ EnumNext.tauTransitions proc
  where proc = viewProcBefore $ TauRule r

sound_EnumRuleEvent :: forall i. CSP1 i ⇒ RuleEvent i → Bool
sound_EnumRuleEvent r
  = all (checkRule proc ∘ EventRule)
      $ EnumNext.eventTransitions sigma proc
  where
    proc = viewProcBefore $ EventRule r
    sigma = allEvents (undefined :: i)

checkRule :: CSP1 i ⇒ Process i → Rule i → Bool
checkRule proc r
  = case viewRuleMaybe r of
      Nothing → False
      Just (p,_,_) → p == proc
```

I use separate properties to check τ, ✓ and regular transitions. The above properties check the naïve proof tree generator. The properties for checking the constraint-based proof tree generator look similar. Note that since the processes that are used for testing are generated indirectly via arbitrary proof trees, deadlock processes are not tested. This is not a principle restriction and could be fixed easily.

Checking Completeness

Completeness means that my proof tree generator finds all possible proof trees. Semi-formally:

$$\forall P.realProofTrees(P) \subseteq generateProofTrees(P)$$

Just given that the proof tree verifier is correct, it is not clear how to check that property directly. It can, however, be checked indirectly. My approach is similar to one used for checking soundness. Instead of starting with an arbitrary process, I start with an arbitrary proof tree r. I extract the process P from the transition $P \xrightarrow{e} P'$ that is justified by the proof tree and then compute the set

s of all possible proof trees for P. The proof tree generator is complete if for any proof tree r, $r \in s$. Semi-formally, I test the property:

$$\forall\, r.(r \in \mathit{generateProofTrees}(P)) \text{ where } P \equiv \mathit{extract}(r)$$

This is the source code of some properties for checking completeness:

```
complete_enumTauRules :: CSP1 i ⇒ RuleTau i → Bool
complete_enumTauRules r
  = r ‘List.elem‘ (EnumNext.tauTransitions $ viewProcBefore $ TauRule r)

complete_enumEventRules :: forall i. CSP1 i ⇒ RuleEvent i → Bool
complete_enumEventRules r
  = r ‘List.elem‘ (EnumNext.eventTransitions sigma
     $ viewProcBefore $ EventRule r)
  where sigma = allEvents (undefined :: i)
```

Again, there are three separate properties for ✓, τ and regular transitions and again I only show the tests for the naive proof tree generator. The tests for the constraint-based generator looks similar.

4.4.2 Equality of the Naïve Proof Tree Generator and the Constraint-based Proof Tree Generator

In the previous sections, I have described how I check the soundness and completeness of the proof tree generators, assuming that the proof tree verifier is a valid implementation of the CSP firing rules.

An additional property one would like to check is that the naïve proof tree generator and the constraint-based proof tree generator always compute the same result. These are the corresponding QuickCheck properties:

```
computeNext_eq_EnumRuleEvent :: forall i. CSP2 i ⇒ RuleEvent i → Bool
computeNext_eq_EnumRuleEvent rule = ruleSet1 == ruleSet2
  where
    ruleSet1 = Set.fromList $ FieldNext.eventTransitions sigma proc
    ruleSet2 = Set.fromList $ EnumNext.eventTransitions sigma proc
    proc = viewProcBefore $ EventRule rule
    sigma = allEvents (undefined :: i)

fieldTau :: forall i. CSP2 i ⇒ RuleTau i → Bool
fieldTau rule = ruleSet1 == ruleSet2
  where
    ruleSet1 = Set.fromList $ EnumNext.tauTransitions proc
    ruleSet2 = Set.fromList $ FieldNext.tauTransitions proc
    proc = viewProcBefore $ TauRule rule

fieldTick :: forall i. CSP2 i ⇒ RuleTau i → Bool
fieldTick rule = ruleSet1 == ruleSet2
  where
    ruleSet1 = Set.fromList $ EnumNext.tickTransitions proc
    ruleSet2 = Set.fromList $ FieldNext.tickTransitions proc
    proc = viewProcBefore $ TauRule rule
```

Again, theses properties use a detour and generate the test case via an arbitrary proof tree.

4.4.3 Code Coverage Analysis

QuickCheck must be able to automatically generate test cases. For the presented approach, this means that one has to implement generators for arbitrary proof trees, processes and several other data types. The quality of the QuickCheck test crucially depends on the quality of the test case generator.

To convince oneself that a test case generator works well, one can use QuickCheck together with a tool for code coverage analysis. I use *hpc* (Haskell program coverage [9]), the code coverage tool which is included in the Glasgow Haskell Compiler.

4.4.4 Mock Implementations

The CSP core language package relies on an external implementation of some functionality. The intended architecture is that this functionality is provided by the implementation of the functional sub-language of CSP$_M$. In the previous section, I have used the term functional sub-language when referring to this external functionality.

However, other ways to implement the external interfaces of the core language package are also possible. In particular for testing, it was useful to define two alternatives which I call `Mock1` and `Mock2`.

`Mock1` only implements the classes `BE` and `BL`, but not `BF`. In other words, `Mock1` does not support multi-field events and can only be used with the naïve proof tree generator. `Mock2` additionally provides `BF`, i.e. the functions for manipulating multi-field event. Therefore `Mock2` can be be used with the naïve proof tree generator and also the constraint-based proof tree generator.

There are three main reasons why I use the mock implementations for testing:

1. Technically the functional sub-language package depends on the core language package but not the other way around. Therefore, the test code inside the core language package cannot rely on the functional sub-language.

2. The mock implementations are small and, hopefully, do not add extra errors. It makes sense to only test one unit at a time.

3. The mock implementations support the generation of arbitrary events, processes, and event sets. This would be more cumbersome when using concrete CSP$_M$ syntax.

It would be reasonable to also randomly generate test cases in CSP$_M$ syntax and to use these for integration tests of the core language package together with the CSP$_M$ functional sub-language. This remains as future work.

4.4.5 QuickCheck Conclusion

My experience with QuickCheck was positive overall. Whenever a property failed to check, there was also a bug or inconsistency in the code that was tested. With QuickCheck, it was possible to check exactly those properties of the implementation that are essential. The QuickCheck properties themselves have a clean and concise implementation in Haskell, and it was not necessary to code any special cases or exceptions in the properties themselves.

QuickCheck is based on randomized testing, which means that it could, in principle, just miss a test case which uncovers a bug. However, this has not yet happened in my application. During the development, I did not encounter any bug in the implementation which is in the scope of the QuickCheck properties that I have tested and that was missed because of an unlucky random generator. The current implementation passes all QuickCheck tests.

It must be noted that testability was one of the main design goals of the implementation. This design goal shows up all over the code. For example, testability is the main reason why I use an explicit representation of the proof trees and a separate proof tree verifier. Putting the focus on testability is one lesson that I learned from earlier, more ad hoc, prototype implementations.

Apart from the fact that the code was generally designed to be testable, the extra effort for using the QuickCheck library was very moderate. I just had to implement the instances for the `Arbitrary` type class and write down the QuickCheck properties.

4.5 Summary

This chapter has described how I model the CSP core language and the firing rules semantics of CSP in Haskell. I have explained how I model processes and events and I have also discussed the advantages of explicit proof trees.

Explicit proof trees help to test and debug the implementation of the firing rules semantics and they help to structure and document the source code. I have described the Haskell implementation of CSP proof trees and the proof tree verifier. An important feature of the proof tree verifier is that it is easy to understand and that it is a direct syntactic translation of the actual firing rules.

The proof tree generator is the function that applies the firing rules in a forward direction, i.e. it computes all possible transitions of a given process. I have described two alternatives for a proof tree generator. The naïve proof tree generator which is concise and straightforward and a constraint-based proof tree generator. The constraint-based proof tree generator is relatively involved and there remains an opportunity for future work. Nevertheless, it works reasonably well.

Finally, I have argued that the presented code is correct. Several measures have been taken to ensure the correctness of the code. One is that the software was designed for testability right from the beginning, for example by using explicit proof trees and a small separate proof tree verifier. Another is that I make heavy use of QuickCheck. I have described how interesting properties, like the soundness and completeness of the proof tree generators, can be checked with QuickCheck.

Chapter 5

Interpreter for the Functional Sub-language of CSP$_M$

5.1 Overview

This section describes the implementation of the functional sub-language of CSP$_M$. The functional sub-language covers everything related to data processing and expression evaluation. Among other things, the following features are implemented by the functional sub-language:

- Declaration and evaluation of functions.
- Declaration of data types and channels.
- Declaration of parametrised processes.
- Built-in functions for lists and sets.
- Computing the value of event fields.
- Pattern matching and input fields.

Everything related to the firing rule semantics of the CSP core language, e.g. process operations like interleaving, parallel composition, process synchronization and computing the possible transitions of a process, is *not* part of the functional sub-language. The implementation of these CSP core language features is described in Chapter 4.

For brevity, I will just use the term *interpreter* in this section when referring to the interpreter for the function sub-language. I use the term *core* for features related to the CSP core semantics as described in Chapter 4.

The interpreter depends on the data type declarations for the CSP$_M$ abstract syntax tree which are defined in the parser package. However, for the displayed pieced of source code only small parts of the AST data type are relevant. Therefore this chapter can be read before the Chapter 6. A complete description of the AST is available in Section 6.2.

This chapter assumes some basic knowledge about functional programming and the interpretation of functional programming languages. A good reference for this subject is the book "The Implementation of Functional Programming Languages" [49].

Chapter Outline

Section 5.2 contains an overview of the design alternatives and the final design of the interpreter. Section 5.3 describes the most important parts of the interpreter source code. Section 5.4 is dedicated to the implementation of equality for CSP$_M$. Finally, Section 5.5 lists some benchmarks that compare my interpreter with other tools and Section 5.6 gives an outlook on a CSP$_M$-to-Haskell compiler.

Section 5.2 and Section 5.4 are perhaps the most important parts of this chapter. They describe a tested and implemented design and can serve as a cookbook for anybody who is interested in writing a CSP$_M$ interpreter. The design is more important than the real source code. Translating the presented design to source code is straightforward.

5.1.1 External Interface of the Interpreter

The external API of the interpreter is simple. The interpreter defines the function :

```
evalModule :: Module INT → Env
```

`evalModule` is called with the abstract syntax tree of a CSP$_M$ specification (`Module INT`) and it returns the top level environment that is defined in the specification. To get the value of an identifier, one just has to look up the identifier in the environment.

5.2 Design Decisions for the Interpreter

I will first discuss some design decisions and requirements for the interpreter before describing the actual code.

5.2.1 FDR Compatibility

The basis of all other considerations is the requirement that the presented CSP$_M$ tool has to be compatible with FDR. FDR compatibility was demanded by an external partner who was involved in the project and it is also important for acceptance of the tool in the CSP community.

The compatibility requirement causes many problems and complications and there are good arguments for giving up on compatibility and designing a new tool from scratch. On the other hand, compatibility makes it possible to compare this project with existing tools and it makes this project a good case study for the use of Haskell.

5.2.2 Model Checking and Equality

An important requirement for the interpreter is that it supports model checking. Model checking requires two features that are not typically present in an interpreter, namely: branching off alternative computations and detecting whether a computation has looped to a state that has been visited before.

There are several frameworks which implement ideas from CSP, for example JCSP [59] and CHP [4]. The problem is that in the case of non-determinism, these frameworks only allow the control flow to follow one possible trace. Also these frameworks do not allow one to detect that a system has looped back to a previous state. Therefore they are not suitable for model checking and my project does not build on one of these frameworks.

I use Haskell as the implementation language. Haskell is a pure functional programming language with immutable data structures. Therefore, branching is easy. To move from one state to the next I use the function :

```
prefixStateNext :: i → PrefixState i → Field i → Maybe (PrefixState i)
```

Whenever this function is called, it computes a new state. The old state is immutable and one can simply call `prefixStateNext` several times with different `Field` arguments to follow alternative branches.

Detecting loops is greatly facilitated by the fact that state is explicit in a pure functional implementation. A process is an expression which can contain free variables, and the state of the process is uniquely determined by the binding for those values. In principle a loop can be detected by simply comparing the current state with all previously visited states.

However, there is one catch. The state of the interpreter is explicit, but I still have to make sure that I can determine when states are equal. This implies two restrictions on the state data type:

1. State must not contain lambda terms.

2. State must not contain infinite or cyclic data structures.

Both restrictions rule out some standard techniques for implementing interpreters in a functional language and both restrictions are relevant for my interpreter.

I will not use a formal definition for the equality of processes in this thesis. For model checking, it is sufficient to detect enough "equality" such that the computation of the transition system terminates. It is always a safe approximation if any two processes compare as not equal. This approximation does not compromise the soundness of model checking. On the other hand, a poor implementation of the quality check for processes can lead to a blow-up in the state space that has to be explored. Failure to detect "enough equality" can also lead to non-termination.

Process equality is not a prominent concept in the theory of CSP. Instead, processes are characterized using refinement properties, full abstraction properties and algebraic laws. Nevertheless, a model checker has to internally implement some kind of process equality. Although the documentation of FDR does not directly address the problem of process equality, the issue is by no means hidden from the user, it only appears in another context.

The FDR solution to the problem is that there are strict side conditions on what specifications are allowed. For example, FDR does only allow a limited form of recursion and all specifications must be built around a finite skeleton of process operations. If a specification is not consistent with these side conditions FDR simply does not terminate. ProB and my new tool lift some of the side conditions on specifications, with the trade-off that equality becomes more difficult.

The related problem of detecting structural sharing is well known in the Haskell community [18]. In retrospect, the requirement to compare states for equality has had impact on almost all design decisions for the interpreter.

Bounded model checking is a variant of model checking which does not require detecting loops in the state space. The idea of bounded model checking is to simply expand the state space for a fixed number of steps k. The crux of bounded model checking is, of course, that the right value for k is often not known in advance. Therefore, bounded model checking is not an option for my project.

5.2.3 Interpreter and Denotational Semantics

The two most important styles of semantics for a programming language are the operational semantics and the denotational semantics. I have decided to base the interpreter for the functional sub-language of CSP$_M$ on a denotational semantics. A denotational semantics defines a function (the denotation) which maps the input to a mathematical model. The denotation should be compositional, which means that the denotation of a compound structure can be computed with the denotations of the structural parts. For more info on denotational and operational semantics see Wikipedia.

The denotation function of the denotational semantics directly corresponds to the `eval` function of my interpreter:

```
eval :: LExp → EM Value
```

`eval` maps expressions of CSP$_M$ to the Haskell representations of CSP$_M$ values. We explain `eval` in more detail in Section 5.3.1.

5.2.4 Environment vs. HOAS

There are two main techniques for implementing function calls (β-reduction) when writing an interpreter (for a functional language) in functional language.

The first one uses an explicit environment that stores the values of all variables that are in scope. Whenever one evaluates an expression, the current environment has to be available. Evaluating a variable means looking up its value in the environment and in order to call a function or to create a closure, a new environment has to be constructed. In the simplest case the environment can be implemented as a list of variable-value pairs.

The second technique is called higher order abstract syntax (HOAS). Functions in the interpreted language are represented as functions in the abstract syntax tree and a function call in the interpreted language is implemented as a function call in the host language. The idea here is to reuse the mechanism of the host language for β-reduction to implement β-reduction in the interpreted language. Since the host language often has highly optimized β-reductions, an interpreter using HOAS can be a lot faster than an interpreter using an explicit environment [3]. A similar design alternative is known in the Prolog community as ground vs non-ground interpreter. ProB uses a non-ground interpreter which, very roughly, corresponds to the HOAS approach.

Although HOAS interpreters are attractive, I have chosen to implement the CSP_M interpreter using explicit environments. The main reason is the requirement from Section 5.2.2, namely that I want to use the interpreter for model checking. Explicit environments make it much easier to examine the state of the interpreter and to detect loops than HOAS. For the same reason, I use environment-expression pairs instead of the built-in Haskell closures for the implementation of CSP_M closures. My intuition is that for model checking, the speed advantages of HOAS for β-reduction are more than outweighed by the more complex loop detection. Another consideration is that explicit environments may be more space efficient. However, I have no empiric data for this.

5.2.5 CSP_M Laziness

The two most important evaluation strategies for functional programming languages are call-by-value and call-by-need. Languages like Haskell favor call-by-need because, among other advantages, call-by-need has the best termination properties. In other words, a program that terminates with a call-by-value strategy is guaranteed to also terminate with a call-by-need strategy, but *not* vice versa. Call-by-need is also called laziness. The opposite of laziness is strictness.

It is guaranteed that all terminating evaluations yield the same result, independently of the evaluation order.[1] However, this is also the crux of laziness. Two functions can compute the same result for almost all inputs, with the exception that for some input, the *more lazy* function terminates while the *more strict* function goes into an infinite loop. For example a *more lazy* function might be able to deal with infinite lists, while a *more strict* function might not. Still, the *more strict* function is by no means incorrect.

The moral of the story is that laziness is a difficult problem. Even in Haskell, the best known lazy functional programming language, laziness is a feature of a particular Haskell implementation and not part of the official language definition [38].[2]

According to Roscoe ([52] page 495) CSP_M is a lazy functional programming language. On the other hand, I'm not aware of any "real-world specifications"[3] that deeply rely on laziness as it is possible in Haskell programs. The use of laziness in CSP_M seems to be restricted to the CSP_M examples that have been explicitly written to demonstrate laziness. As laziness is rarely used in CSP_M, it is unclear how robust the available CSP_M tools are in that respect.

Laziness vs Infinite Lists

Infinite lists are a feature, that can be implemented via laziness. In [52] on page 501, there is the following example for the infinite list of prime numbers (C.3.1):

```
primes =
  let
```

[1]For the exact formulation of the *confluence* of the λ-calculus see: Church-Rosser-Theorem.

[2] Nevertheless, there is a strong consensus in the Haskell community about what kind of laziness one can expect from a Haskell implementation.

[3]For example specifications that have been published in a paper.

```
    factors(n) = < m | m <- <2..n-1>, n%m == 0 >
    is_prime(n) = null(factors(n))
  within < n | n <- <2..>, is_prime(n) >
```

My implementation provides some level of general laziness in the spirit of Haskell. The `primes` example works as expected, but I do not want to give any guarantees for the kind of laziness a specification can rely on. ProB can deal with infinite lists, but only if the list has a simple representation as an open integer interval or if some other hard-coded heuristics apply. In general, infinite lists are just a feature that can be implemented via laziness, but infinite lists are not the same as a lazy evaluation strategy.

One should make a strict distinction between laziness and the fact that some parts of a process expression are only evaluated on demand. For example, in a sequential composition $P ; Q$, Q is only evaluated after P has terminated. Roscoe uses the term that Q is *switched off*. My CSP$_M$ tool implements this feature by explicitly tagging switched-off parts of a process. This makes sure that switched-off processes are, indeed, only evaluated as needed.

5.2.6 Using Haskell Laziness and Knot-Tying

There are two or three places in the interpreter where I use Haskell laziness in a non-trivial way. In particular, I use a technique called knot tying [10] to implement multiple recursive declaration. This is described in detail in Section 5.3.2.

Knot tying can be used to create cyclic data structures. A drawback of cyclic data structures is that functions which handle these data structures must take special precautions to avoid non-termination. This also applies to the environment of my CSP$_M$ interpreter, which is cyclic. The environment contains information about bound functions and the data structure that implements the function itself contains a back-reference to the environment.

An alternative for knot tying and cyclic data structures would have been to use explicit references and updates. However this would have lead to an impure interpreter which has many disadvantages; altogether, the knot tying approach seems to be preferable.

5.2.7 Lambda-lifting

A standard technique for the implementation of a functional programming language is called lambda-lifting ([49] page 220). An alternative design of the interpreter relying on lambda-lifting would have been possible, but this would have also affected other aspects of my design. Taking all considerations into account, a design without lambda-lifting seemed favorable.

Also, a full implementation of lambda-lifting is not trivial. ProB uses a non-ground Prolog representation of functions. This design requires a pre-compilation step, which rewrites the CSP$_M$ specification and which is related to lambda-lifting. Unfortunately, the ProB version of lambda-lifting has the problem that it can rewrite a function with linear complexity to a function with exponential complexity.

Here is an example:

```
f(0) = 1
```

```
f(x) = let a = f (x-1) within a + a
```

`f` has linear complexity, however the pre-compilation step of ProB internally translates `f` to a Prolog representation which roughly corresponds to:

```
f_prolog(0) = 1
f_prolog(x) = a(x) + a(x)
a(x)= f_prolog(x-1)
```

The function `f_prolog` has exponential complexity.

5.2.8 Pure Interpreter

The heart of my interpreter is the `eval` function, which computes the denotational semantics of CSP_M. A very fundamental design decision is whether to make `eval` pure or not. A pure `eval` function can only compute values and a computation cannot have any side effects. Features, that are typically implemented as side effects, are for example: tracing an evaluation, single step execution, profiling, etc. With a pure `eval` function, these features are not possible per se.

An advantage of purity is that a pure interpreter can have a simple interface. `eval` is just a function which returns a value and there are absolutely no preconditions on how or when this function can be called. As Section 5.2.2 explains, this has advantages when the interpreter is used for model checking applications. Another argument in favor of purity is that the CSP_M language is itself pure (with the exception of the print statement).

The final decision was to write the `eval` as a pure function. After all, this does not even rule out features like tracing or profiling. If really needed, these features could still be added with `unsafePerformIO`. `unsafePerformIO` is a *back door* in Haskell which allows one to turn any pure function into a procedure which performs side effects. `unsafePerformIO` is currently *not* used in the CSP_M interpreter and it generally should be avoided because improper use of `unsafePerformIO` can cause bugs which are hard to track. On the other hand, `unsafePerformIO` is acceptable for benign side effects like tracing, profiling or debugging.[4]

5.2.9 Representing CSP_M Values

The FDR tool implements a dynamically typed language (c.f. [34]). Defining a static type system for CSP_M, that is consistent with FDR, is difficult. As a consequence, I also use dynamic typing in my interpreter, or, more precisely, I implement a dynamically typed language. Although this has several disadvantages, for example performance penalties, it is a pragmatic and tested approach. The ProB tool uses similar techniques (presumably FDR also).

For the implementation, this means that I define one big sum-type for all CSP_M values:

```
data Value =
   VInt  Integer
 | VBool Bool
 | VList [Value]
```

[4] Technically `eval` is written in monadic style. It may also be possible to support tracing, etc by using for example the IO monad.

```
  | VTuple [Value]
  | VDotTuple [Value]
  | VSet (Set Value)
  | VClosure ClosureSet
  | VFun FunClosure
  | VProcess Process
  | VChannel Channel
  | VUnit
  | VAllInts
  | VAllSequents (Set Value)
  | VConstructor Constructor
  | VDataType [Constructor]
  | VNameType [FieldSet]
  | VPartialApplied FunClosure [Value]
 deriving (Ord,Eq)
```

This is also called a *boxed* or *tagged* representation of values.

There are some possible alternatives for the exact structure of the `Value` data type, for example special representations could be added for infinite lists. In the end, the exact structure of the `Value` data type is not critical because it is easy to add conversions between the alternative representations of a value.

For example, sets and closure sets have two separate representations in the type `Value`, namely `VSet` and `VClosure`. On the other hand CSP$_M$ does not make a strict distinction between sets and closure sets. Therefore, all functions that work on generic CSP$_M$ sets evaluate their arguments by calling `setFromValueM`. `setFromValueM` converts different representations of sets to an unboxed Haskell set on the fly.

```
setFromValueM :: Value → Maybe (Set Value)
setFromValueM v = case v of
  VSet l → Just l
  VClosure c → Just $ closureToSet c
  VDataType l → Just $ Set.fromList $ map VConstructor l
  _ → Nothing
```

This design is flexible. When I came across one specification which also uses set operations on CSP$_M$ data types, it was easy to extend `setFromValueM` with a conversion from `VDataType` (i.e. the representation of CSP$_M$ data types) to sets. It is also possible to overload CSP$_M$ operation, i.e. to use several alternative Haskell functions that work on the alternative representations of the values.

It is interesting to speculate about the internal implementation of FDR. A possible object-oriented design, would be to define a super class `Value` and a subclass of `Value` for every possible representation. The analog to `setFromValueM` would then be to overwrite a method `toSet` in all subclasses of `Value` that can be converted to a set.

5.3 Implementation

This section describes the most important parts of the source code of my interpreter. The complete code is available via Hackage [14] in the package `CSPM-Interpreter`. The relevant version is `0.5.1.0`.

5.3.1 The `eval` Function

The `eval` function has type:

```
eval :: LExp → EM Value
```

It takes an expression of type `LExp`[5] and computes the value of the expression inside monad `EM`. `EM` is a simple reader monad which carries around the environment, and the environment is implemented as `IntMap`s which hold the values of bound variables.
The definitions are as follows:

```
newtype EM x = EM { unEM ::Reader Env x }
  deriving (Monad,MonadReader Env)

type Bindings = IntMap Value
data Env = Env {
   argBindings :: Bindings
  ,letBindings :: Bindings
  ,letDigests :: IntMap Digest
  }
```

The eval function uses a big case switch with exactly one case for each constructor in the data type `Exp`[6]. The complete listing is shown in Appendix B.1.9. Here is a stripped down version with some exemplary cases:

```
eval :: LExp → EM Value
eval expr = case unLabel expr of
  Var v → lookupIdent v
  IntExp i → return $ VInt i
  Ifte cond t e → do
    c ← evalBool cond
    if c then eval t else eval e
  Stop → return  $ VProcess $ Core.stop
  AndExp a b → do
    av ← evalBool a
    if av then eval b else return $ VBool False
  ProcSharing s a b
   → liftM3 Core.sharing
       (switchedOffProc a)
       (evalClosureExp s)
       (switchedOffProc b)
      >>= return ∘ VProcess
```

The displayed cases are:

(`Var v`) A variable expression causes a look-up of the identifier in the environment.

(`IntExp i`) An integer constant.

(`Ifte cond t e`) The CSP$_M$ if-then-else construct.

(`Stop`) The *STOP* process calls a definition from the core language package.

[5] `type LExp = Labeled Exp`. Labeled expressions are expressions which contain information about source locations (c.f Chapter 6).

[6] The `Exp` data type is discussed in detail in Section 6.2.4.

(`AndExp a b`) The Boolean and-function with shortcut semantics.

(`ProcSharing s a b`) CSP sharing $A \underset{s}{\|} B$ evaluates its arguments and calls `sharing` from the core language.

The `eval` function uses several helper functions which often recursively call back `eval`. For example the following function is used to dynamically check at run-time that an expression is of type `Bool` and return its value.

```
evalBool :: LExp → EM Bool
evalBool e = do
  v ← eval e
  case v of
    VBool b → return b
    _   → throwTypingError "expecting type Bool"
           (Just $ srcLoc e) $ Just v
```

The function throws an exception in case of a type error.

CSP$_M$ has a rich expression language and accordingly `eval` is the largest function of the interpreter. Nevertheless, the Haskell implementation of `eval` is compact and has a clean structure. The translation from the denotational semantics of CSP$_M$ to the Haskell `eval` function is straightforward.

5.3.2 Declarations

In CSP$_M$, local names can be declared inside let expressions.
For example:

```
f(0) = 1
f(x) = let
    b = a + a
    a = f(x-1)
    c = b + b
  within c + c
```

The main idea of local names is some kind of reuse. Instead of "`f(x-1)`" one can just write "`a`", but of course one definitely also expects that `a` is not evaluated more than once. In other words, `let`s are used to introduce explicit sharing of intermediate results.

The general syntax for `let` is

```
let
  declaration_1
  declaration_2
  ...
  declaration_n
within expression
```

where each declaration can be either a pattern match [7] or a function declaration.

All bindings that are introduced in the left-hand-side of a declaration are visible in all right-hand-sides (and of course in the `expression`) and bindings can be mutually recursive. Here is a small example of a mutually recursive let (the complete source code is in Appendix C.3.2):

[7] The simplest form of a pattern match is just a variable as in `c = b + b`

Listing 5.1: CSP$_M$ mutually recursive `let`

```
list = let
    o = <1,2> ^ z
    z = <0,2> ^ o
  within <3> ^ o
```

This declaration defines a list which starts with a 3 and infinitely repeats the sequence `<1,2,0,2>`

In spite of `let` being a relatively simple syntactic from, it is not trivial to implement this, for example, in an imperative language. Fortunately, there is a well known technique that simplifies the implementation of `let`. I deeply rely on the fact that I implement the interpreter in a functional language, strictly speaking, I just reuse Haskell's `let` to implement the CSP$_M$-`let`.

There are three main requirements for the implementation of `let` in the CSP$_M$ interpreter:

- It must respect sharing, i.e. it must not evaluate expressions more than once.
- It must be declarative, i.e. it must not update references or manipulate state.
- It should work well with cyclic declarations like infinite lists.

All the requirements can be achieved by using *knot tying* [10]. Knot tying is a design pattern for combining mutual recursion and lazy evaluation. Knot tying can provide elegant declarative solutions for problems with complicated non-trivial data flow. It can, for example, be used to implement memorisation. Listing 5.1 is an example for *knot tying* in CSP$_M$.

In my interpreter, I use knot tying as follows:

```
 1  type DeclM x = ReaderT (Digest,Env) (State (Bindings, IntMap Digest)) x
 2
 3  processDeclList :: Digest → Env → [LDecl] → Env
 4  processDeclList digest oldEnv decls =
 5    let
 6      (newBinds,newDigests)
 7         = execState action' (getLetBindings oldEnv, letDigests oldEnv)
 8      action :: DeclM ()
 9      action  = mapM_ processDecl decls
10      action' = runReaderT action (digest,newEnv)
11      newEnv  = oldEnv { letBindings = newBinds, letDigests = newDigests}
12    in newEnv
13
14  processDecl :: LDecl → DeclM ()
```

The function `processDeclList` takes a digest [8], the old environment and a list of declarations as input and computes the new environment. The new environment is the extension of the old environment with the declarations. `processDeclList` works by calling `processDecl` for each declaration in turn (`mapM processDecl decls` line 9).

The knot or cycle in the data flow is the following:

[8]Digests are explained in Section 5.4

Lines 6,7 `newBinds` and `newDigests` depend on `action'`.

Line 10 `action'` depends on `newEnv`.

Line 11 `newEnv` depends on `newBinds` and `newDigests`.

This works because the `newEnv` data structure is constructed lazily. The environment maps identifiers to values but the values are computed only as much as needed. To add a new name-value pair to the environment it not necessary to scrutinise the value.

`processDecl` is declared in terms of Monad `DeclM`. `DeclM` (line 1) is a reader transformer of a state monad. In other words, `processDecl` can read a digest and an environment and it can modify a state which consists of `(Bindings, IntMap Digest)`. `processDecl` produces no result apart from the side effects, i.e. the updates to the state. `processDecl` adds the new bindings to the environment, but it does not compute the values.

The view from inside Haskell is that the environment is an immutable data structure which maps names to values. On the machine code level, of course something different happens. A value is represented by a reference to a delayed computation. When the interpreter scrutinises a value—for example, when it checks if a Boolean is `True` or `False`—the delayed computation is executed and the value is stored. After that, the reference points to the value instead of the delayed computation and the value is computed once at most.

However, these updates are unobservable from inside the Haskell program. The implementation of laziness in Haskell is hidden from the programmer. For the implementation of the CSP$_M$ interpreter, I can rely on the abstraction that the environment behaves like an immutable data structure. If the same CSP$_M$ function is called with the same environment, it will compute the same result. This is a consequence of referential transparency.

The advantage of the presented approach is that the actual knot is tied in a single Haskell `let`-expression in line 5 to line 12. These 7 lines are where "the magic" happens. The rest of the code is mostly unaffected by the knot. Other parts of my CSP$_M$ tool can rely on the strong abstraction that the interpreter behaves like a pure function.

The same approach that has been described for `let` is also used for top-level declarations. Top-level declarations of a CSP$_M$ specification are just a special case of a `let`.

Drawbacks

On one hand, it is nice that knot tying hides the treatment of lets and CSP$_M$-laziness from the rest of the implementation. On the other hand, this also means that parts of the execution of the interpreter become implicit. In particular, the interpreter does not control when exactly an expression gets evaluated. This also means that the interpreter cannot detect if a cyclic let-expression in CSP$_M$ does not terminate. It is impossible to detect non-termination in general, but nevertheless it may still be possible to dynamically detect some cases of non-termination.

As I map a CSP$_M$ `let`-expression to a Haskell `let`-expression, I completely rely on the properties of the Haskell `let`. The CSP$_M$-expression

```
let
  a=a+1
within a
```

yields a `loop`-exception with my interpreter.[9] This is a correct behavior, but the loop is not detected by my interpreter but by the Haskell run-time system. Haskell does not guarantee that any loop can be detected. It is possible that the CSP$_M$ interpreter does not detect the loop when compiled for another architecture or with other compiler flags.

It may be possible to detect some problematic cases with a careful static analysis of the CSP$_M$ specification, but this does not solve the problem in general.

5.3.3 The Pattern Matcher

There are three different parts of the CSP$_M$ language that use pattern matching:

1. Declarations.

2. Function definitions with multiple cases.

3. Input fields in prefix operations.

The syntax for patterns loosely resembles the syntax of expressions. There are pattern for constants, tuples, dot-tuples, lists and sets and, just as expressions, pattern can be arbitrarily nested.

Append Patterns

A special feature of CSP$_M$ is the append pattern (^). The ^-operator, which appends two lists when used in an expression, deconstructs lists inside a pattern. The append pattern allows matching lists from both ends. For example the declarations:

```
<h1> ^ rest = <1,2,3,4>
init ^ <l1> = <1,2,3,4>
<h2> ^ body ^ <l2> = <1,2,3,4>
```

contain the following matches:

pattern	match
`h1`	`1`
`rest`	`<2,3,4>`
`init`	`<1,2,3>`
`l1`	`4`
`h2`	`1`
`body`	`<2,3>`
`l2`	`4`

The general syntax for an append pattern is $pat_1\frown pat_2\frown\ldots\frown pat_n$, where the sub-patterns pat_i are again patterns that match a list. A sub-pattern pat_i can either match a fixed length list, for example `<x>` or a variable length list like

[9]On my Linux machine.

`init`, `rest` and `body`. A valid append pattern may contain at most one sub-pattern which can match a variable-length list. For example the pattern `prefix ^ suffix` is not a valid pattern.

I know of no mainstream functional programming language with built-in support for append patterns. Usually, functional languages only allow one to match lists from the head, which can be implemented in constant time. It is not easy to provide an efficient implementation for append patterns. Given the rare use of append patterns and the effort of the implementation, it might have been better to leave them out of the CSP$_M$ language. Still, they had to be implemented in my CSP$_M$ interpreter to remain compatible.

Selectors

To simplify the pattern matcher, particular in the presence of append patterns, the interpreter does not work directly with the AST for patterns as it is returned by the parser. Instead, patterns are statically analysed and rewritten to a flat array of linear selectors.

Selectors are represented with the following data type:

```
data Selector
  = IntSel Integer
  | TrueSel
  | FalseSel
  | SelectThis
  | ConstrSel UniqueIdent
  | DotSel Int Selector
  | SingleSetSel Selector
  | EmptySetSel
  | TupleLengthSel Int Selector
  | TupleIthSel Int Selector
  | ListLengthSel Int Selector
  | ListIthSel Int Selector
  | HeadSel Selector
  | HeadNSel Int Selector
  | PrefixSel Int Int Selector
  | TailSel Selector
  | SliceSel Int Int Selector
  | SuffixSel Int Int Selector
```

A selector describes how to check that a part of a pattern matches a value and how to extract the part of the value that is bound to an identifier. They also describe parts of patterns that match constants. The selectors for lists are the following:

`SelectThis`	Select the current part of the list.
`HeadSel`	Select the head of the list.
`HeadNSel n`	Select the first n elements.
`PrefixSel o l`	Select the first l elements of the list after dropping o elements.
`TailSel`	Selects the tail of the list.
`SliceSel l r`	Select a variable length part of a list by dropping l elements from the left end and r elements from the right end.
`SuffixSel offset len`	The dual of `PrefixSel`. Counting from the right end of the list.
`ListLengthSel`	Check the length of a list.
`ListIthSel i`	Select the ith element of the list.

Example

<a>^<1>^`rest`

is translated to the following three selectors:

```
HeadSel SelectThis
PrefixSel 1 1 (ListLengthSel 1 (ListIthSel 0 (IntSel 1)))
SliceSel 2 0 SelectThis
```

The first selector computes the part of the value bound to `a`, the second verifies the 1 on the second position of the list and the third computes `rest`. The `SeclectThis` selector simply matches the complete value.

Code Example

The function `match :: Value -> Selector -> Maybe Value` is an interpreter for the selector data type. It takes a value and a selector as input and if the selector matches, it returns the part of the value that is matched. Here are some excerpts from `match`:

```
1  match :: Value → Selector → Maybe Value
2  match (VInt a) (IntSel b) = if a==b then return VUnit else failedMatch
3  match v        (IntSel _) = typeError "expecting Int" v
4
5  match (VBool True)  TrueSel = return VUnit
6  match (VBool False) TrueSel = failedMatch
7  match v             TrueSel = typeError "expecting Bool" v
8
9  match (VBool True)  FalseSel = failedMatch
10 match (VBool False) FalseSel = return VUnit
11 match v             FalseSel = typeError "expecting Bool" v
12
13 match x             SelectThis = return x
14
15 match (VTuple b)    (TupleIthSel i next) = match (b !! i) next
16 match v             (TupleIthSel _ n) = typeError "expecting tuple" v
```

Lines 2,3 Match an integer constant

Lines 5-11 Match Boolean constant

Line 13 Match a anything, bind a variable

Line 15 Match the *i*th element of a tuple and recursively call `match` on the tuple element.

The pattern matcher can easily be extended for dynamic typing features of CSP$_M$. For example, the line:

```
match (VInt 0)  FalseSel = return VUnit
```

would allow the integer 0 to match a Boolean false.

Interface of the Pattern Matcher

The following functions are the external interface of the pattern matcher:

```
tryMatchStrict :: Bindings → LPattern → Value → Maybe Bindings
tryMatchLazy :: Bindings → LPattern → Value → Bindings
```

`tryMatchStrict` is used for matches that may fail, i.e. for a function case or a prefix input field. If a function case does not match, the interpreter tries the next case and if a prefix input field does not match, the process simply does not synchronize. Both cases can occur during normal execution of a specification and the interpreter must be able to detect this from inside the `eval` function. `tryMatchStrict` therefore returns `Maybe Bindings`.

`tryMatchLazy` is used for the declarations inside a `let`. A match failure in a declaration is a error in the CSP$_M$ script and the interpreter throws an exception in that case.

Restrictions of the Pattern Matcher

I want to add some comments about the presented matcher. First, the matcher is not optimal for nested pattern, because the path to sub-patterns has to be traversed several times. Also, the data type for selectors contains some redundancies, for example `HeadSel` is a special case of `HeadNSel` and the compiled selectors could also be optimised.

In retrospect, I think it might be better to implement a pattern matcher which works without pre-compiling the patterns to selectors; however I have not tried to implement this.

As the pattern matcher is an important building block of the interpreter, I just wanted to sketch one possible implementation for a pattern matcher in this thesis. The most important feature of the interpreter (concerning pattern matching) is that the interface to the pattern matcher consists only of two functions:

```
tryMatchStrict :: Bindings → LPattern → Value → Maybe Bindings
tryMatchLazy :: Bindings → LPattern → Value → Bindings
```

This means that it is easy to replace the current pattern matcher with a different implementation.

5.3.4 AST Preprocessing

The abstract syntax tree, which is returned by the parser, has to pass several preprocessing steps before it can be passed to the interpreter. The interpreter only works if the following three preprocessing steps have been applied:

renaming	see Section 5.3.5
free names analysis	see Section 5.4
compiling patterns to selectors	see Section 5.3.3

The preprocessings have to be applied in the above order.

5.3.5 Renaming

One preprocessing step for the AST is renaming. During renaming, each identifier is tagged with a unique ID. In addition, renaming determines for every using occurrence of an identifier the place where it has been bound. In the AST, identifiers are represented with the data types:

```
data Ident
  = Ident  {unIdent :: String}
  | UIdent {unUIdent :: UniqueIdent}
  deriving (Show,Eq,Ord,Typeable, Data)

data UniqueIdent = UniqueIdent
  {
   uniqueIdentId :: Int
  ,bindingSide :: NodeId
  ,bindingLoc  :: SrcLoc
  ,idType      :: IDType
  ,realName    :: String
  ,newName     :: String
  ,prologMode  :: PrologMode
  ,bindType    :: BindType
  } deriving (Show,Eq,Ord,Typeable, Data)
```

Before renaming, all identifiers are of variant `Ident` and after renaming all identifier are of variant `UIdent`. `uniqueIdentId` is a unique 32 Bit integer ID for each variable. I use the unique ID as index for the variables in the environment.

5.3.6 Instances for the Core Language

The interpreter defines data types for processes, events, event sets, etc. To connect the interpreter to the core language module it necessary to make some of the interpreter data types an instance of the type classes and type families that are defined by the core language. These instance declarations enable the core language modules to use the interpreter types when computing the firing rules semantics of a CSP$_M$ process.

The interpreter uses the empty data type `INT` as type index and phantom type for all instances that belong to the interpreter.

```
data INT
```

In module `CSPM.Interpreter.Types` the following type family instances are defined:

```
type instance Core.Event INT = Event
type instance Core.EventSet INT = ClosureSet
type instance Core.RenamingRelation INT = RenamingRelation
type instance Core.ClosureState INT = ClosureState
type instance Core.Field INT = Field
type instance Core.FieldSet INT = FieldSet
type instance Core.ExtProcess INT = SwitchedOffProc
type instance Core.Prefix INT = PrefixState
type instance Core.PrefixState INT = GenericBufferPrefix
```

The type instance declaration

```
type instance Core.Event INT = Event
```

has following semantics: Type `Event` from module `Core` with index `INT` is mapped to the type `Event` in the interpreter. The identifier `Event` is used in the core language package and the interpreter package. Technically, both uses of `Event` define completely independent types. The same identifier is used in the two packages because those types model the same concept. The type instance declaration glues both types together. Finally, when the compiler links the core language package and the interpreter package to an executable binary program, there is only one memory layout of the data structure `Event`.

The types of the core modules are indexed. This makes it possible to use the core modules with several alternative implementations of the underlying data types (see also Section 3.3.2). For example, the interpreter uses the type index `INT` and the compiler uses the type index `IVO`.

The module `CSPM.Interpreter.CoreInstances` also contains instance declarations for the classes `BL`, `BE` and `BF`. The instance declarations basically collect the functions defined in the interpreter package together with some glue code. Please consult the source distribution for the actual code.

The implementation of the CSP core language works with an abstract process type. A process computed by the interpreter can be passed directly to the core language module. There are no direct callbacks from core the language module to the interpreter and the implementation of the core semantics does not depend on the interpreter, i.e. it does not include or import any interpreter modules.

Callbacks are only done via the class mechanism. The core modules define an interface which consists of several classes and type families. The interpreter depends on the interface definitions and implements the required instances for the classes and type families.

5.3.7 Built-in Data Types of CSP$_M$

As the reference implementation (FDR) is basically untyped, there are several possibilities of what the basic data types of CSP$_M$ are. For example, closure sets (expressions built with `{|..|}`) could either have a special internal representation or they could simply be implemented as regular sets.

The `Value` data type (Section 5.2.9) lists the types that are fundamental in my interpreter. The most important ones are:

- Sets
- Lists

- Closure sets
- Dot tuples

I have implemented the primitive operations for these types which forms some kind of CSP$_M$ run-time system. This run-time system is part of the interpreter package, but, in fact, it is useful independent of the interpreter. For example, it has been used for a CSP$_M$-compiler, which has been implemented as part of Ivaylo Dobrikov's masters thesis [12].

The run-time system for sets, lists, and dot tuples is very simple. CSP$_M$ sets directly use the `Set` type of the Haskell `containers` package. The only interesting data structure is the one used for closure sets.

Closure Sets

CSP$_M$ uses the syntax `{|` $x_1, .., x_n$ `|}` for the so-called closure operation (see [52] page 507). The closure operation is a convenient way for building event sets without explicitly enumerating a large number events. Closure sets are best explained with an example. Let's consider the following definitions:

```
channel ca: {1,2,3}.{1,2,3}
channel cb: {1,2,3}.{1,2,3}
channel a : {8,9}
s = {|a, ca.1, ca.2.2, cb.2 |}
```

Then `s` is the following set:

```
{a.8, a.9, ca.1.1, ca.1.2, ca.1.3, ca.2.2, cb.2.1, cb.2.2, cb.2.3}
```

Informally, if `c1` and `c2` are two channels then `{|c1, c2|}` is the set of all possible communications on the channels and the closure set of `{|c.x|}` is the set of all possible communications on channel `c` that start with prefix `c.x`.

Closure sets play an important role in the interface between the interpreter and the core semantics. The sets that appear as part of the core language interface are all closure sets and, in particular, the constraint based implementation of the core semantics (Section 4.3) makes heavy use of closure sets.

The core language uses the following (simplified) interface to event closure sets.

```
closureStateNext :: ClosureSet → Field → ClosureSet
viewClosureFields :: ClosureSet → Set Field
```

`closureStateNext` computes the projection of the closure set on the first field:
$closureStateNext(c, f) = \{e | f.e \in c\}$
`viewClosureFields` returns all possible first fields in a closure set:
$viewClosureFields(c) = \{f | \exists r : f.r \in c\}$
These two functions suggest a trie like representation of closure sets.

The current implementation uses a trie data structure; however, it is not complete in the sense that some optimisations have not yet been implemented. For example, channels are typically defined for a range of values, like `channel c:{1..1000}.{1..1000}`. This channel has 1 million possible events. I use the short cut that I first enumerate all events of the channel and then convert this explicit set to the compact symbolic representation. Of course this is bad for performance and should be avoided.

5.4 Equality and Hashing

This section describes how I implement equality for the CSP$_M$ data types. An implementation of equality is needed to detect loops in the transition system during model checking. It is also necessary to address the problem of equality, because the equality operator `==` (and orderings `<` , `>`) are part of the functional sub-language of CSP$_M$ itself.

Equality also shows up implicitly, when CSP$_M$ values are inserted into sets. Since a set cannot contain duplicate values, one needs to be able to compare the elements of a set for equality (and since my implementation uses binary trees to represent sets I also need an ordering relation for set elements).

The definition of a CSP$_M$ process can make full use of the functional sub-language of CSP$_M$. In other words, a process can be defined as `P(x)` where `x` can be any CSP$_M$ value. For model checking, it is important to detect if `P(x)` is equal to `P(y)` and therefore it is also necessary to detect if `x == y`. As `x` and `y` can be any CSP$_M$ value it is in principle necessary to implement an equality relation that works for all CSP$_M$ values.

Haskell has a clean solution for equality. The Haskell equality operator (`==`) is a function of the type class `Eq`. The type system enforces that `==` can only be applied if the type is an instance of type class `Eq`. It is good practice to define only those instances which have a "reasonable" implementation. Unfortunately, CSP$_M$ is not as clean as Haskell. CSP$_M$ is a dynamically typed language and equality can be used with any types. Furthermore equality is an ad hoc built-in of CSP$_M$.

My implementation of equality for CSP$_M$ is a compromise between accuracy and computability. The implementation guarantees, that if two values compare as equal, they are equal in the sense of Leibniz equality. The opposite does not hold (because the opposite is not computable anyway).

The main idea is that I implement equality via (cryptographic) hash values. One can summarise this idea as:

```
1 class Hash a where
2   hash :: a → MD5Digest
3
4 instance Hash MyType ⇒ Eq MyType where
5   a == b = hash a == hash b
6 instance Hash MyType ⇒ Ord MyType where
7   compare a b = compare (hash a) (hash b)
```

Lines 1,2 One defines the class `Hash` for types that have a hash-value.

Lines 4,5 If a type has a hash value, it is also in type class `Eq`. One simply compares the hash-values.

Lines 6,7 The ordering of values is determined by the ordering of their hashes.

All that remains to be done is the definition of the hash functions for the data types of the functional sub-language.

I distinguish two kinds of values. Simple values are constants, sets, lists, tuples, etc. The hash of a simple value can be computed by structural recursion. For example, to hash a list I simply compute the hash values of all elements of the list and then mix all hash values together. Additionally, I have to mix in tags for the actual type of the value to distinguish, for example, between `<1,2,3>` and `{1,2,3}`.

Complex values are function closures, processes and the values of `let`-bound names. The hash value of a complex value is computed with the unique ID of the AST that is used to define the value plus a hash of all values of the free names that occur in the expression.

For example consider the following function definitions:

```
f(x) = c*x
g(x) = c*x
```

The hash value of the function closure `f` is computed with the hash value of `c` and a hash value of the AST node that represents `f(x)=c*x`. This is also an example of an approximation. Since `f` and `g` are defined with two different ASTs (at two different source locations) `f` and `g` have two different hash values, although they are semantically equivalent.

There are subtle interactions between `let` expressions and hash computation. I treat values that get bound in a `let` expression similar to function closures. As free variables of a `let`-bound value, I use all free variables of all right hand sides of the `let` block. This is a simple approximation, which also works for complex, mutually recursive `let` declarations. For example:

```
let
  a = c1 + b
  b = c2
within
```

The hash value of `a` is computed with the AST node of `a = c1 + b` and the hash value of `c1` and the hash value of `c2`.

This is not a very accurate approximation. For example, a possible optimisation is to split up recursive `lets` into the smallest strongly connected compounds. Although there are possibilities for further improvements, the current implementation performs well in "real-world" benchmarks.

To speed up hashing, some types memorize their hash value. For example:

```
data FunClosure = FunClosure {
   getFunCases :: [AST.FunCase]
  ,getFunEnv :: Env
  ,getFunArgNum :: Int
  ,getFunId  :: Digest
  }

instance Eq FunClosure where
  a == b = getFunId a == getFunId b
instance Ord FunClosure where
  compare a b = compare (getFunId a) (getFunId b)
```

The hash value of a function closure is computed only once and then stored in the `getFunId` field.

Sometimes, the set of free variables of an expression is needed to compute the hash values. The set of free variables of those expressions is computed in a preprocessing step and added as an annotation to the AST.

Correctness of Hash-based Equality

Strictly speaking, implementing equality via hash values is *not correct.* If the interpreter happens to compare two values for which there is a hash collision, it will compute a *false result.* To justify my hash-based implementation, one should compare two probabilities:

1. The chance for a hash collision.
2. The chance that the interpreter does not compute the right result for some other reason.

The chance of hash collisions can be roughly estimated according to the birthday problem [60]. It is a function of the number of bits used for the hash value and the number of hash values that are computed. For example, let's assume we use a truly random 128-bit hash function and compute around 10^{11} hash values. Then the probability of at least one hash collision is between 10^{-15} and 10^{-18}.

During the time that we compute our 10^{11} hash values something else can happen. For example:

- Estimates for a bit flip in RAM vary between one bit per hour per gigabyte and one bit per century.
- Vendors of SATA hard drives claim a mean time between failures (MTBF) of about 600,000 hours (approx. 70 years).
- We hit a bug in the implementation.
- Earth gets hit by a large meteor.

The current implementation uses a 128-bit MD5 hash function. MD5 was used because a pure Haskell implementation of MD5 was readily available. I estimate that the computation of hash values costs less then 15% of the running time depending on the specification. This estimate is based on the slowdown I experienced when I replaced a fast 64-bit hash with 128-bit MD5.

The Haskell implementation of MD5 was chosen for convenience. I think that replacing the Haskell MD5 implementation with a highly optimized implementation of SHA512 would result in approximately the same overall performance of the interpreter. In other words, it is straightforward to improve the confidence of hash-based equality by using a better hash function, at very moderate costs.

Conclusion

Concerning the use of hash functions, I come to to following conclusions:

- Using hash functions is not 100% correct.
- Using MD5 works well in practice.
- For high assurance a 512-bit cryptographic hash function should be used.

5.5 Pure Functional Performance

This section contains some benchmarks for the interpreter of the functional sub-language of CSP$_M$. These benchmarks are only meant to give a rough estimate of the performance of the interpreter.

The idea of CSP$_M$ is to specify a system in terms of processes and process operations while the functional sub-language should only be auxiliary. Section 8.5.4 contains several benchmarks using "real-world" specifications from the literature, which cover both the functional sub-language and the CSP core semantic. Nevertheless, it is interesting to benchmark the pure functional performance of CSP$_M$ because it can easily be compared with, for example, the functional programming language Haskell.

`cspm`	CSPM-Interpreter-0.4.2.0 & mtl-1.1.1.1 compiled with ghc-7.0.1
FDR	2.82
ProB	probcli 1.3.2-beta10 (5054)
Python	2.6.5
Comp	CSPM-Compiler-0.0.1.1 plus ghc-6.12.3
GHC	Haskell ghc-6.12.3

Figure 5.1: Tool Versions Used in the Benchmarks

Benchmarks

Table 5.2 shows the measured running times. The benchmarks were run on a 2.66 GHz Intel Core2 Duo CPU using a Linux operating system. The relevant versions of the programs and tools are listed in Table 5.1. The listed running times of my interpreter, ProB, Python and Haskell (GHC) are the times that are reported by the program itself. These running times do not include start-up times, parsing and preprocessing. The FDR times are measured with the UNIX 'time' command, i.e they include a small overhead for starting the FDR interpreter and parsing the specification. The times reported for the compiler are the running times of the compiled binary, excluding the times for the CSP$_M$-to-Haskell and the Haskell-to-binary compilation. The compiler is described in Section 5.6.

Some of the benchmarks do not have a one-to-one translation to Python, for example Python does not support pattern matching. The Python times shown for `fib1`/`fib2` and `smc`/`smc2` actually refer to identical Python functions. The run-times for `smc` and `smc2` for Python and Haskell are the times for running 1000 instances of the benchmark divided by 1000. I have used the default settings for garbage collection for all programs.

Discussion

Among the selected tools, only the Haskell-based implementations are able to compute the value of `ack(5,0)`. Python either crashes with "`RuntimeError: maximum recursion depth exceeded`" or after manually increasing the recursion limit with "`sys.setrecursionlimit(70000)`" it crashes with a segmentation fault. This crash has been confirmed with four different operation systems/ architec-

Absolute running times in seconds

	`cspm`	FDR	ProB	Python	Comp	GHC
fib1	22.1	43.3	475	8	2.5	0.56
fib2	20.5	41.2	234	8	2.7	0.63
ack	37000	error	no res	error	352	70
smc	0.06	0.08	50.8	0.00489	0.0036	0.0012
smc2	0.06	0.13	95.6	0.00489	0.0042	0.0013
ithPrime	11.0	73.9	error	–	2.5	1.2

Relative speed-ups and slow-downs normalizes to the interpreter

	`cspm`	FDR	ProB	Python	Comp	GHC
		slower	slower	faster	faster	faster
fib1	1	1.9	21	2.8	8.8	39
fib2	1	2.0	11	2.6	7.6	33
ack	1	error	no res.	error	105	529
smc	1	1.3	847	12.3	16.7	50
smc2	1	2.2	1593	12.3	14.3	46
ithPrime	1	6.6	error	–	4.4	9

Relative slow-downs normalizes to GHC

	`cspm`	FDR	ProB	Python	Comp	GHC
fib1	39	77	848	14	4.5	1
fib2	33	65.4	371	12.7	4.3	1
ack	539	error	no res.	error	5.0	1
smc	50	66.7	42333	4	3	1
smc2	46	100	73538	3.8	3.2	1
ithPrime	9	61.6	error	–	2.1	1

Benchmarks

	Argument	Description
fib1	35	Fibonacci function using if-then-else
fib2	35	Fibonacci function using pattern matching
ack	(5,0)	Ackermann function
smc	10000	`sum (map square [0..n])` using pattern matching
smc2	10000	`sum (map square [0..n])` using if-then-else
ithPrime	3000	compute the ith prime number

Figure 5.2: Benchmarks for the functional performance of the implementations.

tures. FDR also crashed with a segmentation fault and Prob produces a `"Resource error: insufficient memory"` error.

FDR is the only tool which uses 32-bit integers instead of exact integer arithmetic. This means that in the `smc` benchmark an overflow occurs and FDR computes a *false result.*[10]

The benchmarks are far from being exhaustive. This is problematic, since the running times show a great variation and a small change in the benchmark can have a big impact on the running time. For example `fib1` and `fib2` are basically the same function. The only difference is that `fib1` uses `if-then-else`, while `fib2` is defined via pattern matching. Yet `fib1` is two times slower than `fib2` in ProB. The same effect shows up with `smc` and `smc2`. In Haskell itself, pattern matching and `if-then-else` have roughly the same performance.

The `smc2` benchmarks seems to contain some ProB performance killers. For this benchmark ProB is 1600 times slower than my interpreter and 74,000 times slower than the native Haskell implementation. Similarly, the `ack` benchmark contains a performance killer for `cspm`. Here, my interpreter is about ten times slower compared to the other benchmarks. It is likely that there are more CSP$_M$ language features which are a performance killer for one tool or the other, but which have not been tested.

Normalizing the running times to the native Haskell implementation shows that `cspm` is about 30 to 50 times slower and the FDR tool is about 60 to 100 times slower than Haskell. The CSP$_M$-to-Haskell compiler adds an overhead between a factor of 2 and 5. Haskell shows a relatively poor performance on the `ithPrime` benchmark.

Conclusion

The benchmarks support the following rough estimates for the performance of my interpreter for pure functional benchmarks.

- The interpretation overhead compared to a direct use of Haskell is approximately a factor of 40.
- The interpreter is roughly 2 times faster than FDR.
- The interpreter is roughly an order of magnitude faster than ProB.
- There is a large variance in the performance of the tools, especially in the presence of "performance killers".

The source code of the benchmarks is in Appendix C.1.

5.6 A CSP$_M$-to-Haskell Compiler

The Master thesis of Dobrikov describes a compiler from CSP$_M$ to Haskell [12].[11]

CSP$_M$ is a pure functional programming language and, at first sight, translating CSP$_M$ to Haskell seems straightforward. Indeed, as far as I know, an earlier version of FDR was based on a translation from CSP$_M$ to ML.

[10]The FDR manual clearly states that FDR uses 32-bit integers and declares that this is a feature and not a bug.

[11] The master thesis was carried out as part of the CSP$_M$ project of the author.

However FDR, which is the de facto reference implementation of CSP$_M$, has evolved over many years and today the language contains many particularities. This means that, although it is easy to translate small functions from CSP$_M$ syntax to Haskell, it is rather difficult to implement a CSP$_M$-to-Haskell compiler which works with off-the-shelf CSP$_M$ specifications.

The compiler presented by Dobrikov is systematically derived from the interpreter presented in this thesis, following an approach called staged interpretation. The compiler is based on the following ideas:

Replace the `eval` function with a `compile` function.

The `eval` function is systematically replaced with a `compile` function. The `compile` function performs exactly the same dispatch on the AST as the `eval` function, but instead of returning a CSP$_M$ value the compile function returns the Haskell expression which computes the value. A function call causes a recursion in the `eval` function when executed by the interpreter. The execution of a function call is replaced with the Haskell code for calling a function.

This approach is related to staged interpretation and partial evaluation. Ideally, these techniques can be used to mechanically derive a compiler from an interpreter. The compiler presented by Dobrikov had still been implemented manually.

Replace CSP$_M$ variables with Haskell variables.

The CSP$_M$ interpreter maintains its own environment, which stores the values of all variables that are in scope. In the compiled code, each CSP$_M$ variable corresponds to exactly one Haskell variable and the explicit environment managed by the interpreter is replaced with the native Haskell mechanism for variable binding.

Keep dynamic typing and the boxed representation of values.

The CSP$_M$ compiler uses the same data model as the interpreter. All CSP$_M$ values are mapped to one big sum type and the CSP$_M$ built-in functions dynamically check that their arguments are of the right type at run-time. In other words, the compiler uses a boxed representation of values.

Reuse the implementation of CSP$_M$ core semantics.

The CSP$_M$ compiler reuses the implementation of the CSP$_M$ core semantics presented in Chapter 4. This means that tracing CSP$_M$ processes and computing the LTS of a process work out of the box.

Preliminary results

The compiler is interesting for several reasons:

- It provides an alternative implementation of the functional sub-language of CSP$_M$ and demonstrates the reusage of the implementation of the CSP$_M$ core semantic.
- It can reduce the interpretation overhead of the interpreter.

- It can be a step towards retiring CSP$_M$ and replacing it with a Haskell-based EDSL.

Section 5.5 contains some preliminary benchmarks of the pure functional performance of CSP$_M$ specifications which have been compiled to Haskell. With the current version of the compiler, the dynamic typing of CSP$_M$ adds an overhead of approximately a factor of three compared to a native implementation of the same function in Haskell. I think that it might be possible to completely eliminate this overhead in many cases. One possible approach is to use an approximate type inference for CSP$_M$ and to generate code for dynamic type checks only for those parts of the specification for which one cannot infer a static type.

5.7 Conclusion

In this chapter, I have described an implementation of an interpreter for the functional sub-language of CSP$_M$. Besides the CSP$_M$ parser and the implementation of the firing rule semantics of CSP, the interpreter is one of the main building blocks of a CSP$_M$ animator and model checker.

I have listed some design alternatives and provided arguments for the main design decisions that were take for the interpreter. The main design decisions are:

- Implement a pure interpreter which does not perform side effects.
- Use dynamic typing and a single sum type for CSP$_M$ values.
- Use an explicit environment, do not use HOAS or “non-ground” representations.
- Implement CSP$_M$-`lets` via Haskell-`lets` and knot tying.
- Implement equality via hash values.

I have also shown some exemplary parts of the interpreter source code. The full source code is online in the central repository for cabal packages (Hackage) in package `CSPM-Interpreter`.

Section 5.5 contains some preliminary benchmarks for the performance of the interpreter. I compare the performance with ProB, FDR, native Haskell and a new CSP$_M$-to-Haskell compiler. The benchmarks show that the interpreter competes well against ProB and FDR. It is approximately one order of magnitude faster than ProB and about two times faster that FDR. Still, the interpretation overhead of the interpreter is approximately a factor of 30 compared to native Haskell (GHC).

The benchmarks also show that the performance of my interpreter as well as the performance of ProB is not as predictable as the performance of an industrial strength Haskell compiler. Small syntactic changes, which generally have very little impact on the performance of native Haskell, can drastically change the performance of my interpreter or the performance of ProB. An example is replacing pattern matching with `if-then-else`.

The CSP$_M$ tools are more likely to suffer from "performance killers" than an industrial strength functional programming language. This is not really surprising, given the fact that my interpreter as well as the CSP$_M$-part of ProB have a very low number of active developers and only a moderate number of users.

Finally, this section contains an outlook of a CSP$_M$-to-Haskell compiler that was developed by Ivaylo Dobrikov as part of his master thesis. The compiler reuses the implementation from Chapter 4 and has the same interface as the interpreter. In principle, it can replace the interpreter as building block of a CSP$_M$ tool. This compiler is still in early development and it is not yet as robust as the interpreter, but it already performs well in the benchmarks.

Chapter 6

Parser

This chapter describes the lexer and parser that serve as front-end of my CSP$_M$ tool. The front-end takes a string, containing a CSP$_M$ specification, as input and computes the corresponding abstract syntax tree. The package containing the front-end is available online at Hackage, the central repository for cabal packages, as `CSPM-Frontend`.

The interface of the parser contains several functions for lexing and parsing. A good starting point for testing the front-end is the function

```
parseFile  :: FilePath  → IO ModuleFromParser
```

from module `Language.CSPM.Frontend`. The module `Language.CSPM.AST` contains the definition of the data types that are used for the abstract syntax tree. The AST of a complete module has type `ModuleFromParser`.

I follow the guideline that the parser should be as close as possible to the "real syntax" of CSP$_M$. I have not tried to simplify or "improve" the syntax of CSP$_M$. Instead, I tried to mimic the original parser as closely as possible to ensure a maximal compatibility between our tool and FDR.

This chapter starts with some general remarks about the CSP$_M$ syntax (Section 6.1). After that I describe the structure of the AST (Section 6.2) and finally, I discuss some aspects of the parser implementation and present some benchmarks of the parser performance. The parser is well separated from the rest of the CSP$_M$ tool. The only important interface of the front-end is the AST, which is used by the interpreter in Chapter 5.

The parser is the oldest module of my CSP$_M$ project. Work on the parser started in 2006 and the basic structure of the AST has not changed since the beginning of 2007. Indeed, when my project started, the actual goal of the project was just to develop a CSP$_M$ parser which can be integrated in the ProB tool. The parser has been part of ProB for several years now and my thesis supervisor Michael Leuschel has helped greatly in testing parser. Recently, Ivaylo Dobrikov started to contribute to the parser by testing and also by working on the parser source code. Therefore it makes sense to speak of *our* parser in this section.

6.1 Remarks on the CSP$_M$ Syntax

Machine readable CSP ($\equiv$ CSP$_M$) is the input syntax of the FDR tool. FDR is widely used in the CSP community and accepted as the de-facto standard for CSP tools. For researchers who just use FDR as a black box, this has the advantage that it is easy to compare and exchange CSP$_M$ specifications.

From the point of view of a tool developer, however, there are some aspects of the CSP$_M$ syntax that I do not like. The rest of this section lists my personal most important points of critique. The developers of FDR might disagree with some (or all) of these points and I admit that they are debatable.

6.1.1 Informal Syntax Definition

The available documentation of the CSP$_M$ syntax for the FDR tool is informal and incomplete. It may be adequate for users of the FDR tool but it definitely was insufficient for writing a CSP$_M$ parser. Reverse engineering existing specifications and testing syntax variants with the FDR tool gave much more insights than the existing FDR documentation.

On the other hand, this means that I did apply an implicit closed world assumption. In other words, our parser cannot support syntax which is neither documented nor used in the available examples. It has happened several times that I learned about an new piece of syntax, only because I came across some specification which happens to use this syntax.

Taking FDR as de-facto standard is also problematic for tool developers, because the actual FDR syntax changes from time to time. For example, in the latest FDR version, new built-in operations have been added.

6.1.2 Mixing Built-ins and Core Syntax

The FDR documentation mixes the core syntax of CSP$_M$ with other functions that are built into the FDR tool. In other functional programming languages, there is a clean distinction between the syntax of the language and parts of the language which are not syntax. For example, the Haskell syntax defines the set of keywords of the language and functions like `lengths`, `head` or `tail` are just regular functions, which are imported from a `Prelude` module.

It was only by coincidence and a discussion with an FDR developer that I became aware of the fact that, under the hood, FDR uses a similar approach as Haskell. Parts of the CSP$_M$ programming language are syntax and other parts come from an implicitly imported `Prelude`. However, there is no hint on this in the documentation and I still do not know exactly what is syntax and what is part of the CSP$_M$-`Prelude`.

There are subtle differences between built-in syntax and other functions. For example, one of the latest extensions of FDR is the new `Proc` data type. In FDR, `Proc` is a new predefined identifier which can be redefined in a specification. In our parser however, `Proc` is a built-in keyword which cannot be redefined.[1]

As a side note, here is another example which is related to this: In FDR, `true` and `false` are predefined, but so too are `True` and `False`. However, they are not the same.

[1] This has been fixed in the latest version of our parser.

```
f(x) = if x then 2 else 3

f1(True) =  2
f1(False) = 3

f2(true) = 2
f2(false) = 3
```

`f` and `f2` can return the value 2 or 3 but `f1` always returns 2, because `True` and `False` are just regular identifiers in CSP$_M$. They are predefined as:

```
True = true
False = false
```

In a pattern match `true` only matches the boolean value `true`, while the pattern `True` matches anything and rebinds the identifier `True`.[2]

6.1.3 Mixing Type Checking and Parsing

The FDR parser performs some kind of built-in type checking [53]. For example, the expression `true and 1` is rejected by the parser. To implement this, the grammar of the parser contains different non-terminal symbols for the different types of expressions, namely for Boolean expressions, arithmetic expressions, process expressions, etc.

This has two disadvantages: First, the grammar immediately becomes ambiguous because all variant sub-expressions contain the case of a variable. For example in the declaration `d = x`, `x` could be a Boolean expression consisting of just a variable, it could as well be an arithmetic expression or an expression of any other type. Scattergood has to use an AWK-script which removes these ambiguities and which generated the actual grammar that is fed into the parser generator (BISON in that case).

The second and more serious disadvantage is that it highly complicates the parser. Typically, functional programming languages use one non-terminal `expr` for expressions and all binary infix operators are of the form `expr op expr`. To add a new infix operator, one just has to define the precedence and the fixity of the operator, i.e. whether it is right associative, left associative or non-associative. This has the advantage that it is possible to deal with the precedences and fixities in a systematic way. Operators can be added without changing the syntax or the parser.

The Scattergood parser works differently, however. Since it uses different non-terminals for different types of expressions, it is no longer easy to assign precedences to the infix operations. The parser becomes so involved that the only possible description of the CSP$_M$ syntax is indeed the implementation of the parser itself.

In my opinion, using the parser for type checking is also a bad idea, because this does by no means make up for a proper type checker. The parser only catches the most obvious type errors anyway.

[2] In Haskell, `True` and `true` are also something different, but this is trivial and every Haskell programmer knows about this. The Haskell behavior is exactly the opposite of the FDR behavior.

6.1.4 Strange Syntax

CSP had been in use as a mathematical notation for some time when work on CSP$_M$ started and this, again, was long before Unicode became popular. CSP$_M$ was designed to look similar to the CSP notation that was used in literature and on the blackboard.

Maybe this is the reason why CSP$_M$ uses `[]` for $\Box$, and this again might be the reason why `[` and `]` are not used for lists (as in most other functional languages). Instead lists are written as `<1,2,3>`. This is problematic, since `<` and `>` are also used for the greater-than and less-than relation.

From the point of view of a Haskell programmer, another strange feature of CSP$_M$ syntax is that parentheses are part of function application. A function which takes two arguments can be defined in curried form, for a tuple and in a third variant which does not exit in Haskell. CSP$_M$ syntax:

```
f(x)(y)  = ... -- curried form
f((x,y)) = ... -- tuple form
f(x,y)   = ... -- CSPM specific third variant
```

Haskell syntax:

```
f x y   = ... -- curried form
f (x,y) = ... -- tuple form
```

It is unclear why a third variant has been added to CSP$_M$.

6.1.5 Operator Precedences

The CSP$_M$ syntax uses many prefix and infix operators with specific precedences and fixities. It is doubtful that there are many users who are able to correctly remember the precedences and decide where parenthesis are necessary or not. Complex precedences are also problematic, because all terms, that are built with processes and process operations, are valid specifications regardless of the precedences. There is no type error if the user misunderstands the precedences of process operators—the specification just behaves different. I think it would have been better to make the process operators non-associative, such that parenthesis become mandatory.

6.1.6 Constructor and Channel Names

In CSP$_M$, constructors and channel names are fundamentally different from variable names. A similar distinction exists, for example in Haskell. But in Haskell, this distinction is clearly visible, since all constructor names must start with an upper-case letter. On the other hand, there is no such rule for CSP$_M$. In CSP$_M$, data constructors, variables and channel names can start with lower-case letters as well as upper-case letters. This can lead to hard to find bugs.

For example, let's consider the following declaration:

```
P = in?val → out!val → P
```

The obvious interpretation of this declaration is that `val` is a locally-bound variable, but this interpretation is not generally true. A counter example is the specification:

```
channel in,out : {1..10}

P = in?val → out!val → P

channel val
```

The channel definition `channel val` defines a new constant `val` which means that the identifier `val` can no longer be used as a variable name.

FDR accepts the above example without any warning, but the process `P` does, most likely, not behave as intended.[3] Note that there can be many lines of code between the definition of `channel val` and the use of `val`.

It is easy to break existing specifications by adding a new channel declaration. The problem is, that there seem to be locally scoped variables in CSP$_M$ but, in fact, there are not. A channel or data type declaration can interfere with any seemingly local variable. If one wants to add a channel declaration or if one wants to combine several specifications one has to check the complete specification for bad side effects.

6.2 The AST Data Types

This section describes the data types which represent the abstract syntax tree of CSP$_M$ specifications. The complete definitions of all types can be found in module `Language.CSPM.AST` in Appendix B.1.10.

In general, there are design trade-offs between making a syntax tree more abstract or closer to the concrete syntax. For example, the logical `and` operation of CSP$_M$ could be represented abstractly as a binary function, or it could be explicitly built into the data type of the abstract syntax tree as a special case. The advantage of the abstract representation is that `and` is indeed semantically a binary function. On the other hand the explicit representation is closer to the syntax because `and` is syntactically a special built-in of CSP$_M$. Another example are parentheses. Semantically, parenthesis are superfluous in the abstract syntax tree. However, it is still beneficial to represent them explicitly.

One design decision was that the parser should return an abstract syntax tree that is close to the concrete syntax. In other words, constructs like the CSP$_M$-`and` and parentheses are represented explicitly. One motivation for this design decision was that the parser is also used for refactoring tools and for the CSP$_M$ slicer [35]. A drawback of this design is that some of the particularities of the CSP$_M$ syntax are also present in the AST.

6.2.1 Source Locations and Node Labels

Almost all parts of the AST carry information about source locations. I annotate source locations to an AST node by wrapping it with the data type `Labeled`:

```
data Labeled t = Labeled {
    nodeId :: NodeId
   ,srcLoc  :: SrcLoc
   ,unLabel :: t
   } deriving (Typeable, Data,Show)
```

[3]The type checker that is available from FSE can catch this error.

For convenience, I define several type aliases for labeled nodes, for example

```
type LIdent = Labeled Ident
type LExp = Labeled Exp
type LPattern = Labeled Pattern
type LDecl = Labeled Decl
```

By convention, labeled nodes have a type name prefixed with a `L`.

A lot of effort was put into making the source locations accurate. I follow the rule that the source location of an AST node is always the part of the source code that has been parsed to produce the node. For example, the source span of `if x == 0 then 7 else 8` covers the `if`, although `if` itself is a token and tokens are not themselves represented in the AST.

The following interface can be used to access source locations:

```
type SrcLine = Int
type SrcCol  = Int
type SrcOffset  = Int

getStartLine :: SrcLoc → SrcLine
getStartCol :: SrcLoc → SrcCol
getStartOffset :: SrcLoc → SrcOffset
getEndLine :: SrcLoc → SrcLine
getEndCol :: SrcLoc → SrcCol
getEndOffset :: SrcLoc → SrcOffset
```

Currently source locations do not include the file name of the specification, which means that they are *useless* if a specification uses the `include` feature of CSP$_M$.

6.2.2 Identifier

I use two variants of identifiers in the AST:

```
data Ident
  = Ident  {unIdent :: String}
  | UIdent {unUIdent :: UniqueIdent}
```

`Ident` is a simple string, while `UIdent` stores additional information about the identifier. The parser always returns the first variant. To convert the AST to the second variant, a renaming operation is used.

6.2.3 Additional Constraints on Abstract Syntax Trees

The AST, that is returned by the parser, is passed through several transformations (renaming, free-names analysis) before it is suitable for the interpreter. One design consideration is whether to use separate data types for the intermediate representations or not.[4]

My decision was to use only one data type for all intermediate steps. This approach is simpler but it has the drawback that the type system provides fewer guarantees for the AST. In other words, not all type-correct ASTs are

[4]There are also fancier alternatives like open recursions and type-level fixed points [55]. However, it is unclear whether these techniques are practicable in real-world applications.

valid. There are additional constraints on valid ASTs that are not enforced by the Haskell type system.

For example an identifier has two possible representations `Ident` and `UIdent` (see Section 6.2.2). An AST may only be passed to the renaming transformation if all identifiers in the AST are of variant `Ident`. Otherwise, the renaming transformation will throw a runtime exception. Similarly, the interpreter only works with an AST if all identifiers are of variant `UIdent`. These constraints are not enforced by the type system. See also Section 5.3.4. Investigating a more rigorous design is left as future work.

Total functions help to build clear application interfaces because the type of the function arguments exactly defines all permissible input values. Unfortunately, for the reasons described in this section, the functions of the front-end module are not total. The user of the module should be aware of the additional constraints on valid ASTs.

6.2.4 Expressions

The expression data type (`Exp`) is the biggest type with 43 constructors. It depends on several other types, for example the type for patterns (`Pat`), and it is mutually recursive with the type for declarations (`Decl`). Table 6.1 and Table 6.2 list the constructors and a piece of concrete CSP$_M$ syntax that produces the corresponding AST node. The constructors can be used to "grep" in source code and find all relevant parts of the parser and interpreter that deal with the specific syntactic construct.

The `Exp` data type contains several constructors which do not appear in the ASTs that are returned by the parser. These constructors can appear in syntax trees that are returned by AST transformations. For example, there is an AST transformation which adds information about free names to the abstract syntax tree. This transformation replaces every `PrefixExp`-node with a `PrefixI`-node. The parser itself does not generate `PrefixI` nodes (c.f Section 6.2.3, Section 6.2.2). All infix operations (`+`,`*`,`[]`,etc.) are subsumed by the `Fun2` node type.

6.2.5 Declarations

The data type `Decl` (Table 6.3) represents CSP$_M$ declarations. The parser makes a distinction between top level CSP$_M$ declarations and local declarations with `let-within`. Inside a `let-within` construct, the parser accepts only function declarations (`FunBind`) and declarations via pattern match (`PatBind`), but not channel, data type and name type declarations, which can only appear at the top level. The AST uses the same type `Decl` for both top level CSP$_M$ declarations and local declarations.

Functions are a special case because they have two alternative representations in the AST. When a function is defined via pattern matching for several cases, like, for example:

```
fkt(1) = 1
fkt(x) = x+1
```

each function case is recognized by the parser as a separate declaration, i.e. a separate AST node. This is the first representation. In the renaming phase, this representation is translated to the second representation by merging consecutive

Constructor	Example	Description
`Var`	`x`	a variable
`IntExp`	`10`	an integer constant
`SetExp`	`{1,2,x|x <-{2,3} }`	a set, possibly a set comprehension
`ListExp`	`<1,2,x|x <-<2,3»`	a list, possibly a list comprehension
`ClosureComprehension`	`{|x,y| x <- c.1 |}`	event closure comprehension
`Let`	`let x=1 within x+x`	let expression
`Ifte`	`if a then b else c`	if-then-else expression
`CallFunction`	`myfun (3,4)(7)`	function call (not for built-ins)
`CallBuiltIn`	`null(x)`	call a built-in function
`Lambda`	`(\x@x*x)(10)`	lambda expression
`Stop`	`STOP`	STOP-process
`Skip`	`SKIP`	SKIP-process
`CTrue`	`true`	Boolean constant
`CFalse`	`false`	Boolean constant
`Events`	`Events`	a special set that represents all possible events
`BoolSet`	`Bool`	the set `{true,false}`
`IntSet`	`Int`	the set of all integers
`TupleExp`	`(1,2)`	a tuple
`Parens`	`(3+4)`	an expression in parenthesis
`AndExp`	`a and b`	Boolean `and`-operation
`OrExp`	`a or b`	Boolean `or`-operation
`NotExp`	`not(true)`	logic negation
`NegExp`	`-x`	arithmetic negation

Figure 6.1: CSP$_M$ expressions

Constructor	Example	Description
`Fun1`	`#<1..10>`	unary prefix operation for length of list
`Fun2`	`3+4`	a binary infix operation
`DotTuple`	`1.2.3`	a dot tuple
`Closure`	`{|c1,c2,c3|}`	a simple closure operation
`ProcSharing`	`P[|c|]Q`	sharing
`ProcAParallel`	`P[a||b]Q`	alphabetized parallel
`ProcLinkParallel`	`P[ c1 <-> c2 ]Q`	linked parallel
`ProcRenaming`	`P[[ a<-b ]]`	renaming
`ProcRepSequence`	`;x:s @ P(x)`	replicated sequential composition
`ProcRepInternalChoice`	`|~|x:s @ P(x)`	replicated internal choice
`ProcRepExternalChoice`	`[]x:s @ P(x)`	replicated external choice
`ProcRepInterleave`	`|||x:s @ P(x)`	replicated interleaving
`ProcRepAParallel`	`|| x:s @ [a] P(x)`	replicated alphabetized parallel
`ProcRepLinkParallel`	`[l<->r]x:s@P(x)`	replicated linked parallel
`ProcRepSharing`	`[|a|] x:s @ P(x)`	replicated sharing
`PrefixExp`	`c -> P`	prefix operation
`PrefixI`		not generated by the parser
`LetI`		not generated by the parser
`LambdaI`		not generated by the parser
`ExprWithFreeNames`		not generated by the parser

Figure 6.2: CSP$_M$ expressions (cont.)

patterns of the same function into one AST node. It is important to destinguish both representations, as both use the constructor `FunBind LIdent [FunCase]` for the AST node.

Constructor	Example	Description
`PatBind`	`(fst,snd)=(1,<1..10>)`	declaration via pattern match
`FunBind`	`f(x)=x*x`	declaration of a function
`Assert`	`assert P [FD= Q`	an assertion
`Transparent`	`transparent f1,f2`	declare transparent functions
`SubType`	`subtype C=R|B`	subtype declaration
`DataType`	`datatype C = R|G|B`	data type declaration
`NameType`	`nametype V = 1..9`	name type declaration
`Channel`	`channel c1,c2`	channel declaration
`Print`	`print 4+2`	a print statement

Figure 6.3: CSP$_M$ declarations

6.2.6 Patterns

Pattern matches can appear in CSP$_M$ in several contexts:

- In function declarations.
- In local declaration with `let-within`.
- In input and guarded-input prefix operations.
- In lambdas and replicated operations.
- In comprenensions.

All thoses contexts use the same data type `Pat`. Table 6.4 lists the constructors of the `Pat` data type.

6.2.7 SYB

The following class instances are derived for all data types of the AST:
`Show`, `Eq`, `Ord`, `Typeable` and `Data`.
The `Typeable` and `Data` instances are needed for the SYB framework.

SYB (Scrap-Your-Boilerplate) [30] is a Haskell framework that helps to implement succinct AST transformations. For example, the following function removes all occurrences of explicit parentheses (nodes with constructor `Parens`) from the AST:

```
removeParens :: Data a ⇒ a → a
removeParens ast
  = everywhere (mkT patchExp) ast
  where
    patchExp :: LExp → LExp
```

Constructor	Example	Description
`IntPat`	`fib(1)=1`	an integer constant pattern
`TruePat`	`f(true,true)=1`	match Boolean true
`FalsePat`	`f(false,false)=1`	match Boolean false
`WildCard`	`fun(_)=true`	wildcard pattern; match everything
`ConstrPat`	`fun(Red)=1`	match a constructor
`Also`	`x@@(f,s)`	match two patterns in parallel
`Append`	`<h> ^ b ^ <l>`	list decomposition
`DotPat`	`c?(a.b.c)->P`	dot-tuple pattern
`SingleSetPat`	`f({x})=`	set decomposition
`EmptySetPat`	`f({})=`	match an empty set
`ListEnumPat`	`f(<a,b,c>)=`	match a fixed-length list
`TuplePat`	`f((a,b,c))`	match a tuple
`VarPat`	`f(x)=x*x`	match everything and bind a variable
`Selectors`	no syntax	only used by pattern compiler
`Selector`	no syntax	only used by pattern compiler

Figure 6.4: CSP$_M$ patterns

```
  patchExp x = case unLabel x of
    Parens e → e
    _ → x
```

To delete all information about source locations, or more precisely to set all source location information to `SrcLoc.NoLocation`, the following function can be used:

```
removeSourceLocations :: Data a ⇒ a → a
removeSourceLocations ast
  = everywhere (mkT patchLabel) ast
  where
    patchLabel :: SrcLoc.SrcLoc → SrcLoc.SrcLoc
    patchLabel _ = SrcLoc.NoLocation
```

SYB provides the combinators `everywhere` and `mkT`. The combinator `everywhere` applies a transformation on all nodes of a tree in bottom-up order.

`removeParens` works with all types that are an instance of the type class `Data`. The type class `Data` provides a reflection of the structure of a type. `Data` is a subclass of `Typeable`. The `Typeable` type class makes it possible to write functions that dispatch on the type of a value. The combinator `mkT` turns a regular function into a generic transformation. For example, the function `mkT patchExp` calls `patchExp` if the argument is of type `LExp` and behaves as the identity function otherwise.

The Haskell term for techniques like SYB is *generic programming*.[5] Overall, generic programming is a technique for writing flexible data transformations in Haskell. Generic programming combines expressivity with statically-checked type safety.

Strictly speaking, *generic programming* in Haskell solves a problem which does not exist in dynamically typed languages. For example, similar data transformations can be written in Prolog with the help of the `'=..'/2` and the `functor/3` predicate. On the other hand, Prolog uses untyped Prolog-terms instead of algebraic data types which means that Prolog provides no type safety at all.

My personal experience was that SYB is a powerful tool and that it works well in practice. On the other hand, SYB is still a heavy-weight approach under the hood. Therefore, I only show the code examples above and do not go into details. The technical report “Libraries for Generic Programming in Haskell” [25] compares SYB and several other generic programming frameworks for Haskell and contains a good survey on Haskell generic programming. The paper “Scrap your boilerplate: a practical design pattern for generic programming” [30] is a good reference for SYB.

6.3 The Combinator Parser

The two main characteristics of combinator parsers in Haskell are:

1. The parser is specified in Haskell syntax; there is no need for special purpose syntax for the parser specification.

[5] Generic programming in Haskell has nothing to do with Java generics.

2. Parsers are regular Haskell values and, as such, they are first class citizens. In other words, parsers can appear as arguments of functions, as return type of functions, and they can be stored in data types like lists.

Combinators are just regular functions which are used to build bigger parsers out of smaller parsers. Table 6.5 lists some combinators that are typically used in combinator parsers. Appendix C.2.1 contains a simplistic combinator parser for a Pascal-like language.

Combinator	Description
`>>=`,`>>`	concatenation
`<|>`	biased choice
`many`	replication, ($\star$)-operator
`char`	primitive parser for one character
`string`	primitive parser for one string
`eof`	recognize end of file

Figure 6.5: Some combinators

There are several similar combinator parsers available for Haskell. Some of them are called monadic parser combinators, while others promote an applicative style of parsers. The term *combinator parser* really only implies that the parser is specified with the help of combinators (i.e. functions). It does not say anything about the internal implementation of the parser—for example, it can be top down, bottom up or a mix of top down and bottom up.

The CSP$_M$ front-end uses the `parsec` combinator parser [31], which was state of the art when the development of the parser started in 2006. Although there is quite a variety of combinator parsers available for Haskell, most of them are very similar to `parsec`.

6.3.1 Pros and Cons of `parsec`

This section lists some of the advantages and disadvantages of `parsec` style parsers. Note that the term *combinator parser* is also used for radically different implementations [16] which are not covered by the following discussion.

The advantages of `parsec` parsers are:

- There is no need for a special syntax for the parser specification.
- There is no need to run a parser generator when compiling the parser.
- The parser is thoroughly type checked.
- The parser is flexible. There is no restriction to LALR or any special class of grammar.

Possible disadvantages are:

- The parser is not as declarative as the grammar of a parser generator.
- The syntax definition is Haskell code, it cannot easily be translated to a `bison` grammar, for example.

- The parser is implicitly never ambiguous.
- There are no performance guarantees for the parser.

To some extent the disadvantages are just the other side of the advantages. With `parsec`, it is easy to include any Haskell code in the parser. This means that the parser is flexible. It is not restricted to LALR grammars or any other class of grammars. The accepted language does not even have to be context free. On the other hand, this means that there are no general performance guarantees for the parser and that it is not generally possible to translate a `parsec`-parser to a `bison`-grammar.

The decision to use `parsec` for the implementation of the CSP$_M$ parser was not based on technical reasons.[6] In retrospect, it turned out that `parsec` worked well. In particular, `parsec` allows for an incremental style of development. It was possible to start with the basic syntax of CSP$_M$ and add more complex features later.

6.3.2 Code Examples

This section contains some excerpts of the actual parser code. All sub-parsers are of type `PT a` where `a` is the return type of the parser. The parser maintains an internal state of type `PState` which stores the last token, a counter for node-IDs and flags that are needed to deal with `>` and `<` tokens.

```
type PT a= GenParser Token PState a
data PState
 = PState {
  lastTok        :: Token
 ,gtCounter      :: Int
 ,gtMode         :: GtMode
 ,nodeIdSupply   :: NodeId
 } deriving Show
```

I define a new combinator `withLoc`, which takes a parser for an unlabeled AST-node and returns a parser for the corresponding labeled node.

```
withLoc :: PT a → PT (Labeled a)
withLoc a = do
  s ← getNextPos
  av ← a
  e ← getLastPos
  mkLabeledNode (mkSrcSpan s e) av
```

These are the parsers for let-expressions and if-then-else-expressions:

```
letExp :: PT LExp
letExp = withLoc $ do
  token T_let
  decl ← parseDeclList
  token T_within
  exp ← parseExp
  return $ Let decl exp
```

[6] Basically, I had already used a traditional parser generator before. When the CSP$_M$ project started, I was just starting to learn Haskell and wanted to use a "haskellish" parser. `parsec` happened to be en vogue at that time.

```
ifteExp :: PT LExp
ifteExp = withLoc $ do
  token T_if
  cond ← parseExp
  token T_then
  thenExp ← parseExp
  token T_else
  elseExp ← parseExp
  return $ Ifte cond thenExp elseExp
```

6.3.3 Parser Performance

A draw back of the `parsec` package is that there are no general theoretic guarantees for the performance. Features such as infinite look-ahead and non-deterministic choice make `parsec` a powerful parser framework, but easy-going use of these features can yield parsers with very bad (i.e. exponential) complexity. Its the responsibility of the parser programmer to ensure that his `parsec`-parser meets the complexity requirements.

Parser Performance					
Specification	Size	LOC	Tokens	Time	Speed
	kb			*ms*	*Tokens/s*
abp.csp	12	373	795	20	approx. 40000
Andrew.csp	11	394	3221	110	approx. 30000
crossing.csp	13	360	1416	30	approx. 50000
roscoe_chapter4.csp	6	181	602	20	approx. 30000
roscoe_section2-1.csp	5	141	483	20	approx. 25000
mangle.csp	13	80	2848	90	approx. 30000
scheduler0_1.csp	31	1121	3362	70	approx. 50000
SetTests.refcheck.csp	566	8575	60611	1680	approx. 40000

Figure 6.6: Parser performance

Our parser was tested with about 400 real-world CSP$_M$ specifications. Apart from this, the parser has also been in use as the CSP$_M$ front-end of the ProB tool for about three years. The tests show that our parser works reasonably well in practice. Table 6.3.3 shows some benchmarks. The parsing speed is approximately 35,000 tokens per second.

The main applications of the parser are hand-written specifications which are typically not not much bigger than 10kb. I have not made any attempt whatsoever to speed-up the parser. Nevertheless, the parser also works reasonably well with large, machine generated specifications like `scheduler0_1.csp` and `SetTests.refcheck.csp`.

Still, empirical results are no guarantee that the parser will always behave well. For example, in the course of writing the parser benchmarks, a bug was discovered (and fixed) which causes an exponential running time of one test case. The bug only showed up in this particular test case, namely `mangel.csp` and had been undiscovered for some while. `mangle.csp` is a machine-generated specification which contains deeply nested, full parenthesized expressions. Before the bug-fix, `mangle.csp` had a parsing time of 23 seconds. After the fix, the parsing time was reduced to 90 ms.

My conclusion is that `parsec` is a powerful tool for writing parsers and I think that it was the right choice for reverse engineering the CSP$_M$ parser. On the other hand, it is problematic that there are no performance guarantees for the parser.

6.4 Other Functionality Provided by the Front-End Package

The `CSPM-Frontend` package contains some further functionality, which is unrelated to parsing but which has been put in the package for convenience.

6.4.1 Renaming

To simplify the handling of variables, I use a renaming step which computes a unique ID for each variable. For example the CSP$_M$ expression

```
λx@ (x + (λx@x ∗ x)(x))
```

contains two distinct uses of variable `x`. It gets renamed to:

```
λi1@(i1 + (λi2@ i2 ∗ i2)(i1))
```

I have implemented the renaming function as a top down traversal of the syntax tree. The function maintains as state the information about the environment, i.e. which variable names are in scope. It searches for two interesting variants of nodes: nodes that bind new names and nodes that use a bound name.

Whenever the function reaches a node which binds a new name, it generates a new unique ID and adds a corresponding record to the environment. When it reaches a used occurrence of a variable name it checks that the name is bound and remembers information about the use-site. The renaming function checks for two kinds of errors: uses of unbound variables and illegal redefinitions of variables within a pattern (for example `fun(x,x)=x+x`).

A module can be renamed by calling the function:

```
renameModule ::
     ModuleFromParser
  → Either RenameError (ModuleFromRenaming, RenameInfo)
```

`renameModule` either returns the renamed module (`ModuleFromRenaming`) and the information that was gathered during the renaming (`RenameInfo`) or `RenameError` in case of an error.

```
type AstAnnotation x = IntMap x
type Bindings = Map String UniqueIdent

type UniqueName = Int

data RenameInfo = RenameInfo
  {
    nameSupply :: UniqueName
   ,localBindings :: Bindings
   ,visible  :: Bindings
   ,identDefinition :: AstAnnotation UniqueIdent
   ,identUse  :: AstAnnotation UniqueIdent
   ,usedNames :: Set String
   ,prologMode :: PrologMode
   ,bindType   :: BindType
  } deriving Show

type RM x = StateT RState (Either RenameError) x
rnModule :: LModule → RM ()
```

The actual work is done in several mutually recursive functions which dispatch on all alternatives of the AST data type. These functions are defined in terms of monad `RM`. The function `rnModule` is the entry point of the actual traversal.

Renaming also takes care of the scoping rules for channel names and data constructor names. As explained in Section 6.1.6, the occurrence of a name like `x` in `f(x)=...` has a different semantics, depending on whether a channel declaration for `x` is in scope or not. This cannot be detected by the parser and therefore the parser always parses names in pattern as variable bindings, i.e. AST nodes of variant `VarPat LIdent`. The renaming function checks whether a name is indeed a channel name and if so it rewrites the corresponding AST nodes to `ConstrPat LIdent`.

6.4.2 Interface to ProB

The package `CSPM-Frontend` is also the CSP$_M$ front-end of the ProB model checker. ProB reads the AST of a CSP$_M$ specification in form of a set of Prolog clauses. The current Prolog encoding requires that renaming has been applied on the AST and the encoding also requires some coarse information about used variable names. This information is also gathered by the renaming function.

In particular, the following additional information is computed for identifiers:

```
data IDType
  = VarID | ChannelID | NameTypeID | FunID Int
  | ConstrID String | DataTypeID | TransparentID
  | BuiltInID

data BindType = LetBound | NotLetBound

data PrologMode = PrologGround | PrologVariable
```

6.5 Conclusion

This section describes the CSP$_M$ front-end of my CSP$_M$ tool. The source code for the front-end can be found on Hackage in package `CSPM-Frontend`.

The section starts with some remarks about the CSP$_M$ syntax. The CSP$_M$ syntax contains several doubtful features that make it needlessly difficult to parse. Nevertheless, an important requirement for our parser is that it has to be as compatible as possible with the front-end of the FDR tool. In other words, I decided to accept the CSP$_M$ syntax as it is and did not try to change the syntax. Another challenge, was that many details of the CSP$_M$ syntax could only be determined by a trial-and-error method. The existing documentation of the FDR tool turned out to be insufficient.

After the introductory remarks I describe the data structure for abstract syntax trees for CSP$_M$. The AST is the most important external interface of the front-end. It is implemented as a set of mutually recursive data types. I show tables which list the constructors of the AST types, a corresponding piece of concrete CSP$_M$ syntax and a small description of the syntax for expressions (`Exp`), declarations (`Decl`) and patterns (`Pat`). Appendix B.1.10 contains a complete listing of the definitions for the AST data types.

Finally, I discuss some aspects of the parser source code. The actual parser is implemented as a combinator parser, based on the `parsec` package. This approach allows for an incremental and flexible development of the parser. On the other hand there are no general performance guarantees for `parsec` parsers. I list some benchmarks with real-world examples which show that our parser works well in practice.

The overall goal is to implement a reusable and practicable parser using a state of the art parser library. A requirement for the parser library was that it should have the flexibility to deal with surprises in the CSP$_M$ syntax that were not known a priori. It turns out that the `parsec` library meets these requirements.

Parsers and parser generators are a topic of very active research. Other Haskell libraries for parsing are, for example: HAPPY [37], the PEG parsers [15] and the parsers described by D. Swierstra [58].

Chapter 7

Exploiting Multi-Core Parallelism

This chapter presents an experiment on the question of if it is possible to speed up my CSP$_M$ tools with the help of multi-core parallelism and how difficult it is. The result of the experiment is very promising.

I have implemented a prototype of a parallel CSP$_M$ model checker which is is up to a factor of 5.5 times faster than the corresponding single core version. At the same time, the complete implementation of the model checker consists of a single module of approx. 50 lines of code and makes up only a tiny fraction of the CSP$_M$ project. Given this little effort the speed-up is very reasonable. I will describe the implementation and provide benchmarks.

Model checking often involves exhaustively searching the state space of a specification. In principle, model checking is a promising candidate for parallelization [40]. At the same time, Haskell (GHC) has good support for multi-core parallelism and, therefore, it was obvious to try to exploit this feature.

Haskell features several different approaches to parallelism and concurrency. Among them are explicit threads, locks, transactional memory, data parallel processing and so-called semi-explicit parallelism [39]. This experiment is based on semi-explicit parallelism, which has been available in GHC for some time. I discuss semi-explicit parallelism in Section 7.3.

Just recently, the `monad-par` library—a new, monad based abstraction for parallelism in Haskell—has been proposed [56]. Repeating the experiment with the `monad-par` library is interesting future work.

7.1 Parallel Breadth First LTS Computation

Chapter 4 describes an algorithm for computing all possible transitions of a CSP process, i.e. the operational semantics of CSP$_M$. Given the transition relation and the start process, it is straightforward to exhaustively search the state space of a specification, for example with a depth first traversal or a breadth first traversal or some combination of both.

In this section, I present a function which computes the possible transitions of a set of states in parallel. I want to make a clear distinction between the parallel computation of the transitions of a single state and the parallel computation of the transitions of a set of states. The parallel computation of the transitions of a single state may be possible, but we have not tried this.

Pseudo Code

The textbook version of BFS maintains a queue of graph nodes that need to be visited. The main loop of textbook BFS consists of the following steps:

Step 1 Initialize the queue with the start node.

Step 2 Take a node n from the queue.

Step 3 Compute the successor nodes of n.

Step 4 Add those successor nodes that need to be visited to the queue.

Step 5 If the queue is not empty jump to Step 2.

Textbook BFS processes one node after the other. The difference between the textbook version of the algorithm and my parallelized breadth first search is that the parallelized algorithm proceeds in waves instead of using a queue. A wave is just a set of nodes that is processed in parallel. The following imperative pseudo-code gives a rough intuition parallelized BFS that was implemented. The algorithm consists of the following steps:

Step 1 Start with a wave that consists of just the initial node.

Step 2 If the current wave is empty the search is finished.

Step 3 Otherwise, compute the successor nodes of all nodes in the current wave in parallel.

Step 4 Examine all successor nodes in parallel and filter out old nodes that have been visited before.

Step 5 Jump to Step 2 and iterate with the new wave of nodes that needs to be visited.

This pseudo code gives an intuition of the algorithm but it does not describe the true operational semantics of my implementation. The following section describes the source code of my implementation as is, and I will discuss the tensions between the source code and the operational semantics in Section 7.3.

Source Code

```
mkLtsPar :: Sigma INT → Process INT → LTS
mkLtsPar events process
  = wave [mkLtsNode process] Map.empty
  where
    wave :: [LtsNode] → LTS → LTS
    wave [] lts = lts
    wave w lts = wave (Set.toList uniqueProcesses) newLts
      where
        processNext = ...
        transitions = ...
        uniqueProcesses = ...
        newLts = ...
```

The `wave` function processes the list of new LTS nodes which need to be visited and maintains as state the current LTS. If there are no new nodes, `wave` is done and it returns the current LTS (line 6). Otherwise, it recursively calls `wave` on the new LTS (line 7). `mkLtsPar` is a wrapper which calls `wave` with a list containing only the start-process and an empty LTS (line 3). The type `LtsNode` is just a regular process extended with a digest.

```
data LtsNode
  = LtsNode {
    nodeDigest :: !Interpreter.Digest
   ,nodeProcess :: Interpreter.Process
   }


mkLtsNode :: Interpreter.Process → LtsNode
mkLtsNode p = LtsNode {
   nodeDigest = hash p
  ,nodeProcess = p }
```

I use the digest for fast equality and ordering on `LtsNode`s (c.f. Section 5.4).

```
instance Ord  LtsNode where compare = comparing nodeDigest
instance Eq   LtsNode where (==) = on (==) nodeDigest
```

An `LTS` is just a `Map` from `LtsNode`s to a list of transitions:[1]

```
type LTS = Map LtsNode [Rule INT]
```

Fast equality and ordering on `LtsNode`s is important in speeding-up the operations on the `LTS`. The `LTS` uses the data type `Map` which is implemented as a binary tree.

The function `wave` refers to `uniqueProcesses` and `newLts` which are local declarations. Note that `w` and `lts` are in scope of the `where`-part. I describe the local declarations in bottom-up order.

```
processNext :: LtsNode → (LtsNode, [Rule INT], [LtsNode])
processNext p = (p, rules, map (mkLtsNode ∘ viewProcAfter) rules)
  where rules =  computeTransitions events $ nodeProcess p

transitions = map processNext w

newLts = List.foldl' insertTransition lts transitions
  where
    insertTransition :: LTS → (LtsNode, [Rule INT], [LtsNode]) → LTS
    insertTransition l (p, rules, _) = Map.insert p rules l

uniqueProcesses = List.foldl' insertProcess Set.empty processes
  where
    insertProcess :: Set LtsNode → LtsNode → Set LtsNode
    insertProcess s p = if Map.member p newLts
      then s
      else p `Set.insert` s
    processes = concatMap (λ(_,_,r) → r) transitions
```

[1] More precisely, `Rule INT` is the transition plus the corresponding proof tree.

`processNext` **lines 1 to 3** is a helper function which computes the transitions of one process. It returns the original process, the list of full proof trees, and the list containing just the new processes.

`transitions` **line 5** collects all transitions of the current wave. It calls `processNext` for all new nodes.

`newLts` **lines 7 to 10** The new LTS is computed by adding all `transitions` to the current LTS with a `foldl'`.[2]

`uniqueProcesses` **line 12-18** is the set of new processes that have been discovered in this wave and that have to be explored in the next wave. It is a set, i.e. it contains no duplicates and before I add a process to the set, I check if it is an old process (line 15). `uniqueProcesses` can also be computed with a simple `foldl'`.

Annotations for Parallelism

`transitions` is a Haskell list which is lazy by default. In other words, the default strategy of Haskell is to compute the list lazily (on demand) one element after the other.

To exploit parallelism the list elements should be computed in parallel on all available cores. Also the list elements should not be computed on demand one after the other; instead the complete list should be computed right away.

To achieve this I change the definition of `transitions` to:

```
!transitions = parRules $ map processNext w

parRules ::
          [(LtsNode, [Rule INT], [LtsNode ])]
       → [(LtsNode, [Rule INT], [LtsNode ])]
parRules = withStrategy $
  parList $ evalTuple3 r0 (parList rwhnf) (parList rwhnf)
```

Semantically, `parRules` is equivalent to the identity function. It takes a value and returns exactly that value. The purpose of `parRules` is not to define a function which computes a new value, it is to tell the runtime system to compute its argument with a different *reduction strategy.* The default reduction strategy of Haskell corresponds to the *normal order reduction* of the Lambda calculus. I replace it with another strategy which exploits possible parallelism.

The parallel strategy is defined as:

```
parList $ evalTuple3 r0 (parList rwhnf) (parList rwhnf)
```

The structure of the strategy follows the type of `transitions`.

```
transitions :: [(LtsNode, [Rule INT], [LtsNode])]
```

`transitions` is a list of 3-tuples where two of the 3-tuple elements are themselves lists. The strategy says that the outer list is computed in parallel, the 3-tuples are computed with the strategies `r0` and `(parList rwhnf)`, which means that inner lists are again computed in parallel. `withStrategy` is the function which

[2]`foldl'` is the strict version of `foldl`. However, this is misleading because `Map.insert` is still lazy.

applies a strategy to a value. `parList`, `seqTriple`, `r0` and `rwhnf` are defined in module `Control.Parallel.Strategies` in package `parallel`.

Local definitions inside a `where` clause are by default also evaluated lazily in Haskell. This means that by default, the evaluation of `transitions` would be delayed even before the Haskell runtime system reaches the strategy annotation. To avoid this, I add an additional strictness annotation (the exclamation mark `!`) to the definition of `transitions`. This strictness annotation ensures that the evaluation of `transitions` starts right when the right-hand-side of the corresponding clause of the `wave` function gets evaluated. I also add strictness annotations to the other local definitions of `wave`.

This approach is called semi-explicit parallelism in Haskell. One advantage of semi-explicit parallelism is that it is a light-weight approach. For example, I do not have to deal with any synchronization, locks, etc. Another advantage is that it does not add any non-determinism. Annotating an expression with a strategy is guaranteed to not change the value of the expression. In particular the parallel version of `transitions` will return the list elements in the same order as the non-parallel version.

Using threads, locks, message passing or similar techniques often introduces non-determinism because the result of a computation can become dependent on the exact timing of the events. Non-deterministic computations are difficult to debug. On the other hand, adding strategy annotations to a program is relatively safe. In the worst case the parallel version of the program might take longer than the original or have some memory leaks. It is guaranteed that program will *not* compute a false result because of a strategy annotation. For a critique of semi-explicit parallelism see [56] and Section 7.3.

7.2 Parallel Benchmarks

I start with some remarks before showing diagrams of the measured running times.

Objectiveness

The CSP_M specifications, which are the input of my model checker, are in essence programs themselves. Section 5.5 contains examples that small changes in the specifications can cause big differences in the measured running times.[3] This problem is not specific to my program but it is in general more difficult to measure the *average* performance of a symbolic computation like an interpreter or a model checker than the performance of an algorithm like QuickSort, which works on bulk data. It is difficult for the single core performance but, even more so, for the multi-core performance.

The purpose of the benchmarks is to show that using multi-core parallel Haskell for model checking is interesting and that speed-ups are possible in reality. I do not claim that the presented model checker is already perfectly parallelized.

[3] "The only statistics you can trust are those you falsified yourself."–Winston Churchill

Selection of the Test Cases

The presented parallel BFS works best for specifications which have a sufficient number of branches in the state space. One cannot expect much speed-up for a state space which consists just of a single thread of states which have to be visited one after the other. The regression tests that I use for my model checker[4] often test just one special feature of CSP$_M$ and often have a degenerated, single threaded-state space.

Other specifications are unsuitable because they only run for a view milli-seconds, which means that the measurements can easily jitter. In the end I settled down to the following specifications: `hanoi.csp`, `crossing.csp` and `scheduler.csp`. I use the exact versions of the specifications that have been published in the literature, so in some sense these specifications are "real-world" specifications.

GHC Version

Multi-core parallelism and Haskell is a very active subject of research and there has been significant progress in this area recently [39]. Therefore, I have used the latest version of the GHC compiler that was available at the time of running the benchmarks (ghc-7.0.0.20101021 a release candidate for ghc-7.1).

Overhead of Multi-threading

GHC allows one to compile and link a program with two different runtime systems: a multi-threaded runtime system and a non-threaded runtime system. Using the the multi-threaded runtime system generally causes some overhead compared to the non-threaded runtime system. Appart from this, there can be an extra and disproportional slow-down when using the multi-threaded Haskell runtime system running on only a single CPU core.

The consequence is that, some experiments show speed-ups greater than n when using n CPU cores and taking the multi-threaded runtime system and one CPU core as reference. For example the hanoi-9 benchmark takes 379 seconds using a single core but only 171 seconds using two cores, which would represent a speed up of 2.3.

Speed-ups greater than the number of CPUs are certainly counter-intuitive. To avoid this I take the non-threaded runtime system as reference for all relative speed-ups. For the hanoi-9 benchmark the non-threaded running-time is approx. 250 seconds which gives a speed-up of approx. 1.46 when using two cores (see Figure 7.2).

Compiler Options and Garbage Collection

I have used the following compiler options for the benchmarks:
`-O2 -funbox-strict-fields`. These options are commonly used to give fast executables precedence over fast compile times. I did not use any runtime options for the garbage collection and I did not set a predetermined heap size. The benchmarks start with a small heap size and dynamically allocate memory as needed.

[4]I reuse the test suite of ProB.

Hardware

The tests were run on the *Intel-Manycore-Testing-Lab*, an initiative of Intel which provides Academia free access to state-of-the-art multi-core hardware. At the time of writing, the Manycore-Lab featured multi-core computers with 32 CPU cores based on 2.27GHz Intel Xeon CPUs. The computers are installed with a 64-bit Linux operating system and have 256 GB RAM. (All benchmarks run with less than 1GB RAM).

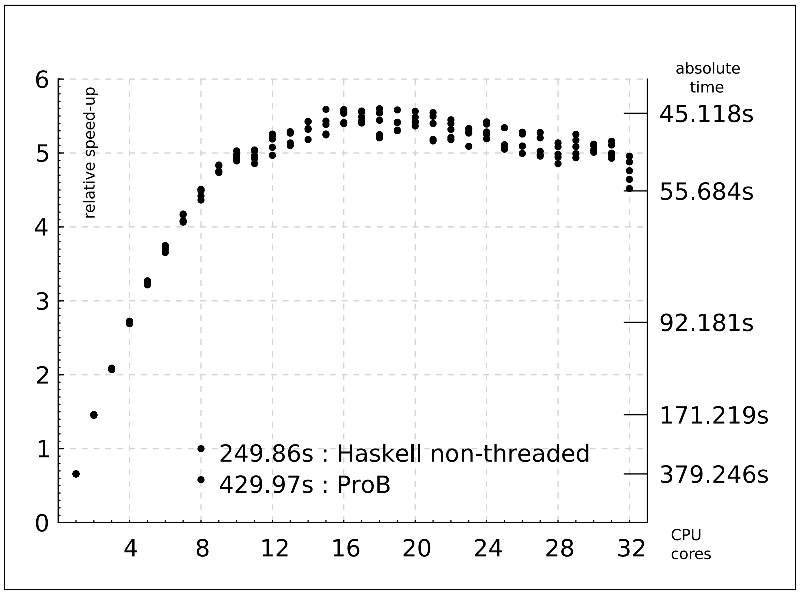


Figure 7.1: Multi-Core Speed-Up: `hanoi.csp` (9 disks)

Benchmarks

For each specification I measured a total of $5*(32+1+1) = 170$ test runs. The multi-threaded Haskell version was tested using between one and 32 CPU cores and for reference I also tested the single-threaded Haskell version and ProB (Version 1.3.2-final (5718)). I recorded 5 running times for each configuration. The running times for ProB and Haskell are the times reported by the tool itself and do not include start-up, parsing and preprocessing.

The diagrams (Figure 7.2, Figure 7.1, Figure 7.3, Figure 7.4) show the speed-ups, i.e. the reciprocal running times normalized to the running time of the single-threaded Haskell version. For graphical reasons the non-threaded Haskell times and the ProB times are marked at 8 cores but these times are independent of the number of cores. On the right y-axis the absolute running times for 1,2,4,8 and 16 cores are marked.

7.2.1 Interpretation of the Results

The diagrams are roughly consistent with Amdahl's law [1].

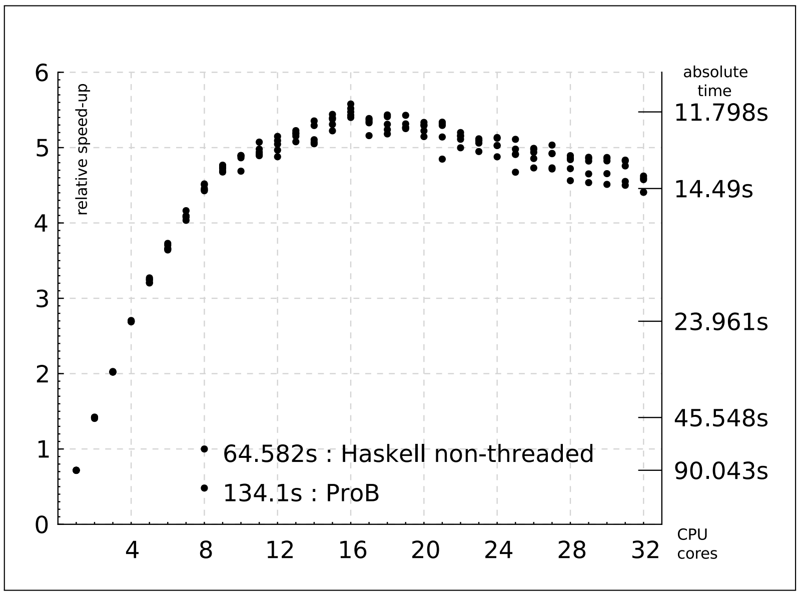


Figure 7.2: Multi-Core Speed-Up: `hanoi.csp` (8 disks)

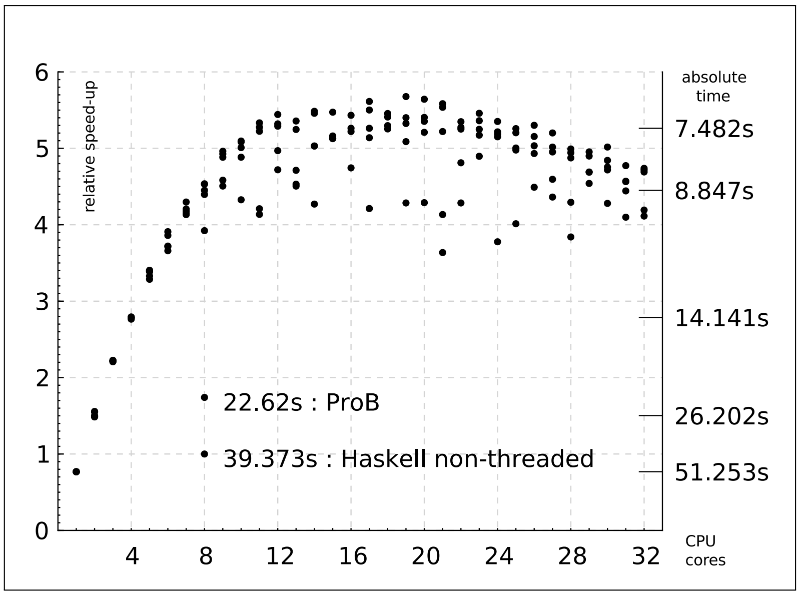


Figure 7.3: Multi-Core Speed-Up: `crossing.csp`

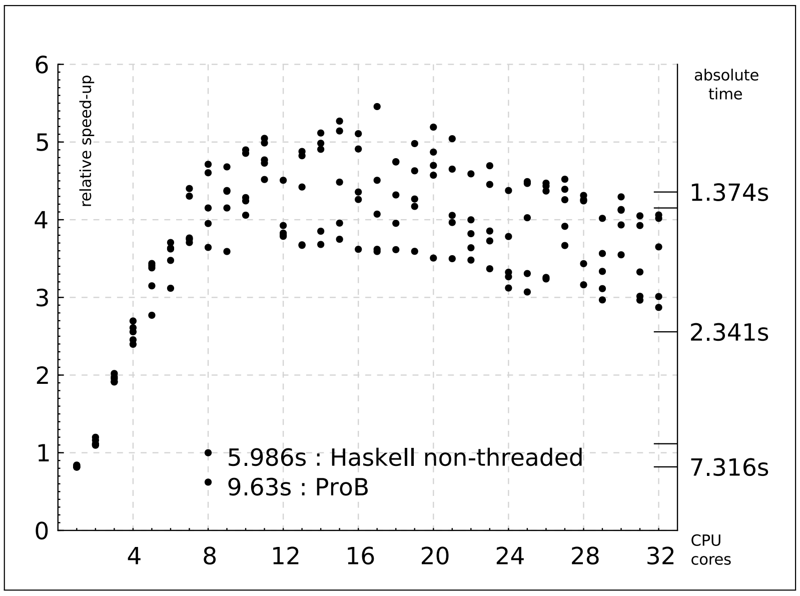


Figure 7.4: Multi-Core Speed-Up: `scheduler.csp`

> Amdahl's law states that if P is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - P)$ is the proportion that cannot be parallelized (remains serial), then the maximum speed-up that can be achieved by using N processors is:[5]

$$\frac{1}{1 - P + \frac{P}{N}}$$

The portion P of the program that can be parallelized with the presented approach and for the presented test cases is approximately 85% (speed-up=5.5, 16 CPU cores). The approach works well for typical, up-to-date PCs which have 8 to 12 CPU cores, but it does not scale beyond 16 cores.

There is an interesting difference between the diagrams. The diagrams for the Hanoi benchmarks show much less variation in the running time for higher numbers of processor cores than the diagrams for `scheduler.csp` and `crossing.csp`. One possible explanation for the difference is that the state space of the the Hanoi specification has a regular shape. This may be the reason why actual computations may also process more regularly.

Semi-explicit parallelism only guaranties that the result of a computation is deterministic. The exact scheduling of which CPU core computes which state of the specification is, in general, still non-deterministic and some of the schedules may be better that others. The *threadscope* tool `http://research.microsoft.com/en-us/projects/threadscope/` can be used to investigate this further—which is left as possible future work.

[5]Quote from Wikipedia: Amdahl's law

7.3 Critique on Semi-explicit Parallelism

The big advantage of semi-explicit parallelism is that it is completely safe to use. Adding strategy annotations for parallelism is guaranteed to not change the result of a function. On the other hand, the author must admit that tweaking the source code for maximal parallel performance can resemble black magic.

In the paper "A monad for deterministic parallelism", Simon Marlow, Ryan Newton and Simon Peyton Jones summarize the problems with semi-explicit parallelism as follows [56]:

> For many years we have advocated the use of the par and pseq operations as the basis for general-purpose deterministic parallelism in Haskell (...). However, a combination of practical experience and investigation has lead us to conclude that this approach is not without drawbacks. In a nutshell, the problem is this: achieving parallelism with par requires that the programmer understand operational properties of the language that are at best implementation-defined (and at worst undefined). This makes par difficult to use, and pitfalls abound — new users have a high failure rate unless they restrict themselves to the pre-defined abstractions provided by the Strategies library.

This quote makes a distinction between the low-level primitives `par` and `pseq` and the pre-defined higher level strategies that I have used, but in general, the critique that these techniques require understanding of operational details also applies to the higher level strategies.

In our experiment, semi-explicit parallelism turned out to work well and the presented benchmarks give a first estimate for speed-ups that can be achieved. The presented source code is a declarative description of what is computed and the pseudo code in Section 7.1 gives an intuition of how the implementation works. The function shown in Section 7.1 was written in a declarative style *concerning the result* of the function. To thoroughly analyse the parallel behaviour of the implementation, one has to argue about the *operational properties of the language*. This is difficult and beyond the scope of this thesis. The new approach, proposed by Simon Marlow et al [56], is to write parallel functions in a monadic style which makes it easier to reason about the operational properties.

7.4 Conclusion

This chapter presents an experiment that was carried out in the course of writing this Phd thesis. Given the little effort, the achieved speed-ups are reasonable. For selected benchmarks I was able to speed-up the computation of the state space by a factor of up to 5.5. The benchmarks in this chapter provide a baseline for the speed-ups that are possible with semi-explicit parallelism.

Theoretically, pure functional programming languages work well together with parallelization. The small experiment in this section shows that these theoretical advantages also hold true in practice—to some extent. The next reasonable step is to try other abstractions for parallelism in Haskell—for example the *new* approach[56].

Chapter 8

Integrated Tool

The previous chapters describe several building blocks of my CSP$_M$ project, namely the parser, the interpreter for the functional sub-language and the implementation of the firing rules semantics of CSP$_M$. This chapter presents an integrated CSP$_M$ tool which combines these building blocks. I use the term `cspm` for this integrated CSP$_M$ tool and also for the software of my CSP$_M$ project in general.

This chapter is organized as follows:

Section 8.1 describes the command line interface for `cspm`.

Section 8.2 contains installation instructions for `cspm`.

Section 8.3 explains an approach for back-box testing CSP$_M$ tools.

Section 8.4 lists the limitations of `cspm` and incompatibilities between `cspm` and the FDR.

Section 8.5 compares `cspm` with ProB and FDR.

Section 8.6 lists some related software and CSP projects.

8.1 Command Line Tool

The `cspm` command-line executable demonstrates some features of my project. It can be used to quickly check if the CSP$_M$ libraries can be installed and run on a system and if a specification is compatible with the libraries. `cspm` also demonstrates how to glue the `cspm` libraries together and it can be used as a template for building custom CSP$_M$ tools.

The `cspm` executable supports several commands, for example:

eval Evaluate an expression. The filename of an additional CSP$_M$ specification can be set with an option, in which case the expression is evaluated in the context of that specification.

trace Interactively trace a process. By default `trace` searches for a process that is declared as `MAIN=`... and uses this process as the initial process. Another initial process can be set with command line options.

lts Compute the LTS of a process with the breadth-first search described in Chapter 7 or with a simple DFS. Possible output formats are, for example, DOT graph and simplified CSP$_M$ which is suitable for refinement checking with FDR. To deal with large or infinite labeled transition systems, a timeout can be set and partial LTSs can be computed.

translate Translate a CSP$_M$ specification into various formats, for example the Prolog encoding expected by ProB.

"`cspm -help`" lists all implemented commands and "`cspm` *command* `-help`" prints a help message for one command, for example "`cspm eval -help`" for the `eval` command.

Figure 8.1 shows examples for the usage of `cspm`. The examples assume that the current directory contains the file called "`funBench.csp`" with the specification from Appendix C.1.

```
/home/fontaine $ cspm eval "4+3"
(VInt 7)
/home/fontaine $ cspm eval --src=funBench.csp "square(7)"
(VInt 49)
/home/fontaine $ cspm trace --main=P2 funBench.csp

Process :
Prefix (PrefixState HashMD5_D3191995AC4F3A0D077CE778CEC2CDFB)
0 :
out.4
Select a Transition
0

Process :
Prefix (PrefixState HashMD5_7838F0F30CD46AD29F5391D12EB45102)
0 :
out.9
Select a Transition
0

Process :
Stop
deadlock state

/home/fontaine $ cspm lts --dotOut=t.dot --main=P2 funBench.csp
/home/fontaine $ cspm --help
```

Figure 8.1: Using the `cspm`-command line tool.

8.2 Installation of the Tools

The preferred way of installing the tools is via the *Haskell Platform*. The Haskell Platform [11] is the canonical Haskell distribution and provides the Glasgow Haskell Compiler (GHC), a set of standard libraries and the tool chain (including the `cabal`-command) that is required for the installation. The Haskell Platform is available for Windows, Mac and Linux. An easy installation of the `cspm` tools is important for the reproducibility of the presented benchmarks and tests.

I use the cabal system for packaging and building the `cspm` tools and the `cspm` libraries. On a current version of the Haskell Platform (`2011.2.0.1`), my `cspm`[1] libraries can be installed with the command:

```
cabal install CSPM-cspm
```

The `cabal` command takes care of downloading and installing the package and the required dependencies from a server called *Hackage*. Hackage is a central repository for cabal packages that is maintained by the Haskell community. The complete source code of the `cspm` libraries and the `CSPM-cspm` package is available on Hackage.

A nice feature of the Hackage server is that it automatically runs a test build when a new package is uploaded and, in particular, it also builds the haddock documentation of libraries. Haddock is a Haskell documentation tool. The haddock documentation of the libraries is online browsable on the Hackage web page and the documentation itself contains back-links to the source code.

The cabal tool is also responsible for managing the dependencies of the `CSPM-cspm` package and the `cspm` libraries. The direct and indirect dependencies of `CSPM-cspm`, which comprise dozens of packages, are unmanageable without tool support.

In general, cabal resolves and installs the dependencies of a package fully automatically. The `CSPM-cspm` package specifies precise constraints for the versions of the direct dependencies. The dependency resolution may fail, however, if a dependency does not comply with the versioning policy of cabal, or if a direct dependency has itself a dependency which fails for some reason. Usually, this problem does not occur if the installation starts with a well-defined set of packages as provided by the Haskell Platform. I will maintain `CSPM-cspm` compatible with future versions of the Haskell Platform for some time.

8.3 Black-Box Testing

Section 4.4 describes an approach for testing the firing-rules semantics that has been implemented in the core-language module. However, these tests have two restrictions: first they only cover the core-language module and not the implementation of the functional sub-language of CSP$_M$. And second, they only test the internal consistency of one part of the core-language module with respect to another part.

Of course, it also makes sense to systematically perform black-box tests that systematically test the integrated CSP$_M$ tool. An interesting approach is to check the correctness of one CSP$_M$ tool A with respect to a second tool B.

The following method can be used to test if two tools A and B implement the same semantics :

Step 1 Start with a CSP specification of a process $P1$. Use tool A to compute the labeled transition system of $P1$

[1]The relevant version of the package is `CSPM-cspm-0.5.6.0`.

Step 2 Translate the LTS back to a CSP specification with entry point $P2$ and use tool B to check that $P1$ is a failures-divergences refinement of $P2$ and that $P2$ is a failures-divergences refinement of $P1$.

When Step 2 of the test fails, it means that tool A and tool B do not agree on the semantics of the given specification. I have applied this approach with my implementation as tool A and FDR as tool B. For a good test coverage one should test with a variety of specifications which cover all interesting parts of the CSP$_M$ semantics.

Translating an LTS to CSP syntax

The translation of an LTS to a CSP specification is complicated by the fact that there is no direct method in CSP$_M$ to specify τ transitions. I use the same trick that is also used for testing ProB [34]. Suppose the LTS contains a state S with the following set of transitions:

$$\{S \xrightarrow{e_1} T_1, S \xrightarrow{e_2} T_2, S \xrightarrow{\checkmark} \Omega, S \xrightarrow{\tau} U_1, S \xrightarrow{\tau} U_2\}$$

The state S is translated to the following CSP$_M$ declaration:

$$S = (e_1 \to T_1 \;\Box\; e_2 \to T2 \;\Box\; \mathit{SKIP}) \rhd (U_1 \sqcap U_2)$$

If S has an empty set of transitions, it gets translated to $S = \mathit{STOP}$.

In other words, transitions that involve a regular event are translated to prefix operations and $\checkmark$-transitions are translated to *SKIP*. Those transitions are combined using $\Box$ (external choice). All τ-transitions are combined using $\sqcap$ (internal choice). The non-τ parts become the left-hand-side of a timeout operation ($\rhd$) and the τ-parts the right-hand-side. If either the τ-part or the non-τ-part is empty, the other part is used directly, with no need for a timeout operation.

I have used the presented method to verify the compatibility of FDR and my CSP$_M$ tool with about 100 test cases from the ProB test suite. Apart from the limitations listed in Section 8.4. the Haskell CSP$_M$ interpreter is almost 100% compatible with FDR.

In practice it turned out that just running Step 1 of the presented method for a large number of specifications was already very helpful for debugging my implementation. Most of the detected problems were related to the functional sub-language. For example, I encountered exception from the interpreter of the functional sub-language, because some features of CSP$_M$ had not been implemented. When Step 1 successfully generated an LTS it usually also passed Step 2. This shows that the separate tests for the CSP core-language (Section 4.4) are effective. The black-box tests did detect some discrepancies between my tool and FDR with respect to *dot-tuples* (Section 8.4.2).

The same approach for black-box testing is also used by Michael Leuschel for ProB. In one case, the routine tests that are run for ProB were able to detect a bug in FDR. The bug was discovered when one test case started to fail after FDR was updated to a new release.

8.4 Know Limitations

This section lists the known limitations and missing features of the current version of cspm. Some of the missing features are non-trivial and the effort for adding these features is hard to estimate. In general, these missing features have been postponed in favour of other possible extensions that are listed in the chapter on future work.

8.4.1 Recursive Data Types

Syntactically, it is possible to define recursive data types in CSP$_M$ like:

```
datatype Nat = Zero | Succ.Nat
channel c:Nat
MAIN = c?x → MAIN
```

The data type Nat contains an infinite number of values, which means that there are some restrictions on how this data type can be used, depending on the tool. For example, ProB can deal with such data types, but it will only enumerate a finite number of alternatives when it searches for a value of such a data type.

FDR has the limitation that it will never search for a value of an infinite recursive data type. If a specification contains non-determinism with respect to a value from a recursive data type, FDR will throw a run-time exception. This rules out most application of such data types in FDR.

Recursive data types are not supported by my tool. If they are used my, tool will most likely compute false results. We have not yet investigated how to fix this.

A related issue is the FDR type constructor Seq. Seq(l) is equivalent to the set of all sequences over the alphabet l, i.e. $l^\star$. The restrictions for the application of Seq are similar to those for recursive data types.

Workaround

Since all known CSP$_M$ implementations[2] use dynamic typing, there is no need for complex data type declarations at all. The implementations make no strict distinction between data types and sets of values. For example, data types can occur in an expression and a set of values can occur in a channel declaration.

```
channel c:NatSet(5)       -- a channel declaration can use a set
MAIN = c?x → MAIN

datatype C1 = Zero | Succ
NatSet(0) = {Zero}        -- a regular set replaces a data type
NatSet(n) = union ( {Zero} , { (Succ,x) | x← NatSet (n-1)})

datatype C2 = Leaf | Node
BinTree(0) = {Leaf}
BinTree(n) = let ts = BinTree(n-1) within
  union ( {Leaf}, {(Node,c1,val,c2) | c1 ← ts, val ← {0,1}, c2 ← ts})

channel c2:BinTree(2)
```

[2]i.e. FDR, ProB, ProBE and cspm.

```
SomeSet=union(Const,{1,2,3})   -- a set expression can use a data type
```

Therefore, it is possible to replace a data type declaration with a regular set. It is sufficient to declare new constants like `Zero` and `Succ`. Instead of `Succ.Succ.Zero` one can write `(Succ,(Succ,Zero))`. This avoids the use of dot-tuples, which have an non-trivial semantics (c.f. Section 8.4.2). This workaround is compatible with FDR, ProB, ProBE and cspm but it relies on dynamic typing.

The latest version of FDR (FDR-2.91 from 2010) has introduced a new `Proc` data type. The presented workaround can also be used to replace a data type that contain `Proc` with a set of untyped tuples.

8.4.2 Dot Tuples

CSP$_M$ has two alternative syntactic variants for writing tuples. A tuple can be written as `(a,b,c,d)` or it can also be written as `a.b.c.d` . The second variant is called a *dot-tuple* in this thesis.

Dot-tuples have several applications in CSP$_M$:

1. All events are dot-tuples of the form $channel.d_1.d_2.\cdots.d_n$.
2. Data types are constructed with the dot-notation, for example `Pair.1.2` .
3. Arbitrary values can be joined with a dot to form a dot-tuple.

The available documentation for dot-tuples in FDR is sparse. The semantics of dot-tuples in CSP$_M$ is basically defined by the internal implementation of the CSP$_M$ tools. This section presents my understanding of FDR. My main approach to investigate how dot-tuples are implemented in FDR was to try a number of test cases. Each test case constructs a dot-tuple value and then deconstructs it again with pattern matching.

For example I use the function :

```
f(a.b.c)=(a,b,c)
```

The function `f` deconstructs a dot-tuple and translates it to regular, comma-separated tuple. It is possible to evaluate `f(1.2)` with the FDR interpreter using the following command line:

```
echo "_evaluate(f(1.2))" | $FDRHOME/bin/state2 Spec.csp
```

This by-passes the GUI and is very handy for this kind of experiment. I have used `fdr-2.91rc4-academic-linux` for the presented experiments.

Table 8.2 contains examples that show the behaviour of dot-tuples. It shows a table of values for the functions `f`, `f2` and `f3`. The experiments lead to the following conclusions:

- Dot-tuples are flat. It it not possible to construct nested dot-tuples (examples (a), (b)).
- A dot-pattern can match a dot-tuple of a different size (examples (c), (d)).
- If the value has more elements than the pattern, then the last element of the pattern matches the rest of the value (example (c)).

f(a.b.c) = (a,b,c)		
x	f(x)	
1.2.3	(1, 2, 3)	
1.(2.3)	(1, 2, 3)	(a)
(1.2).3	(1, 2, 3)	(b)
1.2.3.4	(1, 2, 3.4)	(c)
1.2	(1, 2, ϵ)	(d)
1	Bad dot pattern match	

f2(a.b.c.d) = (a,b,c,d)		
x	f2(x)	
1	Bad dot pattern match	
1.2	Bad dot pattern match	
1.2.3	(1, 2, 3, ϵ)	

f3(a.b) = (a,b)		
x	f3(x)	
1	(1, ϵ)	

Figure 8.2: Behaviour of dot-tuples in FDR

- The pattern can have one element more than the value. In this case the last element of the pattern is bound to a mysterious, invisible value. Table 8.2 shows this value as ϵ (example (d)). The true output of FDR in experiment (d) is `(1,2,)` , i.e. FDR completely omits ϵ when it pretty-prints a value.

In my point of view, the possibility to match a dot-pattern with a dot-value with a different number of elements is a doubtful feature of FDR. I think that this feature adds only a little expressiveness with the cost of an extra corner case.

The implementation of dot-tuples in my tool is not 100% compatible with FDR. For example, my tool throws an exception if the value has less elements than the pattern. Overall, I think that a tool should reject specifications that rely on corner cases of FDR. On the other hand, unfortunately, my tool does not always follow that policy. For example when a dot-value has more elements than the pattern, my tool just discards the extra elements.

A plan for future work is to add a configuration option to my tool which lets the user select between a *strict-mode* (which implements a reasonable semantics without corner cases) and a *compatibility* mode (which tries to be as compatible with FDR as possible).

8.4.3 Slow Link Parallel and Renaming Operations

One of our regression tests [3] contains the following code:

```
channel left,right:{1..20}
channel left’,right’:({1..20},{1..10})
ITER = left’?(d,x) -> right’!(d,f(d,x)) -> ITER
f(d,x) = (x + d/x)/2
```

[3]The test is `roscoe_chapter4.csp`, a specification that comes with the first Roscoe book [52].

```
INIT = left?x -> right'!(x,(x+1)/2) -> INIT
FINAL = left'?(d,x) -> right!x -> FINAL
ROOTER = INIT [right' <-> left']
        (ITER [right' <-> left']
        (ITER [right' <-> left']
         FINAL))
```

The process `ROOTER` is defined with a three-fold nested link-parallel operation. The definition of the linking relation is `right' <-> left'` and the channels `left'` and `right'` each consist of 200 possible events.

`cspm` implements the link-parallel operation as a loop which iterates over the domain of the linking relation. This implementation is inefficient. In the example, the link-parallel operation is threefoldly nested and the domain of the linking relation contains 200 elements. This means that the inner part of the expression (the process `FINAL`) has to be evaluated 200^3 times. Consequently the running times for this example are high. The sequential running time is approximately 40 seconds and the parallel running time is still 10 seconds (c.f. Section 8.5.4).

An obvious way to improve the implementation is to use memorization and to compute the sub-processes of a link-parallel operation only once. This corresponds to the bottom-up approach of FDR. Another idea is to use the fact that the linking-relation is often a function and not a general relation. This is also the case in the presented example.

8.5 Comparing CSP$_M$ Tools

Currently, there are four tools available which use CSP$_M$ as input syntax: FDR, ProBE, ProB and the tool and libraries described in this thesis. This section compares some aspects of these four tools.

FDR is a refinement checker and ProBE is a process tracer. ProB is a model checker which supports full LTL checking and the latest versions of ProB also supports refinement checks. In the current version, `cspm` can only check for deadlock states and safety properties. Both ProB and `cspm` can be used to compute the LTS of a specification.

Strictly speaking, a model checker, a refinement checker and a process tracer represent different functionality. This section will not discuss the difference between model checking and refinement checking. Instead, I will focus on aspects that are directly related to CSP$_M$ and that all tools have in common. All tools are based on the operational semantics of CSP$_M$. In other words, a core part of each tool is the computation of the transition relation of a CSP$_M$ process.

8.5.1 Comparison by Aspects

This section addresses the following aspects:

- Input syntax
- Renaming
- Type checking
- Functional sub-language of CSP$_M$

- Top-down vs. bottom-up
- Firing rules semantics
- Implementation language
- Code metrics

The rest of the thesis contains a detailed discussion of these aspects for `cspm`. A more detailed comparison with the other CSP$_M$ tools would be interesting, but this was beyond the scope of this thesis and I will only skim through the list very briefly.

Input Syntax

For parsing, FDR and ProBE use a LR grammar and the `bison` parser generator. ProB and `cspm` both share the same front-end based on a `parsec` combinator parser (c.f. Chapter 6). The compatibility between the two alternative front-ends is high but they may differ in corner cases. From a user's point of view the `bison` parser and the `parsec` parser work equally well.

Renaming

CSP$_M$ is a statically scoped language and the `cspm` front-end uses a static free-names analysis and a dedicated renaming phase. The renaming phase of the `cspm` front-end can statically catch errors which will pass unnoticed in FDR, for example unbound variables or patterns which contain multiple occurrences of the same variable.

Presumably, FDR and ProBE lack a renaming phase. Presumably, they instead implements the scope of variables dynamically and do not check for illegal rebinding of an identifier. For example, FDR will not detect an illegal definition like `f(x,x)=x*x`; instead it will evaluate `f(4,3)` to `9`. It looks like FDR just processes the pattern `(x,x)` from left to right and that in `f(4,3)` the binding `x:=3` just overwrites the binding `x:=4`. This is at least counter-intuitive.

Type Checking

None of the tools has a built-in static type checker. There are several examples that FDR indeed has features which imply dynamic typing, for example generic buffers [34]. ProB and `cspm` try to be FDR-compatible concerning dynamic typing, however these features remain a source of incompatibilities.

A type checker for CSP$_M$ is available from FSE as as separate program [17] but it has its own limitations. To summarize, the lack of a proper static type checker is a severe problem for all tools and for CSP$_M$ in general.

Functional Sub-language

As expected, Haskell is well-suited for writing interpreters for a functional programming language and `cspm` shows the overall best performance in this aspect (see benchmarks Section 5.5). ProB, which uses the non-ground representation, lambda lifting and precompilation techniques, falls behind in the benchmarks and also misses features like, for example, efficient support for `let`-expressions (c.f. Section 5.2.7, Section 5.3.2). FDR seems to be overall robust but notably slower than `cspm`.

Top-down vs Bottom-up

FDR uses a bottom-up approach for the CSP core semantics. In case of a parallel composition of two sub-processes, FDR first computes the state spaces of the sub-processes and then the parallel composition of these state spaces. The FDR approach does not terminate if one of the sub-processes is infinite, even if the combined state space of the sub-processes is again finite ([32] contains an example).

From a user's point of view, this means that many perfectly correct CSP specifications do not work with FDR. ProB and `cspm` use a top-down approach and can handle specifications which do not work with FDR.

Firing Rules

`cspm` uses explicit proof trees and a separate proof tree verifier. The proof tree verifier is a direct and concise translation of the CSP firing rules to Haskell and the correctness of the CSP core semantics in `cspm` relies only on the correctness of the proof tree verifier (and automated test case generation, see Section 4.4). The implemented firing rules of `cspm` are clearly listed in Appendix A.

ProB uses the built-in search of Prolog to implement the firing rules semantics. A firing rule roughly corresponds to a Prolog clause and a proof tree is only stored implicitly (as part of the SDL tree that is traversed during the execution). ProB computes some explicit information about the proof tree which is used to show visual feedback about a enabled transition in the GUI.

I have no information whether FDR or ProBE use any systematic translation from the firing rules to C or C++ code.

Implementation Language

FDR and ProBE are closed source; it is known that they use C/C++ and Tcl/Tk for the GUI. A version of the FDR parser is available as open source, but it it unclear whether this it the most recent version of the parser.

The core of ProB is implemented in SICStus Prolog. The integrated ProB-tool also consists of parts that are written in Tck/Tk, Java, C/C++ and Haskell. The source code of ProB is available from the ProB web page under the ProB-License.

`cspm` is written in Haskell (Haskell-2010 plus some GHC extensions). The Haskell `CSPM`-libraries and tools are distributed as cabal source code packages via Hackage (the central repository for Haskell packages). The Haskell `CSPM`-tools are under a BSD license.

Code Metrics

Table 8.3 shows a breakup of the code size of the Haskell packages and the CSP part of ProB.[4]

[4] The data was generated the from source code repositories of the tools on Nov 29 2010.

The table lists the lines of code in the package and the total size of the package in kilobytes. Additionally, it lists the biggest single module of the package, the LOC and the size of that module.

	Total Package			Biggest Module	
Package Name	*modules*	*size (LOC)*	*size (kB)*	*size (LOC)*	*size (kB)*
`CSPM-Frontend`	17	3600	110	1100	30
`CSPM-CoreLanguage`	4	300	10	106	3
`CSPM-FiringRules`	13	2400	82	690	22
`CSPM-Interpreter`	15	2600	87	700	23
`CSPM-cspm`	6	670	16	215	6
`cspm` total w/o front end	38	5970	195	700	23
ProB-`CSP`	9	4200	185	1800	82
GUI Tool	20	1800	50	287	8

Figure 8.3: Code metrics for the Haskell packages

The `CSPM-Frontend` package contains the parser for CSP$_M$ which is shared between ProB and `cspm`. Measured in lines of code, the front-end makes up more than a third of the complete tool. The biggest module of the front-end is `Parser.hs` which contains the actual combinator parser. This single module has 1100 LOC and 30kB.

The CSP part of ProB consists of 9 files and approx. 185kB of Prolog code. The biggest Prolog source files are `haskell_csp.pl` (approx. 1800 LOC, 82kB) and `haskell_csp_analyzer.pl` (approx. 900 LOC, 41kB). This does not cover the GUI and glue code, which is shared with the rest of ProB.

`cspm` is almost exactly the same size as the Prolog code in kilobytes. However, the Haskell implementation is split into much smaller modules and it also has more lines of code. The largest Prolog module is more than 3 times bigger than the largest Haskell module (82kB vs 23kB).

Of course looking at the LOC is only a very simplistic approach to comparing Prolog and Haskell code. The tools have different features and the tools do not implement exactly the same functionality. Also, Prolog and Haskell are fundamentally different programming languages. Nevertheless the code metrics are interesting.

Haskell is a statically typed language. A considerable part of the Haskell source code consists of type signatures and declarations for algebraic data types. These type signatures and declarations serve as a compiler checked documentation of the program. In Prolog, simple terms replace the ADTs of Haskell, for example when representing an abstract syntax tree. The AST does not have an explicit declaration in Prolog.

My personal opinion is that Prolog looks more verbose than Haskell. In particular, functional programming in Prolog seems cumbersome to me. For example, nested functions calls, which can be written in Haskell like `f(g x,h y)`, are very verbose in Prolog. Other examples are the lack of higher order combinators like `map`, the lack of `case` expressions, the lack of local declarations, etc.

A thorough comparison of Prolog and Haskell would be interesting, but it is beyond the scope of this thesis. It would also be interesting to look at the source code of FDR and ProBE, but unfortunately it is not open source.

8.5.2 Advertised Features of the Tools

The different CSP$_M$ tools are developed by different teams and each team has different priorities for their tool. This section lists what, in my understanding, are the most advertised features of the tools.

Advertised FDR Features

FDR implements some special heuristics for refinement checking. If a specification is written in a special style and if it is tuned for FDR, FDR can handle large state spaces with billions of states.

Advertised ProB Features

ProB supports many formalisms (for example B, Z, CSP,..), in particular it also supports CSP||B, a special combination of CSP with the B method. ProB supports animation, LTL model checking and several other analyses.

It is based on logic programming (SICStus Prolog) and constraint solving. ProB can deal with channels with large sets of events like `channel c:{1..10000}`, which other tools cannot handle. Also, it uses the top-down approach, which has better termination properties than FDR's bottom-up approach.

Advertised `cspm` Features

The Haskell `CSPM`-tools and libraries are implemented in a purely functional programming language. This helps to improve the correctness and reusability of the code. The Haskell tools use a modular design, explicit proof trees, QuickCheck testing and correct by construction techniques. The Haskell tools were designed to narrow the gap between tool users and tool implementers and they are a good starting point for research on parallel model checking for CSP$_M$.

8.5.3 The CSP$_M$ Tools Seen as a Black Box

A CSP$_M$ tool can be considered a black-box, which takes as input a specification (i.e. a string) and which computes some result. To exactly characterize a tool, one would have to test it with all possible inputs. Unfortunately, this is impossible as there is an infinite number of inputs strings; in principle however, one can roughly distinguish the following cases:

1. The correct result is computed.
2. The correct result is computed but slowly.
3. The implementation rejects a correct specification with an error.
4. The implementation does not terminate, though it should.
5. A plain wrong result is computed.

Case 5: Tool computes a wrong result

Case 5 clearly represents an error. I tried hard to avoid Case 5, but I have no formal proof for the correctness of my program and I cannot rule out errors. The same restriction applies to the other tools.

Case 4: Tool does not terminate

Case 4 is not always easy to distinguish from Case 3, because if a tool goes into an infinite loop it will often run out of memory. Also, the question whether a specification is finite or infinite is similar to the halting-problem and undecidable in general.

Nevertheless, there are concrete examples of specifications where one CSP$_M$ tool terminates and another does not. For example, it can be seen that the bottom-up approach of FDR has principle limitations that are not present in the top-down approach of ProB and `cspm`. However, there are also cases where ProB or `cspm` do not terminate, though they should.

Case 3: Tool throws an exception

This is the preferred result whenever a tool does not implement a feature or if the tool implementer has just forgotten to handle a special case. Haskell has features which help to catch this problem and turn it into a Case 3, as opposed to a Case 5. For example, many functions of the `cspm` tool just consist of one big case distinction over an algebraic data type. The Haskell compiler can often statically check at compile time if a function covers all cases of a ADT. If a Haskell function is called with an argument for which it has not been defined, it will, throw an exception by default.

I have put some effort into generating good error messages. For example, many error messages of `cspm` contain information about a source location of the CSP specification.

Case 2: Tool is slow

Case 2, namely that one tool is much slower than another tool, can also be seen the other way around. It can also be the case that one tool is much faster than the others. There are many possible reasons for this. For example, the bottom-up approach of FDR can be make FDR much faster than the other tools, but it can also be much slower than a top-down approach (depending on the specification).

Constraint solving and special heuristics implemented in ProB can make ProB the fastest tool (depending on the specification). `cspm` has known limitations with renaming and link-parallel operations, but there are also other specifications where it is faster than the other tools.

For the performance of the model checkers, i.e. ProB and `cspm`, it is also important that the equality predicate on the computed states is accurate. If identical states are not recognized by the tool as being identical, it can easily cause an exponential slow-down or non-termination of the tool.

The relative performance of the different CSP$_M$ tools can dramatically vary depending on the specification and even small changes in a specification can make a big difference for the running time. Therefore, there is no clear performance ranking of the tools.

Case 1: Tool works well

In spite of CSP$_M$ being a complex and tricky language, the tools often work as expected. The ProB test suite contains more than 100 test cases for which the CSP$_M$ tools agree.

Conclusion

For an exhaustive comparison of the tools, one would have to list specifications for all the cases listed above. I have not prepared this list, but it surely could be done. Experience shows that the users of the CSP$_M$ tools regularly find problems that the developers have not thought about, and none of the tools is free of errors. At the same time the tools are evolving and the developers fix bugs as they get discovered.

The conclusion is that, from a user's point of view, the different CSP$_M$ tools complement each other. If one tool does not behave as expected, a user can try an alternative tool. For a user of CSP$_M$ it is a big advantage to have several separate tools available.

8.5.4 Benchmarks

Table 8.4 shows some benchmarks for ProB and `cspm`. These benchmarks were run with the same settings and on the same hardware as the parallel benchmarks in Section 7.2.

Specification	*ProB*	`cspm`	`cspm` *parallel (16 Cores)*
`hanoi.csp` (n=9)	430 s	250 s	45 s
`hanoi.csp` (n=8)	134 s	65 s	12 s
`crossing.csp`	23 s	39 s	7.5 s
`scheduler.csp`	9.6 s	6.0 s	1.4 s
`basin_olderog_bank.csp`	9.0 s	0.5 s	no speed-up
`peterson.csp`	500 ms	246 ms	no speed-up
`Peterson_v2.csp`	620 ms	165 ms	no speed-up
`bankv4.csp`	260 ms	172 ms	no speed-up
`roscoe_section2-1.csp`	80 ms	81 ms	no speed-up
`roscoe_chapter4.csp`	80 ms	41 s	10 s
`abp_chapter5_roscoe.csp`	1040 ms	388 ms	no speed-up

Figure 8.4: Benchmarks for `cspm` and ProB

For many benchmarks, `cspm` is between the same speed and three times faster than ProB. For the `hanoi.csp` benchmark, which has an exponentially growing state space, the parallelized version of `cspm` can be up to 10 times faster than ProB. But there are also runaway values, for example `basin_olderog_bank.csp` (`cspm` 18 times faster) and `roscoe_chapter4.csp` (`cspm` 500 times slower).

`roscoe_chapter4.csp` uses the CSP-link-parallel operation. The current implementation of renaming/link-parallel in `cspm` is very inefficient (c.f. Section 8.4). The CSP renaming operation shows that the asymptotic complexity is more important than constant factors. If a tool has a bad asymptotic complexity in one particular case, it is easy to construct a specification where the measured performance will also be bad.

ProB uses constraint logic programming (CLP). The underlying idea of CLP is that a program should be a declarative combination of constraints and that it should be easy to improve a program by just declaring additional constraints. For example, ProB uses integer constraint solving to efficiently compute events for large channels. ProB is the fastest tool for specifications that use integer constraints. Adding constraint-solving techniques to `cspm` is an interesting open question.

CSP tools can scale in several dimensions. They can become faster and compute bigger state spaces—for example solve the Hanoi puzzle for 9 discs instead of 8. But tools can also become more expressive, for example by incorporating constraint solving or by lifting the restrictions of the FDR bottom-up approach.

The specifications from Table 8.4 were taken from the literature and have been written before the existence ProB and `cspm`[5]. Presumably, these specifications have been optimized for FDR, or at least they have been written with FDR in mind. Therefore, these benchmarks only measure the tool performance for one particular flavor of CSP$_M$ specifications.

Conclusion

Benchmarking CSP$_M$ tools is difficult, as tools differ in expressiveness and in the type of specifications that can be handled efficiently. ProB uses techniques such as constraint solving to increase the expressiveness of CSP$_M$. Tools can have different asymptotic complexities for a particular specification and implementations can also contain performance killers.

Overall, the benchmarks show that `cspm` performs reasonably well with specifications that were originally written for FDR. `cspm` is a good starting point for writing a high performance CSP$_M$ tool. From a tool user's point of view, other aspects can be more important than the running times. For example, it is also important that the performance of a tool is predictable.

8.6 Other CSP Software and Related Work

Applications of CSP$_M$

There have been many applications of CSP$_M$ and FDR in research. The current form of CSP$_M$ has been in use for more than a decade now. I only list some recent examples: Moritz Kleine has worked on CSP as a coordination language [29] and for the verification of operating systems [28]. Björn Metzler has worked on compositional verification of CSP$_M$ specifications [41]. Peter Wong has used

[5]except `scheduler.csp`

CSP to give a semantics to the Business Process Modelling Notation [62]. Nick Moffat has worked on symmetry in CSP$_M$ specifications [43].

Libraries Based on Ideas from CSP

There exists a wide variety of software libraries for concurrency which are based on ideas from CSP (according to the library developers). These CSP libraries are meant for writing concurrent programs as opposed to specifications, and therefore these libraries are not directly related to the model checkers which are meant for formal reasoning about the correctness of specifications.

Still I want to list some references for CSP libraries:

- `JCSP` [59] is a framework for taking CSP ideas to Java.
- `CHP` [4] is a Haskell library for CSP.
- `C++CSP2` [5] is a CSP library for `C++`.
- Python-CSP [44] and PyCSP [2] are two approaches for combining CSP with Python.

Other Model Checkers

There are several other model/refinement checkers for CSP. Unfortunately, these do not use the CSP$_M$ syntax and it is difficult to determine how the implemented semantics relates to the semantics of FDR, ProB and `cspm`. For example, the other model checkers do not implement multi-field events and it is unclear how that affects the expressiveness of the tools. Here are some examples of other CSP model checkers:

- The Adelaide Refinement Checker (ARC) [47] uses OBDDs for a compact representation of processes.
- The PAT project [57, 6] is a model checker for CSP extended with hierarchical state.
- JCSProB [63] is a *developing strategy* for translating B/CSP to Java.

CSP and Theory Proving

CSP-Prover [23, 24] uses a formalization of the CSP core semantics for the Isabelle theory prover. It can be used to carry out formal refinement proofs about specifications. It covers the core CSP semantics but does not directly support CSP$_M$ syntax.

Other Process Algebras and Formalisms for Concurrency

CSP is only one example of a wider class of formalisms called process calculi. Another well-known process calculus is the calculus of communicating systems (CCS) [42]. Apart from process calculi, another important formalism for concurrency is Petri nets (c.f. Wikipedia).

Chapter 9

Future Work

There are many possible directions for future work. The future work can roughly be divided in short-term fixes and improvements, medium-term ideas and long-term future projects.

9.1 Short-term

In his new book [51] Roscoe describes an extension to the CSP formalism called the *τ-priority-model.* Informally, the τ-priority-model can be used to reduce unwanted non-determinism and unwanted symmetries in a specification by giving certain transitions priority over others. The non-priority transitions are only enabled after all priority transitions have been executed. The book is accompanied by a new version of FDR and by new CSP$_M$ specification which demonstrate the τ-priority-model and other new features of FDR.

A reasonable short-term improvement is to also implement the τ-priority-model in `cspm` and to support the latest extensions of FDR to remain compatibility.

Another idea is to connect `cspm` and ProB such that the tools can be used to refinement check each other (without the need for FDR). Also, it makes sense in the short term to extend the test-suite and to fix outstanding bugs and performance problems.

9.2 Medium-term

This section lists possible-medium term work on `cspm` and some further ideas. These ideas are not fully elaborated and some of the ideas are overlapping.

Extend `cspm` with an LTL or CTL Model Checker

`cspm` already implements the "model-part" of a model checker, i.e. the computation of the transition relation of a process. To turn `cspm` into a full model checker one would have to extend it with a suitable logic, for example with LTL or CTL, and implement a decision procedure for that logic. LTL and CTL use a notion of *atomic properties.* One would also have to design the atomic properties in the case of CSP$_M$, similarly to how it has been done in ProB [50].

Improve the Parallel Performance

Chapter 7 describes a preliminary experiment on parallel `cspm`. The results of this experiment are encouraging and in the mean time the work on parallel Haskell is making steady progress. Just recently a new framework for parallelism in Haskell has been published [56] and it would be interesting to adapt the experiment from Chapter 7 to that framework. Distributed parallel model checking is another interesting direction to go.

Include an (Approximating) Type Checker for CSP$_M$

The available CSP$_M$ tools implement dynamic typing. On the other hand, it is rare that specification writers intentionally use the dynamic feature of the tools. Furthermore, the FDR developers state that FDR will only work correctly for specifications which pass the FSE type checker and explicitly recommend the use of that external type checker. However, the standalone type checker that is available from FSE also has limitations.

Instead of depending on a external type checker it would be more reasonable to integrate the static type checker directly into the CSP$_M$ tools. Typically, functional programming languages like Haskell use powerful algorithms that perform type checking and also type inference. In the case of CSP$_M$ however, even a limited approximating type checking could be valuable. An approximating CSP$_M$ type checker could return one of the three results:

1. It reports an obvious type error.
2. It reports that a specification is type correct.
3. It reports a warning for specifications that cannot be easily type checked.

A possible type checker could reuse the `CSPM-Frontend` package and it could be easily integrated into `cspm`.

Mix the Bottom-up with a Top-down Approach

One idea is to extend CSP$_M$ with an memorize-annotation which ensures that the LTS of the annotated process is only computed once. The following example shows a possible syntax for this feature:

```
P1 = memorize(some process specification)
MAIN = P1 [] P2
transparent memorize
```

The transparent declaration tells other tools that they should ignore the function call `memorize(..)` and that `memorize` behaves semantically like the identity function. There are two main alternatives for the implementation of `memorize`: a partial LTS for `P1` can be computed on demand or the complete LTS can be computed when the process is used the first time.

In other words, the tool follows a top-down approach overall, but the user can annotate parts of the specification which are precomputed similarly to the bottom-up approach of FDR. An extension of this idea is to investigate whether some of the compression heuristics that are provided by FDR can also be included in `cspm`.

Support Refinement Checking

The `cspm` tool already allows one to compute the transition relation of a process. This means that most functionality for refinement checking has already been implemented. I think that the `cspm` tool can be extended to a refinement checker with only little extra work.

Analyze the Generated Proof Trees

The `cspm` tool generates the proof trees according to the firing rules semantics of CSP, but currently these proof trees are only used as an intermediate data structure, which helps to improve the correctness of the tool. An interesting idea is to analyze the proof trees to gain additional information about a specification.

For example the list of proof trees that correspond to a particular trace can be used to extract information about which parts of the specification have been relevant for this trace. This could be used to implement a slicing operation similar to the CSP$_M$ slicer of ProB [35].

Another idea is to detect symmetries in a specification with the help of proof trees. For example, if the proof trees of a trace of events are "independent" of each other, it could be possible that a permutation to the events of the trace is also a valid trace and that this permutation of events lead to the same process as the initial trace.

Alternative Implementation of the Firing Rules Semantics

The current design is based on several primitive operations on events and event-closure-sets, like equality tests and membership tests. These operations are implemented as functions which get called directly in the algorithms. In the constraint-based approach (c.f. Section 4.3) these functions take partial inputs and compute partial results. An interesting idea is to implement these operations as data structures which can be manipulated symbolically. This would make it possible to use more constraint-solving techniques.

`cspm` uses a modular design which allows one to replace parts of the tool with an alternative implementation. I think it would be interesting to try other possible designs for the implementation of the firing rules semantics. The central idea, which has turned out to work well in the current implementation, is the use of explicit proof trees. I think it is a good idea to also base a new approach on explicit proof trees. A new implementation which uses the same format for proof trees could be directly checked against the old implementations with the QuickCheck as presented in Chapter 4.

Investigate Timed CSP

Timed CSP [54] adds new constructs to CSP which allow one to express and analyze the timed behaviour of processes. To support timed CSP in `cspm` one would have to extend the CSP$_M$ parser, the implementation of the functional sub-language and the implementation of the firing rules semantics in `cspm`. This is an ambitious project. The semantics of timed CSP builds directly on the CSP formalism that is underlying this thesis. In terms of software engineering, extending `cspm` with timed CSP is an interesting case study on the reuseability of the presented software.

9.3 Retiring CSP$_M$

The `cspm` project can be used to explore new ideas and to test them with existing CSP$_M$ specifications. However, CSP$_M$ syntax has some severe limitations (c.f. Chapter 6) and some important parts of the CSP$_M$ semantics are unclear or complicated (c.f. Section 8.4.2, Section 5.4).

Therefore, I think that, in the long-term, it is better to pursue an alternative. Given that CSP$_M$ is really just functional programming plus some small CSP specific extensions, it may make sense to replace CSP$_M$ with a Haskell-based embedded DSL. In other words, one could try to build a CSP library for Haskell with the focus on model checking.

9.4 Case Study: B-method

`cspm` was my Haskell learning project. My personal experience is that Haskell was well-suited for the CSP$_M$ project. In general, many projects use Haskell for symbolic computations like interpreters, theorem provers, compilers, formal methods tools, etc.[1] The comparison between `cspm` and ProB leads to the conclusion that Haskell is an interesting alternative to Prolog.

The ProB tool supports several other formalisms beside CSP. The bigger part of ProB is dedicated to the B-method and the core of ProB is a model finder for the logic that is underlying the B-method. The ProB core is a highly sophisticated piece of software which has been developed over several years. It makes heavy use of constraint logic programming and it is strongly tied to Prolog. A model finder for the B-logic in Haskell could be an interesting case study for a combination of Haskell and constraint programming.

9.5 GUI Tool

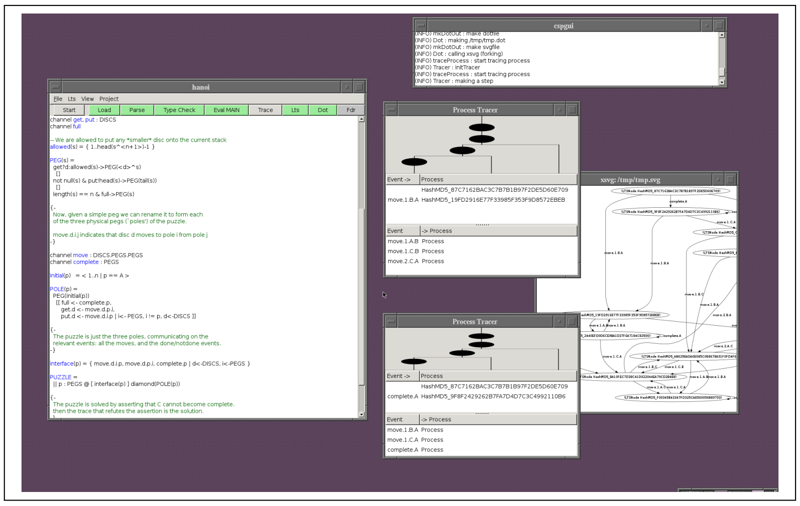
Many users of the formal methods tools highly appreciate a graphical user interface and FDR, ProB and ProBE come with that feature. When working with a CSP$_M$ specification, several kinds of visualization can be useful. For example:

- Syntax highlighting and a syntax aware editor for specifications.
- Visualization of state spaces.
- A GUI for tracing processes.
- A visual representation of the structure of a process.
- Mapping events back to source locations in the specification.
- Visualization of the environment of a process.
- Visualization of inference trees.

[1] The cabal packages listed on `http://hackage.haskell.org` give an overview.

I have implemented a prototype of an integrated graphical CSP$_M$ tool as a proof of concept. The GUI is based on GTK (the Gimp Tool Kit). I will not describe the GUI-tool in detail; instead, I show a screenshot (Figure 9.1). Binaries for Windows, Mac and Linux which demonstrate the GUI are available for download from my web page. The available binaries on the webpage demonstrate the GUI, but the included core functionality lags behind several versions because work on the GUI has been postponed in favour of the implementation of core CSP functionality and in favor of writing this thesis. Note that the GUI is not part of the `CSPM-cspm` package and that the source code of the GUI has not been released on the Hackage repository.

Elaborating the GUI and also implementing a web front-end for my CSP$_M$ libraries is interesting future work.



```
channel get, put : DISCS
channel full

-- We are allowed to put any *smaller* disc onto the current stack
allowed(s) = { 1..head(s^<n+1>)-1 }

PEG(s) =
  get?d:allowed(s)->PEG(<d>^s)
   []
  not null(s) & put!head(s)->PEG(tail(s))
   []
  length(s) == n & full->PEG(s)

{-
  Now, given a simple peg we can rename it to form each
  of the three physical pegs (`poles') of the puzzle.

  move.d.i.j indicates that disc d moves to pole i from pole j
-}

channel move : DISCS.PEGS.PEGS
channel complete : PEGS

initial(p)  = < 1..n | p == A >

POLE(p) =
  PEG(initial(p))
   [[ full <- complete.p,
      get.d <- move.d.p.i,
      put.d <- move.d.i.p | i<- PEGS, i != p, d<-DISCS ]]

{-
  The puzzle is just the three poles, communicating on the
  relevant events: all the moves, and the done/notdone events.
-}

interface(p) = { move.d.i.p, move.d.p.i, complete.p | d<-DISCS, i<-PEGS }

PUZZLE =
  || p : PEGS @ [ interface(p) ] diamond(POLE(p))

{-
  The puzzle is solved by asserting that C cannot become complete.
  then the trace that refutes the assertion is the solution.
```

```
(INFO) mkDotOut : make dotfile
(INFO) Dot : making /tmp/tmp.dot
(INFO) mkDotOut : make svgfile
(INFO) Dot : calling xsvg (forking)
(INFO) traceProcess : start tracing process
(INFO) Tracer : initTracer
(INFO) traceProcess : start tracing process
(INFO) Tracer : making a step
```

Event ->	Process
	HashMD5_87C7162BAC3C7B7B1B97F2DE5D60E709
move.1.B.A	HashMD5_19FD2916E77F33985F353F9D8572EBEB

Event	-> Process
move.1.A.B	Process
move.1.C.B	Process
move.2.C.A	Process

Event ->	Process
	HashMD5_87C7162BAC3C7B7B1B97F2DE5D60E709
complete.A	HashMD5_9F8F2429262B7FA7D4D7C3C4992110B6

Event	-> Process
move.1.B.A	Process
move.1.C.A	Process
complete.A	Process

Figure 9.1: Screenshot of the GUI

Chapter 10

Summary

10.1 Implementation

The bigger part of this thesis consists of a detailed description of the implementation of my CSP$_M$ tool. The implementation is split into three modules, i.e. the parser, the CSP core language and the functional sub-language of CSP$_M$—and the presentation follows that structure.

One requirement for my tool was to achieve high compatibility with FDR and ProB. As a consequence, a lot of effort was put into investigating all the particularities of CSP$_M$ and carefully implementing all details and special cases. One contribution of the thesis is that it provides a thorough documentation of a complete FDR compatible CSP$_M$ tool. The source code of my software and the test cases together with the thesis can serve as a reference implementation for any future work on CSP$_M$ tools.

Given the complexity of the project, correctness is of course a major concern. I took several measures to ensure the correctness of my software. For example, I use randomized test generation, correct by construction techniques, code coverage analysis and extensive back-box testing. Two-way refinement checking of the generated state space versus the original specification has turned out to be extremely useful and helped to find subtle differences between the tools. I have described the approaches in the corresponding sections of the thesis.

The thesis also contains a comparison of my tool with existing CSP$_M$ tools. I have listed some preliminary benchmarks for the performance of my implementation and I have also presented some first steps towards a parallel CSP$_M$ model checker, including some parallel benchmarks.

10.2 Haskell

This thesis is also a case study for the use of the Haskell programming language for writing a formal methods tool. I found that Haskell is well-suited for modeling and implementing symbolic computations. For example Haskell's algebraic data types together with pattern matching perfectly support the implementation of abstract syntax trees, inference trees and tree-structured CSP-processes.

The thesis contains several arguments of why Haskell helps to improve the correctness of the tool. Important features are the use of pure functions, strict control of side effects and strong static typing.

The thesis contains a lot of Haskell source code. Source code can be compiled, executed and tested and it necessarily covers all details of the implementation. At the same time, declarative source code can also serve as an executable documentation. Haskell favors a declarative programming style and I tried to program as declaratively as possible.

An example of this is the firing rules semantics. The implementation of the proof tree verifier can serve as an executable documentation of the firing rules. For illustration, I have typeset the supported firing rules in the usual style for inference rules and linked each rule directly to the Haskell source code of the proof verifier in Appendix A.

The complete source code of my project is available online as `cabal` packages. It is easy to compile and install the `cspm` tool and it is easy to test and reproduce the results. The Haskell community provides the open source GHC-compiler, libraries and other useful infrastructure which helps to make my tool portable and maintainable.

In general, Haskell idioms and techniques are pervasive in the implementation. For example, I use Monads, higher order functions, laziness, polymorphic types, pattern matching, etc. Haskell shines in the implementation of the functional sub-language of CSP$_M$.

In principle, a similar CSP$_M$ tool can be written in any other programming language. However, the thesis lists several reasons why a Haskell implementation provides additional value.

10.3 Criticism of CSP$_M$ and FDR

To make my work interesting for possible users in the CSP community and to make it comparable and compatible with existing tools, I had to support the CSP$_M$ standard as it is. Nevertheless, the thesis lists several points of critique towards CSP$_M$. Here is a summary of the most important points:

Standard by Implementation

The CSP$_M$ standard is basically defined by the reference implementation, namely the FDR tool. FDR has evolved over many years and it contains many corner cases and particularities which should be fixed. Unfortunately, FDR is proprietary code and the development of FDR is centralized in one working group. Most of the CSP community only uses FDR as a black box.

Specification vs. Programming

In principle, CSP$_M$ is just a special purpose programming language which makes it easy to mix CSP constructs with functional programming. The main extra is that CSP$_M$ can be model checked with FDR or other tools.

I found that overall, FDR works well and in my opinion, it contains only few bugs. On the other hand, blind trust in a black-box implementation is still problematic for a formal method and model checking cannot always replace formal correctness proofs.

One drawback of CSP$_M$ is that it leads away from the idea of a process algebra as proposed by C.A.R Hoare, and it leads into the direction of a regular programming language. In general, it is difficult to mechanically prove properties of CSP_M specifications in the sense of theory proving as opposed to model checking.

The fact that CSP_M contains a functional sub-language can lead to specifications which model important parts of a system in a functional style and only use a minimum of original CSP concepts. Such specifications are hard to reason about with the algebraic rules of CSP.

Functional Sub-language of CSP_M

From a Haskell programmer's point of view, it is a pity that FDR defines a new functional programming language. The only really new feature of the CSP_M syntax, with respect to Haskell, is the prefix operation, which works as an additional binder for variable names.

One has to pay a high price for this small syntactic extension. CSP_M is a proprietary and incompatible programming language which has a crippled expressiveness compared to Haskell. CSP_M lacks a proper static type system, a module system, proper algebraic data types, type classes, type signatures and many other features of modern functional programming languages. Furthermore, the semantics of CSP_M is often unclear and important function like the built-in equality operator are only implemented ad-hoc.

Of course, this is only my point of view and it is a current point of view. Historically, FDR and Haskell were developed in parallel and it is therefore not correct to say that FDR deviated from the standard.

10.4 Meta Critique

Yet Another Implementation?

As of today, there are four FDR compatible implementations for CSP_M, namely FDR, ProBE, the CSP part of ProB and `cspm`. The ProB-CSP_M project has been initiated and partly funded by an industrial partner who had explicitly expressed the need for an alternative CSP_M model checker. The work on the `cspm` project has been going on in parallel with the work on ProB. ProB and `cspm` share the same parser and test-suite and there has been a valuable exchange between the authors of the tools.

It may seem that there is little benefit in the re-implementation of an existing tool, but there is a number of reasons why it can still make sense. First of all there was an explicit request from an industrial partner for an alternative tool, and having three alternatives is better than having only two.

Secondly, I found that writing a tool is a very good way to understand and clarify CSP_M. The work on `cspm` has helped me to explore the design space for CSP tools and it could be a good starting point for work on a modern alternative for CSP_M. And finally, all tools are different and all tools have their advantages, disadvantages and their special use cases.

Novelty of the Work

The primary goal of my project was *not* to invent a new formalism or new extensions for CSP. I list some ideas for new features, etc., in the future work section and I think that the presented work is a good starting point for further research.

Although I support the same formalism as existing tools, the presented implementation itself is completely novel. Compared to ProB, I use another programming language and some completely different design ideas which have not been explored in the context of CSP before. On the meta level, having several alternative tools for the same formalism can itself be a novelty. At least, it is in the case of CSP$_M$.

Another novelty is that I try to present the source code itself as an important contribution. A CSP$_M$ model checker is an interesting software project and it can serve as a test case for studding programming languages and software designs. Research in this direction is not possible without source code. The other CSP$_M$ tools are basically presented as a black box for end-users.

Significance of the Arguments ?

A problem of this thesis could be that it contains many weak words. There are several reasons why I found it difficult to write in a strictly scientific style.

A big part of my work was software design. I have tried to explain and argue for the important design decisions of my project and I have tried to objectively compare my software with existing tools. I think that a good design is important for the correctness and reusability of a software. On the other hand, software design usually means making trade-offs and choosing between design alternatives which cannot be fully understood in advance.

There are branches of computer science which develop objective and scientific methods for the evaluation of software designs. However, applying this full scientific methodology was beyond the scope of this work. One could say that I have rather presented my personal opinions than hard facts with respect to the software design. Still, I think that it also makes sense to address the soft aspects of software.

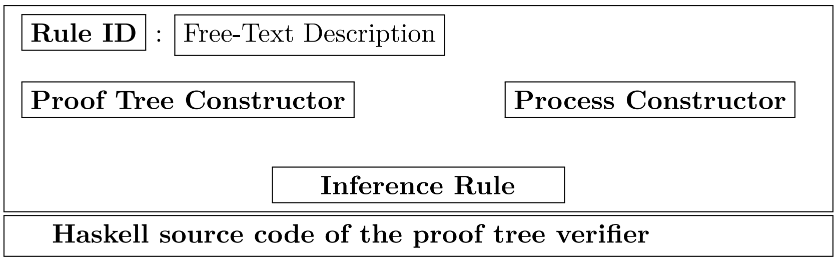
The benchmark sections explain why it is difficult to make conclusive statements about the relative performance of CSP$_M$ tools. In short, the performance of the tools can have extreme variations—depending on the input specification—and there is no canonical fixed set of representative CSP$_M$ specifications. I can only select some specifications and benchmark them.

One solid contribution is that I have presented a software which can be downloaded and executed. For some specifications my software outperforms the existing tools. My arguments are glued together by a working program. The source code can be compiled and tested.

Appendix A

Implemented Firing Rules

This section lists all implemented firing rules together with the relevant Haskell sourcecode of the proof tree verfier. The relation between the firing rules and the Haskell source-code is described in Section 4.2. Rules are printed in the following format:



Rule ID The unique rule ID that is used throughout the thesis.

Proof Tree Constructor The unique constructor in the data type that stores the proof tree.

Process Constructor The constructor from the `Process` data type.

Inference Rule The rule, type-set in the usual style for inference rules.

Haskell source code The implementation of the rule in the proof tree verifier. The contiguous source code for all rules is listed in Appendix B.1.6.

In this section, small letters in the firing rules denote events from Σ, they cannot stand for $\checkmark$ or τ. Furthermore, firing rules that involve a regular event, firing rules for $\checkmark$-transitions and firing rules for τ-transitions are strictly separated. This separation is a difference compared to the how Roscoe presents the firing rules [52], but the separation corresponds one-to-one to the structure of my implementation.

There is one unique Haskell constructor in the proof tree data types for every rule. This constructor can be used to locate all parts of the Haskell source code that are related to this particular inference rule, i.e. the proof tree generator and the proof tree verifier.

The Haskell code that is shown is the complete source code of the proof tree verifier. In principle, there is a schematic translation from the source code of the proof tree verifier to the inference rule. Nevertheless, this documentation has been written by hand and after the fact. In other words, the Haskell source code has been tested but the latex code has only gone through proofreading.

A.0.1 Normal Transitions

R-1: Prefix operation

`HPrefix` `Prefix`

$$\frac{}{(e \to P) \xrightarrow{a} subs(a, e, P)} a \in comms(e)$$

$comms(e)$ is the set of events that are consistent with e.
$subs(a, e, P)$ is the result of substituting the appropriate part of a for each identifier in P bound by e. (quote from [52] page 160).

```
HPrefix e p → do
  p’ ← prefixNext p e
  return (Prefix p, e, p’)
```

R-2: External choice resolves to P

`ExtChoiceL` `ExternalChoice`

$$\frac{P \xrightarrow{e} P'}{P \Box Q \xrightarrow{e} P'}$$

```
ExtChoiceL pp q → do
  (p, e, p’) ← viewRuleEvent pp
  return (ExternalChoice p q, e, p’)
```

R-3: External choice resolves to Q

`ExtChoiceR` `ExternalChoice`

$$\frac{Q \xrightarrow{e} Q'}{P \Box Q \xrightarrow{e} Q'}$$

```
ExtChoiceR p qq → do
  (q, e, q’) ← viewRuleEvent qq
  return (ExternalChoice p q, e, q’)
```

R-4: Interleaving: one step of P

`InterleaveL` `Interleave`

$$\frac{P \xrightarrow{e} P'}{P \mathbin{|||} Q \xrightarrow{e} P' \mathbin{|||} Q}$$

```
InterleaveL pp q → do
  (p, e, p') ← viewRuleEvent pp
  return (Interleave p q, e, Interleave p' q)
```

R-5: Interleaving: one step of Q

`InterleaveR` `Interleave`

$$\frac{Q \xrightarrow{e} Q'}{P \mathbin{|||} Q \xrightarrow{e} P \mathbin{|||} Q'}$$

```
InterleaveR p qq → do
  (q, e, q') ← viewRuleEvent qq
  return (Interleave p q, e, Interleave p q')
```

R-6: Sequential composition: P does not terminate

`SeqNormal` `Seq`

$$\frac{P \xrightarrow{e} P'}{P \,;Q \xrightarrow{e} P' \,;Q}$$

```
SeqNormal pp q → do
  (p, e, p') ← viewRuleEvent pp
  return (Seq p q, e, Seq p' q)
```

R-7: Hiding: the event is not hidden

`NotHidden` `Hide`

$$\frac{P \xrightarrow{e} P'}{P \setminus X \xrightarrow{e} P' \setminus X} e \notin X$$

```
NotHidden c pp → do
  (p, e, p') ← viewRuleEvent pp
  not_in_Closure e c
  return (Hide c p, e, Hide c p')
```

R-8: Sharing: the event is not synchronized

`NotShareL` `Sharing`

$$\frac{P \xrightarrow{e} P'}{P \underset{X}{\parallel} Q \xrightarrow{e} P' \underset{X}{\parallel} Q} e \notin X$$

```
NotShareL c pp q → do
  (p, e, p') ← viewRuleEvent pp
  not_in_Closure e c
  return (Sharing p c q, e, Sharing p' c q)
```

R-9: Sharing: the event is not synchronized

`NotShareR` `Sharing`

$$\frac{Q \xrightarrow{e} Q'}{P \underset{X}{\parallel} Q \xrightarrow{e} P \underset{X}{\parallel} Q'} e \notin X$$

```
NotShareR c p qq → do
  (q, e, q') ← viewRuleEvent qq
  not_in_Closure e c
  return (Sharing p c q, e, Sharing p c q')
```

R-10: Sharing, Processes synchronize

`Shared` `Sharing`

$$\frac{P \xrightarrow{e} P' \; Q \xrightarrow{e} Q'}{P \underset{X}{\parallel} Q \xrightarrow{e} P' \underset{X}{\parallel} Q'} e \in X$$

```
Shared c pp qq → do
  (p, e1, p') ← viewRuleEvent pp
  (q, e2, q') ← viewRuleEvent qq
  guard $ eventEq ty e1 e2
  in_Closure e1 c
  return (Sharing p c q, e1, Sharing p' c q')
```

R-11: Alphabetized parallel

`AParallelL` `AParallel`

$$\frac{P \xrightarrow{e} P'}{P \; {}_X\|_Y \; Q \xrightarrow{e} P' \; {}_X\|_Y \; Q} e \in X \wedge e \notin Y$$

```
AParallelL c1 c2 pp q → do
  (p, e, p’) ← viewRuleEvent pp
  in_Closure e c1
  not_in_Closure e c2
  return (AParallel c1 c2 p q, e, AParallel c1 c2 p’ q)
```

R-12: Alphabetized parallel

`AParallelR` `AParallel`

$$\frac{Q \xrightarrow{e} Q'}{P \; {}_X\|_Y \; Q \xrightarrow{e} P \; {}_X\|_Y \; Q'} e \notin X \wedge e \in Y$$

```
AParallelR c1 c2 p qq → do
  (q, e, q’) ← viewRuleEvent qq
  not_in_Closure e c1
  in_Closure e c2
  return (AParallel c1 c2 p q, e, AParallel c1 c2 p q’)
```

R-13: Alphabetized parallel: processes synchronize

`AParallelBoth` `AParallel`

$$\frac{P \xrightarrow{e} P' \quad Q \xrightarrow{e} Q'}{P \; {}_X\|_Y \; Q \xrightarrow{e} P' \; {}_X\|_Y \; Q'} e \in X \wedge e \in Y$$

```
AParallelBoth c1 c2 pp qq → do
  (p, e2, p’) ← viewRuleEvent pp
  (q, e1, q’) ← viewRuleEvent qq
  guard $ eventEq ty e1 e2
  in_Closure e1 c1
  in_Closure e1 c2
  return (AParallel c1 c2 p q, e1, AParallel c1 c2 p’ q’)
```

R-14: Replicated alphabetized parallel

`RepAParallelEvent` `RepAParallel`

$$\frac{P_j \xrightarrow{e} R}{\mathop{\Vert}_{X_i} P_i \xrightarrow{e} \mathop{\Vert}_{X_i} P_i'} e \in X_j, P_i' = \text{if } i = j \text{ then } R \text{ else } P_i$$

```
RepAParallelEvent l → checkRepAParallel l
```

R-15: Interrupt does not occur

`NoInterrupt` `Interrupt`

$$\frac{P \xrightarrow{e} P'}{P \bigtriangleup Q \xrightarrow{e} P' \bigtriangleup Q}$$

```
NoInterrupt pp q → do
  (p, e, p') ← viewRuleEvent pp
  return (Interrupt p q, e, Interrupt p' q)
```

R-16: Interrupt does occur

`InterruptOccurs` `Interrupt`

$$\frac{Q \xrightarrow{e} Q'}{P \bigtriangleup Q \xrightarrow{e} Q'}$$

```
InterruptOccurs p qq → do
  (q, e, q') ← viewRuleEvent qq
  return (Interrupt p q, e, q')
```

R-17: Timeout resolves by P performing a transition

`TimeoutNo` `Timeout`

$$\frac{P \xrightarrow{e} P'}{P \triangleright Q \xrightarrow{e} P'}$$

```
TimeoutNo pp q → do
  (p, e, p') ← viewRuleEvent pp
  return (Timeout p q, e, p')
```

R-18: Renaming via renaming relation

`Rename` `Renaming`

$$\frac{P \xrightarrow{a} P'}{P[\![R]\!] \xrightarrow{b} P'[\![R]\!]}(a,b) \in R$$

```
Rename rel visibleEvent pp → do
  (p, internalEvent, p') ← viewRuleEvent pp
  guard $ isInRenaming ty rel internalEvent visibleEvent
  return (Renaming rel p, visibleEvent, Renaming rel p')
```

R-19: Transition not effected by renaming relation

`RenameNotInDomain` `Renaming`

$$\frac{P \xrightarrow{e} P'}{P[\![R]\!] \xrightarrow{e} P'[\![R]\!]}e \notin dom(R)$$

```
RenameNotInDomain rel pp → do
  (p, e, p') ← viewRuleEvent pp
  guard $ not $ isInRenamingDomain ty e rel
  return (Renaming rel p, e, Renaming rel p')
```

R-20: Chaos: anything can happen

`ChaosEvent` `Chaos`

$$\frac{}{CHAOS_X \xrightarrow{e} CHAOS_X}e \in X$$

```
ChaosEvent c e → do
  in_Closure e c
  return (Chaos c, e, Chaos c)
```

R-21: Transition outside the link relation

`LinkEventL` `LinkParallel`

$$\frac{P \xrightarrow{a} P'}{P[l \leftrightarrow r]Q \xrightarrow{a} P'[l \leftrightarrow r]Q}a \notin dom([l \leftrightarrow r])$$

```
LinkEventL rel pp q → do
  (p, e, p’) ← viewRuleEvent pp
  guard $ not $ isInRenamingDomain ty e rel
  return (LinkParallel rel p q, e, LinkParallel rel p’ q)
```

R-22: Transition covered by the link relation

`LinkEventR` `LinkParallel`

$$\frac{Q \xrightarrow{a} Q'}{P[l \leftrightarrow r]Q \xrightarrow{a} P[l \leftrightarrow r]Q'} a \notin range([l \leftrightarrow r])$$

```
LinkEventR rel p qq → do
  (q, e, q’) ← viewRuleEvent qq
  guard $ not $ isInRenamingRange ty e rel
  return (LinkParallel rel p q, e, LinkParallel rel p q’)
```

R-23: An exception does not occur

`NoException` `Exception`

$$\frac{P \xrightarrow{e} P'}{P[\![c \rhd Q \xrightarrow{e} P'[\![c \rhd Q} e \notin c$$

```
NoException c pp q → do
  (p, e, p’) ← viewRuleEvent pp
  not_in_Closure e c
  return (Exception c p q, e, Exception c p’ q)
```

R-24: An exception occurs

`ExceptionOccurs` `Exception`

$$\frac{Q \xrightarrow{e} Q'}{P[\![c \rhd Q \xrightarrow{e} Q'} e \in c$$

```
ExceptionOccurs c p qq → do
  (q, e, q’) ← viewRuleEvent qq
  in_Closure e c
  return (Exception c p q, e, q’)
```

A.0.2 ✓ Transitions

R-25: Regular process termination

`SkipTick` `Skip`

$$\frac{}{\mathit{SKIP} \xrightarrow{\checkmark} \Omega}$$

```
SkipTick → return Skip
```

R-26: ✓ cannot be hidden

`HiddenTick` `Hide`

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \setminus X \xrightarrow{\checkmark} \Omega}$$

```
HiddenTick c pp → do
  p ← viewRuleTick pp
  return $ Hide c p
```

R-27: Termination of interrupt

`InterruptTick` `Interrupt`

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \triangle Q \xrightarrow{\checkmark} \Omega}$$

```
InterruptTick pp q → do
  p ← viewRuleTick pp
  return $ Interrupt p q
```

R-28: Termination of timeout

`TimeoutTick` `Timeout`

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \triangleright Q \xrightarrow{\checkmark} \Omega}$$

```
TimeoutTick pp q → do
  p ← viewRuleTick pp
  return $ Timeout p q
```

R-29: Synchronized termination of sharing

`ShareOmega` `Sharing`

$$\frac{}{\Omega \underset{X}{\parallel} \Omega \xrightarrow{\checkmark} \Omega}$$

```
ShareOmega c → return $ Sharing Omega c Omega
```

R-30: Synchronized termination of alphabetized parallel

`AParallelOmega` `AParallel`

$$\frac{}{\Omega \;{}_{X}\|_{Y}\; \Omega \xrightarrow{\checkmark} \Omega}$$

```
AParallelOmega c1 c2 → return $ AParallel c1 c2 Omega Omega
```

R-31: Synchronized termination of replicated alphabetized parallel

`RepAParallelOmega` `RepAParallel`

$$\frac{}{\underset{X_i}{\Big\|}\; \Omega \xrightarrow{\checkmark} \Omega}$$

```
RepAParallelOmega l
  → return $ RepAParallel $ zip l $ repeat Omega
```

R-32: Synchronized termination of interleaving

`InterleaveOmega` `Interleave`

$$\frac{}{\Omega \;|||\; \Omega \xrightarrow{\checkmark} \Omega}$$

```
InterleaveOmega → return (Interleave Omega Omega)
```

R-33: Termination of external choice is not synchronized

`ExtChoiceTickL` `ExternalChoice`

$$\frac{P \xrightarrow{\checkmark} \Omega}{P \mathbin{\Box} Q \xrightarrow{\checkmark} \Omega}$$

```
ExtChoiceTickL pp q → do
  p ← viewRuleTick pp
  return $ ExternalChoice p q
```

R-34: Termination of external choice is not synchronized

`ExtChoiceTickR` `ExternalChoice`

$$\frac{Q \xrightarrow{\checkmark} \Omega}{P \mathbin{\Box} Q \xrightarrow{\checkmark} \Omega}$$

```
ExtChoiceTickR p qq → do
  q ← viewRuleTick qq
  return $ ExternalChoice p q
```

R-35: Termination of renaming

`RenamingTick` `Renaming`

$$\frac{P \xrightarrow{\checkmark} \Omega}{P[\![R]\!] \xrightarrow{\checkmark} \Omega}$$

```
RenamingTick rel pp → do
  p ← viewRuleTick pp
  return $ Renaming rel p
```

R-36: Termination of linked processes

`LinkParallelTick` `LinkParallel`

$$\frac{}{\Omega[l \leftrightarrow r]\Omega \xrightarrow{\checkmark} \Omega}$$

```
LinkParallelTick rel
  → return $ LinkParallel rel Omega Omega
```

A.0.3 τ Transitions

Most of the τ rules are just needed to propagate ✓ and τ events.

R-37: A hidden event: one source of τ events

Hidden Hide

$$\frac{P \xrightarrow{e} P'}{P \setminus X \xrightarrow{\tau} P' \setminus X} e \in X$$

```
Hidden c pp → do
  (p, e, p') ← viewRuleEvent pp
  guard $ member (undefined :: i) e c
  return (Hide c p, Hide c p')
```

R-38: Propagation of τ

HideTau Hide

$$\frac{P \xrightarrow{\tau} P'}{P \setminus X \xrightarrow{\tau} P' \setminus X}$$

```
HideTau c pp → do
  (p, p') ← viewRuleTau pp
  return (Hide c p, Hide c p')
```

R-39: Propagation of τ

SeqTau Seq

$$\frac{P \xrightarrow{\tau} P'}{P \,;\, Q \xrightarrow{\tau} P' \,;\, Q}$$

```
SeqTau pp q → do
  (p, p') ← viewRuleTau pp
  return (Seq p q, Seq p' q)
```

R-40: Resolving of sequential composition produces a τ

`SeqTick` `Seq`

$$\frac{P \xrightarrow{\checkmark} \Omega}{P\,;Q \xrightarrow{\tau} Q}$$

```
SeqTick pp q → do
  p ← viewRuleTick pp
  return (Seq p q, q)
```

R-41: Resolve the internal choice to P

`InternalChoiceL` `InternalChoice`

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P}$$

```
InternalChoiceL p q → return (InternalChoice p q,p)
```

R-42: Resolve the internal choice to Q

`InternalChoiceR` `InternalChoice`

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

```
InternalChoiceR p q → return (InternalChoice p q,q)
```

R-43: *CHAOS* resolves to *STOP*

`ChaosStop` `Chaos`

$$\frac{}{\mathit{CHAOS}_X \xrightarrow{\tau} \mathit{STOP}}$$

```
ChaosStop e → return (Chaos e, Stop)
```

R-44: A timeout occurs

`TimeoutOccurs` `Timeout`

$$\frac{}{P \triangleright Q \xrightarrow{\tau} Q}$$

```
TimeoutOccurs p q → return (Timeout p q, q)
```

R-45: A τ transition of P does not resolve the timeout

`TimeoutTauR` `Timeout`

$$\frac{P \xrightarrow{\tau} P'}{P \triangleright Q \xrightarrow{\tau} P' \triangleright Q}$$

```
TimeoutTauR r q → do
  (p, p') ← viewRuleTau r
  return (Timeout p q, Timeout p' q)
```

R-46: Propagation of τ

`ExtChoiceTauL` `ExternalChoice`

$$\frac{P \xrightarrow{\tau} P'}{P \,\square\, Q \xrightarrow{\tau} P' \,\square\, Q}$$

```
ExtChoiceTauL pp q → do
  (p, p') ← viewRuleTau pp
  return (ExternalChoice p q, ExternalChoice p' q)
```

R-47: Propagation of τ

`ExtChoiceTauR` `ExternalChoice`

$$\frac{Q \xrightarrow{\tau} Q'}{P \,\square\, Q \xrightarrow{\tau} P \,\square\, Q'}$$

```
ExtChoiceTauR p qq → do
  (q, q') ← viewRuleTau qq
  return (ExternalChoice p q, ExternalChoice p q')
```

R-48: Propagation of τ

`InterleaveTauL` `Interleave`

$$\frac{P \xrightarrow{\tau} P'}{P \mid\mid\mid Q \xrightarrow{\tau} P' \mid\mid\mid Q}$$

```
InterleaveTauL pp q → do
  (p, p') ← viewRuleTau pp
  return (Interleave p q, Interleave p' q)
```

R-49: Propagation of τ

`InterleaveTauR` `Interleave`

$$\frac{Q \xrightarrow{\tau} Q'}{P \mid\mid\mid Q \xrightarrow{\tau} P \mid\mid\mid Q'}$$

```
InterleaveTauR p qq → do
  (q, q') ← viewRuleTau qq
  return (Interleave p q, Interleave p q')
```

R-50: Propagation of τ

`InterleaveTickL` `Interleave`

$$\frac{P \xrightarrow{\tau} \Omega}{P \mid\mid\mid Q \xrightarrow{\tau} \Omega \mid\mid\mid Q}$$

```
InterleaveTickL pp q → do
  p ← viewRuleTick pp
  return (Interleave p q, Interleave Omega q)
```

R-51: Propagation of τ

`InterleaveTickR` `Interleave`

$$\frac{Q \xrightarrow{\tau} \Omega}{P \mid\mid\mid Q \xrightarrow{\tau} P \mid\mid\mid \Omega}$$

```
InterleaveTickR p qq → do
  q ← viewRuleTick qq
  return (Interleave p q, Interleave p Omega)
```

R-52: Propagation of τ

ShareTauL Sharing

$$\frac{P \xrightarrow{\tau} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P' \underset{X}{\parallel} Q}$$

```
ShareTauL c pp q → do
  (p, p') ← viewRuleTau pp
  return (Sharing p c q, Sharing p' c q)
```

R-53: Propagation of τ

ShareTauR Sharing

$$\frac{Q \xrightarrow{\tau} Q'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} Q'}$$

```
ShareTauR c p qq → do
  (q, q') ← viewRuleTau qq
  return (Sharing p c q, Sharing p c q')
```

R-54: Propagation of τ

ShareTickL Sharing

$$\frac{P \xrightarrow{\tau} \Omega}{P \underset{X}{\parallel} Q \xrightarrow{\tau} \Omega \underset{X}{\parallel} Q}$$

```
ShareTickL c pp q → do
  p ← viewRuleTick pp
  return (Sharing p c q, Sharing Omega c q)
```

R-55: Propagation of τ

`ShareTickR` `Sharing`

$$\frac{Q \xrightarrow{\tau} \Omega}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} \Omega}$$

```
ShareTickR c p qq → do
  q ← viewRuleTick qq
  return (Sharing p c q, Sharing p c Omega)
```

R-56: Propagation of τ

`AParallelTauL` `AParallel`

$$\frac{P \xrightarrow{\tau} P'}{P \ {}_X\|_Y \ Q \xrightarrow{\tau} P' \ {}_X\|_Y \ Q}$$

```
AParallelTauL pc qc r q → do
  (p, p') ← viewRuleTau r
  return (AParallel pc qc p q, AParallel pc qc p' q)
```

R-57: Propagation of τ

`AParallelTauR` `AParallel`

$$\frac{Q \xrightarrow{\tau} Q'}{P \ {}_X\|_Y \ Q \xrightarrow{\tau} P \ {}_X\|_Y \ Q'}$$

```
AParallelTauR pc qc p r → do
  (q, q') ← viewRuleTau r
  return (AParallel pc qc p q, AParallel pc qc p q')
```

R-58: Propagation of τ

`AParallelTickL` `AParallel`

$$\frac{P \xrightarrow{\tau} \Omega}{P \ {}_X\|_Y \ Q \xrightarrow{\tau} \Omega \ {}_X\|_Y \ Q}$$

```
AParallelTickL pc qc r q → do
  p ← viewRuleTick r
  return (AParallel pc qc p q, AParallel pc qc Omega q)
```

R-59: Propagation of τ

`AParallelTickR` `AParallel`

$$\frac{Q \xrightarrow{\tau} \Omega}{P \;{}_X\|_Y\; Q \xrightarrow{\tau} P \;{}_X\|_Y\; \Omega}$$

```
AParallelTickR pc qc p r → do
  q ← viewRuleTick r
  return (AParallel pc qc p q, AParallel pc qc p Omega)
```

R-60: Propagation of τ for replicated alphabetized parallel

`TauRepAParallel` `RepAParallel`

```
TauRepAParallel l → do
  parts ← forM l $ λx → case x of
    Left a → return (a, a)
    Right (c, r) → do
      (p, p') ← viewRuleTau r
      return ((c,p), (c,p'))
  return (RepAParallel $ map fst parts
         ,RepAParallel $ map snd parts)
```

R-61: Propagation of τ

`InterruptTauL` `Interrupt`

$$\frac{P \xrightarrow{\tau} P'}{P \triangle Q \xrightarrow{\tau} P' \triangle Q}$$

```
InterruptTauL r q → do
  (p, p') ← viewRuleTau r
  return (Interrupt p q, Interrupt p' q)
```

R-62: Propagation of τ

`InterruptTauR` `Interrupt`

$$\frac{Q \xrightarrow{\tau} Q'}{P \bigtriangleup Q \xrightarrow{\tau} P \bigtriangleup Q'}$$

```
InterruptTauR p r → do
  (q, q') ← viewRuleTau r
  return (Interrupt p q, Interrupt p q')
```

R-63: Propagation of τ

`RenamingTau` `Renaming`

$$\frac{P \xrightarrow{\tau} P'}{P[\![R]\!] \xrightarrow{\tau} P'[\![R]\!]}$$

```
RenamingTau rel pp → do
   (p, p') ← viewRuleTau pp
   return (Renaming rel p, Renaming rel p')
```

R-64: Propagation of τ

`LinkTauL` `LinkParallel`

$$\frac{P \xrightarrow{\tau} P'}{P[l \leftrightarrow r]Q \xrightarrow{\tau} P'[l \leftrightarrow r]Q}$$

```
LinkTauL rel pp q → do
  (p, p') ← viewRuleTau pp
  return (LinkParallel rel p q, LinkParallel rel p' q)
```

R-65: Propagation of τ

`LinkTauR` `LinkParallel`

$$\frac{Q \xrightarrow{\tau} Q'}{P[l \leftrightarrow r]Q \xrightarrow{\tau} P[l \leftrightarrow r]Q'}$$

```
LinkTauR rel p qq → do
  (q, q’) ← viewRuleTau qq
  return (LinkParallel rel p q, LinkParallel rel p q’)
```

R-66: Propagation of τ

`LinkTickL` `LinkParallel`

$$\frac{P \xrightarrow{\tau} \Omega}{P[l \leftrightarrow r]Q \xrightarrow{\tau} \Omega[l \leftrightarrow r]Q}$$

```
LinkTickL rel pp q → do
  p ← viewRuleTick pp
  return (LinkParallel rel p q, LinkParallel rel Omega q)
```

R-67: Propagation of τ

`LinkTickR` `LinkParallel`

$$\frac{Q \xrightarrow{\tau} \Omega}{P[l \leftrightarrow r]Q \xrightarrow{\tau} P[l \leftrightarrow r]\Omega}$$

```
LinkTickR rel p qq → do
  q ← viewRuleTick qq
  return (LinkParallel rel p q, LinkParallel rel p Omega)
```

R-68: Linking two processes hides internal communications

`LinkLinked` `LinkParallel`

$$\frac{P \xrightarrow{a} P' \; Q \xrightarrow{b} Q'}{P[l \leftrightarrow r]Q \xrightarrow{\tau} P'[l \leftrightarrow r]Q'} (a, b) \in linkRelation([l \leftrightarrow r])$$

```
LinkLinked rel pp qq → do
  (p, e1, p’) ← viewRuleEvent pp
  (q, e2, q’) ← viewRuleEvent qq
  guard $ isInRenaming (undefined :: i) rel e1 e2
  return (LinkParallel rel p q, LinkParallel rel p’ q’)
```

Appendix B

Source Code Listings

This section contains the complete listings of some of the implemented modules.

B.1 CSP Core Language

B.1.1 Processes

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.CoreLanguage.Process
-- Copyright   :  (c) Fontaine 2011
-- License     :  BSD
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- This modules defines an FDR-compatible CSP core language.
-- The core language deals with CSP-related constructs like processes and events.
--
-- The implementation of the underlying language
-- must provide instances for the type families 'Prefix', 'ExtProcess'
-- and class 'BL'.
-----------------------------------------------------------------------------
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DeriveDataTypeable #-}

module CSPM.CoreLanguage.Process
where
import Data.Typeable

import CSPM.CoreLanguage.Event

-- | A prefix expression.
type family Prefix i

-- | A process that has not yet been switched on.
type family ExtProcess i

class (BE i) ⇒ BL i where
  -- | Try to perform an 'Event' return the successor 'Process' or Nothing
  --   if the event is not possible.
  prefixNext :: Prefix i → Event i → Maybe (Process i)
  switchOn :: ExtProcess i → Process i

{-|
  A data type for CSPM processes.
  For efficiency, replicated alphabetized parallel has an explicit constructor.
  Other replicated operations get translated on the fly.
  For constructing processes one should rather use the wrappers from
  CSPM.CoreLanguage.ProcessWrappers.
-}
data Process i
  = Prefix (Prefix i)
  | ExternalChoice (Process i) (Process i)
  | InternalChoice (Process i) (Process i)
  | Interleave  (Process i) (Process i)
  | Interrupt (Process i) (Process i)
  | Timeout (Process i) (Process i)
  | Sharing (Process i) (EventSet i) (Process i)
  | AParallel (EventSet i) (EventSet i) (Process i) (Process i)
  | RepAParallel [(EventSet i,Process i)]
  | Seq (Process i) (Process i)
  | Hide (EventSet i) (Process i)
  | Stop
  | Skip
  | Omega
  | Chaos (EventSet i)
  | AProcess Int -- ^ Just for debugging.
  | SwitchedOff  (ExtProcess i)
  | Renaming (RenamingRelation i) (Process i)
  | LinkParallel (RenamingRelation i) (Process i) (Process i)
  | Exception (EventSet i) (Process i) (Process i)
  deriving Typeable

isOmega :: Process i → Bool
isOmega Omega = True
isOmega _ = False
```

B.1.2 ProcessWrapper

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.CoreLanguage.Process
-- Copyright   :  (c) Fontaine 2010
-- License     :  BSD
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- Wrappers for the constructors of data type 'Process' and some
-- rewriting rules for replicated operations.
--
-- This can also be used as EDSL for CSP.
--
-----------------------------------------------------------------------------

module CSPM.CoreLanguage.ProcessWrapper
where

import CSPM.CoreLanguage.Process
import CSPM.CoreLanguage.Event

prefix :: Prefix i → Process i
prefix = Prefix

externalChoice :: Process i → Process i → Process i
externalChoice = ExternalChoice

internalChoice :: Process i → Process i → Process i
internalChoice = InternalChoice

interleave :: Process i → Process i → Process i
interleave = Interleave

interrupt :: Process i → Process i → Process i
interrupt = Interrupt

timeout :: Process i → Process i → Process i
timeout = Timeout

sharing :: Process i → EventSet i → Process i → Process i
sharing = Sharing

aparallel ::
     EventSet i → EventSet i
  → Process i → Process i
  → Process i
aparallel = AParallel

seq :: Process i → Process i → Process i
seq = Seq

hide :: EventSet i → Process i → Process i
hide = Hide

stop :: Process i
stop = Stop

skip :: Process i
skip = Skip

switchedOff  :: ExtProcess i → Process i
switchedOff = SwitchedOff

renaming :: RenamingRelation i → Process i → Process i
renaming = Renaming

linkParallel :: RenamingRelation i → Process i → Process i → Process i
linkParallel = LinkParallel

repSeq :: [Process i] → Process i
repSeq = foldr CSPM.CoreLanguage.ProcessWrapper.seq  skip

{- todo: create balanced trees of operators instead of list -}
repInternalChoice :: [Process i] → Process i
repInternalChoice [] = stop
repInternalChoice l = foldr1 internalChoice l

repExternalChoice :: [Process i] → Process i
repExternalChoice [] = stop
repExternalChoice l = foldr1 externalChoice l

repInterleave :: [Process i] → Process i
repInterleave = foldr interleave skip
```

```
repAParallel :: [(EventSet i,Process i)] → Process i
repAParallel l = case l of
  [] → error "ProcessWrapper.hs: empty repAParallel"
  [(_,p)] → p
  _ → RepAParallel l

repSharing :: EventSet i → [Process i] → Process i
repSharing _ [] = error "ProcessWrapper.hs: empty repSharing"
repSharing _ [p] = p
repSharing c l = foldr1 (λa b → sharing a c b) l

repLinkParallel :: RenamingRelation i → [Process i] → Process i
repLinkParallel _ [] = error "ProcessWrapper.hs: empty repLinkParallel"
repLinkParallel _ [_]
  = error "ProcessWrapper.hs: repLinkParallel over one process"
repLinkParallel rel l = foldr1 (λa b → linkParallel rel a b) l

chaos :: EventSet i → Process i
chaos = Chaos
```

B.1.3 Events

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.CoreLanguage.Event
-- Copyright   :  (c) Fontaine 2010 - 2011
-- License     :  BSD3
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- This module defines the event-related part of an interface between
-- the CSPM-CoreLanguage and the underlying implementation.
-- The underlying implementation has to instantiate the type families 'Event',
-- 'EventSet', 'RenamingRelation'
-- and the class 'BE' ('BE'== base event).
--
-- For full CSPM support (channels with multiple fields, event closure sets etc.)
-- CSPM.CoreLanguage.Field is also needed.
-----------------------------------------------------------------------------
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DeriveDataTypeable #-}

module CSPM.CoreLanguage.Event
where
import Data.Typeable

type family Event i
type family EventSet i
type family RenamingRelation i

-- | Sigma is the set of all events that appear in a system.
type Sigma i = EventSet i

-- | The first argument of all functions in 'BE' is a phantom-type-argument, i.e.
-- applications pass _|_ and implementations must not use this value.
class BE i where
  eventEq :: i → Event i → Event i → Bool
  member ::  i → Event i → EventSet i → Bool
  intersection :: i → EventSet i → EventSet i → EventSet i
  difference :: i → EventSet i → EventSet i → EventSet i
  union :: i → EventSet i → EventSet i → EventSet i
  null :: i → EventSet i → Bool
  singleton :: i → Event i → EventSet i
  insert :: i → Event i → EventSet i → EventSet i
  delete :: i → Event i → EventSet i → EventSet i
  eventSetToList :: i → EventSet i → [Event i]
  allEvents :: i → EventSet i
  isInRenaming :: i → RenamingRelation i → Event i → Event i → Bool
  imageRenaming :: i → RenamingRelation i → Event i → [Event i]
  preImageRenaming :: i → RenamingRelation i → Event i → [Event i]
  isInRenamingDomain :: i → Event i → RenamingRelation i → Bool
  isInRenamingRange :: i → Event i → RenamingRelation i → Bool
  getRenamingDomain :: i → RenamingRelation i → [Event i]
  getRenamingRange  :: i → RenamingRelation i → [Event i]
  renamingFromList :: i → [(Event i, Event i)] → RenamingRelation i
  renamingToList :: i → RenamingRelation i → [(Event i, Event i)]
  singleEventToClosureSet :: i → Event i → EventSet i

-- | A wrapper for tick-events, tau-events and events from Sigma.
data TTE i
  = TickEvent
  | TauEvent
  | SEvent (Event i)
  deriving Typeable

class ShowEvent i where showEvent :: i → String
class ShowTTE i where showTTE :: i → String
```

B.1.4 Fields

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.CoreLanguage.Field
-- Copyright   :  (c) Fontaine 2010 - 2011
-- License     :  BSD
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- This module defines the class 'BF' for versions of CSP
-- that also support multi-field-events and event-closure sets.
-----------------------------------------------------------------------------
{-# LANGUAGE TypeFamilies #-}

module CSPM.CoreLanguage.Field
where

import CSPM.CoreLanguage.Event
import CSPM.CoreLanguage.Process

type family Field i
type family FieldSet i
type family ClosureState i
type family PrefixState i

class BL i ⇒ BF i where
  fieldEq :: i → Field i → Field i → Bool
  member :: i → Field i → FieldSet i → Bool
  intersection :: i → FieldSet i → FieldSet i → FieldSet i
  difference :: i → FieldSet i → FieldSet i → FieldSet i
  union :: i → FieldSet i → FieldSet i → FieldSet i
  null :: i → FieldSet i → Bool
  singleton :: i → Field i → FieldSet i
  insert :: i → Field i → FieldSet i → FieldSet i
  delete :: i → Field i → FieldSet i → FieldSet i
  fieldSetToList :: i → FieldSet i → [Field i]
  fieldSetFromList :: i → [Field i] → FieldSet i

  joinFields :: i → [Field i] → Event i
  splitFields :: i → Event i → [Field i]
  channelLen :: i → Field i → Int

  closureStateInit :: i → EventSet i → ClosureState i
  closureStateNext :: i → ClosureState i → Field i → ClosureState i
  closureRestore   :: i → ClosureState i → EventSet i
  viewClosureState :: i → ClosureState i → ClosureView
  viewClosureFields :: i → ClosureState i → FieldSet i
  seenPrefixInClosure :: i → ClosureState i → Bool

  prefixStateInit :: i → Prefix i → PrefixState i
  prefixStateNext :: i → PrefixState i → Field i → Maybe (PrefixState i)
  prefixStateFinalize :: i → PrefixState i → Maybe (Prefix i)
  viewPrefixState :: i → PrefixState i → PrefixFieldView i

data ClosureView
  = InClosure
  | NotInClosure
  | MaybeInClosure
  deriving (Show,Eq,Ord)

data PrefixFieldView i
  = FieldOut (Field i)
  | FieldIn
  | FieldGuard (FieldSet i)
```

B.1.5 Firing Rules

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.FiringRules.Rules
-- Copyright   :  (c) Fontaine 2010 - 2011
-- License     :  BSD3
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- This module defines data types for (CSP) proof trees.
-- A proof tree shows that a particular transition is valid
-- with respect to the firing rules semantics.
--
-- (For more info on the firing rule semantics
-- see: The Theory and Practice of Concurrency A.W. Roscoe 1999.)
--
-- We use three separate data types:
-- 'RuleTau' stores a proof tree for a tau rule,
-- 'RuleTick' stores a proof tree for a tick rule and
-- 'RuleEvent' stores a proof tree for an event from Sigma.
--
-- There is a one-to-one correspondence between
-- each constructor of the data types 'RuleTau', 'RuleTick', 'RuleEvent'
-- and one fireing rule.
--
```

```
-----------------------------------------------------------------------------

{-# LANGUAGE FlexibleContexts, StandaloneDeriving, UndecidableInstances #-}
{-# LANGUAGE DeriveDataTypeable #-}
module CSPM.FiringRules.Rules
where
import CSPM.CoreLanguage
import Data.Typeable

-- | A sum-type for tau, tick and regular proof trees.
data Rule i
  = TauRule (RuleTau i)
  | TickRule (RuleTick i)
  | EventRule (RuleEvent i)
  deriving Typeable

-- | Is this a proof tree for a tau-transition ?
isTauRule :: Rule i → Bool
isTauRule (TauRule {}) = True
isTauRule _ = False

-- | Representation of tau proof trees.
data RuleTau i
  = Hidden (EventSet i) (RuleEvent i)
  | HideTau (EventSet i) (RuleTau i)
  | SeqTau (RuleTau i) (Process i)
  | SeqTick (RuleTick i) (Process i)
  | InternalChoiceL (Process i) (Process i)
  | InternalChoiceR (Process i) (Process i)
  | ChaosStop (EventSet i)
  | TimeoutOccurs (Process i) (Process i)
  | TimeoutTauR (RuleTau i) (Process i)
  | ExtChoiceTauL (RuleTau i) (Process i)
  | ExtChoiceTauR (Process i) (RuleTau i)
  | InterleaveTauL (RuleTau i) (Process i)
  | InterleaveTauR (Process i) (RuleTau i)
  | InterleaveTickL (RuleTick i) (Process i)
  | InterleaveTickR (Process i) (RuleTick i)
  | ShareTauL (EventSet i) (RuleTau i) (Process i)
  | ShareTauR (EventSet i) (Process i) (RuleTau i)
  | ShareTickL (EventSet i) (RuleTick i) (Process i)
  | ShareTickR (EventSet i) (Process i) (RuleTick i)
  | AParallelTauL (EventSet i) (EventSet i) (RuleTau i) (Process i)
  | AParallelTauR (EventSet i) (EventSet i) (Process i) (RuleTau i)
  | AParallelTickL (EventSet i) (EventSet i) (RuleTick i) (Process i)
  | AParallelTickR (EventSet i) (EventSet i) (Process i) (RuleTick i)
  | InterruptTauL (RuleTau i) (Process i)
  | InterruptTauR (Process i) (RuleTau i)
  | TauRepAParallel [Either (EventSet i,Process i) (EventSet i,RuleTau i)]
  | RenamingTau (RenamingRelation i) (RuleTau i)
  | LinkLinked (RenamingRelation i) (RuleEvent i) (RuleEvent i)
  | LinkTauL (RenamingRelation i) (RuleTau i) (Process i)
  | LinkTauR (RenamingRelation i) (Process i) (RuleTau i)
  | LinkTickL (RenamingRelation i) (RuleTick i) (Process i)
  | LinkTickR (RenamingRelation i)  (Process i) (RuleTick i)
  | ExceptionTauL (EventSet i) (RuleTau i) (Process i)
  | ExceptionTauR (EventSet i) (Process i) (RuleTau i)
  | TraceSwitchOn (Process i) -- pseudo-tau for debugging

-- | Representation of tick proof trees.
data RuleTick i
  = SkipTick
  | HiddenTick (EventSet i) (RuleTick i)
  | InterruptTick (RuleTick i) (Process i)
  | TimeoutTick (RuleTick i) (Process i)
  | ShareOmega (EventSet i)
  | AParallelOmega (EventSet i) (EventSet i)
  | RepAParallelOmega [EventSet i]
  | InterleaveOmega
  | ExtChoiceTickL (RuleTick i) (Process i)
  | ExtChoiceTickR (Process i) (RuleTick i)
  | RenamingTick (RenamingRelation i) (RuleTick i)
  | LinkParallelTick (RenamingRelation i)

-- | Representation of regular proof trees.
data RuleEvent i
  = HPrefix (Event i) (Prefix i)
  | ExtChoiceL (RuleEvent i) (Process i)
  | ExtChoiceR (Process i) (RuleEvent i)
  | InterleaveL (RuleEvent i) (Process i)
  | InterleaveR (Process i) (RuleEvent i)
  | SeqNormal (RuleEvent i) (Process i)
  | NotHidden (EventSet i) (RuleEvent i)
  | NotShareL (EventSet i) (RuleEvent i) (Process i)
  | NotShareR (EventSet i) (Process i) (RuleEvent i)
  | Shared (EventSet i) (RuleEvent i) (RuleEvent i)
  | AParallelL (EventSet i) (EventSet i) (RuleEvent i) (Process i)
  | AParallelR (EventSet i) (EventSet i) (Process i) (RuleEvent i)
  | AParallelBoth (EventSet i) (EventSet i) (RuleEvent i) (RuleEvent i)
  | RepAParallelEvent [EventRepAPart i]
  | NoInterrupt (RuleEvent i) (Process i)
  | InterruptOccurs (Process i) (RuleEvent i)
  | TimeoutNo (RuleEvent i) (Process i)
  | Rename (RenamingRelation i) (Event i) (RuleEvent i)
-- todo make special cases for Rename injective and rename relational
  | RenameNotInDomain (RenamingRelation i) (RuleEvent i)
  | ChaosEvent (EventSet i) (Event i)
  | LinkEventL (RenamingRelation i) (RuleEvent i) (Process i)
  | LinkEventR (RenamingRelation i) (Process i) (RuleEvent i)
  | NoException (EventSet i) (RuleEvent i)  (Process i)
  | ExceptionOccurs (EventSet i) (Process i) (RuleEvent i)

type EventRepAPart i
  = Either (EventSet i, Process i) (EventSet i, RuleEvent i)

{-
Not sure about this.
Maybe this moves somewhere else or should be implemented differently.
This is somehow complicated by the use of type families
-}
deriving instance
  (Show (Event i), Show (Prefix i), Show (Process i), Show (ExtProcess i)
  ,Show (EventSet i), Show (RenamingRelation i))
  ⇒ Show (RuleEvent i)
deriving instance
  (Eq (Event i), Eq (Prefix i), Eq (Process i), Eq (ExtProcess i)
  ,Eq (EventSet i), Eq (RenamingRelation i) )
  ⇒ Eq (RuleEvent i)
deriving instance
  (Ord (Event i), Ord (Prefix i), Ord (Process i), Ord (ExtProcess i)
  ,Ord (EventSet i), Ord (RenamingRelation i) )
  ⇒ Ord (RuleEvent i)
```

```
deriving instance
  (Show (Process i), Show (EventSet i), Show (Prefix i), Show (ExtProcess i)
  ,Show (RenamingRelation i))
  ⇒ Show (RuleTick i)
deriving instance
  (Eq (Process i), Eq (EventSet i), Eq (Prefix i), Eq (ExtProcess i)
  ,Eq (RenamingRelation i))
  ⇒ Eq (RuleTick i)
deriving instance
  (Ord (Process i), Ord (EventSet i), Ord (Prefix i), Ord (ExtProcess i)
  ,Ord (RenamingRelation i))
   ⇒ Ord (RuleTick i)


deriving instance
  (Show (RuleEvent i), Show (RuleTick i), Show (Process i)
  ,Show (EventSet i), Show (RenamingRelation i))
  ⇒ Show (RuleTau i)
deriving instance
  (Eq (RuleEvent i), Eq (RuleTick i), Eq (Process i)
  ,Eq (EventSet i), Eq (RenamingRelation i))
  ⇒ Eq (RuleTau i)
deriving instance
  (Ord (RuleEvent i), Ord (RuleTick i), Ord (Process i), Ord (EventSet i)
  ,Ord (ExtProcess i), Ord (Prefix i), Ord (Event i),Ord (RenamingRelation i) )
  ⇒ Ord (RuleTau i)

deriving instance
  (Show (RuleEvent i), Show (RuleTick i), Show (RuleTau i))
  ⇒ Show (Rule i)

deriving instance
  (Eq (RuleEvent i), Eq (RuleTick i), Eq (RuleTau i))
  ⇒ Eq (Rule i)
```

B.1.6 Proof Tree Verifier

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.FiringRules.Verifier
-- Copyright   :  (c) Fontaine 2010 - 2011
-- License     :  BSD3
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- A checker for the firing rules semantics of CSPM.
--
-- 'viewRuleMaybe' checks that a proof tree is valid
-- with respect to the firing rules semantics of CSPM.
-- It checks that the proof tree is syntactically valid
-- and that all side conditions hold.
--
-- The 'Rule' data type stores proof trees in a compressed form.
-- 'viewRuleMaybe' constructs an explicit representation of the transition.
--
-- 'viewRule' calls 'viewRuleMaybe' and throws an exception if
-- the proof tree was not valid.
-- The proof tree generators in this package only generate valid proof trees.
-- 'viewRule' is used to check that assertion.
-----------------------------------------------------------------------------
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module CSPM.FiringRules.Verifier
  (
   viewRule
  ,viewProcBefore
  ,viewEvent
  ,viewProcAfter
  ,viewRuleMaybe
  ,viewRuleTau
  ,viewRuleTick
  ,viewRuleEvent
  )
where

import CSPM.CoreLanguage
import CSPM.CoreLanguage.Event
import CSPM.FiringRules.Rules

import Control.Monad
import Data.Maybe
import qualified Data.List as List

{-|
  This function constructs an explict representation of the transition
  from the proof tree of the transition.
  The transition as a triple
  (predecessor 'Process', Event, successor 'Process').
  If the proof tree is invalid it throws an exception.
-}
viewRule :: BL i ⇒ Rule i → (Process i, TTE i, Process i)
viewRule proofTree = case viewRuleMaybe proofTree of
  Nothing → error "viewRule : internal error malformed Rule"
  Just v → v

-- | Like 'viewRule' but just return the predecessor process.
viewProcBefore :: BL i ⇒ Rule i → Process i
viewProcBefore = (λ(p,_,_) → p) ∘ viewRule

-- | Like 'viewRule' but just return the event.
viewEvent :: BL i ⇒ Rule i → TTE i
viewEvent =  (λ(_,e,_) → e) ∘ viewRule

-- | Like 'viewRule' but just return the successor process.
viewProcAfter :: BL i ⇒ Rule i  → Process i
viewProcAfter =  (λ(_,_,p) → p) ∘ viewRule

-- | Like 'viewRule' but returns 'Nothing' in case of an invalid proof tree.
viewRuleMaybe :: BL i ⇒ Rule i → Maybe (Process i, TTE i, Process i)
viewRuleMaybe proofTree = case proofTree of
  TauRule r → case viewRuleTau r of
    Just (p, p') → Just (p, TauEvent, p')
    Nothing      → Nothing
  TickRule r → case viewRuleTick r of
    Just p  → Just (p, TickEvent, Omega)
    Nothing → Nothing
  EventRule r → case viewRuleEvent r of
    Just (p, e, p') → Just (p, SEvent e, p')
    Nothing → Nothing

-- | Check a tau rule.
viewRuleTau :: forall i. BL i ⇒ RuleTau i → Maybe (Process i, Process i)
viewRuleTau rule = case rule of
  ExtChoiceTauL pp q → do
    (p, p') ← viewRuleTau pp
    return (ExternalChoice p q, ExternalChoice p' q)
  ExtChoiceTauR p qq → do
    (q, q') ← viewRuleTau qq
    return (ExternalChoice p q, ExternalChoice p q')
  InternalChoiceL p q → return (InternalChoice p q,p)
  InternalChoiceR p q → return (InternalChoice p q,q)
  InterleaveTauL pp q → do
    (p, p') ← viewRuleTau pp
    return (Interleave p q, Interleave p' q)
  InterleaveTauR p qq → do
    (q, q') ← viewRuleTau qq
    return (Interleave p q, Interleave p q')
  InterleaveTickL pp q → do
    p ← viewRuleTick pp
    return (Interleave p q, Interleave Omega q)
  InterleaveTickR p qq → do
    q ← viewRuleTick qq
    return (Interleave p q, Interleave p Omega)
  SeqTau pp q → do
    (p, p') ← viewRuleTau pp
    return (Seq p q, Seq p' q)
  SeqTick pp q → do
    p ← viewRuleTick pp
    return (Seq p q, q)
  Hidden c pp → do
    (p, e, p') ← viewRuleEvent pp
    guard $ member (undefined :: i) e c
    return (Hide c p, Hide c p')
  HideTau c pp → do
    (p, p') ← viewRuleTau pp
    return (Hide c p, Hide c p')
  ShareTauL c pp q → do
    (p, p') ← viewRuleTau pp
    return (Sharing p c q, Sharing p' c q)
  ShareTauR c p qq → do
    (q, q') ← viewRuleTau qq
    return (Sharing p c q, Sharing p c q')
  ShareTickL c pp q → do
    p ← viewRuleTick pp
    return (Sharing p c q, Sharing Omega c q)
  ShareTickR c p qq → do
    q ← viewRuleTick qq
    return (Sharing p c q, Sharing p c Omega)
  AParallelTauL pc qc r q → do
    (p, p') ← viewRuleTau r
    return (AParallel pc qc p q, AParallel pc qc p' q)
  AParallelTauR pc qc p r → do
    (q, q') ← viewRuleTau r
    return (AParallel pc qc p q, AParallel pc qc p q')
  AParallelTickL pc qc r q → do
    p ← viewRuleTick r
    return (AParallel pc qc p q, AParallel pc qc Omega q)
  AParallelTickR pc qc p r → do
    q ← viewRuleTick r
    return (AParallel pc qc p q, AParallel pc qc p Omega)
  InterruptTauL r q → do
    (p, p') ← viewRuleTau r
    return (Interrupt p q, Interrupt p' q)
  InterruptTauR p r → do
```

```
    (q, q') ← viewRuleTau r
    return (Interrupt p q, Interrupt p q')
  TauRepAParallel l → do
    parts ← forM l $ λx → case x of
      Left a → return (a, a)
      Right (c, r) → do
        (p, p') ← viewRuleTau r
        return ((c,p), (c,p'))
    return (RepAParallel $ map fst parts, RepAParallel $ map snd parts)
  TimeoutTauR r q → do
    (p, p') ← viewRuleTau r
    return (Timeout p q, Timeout p' q)
  TimeoutOccurs p q → return (Timeout p q, q)
  RenamingTau rel pp → do
    (p, p') ← viewRuleTau pp
    return (Renaming rel p, Renaming rel p')
  ChaosStop e → return (Chaos e, Stop)
  LinkTauL rel pp q → do
    (p, p') ← viewRuleTau pp
    return (LinkParallel rel p q, LinkParallel rel p' q)
  LinkTauR rel p qq → do
    (q, q') ← viewRuleTau qq
    return (LinkParallel rel p q, LinkParallel rel p q')
  LinkTickL rel pp q → do
    p ← viewRuleTick pp
    return (LinkParallel rel p q, LinkParallel rel Omega q)
  LinkTickR rel p qq → do
    q ← viewRuleTick qq
    return (LinkParallel rel p q, LinkParallel rel p Omega)
  LinkLinked rel pp qq → do
    (p, e1, p') ← viewRuleEvent pp
    (q, e2, q') ← viewRuleEvent qq
    guard $ isInRenaming (undefined :: i) rel e1 e2
    return (LinkParallel rel p q, LinkParallel rel p' q')
  TraceSwitchOn p → return (p, p)

-- | Check a tick rule.
viewRuleTick :: BL i ⇒ RuleTick i → Maybe (Process i)
viewRuleTick rule = case rule of
  InterleaveOmega → return (Interleave Omega Omega)
  HiddenTick c pp → do
    p ← viewRuleTick pp
    return $ Hide c p
  ShareOmega c → return $ Sharing Omega c Omega
  AParallelOmega c1 c2 → return $ AParallel c1 c2 Omega Omega
  SkipTick → return Skip
  ExtChoiceTickL pp q → do
    p ← viewRuleTick pp
    return $ ExternalChoice p q
  ExtChoiceTickR p qq → do
    q ← viewRuleTick qq
    return $ ExternalChoice p q
  InterruptTick pp q → do
    p ← viewRuleTick pp
    return $ Interrupt p q
  TimeoutTick pp q → do
    p ← viewRuleTick pp
    return $ Timeout p q
  RepAParallelOmega l
    → return $ RepAParallel $ zip l $ repeat Omega
  RenamingTick rel pp → do
    p ← viewRuleTick pp
    return $ Renaming rel p
  LinkParallelTick rel
    → return $ LinkParallel rel Omega Omega

-- | Check a regular rule
viewRuleEvent :: forall i. BL i
  ⇒ RuleEvent i → Maybe (Process i, Event i, Process i)
viewRuleEvent rule = case rule of
  HPrefix e p → do
    p’ ← prefixNext p e
    return (Prefix p, e, p’)
  ExtChoiceL pp q → do
    (p, e, p’) ← viewRuleEvent pp
    return (ExternalChoice p q, e, p’)
  ExtChoiceR p qq → do
    (q, e, q’) ← viewRuleEvent qq
    return (ExternalChoice p q, e, q’)
  InterleaveL pp q → do
    (p, e, p’) ← viewRuleEvent pp
    return (Interleave p q, e, Interleave p’ q)
  InterleaveR p qq → do
    (q, e, q’) ← viewRuleEvent qq
    return (Interleave p q, e, Interleave p q’)
  SeqNormal pp q → do
    (p, e, p’) ← viewRuleEvent pp
    return (Seq p q, e, Seq p’ q)
  NotHidden c pp → do
    (p, e, p’) ← viewRuleEvent pp
    not_in_Closure e c
    return (Hide c p, e, Hide c p’)
  NotShareL c pp q → do
    (p, e, p’) ← viewRuleEvent pp
    not_in_Closure e c
    return (Sharing p c q, e, Sharing p’ c q)
  NotShareR c p qq → do
    (q, e, q’) ← viewRuleEvent qq
    not_in_Closure e c
    return (Sharing p c q, e, Sharing p c q’)
  Shared c pp qq → do
    (p, e1, p’) ← viewRuleEvent pp
    (q, e2, q’) ← viewRuleEvent qq
    guard $ eventEq ty e1 e2
    in_Closure e1 c
    return (Sharing p c q, e1, Sharing p’ c q’)
  AParallelL c1 c2 pp q → do
    (p, e, p’) ← viewRuleEvent pp
    in_Closure e c1
    not_in_Closure e c2
    return (AParallel c1 c2 p q, e, AParallel c1 c2 p’ q)
  AParallelR c1 c2 p qq → do
    (q, e, q’) ← viewRuleEvent qq
    not_in_Closure e c1
    in_Closure e c2
    return (AParallel c1 c2 p q, e, AParallel c1 c2 p q’)
  AParallelBoth c1 c2 pp qq → do
    (p, e2, p’) ← viewRuleEvent pp
    (q, e1, q’) ← viewRuleEvent qq
    guard $ eventEq ty e1 e2
    in_Closure e1 c1
    in_Closure e1 c2
    return (AParallel c1 c2 p q, e1, AParallel c1 c2 p’ q’)
  NoInterrupt pp q → do
```

```
    (p, e, p’) ← viewRuleEvent pp
    return (Interrupt p q, e, Interrupt p’ q)
  InterruptOccurs p qq → do
    (q, e, q’) ← viewRuleEvent qq
    return (Interrupt p q, e, q’)
  TimeoutNo pp q → do
    (p, e, p’) ← viewRuleEvent pp
    return (Timeout p q, e, p’)
  RepAParallelEvent l → checkRepAParallel l
  Rename rel visibleEvent pp → do
    (p, internalEvent, p’) ← viewRuleEvent pp
    guard $ isInRenaming ty rel internalEvent visibleEvent
    return (Renaming rel p, visibleEvent, Renaming rel p’)
  RenameNotInDomain rel pp → do
    (p, e, p’) ← viewRuleEvent pp
    guard $ not $ isInRenamingDomain ty e rel
    return (Renaming rel p, e, Renaming rel p’)
  ChaosEvent c e → do
    in_Closure e c
    return (Chaos c, e, Chaos c)
  LinkEventL rel pp q → do
    (p, e, p’) ← viewRuleEvent pp
    guard $ not $ isInRenamingDomain ty e rel
    return (LinkParallel rel p q, e, LinkParallel rel p’ q)
  LinkEventR rel p qq → do
    (q, e, q’) ← viewRuleEvent qq
    guard $ not $ isInRenamingRange ty e rel
    return (LinkParallel rel p q, e, LinkParallel rel p q’)
  NoException c pp q → do
    (p, e, p’) ← viewRuleEvent pp
    not_in_Closure e c
    return (Exception c p q, e, Exception c p’ q)
  ExceptionOccurs c p qq → do
    (q, e, q’) ← viewRuleEvent qq
    in_Closure e c
    return (Exception c p q, e, q’)
  where
    ty = (undefined :: i)
    in_Closure e c = guard $ member ty e c
    not_in_Closure e c = guard $ not $ member ty e c

    checkRepAParallel :: [EventRepAPart i] → Maybe (Process i,Event i,Process i)
    checkRepAParallel l = do
      parts ← forM l $ λx → case x of
        Left w → return $ Left w
        Right (c,r) → do { v ← viewRuleEvent r; return $ Right (c,v) }
--    Check that all events are equal.
      let events = flip mapMaybe parts $ λx → case x of
            Left _ → Nothing
            Right (_,(_,e,_)) → Just e
      guard $ (not $ List.null events)
         && (all (eventEq ty $ head events) $ tail events)
{-
Check that if the event is in a closure set the corresponding process has
also taken part in the event.
-}
      let event = head events
      guard $ flip all parts $ λx → case x of
            Left (closure,_) → not $ member ty event closure
            Right (closure,_) → member ty event closure
      let
        procs = flip map parts $ λx → case x of
```

```
          Left pair → pair
          Right (c,(p,_,_)) → (c,p)
        procs' = flip map parts $ λx → case x of
          Left pair → pair
          Right (c,(_,_,p')) → (c,p')
      return (RepAParallel procs, event, RepAParallel procs')
```

B.1.7 Naive Proof Tree Generation

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.FiringRules.EnumerateEvents
-- Copyright   :  (c) Fontaine 2010
-- License     :  BSD
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- Brute-force computation of all possible transitions of a process.
-- Enumerates all events in 'Sigma'.
--
-----------------------------------------------------------------------------

{-# LANGUAGE ScopedTypeVariables #-}
module CSPM.FiringRules.EnumerateEvents
(
  computeTransitions
 ,eventTransitions
 ,tauTransitions
 ,tickTransitions
)
where

import CSPM.CoreLanguage
import CSPM.CoreLanguage.Event
import CSPM.FiringRules.Rules
import CSPM.FiringRules.Search

import Control.Monad
import Control.Applicative
import Data.Either as Either
import Data.List as List


-- | Compute all possible transitions (via an event from Sigma) for a process.
computeTransitions ::  forall i. BL i
  ⇒ Sigma i → Process i → Search (Rule i)
computeTransitions events p
  = (liftM EventRule $ eventTransitions events p)
         `mplus` (liftM TickRule $ tickTransitions p)
         `mplus` (liftM TauRule $ tauTransitions p)

eventTransitions :: forall i.
     BL i
  ⇒ Sigma i
  → Process i
  → Search (RuleEvent i)
eventTransitions sigma p = do
  e ← anyEvent ty sigma
  buildRuleEvent e p
  where
```

```
    ty = (undefined :: i)

anyEvent :: forall i. BL i ⇒ i → EventSet i → Search (Event i)
anyEvent ty sigma
  = anyOf $ eventSetToList ty sigma

buildRuleEvent :: forall i. BL i ⇒ Event i → Process i → Search (RuleEvent i)
buildRuleEvent event proc = case proc of
  SwitchedOff p → rp $ switchOn p
  Prefix p  → case (prefixNext p event :: Maybe (Process i)) of
    Nothing → mzero
    Just _ → return $ HPrefix event p
  ExternalChoice p q
    →       (ExtChoiceL <$> rp p <*> pure q)
     `mplus` (ExtChoiceR p <$> rp q)
  InternalChoice _ _ → mzero
  Interleave p q
    →       (InterleaveL <$> rp p <*> pure q)
     `mplus` (InterleaveR p <$> rp q)
  Interrupt p q → (NoInterrupt <$> rp p <*> pure q)
     `mplus` (InterruptOccurs p <$> rp q)
  Timeout p q → TimeoutNo <$> rp p <*> pure q
  Sharing p c q → if member ty event c
      then Shared c <$> rp p <*> rp q
      else (NotShareL c <$> rp p <*> pure q)
           `mplus` (NotShareR c p <$> rp q)
  Seq p q → SeqNormal <$> rp p <*> pure q
  AParallel x y p q → case (member ty event x, member ty event y) of
    (True, True) →  AParallelBoth x y <$> rp p <*> rp q
    (True, False) → AParallelL x y <$> rp p <*> pure q
    (False, True) → AParallelR x y p <$> rp q
    (False,False) → mzero
  RepAParallel l → buildRuleRepAParallel event l
  Hide c p → if member ty event c
      then mzero
      else NotHidden c <$> rp p
  Stop → mzero
  Skip → mzero
  Omega → mzero
  AProcess _n → mzero
  Renaming rel p → (do
    e2 ← anyEvent ty (allEvents ty)
    guard $ isInRenaming ty rel e2 event
    rule ← buildRuleEvent e2 p
    return $ Rename rel event rule
    )
    `mplus` (do
       guard $ not $ isInRenamingDomain ty event rel
       RenameNotInDomain rel <$> rp p
       )
  Chaos c → if member ty event c
    then return $ ChaosEvent c event
    else mzero
  LinkParallel rel p q → (do
      guard $ not $ isInRenamingDomain ty event rel
      LinkEventL rel <$> rp p <*> pure q
    ) `mplus` (do
      guard $ not $ isInRenamingRange ty event rel
      LinkEventR rel p <$> rp q
    )
  Exception c p q → if member ty event c
    then ExceptionOccurs c p <$> rp q
    else NoException c <$> rp p <*> pure q
  where
    rp = buildRuleEvent event
    ty = (undefined :: i)

buildRuleRepAParallel :: forall i. BL i
  ⇒ Event i
  → [(EventSet i, Process i)] → Search (RuleEvent i)
buildRuleRepAParallel event l = do
  l2 ← mapM parPart l
  if List.null $ Either.rights l2
    then mzero
    else return $ RepAParallelEvent l2
  where
    parPart c@(alpha, p) = if member ty event alpha
      then do
        r ← buildRuleEvent event p
        return $ Right (alpha, r)
      else return $ Left c
    ty = (undefined :: i)

tauTransitions :: forall i. BL i ⇒ Process i → Search (RuleTau i)
tauTransitions proc = case proc of
  SwitchedOff p → tauTransitions $ switchOn p
  Prefix {} → mzero
  ExternalChoice p q
    →       (ExtChoiceTauL <$> tauTransitions p <*> pure q)
     `mplus` (ExtChoiceTauR p <$> tauTransitions q)
  InternalChoice p q
    →       (return $ InternalChoiceL p q)
     `mplus` (return $ InternalChoiceR p q)
  Interleave p q
    →       (InterleaveTauL <$> tauTransitions p <*> pure q)
     `mplus` (InterleaveTauR p <$> tauTransitions q)
     `mplus` (InterleaveTickL <$> tickTransitions p <*> pure q)
     `mplus` (InterleaveTickR p <$> tickTransitions q)
  Interrupt p q
    →       (InterruptTauL <$> tauTransitions p <*> pure q)
     `mplus` (InterruptTauR p <$> tauTransitions q)
  Timeout p q
    →       (TimeoutTauR <$> tauTransitions p <*> pure q)
     `mplus` (return $ TimeoutOccurs p q)
  Sharing p c q
    →       (ShareTauL c <$> tauTransitions p <*> pure q)
     `mplus` (ShareTauR c p <$> tauTransitions q)
     `mplus` (ShareTickL c <$> tickTransitions p <*> pure q)
     `mplus` (ShareTickR c p <$> tickTransitions q)
  AParallel pc qc p q
    →       (AParallelTauL pc qc <$> tauTransitions p <*> pure q)
     `mplus` (AParallelTauR pc qc p <$> tauTransitions q)
     `mplus` (AParallelTickL pc qc <$> tickTransitions p <*> pure q)
     `mplus` (AParallelTickR pc qc p <$> tickTransitions q)
  Seq p q
    →        (SeqTau <$> tauTransitions p <*> pure q)
      `mplus` (SeqTick <$> tickTransitions p <*> pure q)
  Hide hidden p → (do
    e ← anyEvent ty hidden
    rule ← buildRuleEvent e p
    return $ Hidden hidden rule)
   `mplus` (HideTau hidden <$> tauTransitions p)
  Stop → mzero
  Skip → mzero
  Omega → mzero
  AProcess _n → mzero
  RepAParallel l → mzero -- TODO ! tau for replicated AParallel
  Renaming rel p → RenamingTau rel <$> tauTransitions p
  Chaos c → return $ ChaosStop c
  LinkParallel rel p q
    →        (LinkTauL rel <$> tauTransitions p <*> pure q)
      `mplus` (LinkTauR rel p <$> tauTransitions q)
      `mplus` (LinkTickL rel <$> tickTransitions p <*> pure q)
      `mplus` (LinkTickR rel p <$> tickTransitions q)
      `mplus` mkLinkedRules rel p q
  Exception c p q → mzero -- TODO
  where
    ty = (undefined :: i)

mkLinkedRules :: forall i. BL i
   ⇒ RenamingRelation i
   → Process i
   → Process i
   → Search (RuleTau i)
mkLinkedRules rel p q = do
  (e1, r1) ← rules1
  (e2, r2) ← rules2
  guard $ isInRenaming ty rel e1 e2
  return $ LinkLinked rel r1 r2
  where
    rules1 :: Search (Event i, RuleEvent i)
    rules1 = rules (getRenamingDomain ty rel) p
    rules2 = rules (getRenamingRange ty rel) q
    rules :: [Event i] → Process i → Search (Event i, RuleEvent i)
    rules s proc = do
      e ← anyOf s
      r ← buildRuleEvent e proc
      return (e,r)
    ty = (undefined :: i)

tickTransitions :: BL i ⇒ Process i → Search (RuleTick i)
tickTransitions proc = case proc of
  SwitchedOff p → tickTransitions $ switchOn p
  Prefix {} → mzero
  ExternalChoice p q
    →       (ExtChoiceTickL <$> tickTransitions p <*> pure q)
     `mplus` (ExtChoiceTickR p <$> tickTransitions q)
  InternalChoice _p _q → mzero
  Interleave Omega Omega → return $ InterleaveOmega
  Interleave _ _ → mzero
  Interrupt p q → InterruptTick <$> tickTransitions p <*> pure q
  Timeout p q → TimeoutTick <$> tickTransitions p <*> pure q
  Sharing Omega c Omega → return $ ShareOmega c
  Sharing _ _ _ → mzero
  AParallel c1 c2 Omega Omega → return $ AParallelOmega c1 c2
  AParallel _ _ _ _ → mzero
  Seq _p _q → mzero
  Hide c p → HiddenTick c <$> tickTransitions p
  Stop → mzero
  Skip → return SkipTick
  Omega → mzero
  AProcess _n → mzero
  RepAParallel l → if all (isOmega ∘ snd) l
    then return $ RepAParallelOmega $ map fst l
    else mzero
  Renaming rel p → RenamingTick rel <$> tickTransitions p
  Chaos _ → mzero
  LinkParallel rel Omega Omega → return $ LinkParallelTick rel
  LinkParallel _ _ _ → mzero
  Exception c p q → mzero -- TODO
```

B.1.8 Proof Tree Generation with Constraints

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.FiringRules.FieldConstraintsSearch
-- Copyright   :  (c) Fontaine 2010 - 2011
-- License     :  BSD
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- Field-wise generation of transitions.
-- Uses some kind of abstract interpretation/constraint propagation to avoid
-- enumeration of 'Sigma' in some cases.
--
-----------------------------------------------------------------------------

{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE ViewPatterns #-}
module CSPM.FiringRules.FieldConstraintsSearch
(
  computeTransitions
 ,eventTransitions
 ,tauTransitions
 ,tickTransitions
)
where

import CSPM.CoreLanguage.Process
import qualified CSPM.CoreLanguage.Event as Event
import CSPM.CoreLanguage.Field as Field
import CSPM.FiringRules.Rules as Rules
import CSPM.FiringRules.Search

import Control.Arrow
import Control.Monad.State
import Control.Applicative
import Data.Maybe
import qualified Data.List as List


computeTransitions ::  forall i. BF i
  ⇒ Event.Sigma i → Process i → Search (Rule i)
computeTransitions events p
  =  (liftM EventRule $ eventTransitions events p)
        `mplus` (liftM TickRule $ tickTransitions p)
        `mplus` (liftM TauRule $ tauTransitions p)

data RuleField i
 = FPrefix (PrefixState i)
 | FExtChoiceL (RuleField i) (Process i)
 | FExtChoiceR (Process i) (RuleField i)
 | FExtChoice (RepExtChoicePart i) (RepExtChoicePart i)
 | FInterleaveL (RuleField i) (Process i)
 | FInterleaveR (Process i) (RuleField i)
 | FSeqNormal (RuleField i) (Process i)
```

```
  | FNotHidden (ClosureState i) (RuleField i)
  | FNotShareL (ClosureState i) (RuleField i) (Process i)
  | FNotShareR (ClosureState i) (Process i) (RuleField i)
  | FShared (ClosureState i) (RuleField i) (RuleField i)
  | FAParallelL (ClosureState i) (ClosureState i) (RuleField i) (Process i)
  | FAParallelR (ClosureState i) (ClosureState i) (Process i) (RuleField i)
  | FAParallelBoth (ClosureState i) (ClosureState i) (RuleField i) (RuleField i)
  | FNoInterrupt (RuleField i) (Process i)
  | FInterrupt (Process i) (RuleField i)
  | FTimeout (RuleField i) (Process i)
  | FRepAParallel (RepAP i)
  | FRenaming (Event.RenamingRelation i) (Process i)
  | FChaos (ClosureState i)
  | FLinkEventL (Event.RenamingRelation i) (RuleField i) (Process i)
  | FLinkEventR (Event.RenamingRelation i) (Process i) (RuleField i)
  | FNoException (ClosureState i) (RuleField i) (Process i)
  | FExceptionOccurs (ClosureState i) (Process i) (RuleField i)

rulePattern :: forall i.
  BF i ⇒ Event.EventSet i → Process i → Search (RuleField i)
rulePattern events proc = case proc of
  SwitchedOff p → rp $ switchOn p
  Prefix p → return $ FPrefix $ prefixStateInit ty p
  ExternalChoice p q
    → joinRepExtChoiceParts
         (initRepExtChoicePart events p)
         (initRepExtChoicePart events q)
  InternalChoice _p _q → mzero
  Interleave p q
    →       (FInterleaveL <$> rp p <*> pure q)
     ‘mplus‘ (FInterleaveR p <$> rp q)
  Interrupt p q → (FNoInterrupt <$> rp p <*> pure q)
     ‘mplus‘ (FInterrupt p <$> rp q)
  Timeout p q → FTimeout <$> rp p <*> pure q
  Sharing p c q
    →       (FShared (initClosure c) <$> rp p <*> rp q)
     ‘mplus‘ (FNotShareL (initClosure c) <$> rp p <*> pure q)
     ‘mplus‘ (FNotShareR (initClosure c) p <$> rp q)
  AParallel pc qc p q
    →       (FAParallelL (initClosure pc) (initClosure qc) <$> rp p <*> pure q)
     ‘mplus‘ (FAParallelR (initClosure pc) (initClosure qc) <$> pure p <*> rp q)
     ‘mplus‘ (FAParallelBoth (initClosure pc) (initClosure qc) <$> rp p <*> rp q)
  Seq p q → FSeqNormal <$> rp p <*> pure q
  Hide c p → FNotHidden (initClosure c) <$> rp p
  Stop → mzero
  Skip → mzero
  Omega → mzero
  AProcess _n → mzero
  RepAParallel l → return $ FRepAParallel $ initRepAParallel l
  Renaming rel p → return $ FRenaming rel p
  Chaos c → return $ FChaos $ initClosure c
  LinkParallel rel p q
    →         (FLinkEventL rel <$> rp p <*> pure q)
       ‘mplus‘ (FLinkEventR rel p <$> rp q)
  Exception c p q
    →         (FNoException (initClosure c) <$> rp p <*> pure q)
       ‘mplus‘ (FExceptionOccurs (initClosure c) p <$> rp q)
  where
    ty = (undefined :: i)
    initClosure = closureStateInit ty
    rp = rulePattern events
```

```
type PropM i a = StateT (FieldSet i) Maybe a

propField :: forall i. BF i ⇒ RuleField i → PropM i ()
propField rule = case rule of
  FPrefix p → case viewPrefixState ty p of
    FieldOut f → fixField f
    FieldIn → return ()
    FieldGuard g → restrictField $ λe → intersection ty e g
  FExtChoiceL r _ → propField r
  FExtChoiceR _ r → propField r
  FExtChoice _p _q → return ()
  FInterleaveL r _ → propField r
  FInterleaveR _ r → propField r
  FSeqNormal r _ → propField r
  FNotHidden hidden r → if closureState hidden == InClosure
    then impossibleRule
    else propField r
  FNotShareL c r _ → if closureState c == InClosure
    then impossibleRule
    else propField r
  FNotShareR c _ r → if closureState c == InClosure
    then impossibleRule
    else propField r
  FShared c r1 r2 → if closureState c == NotInClosure
    then impossibleRule
    else do
      restrictField $ λe → intersection ty e (closureFields c)
      propField r1
      propField r2
  FAParallelL c1 c2 r _ → case (closureState c1,closureState c2) of
    (NotInClosure,_) → impossibleRule
    (_,InClosure) → impossibleRule
    _ → do
      restrictField $ λe → intersection ty e (closureFields c1)
      propField r
  FAParallelR c1 c2 _ r → case (closureState c1,closureState c2) of
    (_,NotInClosure) → impossibleRule
    (InClosure,_) → impossibleRule
    _ → do
      restrictField $ λe → intersection ty e (closureFields c2)
      propField r
  FAParallelBoth c1 c2 r1 r2 → case (closureState c1,closureState c2) of
    (NotInClosure,_) → impossibleRule
    (_,NotInClosure) → impossibleRule
    _ → do
      restrictField $ λe → intersection ty e (closureFields c1)
      restrictField $ λe → intersection ty e (closureFields c2)
      propField r1
      propField r2
  FNoInterrupt r _ → propField r
  FInterrupt _ r → propField r
  FTimeout r _ → propField r
  FRepAParallel RepAPFailed → impossibleRule
  FRepAParallel x → restrictField $ λe → intersection ty e (repInitials x)
  FRenaming _ _  → return () -- todo: some properagtion for renaming
  FChaos c → restrictField $ λe → intersection ty e (closureFields c)
  FLinkEventL _ r _ → propField r
  FLinkEventR _ _ r → propField r
  FNoException c r _ → if closureState c == InClosure
    then impossibleRule
    else propField r
  FExceptionOccurs c _ r → if closureState c == NotInClosure
```

```
    then impossibleRule
    else propField r
  where
    restrictField :: (FieldSet i → FieldSet i) → PropM i ()
    restrictField fkt = do
      possible ← get
      let restricted = fkt possible
      if Field.null ty restricted
        then impossibleRule
        else put restricted

    fixField :: Field i → PropM i ()
    fixField e = do
      possible ← get
      if member ty e possible
        then put $ singleton ty e
        else impossibleRule

    impossibleRule :: PropM i ()
    impossibleRule = mzero
    closureState :: ClosureState i → ClosureView
    closureState = viewClosureState ty
    closureFields :: ClosureState i → FieldSet i
    closureFields = viewClosureFields ty
    ty = (undefined :: i)

{-
Fix one field in the event.
-}
nextField :: forall i. BF i ⇒ RuleField i → Field i → Search (RuleField i)
nextField rule field = case rule of
  FPrefix p → case prefixStateNext ty p field of
    Just a → return $ FPrefix a
    Nothing → mzero
  FExtChoiceL r p → FExtChoiceL <$> rec r <*> pure p
  FExtChoiceR p r → FExtChoiceR p <$> rec r
  FExtChoice p q
    → joinRepExtChoiceParts
         (nextRepExtChoicePart p field)
         (nextRepExtChoicePart q field)
  FInterleaveL r p → FInterleaveL <$> rec r <*> pure p
  FInterleaveR p r → FInterleaveR p <$> rec r
  FSeqNormal r p → FSeqNormal <$> rec r <*> pure p
  FNotHidden c r → FNotHidden (fc c) <$> rec r
  FNotShareL c r p → FNotShareL (fc c) <$> rec r <*> pure p
  FNotShareR c p r → FNotShareR (fc c) p <$> rec r
  FShared c r1 r2 → FShared (fc c) <$> rec r1 <*> rec r2
  FAParallelL c1 c2 r q
    → FAParallelL (fc c1) (fc c2) <$> rec r <*> pure q
  FAParallelR c1 c2 p r
    → FAParallelR (fc c1) (fc c2) p <$> rec r
  FAParallelBoth c1 c2 r1 r2
    → FAParallelBoth (fc c1) (fc c2) <$> rec r1 <*> rec r2
  FNoInterrupt r q → FNoInterrupt <$> rec r <*> pure q
  FInterrupt p r → FInterrupt p <$> rec r
  FTimeout r q → FTimeout <$> rec r <*> pure q
  FRepAParallel x → return $ FRepAParallel $ repNextField field x
  FRenaming rel p → return $ FRenaming rel p
  FChaos c → return $ FChaos (fc c)
  FLinkEventL rel r q → FLinkEventL rel <$> rec r <*> pure q
  FLinkEventR rel p r → FLinkEventR rel p <$> rec r
  FNoException c r q → FNoException (fc c) <$> rec r <*> pure q
  FExceptionOccurs c p r → FExceptionOccurs (fc c) <$> pure p <*> rec r
  where
    rec r = nextField r field
    ty = (undefined :: i)
    fc c = closureStateNext ty c field

{-
Check constraints after last field and
convert RuleField to RuleEvent.
We must check all constraints here!
-}
lastField :: forall i. BF i
  ⇒ RuleField i → Event.Event i → Search (RuleEvent i)
lastField rule event = case rule of
  FPrefix p → case prefixStateFinalize ty p of
    Nothing → mzero
    Just x → return $ HPrefix event x
  FExtChoiceL r p → ExtChoiceL <$> rec r <*> pure p
  FExtChoiceR p r → ExtChoiceR p <$> rec r
  FExtChoice (Right (p,rp)) (Right (q,rq))
    →       (ExtChoiceL <$> (anyOf rp >>= rec) <*> pure q)
    `mplus`  (ExtChoiceR p <$> (anyOf rq >>= rec) )
  FExtChoice _ _ → error "unreachable: this case is handled by nextField"
  FInterleaveL r p → InterleaveL <$> rec r <*> pure p
  FInterleaveR p r → InterleaveR p <$> rec r
  FSeqNormal r p → SeqNormal <$> rec r <*> pure p
  FNotHidden hidden r → do
    guard_not_inClosure hidden
    NotHidden (restoreClosure hidden) <$> rec r
  FNotShareL c r p → do
    guard_not_inClosure c
    NotShareL (restoreClosure c) <$> rec r <*> pure p
  FNotShareR c p r → do
    guard_not_inClosure c
    NotShareR (restoreClosure c) p <$> rec r
  FShared c r1 r2 → do
    guard_inClosure c
    Shared (restoreClosure c) <$> rec r1 <*> rec r2
  FAParallelL c1 c2 r q → case (inClosure c1,inClosure c2) of
    (True,False) → AParallelL (restoreClosure c1) (restoreClosure c2) <$> rec r <*>
pure q
    _ → mzero
  FAParallelR c1 c2 p r → case (inClosure c1,inClosure c2) of
    (False,True) → AParallelR (restoreClosure c1) (restoreClosure c2) <$> pure p <*>
rec r
    _ → mzero
  FAParallelBoth c1 c2 r1 r2 →  case (inClosure c1,inClosure c2) of
    (True,True) → AParallelBoth (restoreClosure c1) (restoreClosure c2)
                  <$> rec r1 <*> rec r2
    _ → mzero
  FNoInterrupt r q → NoInterrupt <$> rec r <*> pure q
  FInterrupt p r → InterruptOccurs p <$> rec r
  FTimeout r q → TimeoutNo <$> rec r <*> pure q
  FRepAParallel RepAPFailed → mzero
  FRepAParallel x → repToRules event x
  FRenaming rel p → renamingRules rel p event
  FChaos c → if inClosure c
    then return $ ChaosEvent (restoreClosure c) event
    else mzero
  FLinkEventL rel r q → do
    guard $ not $ Event.isInRenamingDomain ty event rel
    LinkEventL rel <$> rec r <*> pure q
  FLinkEventR rel p r → do
    guard $ not $ Event.isInRenamingRange ty event rel
    LinkEventR rel p <$> rec r
  FNoException c r p → do
    guard_not_inClosure c
    NoException (restoreClosure c) <$> rec r <*> pure p
  FExceptionOccurs c p r → do
    guard_inClosure c
    ExceptionOccurs (restoreClosure c) p <$> rec r
  where
    rec r = lastField r event
    ty = (undefined :: i)
    restoreClosure = closureRestore ty
    inClosure = seenPrefixInClosure ty
    guard_inClosure = guard ∘ seenPrefixInClosure ty
    guard_not_inClosure = guard ∘ not ∘ seenPrefixInClosure ty

eventTransitions :: BF i ⇒ Event.EventSet i → Process i → Search (RuleEvent i)
eventTransitions events proc = liftM snd $ computeNextE events proc

computeNextE :: BF i
  ⇒ Event.EventSet i → Process i → Search (Event.Event i, RuleEvent i)
computeNextE events proc = rulePattern events proc ≫= runFields events

runFields :: forall i. BF i
  ⇒ Event.EventSet i → RuleField i → Search (Event.Event i, RuleEvent i)
runFields events r = do
      let baseEvents = closureStateInit ty events
      (chan,next) ← enumField (viewClosureFields ty baseEvents ) r
      (e,final) ← loopFields
         (closureStateNext ty baseEvents chan)
         [chan] -- the accumulator for fields
         next
         (channelLen ty chan -1)
      let event = joinFields ty $ reverse e
      rule ← lastField final event
      return (event,rule)
   where ty = (undefined :: i)

loopFields :: forall i.
  BF i ⇒
     ClosureState i   -- the universe for events
  → [Field i]         -- accumulator for fields
  → RuleField i       -- current rule
  → Int               -- number fields left in prefix
  → Search ([Field i], RuleField i)
loopFields _ eventAcc rule 0 = return (eventAcc, rule)
loopFields closureState eventAcc rule n = do
      (f,next) ← enumField (viewClosureFields ty closureState) rule
      loopFields
        (closureStateNext ty closureState f)
        (f:eventAcc)
        next
        (n-1)
   where ty = (undefined :: i)

enumField :: forall i. BF i ⇒ FieldSet i → RuleField i → Search (Field i, RuleField i)
enumField top r = case execStateT (propField r) top of
      Just s → do
        f ← anyOf $ fieldSetToList ty s
        nr ← nextField r f
        return (f ,nr )
      Nothing → mzero
  where ty = (undefined :: i)

tauTransitions :: forall i. BF i ⇒ Process i → Search (RuleTau i)
tauTransitions proc = case proc of
  SwitchedOff p → tauTransitions $ switchOn p
--  SwitchedOff p → mzero
--  SwitchedOff p → return $ TraceSwitchOn $ switchOn p
  Prefix {} → mzero
  ExternalChoice p q
    →       (ExtChoiceTauL <$> tauTransitions p <*> pure q)
     `mplus` (ExtChoiceTauR p <$> tauTransitions q)
  InternalChoice p q
    →       (return $ InternalChoiceL p q)
     `mplus` (return $ InternalChoiceR p q)
  Interleave p q
    →       (InterleaveTauL <$> tauTransitions p <*> pure q)
     `mplus` (InterleaveTauR p <$> tauTransitions q)
     `mplus` (InterleaveTickL <$> tickTransitions p <*> pure q)
     `mplus` (InterleaveTickR p <$> tickTransitions q)
  Interrupt p q
    →       (InterruptTauL <$> tauTransitions p <*> pure q)
     `mplus` (InterruptTauR p <$> tauTransitions q)
  Timeout p q
    →       (TimeoutTauR <$> tauTransitions p <*> pure q)
     `mplus` (return $ TimeoutOccurs p q)
  Sharing p c q
    →       (ShareTauL c <$> tauTransitions p <*> pure q)
     `mplus` (ShareTauR c p <$> tauTransitions q)
     `mplus` (ShareTickL c <$> tickTransitions p <*> pure q)
     `mplus` (ShareTickR c p <$> tickTransitions q)
  AParallel pc qc p q
    →       (AParallelTauL pc qc <$> tauTransitions p <*> pure q)
     `mplus` (AParallelTauR pc qc p <$> tauTransitions q)
     `mplus` (AParallelTickL pc qc <$> tickTransitions p <*> pure q)
     `mplus` (AParallelTickR pc qc p <$> tickTransitions q)
  Seq p q
    →        (SeqTau <$> tauTransitions p <*> pure q)
      `mplus` (SeqTick <$> tickTransitions p <*> pure q)
  Hide hidden p → (do
    rule ← (eventTransitions hidden p)
    return $ Hidden hidden rule)
   `mplus` (HideTau hidden <$> tauTransitions p)
  Stop → mzero
  Skip → mzero
  Omega → mzero
  AProcess _n → mzero
  RepAParallel _ → mzero -- TODO ! tau for replicated AParallel
  Renaming rel p → RenamingTau rel <$> tauTransitions p
  Chaos c → return $ ChaosStop c
  LinkParallel rel p q
    →        (LinkTauL rel <$> tauTransitions p <*> pure q)
      `mplus` (LinkTauR rel p <$> tauTransitions q)
      `mplus` (LinkTickL rel <$> tickTransitions p <*> pure q)
      `mplus` (LinkTickR rel p <$> tickTransitions q)
      `mplus` mkLinkedRules rel p q
  Exception c p q → mzero -- TODO

tickTransitions :: BL i ⇒ Process i → Search (RuleTick i)
tickTransitions proc = case proc of
  SwitchedOff p → tickTransitions $ switchOn p
  Prefix {} → mzero
  ExternalChoice p q
    →       (ExtChoiceTickL <$> tickTransitions p <*> pure q)
     ‘mplus‘ (ExtChoiceTickR p <$> tickTransitions q)
  InternalChoice _p _q → mzero
  Interleave Omega Omega → return $ InterleaveOmega
  Interleave _ _ → mzero
  Interrupt p q → InterruptTick <$> tickTransitions p <*> pure q
  Timeout p q → TimeoutTick <$> tickTransitions p <*> pure q
  Sharing Omega c Omega → return $ ShareOmega c
  Sharing _ _ _ → mzero
  AParallel c1 c2 Omega Omega → return $ AParallelOmega c1 c2
  AParallel _ _ _ _ → mzero
  RepAParallel l → if all (isOmega ∘ snd) l
    then return $ RepAParallelOmega $ map fst l
    else mzero
  Seq _p _q → mzero
  Hide c p → HiddenTick c <$> tickTransitions p
  Stop → mzero
  Skip → return $ SkipTick
  Omega → mzero
  AProcess _n → mzero
  Renaming rel p → RenamingTick rel <$> tickTransitions p
  Chaos _ → mzero
  LinkParallel rel Omega Omega → return $ LinkParallelTick rel
  LinkParallel _ _ _ → mzero
  Exception c p q → mzero -- TODO

type RepAPProc i = (ClosureState i, Process i, [([Field.Field i], RuleEvent i)])
                                     -- why not do this field wise ^
data RepAP i
  = RepAP {
      repInitials :: FieldSet i
     ,repProcs :: [RepAPProc i]
     }
  | RepAPFailed

instance Show (RepAP i) where show _ = "RepAP"

initRepAParallel :: forall i. BF i
  ⇒ [(Event.EventSet i, Process i)]
  → RepAP i
initRepAParallel l = RepAP {
   repInitials = joinInitials ln
  ,repProcs = ln
  }
  where
    ty = (undefined :: i)
    ln = map mkLn l
    mkLn :: (Event.EventSet i, Process i) → RepAPProc i
    mkLn (closure,p)
      = (closureStateInit ty closure
        ,p
        ,map (first (splitFields ty)) $ runSearch $ computeNextE closure p)

joinInitials :: forall i. BF i
  ⇒ [RepAPProc i]
  → FieldSet i
joinInitials l= fieldSetFromList ty $ concatMap jf l where
  jf (_,_,a) = mapMaybe il a
  il ([],_) = Nothing
  il (h:_,_) = Just h
  ty = (undefined :: i)
```

```
repNextField :: forall i. BF i
  ⇒ Field i → RepAP i → RepAP i
repNextField _ RepAPFailed = RepAPFailed
repNextField field x = RepAP {
   repInitials = joinInitials newProcs
  ,repProcs = newProcs
  }
  where
    ty = (undefined :: i)
    newProcs :: [RepAPProc i]
    newProcs = map filterRules $ repProcs x
    filterRules :: RepAPProc i → RepAPProc i
    filterRules (closure, p, rules)
      = (closureStateNext ty closure field, p, mapMaybe nextR rules )
    nextR ([], _r) = Nothing
    nextR (h:t, r) | fieldEq ty field h = Just (t,r)
    nextR _ = Nothing

repToRules :: forall i. BF i
  ⇒ Event.Event i
  → RepAP i
  → Search (RuleEvent i)
repToRules event ra = do
  parts ← mapM mkPart $ repProcs ra
  if all isLeft parts
    then mzero
    else return $ RepAParallelEvent parts
  where
    mkPart :: (ClosureState i, Process i, [([Field.Field i], RuleEvent i)])
      → Search (EventRepAPart i)
    mkPart (closure, origProc, []) = do
      guard (not $ inClosure closure)
      return $ Left (restoreClosure closure, origProc)
    mkPart (closure, _origProc, (map snd → rules)) = do
      r ← anyOf rules
      return $ Right (restoreClosure closure, r)
    restoreClosure = closureRestore ty
    inClosure = seenPrefixInClosure ty
    ty = (undefined :: i)
    isLeft (Left _) = True
    isLeft _ = False

{-
  todo : special cases for injective and relational renamings
-}
renamingRules :: forall i. BF i
  ⇒ Event.RenamingRelation i
  → Process i
  → Event.Event i
  → Search (RuleEvent i)
renamingRules rel proc event = do
    fromEvent ← anyOf $ Event.preImageRenaming ty rel event
    rule ← eventTransitions (Event.singleEventToClosureSet ty fromEvent) proc
    return $ Rename rel event rule
  `mplus` (do
    guard $ not $ Event.isInRenamingDomain ty event rel
    -- here we could callback on enumNext !
    rule ← eventTransitions (Event.singleEventToClosureSet ty event) proc
    return $ RenameNotInDomain rel rule)
  where
    ty = (undefined :: i)
```

```
{-
We just enumerate everything,
very inefficient!
-}
mkLinkedRules :: forall i. BF i
   ⇒ Event.RenamingRelation i
   → Process i
   → Process i
   → Search (RuleTau i)
mkLinkedRules rel p q = do
  (e1, r1) ← rules1
  (e2, r2) ← rules2
  guard $ Event.isInRenaming ty rel e1 e2
  return $ LinkLinked rel r1 r2
  where
    rules1 :: Search (Event.Event i, RuleEvent i)
    rules1 = rules (Event.getRenamingDomain ty rel) p
    rules2 = rules (Event.getRenamingRange ty rel) q
    rules :: [Event.Event i] → Process i → Search (Event.Event i, RuleEvent i)
    rules s proc = do
      e ← anyOf s
      -- Use EnumNext instead!
      computeNextE (Event.singleEventToClosureSet ty e) proc
    ty = (undefined :: i)


type RepExtChoicePart i = Either (Process i) (Process i,[RuleField i])

initRepExtChoicePart :: forall i. BF i
  ⇒ Event.EventSet i → Process i → RepExtChoicePart i
initRepExtChoicePart events p
  = if List.null rules
    then Left p
    else Right (p,rules)
  where rules = runSearch $ rulePattern events p

{-
nextRepExtChoicePart may call nextField with invalid fields.
nextRepExtChoicePart is only an approximation, it might return invalid rules.
-}
nextRepExtChoicePart :: forall i. BF i
  ⇒ RepExtChoicePart i → Field i → RepExtChoicePart i
nextRepExtChoicePart (Left p) _ = (Left p)
nextRepExtChoicePart (Right (p,rules)) field
{-
This is an error, we cannot rely on nextField to check the constraints
nextField might return invalid rules
-}
  = if List.null newRules
    then Left p
    else Right (p,newRules)
  where newRules = runSearch $ msum $ map (flip nextField field) rules

joinRepExtChoiceParts :: forall i. BF i
  ⇒ RepExtChoicePart i → RepExtChoicePart i → Search (RuleField i)
joinRepExtChoiceParts l r = case (l,r) of
  (Left _,Left _) → mzero
  (Right (_,rules), Left q) → FExtChoiceL <$> anyOf rules <*> pure q
  (Left p, Right (_,rules)) → FExtChoiceR p <$> anyOf rules
  (Right _,Right _) → return $ FExtChoice l r
```

B.1.9 Eval Function

```haskell
---------------------------------------------------------------------------
-- |
-- Module      :  CSPM.Interpreter.Eval
-- Copyright   :  (c) Fontaine 2009 - 2011
-- License     :  BSD
--
-- Maintainer  :  Fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- The main eval function of the Interpreter.
--
---------------------------------------------------------------------------
{-# LANGUAGE ViewPatterns #-}
{-# LANGUAGE BangPatterns #-}
module CSPM.Interpreter.Eval
(
  eval
 ,runEM
 ,getSigma
 ,evalBool
 ,evalOutField
 ,evalFieldSet
 ,evalProcess
 ,evalModule
)
where

import qualified CSPM.CoreLanguage as Core

import Language.CSPM.AST as AST hiding (Bindings)

import CSPM.Interpreter.Types as Types
import CSPM.Interpreter.Bindings as Bindings
import CSPM.Interpreter.PatternMatcher
import CSPM.Interpreter.Hash as Hash
import CSPM.Interpreter.SSet as SSet
import CSPM.Interpreter.ClosureSet as ClosureSet
import CSPM.Interpreter.Renaming as Renaming

import Data.Digest.Pure.HashMD5 as HashClass

import Control.Arrow
import Control.Monad.Reader as Reader
import Control.Monad.State.Strict
--import Control.Monad hiding (guard)
import qualified Data.Set as Set
import Data.Set (Set)
import qualified Data.IntMap as IntMap
import Data.IntMap (IntMap)
import qualified Data.List as List

-- | Evaluate an expression in an envirionment.
runEval :: Env → AST.LExp → Value
runEval env expr = runEM (eval expr) env

-- | Run the 'EM' monad with a given envirionment.
runEM  :: EM x → Env → x
runEM action env = Reader.runReader (unEM action) env
```

```
runEnv :: Env → EM x → x
runEnv env action = Reader.runReader (unEM action) env

-- | Evaluate an expression in the 'EM' monad.
eval :: LExp → EM Value
eval expr = case unLabel expr of
  Var v → lookupIdent v
  IntExp i → return $ VInt i
  SetExp (unLabel → RangeOpen _ ) _
    → throwFeatureNotImplemented "open sets" $ Just $ srcLoc expr
  SetExp r Nothing → evalRange r ≫= return ∘ VSet ∘ Set.fromList
  SetExp r (Just comps) → do
    l ← evalSetComp ret comps
    return $ VSet l
    where ret = evalRange r ≫= return ∘ Set.fromList
  ListExp r Nothing → liftM VList $ evalRange r
  ListExp r (Just comps) → liftM VList $ evalListComp (evalRange r) comps
  ClosureComprehension (el, comps) → do
    l ← evalListComp (mapM eval el) comps
    ClosureSet.mkEventClosure l ≫= return ∘ VClosure
  LetI decls freenames e → do
    env ← getEnv
    let digest = closureDigest expr env freenames
    return $ runEval (processDeclList digest env decls) e
  Ifte cond t e → do
    c ← evalBool cond
    if c then eval t else eval e
  CallFunction fkt args → do
    f ← eval fkt
    parameter ← mapM eval $ concat args
    functionCall f parameter
  CallBuiltIn bi [[e]] → builtIn1 bi e
  CallBuiltIn bi [[a,b]] → builtIn2 bi a b
  CallBuiltIn _ _
    → throwScriptError "calling builtIn with worng number of args"
         (Just $ srcLoc expr) Nothing
  Lambda {} → throwInternalError "not expection Constructor Lambda"
                (Just $ srcLoc expr) $ Nothing
  LambdaI freeNames patL body → do
    env ← getEnv
    return $ VFun $ FunClosure {
       getFunCases = [FunCaseI patL body]
      ,getFunEnv = env
      ,getFunArgNum = length patL
      ,getFunId = closureDigest expr env freeNames
      }
  Stop → return  $ VProcess $ Core.stop
  Skip → return  $ VProcess $ Core.skip
  CTrue  → return $ VBool True
  Events → liftM VClosure evalAllEvents
  CFalse → return $ VBool False
  BoolSet → return $ VSet $ Set.fromList [VBool True,VBool False]
{-
  Many prob test contain unboundet INT
  IntSet → return $ VAllInts
-}
  IntSet → return $ VSet $ Set.fromList $ map VInt [0..100] --ToDo: Fix this !!
  TupleExp l → mapM eval l ≫= return ∘ VTuple
  Parens e → eval e
  AndExp a b → do
    av ← evalBool a
    if av then eval b else return $ VBool False
```

```
OrExp a b → do
  av ← evalBool a
  if av then return $ VBool True else eval b
NotExp e → evalBool e ≫= return ∘ VBool ∘ not
NegExp e → evalInt e ≫= return ∘ VInt ∘ negate
Fun1 bi e → builtIn1 bi e
Fun2 bi a b → builtIn2 bi a b
DotTuple l → mapM eval l ≫= return ∘ VDotTuple ∘ concatMap flatTuple
  where
    flatTuple (VDotTuple x ) = x
    flatTuple x = [x]
Closure l → mapM eval l ≫= ClosureSet.mkEventClosure ≫= return ∘ VClosure
ProcSharing s a b
 → liftM3 Core.sharing
     (switchedOffProc a)
     (evalClosureExp s)
     (switchedOffProc b)
    ≫= return ∘ VProcess
ProcAParallel aLeft aRight pLeft pRight
  → liftM4 Core.aparallel
      (evalClosureExp aLeft)
      (evalClosureExp aRight)
      (switchedOffProc pLeft)
      (switchedOffProc pRight)
    ≫= return ∘ VProcess
ProcLinkParallel l p q
  → liftM3 Core.linkParallel
      (evalLinkList l)
      (switchedOffProc p)
      (switchedOffProc q)
     ≫= return ∘ VProcess
ProcRenaming rlist gen proc → do
  pairs ← case gen of
    Nothing → mapM evalRenaming rlist
    Just gens → evalListComp (mapM evalRenaming rlist ) $ unLabel gens
  p ← switchedOffProc proc
  return $ VProcess $ Core.renaming (toRenaming pairs) p
  where
    evalRenaming :: LRename → EM (Value,Value)
    evalRenaming (unLabel → Rename a b) = liftM2 (,) (eval a) (eval b)
ProcRepSequence comp p
  → evalProcCompL p comp ≫= return ∘ VProcess ∘ Core.repSeq
ProcRepInternalChoice comp p
  → evalProcCompS p comp ≫= return ∘ VProcess ∘ Core.repInternalChoice
ProcRepExternalChoice comp p
  → evalProcCompS p comp ≫= return ∘ VProcess ∘ Core.repExternalChoice
ProcRepInterleave comp p
  → evalProcCompS p comp ≫= return ∘ VProcess ∘ Core.repInterleave
ProcRepAParallel comp c p
  → evalListComp ret (unLabel comp)
       ≫= return ∘ VProcess ∘ Core.repAParallel
  where ret = do { x ← evalClosureExp c; y ← switchedOffProc p; return [(x,y)]}
ProcRepLinkParallel comp link p
  → liftM2 Core.repLinkParallel
      (evalLinkList link)
      (evalProcCompL p comp)
     ≫= return ∘ VProcess
ProcRepSharing comp closure p → do
  l ← evalProcCompS p comp
  c ← evalClosureExp closure
  return $ VProcess $ Core.repSharing c l
PrefixI free chan fields body → do
```

```
    env ← getEnv
    return $ VProcess $ Core.prefix $ PrefixState {
        prefixEnv = env
       ,prefixFields = chanOut:fields
       ,prefixBody = body
       ,prefixRHS = throwInternalError "prefixRHS undefiend" (Just $ srcLoc expr) Nothing
       ,prefixDigest = closureDigest body env free
       ,prefixPatternFailed = False
     }
      where chanOut = setNode chan $ OutComm chan
  ExprWithFreeNames {}
    → throwInternalError "didn't expect ExprWithFreeNames" (Just $ srcLoc expr) Nothing
  _ → throwFeatureNotImplemented "hit catch-all case of eval function"
        $ Just $ srcLoc expr

evalRange :: LRange → EM [Value]
evalRange r = case unLabel r of
  RangeEnum l → mapM eval l
  RangeClosed start end → do
    s ← evalInt start
    e ← evalInt end
    return $ map VInt [s..e]
  RangeOpen start → do
    s ← evalInt start
    return $ map VInt [s..]

evalBool :: LExp → EM Bool
evalBool e = do
  v ← eval e
  case v of
    VBool b → return b
    _  → throwTypingError "expecting type Bool" (Just $ srcLoc e) $ Just v


evalInt :: LExp → EM Integer
evalInt e = do
  v ← eval e
  case v of
    VInt b → return b
    _ → throwTypingError "expecting type Integer" (Just $ srcLoc e) $ Just v

evalList :: LExp → EM [Value]
evalList e = do
  v ← eval e
  case v of
    VList l → return l

--  used in mydemos/SimpleRepAlphParallel.csp SYSTEM
    VDataType l → return $ map VConstructor l

--  because of a hack in RepAParalle
    VSet l → return $ Set.toList l
--  because of a hack in evalProcCompS
    VClosure c → return $ Set.toList $ closureToSet c

    _ → throwTypingError "expecting type List" (Just $ srcLoc e) $ Just v

setFromValue :: Value → EM (Set Value)
setFromValue v = case setFromValueM v of
  Just l → return l
  Nothing → throwTypingError "expecting type Set" Nothing $ Just v
```

```
evalSet :: LExp → EM (Set Value)
evalSet e = do
  v ← eval e
  case setFromValueM v of
    Just l → return l
    Nothing → throwTypingError "expecting type Set" (Just $ srcLoc e) $ Just v

setFromValueM :: Value → Maybe (Set Value)
setFromValueM v = case v of
  VSet l → Just l
  VClosure c → Just $ closureToSet c
  VDataType l → Just $ Set.fromList  --used in basin_olderog_bank.csp
                   $ map VConstructor l
  _ → Nothing

evalProcess :: LExp → EM Process
evalProcess e = do
  v ← eval e
  case v of
    VProcess p → return p
    _  → throwTypingError "expecting type Process" (Just $ srcLoc e) $ Just v

evalClosureExp :: LExp → EM ClosureSet
evalClosureExp e = do
  v ← eval e
  case v of
    VClosure x → return x
--    VAllEvents → evalAllEvents
    VSet s → return $ setToClosure s
    _ → throwTypingError "expecting type Event-Closure" (Just $ srcLoc e) $ Just v

listFromValue :: Value → EM [Value]
listFromValue (VList l) = return l
listFromValue v = throwTypingError "expecting type List" Nothing $ Just v

builtIn1 :: LBuiltIn → LExp → EM Value
builtIn1 op expr
  = case lBuiltInToConst op of
    F_Seq → evalSet expr ≫= return ∘ VAllSequences
    F_card → do
      s ← evalSet expr
      return $ VInt $ fromIntegral $ Set.size s
    F_empty  → evalSet expr ≫= return ∘ VBool ∘ Set.null
    F_head   → do
      l ← evalList expr
      case l of
        [] → throwScriptError "head of empty list" (Just $ srcLoc expr) Nothing
        h:_tail → return h
    F_tail   → do
      l ← evalList expr
      case l of
        [] → throwScriptError "tail of empty list" (Just $ srcLoc expr) Nothing
        _head:rest → return $ VList rest
    F_length → evalList expr ≫= return ∘ VInt ∘ fromIntegral ∘ List.length
    F_Len2   → evalList expr ≫= return ∘ VInt ∘ fromIntegral ∘ List.length
    F_Union → do
      s ← evalSet expr
      setList ← mapM setFromValue $ Set.elems s
      return $ VSet $ Set.unions setList
    F_Inter  → do
      s ← evalSet expr
      setList ← mapM setFromValue $ Set.elems s
      case setList of
        [] →  throwScriptError "intersection of empty set of sets"
                  (Just $ srcLoc expr) Nothing
        l  → return $ VSet $ List.foldl1' Set.intersection l
    F_set    → evalList expr ≫= return ∘ VSet ∘ Set.fromList
    F_Set    → do
      s ← evalSet expr
      return $ VSet $ Set.fromList $ map (VSet ∘ Set.fromList )
        $ List.subsequences $ Set.toList s
    F_concat → do
      l ← evalList expr ≫= mapM listFromValue
      return $ VList $ List.concat l
    F_null → do
      l ← evalList expr
      return $ VBool (List.null l)
    F_CHAOS → liftM (VProcess ∘ Core.chaos) $ evalClosureExp expr
    _ → throwInternalError "malformed AST1" (Just $ srcLoc expr) Nothing

builtIn2 :: LBuiltIn → LExp → LExp → EM Value
builtIn2 op a b =
  case lBuiltInToConst op of
    F_union  → setOp Set.union
    F_inter  → setOp Set.intersection
    F_diff   → setOp Set.difference
    F_member → do
      av ← eval a
      s ← evalSet b
      return $ VBool $ Set.member av s
    F_Seq    → throwFeatureNotImplemented "builtIn2 FSeq" Nothing
    F_elem   → do
      av ← eval a
      l  ← evalList b
      return $ VBool $ List.elem av l
    F_Concat → do
      x ← evalList a
      y ← evalList b
      return $ VList $ x ++y
    F_Mult   → intOp (*)
    F_Div    → intOp div
    F_Mod    → intOp mod
    F_Add    → intOp (+)
    F_Sub    → intOp (-)
    F_Eq     → do
      x ← eval a
      y ← eval b
      return $ VBool (x == y)
    F_NEq    → do
      x ← eval a
      y ← eval b
      return $ VBool (x /= y)
    F_GE     → intCmp (≥)
    F_LE     → intCmp (≤)
    F_LT     → intCmp (<)
    F_GT     → intCmp (>)
    F_Sequential → procOp Core.seq
    F_Interrupt  → procOp Core.interrupt
    F_ExtChoice  → do
      x ← switchedOffProc a
      y ← switchedOffProc b
      return $ VProcess $ Core.externalChoice x y
    F_Timeout    → procOp Core.timeout
    F_IntChoice  → do
```

```
      x ← switchedOffProc a
      y ← switchedOffProc b
      return $ VProcess $ Core.internalChoice x y
    F_Interleave → do
      x ← switchedOffProc a
      y ← switchedOffProc b
      return $ VProcess $ Core.interleave x y
    F_Hiding → do
      proc ← switchedOffProc a
      hidden ← evalClosureExp b
      return $ VProcess $ Core.hide hidden proc
    F_Guard → do
      cond ← evalBool a
      if cond then liftM VProcess $ switchedOffProc b
              else return $ VProcess Core.stop
    _ → throwInternalError "malformed AST2"  (Just $ srcLoc op) Nothing
  where
    intOp :: (Integer → Integer → Integer) → EM Value
    intOp o = do
      x ← evalInt a
      y ← evalInt b
      return $ VInt $ o x y
    intCmp :: (Integer → Integer → Bool) → EM Value
    intCmp rel = do
      x ← evalInt a
      y ← evalInt b
      return $ VBool $ rel x y
    setOp :: (Set Value → Set Value → Set Value) → EM Value
    setOp o = do
      x ← evalSet a
      y ← evalSet b
      return $ VSet $ o x y
    procOp :: (Process → Process → Process) → EM Value
    procOp o = do
      x ← switchedOffProc a
      y ← switchedOffProc b
      return $ VProcess $ o x y

-- | Process a module and return the top-level envirionment.
evalModule :: Module INT → Env
evalModule m
  = processDeclList (hs "TopLevelEnvirionment") emptyEnvirionment
      $ AST.moduleDecls m

type DeclM x = ReaderT (Digest,Env) (State (Bindings, IntMap Digest)) x

processDeclList :: Digest → Env → [LDecl] → Env
processDeclList digest oldEnv decls =
  let
    (newBinds,newDigests)
       = execState action' (getLetBindings oldEnv, letDigests oldEnv)
    action :: DeclM ()
    action  = mapM_ processDecl decls
    action' = runReaderT action (digest,newEnv)
    newEnv  = oldEnv { letBindings = newBinds, letDigests = newDigests}
  in newEnv

bindIdentM :: LIdent → Value → DeclM ()
bindIdentM i v = do
  d ← asks fst
  modify $ λ(values,digests) →
    (bindIdent i v values
    ,IntMap.insert (identId i) (HashClass.mixInt d $ identId i) digests)

processDecl :: LDecl → DeclM ()
processDecl decl = do
  case unLabel decl of
    PatBind pat expr → do
      finalEnv ← asks snd
      let rhs = runEval finalEnv expr  -- evaluate the righthand side
      modify $ first $ λoldBinds → tryMatchLazy oldBinds pat rhs
      digest ← asks fst
      forM_ (boundNames pat) $ λi → modify $ second
        $ IntMap.insert (identId i) (HashClass.mixInt digest $ identId i)
    FunBind i cases → do
        finalEnv ← asks snd
        digest ← asks fst
        bindIdentM i $ VFun $ FunClosure {
          getFunCases = cases
         ,getFunEnv = finalEnv
         ,getFunArgNum = length $ casePattern $ head cases
         ,getFunId  = mixInt digest $ AST.unNodeId $ AST.nodeId decl
         }
        where
          casePattern (FunCaseI pl _ ) = pl
          casePattern _ = throwInternalError "unexpected FunCase in AST"
                           (Just $ srcLoc i) Nothing
    Assert {} → return ()
    Transparent names →  forM_ names $ λn → bindIdentM n cspIdentityFunction
    SubType tname constrList → do
{-
       subtypes are like data types except that we do not bind the constructs
       todo : check subtype declaration is correct, i.e. it really declares subtype
-}
       constrs ← mapM (constrDecl False) constrList
       bindIdentM tname (VDataType constrs )
    DataType tname constrList → do
       constrs ← mapM (constrDecl True) constrList
       bindIdentM tname (VDataType constrs )
    NameType tname t → do
      finalEnv ← asks snd
      bindIdentM tname (VNameType $ runEnv finalEnv $ evalTypeDef t)
    Print _expr → return ()
    AST.Channel idList t → do
      finalEnv ← asks snd
      forM_ idList $ λi → bindIdentM i $ VChannel $ Types.Channel {
              chanId = AST.uniqueIdentId $ AST.unUIdent $ unLabel i
             ,chanName = AST.realName $ AST.unUIdent $ AST.unLabel i
             ,chanLen = case t of
                Nothing → 1
                Just ty → case unLabel ty of
                  TypeTuple _l → 2
                  TypeDot l  → length l+1
             ,chanFields = case t of
                Nothing → []
                Just l → runEnv finalEnv $ evalTypeDef l
             }

constrDecl :: Bool → LConstructor → DeclM Types.Constructor
constrDecl performBinding (unLabel → AST.Constructor ident td) = do
  finalEnv ← asks snd
  let
    cl = case td of
      Nothing → []
      Just l → runEnv finalEnv $ evalTypeDef l

    constr = Types.Constructor
               (AST.uniqueIdentId $ AST.unUIdent $ unLabel ident)
               (AST.realName $ AST.unUIdent $ unLabel ident)
               cl
  when performBinding $ bindIdentM ident $ VConstructor constr
  return constr

evalTypeDef :: LTypeDef → EM [FieldSet] -- ← this is too restrictive ?
evalTypeDef t = case unLabel t of
  TypeDot l  → mapM evalFieldSet l -- ← meight be a tuple of one
  TypeTuple l → do
    el ← mapM evalFieldSet l
    -- cross-product
    return [SSet.fromList $ map VTuple $ sequence $ map SSet.toList el]

evalFieldSet :: LExp → EM FieldSet
evalFieldSet expr = do
  v ← eval expr
  case v of
    VInt {} → return $ SSet.singleton v
    VChannel {} → return $ SSet.singleton v
    VSet s → return $ SSet.Proper s
-- todo: Fix this when we have ClosureExpressions.
-- todo: This does not work for constructors that have fields.
    VDataType constrList → return $ SSet.fromList $ map VConstructor constrList
    VNameType _ → throwInternalError "nametype not implemented" (Just $ srcLoc expr) $ Just v
    VAllInts → return $ SSet.fromList $ map VInt [0..10] --todo
    _ → throwTypingError "evalFieldSet" (Just $ srcLoc expr) $ Just v

switchedOffProc :: LExp → EM Process
switchedOffProc (unLabel → ExprWithFreeNames free expr) = do
  env ← getEnv
  return $ Core.switchedOff $ SwitchedOffProc {
    switchedOffDigest = (closureDigest expr env free)
   ,switchedOffExpr = expr
   ,switchedOffProcess = runEM (evalProcess expr) env
    }
switchedOffProc expr
 = throwInternalError "cannot determine free variables" (Just $ srcLoc expr) Nothing

evalOutField :: LExp → EM Field
evalOutField expr = do
  v ← eval expr
  case v of
    VInt {} → return v
    VChannel {} → return v
    VConstructor {} → return v
    VTuple {} → return v
    VDotTuple {} → return v -- todo : Fix for genric buffers
    VBool {} → return v
{-
todo: Dupport lists and sets as channel fields.
Write test for VSet and VList.
-}
    VSet {} → return v
    VList {} → return v

    _ → throwTypingError "Eval.hs : evalOutField" (Just $ srcLoc expr) $ Just v
```

```
{- redo this: Most procComprehensions work on sets ! -}
evalProcCompL :: LExp → LCompGenList → EM [Process]
evalProcCompL p comp = evalListComp ret $ unLabel comp
  where
    ret = do
      r ← switchedOffProc p
      return [r]

{-
fdr does not remove duplicates from replicatesProc compostions,
see examples/CSP/FDRFeatureTests/ReplicatedInterleaveSetDef.csp
-}
evalProcCompS :: LExp → LCompGenList → EM [Process]
evalProcCompS = evalProcCompL
{-
evalProcCompS p comp
  =     (evalSetComp ret $ unLabel comp)
    ≫= (mapM processFromValue) ∘ Set.toList
  where
{-
We intermediateley wrap processes with VProcess.
If we make evalSetComp polymorphic we get the following error
src/Language/CSPM/Interpreter/Eval.hs:536:0:
    Contexts differ in length
      (Use -XRelaxedPolyRec to allow this)
-}
    ret = switchedOffProc p ≫= return ∘ Set.singleton ∘ VProcess
-}

evalListComp :: EM [x] → [LCompGen] → EM [x]
evalListComp ret [] = ret
evalListComp ret (h:t) = case unLabel h of
  Guard g → do
    b ← evalBool g
    if b then evalListComp ret t
         else return []
  Generator pat gen → do
    list ← evalList gen
    rets ← mapM (evalCompPat pat) list
    return $ concat rets
  where
    evalCompPat pat val = do
      e ← getEnv
      case tryMatchStrict (getArgBindings e) pat val of
        Nothing → return []
        Just newBinds
          → return $ runEM
               (evalListComp ret t)
               (setArgBindings e newBinds)

evalSetComp :: EM (Set Value) → [LCompGen] → EM (Set Value)
evalSetComp ret [] = ret
evalSetComp ret (h:t) = case unLabel h of
    Guard g → do
      b ← evalBool g
      if b then evalSetComp ret t
           else return Set.empty
    Generator pat gen → do
      set ← evalSet gen
      rets ← mapM (evalCompPat pat) $ Set.elems set
      return $ Set.unions rets
    where
```

```
      evalCompPat pat val = do
        e ← getEnv
        case tryMatchStrict (getArgBindings e) pat val of
          Nothing → return Set.empty
          Just newBinds
            → return $ runEM
                 (evalSetComp ret t)
                 (setArgBindings e newBinds)

evalAllEvents :: EM ClosureSet
evalAllEvents = do
  channels ← lookupAllChannels
  ClosureSet.mkEventClosure $ map VChannel channels

getSigma :: Env → Sigma
getSigma = runEM evalAllEvents

cspIdentityFunction :: Value
cspIdentityFunction = VFun $ FunClosure {
   getFunCases = [funCase]
  ,getFunEnv = emptyEnvirionment
  ,getFunArgNum = 1
  ,getFunId = Hash.hash "cspIdentityFunction"
  }
  where
    funCase = FunCaseI [ labeled $ VarPat someId] (labeled $ Var someId)
    someId = labeled $ UIdent $ UniqueIdent {
      uniqueIdentId = -1
     ,bindingSide = e
     ,bindingLoc = e
     ,idType = e
     ,realName = e
     ,newName = e
     ,prologMode = e
     ,bindType = NotLetBound }
    e = throwInternalError "use identityFunction magic constants" Nothing Nothing

evalLinkList :: LLinkList → EM RenamingRelation
evalLinkList l = case unLabel l of
  LinkList x → liftM toRenaming $ mapM evalLink x
  LinkListComprehension gen links
    → liftM toRenaming $ evalListComp (mapM evalLink links ) gen
  where
    evalLink :: LLink → EM (Value,Value)
    evalLink (unLabel → Link a b) = liftM2 (,) (eval a) (eval b)

functionCall :: Value → [Value] → EM (Value)
functionCall v arguments = case v of
  VFun fkt → callFkt fkt arguments
  VPartialApplied fkt oldArgs → callFkt fkt (oldArgs ++ arguments)
  f → throwTypingError "calling non-function" Nothing $ Just f
  where
    tryFunCases :: [FunCase] → [Value] → Env → Value
    tryFunCases [] _ _ = throwPatternMatchError "no matching function case" Nothing
    tryFunCases ((FunCaseI parameter fktBody) : moreCases) args env =
      case matchList parameter args (getArgBindings env) of
        Just newBinds → runEval (setArgBindings env newBinds) fktBody
        Nothing → tryFunCases moreCases args env
    tryFunCases (FunCase {} : _) _ _
      = throwInternalError "not expecting FunCase-Constructor" Nothing Nothing

    matchList :: [LPattern] → [Value] → Bindings → Maybe Bindings
    matchList patList valList env
      = foldM (λe (pat,val) → tryMatchStrict e pat val)
         env (zip patList valList)

{-
  Going from
  callFkt fkt args = return $ tryFunCases (getFunCases fkt) args (getFunEnv fkt)
  to the version which supports partial application
  costs approx. 17 % in the fibonacci -example.
-}
    callFkt :: FunClosure → [Value] → EM Value
    callFkt fkt args
       = case compare haveArgs needArgs of
           EQ → return $ tryFunCases (getFunCases fkt) args (getFunEnv fkt)
           GT → do
             f2 ← callFkt fkt $ take needArgs args
             functionCall f2 $ drop needArgs args
           LT → return $ VPartialApplied fkt args
       where
         haveArgs = length args
         needArgs = getFunArgNum fkt
```

B.1.10 Abstract Syntax Tree

```
-----------------------------------------------------------------------------
-- |
-- Module      :  Language.CSPM.AST
-- Copyright   :  (c) Fontaine 2008 - 2011
-- License     :  BSD3
--
-- Maintainer  :  Fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- This module defines an Abstract Syntax Tree for CSPM.
-- This is the AST that is computed by the parser.
-- For historical reasons, it is rather unstructured.

{-# LANGUAGE DeriveDataTypeable, GeneralizedNewtypeDeriving #-}
{-# LANGUAGE EmptyDataDecls, RankNTypes #-}
{-# LANGUAGE RecordWildCards #-}
module Language.CSPM.AST
where

import Language.CSPM.Token
import Language.CSPM.SrcLoc (SrcLoc(..))

import Data.Typeable (Typeable)
import Data.Generics.Basics (Data)
import Data.Generics.Instances ()
import Data.IntMap (IntMap)
import Data.Map (Map)
import Data.Array.IArray

type AstAnnotation x = IntMap x
type Bindings = Map String UniqueIdent
type FreeNames = IntMap UniqueIdent

newtype NodeId = NodeId {unNodeId :: Int}
  deriving (Eq, Ord, Show, Enum, Ix, Typeable, Data)

mkNodeId :: Int → NodeId
mkNodeId = NodeId

data Labeled t = Labeled {
    nodeId :: NodeId
   ,srcLoc  :: SrcLoc
   ,unLabel :: t
   } deriving (Eq, Ord, Typeable, Data, Show)

-- | Wrap a node with a dummyLabel.
-- todo: Redo we need a specal case in DataConstructor Labeled.
labeled :: t → Labeled t
labeled t = Labeled {
 nodeId  = NodeId (-1)
 ,unLabel = t
 ,srcLoc  = NoLocation
 }

setNode :: Labeled t → y → Labeled y
setNode l n = l {unLabel = n}

type LIdent = Labeled Ident

data Ident
  = Ident  {unIdent :: String}
  | UIdent UniqueIdent
  deriving (Eq, Ord, Show, Typeable, Data)


unUIdent :: Ident → UniqueIdent
unUIdent (UIdent u) = u
unUIdent other = error
  $ "Identifier is not of variant UIdent (missing Renaming) " ++ show other

identId :: LIdent → Int
identId = uniqueIdentId ∘ unUIdent ∘ unLabel

data UniqueIdent = UniqueIdent
  {
   uniqueIdentId :: Int
  ,bindingSide :: NodeId
  ,bindingLoc  :: SrcLoc
  ,idType      :: IDType
  ,realName    :: String
  ,newName     :: String
  ,prologMode  :: PrologMode
  ,bindType    :: BindType
  } deriving (Eq, Ord, Show, Typeable, Data)

data IDType
  = VarID | ChannelID | NameTypeID | FunID
  | ConstrID | DataTypeID | TransparentID
  | BuiltInID
  deriving (Eq, Ord, Show, Typeable, Data)

data PrologMode = PrologGround | PrologVariable
  deriving (Eq, Ord, Show, Typeable, Data)

{- Actually BindType and PrologMode are semantically aquivalent -}
data BindType = LetBound | NotLetBound
  deriving (Eq, Ord, Show, Typeable, Data)

isLetBound :: BindType → Bool
isLetBound x = x==LetBound

data Module a = Module {
   moduleDecls :: [LDecl]
  ,moduleTokens :: Maybe [Token]
  ,moduleSrcLoc :: SrcLoc
  ,moduleComments :: [LocComment]
  ,modulePragmas :: [Pragma]
  } deriving (Eq, Ord, Show, Typeable, Data)

data FromParser deriving Typeable
instance Data FromParser
instance Eq FromParser

castModule :: Module a → Module b
castModule Module {..} = Module {..}

type ModuleFromParser = Module FromParser

type LExp = Labeled Exp
type LProc = LExp --LProc is just a typealias for better readablility

data Exp
  = Var LIdent
  | IntExp Integer
  | SetExp LRange (Maybe [LCompGen])
  | ListExp LRange (Maybe [LCompGen])
  | ClosureComprehension ([LExp],[LCompGen])
  | Let [LDecl] LExp
  | Ifte LExp LExp LExp
  | CallFunction LExp [[LExp]]
  | CallBuiltIn LBuiltIn [[LExp]]
  | Lambda [LPattern] LExp
  | Stop
  | Skip
  | CTrue
  | CFalse
  | Events
  | BoolSet
  | IntSet
  | TupleExp [LExp]
  | Parens LExp
  | AndExp LExp LExp
  | OrExp LExp LExp
  | NotExp LExp
  | NegExp LExp
  | Fun1 LBuiltIn LExp
  | Fun2 LBuiltIn LExp LExp
  | DotTuple [LExp]
  | Closure [LExp]
  | ProcSharing LExp LProc LProc
  | ProcAParallel LExp LExp LProc LProc
  | ProcLinkParallel LLinkList LProc LProc
  | ProcRenaming [LRename] (Maybe LCompGenList) LProc
  | ProcException LExp LProc LProc
  | ProcRepSequence LCompGenList LProc
  | ProcRepInternalChoice LCompGenList LProc
  | ProcRepExternalChoice LCompGenList LProc
  | ProcRepInterleave LCompGenList LProc
  | ProcRepAParallel LCompGenList LExp LProc
  | ProcRepLinkParallel LCompGenList LLinkList LProc
  | ProcRepSharing LCompGenList LExp LProc--
  | PrefixExp LExp [LCommField] LProc--
-- Only used in later stages.
  | PrefixI FreeNames LExp [LCommField] LProc
  | LetI [LDecl] FreeNames LExp -- freenames of all localBound names
  | LambdaI FreeNames [LPattern] LExp
  | ExprWithFreeNames FreeNames LExp
  deriving (Eq, Ord, Show, Typeable, Data)

type LRange = Labeled Range
data Range
  = RangeEnum [LExp]
  | RangeClosed LExp LExp
  | RangeOpen LExp
  deriving (Eq, Ord, Show, Typeable, Data)

type LCommField = Labeled CommField
data CommField
  =  InComm LPattern
  | InCommGuarded LPattern LExp
  | OutComm LExp
  deriving (Eq, Ord, Show, Typeable, Data)

type LLinkList = Labeled LinkList
data LinkList
  = LinkList [LLink]
  | LinkListComprehension [LCompGen] [LLink]
  deriving (Eq, Ord, Show, Typeable, Data)

type LLink = Labeled Link
data Link = Link LExp LExp deriving (Eq, Ord, Show, Typeable, Data)

type LRename = Labeled Rename
data Rename = Rename LExp LExp deriving (Eq, Ord, Show, Typeable, Data)

type LBuiltIn = Labeled BuiltIn
data BuiltIn = BuiltIn Const deriving (Eq, Ord, Show, Typeable, Data)

lBuiltInToConst :: LBuiltIn → Const
lBuiltInToConst = h ∘ unLabel where
  h (BuiltIn c) = c

type LCompGenList = Labeled [LCompGen]
type LCompGen = Labeled CompGen
data CompGen
  = Generator LPattern LExp
  | Guard LExp
  deriving (Eq, Ord, Show, Typeable, Data)

type LPattern = Labeled Pattern
data Pattern
  = IntPat Integer
  | TruePat
  | FalsePat
  | WildCard
  | Also [LPattern]
  | Append [LPattern]
  | DotPat [LPattern]
  | SingleSetPat LPattern
  | EmptySetPat
  | ListEnumPat [LPattern]
  | TuplePat [LPattern]
-- ConstrPat is generated by renaming
```

```
  | ConstrPat LIdent
-- This the result of pattern-match-compilation.
  | VarPat LIdent
  | Selectors { --origPat :: LPattern
 -- fixme: This creates an infinite tree with SYB everywehre'
                selectors :: Array Int Selector
               ,idents :: Array Int (Maybe LIdent) }
  | Selector Selector (Maybe LIdent)
  deriving (Eq, Ord, Show, Typeable, Data)

{- A Selector is a path in a Pattern/Expression. -}
data Selector
  = IntSel Integer
  | TrueSel
  | FalseSel
  | SelectThis
  | ConstrSel UniqueIdent
  | DotSel Int Selector
  | SingleSetSel Selector
  | EmptySetSel
  | TupleLengthSel Int Selector
  | TupleIthSel Int Selector
  | ListLengthSel Int Selector
  | ListIthSel Int Selector
  | HeadSel Selector
  | HeadNSel Int Selector
  | PrefixSel Int Int Selector
  | TailSel Selector
  | SliceSel Int Int Selector
  | SuffixSel Int Int Selector
  deriving (Eq, Ord, Show, Typeable, Data)

type LDecl = Labeled Decl
data Decl
  = PatBind LPattern LExp
  | FunBind LIdent [FunCase]
  | Assert LAssertDecl
  | Transparent [LIdent]
  | SubType LIdent [LConstructor]
  | DataType LIdent [LConstructor]
  | NameType LIdent LTypeDef
  | Channel [LIdent] (Maybe LTypeDef)
  | Print LExp
  deriving (Show, Eq, Ord, Typeable, Data)

{-
We want to use                1) type FunArgs = [LPattern]
it is not clear why we used   2) type FunArgs = [[LPattern]].
If 1) works in the interpreter, we will refactor
Renaming, and the Prolog interface to 1).
For now we just patch the AST just before PatternCompilation.
-}
type FunArgs = [[LPattern]]
data FunCase
  = FunCase FunArgs LExp
  | FunCaseI [LPattern] LExp
  deriving (Eq, Ord, Show, Typeable, Data)

type LTypeDef = Labeled TypeDef
data TypeDef
  = TypeTuple [LExp]
  | TypeDot [LExp]
  deriving ( Eq, Ord, Show,Typeable, Data)

type LConstructor = Labeled Constructor
data Constructor
  = Constructor LIdent (Maybe LTypeDef)
  deriving (Eq, Ord, Show, Typeable, Data)

withLabel :: ( NodeId -> a -> b ) -> Labeled a -> Labeled b
withLabel f x = x {unLabel = f (nodeId x) (unLabel x) }

type LAssertDecl = Labeled AssertDecl
data AssertDecl
  = AssertBool LExp
  | AssertRefine   Bool LExp LRefineOp    LExp
  | AssertTauPrio  Bool LExp LTauRefineOp LExp LExp
  | AssertModelCheck Bool LExp LFDRModels (Maybe LFdrExt)
  deriving (Eq, Ord, Show, Typeable, Data)

type LFDRModels = Labeled FDRModels
data FDRModels
  = DeadlockFree
  | Deterministic
  | LivelockFree
  deriving (Eq, Ord, Show, Typeable, Data)

type LFdrExt = Labeled FdrExt
data FdrExt
  = F
  | FD
  | T
  deriving (Eq, Ord, Show, Typeable, Data)

type LTauRefineOp = Labeled TauRefineOp
data TauRefineOp
  = TauTrace
  | TauRefine
 deriving (Eq, Ord, Show, Typeable, Data)

type LRefineOp = Labeled RefineOp
data RefineOp
  = Trace
  | Failure
  | FailureDivergence
  | RefusalTesting
  | RefusalTestingDiv
  | RevivalTesting
  | RevivalTestingDiv
  | TauPriorityOp
  deriving (Eq, Ord, Show, Typeable, Data)

data Const
  = F_true
  | F_false
  | F_not
  | F_and
  | F_or
  | F_union
  | F_inter
  | F_diff
  | F_Union
  | F_Inter
  | F_member
  | F_card
  | F_empty
  | F_set
  | F_Set
  | F_Seq
  | F_null
  | F_head
  | F_tail
  | F_concat -- fix this: Confusing F_Concat.
  | F_elem
  | F_length
  | F_STOP
  | F_SKIP
  | F_Events
  | F_Int
  | F_Bool
  | F_CHAOS
  | F_Concat -- fix this: Confusing F_concat.
  | F_Len2
  | F_Mult
  | F_Div
  | F_Mod
  | F_Add
  | F_Sub
  | F_Eq
  | F_NEq
  | F_GE
  | F_LE
  | F_LT
  | F_GT
  | F_Guard
  | F_Sequential
  | F_Interrupt
  | F_ExtChoice
  | F_IntChoice
  | F_Hiding
  | F_Timeout
  | F_Interleave
  deriving (Eq, Ord, Show, Typeable, Data)

type Pragma = String
type LocComment = (Comment, SrcLoc)
data Comment
  = LineComment String
  | BlockComment String
  | PragmaComment Pragma
  deriving (Eq, Ord, Show, Typeable, Data)
```

B.1.11 Quickcheck

```
-----------------------------------------------------------------------------
-- |
-- Module      :  CSPM.FiringRules.Test.Test
-- Copyright   :  (c) Fontaine 2010
-- License     :  BSD
--
-- Maintainer  :  fontaine@cs.uni-duesseldorf.de
-- Stability   :  experimental
-- Portability :  GHC-only
--
-- QuickCheck tests for the proof tree generators in
-- module CSPM.FiringRules.EnumerateEvents and
-- CSPM.FiringRules.FieldConstraints.
-- These QuickCheck properties check for soundness, completeness
-- and that both proof tree generators yield the same result.
--
----------------------------------------------------------------------------

{-# LANGUAGE StandaloneDeriving,FlexibleInstances #-}
{-# LANGUAGE ScopedTypeVariables #-}
module CSPM.FiringRules.Test.Test
(
  main
)
where

import CSPM.CoreLanguage
import CSPM.CoreLanguage.Event (allEvents)

import CSPM.FiringRules.Rules
import CSPM.FiringRules.Verifier
import CSPM.FiringRules.Test.Mock1
import CSPM.FiringRules.Test.Mock2
import qualified CSPM.FiringRules.EnumerateEventsList as EnumNext
import qualified CSPM.FiringRules.FieldConstraints as FieldNext
import CSPM.FiringRules.HelperClasses

import System.Random
import Test.QuickCheck as QC
import Data.Maybe
import qualified Data.List as List
import qualified Data.Set as Set
import Control.Monad

-- | Run a number of QuickCheck tests (with fixed seed).
main :: IO ()
main = forM_ [1,2,3,4] $ λseed → do
  putStrLn $ "λnλnλnSeed " ++ show seed
  mainDet seed

mainDet :: Int → IO ()
mainDet i = do
  setStdGen $ mkStdGen i
  testAll

testAll :: IO ()
testAll = do
  testMock1
  testMock2
  testFields

testMock1 :: IO ()
testMock1 = do
  putStrLn "testing Mock1"
  quickCheck $ QC.label "generator Tau rules"
    ((isJust ∘ viewRuleTau) :: RuleTau M1 → Bool)
  quickCheck $ QC.label "generator Tick rules"
    ((isJust ∘ viewRuleTick) :: RuleTick M1 → Bool)
  quickCheck $ QC.label "generator Event rules"
    ((isJust ∘ viewRuleEvent) :: RuleEvent M1 → Bool)
  quickCheck $ QC.label "sound enum Tick rules"
    (sound_EnumRuleTick :: RuleTick M1 → Bool)
  quickCheck $ QC.label "sound enum Tau rules"
    (sound_EnumRuleTau :: RuleTau M1 → Bool)
  quickCheck $ QC.label "sound enum Event rules"
    (sound_EnumRuleEvent :: RuleEvent M1 → Bool)
  quickCheck $ QC.label "complete enum Tick rules"
    (complete_enumTickRules :: RuleTick M1 → Bool)
  quickCheck $ QC.label "complete enum Tau rules"
    (complete_enumTauRules :: RuleTau M1 → Bool)
  quickCheck $ QC.label "complete enum Event rules"
    (complete_enumEventRules :: RuleEvent M1 → Bool)

testMock2 :: IO ()
testMock2 = do
  putStrLn "\n\ntesting Mock2\n\n"
  quickCheck $ QC.label "generator Tau rules"
    ((isJust ∘ viewRuleTau) :: RuleTau M2 → Bool)
  quickCheck $ QC.label "generator Tick rules"
    ((isJust ∘ viewRuleTick) :: RuleTick M2 → Bool)
  quickCheck $ QC.label "generator Event rules"
    ((isJust ∘ viewRuleEvent) :: RuleEvent M2 → Bool)
  quickCheck $ QC.label "sound enum Tick rules"
    (sound_EnumRuleTick :: RuleTick M2 → Bool)
  quickCheck $ QC.label "sound enum Tau rules"
    (sound_EnumRuleTau :: RuleTau M2 → Bool)
  quickCheck $ QC.label "sound enum Event rules"
    (sound_EnumRuleEvent :: RuleEvent M2 → Bool)
  quickCheck $ QC.label "complete enum Tick rules"
    (complete_enumTickRules :: RuleTick M2 → Bool)
  quickCheck $ QC.label "complete enum Tau rules"
    (complete_enumTauRules :: RuleTau M2 → Bool)
  quickCheck $ QC.label "complete enum Event rules"
    (complete_enumEventRules :: RuleEvent M2 → Bool)
  quickCheck $ QC.label "enum Event rules ≡ evalEventRules"
    (computeNext_eq_EnumRuleEvent :: RuleEvent M2 → Bool)
  quickCheck $ QC.label "enum Tau rules ≡ symRuleTau"
    (fieldTau :: RuleTau M2 → Bool)
  quickCheck $ QC.label "enum Tick rules ≡ symRuleTick"
    (fieldTick :: RuleTick M2 → Bool)


sound_EnumRuleTick :: CSP1 i ⇒ RuleTick i → Bool
sound_EnumRuleTick r
  = all (checkRule proc ∘ TickRule) $ EnumNext.tickTransitions proc
  where proc = viewProcBefore $ TickRule r

sound_EnumRuleTau :: CSP1 i ⇒ RuleTau i → Bool
sound_EnumRuleTau r
  = all (checkRule proc ∘ TauRule) $ EnumNext.tauTransitions proc
  where proc = viewProcBefore $ TauRule r

sound_EnumRuleEvent :: forall i. CSP1 i ⇒ RuleEvent i → Bool
sound_EnumRuleEvent r
  = all (checkRule proc ∘ EventRule) $ EnumNext.eventTransitions sigma proc
  where
    proc = viewProcBefore $ EventRule r
    sigma = allEvents (undefined :: i)

checkRule :: CSP1 i ⇒ Process i → Rule i → Bool
checkRule proc r
  = case viewRuleMaybe r of
      Nothing → False
      Just (p,_,_) → p ≡ proc

complete_enumTickRules :: CSP1 i ⇒ RuleTick i → Bool
complete_enumTickRules r
  = r ‘List.elem‘ (EnumNext.tickTransitions $ viewProcBefore $ TickRule r)

complete_enumTauRules :: CSP1 i ⇒ RuleTau i → Bool
complete_enumTauRules r
  = r ‘List.elem‘ (EnumNext.tauTransitions $ viewProcBefore $ TauRule r)

complete_enumEventRules :: forall i. CSP1 i ⇒ RuleEvent i → Bool
complete_enumEventRules r
  = r ‘List.elem‘ (EnumNext.eventTransitions sigma $ viewProcBefore $ EventRule r)
  where sigma = allEvents (undefined :: i)

testFields :: IO ()
testFields = do
  putStrLn "λnλnTesting computeNext"
  quickCheck $ QC.label "sound_computeNext"
    (sound_computeNext :: RuleEvent M2 → Bool)

  quickCheck $ QC.label "complete_computeNext"
    (complete_computeNext :: RuleEvent M2 → Bool)

  quickCheck $ QC.label "FieldNext.eventTransitions == EnumNext.eventTransitions"
    (computeNext_eq_EnumRuleEvent :: RuleEvent M2 → Bool)

  quickCheck $ QC.label "FieldNext.tauTransitions == EnumNext.tauTransitions"
    (fieldTau :: RuleTau M2 → Bool)

  quickCheck $ QC.label "FieldNext.tickTransitions == EnumNext.tickTransitions"
    (fieldTick :: RuleTick M2 → Bool)

sound_computeNext :: forall i. CSP2 i ⇒ RuleEvent i → Bool
sound_computeNext r
  = all (checkRule proc ∘ EventRule) $ FieldNext.eventTransitions sigma proc
  where
    proc = viewProcBefore $ EventRule r
    sigma = allEvents (undefined :: i)

complete_computeNext :: forall i. CSP2 i ⇒ RuleEvent i → Bool
complete_computeNext r
  = r ‘List.elem‘ (FieldNext.eventTransitions sigma $ viewProcBefore $ EventRule r)
  where
    sigma = allEvents (undefined :: i)

computeNext_eq_EnumRuleEvent :: forall i. CSP2 i ⇒ RuleEvent i → Bool
computeNext_eq_EnumRuleEvent rule = ruleSet1 == ruleSet2
  where
    ruleSet1 = Set.fromList $ FieldNext.eventTransitions sigma proc
    ruleSet2 = Set.fromList $ EnumNext.eventTransitions sigma proc
    proc = viewProcBefore $ EventRule rule
    sigma = allEvents (undefined :: i)

fieldTau :: forall i. CSP2 i ⇒ RuleTau i → Bool
fieldTau rule = ruleSet1 == ruleSet2
  where
    ruleSet1 = Set.fromList $ EnumNext.tauTransitions proc
    ruleSet2 = Set.fromList $ FieldNext.tauTransitions proc
    proc = viewProcBefore $ TauRule rule

fieldTick :: forall i. CSP2 i ⇒ RuleTick i → Bool
fieldTick rule = ruleSet1 == ruleSet2
  where
    ruleSet1 = Set.fromList $ EnumNext.tickTransitions proc
```

```
ruleSet2 = Set.fromList $ FieldNext.tickTransitions proc
proc = viewProcBefore $ TickRule rule
```

Appendix C

Listings of Benchmarks, Test Cases and Examples

C.1 Pure Functional Benchmarks

The CSP$_M$ code

```
channel out:{0..99}
MAIN = out!(ack(3,0)%100) → STOP

P2 = out!square(2) → out!square(3) → STOP

fib1(x) = if x <2 then 1 else fib1(x-1)+fib1(x-2)

fib2(0) = 1
fib2(1) = 1
fib2(x) = fib2(x-1)+fib2(x-2)

ack ( x, y) =
    if x == 0 then y + 1 else
        if y == 0 then ack (x - 1, 1) else
            ack (x - 1, ack (x, y - 1))

square(x) = x *x

sum(l) = let
  worker(acc,<>) = acc
  worker(acc,<h>^t) = worker (acc+h,t)
  within worker(0,l)

map (f,<>) = <>
map (f,<h>^t) = <f(h)> ^ map(f,t)

smc(n) = sum(map(square,<0..n>))
sum2(l) = let
  worker(acc,l) = if null(l)
      then acc
      else worker (acc + head(l),tail(l))
  within worker(0,l)

map2(f,l) = if null(l)
    then <>
    else <f(head(l))> ^ map2(f,tail(l))
smc2(n) = sum2(map2(square,<0..n>))

primes =
  let
    factors(n) = < m | m ← <2..n-1>, n%m == 0 >
    is_prime(n) = null(factors(n))
  within <n | n ← <2..>, is_prime(n) >

ith (0,l) = head(l)
ith (n,l) = ith(n-1,tail(l))

ithPrime(n) = ith(n-1,primes)
```

The Haskell code

```
module Main
where
import System.CPUTime
import Control.Exception (evaluate)
import Control.Monad

fib1 x = if x <2 then 1 else fib1 (x-1) + fib1 (x-2)

fib2 0 = 1
fib2 1 = 1
fib2 x  = fib2 (x-1) + fib2 (x-2)

ackermann :: Integer → Integer → Integer
ackermann x y
  = if x == 0
      then y + 1
      else if y == 0
        then ackermann (x - 1) 1
        else ackermann (x - 1) (ackermann x (y - 1))

square :: Integer → Integer
square(x) = x *x

mySum :: [Integer] → Integer
mySum l = worker 0 l
  where
    worker acc [] = acc
    worker acc (h:t) = worker (acc + h) t

smc n = mySum $ map square [0..n]

mySum2 :: [Integer] → Integer
mySum2 l = worker 0 l
  where
    worker acc l = if null l
```

```
      then acc
      else worker (acc + head l) (tail l)

myMap :: (a → b) → [a] → [b]
myMap f l = if null l then [] else (f $ head l) : myMap f (tail l)
smc2 n = mySum2 $ myMap square [0..n]

primes :: [Integer]
primes = filter is_prime [2..]
  where
    factors :: Integer → [Integer]
    factors n = [m | m ← [2..n-1] , n ‘mod‘ m == 0 ]
    is_prime :: Integer → Bool
    is_prime = null ∘ factors

ithPrime :: Int → Integer
ithPrime i = primes !! (i-1)

showTime :: Integer → String
showTime a = show (div a 1000000000) ++ "ms"

main = do
  putStrLn "Starting test"
  time_start_test ← getCPUTime
  result ← evaluate $ ack (5,0)
  time_finish_execute ← getCPUTime
  putStrLn $ "Total time :" ++ showTime (time_finish_execute - time_start_test)
  print result
```

The Python code

```
import timeit
import sys
sys.setrecursionlimit(70000)


def fib(n):
  if n < 2:
     return 1
  else:
     return fib(n-1) + fib (n-2)
def ack(n, m):
     if n == 0:
         return m + 1
     else:
       if m == 0:
         return ack(n - 1, 1)
       else:
         return ack(n - 1, ack(n, m - 1))

def square(n): return n*n
def sum(l):
  s=0
  for x in l: s = s+x
  return s
```

```
def smc(x):
  return (sum(map(square,range(0,x+1))))

print("start test")
t = timeit.Timer("res=ack(5,0)","from __main__ import ack;'gc.enable()'")
print(t.timeit(1000)/1000)
```

C.2 Other examples

C.2.1 Simplistic Parser

A simplistic parser implemented with `parsec`.

```
module Parser
where
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Expr

data Exp
  = Sum Exp Exp
  | Diff Exp Exp
  | Prod Exp Exp
  | Quot Exp Exp
  | Neg Exp
  | Equal Exp Exp
  | NEqual Exp Exp
  | Ident String
  | Const Integer
  deriving (Show,Eq)

data Stmt
  = Assign String Exp
  | Print Exp
  | While Exp SBlock
  | If Exp SBlock (Maybe SBlock)
  deriving (Show,Eq)

type SBlock = [Stmt]

whiteSpace :: Parser Char
whiteSpace = space <|> tab <|> newline

skipWs :: Parser a → Parser a
skipWs x = do
  xval ← x
  many whiteSpace
  return xval

lexSym :: String → Parser String
lexSym s = skipWs $ string s

lexKey :: String → Parser ()
lexKey s = try $ skipWs $ do
  string s
  notFollowedBy alphaNum
```

```
parseIdent :: Parser String
parseIdent = skipWs $ do
  h ← letter
  r ← many alphaNum
  return (h:r)

intLit :: Parser Integer
intLit = skipWs $ do
  s ← many1 digit
  return ((read s)::Integer)

baseExp :: Parser Exp
baseExp =
  between (lexSym "(") (lexSym ")") parseExp
  <|> do
    lexSym "-"
    e ← baseExp
    return $ Neg e
  <|> do
    i ← intLit
    return $ Const i
  <|> do
    i ← parseIdent
    return $ Ident i

opList :: [[Operator Char () Exp]]
opList = [
  [ Infix (binOp "*" Prod) AssocLeft
  , Infix (binOp "/" Quot) AssocLeft
  ],
  [ Infix (binOp "+" Sum) AssocLeft
  , Infix (binOp "-" Diff) AssocLeft
  ],
  [ Infix (binOp "==" Equal) AssocLeft
  , Infix (binOp "!=" NEqual) AssocLeft
  ]
 ]

binOp :: String → (Exp → Exp → Exp )
  → Parser (Exp → Exp → Exp)
binOp sym constr= do
  lexSym sym
  return (λ x y → constr x y)

parseExp :: Parser Exp
parseExp = buildExpressionParser opList baseExp

parseStmt :: Parser Stmt
parseStmt =
  do
    lexKey "let"
    ident ← parseIdent
    lexSym "="
    e←parseExp
    return $ Assign ident e
  <|> do
    lexKey "print"
    e ← parseExp
    return $ Print e
  <|> do
    lexKey "while"
    e ← parseExp
    bl ← parseSBlock
    return $ While e bl
  <|> do
    lexKey "if"
    e ← parseExp
    lexKey "then"
    bl1 ← parseSBlock
    ebl ← option Nothing $ do
       lexKey "else"
       bl2 ← parseSBlock
       return $ Just bl2
    return $ If e bl1 ebl

parseSBlock :: Parser SBlock
parseSBlock = do
  lexKey "begin"
  stmtl ← sepBy parseStmt (lexSym ";")
  lexKey "end"
  return stmtl

parsePrg :: Parser SBlock
parsePrg = do
  many whiteSpace
  prg ← parseSBlock
  lexSym "."
  eof
  return prg

testSrc :: String
testSrc =
     "beginλn  let x=10;λn  let y=x∗x;λn  while (x!=0) beginλn"
  ++ "let x=x-1;λn    let y=-(x+y)∗z+4λn  endλnend."

test :: IO ()
test = do
  putStrLn ""
  putStrLn testSrc
  putStrLn ""
  parseTest parsePrg testSrc
```

C.3 CSP$_M$ Testcases

C.3.1 Primes

```
primes =
  let
    factors(n) = < m | m ← <2 ◦ . n-1>, n%m == 0 >
    is_prime(n) = null(factors(n))
  within < n | n ← <2..>, is_prime(n) >

channel p:{1..1000}

take(0,l) = <>
take(n,l) = <head (l)> ^ take(n-1,tail(l))

MAIN = ; x:take(5,primes)@ p!x → SKIP

P2 = p!2 → p!3 → p!5 → p!7 → p!11 → SKIP

assert MAIN [FD= P2
assert P2 [FD= MAIN
```

C.3.2 Mutual Recursive Let

```
channel out:{0,1,2,3}

list = let
    o = <1,2> ^ z
    z = <0,2> ^ o
  within <3> ^ o

take(0,l) = <>
take(n,l) = <head (l)> ^ take(n-1,tail(l))

MAIN = ; x:take(10,list)@ out!x → SKIP
```

C.3.3 A Specification of the Hanoi Puzzle

```
{-
  An version of the Towers of Hanoi using lots of features
  which were not present in FDR 1.4
  JBS 6 March 1995 (based loosely on AWR's version for FDR 1.4)
-}

transparent diamond

n             = 9 -- How many discs

-- Discs are numbered
DISCS         = {1..n}

-- But the pegs are labelled
datatype PEGS = A | B | C

{-
  For a given peg, we can get a new disc or put the
  top disc somewhere else.  We are also allowed to
  to indicate when the peg is full.
-}

channel get, put : DISCS
channel full

-- We are allowed to put any *smaller* disc onto the current stack
allowed(s) = { 1..head(s^<n+1>)-1 }

PEGnil = PEG(<>)
PEG(s) =
  get?d:allowed(s)->PEG(<d>^s)
    []
  not null(s) & put!head(s)->PEG(tail(s))
    []
  length(s) == n & full->PEG(s)

{-
  Now, given a simple peg we can rename it to form each
  of the three physical pegs ('poles') of the puzzle.

  move.d.i.j indicates that disc d moves to pole i from pole j
-}

channel move : DISCS.PEGS.PEGS
channel complete : PEGS

initial(p)   = if p==A then < 1..n > else <>

POLE_A = POLE(A)
POLE_B = POLE(B)
POLE_C = POLE(C)
POLE_Cb = PEG(initial(C)) [[ get.1 <- move.1.C.C]]

POLE(p) =
  PEG(initial(p))
    [[ full <- complete.p,
       get.d <- move.d.p.i,
       put.d <- move.d.i.p | i<- PEGS, i != p, d<-DISCS ]]

{-
  The puzzle is just the three poles, communicating on the
  relevant events: all the moves, and the done/notdone events.
-}

interface(p) = { move.d.i.p, move.d.p.i, complete.p | d<-DISCS, i<-PEGS }

PUZZLE1 = full ->
--replicated alphabet parallel
 ( || p : PEGS @ [ interface(p) ] diamond(POLE(p)) )

PUZZLE =
--replicated alphabet parallel
-- Variation of PUZZLE1; also checks whether compilation works properly
  || p : PEGS @ [ interface(p) ]
 ( PEG(initial(p))
    [[ full ← complete.p,
       get.d ← move.d.p.i,
       put.d ← move.d.i.p | i← PEGS, i != p, d←DISCS ]])


{-
  The puzzle is solved by asserting that C cannot become complete.
  then the trace that refutes the assertion is the solution.
-}

EPUZZLE = (POLE_A [ interface(A) || interface(B) ] POLE_B) [ union(interface(A),interface(B))
interface(C) ] POLE_C

MAIN = PUZZLE

NOTSOLVED = complete?x:{A,B}→ NOTSOLVED [] move?x → NOTSOLVED

-- assert NOTSOLVED [T= PUZZLE
-- assert PUZZLE λ {| complete.A, complete.B, move |} [F= STOP
```

C.3.4 A Model of a Level Crossing Gate

```
-- Model of a level crossing gate for FDR: revised version
-- Illustrating discrete-time modelling using untimed CSP

-- (c) Bill Roscoe, November 1992 and July 1995
-- Revised for FDR 2.11 May 1997

{-
  This file contains a revised version, to coincide with my 1995
  notes, of the level crossing gate example which was the first CSP
  program to use the "tock" model of time.

  The present version has (I think) a marginally better incorporation
  of timing information.
-}

-- Time to compute state space: 58.5 seconds

-- LTL Formulas
-- G F e(enter) → 0.26 secs
-- G F [enter] → FALSE  412.39 secs

-- The tock event represents the passing of a unit of time

channel tock

-- The following are the communications between the controller process and
-- the gate process

datatype GateControl = go_down | go_up | up | down

-- where we can think of the first two as being commands to it, and the
-- last two as being confirmations from a sensor that they are up or down.

channel gate : GateControl

-- For reasons discussed below, we introduce a special error event:

channel error

-- To model the speed of trains, and also the separation of more than one
-- trains, we divide the track into segments that the trains can enter or
-- leave.

Segments = 5    -- the number of segments including the outside one
LastSeg = Segments - 1
TRACKS = {0..LastSeg}
REALTRACKS = {1..LastSeg}

-- Here, segment 0 represents theo outside world, and [1,Segment) actual
-- track segments; including the crossing, which is at

GateSeg==3

-- This model handles two trains

datatype TRAINS = Thomas | Gordon

-- which can move between track segments

channel enter, leave : TRACKS.TRAINS

-- Trains are detected when they enter the first track segment by a sensor,
-- which drives the controller, and are also detected by a second sensor
-- when they leave GateSeg

datatype sensed = in | out

channel sensor : sensed


-- The following gives an untimed description of Train A on track segment j
-- A train not currently in the domain of interest is given index 0.

Train(A,j) =  enter.((j+1)%Segments).A → leave.j.A → Train(A,(j+1)%Segments)

-- There is no direct interference between the trains

Trains = Train(Thomas,0) ||| Train(Gordon,0)

-- The real track segments can be occupied by one train at a time, and each
-- time a train enters segment 1 or leaves GateSeg the sensors fire.

Track(j) =
  let
    Empty   = enter.j?A → if j==1 then sensor.in → Full(A) else Full(A)
    Full(A) = leave.j.A → if j==GateSeg then sensor.out → Empty else Empty
  within Empty

-- Like the trains, the untimed track segments do not communicate with
-- each other

Tracks = ||| j : REALTRACKS @ Track(j)

-- And we can put together the untimed network, noting that since there is
-- no process modelling the outside world there is no need to synchronise
-- on the enter and leave events for this area.

Network = Trains [|{|enter.j, leave.j | j<-REALTRACKS|}|] Tracks

-- We make assumptions about the speed of trains by placing (uniform)
-- upper and lower "speed limits" on the track segments:

-- MinTocksPerSeg = 3 -- make this a parameter to experiment with it
SlowTrain = 4       -- inverse speed parameter, MinTocksPerSegment
NormalTrain = 3
FastTrain = 2

MaxTocksPerSeg = 6

-- The speed regulators express bounds on the times between successive
-- enter events.

SpeedReg(j,MinTocksPerSeg) =
  let
    Empty   = enter.j?A → Full(0) [] tock → Empty
    Full(n) = n <  MaxTocksPerSeg & tock → Full(n+1)
           [] MinTocksPerSeg ≤ n & enter.(j+1)%Segments?A → Empty
  within Empty

-- The following pair of processes express the timing contraint that
-- the two sensor events occur within one time unit of a train entering
-- or leaving the domain.

InSensorTiming = tock → InSensorTiming
             [] enter.1?A → sensor.in → InSensorTiming

OutSensorTiming = tock → OutSensorTiming
              [] leave.GateSeg?A → sensor.out → OutSensorTiming

-- The timing constraints of the trains and sensors are combined into the
-- network as follows, noting that no speed limits are used outside the domain:

SpeedRegs(min) =
  || j : REALTRACKS @ [{|tock, enter.j, enter.(j+1)%Segments|}] SpeedReg(j,min)
-- replicated alphabet parallel now supported



SensorTiming = InSensorTiming [|{tock}|] OutSensorTiming
```

```
NetworkTiming(min) = SpeedRegs(min) [|{|tock, enter.1|}|] SensorTiming

TimedNetwork(min) =
  Network [|{|enter, sensor, leave.GateSeg|}|] NetworkTiming(min)

-- The last component of our system is a controller for the gate, whose duties
-- are to ensure that the gate is always down when there is a train on the
-- gate, and that it is up whenever prudent.

-- Unlike the first version of this example, here we will separate the
-- timing assumptions about how the gate behaves into a separate process.
-- But some timing details (relating to the intervals between sensors
-- firing and signals being sent to the gate) are coded directly into this
-- process, to illustrate a different coding style to that used above:



Controller =
  let
    -- When the gate is up, the controller does nothing until the sensor
    -- detects an approaching train.
    -- In this state, time is allowed to pass arbitrarily, except that the
    -- signal for the gate to go down is sent immediately on the occurrence of
    -- the sensor event.
    ControllerUp = sensor.in -> gate!go_down -> ControllerGoingDown(1)
                [] sensor.out -> ERROR
                [] tock -> ControllerUp
    -- The two states ControllerGoingDown and ControllerDown both keep
    -- a record of how many trains have to pass before the gate may go
    -- up.
    -- Each time the sensor event occurs this count is increased.
    -- The count should not get greater than the number of trains that
    -- can legally be between the sensor and the gate (which equals
    -- the number of track segments).
    -- The ControllerGoingDown state comes to an end when the
    -- gate.down event occurs
    ControllerGoingDown(n) =
         (if GateSeg < n then ERROR else sensor.in -> ControllerGoingDown(n+1))
      [] gate.down -> ControllerDown(n)
      [] tock -> ControllerGoingDown(n)
      [] sensor.out -> ERROR
    -- When the gate is down, the occurrence of a train entering its
    -- sector causes no alarm, and each time a train leaves the gate
    -- sector the remaining count goes down, or the gate is signalled
    -- to go up, as appropriate.
    -- Time is allowed to pass arbitrarily in this state, except that
    -- the direction to the gate to go up is instantaneous when due.
    ControllerDown(n) =
         (if GateSeg < n then ERROR else sensor.in -> ControllerDown(n+1))
      [] sensor.out -> (if n==1 then gate!go_up -> ControllerGoingUp
                               else ControllerDown(n-1))
      [] tock -> ControllerDown(n)
    -- When the gate is going up, the inward sensor may still fire,
    -- which means that the gate must be signalled to go down again.
    -- Otherwise the gate goes up after UpTime units.
    ControllerGoingUp =  gate!up → ControllerUp
                     [] tock → ControllerGoingUp
                     [] sensor.in → gate!go_down → ControllerGoingDown(1)
                     [] sensor.out → ERROR
  within ControllerUp

-- Any process will be allowed to generate an error event, and since we will
-- be establishing that these do not occur, we can make the successor process
-- anything we please, in this case STOP.

ERROR = error → STOP

-- The following are the times we assume here for the gate to go up
-- and go down.  They represent upper bounds in each case.

-- DownTime = 5 -- make this a parameter for experimentation
VeryFastGate = 3
FastGate = 4
NormalGate = 5
SlowGate = 6

UpTime = 2

Gate(DownTime) =
  let
    GateUp = gate.go_up → GateUp
          [] gate.go_down → GateGoingDown(0)
          [] tock → GateUp
    GateGoingDown(n) =
         gate.go_down → GateGoingDown(n)
      [] if n == DownTime
         then gate.down → GateDown
         else gate.down → GateDown |~| tock → GateGoingDown(n+1)
    GateDown = gate.go_down → GateDown
            [] gate.go_up → GateGoingUp(0)
            [] tock → GateDown
    GateGoingUp(n) = gate.go_up → GateGoingUp(n)
                  [] gate.go_down → GateGoingDown(0)
                  [] if n == UpTime
                     then gate.up → GateUp
                     else gate.up → GateUp |~| tock → GateGoingUp(n+1)
  within GateUp

-- Since Gate has explicitly nondeterministic behaviour, we can expect
-- to gain by applying a compression function, such as diamond, to it;
-- we declare a number of "transparent" compression functions

transparent sbisim
transparent normalise
transparent explicate
transparent diamond
-- sbisim(X) = X  -- added by leuschel
-- explicate(X) = X  -- added by leuschel
-- diamond(X) = X    -- added by leuschel
-- normalise(X) = X  -- added by leuschel

GateAndController(dt) = Controller [|{|tock,gate|}|] diamond(Gate(dt))

-- Finally, we put the network together with the gate unit to give our
-- overall system

System(invmaxspeed,gatedowntime) =
  TimedNetwork(invmaxspeed) [|{|sensor,tock|}|] GateAndController(gatedowntime)


MAIN = System(NormalTrain,NormalGate)  -- added by leuschel

-- And now for specifications.  Since we have not synchronised on any
-- error events, they would remain visible if they occurred.  Their
-- absence can be checked with

NoError = CHAOS(diff(Events,{error}))

-- assert NoError [T= System(NormalTrain,NormalGate)

-- This shows that none of the explicitly caught error conditions arises,
-- but does not show that the system has the required safety property of
-- having no train on the GateSeg when the gate is other than down.

-- The required specifications are slight generalisations of those
-- discussed in specs.csp; the following notation and development is
-- consistent with that discussed there.

SETBETWEENx(EN,DIS,C) = ([]x:EN @ x -> SETOUTSIDEx(DIS,EN,C))
                        [] ([] x:DIS @ x -> SETBETWEENx(EN,DIS,C))

SETOUTSIDEx(DIS,EN,C) = ([] c:C @ c -> SETOUTSIDEx(DIS,EN,C))
                        [] ([] x: EN @ x -> SETOUTSIDEx(DIS,EN,C))
                        [] ([] x:DIS @ x -> SETBETWEENx(EN,DIS,C))

-- The above capture the sort of relationships we need between the
-- relevant events.  If we want to stay within Failures-Divergence Refinement
-- (as opposed to using Trace checking subtly), we need to do the following to
-- turn them into the conditions we need:

EnterWhenDown =
  SETBETWEENx({gate.down},
              {gate.up,gate.go_up,gate.go_down},
              {|enter.GateSeg|})
  [|{|gate, enter.GateSeg|}|]
  CHAOS(Events)

GateStillWhenTrain =
  SETOUTSIDEx({|enter.GateSeg|},{|leave.GateSeg|},{|gate|})
  [|{|gate,enter.GateSeg,leave.GateSeg|}|]
  CHAOS(Events)
```

```
-- So we can form a single safety spec by conjoining these:

Safety = EnterWhenDown [|Events|] GateStillWhenTrain

-- There are a number of possible combinations which may be of interest; try

-- assert Safety [T= System(SlowTrain,NormalGate)
-- assert Safety [T= System(NormalTrain,NormalGate)
-- assert NoError [T= System(FastTrain,SlowGate)
-- assert Safety [T= System(FastTrain,NormalGate)
-- assert NoError [T= System(FastTrain,NormalGate)
-- assert Safety [T= System(SlowTrain,SlowGate)
-- assert Safety [T= System(FastTrain,FastGate)
-- assert Safety [T= System(FastTrain,VeryFastGate)


-- An important form of liveness we have thus far ignored is that the clock
-- is not stopped: for this it is sufficient that TimingConsistency
-- refines TOCKS, where

TOCKS = tock → TOCKS

-- The following is the set of events that we cannot rely on the environment
-- not delaying.

Delayable = {|enter.1|}
NonTock = diff(Events,{tock})
TimingConsistency(ts,gs) =
  explicate(System(ts,gs)[|Delayable|]normalise(CHAOS(Delayable))\NonTock)

-- assert TOCKS [FD= TimingConsistency(NormalTrain,NormalGate)

-- The Safety condition completely ignored time (although, if you change some
-- of the timing constants enough, you will find it relies upon timing for
-- it to be satisfied).  Because of the way we are modelling time, the
-- main liveness constraint (that the gate is up when prudent) actually
-- becomes a safety condition (one on traces).  It is the combination of this
-- with the TOCKS condition above (asserting that time passes) that gives
-- it the desired meaning.

-- We will specify that when X units of time has passed since the last
-- train left the gate, it must be open, and remain so until another
-- train enters the system.  This is done by the following,  which monitor
-- the number of trains in the system and, once the last has left, no
-- more than X units of time pass (tock events) before the gate is up.  The
-- gate is not permitted to go down until a train is in the system.

Liveness(X) =
  let
    Idle = tock → Idle
        [] enter.1?_ → Busy(1)
    Busy(n) = tock → Busy(n)
           [] enter.1?_ → Busy(if n < GateSeg then (n+1) else n)
```

```
          [] leave.GateSeg?_ -> (if n==1 then UpBefore(X) else Busy(n-1))
          [] gate?_ -> Busy(n)
    UpBefore(m) = m != 0 & tock -> UpBefore(m-1)
              [] gate?x -> (if x==up then Idle else UpBefore(m))
              [] enter.1?_ -> Busy(1)
  -- Initially the gate is up in the system, so the liveness condition
  -- takes this into account.
  within Idle

GateLive(X) = Liveness(X) [|{|tock,gate,enter.1,leave.GateSeg|}|]CHAOS(Events)

-- assert GateLive(3) [T= System(NormalTrain,NormalGate)
-- assert GateLive(2) [T= System(NormalTrain,NormalGate)
-- assert GateLive(1) [T= System(NormalTrain,NormalGate)

-- Note that GateLive is antitonic, so for instance

-- assert GateLive(3) [T= GateLive(2)
```

C.3.5 A Specification of a Scheduler

```
-- A CSP specification and refinement of a scheduler

psize = 5
PID = {1..psize}
channel new : PID
channel delete : PID
channel ready : PID
channel enter : PID
channel leave : PID

-- Specification

NEWPROC(p) = new.p -> PROC(p)

PROC(p) =
    ready.p -> enter.p -> leave.p -> PROC(p)
    [] delete.p -> NEWPROC(p)

MUTEX = enter?p -> leave.p -> MUTEX

SCHEDULER0 =
(||| p:PID @ NEWPROC(p)) [| {| enter, leave |} |]  MUTEX

-- Refinement

QUEUE(q) =
       #q<psize & ready?p -> QUEUE(q^<p>)
    [] q!=<> & enter.head(q)   -> QUEUE(tail(q))

SCHEDULER1 =
((||| p:PID @ NEWPROC(p)) [| {| enter, leave |} |]  MUTEX)
      [| {| ready, enter |} |] QUEUE(<>)
```

```
assert SCHEDULER0 [T= SCHEDULER1

MAIN = SCHEDULER1
```

C.3.6 A Specification of a Bank System

```
-- A model from the paper
-- David A. Basin, Ernst-Rdiger Olderog, Paul E. Sevin:
-- Specifying and analyzing security automata using CSP-OZ. ASIACCS 2007: 70-81

-- slightly adapted for the Haskell CSPM tool

m = λ x,S @ member(x,S)
-- abbreviation for membership function
-- Definitions of constants
datatype UserID = u1 | u2 | u3
-- concrete set of user ids
datatype AccID = ac1 | ac2
-- concrete set of accounts
Val = {(-6)..6}
-- concrete set of values accounts may assume
Sum = {1..6}
-- concrete set of sums customers may transfer
-- CSP Part Bank
channel login: UserID.Bool
channel logout
channel balance: AccID.Val
channel transferReq: Sum.AccID.AccID.Bool
channel transferExec: Sum.AccID.AccID
channel abort
mainB = login?u?ok → (ok & Operate
    [] not ok & mainB)

Operate = (balance?a?v → Operate
        [] transferReq?s?a1?a2?ok →
              (transferExec!s!a1!a2 → Operate
               [] abort → Operate)
        [] logout → mainB)
-- OZ Part Bank
-- We represent the current balance bal as a set of
-- pairs (account-id, value). This requires some
-- auxiliary functions defined below:
ValSet = λb,a @ { v | v ← Val, m((a,v),b) }
pick({x}) = x
PickVal = λb,a @ pick(ValSet(b,a))
withdrawOK = λb,a1,a2,s @
         not(a1==a2) and
         (PickVal(b,a1) - s ≥ 0)
upd = λb,a,v @
   let
         bminus = diff(b,{(a,vold) | vold ←Val })
   within
         union(bminus, {(a,v)})
-- The set of customers is defined as a concrete
-- subset of UserID. It appears as a global parameter
-- of the process OZB.
cust = {u1, u2}
OZB(bal,transferOK) =
-- next line is disabled for Haskell tool:
     (-- m(bal,Set({(a,v)|a←AccID,v←Val})) and
      m(transferOK,Bool)) &
         (
         ([] (u,ok): {(u,m(u,cust)) | u ← UserID } @
              login.u.ok → OZB(bal,transferOK))
         []
          ([] (a,v) :
           {(a,PickVal(bal,a)) |
              a ← AccID, card(ValSet(bal,a))==1 } @
           balance.a.v → OZB(bal,transferOK))
        []
         ([] (s,a1,a2,ok):
         {(s,a1,a2,withdrawOK(bal,a1,a2,s)) |
            s←Sum, a1←AccID, a2 ←AccID,
           card(ValSet(bal,a1)) == 1 } @
          transferReq.s.a1.a2.ok → OZB(bal,ok))
        [] transferExec?s?a1?a2 →
             if transferOK and card(ValSet(bal,a1))==1 and
                card(ValSet(bal,a2))==1
             then
              let
                 v1 = PickVal(bal,a1) - s
                 v2 = PickVal(bal,a2) + s
             within
               OZB(upd(upd(bal,a1,v1),a2,v2), transferOK)
           else OZB(bal,transferOK)
      )
-- Parallel Composition of CSP and OZ part of the Bank
-- starts with the following initial balance of the
-- accounts:
bal = { (ac1,3), (ac2,-2) }
Bank =  mainB
         [|{| login,balance,transferReq,transferExec |}|]
        OZB(bal,false)

OZB_bal_false = OZB(bal,false) -- added by mal

-- Unprotected System
UnpSys = Bank
-- SecAut
datatype Actions = Balance | Transfer
datatype PIN = p1 | p2
-- concrete set of pins
datatype TN = t1 | t2 |t3
-- concrete set of tans
-- CSP Part SecAut
channel pin: PIN.Bool
channel tan: TN.Bool
mainS = login?u?ok → (ok & Identify
    [] not ok & mainS)
Identify = pin?p?ok → (ok & SecOperate
    [] not ok & Identify)
SecOperate =  balance?a?val → SecOperate
    [] transferReq?s?a1?a2?ok → TanCheckExec
    [] logout → mainS
TanCheckExec =
    tan?t?ok → (ok & transferExec?s?a1?a2 → SecOperate
    [] not ok & abort → SecOperate)
    [] logout → mainS
-- OZ Part SecAut
-- The following definitions appear as
-- global parameters of the process OZS:
priv = { (u1,ac1,Balance),
         (u1,ac1,Transfer),
         (u2,ac2,Balance),
         (u2,ac2,Transfer),
         (u3,ac1,Balance),
         (u3,ac1,Transfer),
         (u3,ac2,Balance) }
-- concrete set of privileges
cred(u1) = p1
cred(u2) = p2
-- concrete set of credentials
N = 2
-- N+1 is the concrete length of the tanlist
tanlist(u1,0) = t1
tanlist(u1,1) = t3
tanlist(u1,2) = t2
tanlist(u2,0) = t1
tanlist(u2,1) = t2
tanlist(u2,2) = t3
-- concrete tanlist
tid0 = { (u1,0), (u2,0) }
-- initial tan indices
OZS(uid,tid) =
    ( m(uid,UserID) and m(tid,Set({(u,v) |  u ←cust, v←{0..N} }) ) ) &
    (login?u?ok → OZS(u,tid)
    []([] (p,ok): {(p,m(uid,cust) and p == cred(uid)) |
    p ← PIN } @ pin.p.ok → OZS(uid,tid))
    []([] a: {a | a ← AccID,
    m((uid,a,Balance),priv)} @
    balance.a?v → OZS(uid,tid))
    []([] a1: {a1 | a1 ← AccID,
    m((uid,a1,Transfer),priv)} @
    transferReq?s.a1?a2?ok → OZS(uid,tid))
    []([] (t,ti,ok): {(t,ti,t == tanlist(uid,ti)) |
    t ← TN, ti ← {0..N},
    card(ValSet(tid,uid)) == 1,
    ti == PickVal(tid,uid) } @
    tan.t.ok → ( (ok and (ti < N) &
    OZS(uid,upd(tid,uid,ti+1))
    [](not ok or (ti == N) &
    OZS(uid,tid)) )) )
    )

-- Parallel Composition of CSP and OZ part of the SecAut
SecAut =
mainS
[|{| login,pin,balance,transferReq,tan |}|]
OZS(u3,tid0)
-- Secure System
A = {| login, balance, transferReq, transferExec,
       abort, logout |}
    SecSys = Bank [| A |] SecAut
-- has alphabet A union {| pin, tan |}
--------------------------------------------------------
-- Individual Traces (Use Cases): two examples
--------------------------------------------------------
-- assert SecSys [T= login.u1.true → pin.p1.true →
--  transferReq.3.ac1.ac2.true →
--  tan.t1.true →
--  transferExec.3.ac1.ac2 → STOP
-- satisfied
-- assert SecSys [T= login.u1.true → pin.p1.true →
--    transferReq.3.ac1.ac2.true →
--    tan.t2.false →
--    transferExec.3.ac1.ac2 → STOP
-- not satisfied: tan t2 is false
--------------------------------------------------------
-- General Properties
--------------------------------------------------------
-- Deadlock
-- Bank, UnpSys, SecAut, SecSys are all
-- deadlock free. -- checked
-- Livelock (Divergence)
-- Bank, UnpSys, SecAut, SecSys are all
-- livelock free. -- checked
-- Determinism
-- Bank, UnpSys, SecAut, SecSys are all
-- deterministic. -- checked
--------------------------------------------------------
-- Refinement Properties
--------------------------------------------------------
-- assert UnpSys [T= SecSys
-- not satisfied due to pin and tan
-- assert UnpSys [T= SecSys λ {|pin, tan |}
-- checked for values up to 6
--------------------------------------------------------
-- Security Properties
--------------------------------------------------------
-- No balance check before a sequence of successful
-- login and pin, belonging to the credentials of
-- the user.
A1 = {| transferReq, transferExec, abort, tan |}
P1 = ([] u : { u | u ← UserID,
member(u,cust) } @
login.u.true → P1L(u))
```

```
    [](([] u : { u | u ← UserID,
not member(u,cust) } @
login.u.false → P1)
    [] logout → P1
    [] ([] x : A1 @ x → P1)
P1L(u) =
    ([] p : { p | p ← PIN, p == cred(u) } @
    pin.p.true → P1LP)
    []([] p : { p | p ← PIN,
    not(p == cred(u)) } @
    pin.p.false → P1L(u))
    [] logout → P1
    [] ([] x : A1 @ x → P1L(u))
P1LP = balance?a?v → P1LP
    [] logout → P1
    [] ([] x : A1 @ x → P1LP)
-- assert P1 [T= SecSys
    -- satisfied

-- No transferExec before a successful tan.
A2 = union({| login, balance, transferReq,
    abort, logout, pin |},
    { tan.t.false | t ← TN })

P2 = tan?t!true {- mal: changed ∘ true to !true -}
  → P2T
    [] ([] x : A2 @ x → P2)
P2T = transferExec?s?a1?a2 → P2
    [] ([] x : A2 @ x → P2)
-- assert P2 [T= SecSys
-- satisfied

MAIN = Bank

-- LTL Formulas checked:
-- G ( [transferExec] ⇒ O [transferReq])
-- G ( [abort] ⇒ O [transferReq])
```

Bibliography

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] O. J. Anshus, J. M. Bjørndalen, and B. Vinter. PyCSP - Communicating Sequential Processes for Python. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, July 2007.

[3] L. Augustsson. λ-calculus cooked four ways.

[4] N. C. Brown. Communicating Haskell processes: Composable explicit concurrency using monads. In P. H. Welch, S. Stepney, F. A. Polack, F. R. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 67–83, Amsterdam, The Netherlands, September 2008. WoTUG, IOS Press.

[5] N. C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.

[6] R. Colvin and I. Hayes. Csp with hierarchical state. In M. Leuschel and H. Wehrheim, editors, *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 118–135. Springer Berlin / Heidelberg, 2009.

[7] Community. Enumerator and iteratee. `http://www.haskell.org/haskellwiki/Enumerator_and_iteratee`.

[8] Community. Ghc/type families. `http://www.haskell.org/haskellwiki/GHC/Type_families`.

[9] Community. Haskell programm coverage. `http://www.haskell.org/haskellwiki/Haskell_program_coverage`.

[10] Community. Tying the knot. `http://haskell.org/haskellwiki/Tying_the_Knot`.

[11] Community. The Haskell Platform. `http://hackage.haskell.org/platform`, 2010.

[12] I. M. Dobrikov. Übersetzung von CSP-M nach Haskell (in German), 2010.

[13] S. Fischer. *On Functional Logic Programming and its Application to Testing*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2010.

[14] M. Fontaine. CSPM-cspm: cspm command line tool for analyzing CSPM specifications. `http://hackage.haskell.org/package/CSPM-cspm`, 2010.

[15] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[16] B. Ford and M. F. Kaashoek. Packrat parsing: a practical linear-time algorithm with backtracking, 2002.

[17] FormalSystem. Typechecker download. `http://www.fsel.com/typechecker_download.html`.

[18] A. Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[19] M. Goldsmith et al. Process behaviour explorer. ProBE user manual. `http://www.fsel.com/documentation/probe/probe-doc.pdf`, 2003.

[20] M. Goldsmith et al. Failures-divergences refinement. fdr2 user manual. `http://www.fsel.com/documentation/fdr2/fdr2manual.pdf`, 2005.

[21] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[22] C. A. R. Hoare. The emperor's old clothes. *Commun. ACM*, 24:75–83, February 1981.

[23] Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.

[24] Y. Isobe and M. Roggenbach. Csp-prover - a proof tool for the verification of scalable concurrent systems. *JSSST (Japan Society for Software Science and Technology) Computer Software*, 25, 2008.

[25] J. Jeuring, S. Leather, J. P. Magalhães, and A. Rodriguez Yakushev. Libraries for generic programming in Haskell. Technical Report UU-CS-2008-025, Department of Information and Computing Sciences, Utrecht University, 2008.

[26] O. Kiselyov. Incremental multi-level input processing with left-fold enumerator. ACM SIGPLAN 2008 (Developer Tracks on Functional Programming), 2008.

[27] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers. *SIGPLAN Not.*, 40:192–203, September 2005.

[28] M. Kleine, B. Bartels, T. Gothel, and S. Glesner. Verifying the implementation of an operating system scheduler. In *Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '09, pages 285–286, Washington, DC, USA, 2009. IEEE Computer Society.

[29] M. Kleine and T. Gothel. Specification, verification and implementation of business processes using csp. In *Proceedings of the 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '10, pages 145–154, Washington, DC, USA, 2010. IEEE Computer Society.

[30] R. Lämmel and S. L. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI*, pages 26–37, 2003.

[31] D. J. P. Leijen and H. J. M. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

[32] M. Leuschel. Declarative Programming for Verification: Lessons and Outlook. In *Proceedings PPDP'2008*, pages 1–7. ACM Press, July 2008.

[33] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[34] M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, pages 278–297, 2008.

[35] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The meb and ceb static analysis for csp specifications. In *LOPSTR*, pages 103–118, 2008.

[36] M. Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, 2011.

[37] S. Marlow. Happy, a parser-generator for Haskell. `http://www.haskell.org/happy/`.

[38] S. Marlow. Haskell 2010 language report, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

[39] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[40] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. In A. Valmari, editor, *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 108–125. Springer Berlin / Heidelberg, 2006.

[41] B. Metzler, H. Wehrheim, and D. Wonisch. Decomposition for compositional verification. In *Proceedings International Conference on Formal Engineering Methods (ICFEM 2008)*, volume 5256, pages 105–125. Springer, Oktober 2008.

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

[43] N. Moffat, M. Goldsmith, and B. Roscoe. A representative function approach to symmetry exploitation for csp refinement checking. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 258–277, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] S. Mount, M. Hammoudeh, S. Wilson, and R. M. Newman. Csp as a domain-specific language embedded in python and jython. In *CPA*, pages 293–309, 2009.

[45] C.-c. S. Oleg Kiselyov, Simon Peyton Jones. Fun with type functions. presented at Tony Hoare's 75th birthday celebration, Cambridge, 17 April 2009, 2009-2010.

[46] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[47] A. N. Parashkevov and J. Yantchev. Arc - a tool for efficient refinement and equivalence checking for csp. In *In IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing ICA3PP '96*, pages 68–75, 1996.

[48] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. `http://www.haskell.org/onlinereport`.

[49] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[50] D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. *STTT*, 12(1):9–21, 2010.

[51] A. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[52] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.

[53] J. B. Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, Oxford University Computing Laboratory., 1998.

[54] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 1999.

[55] T. Sheard. Generic unification via two-level types and parameterized modules. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 86–97, New York, NY, USA, 2001. ACM.

[56] S. P. J. Simon Marlow, Ryan Newton. A monad for deterministic parallelism. submission ICFP'2011.

[57] J. Sun, Y. Liu, and J. S. Dong. Model checking csp revisited: Introducing a process analysis toolkit. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer Berlin Heidelberg, 2009.

[58] D. Swierstra. Combinator parsing: A short tutorial. Technical Report UU-CS-2008-044, Department of Information and Computing Sciences, Utrecht University, 2008.

[59] P. Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[60] Wikipedia. Birthday problem. `http://en.wikipedia.org/wiki/Birthday_problem`.

[61] Wikipedia. QuickCheck. `http://en.wikipedia.org/wiki/QuickCheck`, 2008.

[62] P. Y. Wong and J. Gibbons. A process semantics for bpmn. In *Proceedings International Conference on Formal Engineering Methods (ICFEM 2008)*, ICFEM '08, pages 355–374, Berlin, Heidelberg, 2008. Springer-Verlag.

[63] L. Yang and M. R. Poppleton. JCSProB: Implementing Integrated Formal Specifications in Concurrent Java. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2007*, pages 67–88, July 2007.

[64] B. Yorgey. The Typeclassopedia. *The Monad.Reader*, 13:17–68, Mar. 2009.