UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

MSc Thesis in Computer Science

Dawit Legesse Tirore

Mechanized formalization of a propositional calculus for contract specification

Supervisor: Fritz Henglein
Co-supervisor: Agata Anna Murawska

Handed in: October 3, 2023

Abstract

The successful management of commercial contracts is vital for businesses. Improper management is a costly affair, at worst leading to unintended contract breaches with hefty legal fees. Tools that support proper management of contracts are therefore highly desirable. One such tool is the contract specification language CSL, developed by Andersen et al. [1], supporting compositional specification of contracts. In this thesis, we formalize and mechanize a calculus for a restricted variant of CSL, with the mechanization carried out in the proof-assistant Coq. The calculus presented here will be used in the later formalization and mechanization of a calculus for CSL2, the successor of the CSL language.

Acknowledgments

I would like to thank my supervisors Fritz Henglein and Agata Anna Murawska for their guidance. I would also like to thank my parents Legesse and Bezunesh, and my sisters Sarah and Rebecca for your emotional support. Your support has helped me a lot. Lastly I'd like to thank Nicolaj Zøllner, who also has been writing his thesis during this period. I really enjoyed having someone to talk to who intimately could relate to the ups and downs one goes through in writing a thesis.

Contents

1 Introduction

A commercial contract states the terms and conditions for the exchange of goods and services between two or more agents (companies, independent contractors, etc). After the signing of a commercial contract, an essential problem is how the contract is managed. One issue when managing a contract is keeping track of its current state. What contract related events have happened in the past and what allowed actions can be taken now? Has the past events left the contract in a state of contract breach? These are concerns about the *monitoring* of a contract, one of the several tasks involved in managing contracts. Commonly ERP (Enterprise Resource Planning) systems are used for the general management of contracts. ERPs however share the drawback of being tailored to concrete industries. The sub-modules of the ERP system that deals with the management of contracts only support a set of contract templates common for their target industry. As a consequence of this, often some parts of a contract is managed with the ERP while others are only managed informally (written email, discussed at meetings). The informal management of contracts is a costly affair. It is estimated that a major investment bank in France has costs of about 50 mio. euro anually attributable to either disagreement about what a contract requires or violating a contract [1]. For these reasons, tools that support proper management of commercial contracts are desirable. One such tool is the contract specification language CSL, developed by Andersen et al. [1]. This language is compositional, meaning larger contract-specifications are composed of smaller specifications. CSL is a trace language and its alphabet contain events that hold data (e.g. $transmit(a_1, a_2, r, t)$). Constraints on this data can be given with logical predicates (e.g. $t \leq 5$).

In this thesis we give a mechanized formalization of a propositional calculus for the restricted variant of CSL that captures its compositionality but disregards logical predicates on data. The mechanization will be carried out in the proof-assistant Coq. A propositional calculus for contract specification (from now on simply a contract-calculus) is a formal system that allows contract specifications (from now on simply contracts) to be treated symbolically. We know that the truth value of $A \wedge B$ is the same as $B \wedge A$, making $\wedge$ a commutative logical connective. Similarly for two contracts c_0 and c_1 and for some binary operator $+$ we would for example want to know whether $c_0 + c_1$ and $c_1 + c_0$ has the same semantic meaning. The semantics of contracts will be defined by two distinct semantics, the first being a compositional semantics and the second being an operational semantics. These will be shown to be equivalent. The contract-calculus will be presented as an inference system and it will be shown that it indeed models semantic equivalence of contracts (soundness of the calculus) and that all equivalences can be derived within the system (completeness of the calculus). This will be our formalization, i.e. our set of definitions and theorems about these definition *on paper*. To ensure that we arrive at a sound formalization, all definitions are also represented in Coq and the theorems about them mechanically checked. This part is our mechanization.

Approach The approach that will be taken in this thesis is building the calculus incrementally. We start with the most restricted variant CSL_0 consisting solely of four constructs. Later we add a parallel operator yielding variant $CSL_{||}$ and finally we add iteration yielding variant CSL_*, corresponding to CSL without logical predicates and with general recursion restricted to only tail-recursion.

Contributions We make the following contributions in this thesis

- For the restricted CSL language without predicates, we formalize a sound and complete *coinductive* axiomatization of contract equivalence. The axiomatization was motivated by a coinductive decomposition rule that was introduced by Grabmeyer [2] in his coinductive axiomatization of regular expression equivalence, who himself was inspired by a similar rule presented in a type-theoretic context by Brandt and Henglein [3]. The purpose of Grabmeyer's axiomatization was different than ours, so he included semantic notions of regular expressions in some rules. Our axiomatization does on the other hand not refer to semantic notions in any rules. Because our restricted variant of CSL corresponds to the parallel regular expressions, the axiomatization is therefore equally a sound and complete axiomatization of parallel regular expression equivalence wrt. to its membership semantic.

- We mechanize this coinductive axiomatization in Coq using the paco library that mechanizes the notion of parameterized coinduction introduced by Hur et al. [4]. Mixing use of inductive and coinductive rules in the same definition is hard to represent in Coq. Examples from Hur et al. showing how to define predicates on streams in Coq by parameterized inductive definitions, inspired us to equally represent the axiomatization as a parameterized inductive definition. This allowed us to mix the use of inductive and coinductive rules.

Outline We start in chapter 2 with an introduction to the full-size CSL, providing the context for our later investigation of its restricted variants CSL_0, $CSL_{||}$ and CSL_*. In chapter 3 we formalize CSL_0. Chapter 4 gives an introduction to Coq, followed by chapter 5 where we mechanize CSL_0. Chapter 6 and 7 respectively formalize and mechanize $CSL_{||}$. Likewise chapter 8 and 9 respectively formalize and mechanize CSL_*. We then discuss related work and possibly future work in chapter 10 and end with the conclusion in chapter 11. In appendix A an example derivation of the mechanized calculus is given. To avoid cluttering the thesis with proofs, only the most illustrative parts are given and parts that were cut during

Section 1. (Sale of goods) Seller shall sell and deliver to buyer (description of goods) no later than (date).
Section 2. (Consideration) In consideration hereof, buyer shall pay (amount in dollars) in cash on delivery at the place where the goods are received by buyer.
Section 3. (Right of inspection) Buyer shall have the right to inspect the goods on arrival and, within (days) business days after delivery, buyer must give notice (detailed-claim) to seller of any claim for damages on goods.

Figure 1: Agreement to Sell Goods

revision can be found in Appendix B. Appendix C contains the full Coq source code.

2 Introduction to CSL

The full-size industrial strength contract specification language CSL will be introduced in this chapter. This chapter paraphrases chapter 2 and chapter 3 from Andersen et al. The theorems presented in this chapter have *not* been mechanized as part of this thesis. All figures presented in this chapter were borrowed from their paper.

2.1 Modelling contracts

The base construct of a CSL contract is a commitment. A commitment could for example represent the transfer of a good or the notification of a shipment delay. Commitments can be composed with operators that capture general contract patterns. These contract patterns will now be illustrated:

- Consider Section 1 of the commercial contract in Figure 1. It places two commitments on the Seller, namely to sell a good and deliver it to the buyer. The order in which the Seller fulfills the two commitments is irrelevant. This contract pattern will be called parallel composition, represented by the operator $||$.

- In the same figure consider now both Section 1 and 2. The commitments of the Seller in Section 1 precedes the commitment of the Buyer to pay for the good. Failing to deliver the good only leaves the Seller in breach of the contract. This contract pattern will be called sequential composition, represented by the operator ;.

- Finally consider Section 1 in Figure 2, where an attorney agrees to deliver monthly legal service. This reoccurring service includes a mandatory part as well as an optional part. The choice between fulfilling one of two commitments will be called an *alternative*, represented by the operator $+$. Section 1 is then a repetition of an alternative and we will soon see how repetition is modeled in the language.

Section 1. The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.
Section 2. In consideration hereof, the company shall pay a monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receival of an invoice.
Section 3. This contract is valid 1/1-12/31, 2004.

Figure 2: Reoccuring services that includes alternatives

2.2 Syntax

The syntax of the language can be seen below:

$$
\begin{aligned}
c ::= {} & \text{Success} \mid \text{Failure} \mid f(\mathbf{a}) \mid \\
& \text{transmit}(A_1, A_2, R, T \mid P).\, c \mid \\
& c_1 + c_2 \mid c_1 \parallel c_2 \mid c_1 ; c_2 .
\end{aligned}
$$

Success represents the completed contract with no remaining commitments whereas Failure represents contract breach. $f(\mathbf{a})$ is the instantiation of contract template f with argument vector $\mathbf{a}$. In mathematics, a structure is a set endowed with some operations. CSL is built on top of a base structure of domains $(\mathcal{A}, \mathcal{R}, \mathcal{T})$ representing $agents$, $resources$ and $time$, where the timepoints of $\mathcal{T}$ are totally ordered. CSL is parameterized over an expression language $\mathcal{P}$. The construct $transmit(A_1, A_2, R, T|P).c$ is a contract where the commitment $transmit(A_1, A_2, R, T|P)$ must be matched first. Here A_1, A_2, R and T are variable occurrences whose scope is P and c. A commitment is matched against an incoming event, binding the variable occurrences to the data contained in the event. Matching the commitment against the event $transmit(a_1, a_2, r, t)$, binds the values a_1, a_2, r and t to resp. the variables A_1, A_2, R and T. The predicate P in the commitment may refer to these variables, defining a constraint that the incoming event must respect (for example a deadline). Finally, $c_1 + c_2$ is alternative, $c_1 || c_2$ is parallel composition and $c_1; c_2$ is sequential composition.

The CSL specification of the contract in Figure 1 can be seen in Figure 3. The example defines the contract templates `nonconforming` and `sale`. Though not seen in this example, contract templates can be recursively defined and repetition can therefore modeled by a tail-recursive contract template.

2.3 Event traces and contract satisfaction

A contract specifies a set of satisfying traces. A satisfying trace is one of possibly many ways of matching the commitments in a contract and concluding it. For illustrative purposes the alphabet has been restricted to the single event $transmit(a_1, a_2, r, t)$, where $a_1, a_2 \in \mathcal{A}$, $r \in \mathcal{R}$ and $t \in \mathcal{T}$. A trace is a sequence of events, denoting the empty trace as $\langle\rangle$, the singleton trace as e and concatenation of traces s_0 and s_1 by their juxtaposition $s_0 s_1$. The interleaving of traces s_0 and s_1 is written as $(s_0, s_1) \rightsquigarrow s_2$.

```
letrec
  nonconforming [seller, buyer, goods, payment, days, t1, notice] =
          transmit (buyer, seller, notice, T |
                    T < t1 + days d and #(goods,broken,t1) = 1).
          transmit (seller, buyer, payment/2, T' | T' < T + days d).

  sale [seller, buyer, goods, payment, t1, days, notice] =
          transmit (seller, buyer, goods, T | T < t1).
          transmit (buyer, seller, payment, T' | T' < t1).
          (Success + nonconforming (seller, buyer, goods, days, T', notice))
in
  sale ("Furniture maker", "Me", "Chair", 40, 2004.7.1, 8, "Chair broken")
```

Figure 3: CSL specification of sales contract

Contract satisfaction is defined in Figure 4. Here $D = \{f_i[\mathbf{X_i}] = c_i\}_{i=1}^m$ is a finite set of named contract templates. $\oplus$ is the extension operator on maps defined as

$$(m \oplus m')(x) = \begin{cases} m'(x), & \text{if } x \in domain(m') \\ m(x), & otherwise \end{cases}$$

The judgment $\delta' \vdash_D^\delta s : c$, expresses that s satisfies contract c, in the presence of the contract templates defined in D with δ as the top-level environment for D and c and additionally the local environment δ' only for c.

$$\delta' \vdash_{\mathrm{D}}^{\delta} \langle\rangle : \mathrm{Success} \qquad \frac{\mathbf{X} \mapsto \mathbf{v} \vdash_{\mathrm{D}}^{\delta} s : c \quad (f(\mathbf{X}) = c) \in \mathrm{D}, \mathbf{v} = \mathcal{Q}[\![\mathbf{a}]\!]^{\delta \oplus \delta'}}{\delta' \vdash_{\mathrm{D}}^{\delta} s : f(\mathbf{a})}$$

$$\frac{\delta \oplus \delta'' \models P \quad \delta'' \vdash_{\mathrm{D}}^{\delta} s : c \qquad (\delta'' = \delta' \oplus \{\mathbf{X} \mapsto \mathbf{v}\})}{\delta' \vdash_{\mathrm{D}}^{\delta} \mathrm{transmit}(\mathbf{v})\, s : \mathrm{transmit}(\mathbf{X}|P).\, c}$$

$$\frac{\delta' \vdash_{\mathrm{D}}^{\delta} s_1 : c_1 \quad \delta' \vdash_{\mathrm{D}}^{\delta} s_2 : c_2 \quad (s_1, s_2) \rightsquigarrow s}{\delta' \vdash_{\mathrm{D}}^{\delta} s : c_1 \parallel c_2} \qquad \frac{\delta' \vdash_{\mathrm{D}}^{\delta} s_1 : c_1 \quad \delta' \vdash_{\mathrm{D}}^{\delta} s_2 : c_2}{\delta' \vdash_{\mathrm{D}}^{\delta} s_1 s_2 : c_1; c_2}$$

$$\frac{\delta' \vdash_{\mathrm{D}}^{\delta} s : c_1}{\delta' \vdash_{\mathrm{D}}^{\delta} s : c_1 + c_2} \qquad \frac{\delta' \vdash_{\mathrm{D}}^{\delta} s : c_2}{\delta' \vdash_{\mathrm{D}}^{\delta} s : c_1 + c_2}$$

Figure 4: Contract satisfaction

With $\mathcal{Q}[\![\cdot]\!]$ as the evaluation function of for the expression language P, the second satisfaction rule defines a trace s to be satisfying the instantiated contract $f(\mathbf{a})$, when two conditions are met: Firstly, s must satisfy c in the local environment $\mathbf{X} \mapsto \mathbf{v}$. Here v is the evaluation of $\mathbf{a}$ in the environment $\delta \oplus \delta'$. Secondly, the contract template $f(\mathbf{X})$ must be present in D.
In the third rule, i.e. the one dealing with $transmit(A_1, A_2, R, T|P).c$, the nota-

tion, $\delta \oplus \delta' \vDash P$, means that the evaluation of P in the environment $\delta \oplus \delta'$ must return true.

2.4 Denotational semantics

The denotational semantics map contracts to mathematical objects and as contracts specify sets of satisfying traces, they are mapped to sets of traces. The denotation of a contract c could be defined as $\{s : \emptyset \vdash_D^\delta s : c\}$, but this is not a compositional definition. A compositional definition is desirable because it relates the contract operations to corresponding set operations. Figure 5 defines the domains of types. A compositional denotational semantics can be seen in Figure 6. c is said to denote trace set S in context D, δ when $\mathcal{C}[\![c]\!]^{D;\delta} = S$.

$$
\begin{aligned}
Dom[\![\text{Boolean}]\!] &= (\{\text{true}, \text{false}\}, =) \\
Dom[\![\text{Agent}]\!] &= (\mathcal{A}, =) \\
Dom[\![\text{Resource}]\!] &= (\mathcal{R}, =) \\
Dom[\![\text{Time}]\!] &= (\mathcal{T}, =) \\
\mathcal{E} &= \mathcal{A} \times \mathcal{A} \times \mathcal{R} \times \mathcal{T} \\
Tr &= (\mathcal{E}^*, =) \\
Dom[\![\text{Contract}]\!] &= (2^{Tr}, \subseteq) \\
Dom[\![\tau_1 \times \ldots \times \tau_n \to \text{Contract}]\!] &= Dom[\![\tau_1]\!] \times \ldots \times Dom[\![\tau_n]\!] \to Dom[\![\text{Contract}]\!] \\
Dom[\![\Gamma]\!] &= \{\{f_i \mapsto v_i\}_{i=1}^m \mid v_i \in Dom[\![\tau_{i1}]\!] \times \ldots \times Dom[\![\tau_{in_i}]\!] \to Dom[\![\text{Contract}]\!]\} \\
&\quad \text{where } \Gamma = \{f_i \mapsto \tau_{i1} \times \ldots \times \tau_{in_i} \to \text{Contract}\}_{i=1}^m \\
Dom[\![\Delta]\!] &= \{\{X_i \mapsto v_i\}_{i=1}^m \mid v_i \in Dom[\![\tau_i]\!]\} \\
&\quad \text{where } \Delta = \{X_i : \tau_i\}_{i=1}^m \\
Dom[\![\Gamma; \Delta \vdash c : \text{Contract}]\!] &= Dom[\![\Gamma]\!] \times Dom[\![\Delta]\!] \to Dom[\![\text{Contract}]\!]
\end{aligned}
$$

Figure 5: Domains

$$
\begin{aligned}
\mathcal{C}[\![\text{Success}]\!]^{\gamma;\delta} &= \{\langle\rangle\} \\
\mathcal{C}[\![\text{Failure}]\!]^{\gamma;\delta} &= \emptyset \\
\mathcal{C}[\![f(\mathbf{a})]\!]^{\gamma;\delta} &= \gamma(f)(\mathcal{Q}[\![\mathbf{a}]\!]^{\delta}) \\
\mathcal{C}[\![\text{transmit}(\mathbf{X} \mid P).\, c]\!]^{\gamma;\delta} &= \{\text{transmit}(\mathbf{v})\, s : \mathbf{v} \in \mathcal{E}, s \in Tr \mid \\
&\qquad \mathcal{Q}[\![P]\!]^{\delta \oplus \mathbf{X} \mapsto \mathbf{v}} = \text{true} \wedge s \in \mathcal{C}[\![c]\!]^{\gamma;\delta \oplus \mathbf{X} \mapsto \mathbf{v}}\} \\
\mathcal{C}[\![c_1 + c_2]\!]^{\gamma;\delta} &= \mathcal{C}[\![c_1]\!]^{\gamma;\delta} \cup \mathcal{C}[\![c_2]\!]^{\gamma;\delta} \\
\mathcal{C}[\![c_1 \parallel c_2]\!]^{\gamma;\delta} &= \left\{s : s \in Tr \mid \exists s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta}, s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}.\, (s_1, s_2) \rightsquigarrow s\right\} \\
\mathcal{C}[\![c_1; c_2]\!]^{\gamma;\delta} &= \{s_1 s_2 : s_1, s_2 \in Tr \mid s_1 \in \mathcal{C}[\![c_1]\!]^{\gamma;\delta} \wedge s_2 \in \mathcal{C}[\![c_2]\!]^{\gamma;\delta}\} \\
\mathcal{D}[\![\{f_i[\mathbf{X_i}] = c_i\}_{i=1}^m]\!]^{\delta} &= \mathit{least}\, \gamma : \gamma = \{f_i \mapsto \lambda \mathbf{v}_i.\mathcal{C}[\![c_i]\!]^{\gamma;\delta \oplus \mathbf{X}_i \mapsto \mathbf{v}_i}\}_{i=1}^m \\
\mathcal{E}[\![\text{letrec}\, \{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m \text{ in } c]\!]^{\delta} &= \mathcal{C}[\![c]\!]^{\mathcal{D}[\![\{f_i[\mathbf{X}_i] = c_i\}_{i=1}^m]\!]^{\delta};\delta}
\end{aligned}
$$

Figure 6: Denotational semantics

The theorem below states that the denotational semantics characterizes the satisfaction relation.

Theorem 2.1 (Denotational characterization of contract satisfaction) $\mathcal{C}[\![c]\!]^{\mathcal{D}[\![D]\!]^{\delta};\delta\oplus\delta'} = \{s \mid \delta' \vdash^{\delta}_{D} s : c\}$

As mentioned in the introduction, one aspect of contract management is monitoring. The satisfaction relation does not specify how a contract can be monitored, it only defines how to compositionally derive a satisfaction relation from other satisfactions. Neither does the denotational semantics, as it just gives a mathematical interpreation of a contract as a set of satisfying traces. To now aid us in the monitoring of a contract we introduce the residuation operator $\cdot\backslash\cdot$. For a trace set S, we define the residuation operator $\cdot\backslash\cdot$ as $e\backslash S := \{s|es \in S\}$. This operator can be seen as a filtering of S by those traces that begin with e, taking the tail of each of those traces. This idea can be lifted to contracts. If we let S be the set of traces matching c, then $e\backslash S$ is the set of traces matching $e\backslash c$, or more specifically $\mathcal{C}[\![e\backslash c]\!]^{\gamma;\delta} = \{s' \mid \exists s \in \mathcal{C}[\![c]\!]^{\gamma;\delta} : es' = s\}$. We also say that $e\backslash c$ is a residual contract of c.

The presence of a residuation operator can be used to define a monitoring semantics for contracts. During the monitoring of contract c, when an incoming event e is received, apply $e\backslash c$. If we receive a request of terminating the contract, check whether the contract is terminable now, if that is the case, successfully terminate, otherwise report that the contract cannot be terminated now and continue receiving events.

Figure 7 shows equalities that holds for residuation. In the figure $D, \delta \Vdash c = c'$ is short for $\mathcal{C}[\![c]\!]^{\gamma;\delta\oplus\delta'} = \mathcal{C}[\![c']\!]^{\gamma;\delta\oplus\delta'}$ and analogously for $D, \delta \Vdash c \subseteq c'$.

Lemma 2.2 (Correctness of residution) *The residuation equalities in Figure 7 are true.*

$$\mathrm{D}, \delta \models e\backslash\mathrm{Success} = \mathrm{Failure}$$
$$\mathrm{D}, \delta \models e\backslash\mathrm{Failure} = \mathrm{Failure}$$
$$\mathrm{D}, \delta \models e\backslash f(\mathbf{a}) = e\backslash c[\mathbf{v}/\mathbf{X}] \text{ if } (f(\mathbf{X}) = c) \in \mathrm{D}, v = \mathcal{Q}[\![a]\!]^{\delta}$$
$$\mathrm{D}, \delta \models \mathrm{transmit}(\mathbf{v})\backslash(\mathrm{transmit}(\mathbf{X} \mid P).\, c) = \begin{cases} c[\mathbf{v}/\mathbf{X}] \text{ if } \delta \oplus \{\mathbf{X} \mapsto \mathbf{v}\} \models P \\ \text{Failure otherwise} \end{cases}$$
$$\mathrm{D}, \delta \models e\backslash(c_1 + c_2) = e\backslash c_1 + e\backslash c_2$$
$$\mathrm{D}, \delta \models e\backslash(c_1 \parallel c_2) = e\backslash c_1 \parallel c_2 + c_1 \parallel e\backslash c_2$$
$$\mathrm{D}, \delta \models e\backslash(c_1; c_2) = \begin{cases} (e\backslash c_1; c_2) + e\backslash c_2 & \text{if } \mathrm{D}, \delta \models \mathrm{Success} \subseteq c_1 \\ e\backslash c_1; c_2 & \text{otherwise} \end{cases}$$

Figure 7: Residuation equalities

3 Formalizing CSL_0

In this chapter, we define a restricted variant of CSL by only allowing alternative and sequential composition of events that do not contain data. This language, CSL_0 is defined by the syntax:

$$c := Failure \mid Success \mid e \mid c_1 + c_2 \mid c_1; c_2$$

The language does not contain contract templates or indeed any form of iteration. Moreover, the $transmit(A_1, A_2, R, T|P).c$ construct has been replaced by e ranging over a finite set of data-free events. The exclusion of event-data and contract templates makes predicates pointless and therefore also not part of CSL_0. We will in this thesis consider the data-free event set $\{T, N\}$ (short for Transfer and Notify). All binary operators are left associative and ; binds tighter than $+$.

Replacing a construct The reader might have expected to see the construct $e.c$ rather than e as that would correspond to the data-free version of the CSL construct $transmit(A_1, A_2, R, T|P).c$. The reason the construct $e.c$ is not used is due an undesirable semantic about map extensions under sequential composition, something that was noted by Andersen et al. Briefly stated, the compositional semantics of the full-size CSL only extends the local environment in the rule for the construct $transmit(A_1, A_2, R, T|P).c$. This construct can be thought of as a special case of sequential composition, where the left sub-contract is a commitment expecting a transmit event. This special case of sequential composition has the desirable property of extending the local environment in which its proceeding contract is evaluated, something that is not the case for sequential composition in general. If the semantics did extend the local environment during sequential composition, $transmit(A_1, A_2, R, T|P)$ could be treated as a separate construct, such that $transmit(A_1, A_2, R, T|P).c$ would just be a short-hand for $transmit(A_1, A_2, R, T|P); c$. Andersen et al intend to make this change to the semantics in the next generation of the language CSL2. We will in thesis therefore use the construct e.

Satisfaction The satisfaction relation for CSL_0 is given by the judgment $s : c$, defined in Figure 8. Success matches only the empty trace. The contract e matches only the trace containing the single event e. Sequencing is written as $c_0; c_1$, matching a trace if it can be decomposed to two traces, each matching their respective contract. $c_0 + c_1$ represents choice, matching either on the left or right side. If contract c is satisfied by the empty trace, we say that c is nullable.

$$\frac{}{\langle\rangle : Success} \text{ (MSuccess)}$$

$$\frac{}{e : e} \text{ (MEvent)}$$

$$\frac{s_1 : c_1 \qquad s_2 : c_2}{s_1 s_2 : c_1; c_2} \text{ (MSeq)}$$

$$\frac{s : c_1}{s : c_1 + c_2} \text{ (MPlusL)}$$

$$\frac{s : c_2}{s : c_1 + c_2} \text{ (MPlusR)}$$

Figure 8: Compositional Semantics

3.1 Monitoring semantics

We saw in the last chapter that from a denotational semantic, one could derive a residuation operator to be used for monitoring contracts. We will not give a denotational semantic of CSL_0, so rather than *deriving* a residuation operator, we will *define* it.

As an auxillary function we first define $nu : Contract \rightarrow \{0, 1\}$.

$$nu(Success) := 1 \quad nu(Failure) := 0 \quad nu(e) := 0$$

$$nu(c_0 + c_1) := \begin{cases} 1, & \text{if } nu\ c_0 = 1 \lor nu\ c_1 = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$nu(c_0; c_1) := \begin{cases} 1, & \text{if } nu\ c_0 = nu\ c_1 = 1 \\ 0, & \text{otherwise} \end{cases}$$

We expect that if a contract c is nullable then $nu(c) = 1$ and if its not nullable then $nu(c) = 0$.

Lemma 3.1 *For all contracts c, if* $nu(c) = 1$ *then* $\langle\rangle : c$
Proof is by induction on c (not shown).

The residuation function $\cdot\backslash\cdot$ (which binds tighter than ; and $+$) is defined by induction on c.

$$e\backslash Failure := Failure \quad e\backslash Success := Failure$$

$$e\backslash e' := \begin{cases} Success, & \text{if } e = e' \\ Failure, & \text{otherwise} \end{cases}$$

$$e\backslash c_0 + c_1 := e\backslash c_0 + e\backslash c_1$$

$$(e\backslash c_0); c_1 := \begin{cases} e\backslash c_0; c_1 + e\backslash c_1, & \text{if } nu\ c_0 = 1 \\ (e\backslash c_0); c_1, & \text{otherwise} \end{cases}$$

Likewise we expect that this definition correctly defines the residual of a contract, so it should be the case that $es : c \iff s : e\backslash c$.

We can generalize this idea of residuating a contract with an event, to residuating a contract with a trace.

$$s\backslash\backslash c := \begin{cases} c, & \text{if } s = \langle\rangle \\ s'\backslash\backslash(e\backslash c), & \text{if } s = es' \end{cases}$$

For readability, expressions involving both event and trace residuation, such as $s\backslash\backslash(e\backslash c)$ is written as $\frac{e\backslash c}{s}$. Trace residuation should have the property that $s : c \iff \langle\rangle : s\backslash\backslash c$. Since we also expect a nullabe contract c to have $nu(c) = 1$, it should be the case that $s : c \iff nu(s\backslash\backslash c) = 1$. We now prove this.

Lemma 3.2 *For all traces s and contracts c, if $s : c$ then $nu(s\backslash\backslash c) = 1$*

We only show the case of MSeq.

We must show for all traces s_1, s_2 and contracts c_1, c_2:

$$nu(s_1\backslash\backslash c_1) = 1 \implies nu(s_2\backslash\backslash c_2) = 1 \implies nu(s_1 s_2\backslash\backslash c_1; c_2) = 1$$

Proof by induction on s_1

- Case $s_1 = []$.
  If it is also the case that $s_2 = []$, then the statement trivially holds because $nu(c_1) = nu(c_2) = nu(c_1; c_2) = 1$. If $s_2 = es_2'$ then because $nu(c_1) = 1$ we have $es_2'\backslash(c_1; c_2) = \frac{e\backslash c_1; c_2 + e\backslash c_2}{s_2'}$. Residuation distributes over $+$ so this is equivalent to $\frac{e\backslash\backslash c_1; c_2}{s_2'} + es_2'\backslash\backslash c_2$. By assumption $nu(es_2'\backslash\backslash c_2) = 1$. Therefore $nu(es_2'\backslash\backslash c_1; c_2 + es_2'\backslash\backslash c_2) = 1$.

- Case $s_1 = es_1'$.
  We must show $nu(es_1' s_2\backslash\backslash(c_1; c_2)) = 1$. We proceed by case distinction on $nu(c_1)$.

- Case $nu(c_1) = 1$.
  Then $es'_1s_2\backslash\backslash(c_1; c_2) = \frac{e\backslash c_1;c_2}{s'_1s_2} + es'_1s_2\backslash\backslash c_2$. We show the left operand is nullable. By IH, to show $nu(\frac{e\backslash c_1;c_2}{s'_1s_2}) = 1$, it suffices to show $nu(\frac{e\backslash c_1}{s'_1}) = 1$ and $nu(s_2\backslash\backslash c_2) = 1$, which we have by assumption.
- Case $nu(c_1) = 0$
  Then $es'_1s_2\backslash\backslash(c_1; c_2) = \frac{e\backslash c_1;c_2}{s'_1s_2}$ and we apply same steps as before.

Lemma 3.3 *For all events e, traces s and contracts c, if $s : e\backslash c$ then $es : c$.*

Proof by induction on c. The only interesting case is $c = c_0; c_1$, which we show now.
From $s : e\backslash(c_0; c_1)$ we must show $es : c_0; c_1$.
We proceed by case distinction on $nu(c_0)$.

- Case $nu(c_0) = 1$
  Then $e\backslash(c_1; c_2) = e\backslash c_1; c_2 + e\backslash c_2$ and $s : e\backslash c_1; c_2 + e\backslash c_2$ must have ended in either MPlusL or MPlusR.
  - Case MPlusL.
    Premise of MPlusL, $s : e\backslash c_1; c_2$, must have ended in MSeq, such that $s = s_1s_2$ and $s_1 : e\backslash c_1$ and $s_2 : c_2$. By IH on $s_1 : e\backslash c_1$ we have $es_1 : c_1$ from which we with $s_2 : c_2$ can compose $es : c_1; c_2$.
  - Case MPlusR.
    Premise of MPlusR is $s : e\backslash c_2$. By IH we have $es : c_2$. Because $nu(c_0) = 1$, by Lemma 3.1 we have $\langle\rangle : c_0$ from which we with MSeq have $es : c_1; c_2$.
- Case $nu(c_0) = 1$
  Then $e\backslash(c_1; c_2) = e\backslash c_1; c_2$ and $s : e\backslash c_1; c_2$ must have ended in MSeq, making the case analogous to the previous one.

Lemma 3.4 *For all traces s and contracts c, if $nu(s\backslash\backslash c) = 1$ then $s : c$.*

Proof by induction on s.

- Case $s = \langle\rangle$.
  Proved by Lemma 3.1
- Case $s = es'$.
  From $nu(es'\backslash\backslash c) = nu(\frac{e\backslash c}{s'}) = 1$ we must show $es' : c$.
  By IH on $nu(\frac{e\backslash c}{s'}) = 1$, we have $s' : e\backslash c$ and by Lemma 3.4 we then have $es' : c$.

Theorem 3.5 (Equivalence of semantics) *For all traces s and contracts c, $s : c \iff nu(s\backslash\backslash c) = 1$*

($\implies$) is proved by Lemma 3.2 and ($\impliedby$) by Lemma 3.4.

3.2 Contract equivalence

Contract satisfaction can be used to give an extensional definition of contract equivalence. We will say that two contracts c_0 and c_1 are satisfiably equivalent when they are satisfied by the same set of contracts, stated as $\forall\ s, s : c_0 \iff s : c_1$. An intuitive equivalence is that $c_0 + c_1$ is satisfiably equivalent to $c_1 + c_0$, making $+$ a commutative operator. Also $c_0; c_1; c_2$ is satisfiably equivalent to $c_0; (c_1; c_2)$, making ; an associative operator. Figure 9 defines our calculus for CSL_0 equivalence, i.e. our inference system for deriving equivalences. We will equally call this for an axiomatization of contract equivalence. When one can derive $c_0 == c_1$ in the system of Figure 9, we will say that c_0 and c_1 are derivably equivalent.

$$(c_0 + c_1) + c_2 == c_0 + (c_1 + c_2) \tag{1}$$
$$c_0 + c_1 == c_1 + c_0 \tag{2}$$
$$c + Failure == c \tag{3}$$
$$c + c == c \tag{4}$$
$$(c_0; c_1); c_2 == c_0; (c_1; c_2) \tag{5}$$
$$(Success; c) == c \tag{6}$$
$$c; Success == c \tag{7}$$
$$Failure; c == Failure \tag{8}$$
$$c; Failure == Failure \tag{9}$$
$$c_0; (c_1 + c_2) == (c_0; c_1) + (c_0; c_2) \tag{10}$$
$$(c_0 + c_1); c_2 == (c_0; c_2) + (c_1; c_2) \tag{11}$$
$$c == c \tag{12}$$

$$\frac{c_0 == c_1}{c_1 == c_0}\ \text{(Sym)} \qquad \frac{c_0 == c_1 \qquad c_1 == c_2}{c_0 == c_2}\ \text{(Trans)}$$

$$\frac{c_0 == c_0' \qquad c_1 == c_1'}{c_0 + c_1 == c_0' + c_1'}\ \text{(Ctx-plus)} \qquad \frac{c_0 == c_0' \qquad c_1 == c_1'}{c_0; c_1 == c_0'; c_1'}\ \text{(Ctx-seq)}$$

Figure 9: Axiomatization of contract equivalence. 12 axioms and 4 inference rules

We expect that a derivable equivalence implies a satisfiable equivalence.

Theorem 3.6 (Soundness) *For all contracts* $c_0\ c_1$, $c_0 == c_1 \implies \forall s.\ s : c_0 \iff s : c_1$
Proof is by induction on $c_0 == c_1$ *(skipped).*

3.3 Completeness

Whereas the soundness proof straightforwardly proceeds by induction on the derivation $c_0 == c_1$, completeness must be shown another way as satisfaction equivalence is not inductively defined. Satisfaction equivalence provides no information on the syntactic shape of c_0 and c_1, in contrast, derivable equivalence is entirely syntax based essentially deriving an equivalence by showing one contract can be rewritten into the other by a sequence of rewrites.

To fill the gap between the two equivalence relations, we introduce a third equivalence relation, *trace equivalence*. Letting Tr be the set of all traces, trace equivalence can with the use of projection function $L : Contract \to 2^{Tr}$ and embedding function $L^{-1} : 2^{Tr} \to Contract$ capture both semantic and syntactic properties about c_0 and c_1. On a high-level one can then show completeness in three steps.

1. Show that with the use of L, the satisfaction equivalence ($\forall s.\ s =\sim c_0 \iff s =\sim c_1$) can be turned into a trace equivalence $L(c_0) = L(c_1)$.
2. Use the embedding function L^{-1} to turn the trace equivalence into the derivable equivalence $(L^{-1} \,.\, L)(c_0) == (L^{-1} \,.\, L)(c_1)$.
3. Finally show the composed function $L^{-1} \,.\, L : Contract \to Contract$ respects the axiomatization, letting us conclude that $c_0 == c_1$

Definitions Writing $\{s_0 s_1 | (s_0, s_1) \in S_0 \times S_1\}$, as $S_0 S_1$, let the projection function $L : Contract \to 2^{Tr}$ be the function mapping a contract to its set of satisfying traces that match it, defined below:

$$L(c) = \begin{cases} \{[]\}, & \text{if } c = Success \\ \{\}, & \text{if } c = Failure \\ L(c_0) \cup L(c_1), & \text{if } c = c_0 + c_1 \\ L(c_0)L(c_1), & \text{if } c = c_0; c_1 \end{cases}$$

c_0 and c_1 are said to be *trace equivalent* when $L(c_0) = L(c_1)$, that is, the set of traces that match c_0 is the same that matches c_1.

$L(c)$ consists of all traces that match c and only those traces.

Lemma 3.7 *For all s c,* $s =\sim c \iff s \in L(c)$
Proof of ($\implies$) *is by induction on the derivation of* $s : c$ *and* ($\impliedby$) *is by induction on c (skipped).*

Lemma 3.8 *For all* $c_0\ c_1$, $(\forall s.\ s =\sim c_0 \iff s =\sim c_1) \implies L(c_0) = L(c_1)$
Immediate from Lemma 3.7.

By soundness of the axiomatization, associativity, commutativity and idempotence of $+$ and neutrality of Failure, means that the summation of a set of contracts can be represented with the big operatorΣ. Likewise the folding of an ordered sequence of contracts by ; can be represented with the big operator $\prod$ (with Success as its neutral element). Of course we must remember that ; is not commutative, making some laws that usually would hold for the operator $\prod$ unsound in our axiomatization and therefore not to be used.

The trace embedding of trace s is now defined as $\prod_{i=1}^{n_s} s^i$, where s^i is the i'th event of the trace and n_s is the length of the trace. The trace set embedding of trace set S, or simply the embedding of S, is then defined as $\Sigma_{s\in S} \prod_{i=1}^{n_s} s^i$. By definition of the big operators for any trace equivalent c_0 and c_1, applying projection followed by embedding on both, by definition must be a derivable equivalence. More precisely:

$$L(c_0) = L(c_1) \implies \Sigma_{s\in L(c_0)} \prod_{i=1}^{n_s} s^i == \Sigma_{s\in L(c_1)} \prod_{i=1}^{n_s} s^i.$$

We now show that all contracts are derivably equivalent to their trace set embedded as a contract, which we then use to show completeness in Theorem 3.10.

Lemma 3.9 *For all contracts c,* $\Sigma_{s\in L(c)} \prod_{i=1}^{n_s} s^i == c$.

Proof by induction on c (showing cases for $+$ and ;).

- Case $c = c_0 + c_1$.
  We must show

  $$\Sigma_{s\in L(c_0+c_1)} \prod_{i=1}^{n_s} s^i == c_0; c_1$$

  We have that

  $$\Sigma_{s\in L(c_0+c_1)} \prod_{i=1}^{n_s} s^i == \big(\Sigma_{s\in L(c_0)} \prod_{i=1}^{n_s} s^i\big) + \big(\Sigma_{s\in L(c_1)} \prod_{i=1}^{n_s} s^i\big)$$

  With context rule of $+$ and appeal to IHs we then have.

  $$\big(\Sigma_{s\in L(c_0)} \prod_{i=1}^{n_s} s^i\big) + \big(\Sigma_{s\in L(c_1)} \prod_{i=1}^{n_s} s^i\big) == c_0; c_1$$

- Case $c = c_0; c_1$.
  We must show:

  $$\Sigma_{s\in (L(c_0)L(c_1))} \prod_{i=1}^{n_s} s^i == c_0; c_1.$$

  By IH on c_0 and c_1 with context rule for sequence, it suffices to show:

  $$\Sigma_{s\in (L(c_0)L(c_1))} \prod_{i=1}^{n_s} s^i == \big(\Sigma_{s\in L(c_0)} \prod_{i=1}^{n_s} s^i\ \big); \big(\Sigma_{s\in L(c_1)} \prod_{i=1}^{n_s} s^i\big).$$

By distributivity of ; over $+$ this is equivalent to:

$$\Sigma_{s\in(L(c_0)L(c_1))}\prod_{i=1}^{n_s} s^i == \Sigma_{s_0\in L(c_0)}\Sigma_{s_1\in L(c_1)}\big(\prod_{i=1}^{n_{s_0}} s_0^i \ ; \ \prod_{j=1}^{n_{s_1}} s_1^j\big).$$

By associativity of ; we then have:

$$\Sigma_{s\in(L(c_0)L(c_1))}\prod_{i=1}^{n_s} s^i == \Sigma_{s_0\in L(c_0)}\Sigma_{s_1\in L(c_1)}\prod_{i=1}^{n_{s_0}+n_{s_1}} (s_0 s_1)^i$$

Which by definition of cartesian products are derivably equivalent.

Theorem 3.10 (Completeness) *For all* c_0 c_1, $(\forall s.\ s =\sim c_0 \iff s =\sim c_1) \implies c_0 == c_1$

Immediate from Lemma 3.8 and 3.9.

4 Introduction to proving in Coq

Coq is based on the Logic of Inductive Constructions (LIC) which is a typed lambda calculus extended with inductive definitions [5]. An inductive definition is a collection of constructors each with an arity, that together defines a set of closed terms built from these constructors. Coq provides a unified way to code both logical propositions and functional programs in the same language, Gallina. Designed around the Curry Howard Isomorphism, a Gallina expression p of type t, denoted by the typing judgment $p : t$, can be read both as "program p has type t" and "proof term p, proves statement t". Building proofs by explicitly giving the inductive construction can be cumbersome and therefore Coq also provides a mechanism for building proofs by backward reasoning using tactics written in the tactic-language Ltac. Using Ltac we can automate much of the proof construction. As a warm up to the subsequent mechanization chapters, we now see an example of how to represent the natural numbers as well as proofs about them as inductive constructions in Coq. We end the chapter with showing how to represent decision procedures in Coq.

4.1 Natural numbers in Coq

The natural numbers are represented by the inductive definition `nat`, consisting of the null and successor constructors.

```
Inductive nat : Set :=
    O : nat | S : nat -> nat
```

The type of `nat` (which is itself a type) is `Set`. Types of types are also called sorts, and all types fall within one of the three sorts `Set`, `Prop` and `Type`. The sort `Set` contains types that are *informational*, i.e. the values of the given type carry information such as the natural numbers. `Prop` contains types that are logical propositions. In general these types are not informational, as values from these types are proof terms. `Prop` has *proof-irrelevance* meaning that for type t of sort `Prop`, it holds that $\forall p_0\ p_1 : t.\ p_0 = p_1$. Intuitively this means that we consider distinct proofs of the same logical proposition t as equal. Returning to the example above, the essential point is that `O` should be distinct from `S O` and we therefore do not want *proof-irrelevance*, seen by its type `nat:Set`. Finally there is the sort `Type` which contains both `Set` and `Prop`, where definitions can be used both as informational types and logical propositions. Consider for example the definition of `list A` that is parameterized over type `A` of sort `Type`.

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A
```

This definition of lists can be used both for `Set` (list of natural numbers) and for `Prop` (list of proof terms). Equality is represented by the type `eq` and as equality is a logical proposition, it makes sense that it lies in `Prop`, seen by its type.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
| eq_refl : eq x x.
```

Coq is dependently typed, which informally means types can dependent on values. The type `eq` depends on the type A and value x, which equally can be thought of as A and x being universally quantified. The type `eq` is a binary predicate and the only way to prove it, is with `eq_refl`. For the specialized type `eq nat 2`, `eq_refl` has type `eq 2 2`. For the general type `eq`, the type of `eq_refl` is

```
@eq_refl : forall (A : Type) (x : A), x = x
```

To prove `eq 0 0`, notationally given as `0 = 0`, we construct the proof term the same way as we would define a typed expression in a functional language:

```
Definition three_plus_two : nat := 3 + 2
Definition eq0_proposition : 0 = 0 := eq_refl.
```

Another way is to use the Lemma keyword followed by the name of the lemma and its type. Below the first line we then build the proof using tactics. We construct the proof with the `exact` tactic, expecting a single argument which is the proof term that has the type of the logical proposition we want to prove.

```
Lemma eq0_lemma : 0 = 0.
Proof.
exact eq_refl.
Qed.
```

If the term does not have the correct type, the exact tactic will fail. Now consider the implication:

```
Lemma zero_i_zero : 0 = 0 -> 0 = 0.
Proof.
intros. exact H.
Qed.
```

The initial goal before any tactics have been used is:

```
1 subgoal
______________________________________(1/1)
0 = 0 -> 0 = 0
```

The intros tactic, performs introduction rules. In this case it performs implication introduction, introducing $0 = 0$ to the proof context.

```
1 subgoal
H : 0 = 0
______________________________________(1/1)
0 = 0
```

The apply tactic expects one argument H and if H is a closed term it must match the goal exactly. If H is an open term, i.e. it is a function with some arity n and type $t_0 \to t_1 \to ... \to t_n$, then t_n must match the goal and n new sub-goals are generated correponding to each of the n arguments to H. This can be seen in the small example below.

```
Lemma use_imp : 0=0.
Proof.
apply zero_i_zero. apply eq_refl.
Qed.
```

The first use of apply, replaces the current goal with the (identical) sub-goal $0 = 0$ which we then prove as before.

An important tactic for automation is `auto` and its use will be illustrated by an example. Consider the proof below showing $p + Sn > p + n$.

```
Lemma plus_Sn_gt : forall n m p : nat,
p + S n > p + n.
Proof.
intros.
apply Gt.plus_gt_compat_l. (*remove p on both sides*)
apply Gt.gt_Sn_n.
Qed.
```

The proof consists of three steps. First the universally quantified variables are introduced to the proof context. Then we apply the implication `Gt.plus_gt_compat_l` leaving us with a sub-goal where `p` is removed on both sides. Finally we apply `Gt.gt_Sn_n` which states that $Sn > n$ for all n. Because this proof uses only the `intros` and `apply` tactics, we can instead use `auto`, yielding the shorter proof

```
Lemma plus_Sn_gt : forall n m p : nat, p + S n > p + n.
Proof.
auto using Gt.plus_gt_compat_l, Gt.gt_Sn_n.
Qed.
```

`auto` solves a goal using only `intros` and `apply`. `auto` knows a small set of lemmas that it will try to use as arguments for `apply` and this set can be extended with the `using` clause, seen in the example above. If we have a large set of lemmas that are used often in this way, we can instead collect the lemmas in a hint database and instruct `auto` to use that database. The two lemmas, `Gt.plus_gt_compat_l` and `Gt.gt_Sn_n` are contained in the hint database `arith`, so an even shorter proof would be:

```
Lemma plus_Sn_gt : forall n m p : nat,
p + S n > p + n.
Proof.
```

```
auto with arith.
Qed.
```

If `auto` does not completely solve the goal its actions are reverted, leaving no effect on the proof state. This idempotent behaviour is convenient when the same invocation of `auto` is applied to multiple sub-goals.

We will end this chapter with proving that addition on the natural numbers is commutative and then try to shorten the proof.

To declare recursive functions, we use the `Fixpoint` directive. Addition is defined as:

```
Fixpoint add n m :=
  match n with
  | 0 => m
  | S p => S (p + m)
  end
where "n + m" := (add n m) : nat_scope.
```

The last line declares notation to use instead of the function name.
We now prove this function is commutative.

```
Lemma plus_comm : forall (n0 n1 : nat), n0 + n1 = n1 + n0.
Proof.
induction n0.
- intros. (*Case n0=O*)
  simpl.
  induction n1.
  * reflexivity. (*solves n0 + 0 = 0 + n0 *)
  * simpl. (*n0 + 0 = 0 + n0 *)
    rewrite <- IHn1.
    reflexivity.
- intros. (*Case n0=S n0'*)
  simpl.
  rewrite IHn0.
  rewrite plus_n_Sm.
  reflexivity.
Qed.
```

The tactic `induction`, applies the induction principle for natural numbers. Such a principle is implicitly declared for all inductive definitions. Applying `induction n0` produces the two sub-goals

```
2 subgoals
______________________________________(1/2)
```

```
forall n1 : nat, 0 + n1 = n1 + 0
______________________________________(2/2)
forall n1 : nat, S n0 + n1 = n1 + S n0
```

Bullets (-,*) are used to highlight the structure of the proofs. Proceeding with the first case starting at line 4, `intros` adds `n1` to the context and `simpl` evaluates the addition on the left, yielding `n1 = n1 + 0`. The addition on the right cannot be evaluated because `add` is defined by case distinction on its first argument. To show `n1+0 = n1` we do an inner induction on `n1` (l. 6-10), producing the subgoals:

```
2 subgoals
______________________________________(1/2)
0 = 0 + 0
______________________________________(2/2)
S n1 = S n1 + 0
```

The first sub-goal (l. 7) is solved by reflexivity, which corresponds to `exact eq_refl`. The second sub-goal (l. 8-10) simplifies `(Sn1) + 0` to `S(n1+0)`. From here we rewrite with induction hypothesis `n1 = n1 + 0` in the $\leftarrow$ direction (l. 9) and end with `reflexivity` (l. 10).
This finishes the first case of the outer induction proof. For the second case (l. 11-15), after fixing `n1` and simplifying `(S n0) + n1` to `S(n0 + n1)`, the induction hypothesis `forall n1, n0 + n1 = n1 + n0` is used as a rewrite in the $\rightarrow$ direction (l. 13). The remaining goal is then:

```
1 subgoal
n0 : nat
IHn0 : forall n1 : nat, n0 + n1 = n1 + n0
n1 : nat
______________________________________(1/1)
S (n1 + n0) = n1 + S n0
```

From the standard library we have available the fact that S can be grouped to the second plus-operand:

```
plus_n_Sm : forall n m : nat, S (n + m) = n + S m
```

Using this as a rewrite the proof is finished with `reflexivity`.

This proof can be shortened to:

```
Lemma plus_comm2 : forall (n0 n1 : nat), n0 + n1 = n1 + n0.
Proof.
induction n0;intros;simpl.
- induction n1; [ | simpl; rewrite <- IHn1]; reflexivity.
- rewrite IHn0. rewrite plus_n_Sm. reflexivity.
Qed.
```

The semicolon is used to sequence tactics. In line 3, the two generated sub-goals of `induction n0` are piped through `intros` and `simpl`. In line 4 this pattern is repeated with the slight change that the semicolon is proceeded by a squarebracketed expression. The notation `tac ; [tac1 | tac2 |...| tacn]` means `tac` generates n sub goals with tac_i applied to sub goal i. In our case `tac1` is not provided so the first sub-goal is not affected by this tactic.

4.2 Decision procedures

We can define decision procedures that return proofs. Consider the decision procedure for equality on `nat` that either returns a proof of `n = m` or `n <> m`.

```
Lemma eq_dec : forall n m : nat, {n = m} + {n <> m}.
```

The notation $\{A\} + \{B\}$, where A and B are logical propositions, is used for the type `sumbool`, the type for a boolean value that is accompanied by a proof. `sumbool` is defined in the standard library as:

```
Inductive sumbool (A B : Prop) : Set :=
| left : A -> {A} + {B}
| right : B -> {A} + {B}
```

Note `A` and `B` lies in Prop but `sumbool` lies in Set. For any `x` and `y` the type of `eq_dec x y` is then either a proof of `x=y` or `x <> y` and this fact can be used during computation. For example a `sumbool` can be used as a conditional in an if-statement with `left` corresponding to true and `right` as false. This is useful when doing case distinction in proofs because destructing (case distinction) the sumbool `{A}+{B}` generates two sub-goals, each replacing the `sumbool` with one of its constructors and adding its argument as an hypothesis in the proof state.

5 Mechanizing CSL_0

In this chapter we represent contracts, their satisfaction and monitoring semantics in Coq and mechanize the proofs we have seen so far.

5.1 Inductive definitions

Events are represented as:

```
Inductive EventType : Type :=
| Transfer : EventType
| Notify : EventType.

Scheme Equality for EventType.
```

The last line is a convenient utility for generating a decision procedure `EventType_eq_dec` that decides equality for `EventType`.

```
EventType_eq_dec : forall x y : EventType, {x = y} + {x <> y}
```

We define the type `Trace` as a shorthand for a list of events.

```
Definition Trace := list EventType % type.
```

A `Contract` is inductively defined.

```
Inductive Contract : Set :=
| Success : Contract
| Failure : Contract
| Event : EventType -> Contract
| CPlus : Contract -> Contract -> Contract
| CSeq : Contract -> Contract -> Contract.

Notation "c0 _;_ c1"  := (CSeq c0 c1)
                         (at level 52, left associativity).

Notation "c0 _+_ c1"  := (CPlus c0 c1)
                         (at level 53, left associativity).

Scheme Equality for Contract.
```

Notation is also added, so that for example `CSeq c0 c1` can be written as `c0_;_c1`. Like for `EventType`, the decision procedure for syntactically equal contracts is generated in the last line.

The satisfaction relation is represented by the `Matches_Comp`.

```
Inductive Matches_Comp : Trace -> Contract -> Prop :=
  | MSuccess : Matches_Comp [] Success
  | MEvent x : Matches_Comp [x] (Event x)
  | MSeq s1 c1 s2 c2
             (H1 : Matches_Comp s1 c1)
             (H2 : Matches_Comp s2 c2)
           : Matches_Comp (s1 ++ s2) (c1 _;_ c2)
  | MPlusL s1 c1 c2
                (H1 : Matches_Comp s1 c1)
              : Matches_Comp s1 (c1 _+_ c2)
  | MPlusR c1 s2 c2
                (H2 : Matches_Comp s2 c2)
              : Matches_Comp s2 (c1 _+_ c2).
```

For readability we declare some more familiar notation[1].

```
Notation "s (:) c" := (Matches_Comp s c)(at level 63).
```

[1]Here `(at level 63)` is a parsing rule, letting Coq know the binding strength of `(:)`

As an example, the event constructor should be read as, for all `x`, `MEvent x` proves `Matches_Comp [x] (Event x)`. Similarly the sequence constructor is read as, for all `s1 c1 s2 c2` along with hypotheses `H1` proving `Matches_Comp s1 c1` and `H2` proving `Matches_Comp s2 c2`, we may infer `Mathes_Comp (s1++s2) (c1 _;_ c2)`.

5.2 Monitoring semantics

The monitoring semantics is defined in terms of the functions `nu` and `derive`.

```
Fixpoint nu(c:Contract):bool :=
match c with
| Success => true
| Failure => false
| Event e => false
| c0 _;_ c1 => nu c0 && nu c1
| c0 _+_ c1 => nu c0 || nu c1
end.

Fixpoint derive (e:EventType) (c:Contract) :Contract :=
match c with
| Success => Failure
| Failure => Failure
| Event e' => if (EventType_eq_dec e' e) then Success else Failure
| c0 _;_ c1 => if nu c0 then
                  ((e \ c0) _;_ c1) _+_ (e \ c1)
                 else (e \ c0) _;_ c1
| c0 _+_ c1 => e \ c0 _+_ e \ c1
end
where "e \ c" := (derive e c).
```

Here the use of a `sumbool` in an if-statement can be seen in the case of `Event e'`.

Trace residuation is represented by `trace_derive`.

```
Fixpoint trace_derive (s : Trace) (c : Contract)  : Contract :=
match s with
| [] => c
| e::s' => s' \\ (e \ c)
end
where "s \\ c" := (trace_derive s c).
```

5.3 Equivalence proof

Equivalence of semantics (Theorem 3.5) is mechanized as:

```
Theorem Matches_Comp_iff_matchesb : forall (c : Contract)(s : Trace),
s (:) c <-> nu (s \\ c) = true.
Proof.
split;intros.
- auto using Matches_Comp_i_matchesb.
- generalize dependent c. induction s;intros.
  simpl in H. auto using Matches_Comp_nil_nu.
  auto using Matches_Comp_derive.
Qed.
```

Here the `split` tactic reduces the showing of ($\iff$) to the sub-goals ($\implies$) and ($\impliedby$). The structure of the proof is the same as for the paper proof. The ($\implies$) direction is by induction on `c` and ($\impliedby$) is by induction on s. For conciseness we will focus on the mechanization details of showing ($\implies$) as it demonstrates some principles about proof automation. This direction is shown by the lemma `Matches_Comp_i_matchesb`.

```
1 Lemma Matches_Comp_i_matchesb : forall (c : Contract)(s : Trace),
2 s (:) c -> nu (s\\c) = true.
3 Proof.
4 intros; induction H;
5 solve [ autorewrite with cDB; simpl; auto with bool
6       | simpl;eq_event_destruct;auto ].
7 Qed.
```

Here `intros` adds the assumption `s (:) c` to the context with name `H` and `induction H` applies induction on `H` producing 5 sub-goals. We use sequencing of tactics with the square-bracketed notation introduced in the last chapter. Here the first of the bracketed tactics (spanning l. 5) solves all generated sub-goals except for the case of MEvent, which is solved by the second bracketed tactic (spanning l. 6). The rest of this section will describe how these tactics solve their goals.

Automation with Hint databases

The first tactic solves each of the cases MSuccess, MPlusL, MPlusR and MSeq in three steps. First, the goal is rewritten as long as possible using rewrite rules from the user-defined hint database `cDB`. Secondly, the rewritten sub-goal is simplified, i.e. its expressions are evaluated as much as possible. Simplifying $nu\ Success$ results in $true$, but simplifying $nu\ c$ does not alter the expression. Thirdly, `auto` is applied with the hint database `bool`. Note that `autorewrite` and `auto` serve very different purposes. By applying `autorewrite` with a carefully defined hint database `cDB` and simplifying the result, the goal is rewritten into a form that only requires repeatedly applying proof terms, which is handled by `auto`.

A hint database may contain different kinds of hints. The two kinds of hints that have been used in `cDB` are rewrite hints (used by `autorewrite`) and resolve hints (used by `auto`). The hint database `bool` is defined in Coq's standard library and contain proofs related to boolean expressions such as commutativity of ||. Allowing a commutativity rule to be applied exhaustively would make rewriting non-terminating, as applying the rewrite always results in a new goal where it can be applied again. Therefore such rules must only be added as resolve hints.

The high-level aspects of the proof are handled by rewriting rules in `cDB` that have been proved separately. For the case of MSeq, the critical lemma in `cDB` is the one that states that a boolean match respects sequential composition given by `matchesb_seq`.

```
Lemma matchesb_seq : forall (s0 s1 : Trace)(c0 c1 : Contract),
nu (s0\\c0) = true -> nu (s1\\c1) = true ->
nu ((s0++s1)\\(c0 _;_c1)) = true.
```

Since this lemma has the shape $H_0 \implies H_1 \implies b_0 = b_1$, H_0 and H_1 are side-conditions that must be satisfied for b_0 to be rewritten to b_1.

The critical lemma in `cDB` for MPlusL and MPlusR shows that residuation distributes over plus.

```
Lemma derive_distr_plus : forall (s : Trace)(c0 c1 : Contract),
s \\ (c0 _+_ c1) = s \\ c0 _+_ s \\ c1.
```

Automation with specialized custom tactics

In the paper-proof, the case of MEvent was skipped because it was trivial. It is however not trivial enough to be handled by `auto` because we need to do a case distinction. Recalling that residuation for `Event e` is:

```
Fixpoint derive (e:EventType) (c:Contract) :Contract :=
match c with
...
| Event e' => if (EventType_eq_dec e' e) then Success else Failure
...
end
```

A case distinction has to be made on `EventType_eq_dec e' e` and this is automated by our tactic `eq_event_destruct`.

```
Ltac eq_event_destruct :=
  repeat match goal with
          | [ |- context[EventType_eq_dec ?e ?e0] ] =>
                destruct (EventType_eq_dec e e0);try contradiction
          | [ _ : context[EventType_eq_dec ?e ?e0]  |- _ ] =>
                destruct (EventType_eq_dec e e0);try contradiction
          end.
```

This tactic checks if there is some expression containing `EventType_eq_dec`, either in an assumption or the proof-goal and if that is the case, applies case distinction on `EventType_eq_dec e' e`. This produces two sub-goals, where the if-statement is reduced to either branch along with the added assumption $e' = e$ or $e' \neq e$ to the proof state. If e' and e are the same event, say Transfer, then the presence of an assumption $Transfer \neq Transfer$ solves the sub-goal by contradiction. The tactic consists of a repeat-loop with a match on `goal` which is the list of assumptions in the proof-context where the tactic is invoked along with the current statement to be proved, written as $[H_0\ H_1\ ...\ H_n\ \vdash P]$. The first case looks in the current goal and the second case looks in the assumptions.

5.4 Mechanizing axiomatization

The 12 aximos and 4 inference rules of the inference system is represented by the type `c_eq`.

```
Inductive c_eq : Contract -> Contract -> Prop :=
  | c_plus_assoc c0 c1 c2 : (c0 _+_ c1) _+_ c2 == c0 _+_ (c1 _+_ c2)
  | c_plus_comm c0 c1: c0 _+_ c1 == c1 _+_ c0
  | c_plus_neut c: c _+_ Failure == c
  | c_plus_idemp c : c _+_ c == c
  | c_seq_assoc c0 c1 c2 : (c0 _;_ c1) _;_ c2 == c0 _;_ (c1 _;_ c2)
  | c_seq_neut_l c : (Success _;_ c) == c
  | c_seq_neut_r c : c _;_ Success == c
  | c_seq_failure_l c : Failure _;_ c == Failure
  | c_seq_failure_r c :  c _;_ Failure == Failure
  | c_distr_l c0 c1 c2 : c0 _;_ (c1 _+_ c2) ==
                         (c0 _;_ c1) _+_ (c0 _;_ c2)
  | c_distr_r c0 c1 c2 : (c0 _+_ c1) _;_ c2 ==
                         (c0 _;_ c2) _+_ (c1 _;_ c2)
  | c_refl c : c == c
  | c_sym c0 c1 (H: c0 == c1) : c1 == c0
  | c_trans c0 c1 c2 (H1 : c0 == c1)
                     (H2 : c1 == c2) : c0 == c2
  | c_plus_ctx c0 c0' c1 c1' (H1 : c0 == c0')
                             (H2 : c1 == c1') : c0 _+_ c1 == c0' _+_ c1'
  | c_seq_ctx c0 c0' c1 c1' (H1 : c0 == c0')
                            (H2 : c1 == c1') : c0 _;_ c1 == c0' _;_ c1'
  where "c1 == c2" := (c_eq c1 c2).
```

Soundness

Like the paper-proof, the mechanized soundness proof is by induction on the derivation of `c0 == c1` which in the proof below is introduced to the context with name `H`.

```
Lemma c_eq_soundness : forall (c0 c1 : Contract),
c0 == c1 -> (forall s : Trace, s (:) c0 <-> s (:) c1).
Proof.
intros c0 c1 H. induction H ;intros;
                  try solve [split;intros;c_inversion].
  * split;intros;c_inversion; [ rewrite <- app_assoc |
                                rewrite app_assoc ]
                             ; auto with cDB.
  * rewrite <- (app_nil_l s). split;intros;c_inversion.
  * rewrite <- (app_nil_r s) at 1. split;intros;c_inversion. subst.
        repeat rewrite app_nil_r in H1. now rewrite <- H1.
  * now symmetry.
  * eauto using iff_trans.
  * split;intros; inversion H1; [ rewrite IHc_eq1 in H4
                                | rewrite IHc_eq2 in H4
                                | rewrite <- IHc_eq1 in H4
                                | rewrite <- IHc_eq2 in H4]
                              ;auto with cDB.
  * split;intros; c_inversion; constructor;
                                [ rewrite <- IHc_eq1
                                | rewrite <- IHc_eq2
                                | rewrite IHc_eq1
                                | rewrite IHc_eq2]
                              ;auto.
Qed.
```

Most of the axiom-cases are similar. First s is fixed, then ($\Longleftrightarrow$) is split into showing ($\Longrightarrow$) and ($\Longleftarrow$). For each of these two sub-goals we reason by what rule the assumption must have ended in. Doing this provides us either new assumptions, instantiates fixed but arbitrary variables or allows us to finish the proof by contradiction. These steps followed by `auto` is enough to solve these cases. Reasoning by what rule a derivation must have ended in is a principle called *inversion* and the Coq standard library defines the tactic `inversion` that applies this principle. Using this tactic we can define a higher-level tactic `c_inversion`, tailored to the case where an assumption of shape `s:c` is present in the proof context. It applys inversion repeatedly until it fails and then applies `auto`.

```
Ltac c_inversion :=
  (repeat match goal with
         | [ H: _ (:) Failure |- _ ] => inversion H
         | [ H: ?s (:) _ _+_ _ |- _ ] => inversion H; clear H
         | [ H: ?s (:) _ _;_ _ |- _ ] => inversion H; clear H
         | [ H: [] (:) Success |- _ ] => fail
         | [ H: _ (:) Success |- _ ] => inversion H; clear H
         end);auto with cDB.
```

The tactic checks if one of five assumption shapes are present in the proof context, with underscore used as wildcard. Whereas the first case solves the goal by contradiction and the two cases that follow replace H with its premises, the last case can be matched repeatedly. To avoid this, case four explicitly fails, exiting the repeat loop.

The axiom cases that cannot be completely automated in this way are `c_seq_assoc`, `c_seq_neut_l` and `c_seq_neut_r`. These cases are harder to automate completely because they require specific rewrite rules about trace concatenation. Each of the cases requires rewriting `s=[]++s` or `s=s++[]` either in the $\rightarrow$ or $\leftarrow$ direction, possibly more than once.

Returning to the soundness proof `c_eq_soundness`, most cases are solved in line 5 with the sequenced tactic `split;intros;c_inversion`. Here `split` reduces showing ($\iff$) to showing ($\implies$) and ($\impliedby$) and `intros` performs the implication introduction, so that the assumption can be found by `c_inversion`. The indented bullets $\times$ indicate the remaining sub-goals, which there are seven of. The first three are `c_seq_assoc`, `c_seq_neut_l` and `c_seq_neut_r` while the remaining four are symmetry, transitivity and context-rules. The context-rules whose proofs start resp. at line 14 and line 19 each has four cases, only differing in which of the two induction hypotheses to use and in what direction.

Completeness

In the formalization we sometimes referred to the summation over sets of contracts (e.g. $\Sigma_{e \in E}\ e \backslash c$) and over sequences of contracts (e.g. $\Sigma_{i=1}^{n} c_i$). In the mechanization, sets and sequences of contracts will both be represented by lists of contracts. As a consequence of this, the fact that satisfaction equivalence implies trace equivalence is no longer immediate, but most be shown. We start by defining the projection function `L`.

```
Fixpoint L (c : Contract) : list Trace :=
match c with
| Success => [[]]
| Failure => []
| Event e => [[e]]
| c0 _+_ c1 => (L c0) ++ (L c1)
| c0 _;_ c1 => map (fun p => (fst p)++(snd p)) (list_prod (L c0) (L c1))
end.
```

Here the function `list_prod:forall A B : Type, list A -> list B -> list (A * B)`, returns the product of the input lists as a tupled list. We map over this list, replacing tuples of traces by their concatenation.

For c_0 and c_1 to be trace equivalent in terms of their trace lists, the elements of

$L(c_0)$ and $L(c_1)$ must coincide. This can succinctly be expressed with the abbreviation `incl` which corresponds to set inclusion on lists.

```
incl =
fun (A : Type) (l m : list A) => forall a : A, In a l -> In a m
     : forall A : Type, list A -> list A -> Prop
```

We use `incl` in the lemma `Matches_eq_i_incl_and` which proves that satisfaction equivalence implies trace equivalence.

```
Theorem Matches_eq_i_incl_and : forall (c0 c1 : Contract),
(forall (s : Trace), s (:) c0 <-> s (:) c1) ->
incl (L c0) (L c1) /\ incl (L c1) (L c0) .
Proof.
intros. apply comp_equiv_destruct in H.
destruct H. split; auto using Matches_incl.
Qed.
```

We use `comp_equiv_destruct` to split the assumption $\forall s.\ s : c_0 \iff s : c_1$ into the two assumptions $\forall s.\ s : c_0 \implies s : c_1$ and $\forall s.\ s : c_1 \implies s : c_0$, from which we prove each of the conjuncts in the goal using the helper lemma `Matches_incl`.

```
Lemma Matches_incl : forall (c0 c1 : Contract),
(forall (s : Trace), s (:) c0 -> s (:) c1) -> incl (L c0) (L c1).
```

Representing Σ Σ is represented as a function that folds a list of contracts by $+$.

```
Fixpoint Σ (l : list Contract) : Contract :=
match l with
 [] => Failure
 c ::l => c _+_ (Σ l)
end.
```

We can show some properties of Σ.

Σ is associative:

```
Lemma Σ_app : forall (l1 l2 : list Contract),
Σ (l1 ++ l2) == (Σ l1) _+_ (Σ l2).
```

Σ is idempotent:

```
Lemma incl_Σ_idemp : forall (l1 l2 : list Contract),
incl l1 l2 -> Σ l2 == Σ (l1++l2).
```

Σ is commutative:

```
Lemma Σ_app_comm : forall (l1 l2 : list Contract),
Σ (l1++l2) == Σ (l2++l1).
```

From these properties we can show that for the coinciding lists of contracts `l1` and `l2`, their summations are derivably equivalent:

```
Theorem incl_Σ_c_eq : forall (l1 l2 : list Contract),
incl l1 l2 -> incl l2 l1-> Σ l1 == Σ l2.
```

Representing $\prod$ The representation of $\prod$ is:

```
Fixpoint ∏ (s : Trace) :=
match s with
 [] => Success
 e::s' => (Event e) _;_ (∏ s')
end.
```

$\prod$ is associative:

```
Lemma ∏_app : forall (l1 l2 : Trace), ∏ l1 _;_ ∏ l2 == ∏ (l1++l2).
```

In the paper proof by distributibity of sequence over addition, we simply assumed:

$$\Sigma_{s\in L(c_0)} \prod_{i=1}^{n_s} s^i \ ; \ \Sigma_{s\in L(c_1)} \prod_{i=1}^{n_s} s^i == \Sigma_{s_0\in L(c_0)} \Sigma_{s_1\in L(c_1)} \prod_{i=1}^{n_{s0}} s_0^i \ ; \ \prod_{j=1}^{n_{s1}} s_1^j .$$

This is proved by:

```
Lemma ∏_distr : forall l0 l1,
  Σ (map ∏ l0) _;_ Σ (map ∏ l1) ==
  Σ (map (fun x => ∏ (fst x ++ snd x)) (list_prod l0 l1)).
```

Note that the statement that is proved by $\prod$_distr is slightly different because the $\prod$_app lemma already has been applied inside the map. We might have expected the lemma to instead look like

```
Lemma ∏_distr2 : forall l0 l1,  Σ (map ∏ l0) _;_ Σ (map ∏ l1) ==
 Σ (map (fun x => ∏ (fst x) _;_ ∏ (snd x)) (list_prod l0 l1)).
```

This would be an inconvenient way to present the lemma, as $\prod$_app lemma cannot easily be applied inside the mapped function. The reason for this is that we are trying to assert that with respect to derivable equivalence, the functions $f(x) := \prod fst\ x + +snd\ x$ and $f'(x) := \prod fst\ x\_;\_\prod snd\ x$ are equal. This would have to be proved as a separate theorem and to avoid this, it yields a shorter proof to show $\prod\_distr$ in terms of f directly. In the later mechanization chapters we will start to reason about functions that are mapped over summations.

Preseriving derivable equivalence The theorem used to show that derivable equivalence is preserved is:

```
Theorem ∏_L_ceq : forall (c : Contract), Σ (map ∏ (L c)) == c.
Proof.
induction c; simpl; try solve [auto_rwd_eqDB].
- rewrite map_app. rewrite Σ_app.
  auto using c_plus_ctx.
- rewrite map_map.
  rewrite <- IHc1 at 2. rewrite <- IHc2 at 2.
  symmetry. apply ∏_distr.
Qed.
```

The structure of the proof is very similar to the paper proof. The immediate cases are solved by `auto_rwd_eqDB` which is a shorthand for `autorewrite with eqDB;auto with eqDB`. The case of addition is solved by decomposing the summation and applying the IHs. The case of sequence first rewrites the IHs before appealing to distributivity of sequence over addition.

The final completeness proof is then:

```
Lemma c_eq_completeness : forall (c0 c1 : Contract),
(forall s : Trace, s (:) c0 <-> s (:) c1) -> c0 == c1.
Proof.
intros. rewrite <- ∏_L_ceq. rewrite <- (∏_L_ceq c1).
apply Matches_eq_i_incl_and in H.
destruct H. auto using incl_map, incl_Σ_c_eq.
Qed.
```

6 Formalizing $CSL_{||}$

We now extend the language with parallel composition, written as $||$.

$$c := ...|\ c_0||c_1$$

$||$ is left associtative and binds weaker than ;, but tighter than $+$. For example, the contract $c_0 + c_1; c_2||c_3$ is parsed as $c_0 + ((c_1; c_2)||c_3)$. We extend the satisfaction with the rule MPar.

$$\frac{s_0 : c_0 \qquad s_1 : c_1 \qquad (s_1, s_2) \rightsquigarrow s}{s : c_0||c_1}\ \text{MPar}$$

Here $(s_0, s_1) \rightsquigarrow s$ means that s is an interleaving of s_0 and s_1. As an example, all permutations of the trace $TTNN$ are satisfied by the contract $T; N||N; T$.

$nu(c_0||c_1)$ is defined as:

$$nu(c_0||c_1) := \begin{cases} 1, & \text{if } nu\ c_0 = nu\ c_1 = 1 \\ 0, & \text{otherwise} \end{cases}$$

Residuation on $c_0||c_1$ is defined as:

$$e\backslash c_0||c_1 := (e\backslash c_0)||c_1 + (c_0||e\backslash c_1)$$

We expect the two semantics to remain equivalent after these extensions.

Theorem 6.1 (Equivalence of semantics) *For all s c, $s : c \iff nu(s\backslash\backslash c) = 1$*
Proof of ($\implies$) is by induction on the derivation of $s : c$ and ($\impliedby$) is by induction on c (not shown).

As parallel composition is based on the interleaving of traces we would additionally expect that properties of interleavings to also hold for contracts that are composed with $||$. For example if s is an interleaving of s_0 and s_1,$(s_0, s_1) \rightsquigarrow s$, then its also the case that s is an interleaving of s_1 and s_0, $(s_1, s_0) \rightsquigarrow s$. This suggests that $||$ is a commutative operator. By a similar argument $||$ should also be associtative. The axiomatization for $CSL_{||}$ is the axiomatization for CSL_0 extended with the rules in Figure 10.

$$\begin{aligned} c_0||c_1||c_2 &== c_0||(c_1||c_2) && (13) \\ c||Success &== c && (14) \\ c||Failure &== Failure && (15) \\ c_0||(c_1 + c_2) &== c_0||c_1 + c_0||c_2 && (16) \\ e_0; c_0||e_1; c_1 &== e_0; (c_0||e_1; c_1) + e_1; (e_0; c_0||c_1) && (17) \end{aligned}$$

$$\frac{c_0 == c_0' \qquad c_1 == c_1'}{c_0||c_1 == c_0'||c_1'} \text{ (Ctx-par)}$$

Figure 10: Axiomatization of contract equivalence for $CSL_{||}$

Theorem 6.2 (Soundness of axiomatization) *For all contracts c_0 c_1, $c_0 == c_1$ implies $\forall s.\ s :\ c_0 \iff s :\ c_1$*
Proof by induction derivation of $c_0 == c_1$ (not shown).

6.1 Completeness

The essential property of $||$ that will allow us to reuse the completeness of the axiomatization for CSL_0 is that $||$ can be eliminated, such that all contracts c that contains parallel composition, can be normalized into a form $|c|$ that lies in the set

CSL_0. Completeness then reduces to showing that c and $|c|$ are both satisfiably equivalent and derivably equivalent in our axiomatization for $CSL_{||}$.

Thinking operationally, the defining charicteristics of a contract is if it is nullable what its set of residual contracts are. Whether a contract is nullable or not can be represented as contracts.

$$o(c) := \begin{cases} Success, & \text{if } nu\ c = 1 \\ Failure, & \text{otherwise} \end{cases}$$

It should then be possible to write c as $o(c) + \Sigma_{e \in E} e; e \backslash c$. Whether c is nullable is captured by $o(c)$ and the set of residual contracts is captured by $\Sigma_{e \in E} e; e \backslash c$. The normalization of a contract c is then applying this rewrite for c and recursively on the residuals of c. We must take care to ensure such a procedure terminates. Consider a naive definition of the normalization procedure:

$$|c| = \begin{cases} Failure, & \text{if } c = Failure \\ o(c) + \Sigma_{e \in E} e; |e \backslash c|, & \text{otherwise} \end{cases}$$

This definition does not always terminate. As an example $|Failure + Failure|$ is undefined because it unfolds to $o(Failure + Failure) + \Sigma_{e \in E} e; |Failure + Failure|$. A more general base case is needed, which captures contracts that in some sense are stuck. A stuck contract could be defined as one that is equal to all its residuals, as is the case for $Failure + Failure$. This does however not cover the case of $Failure||Failure$, whose sole residual is $e \backslash Failure||Failure = Failure||Failure + Failure||Failure$. To capture the stuckness of a contract we define the $Stuck\ c$ judgment inductively seen in Figure 11.

$$\frac{}{Stuck\ Failure} \qquad \frac{Stuck\ c_0 \qquad Stuck\ c_1}{Stuck\ c_0 + c_1} \qquad \frac{Stuck\ c_0}{Stuck\ c_0; c_1}$$

$$\frac{Stuck\ c_0}{Stuck\ c_0||c_1} \qquad \frac{Stuck\ c_1}{Stuck\ c_0||c_1}$$

Figure 11: Stuck judgment

With E being our alphabet, the normalization function can be defined as

$$|c| = \begin{cases} Failure, & \text{if } Stuck\ c \\ o(c) + \Sigma_{e \in E} e; |e \backslash c|, & \text{otherwise} \end{cases}$$

Lemma 6.3 (Termination of normalization) *For all contracts c, $|c|$ terminates. Proof is by induction on c (not shown).*

Because $|\cdot|$ terminates we may prove properties about it by *functional induction*, giving us an IH on the argument of the recursive call. $|\cdot|$ can equivalently be represented inductively by the judgment $norm\ c\ c'$.

$$\frac{Stuck\ c}{norm\ c\ Failure} \qquad \frac{\forall e \in E.\ norm\ e\backslash c\ c_e \qquad \neg Stuck\ c}{norm\ c\ \big(o(c) + \Sigma_{e\in E} e c_e\big)}$$

We will use this induction principle in showing that normalization preserves satisfaction.

Lemma 6.4 (Normalization preserves satisfaction) *For all contracts c,* $\forall s.\ s:\ c \iff s:\ |c|$

The proof is by functional induction on $|c|$.

- Case $|c| = Failure$.
  We may assume $H\ :\ Stuck\ c$. By induction on H it is straightforward to show

$$\forall s.\ s:\ c \iff s:\ Failure$$

- Case $|c| = o(c) + \Sigma_{e\in E} e; |e\backslash c|$.
  We have the following induction hypothesis on the recursive call:

$$\forall e\ s.\ s:\ e\backslash c \iff s:\ |e\backslash c|$$

  We now do case distinction on s

  - Sub case: $s = \langle\rangle$.
    We must show:

$$\langle\rangle : c \iff \langle\rangle : o(c) + \Sigma_{e\in E} e; |e\backslash c|$$

    $\langle\rangle : o(c) + \Sigma_{e\in E} e; |e\backslash c|$ must have ended in MPlusL as $\Sigma_{e\in E} e; |e\backslash c|$ is not nullable. It suffices to show:

$$\langle\rangle : c \iff \langle\rangle : o(c)$$

    Now by case distinction on $nu(c)$, case $nu(c) = 1$ is immediate. For case $nu(c) = 0$, ($\implies$) is contradictory as we assume both $\langle\rangle : c$ and $nu(c) = 0$. ($\impliedby$) is contradictory as we assume $\langle\rangle : Failure$.
  - Sub case: $s = e's'$ We must show:

$$e's' : c \iff e's' : o(c) + \Sigma_{e\in E} e; |e\backslash c|$$

    $e's'\ : o(c) + \Sigma_{e\in E} e; |e\backslash c|$ must have ended in MPlusR for both cases of $o(c)$. It suffices to show

$$e's' : c \iff e's' : \Sigma_{e\in E} e; |e\backslash c|$$

    The sum $\Sigma_{e\in E} e; |e\backslash c|$ consists of sequences, each beginning with a distinct event from our finite set of events. Naturally only the summand $e'; |e'\backslash c|$ can possibly be satisfied by the trace $e's'$, therefore any derivation of $\Sigma_{e \in E} event\ e; |e \backslash c|$ must have been a repeated application of MPlusL and MPlusR with an initial premise $e's' : e'; |e' \backslash c|$. After having applied the IH, it suffices to show:

$$e's' : c \iff e's' : e'; e' \backslash c$$

Which after residuation on the right side is

$$s' : e' \backslash c \iff s' : Success; e' \backslash c$$

Which we know holds. This ends the proof.

We now show that normalization is derivable in the axiomatization from which we show completeness (Theorem 6.6).

Theorem 6.5 (Derivabilitity of Normalization) *For all contracts c, $c == |c|$*

We proceed by induction on c. We show only the case for $*$.
We must show

$$c_0 * c_1 == o(c_0 * c_1) + \Sigma_{e \in E}\ event\ e; e \backslash (c_0 * c_1)$$

We apply the IHs on the left hand side, yielding

$$\big(o(c_0) + \Sigma_{e \in E}\ event\ e; e \backslash c_0\big) * \big(o(c_1) + \Sigma_{e \in E}\ event\ e; e \backslash c_1\big) == \\ o(c_0 * c_1) + \Sigma_{e \in E}\ event\ e; e \backslash (c_0 * c_1)$$

After distribution this yields

$$o(c_0) * o(c_1) + o(c_0) * \big(\Sigma_{e \in E} e; e \backslash c_1\big) + \big(\Sigma_{e \in E} e; e \backslash c_0\big) * o(c_1)\ + \big(\Sigma_{e \in E} e; e \backslash c_0\big) * \big(\Sigma_{e \in E} e; e \backslash c_1\big) \\ == \\ o(c_0 * c_1) + \Sigma_{e \in E} e; e \backslash (c_0 * c_1)$$

We know that $o(c_0) * o(c_1) == o(c_0 * c_1)$ cancelling out the left most terms. After computing the residual on the right hand side and decomposing the sum, we get.

$$o(c_0) * \big(\Sigma_{e \in E} e; e \backslash c_1\big) + \big(\Sigma_{e \in E} e; e \backslash c_0\big) * o(c_1) + \big(\Sigma_{e \in E} e; e \backslash c_0\big) * \big(\Sigma_{e \in E} e; e \backslash c_1\big) \\ == \\ \big(\Sigma_{e \in E} e; (e \backslash c_0 * c_1)\big) + \big(\Sigma_{e \in E} e; (c_0 * e \backslash c_1)\big)$$

We now apply the IHs on the right hand side.

$$o(c_0) * \big(\Sigma_{e \in E} e; e \backslash c_1\big) + \big(\Sigma_{e \in E} e; e \backslash c_0\big) * o(c_1) + \big(\Sigma_{e \in E} e; e \backslash c_0\big) * \big(\Sigma_{e \in E} e; e \backslash c_1\big) \\ == \\ \big(\Sigma_{e \in E} e; (e \backslash c_0 * (o(c_1) + \Sigma_{e' \in E} e'; e' \backslash c_1)\big) + \big(\Sigma_{e \in E} e; ((o(c_0) + \Sigma_{e' \in E} e'; e' \backslash c_0) * e \backslash c_1)\big)$$

After distributing we get

$$o(c_0) * \big(\Sigma_{e\in E} e; e\backslash c_1\big) + \big(\Sigma_{e\in E} e; e\backslash c_0\big) * o(c_1) + \big(\Sigma_{e\in E} e; e\backslash c_0\big) * \big(\Sigma_{e\in E} e; e\backslash c_1\big)$$
$$==$$
$$\big(\Sigma_{e\in E} e; (e\backslash c_0 * o(c_1))\big) + \big(\Sigma_{e\in E} e; (e\backslash c_0 * \Sigma_{e'\in E} e'; e'\backslash c_1)\big) +$$
$$\big(\Sigma_{e\in E} e; (o(c_0) * e\backslash c_1)\big) + \big(\Sigma_{e\in E} e; \big(\Sigma_{e'\in E} e'; e'\backslash c_0\big) * e\backslash c_1\big)$$

We know that $o(c_0) * \Sigma_{e\in E} e; e\backslash c_1 == \Sigma_{e\in E} e; (o(c_0) * e\backslash c_1)$ because if $o(c_0) = Success$, the term is reduced away and in $o(c_0) = Failure$ both terms reduce to Failure. By the same argument $\big(\Sigma_{e\in E} e; e\backslash c_1\big) * o(c_1) == \Sigma_{e\in E} e; (e\backslash c_0 * o(c_1))$. We match these terms against each other, leaving us to show

$$\big(\Sigma_{e\in E} e; e\backslash c_0\big) * \big(\Sigma_{e\in E} e; e\backslash c_1\big) == \big(\Sigma_{e\in E} e; (e\backslash c_0 * \Sigma_{e'\in E} e'; e'\backslash c_1)\big) + \big(\Sigma_{e\in E} e; \big(\Sigma_{e'\in E} e'; e'\backslash c_0\big) * e\backslash c_1\big)$$

We now apply distributivity of $*$ on the left hand side

$$\big(\Sigma_{e\in E}\Sigma_{e'\in E}(e; e\backslash c_0) * (e'; e'\backslash c_1)\big) ==$$
$$\big(\Sigma_{e\in E} e; (e\backslash c_0 * \Sigma_{e'\in E} e'; e'\backslash c_1)\big) + \big(\Sigma_{e\in E} e; \big(\Sigma_{e'\in E} e'; e'\backslash c_0\big) * e\backslash c_1\big)$$

We now apply axiom (17) on the left, distributing afterwards.

$$\big(\Sigma_{e\in E}\Sigma_{e'\in E} e; (e\backslash c_0 * (e'; e'\backslash c_1))\big) + \big(\Sigma_{e\in E}\Sigma_{e'\in E} e'; (e; e\backslash c_0) * e'\backslash c_1\big) ==$$
$$\big(\Sigma_{e\in E} e; (e\backslash c_0 * \Sigma_{e'\in E} e'; e'\backslash c_1)\big) + \big(\Sigma_{e\in E} e; \big(\Sigma_{e'\in E} e'; e'\backslash c_0\big) * e\backslash c_1\big)$$

Finally these terms are derivably equivalent after distributing over ; and $*$. This ends the proof.

Theorem 6.6 (Completeness) *For all contracts c_0 c_1, $\forall s.\ s : c_0 \iff s : c_1$ implies $c_0 == c_1$*

By lemma 6.4 we have that $\forall s.\ s : |c_0| \iff s : |c_1|$ and by lemma 6.5 it suffices to show $|c_0| == |c_1|$. Because the image of $|\cdot|$ is contained in CSL_0, we finally appeal to the completeness of the axiomatization of CSL_0.

7 Mechanizing $CSL_{||}$

Unlike the previous mechanization chapter which in detail showed how reasoning from the formalization translated to the implementation in Coq, we will due to the size of the development in this chapter only highlight the essential aspects of the mechanization of $CSL_{||}$. To disambigute definitions given in the two distinct Coq projects, when in doubt we will prefix the name by Core when refering to CSL_0 and Parallel when reffering to $CSL_{||}$.

7.1 The Interleave predicate

The satisfaction predicate for the Parallel.Contract type is nearly the same as the one for Core.Contract, with this new constructor for the parallel operator:

```
Inductive Matches_Comp : Trace -> Contract -> Prop :=
(...)
  | MPar s1 c1 s2 c2 s
             (H1 : s1 (:) c1)
             (H2 : s2 (:) c2)
             (H3 : interleave s1 s2 s)
           : s (:) (c1 _||_ c2)
  where "s (:) c" := (Matches_Comp s c).
```

The interleave predicate is defined as:

```
Inductive interleave (A : Set) : list A -> list A -> list A -> Prop :=
| IntLeftNil t : interleave nil t t
| IntRightNil t : interleave t nil t
| IntLeftCons t1 t2 t3 e (H: interleave t1 t2 t3) :
                        interleave (e :: t1) t2 (e :: t3)
| IntRightCons t1 t2 t3 e (H: interleave t1 t2 t3) :
                         interleave t1 (e :: t2) (e :: t3).
```

This definition intuitively defines an interleaving compositionally. Given some interleaving, `interleave s0 s1 s`, then we also have the interleaving `interleave e::s0 s1 e::s`. Because this definition is intuitive, we trust that it captures the meaning of $(s_0, s_1) \rightsquigarrow s$. On the other hand this definition is not convenient for proving properties about interleavings. If we were to prove a lemma by induction on the structure of interleavings, the cases of IntLeftCons and IntRightCons do not provide a strong induction hypothesis. For example for IntLeftCons one must from `interleave s0 s1 s` show `interleave e::s0 s1 e::s`, where `s0` and `s1` are fixed. Interleavings can be defined in another way that though less intuitive, allows a stronger induction hypothesis.

```
Fixpoint interleave_fun (A : Set) (l0 l1 l2 : list A ) : Prop :=
match l2 with
| [] => l0 = [] /\ l1 = []
| a2::l2' => match l0 with
             | [] => l1 = l2
             | a0::l0' => a2=a0 /\ interleave_fun l0' l1 l2'
                \/ match l1 with
                   | [] => l0 = l2
                   | a1::l1' => a2=a1 /\ interleave_fun l0 l1' l2'
                   end
            end
end.
```

`interleave_fun` is a function that given three lists compute a Prop. As an example for any `l0` and `l1`, `interleave_fun l0 l1 []` evaluates to the proposition `l0 = [] ∧ l1 = []`. To prove a property about interleavings in terms of the `interleave_fun` predicate, one can simply do induction on the last parameter `l2`. This allows a much stronger induction hypothesis as `l0` and `l1` may be universally quantified. Naturally these two predicates must be shown to be equivalent but this is straightforward to show.

7.2 Well-foundedness of Normalization

When defining functions in Coq where termination is not obvious, one can use the Equations package which allows one to accompany the function definition with a well-founded relation R. One must then prove that the input argument in_{arg} and argument of the recurisvel call in_{rec} lie in a well-founded relation R ($R\ in_{arg}\ in_{rec}$). A well-founded relation is one that does not contain infinitely descending sequences. As an example, $<$ on the natural numbers is a well-founded relation as any decreasing sequence of natural numbers wrt. to $<$ must end in 0. On the other hand any decreasing sequence wrt. to $\leq$ may be infinitely long. Showing in_{arg} and in_{rec} are related by a wellfounded relation is therefore a termination proof. The wellfounded relation we use in defining the normalization function `plus_norm` relates two contracts iff the first is smaller in size than the other, {`(c0,c1)| con_size c0 < con_size c1`}. Size is defined by the function `con_size`.

```
Fixpoint con_size (c:Contract):nat :=
match c with
| Failure => 0
| Success => 1
| Event _ => 2
| c0 _+_ c1 => max (con_size c0) (con_size c1)
| c0 _;_ c1 => if stuck_dec c0 then 0 else (con_size c0) +
                                             (con_size c1)
| c0 _*_ c1 => if  sumbool_or _ _ _ _ (stuck_dec c0)
                                        (stuck_dec c1)
                    then 0 else (con_size c0) +
                                 (con_size c1)
end.
```

This definition was motivated by a few desirable properties. Firstly, if a contract is stuck then its size should be 0. This can be seen to be enforced by the stuckness tests in case `_;_` and `_||_` that use the decision procedure `stuck_dec`. Secondly, if a contract is not stuck, it should decrease in size. These properties hold for the size function, but the proof of the second property is long and convoluted. This is a consequence of computing the size of `c0 + c1` using the max operation and the local tests of stucknesss in `_;_` and `_||_`. Had time allowed, `con_size` could have been defined by an initial test of stuckness, returning zero for a stuck contract and proceeding by case distinction otherwises (without local tests of stuckness). This would result in a much simpler proof.

Normalization is represented by the function `plus_norm`, using the Equations directive to allow us to specify the wellfounded relation

```
Equations plus_norm (c : Contract) : (Contract) by  wf (con_size c) :=
plus_norm  c :=
  if stuck_dec c
    then Failure
    else (o c) _+_
         Σ alphabet (fun e => (Event e) _;_ (plus_norm (e \ c))).
```

On the right of the "by" clause on the first line, we provide the input contract's size `(con_size c)`. As the well-founded relation is not provided, it defaults to $<$. To prove termination of `plus_norm` we must show that if `c` is not stuck (the condition for entering the else-branch), then for all events `e`,
`con_size (e \ c) < con_size c`, which was one of the properties we ensured that `con_size` satisfied.

7.3 Distributivity

An important issue is how one represents distributivity laws. In our last chapter, our approach to showing derivability of the normalization function, was mostly by distributivity rules. Recall that summation in the mechanization of CSL_0 was implemented in Coq as a function that folded a list of contracts with $+$. In the mechanization of $CSL_{||}$, we will paramterize the summation by a function, mapped over the folded list. This simplifies expressions that are summations over mapped lists. In the mechanization of CSL_0, mapping a function `f:Contract -> Contract` over a list of contracts `cs` and taking the sum, would be written as `Σ (map f cs)`. Defining Σ parameterized over a mapping function `f`, lets us write mapped summation as `Σ cs f`.

```
Fixpoint Σ (A:Type) (l : list A) (f : A -> Contract) : Contract :=
match l with
 [] => Failure
 c ::l => f c _+_ (Σ l f)
end.
```

We can define lemmas that factor terms out of the summation. As an example consider distributivity of ; over $+$ lifted to Σ:

```
Lemma Σ_factor_seq_l : forall l (P: EventType -> Contract) c,
Σ l (fun a  => c _;_ P a) == c _;_ Σ l (fun a  => P a).
```

We can also define lemmas that directly manipuate the function argument of Σ:

```
Lemma Σ_distr_par_l_fun : forall f0 f1 f2, f0   λ||λ (f1 λ+λ f2) =λ=
f0 λ||λ f1 λ+λ f0 λ||λ f2.
```

This lemma is stated in terms of the relation =λ=. This is an equivalence on functions of type `EvenType -> Contract`. `f0=λ=f1` is short for `forall e, f0 e == f1 e`. `f0` and `f1` can be thought of as contexts, each abstracted over an event. Just as _+_ lets us combine contracts we define the operator λ+λ for combining contexts in a similar way. This operator is notation for the function `plus_fun`.

```
Definition plus_fun (f0 f1 : EventType -> Contract) :=
    fun a => f0 a _+_ f1 a.
```

The other operators, `λ;λ` and `λ||λ` are defined similarly. The power of defining these operators is that Coq can be instructed to not unfold them. If we allow Coq to unfold the definition, then `f0λ+λf1` simplifies to `fun a => f0 a + f1 a` which disallows (or at least makes challenging) the later rewriting of `f0` and `f1` as they are located under the binder `a`. Instructing Coq not to unfold `plus_fun` lets us work with it as a constructor.

To support high-level reasoning about summations a large part of the mechanization is therefore dedicated to proving these lemmas about factoring terms out of summations and rewriting the function argument of the summation, which are all gathered in rewriting databases. This automates much of the work when proving derivability of normalization.

Once distributivity has been applied, we apply the tactic analogues of "matching the left-most terms" and "matching the right-most terms", which are the tactics `eq_m_left` and `eq_m_right`. The first is defined as:

```
Ltac eq_m_left :=
    repeat rewrite c_plus_assoc;
    apply c_plus_ctx;
    auto_rwd_eqDB.
```

It associates + expressions towards the right as much as possible which isolates the left most term on both sides, applies the context rule of + and finally applies `auto` and `autorewrite`. `eq_m_right` is defined similarly, associating to the left instead.

High level distributivity laws used as rewrites, followed by matching to the left and right can be seen in the short proof of derivability of the normaliztion for the case of ;. It has been factored out into its own independent lemma with the two assumptions as the IHs.

```
Lemma derive_unfold_seq : forall c1 c2,
o c1 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c1) == c1 ->
```

```
o c2 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c2) == c2 ->
o (c1 _;_ c2) _+_
Σ alphabet (fun a : EventType => Event a _;_ a \ (c1 _;_ c2)) ==
c1 _;_ c2.
Proof.
intros. rewrite <- H at 2. rewrite <- H0 at 2. (*IHs*)
autorewrite with funDB eqDB. (*Distributivity*)
eq_m_left. (*Match left*)
rewrite Σ_seq_assoc_right_fun. rewrite Σ_factor_seq_l_fun.
rewrite <- H0 at 1. (*IH*)
autorewrite with eqDB funDB. (*Distributivity*)
rewrite c_plus_assoc.
rewrite (c_plus_comm (Σ _ _ _;_  Σ _ _)).
eq_m_right. (*Match right*)
Qed.
```

The proof starts by applying the IHs of c1 and c2 on the right hand side, after which it distributes by rewriting with equivalence lemmas about functions from funDB and countracts from eqDB. This step applies all needed distributivity rewrites, except for Σ_seq_assoc_right_fun. This we must rewrite manually because it first can be applied after an associativity rewrite (grouped towards the right) for the left associative operator $\lambda;\lambda$. Associativity grouped towards the right the right is in general not desirable as a rewrite hint for a left-associative operator as it renders many of our lemmas (stated left-associatively) useless for automation.

7.4 Relating the two axiomatizations

In the paper proof, we argued that any contract equivalence involving contracts from $CSL_{||}$ after normalization would lie in the set CSL_0, Letting us appeal to completeness of the axiomatization for CSL_0. In the mechanization, the contract type representing CSL_0 (Core.Contract) is distinct from the contract type representing $CSL_{||}$ (Parallel.Contract). Likewise they each have their own satisfaction predicates Core.Matches_comp and Parallel.Matches_Comp. The relation between the contract types is seen by the first rule in Parallel.c_eq:

```
Inductive c_eq : Contract -> Contract -> Prop :=
| c_core p0 p1 c0 c1 (H0: translate_aux p0 = Some c0)
                     (H1:translate_aux p1 = Some c1)
                     (H2: CSLEQ.c_eq c0 c1) : p0 == p1
(...)
```

The rule states that an equality between the contracts p0 and p1 of type Parallel.Contract can be derived if they respectively translate to contracts c0 and c1 of type Core.Contract that are derivably equivalent in their own axiomatization (CSLEQ.c_eq). translate_aux is a projection function mapping a Parallel.Contract to a Core.Contract.

This must be a partial function because the parallel operator has no representation in `Core.Contract`. The partial function has been implemented with an option type, returning `Some c` when it is defined and `None` otherwise.

8 Formalizing CSL_*

We now introduce iteration, yielding CSL^*.

$$c := ...|\ c^*$$

The satisfaction relation is extended with two rules for iteration seen in Figure 12.

$$\frac{}{\langle\rangle : c^*}\text{ MStar0} \qquad \frac{s_0 : c \qquad s_1 : c*}{s_0 s_1 : c^*}\text{ MStarApp}$$

Figure 12: Satisfaction for iteration

We also extend the definitions of nu and $\cdot\backslash\cdot$.

$$nu(c^*) := 1$$
$$e\backslash c^* := (e\backslash c); c^*$$

Theorem 8.1 *For all contracts c and traces s, $s : c \iff nu(s\backslash\backslash c) = 1$*
Proof proceeds similarly to semantic equivalence proof for $CSL_{||}$ (skipped).

We will follow the methods of Brandt and Henglein [3] and Hur et al. [4] in defining our axiomatization. We could have based ourselves on other existing axiomatizations of regular expression equivalence and the discussion will mention alternatives and why we chose this approach.

8.1 Inductively and coinductively defined sets

Inductive sets We say that $\mathcal{F}$ is a monotonic operator on sets if $X \subseteq Y \implies \mathcal{F}(X) \subseteq \mathcal{F}(Y)$. Writing the least fixpoint of function f as μf, an inductively defined set, or simply *an inductive set* S is the least fixpoint of a monotonic operator under set containment, i.e. $S = \mathcal{F}(S)$ and S is the smallest set with this property. The Knaster-Tarski theorem states that $S = \mu\mathcal{F} = \bigcap\{X|\mathcal{F}(X) \subseteq X\}$. The induction principle states that to prove $S \subseteq P$ it is sufficient to prove $\mathcal{F}(P) \subseteq P$.
As an example, consider the underlying monotonic operator for the definition of lists.

$$\mathcal{F}_{list}(X) := \{[]\} \cup \{n :: l | n \in \mathbf{N}, l \in X\}$$

The set $\mu\mathcal{F}_{list}$ is an inductively defined set containing all the finitely long lists of natural numbers. Proving a statement that holds for all lists in $\mu\mathcal{F}_{list}$, means to define a set of lists P satisfying the statement and showing $\mathcal{F}(P) = \{[]\} \cup \{n :: l|n \in \mathbf{N}, l \in P\} \subseteq P$.

Coinductive sets Writing the greatest fixpoint of function f as νf, a coinductively defined set S is the largest fixpoint of a monotonic operator $\mathcal{F}$ under set containment, i.e. $S = \mathcal{F}(S)$ and S is the largest set with this property. The Knaster-Tarski theorem states that $S = \nu\mathcal{F} = \bigcup\{X|X \subseteq \mathcal{F}(X)\}$. The coinduction principle states that to prove $P \subseteq S$ it is sufficient to prove $P \subseteq \mathcal{F}(P)$. Therefore to prove some s is a member of the coinductive set S, we must show there exists an X, s.t. $s \in X$ and $X \subseteq \mathcal{F}(X)$. If X is a set of statements of the form $t_1 = t_2$, then X is called a bisimulation and the bisimilarity is the largest such bisimulation. Finally there also is a *strong coinduction principle* which states that for any X, $X \subseteq \nu\mathcal{F} \iff X \subseteq \mathcal{F}(X \cup \nu\mathcal{F})$ [6].

Inference systems An inference system (set of axioms and inference rules) defines a monotonic operator. Consider a smaller inference system for $CSL_{||}$ with the axioms $c + Failure == c$, reflexivity and a transitivity rule. Its monotonic operator is

$$\begin{aligned}\mathcal{F}(X) := \{(c + Failure, c)|c \in CSL_{||}\} \cup \{(c, c)|c \in CSL_{||}\}\\ \cup\{(c_0, c_2)|(c_0, c_1) \in X, (c_1, c_2) \in X\}\end{aligned}$$

A typical derivation in such a system is an inductive derivation. Such a derivation where one starts from c and repeatedly remove $Failure$, resulting in c', corresponds to showing $(c, c') \in \mu\mathcal{F}$. From a coinductive derivation, we can derive all pairs that can be inductively derived and more. A coinductive derivation of $Failure == Success$ means to show $(Failure, Success) \in \nu\mathcal{F}$ but it suffices to find a set X where $(Failure, Success) \in X$ and show $X \subseteq \mathcal{F}(X)$. To show this we could use the set $X := \{(Failure, Success), (Success, Success)\}$ and show

$$\begin{aligned}X \subseteq \{(c + Failure, c)|c \in CSL_{||}\} \cup \{(c, c)|c \in CSL_{||}\}\\ \cup\{(c_0, c_2)|(c_0, c_1) \in X, (c_1, c_2) \in X\}\end{aligned}$$

Clearly $(Success, Success)$ is contained in $\mathcal{F}(X)$. This is also the case for $(Failure, Success)$ because $(Failure, Success) \in X$ and $(Success, Succes) \in X$ implies $(Failure, Success) \in \mathcal{F}(X)$.

Brandt and Henglein's method Their method is used for giving a coinductive interpretation of an inductively defined set. This is done by extending an inference system (whose monotonic operator is $\mathcal{F}$) with a suitable fixpoint-rule (what exactly this rule is will depend on the context). This yields a new inference system (whose monotonic operator is $\mathcal{F}'$) that has a larger set of inductively derivable statements, that is $\mu\mathcal{F} \subseteq \mu\mathcal{F}'$. More specifically, the set of coinductively derivable statements in the smaller system is the same as the inductively derivable statements in the larger system, that is $\nu\mathcal{F} = \mu\mathcal{F}'$. This allows us the derive that a statement lies in $\nu\mathcal{F}$, by usual inductive derivations in the inference system of $\mathcal{F}'$.

Method of Hur et al. They show how one can define a coinductive set as the greatest fixpoint of a parameterized inductive definition in Coq. Reusing their example consider equality on the set of infinite lists. This set can be defined in terms of the inductive definition leq_R parameterized over the set of list pairs R. The definition has a single rule: $\dfrac{(l_0, l_1) \in R}{(n :: l_0, n :: l_1) \in leq\_R}$ Now defining $\mathcal{F}_{leq}(R) := leq\_R$, the set of all equal infinite lists can be defined as $\nu\mathcal{F}_{leq}$.

8.2 Adding a fix-rule

Recall that completeness of the axiomatization for $CSL_{||}$ was demonstrated by showing that all contracts c had a derivable normal form $o(c) + \Sigma_{e \in E} e \backslash c$ and the repeated application of this fact on the residual contracts, eliminated the parallel operator. It is not possible to similarly define a procedure that eliminates iteration, but we can get around this by introducing a fix-rule that allows us to use the equality we are showing in its proof. Inspired by Grabmeyer's Comp-fix rule [2] and following the Brandt and Henglein method, consider the fix-rule called Sum-fix-ctx.

$$\frac{\Gamma, \Sigma_{i=1}^{n} e_i; c_i == \Sigma_{i=1}^{n} e_i; d_i \vdash \forall i.\ c_i == d_i}{\Gamma \vdash \Sigma_{i=1}^{n} e_i; c_i == \Sigma_{i=1}^{n} e_i; d_i} \text{ Sum-fix-ctx}$$

An equality which has been inductively derived only using Sum-fix-ctx, corresponds to a coinductive derivation using only Sum-fix' defined as:

$$\frac{\forall i.\ c_i == d_i}{\Sigma_{i=1}^{n} e_i; c_i == \Sigma_{i=1}^{n} e_i; d_i} \text{ Sum-fix'}$$

We saw that allowing a coinductive use of transitivity let us prove $Failure == Success$, which is unsound. It is therefore important to restrict which rules that may be used coinductively. One way to define our axiomatization would be to stay in line with the Brandt and Henglein method and simply introduce Sum-fix-ctx as a rule in our system and restrict all rules to be used only inductively (as a traditional inference system). This is hard to mechanize in Coq (mentioned in discussion), so we also take inspiration from the method from Hur et al. [4]. By defining our inference system as a parameterized inductive definition parameterized over some R, we can express the fix-point rule Sum-fix that will be used in our axiomatization.

$$\frac{\forall i.\ (c_i, d_i) \in R}{\Sigma_{i=1}^{n} e_i; c_i == \Sigma_{i=1}^{n} e_i; d_i} \text{ Sum-fix}$$

With the right choice of R this will correspond to an inductive use of Sum-fix-ctx (or coinductive use of Sum-fix') and we will later see how this can be mechanized. In Figure 13 we define the inductive relation $==_R$ parameterized over some relation R. Letting $\mathcal{F}_{eq}(S)$ be the instantiation $(==_S)$, our axiomatization for CSL_* is then the instantiation $==_{\nu\mathcal{F}_{eq}}$.

$$(c_0 + c_1) + c_2 ==_R c_0 + (c_1 + c_2) \tag{18}$$
$$c_0 + c_1 ==_R c_1 + c_0 \tag{19}$$
$$c + Failure ==_R c \tag{20}$$
$$c + c ==_R c \tag{21}$$
$$(c_0; c_1); c_2 ==_R c_0; (c_1; c_2) \tag{22}$$
$$(Success; c) ==_R c \tag{23}$$
$$c; Success ==_R c \tag{24}$$
$$Failure; c ==_R Failure \tag{25}$$
$$c; Failure ==_R Failure \tag{26}$$
$$c_0; (c_1 + c_2) ==_R (c_0; c_1) + (c_0; c_2) \tag{27}$$
$$(c_0 + c_1); c_2 ==_R (c_0; c_2) + (c_1; c_2) \tag{28}$$
$$c ==_R c \tag{29}$$
$$c_0 * c_1 * c_2 ==_R c_0 * (c_1 * c_2) \tag{30}$$
$$c * Success ==_R c \tag{31}$$
$$c * Failure ==_R Failure \tag{32}$$
$$c_0 * (c_1 + c_2) ==_R c_0 * c_1 + c_0 * c_2 \tag{33}$$
$$e_0; c_0 * e_1; c_1 ==_R e_0; (c_0 * e_1; c_1) + e_1; (e_0; c_0 * c_1) \tag{34}$$
$$Success + c; c^* ==_R c^* \tag{35}$$
$$(c + Success)* ==_R c^* \tag{36}$$

$$\frac{c_0 ==_R c_1}{c_1 ==_R c_0}\ \text{(Sym)} \qquad \frac{c_0 ==_R c_1 \qquad c_1 ==_R c_2}{c_0 ==_R c_2}\ \text{(Trans)}$$

$$\frac{c_0 ==_R c_0' \qquad c_1 ==_R c_1'}{c_0 + c_1 ==_R c_0' + c_1'}\ \text{(Ctx-plus)}$$

$$\frac{c_0 ==_R c_0' \qquad c_1 ==_R c_1'}{c_0; c_1 ==_R c_0'; c_1'}\ \text{(Ctx-seq)} \quad \frac{c_0 ==_R c_0' \qquad c_1 ==_R c_1'}{c_0 * c_1 ==_R c_0' * c_1'}\ \text{(Ctx-par)}$$

$$\frac{c ==_R c'}{c^* ==_R c'^*}\ \text{(Star-ctx)}$$

$$\frac{\forall i.\ c_i R d_i}{\Sigma_{i=1}^n e_i; c_i ==_R \Sigma_{i=1}^n e_i; d_i}\ \text{Sum-fix}$$

Figure 13: Axiomatization of contract equivalence. 12 axioms and 4 inference rules

8.3 Satisfaction and the Bisimilarity

Our definition of $==_{\nu\mathcal{F}_{eq}}$ will allow us to reason coinductively about derivable equivalence. We will need to do the same for satisfaction equivalence and therefore introduce a notion of semantic equivalence that is inspired by Grabmeyer: c_0 and c_1 are bisimilar, written as $c_0 \sim c_1$, iff there exists a bisimulation R s.t. $(c_0, c_1) \in R$. The bisimulations R we will consider are those that satisfy $R \subseteq \mathcal{F}_{bi}(R)$, where $\mathcal{F}_{bi}(R)$ is a inductive definition parameterized over R, with the single rule.

$$\frac{\forall e.(e\backslash c, e\backslash c') \in R \qquad nu\ c = nu\ c'}{(c, c') \in \mathcal{F}_{bi}(R)}$$

The largest relation R satisfying $R \subseteq \mathcal{F}_{bi}(R)$ is the bisimilarity which is the same set as $\nu\mathcal{F}_{bi}$, which contains all bisimulations. Referring to the set of satisfiably equivalent contract pairs simply as sat, we now show that sat and $\nu\mathcal{F}_{bi}$ are the same set.

Theorem 8.2 $sat = \nu\mathcal{F}_{bi}$.

Proof. We first show $sat \subseteq \nu\mathcal{F}_{bi}$ then $\nu\mathcal{F}_{bi} \subseteq sat$

- Case $sat \subseteq \nu\mathcal{F}_{bi}$
  It suffices to show $sat \subseteq \mathcal{F}_{bi}(sat)$, that is we must show for all c_0 and c_1, if $\forall s.\ s : c_0 \iff s : c_1$ then also $\forall es.\ s : (e\backslash c_0) \iff s : (e\backslash c_1)$ and $nu\ c_0 = nu\ c_1$. We have $nu\ c_0 = nu\ c_1$ because c_0 and c_1 are equally satisfiable on all traces, in particular the empty trace.
  $\forall es.s : e\backslash c_0 \iff s : e\backslash c_1$ holds because we after undoing residuation and fixing e and s have

  $$es : c \iff es : d$$

  Which again holds by satisfiable equivalence of c_0 and c_1.

- Case $\nu\mathcal{F}_{bi} \subseteq sat$
  For all $(c_0, c_1) \in \nu\mathcal{F}_{bi}$, we must show $\forall s.\ s : c_0 \iff s : c_1$. We show this by induction on s.

  - Case $s = \langle\rangle$.
    We must show $\langle\rangle : c_0 \iff \langle\rangle : c_1$. Being a fixpoint, we have $\nu\mathcal{F}_{bi} = \mathcal{F}_{bi}(\nu\mathcal{F}_{bi})$. By inversion on $(c_0, c_1) \in \mathcal{F}_{bi}(\nu\mathcal{F}_{bi})$ we have $nu\ c_0 = nu\ c_1$.
  - Case $s = es'$.
    We must show $es' : c_0 \iff es' : c_1$. After residuation on both sides yielding $s' : e\backslash c_0 \iff s' : e\backslash c_1$. By IH it suffices to show $(e\backslash c_0, e\backslash c_1) \in \nu\mathcal{F}_{bi}$. Again this holds by inversion on $(e\backslash c_0, e\backslash c_1) \in \mathcal{F}_{bi}(\nu\mathcal{F}_{bi})$.

8.4 Soundness

For $==_{\nu\mathcal{F}_{eq}}$ to be sound, derivable equivalence must imply satisfiable equivalence. Knowing that the set of satisfiably equivalent contract pairs is equal to the bisimilarity, it suffices to show that set of derivable equivalences are contained in the bisimilarity, shown by Lemma 8.3 and 8.4 below, which lets us conclude soundness in Theorem 8.6.

Lemma 8.3 *For all c_0 and c_1 and e, $c_0 ==_{\nu\mathcal{F}_{eq}} c_1 \implies e\backslash c_0 ==_{\nu\mathcal{F}_{eq}} e\backslash c_1$*

Proof by induction on $c_0 ==_{\nu\mathcal{F}_{eq}} c_1$ (only showing the previously troublesome 38 and Sum-fix).

- Case rule (38):
  We must show:

  $$e\backslash(c + Success)^* ==_{\nu\mathcal{F}_{eq}} e\backslash c^*$$

  Which is equivalent to

  $$(e\backslash c + Failure); (c + Success)^* ==_{\nu\mathcal{F}_{eq}} e\backslash c; c^*$$

  This holds by neutrality of Failure and rewriting with (38) on the left-hand side.

- Case sum-fix:
  We may assume $\forall i.\ (c_i, d_i) \in \nu\mathcal{F}_{eq}$ and must show:

  $$e\backslash\Sigma_{i=1}^{n} e_i; c_i ==_{\nu\mathcal{F}_{eq}} e\backslash\Sigma_{i=1}^{n} e_i; d_i$$

  Let the sequence of natural numbers $n_0, ..., n_k$ where $0 \leq k$, be such that $e_{n_0}, ...e_{n_k}$ is the sub-sequence of $e_0, ..., e_n$ only containing the events equal to e. If $k = 0$, both summations solely contains sequences beginning with Failure, which can be rewritten to $Failure ==_{\nu\mathcal{F}_{eq}} Failure$. If $k > 0$, by neutrality of Success, it suffices to show

  $$\Sigma_{i=1}^{k} c_{n_i} ==_{\nu\mathcal{F}_{eq}} \Sigma_{i=1}^{k} d_{n_i}$$

  By context rule of $+$ we can just show that for all $i \leq k$

  $$c_{n_i} ==_{\nu\mathcal{F}_{eq}} d_{n_i}$$

  Which is equivalent to saying for all $i \leq k$

  $$(c_{n_i}, d_{n_i}) \in \nu\mathcal{F}_{eq}$$

  Which we have by assumption.

Lemma 8.4 *For all c_0 and c_1 $c_0 ==_{\nu\mathcal{F}_{eq}} c_1$ implies $nu\ c_0 = nu\ c_1$*
Proof by induction on $c_0 ==_{\nu\mathcal{F}_{eq}} c_1$ (skipped).

Lemma 8.5 *For all contract c_0, c_1, if $c_0 ==_{\nu\mathcal{F}_{eq}} c_1$ then c_0 and c_1 are bisimilar.*

Proof.
Writing the bisimilarity as $bisim$, we must show $(==_{\nu\mathcal{F}_{eq}}) \subseteq bisim$.
By coinduction it suffices to show $(==_{\nu\mathcal{F}_{eq}}) \subseteq \mathcal{F}_{bi}(==_{\nu\mathcal{F}_{eq}})$, that is, it we must show for all c_0 and c_1 if $c_0 ==_{\nu\mathcal{F}_{eq}} c_1$ then for all e, $e\backslash c_0 ==_{\nu\mathcal{F}_{eq}} e\backslash c_1$ and $nu\ c_0 = nu\ c_1$, which is shown by Lemma 8.3 and Lemma 8.5

Theorem 8.6 (Soundness) *For all contracts c_0 and c_1, $c_0 ==_{\nu\mathcal{F}_{eq}} c_1 \implies \forall s.s : c_0 \iff s : c_1$*

Immediate from Lemma 8.2 and 8.5.

8.5 Completeness

To show $==_{\nu\mathcal{F}_{eq}}$ is complete, we show that the normal form is derivable in the system (Lemma 8.7) and use it in showing that all bisimilar c_0 and c_1 are derivable (Lemma 8.8). This lets us conclude completeness (Theorem 8.9).

Lemma 8.7 (Derivability of normal form) *For all contract relations R, c, $c ==_{\nu\mathcal{F}_{eq}} o(c) + \Sigma_{e\in E} e; e\backslash c$*

Proof is by induction on c, where all other cases than iteration, are identical to what was shown in $CSL_{||}$. The only new case is iteration where we must show

$$c^* ==_{\nu\mathcal{F}_{eq}} Success + \Sigma_{e\in E} e; (e\backslash c; c*)$$

We apply IH on on the left

$$(o(c) + \Sigma_{e\in E} e; e\backslash c)^* ==_{\nu\mathcal{F}_{eq}} Success + \Sigma_{e\in E} e; e\backslash c; c*$$

By associativity and distributivity, we move c^* out of the sum on the right hand side

$$(o(c) + \Sigma_{e\in E} e; e\backslash c)^* ==_{\nu\mathcal{F}_{eq}} Success + \big(\Sigma_{e\in E} e; e\backslash c\big); c*$$

We now apply IH on the right

$$(o(c) + \Sigma_{e\in E} e; e\backslash c)^* ==_{\nu\mathcal{F}_{eq}} Success + \big(\Sigma_{e\in E} e; e\backslash c\big); (o(c) + \Sigma_{e\in E} e; e\backslash c)^*$$

We proceed by case distinction on $o\ c$.

- Case $o\ c = Success$.
  We must show:

  $$(Success + \Sigma_{e\in E} e; e\backslash c)^* ==_{\nu\mathcal{F}_{eq}} Success + \big(\Sigma_{e\in E} e; e\backslash c\big); (Success + \Sigma_{e\in E} e; e\backslash c)^*$$

  After applying (36) eliminating Success under iteration, the remaining equation is just an instantiation of the unfold rule (35), finishing this case.

- Case $o\ c = Failure$.
  We must show:

  $$(Failure + \Sigma_{e \in E} e; e\backslash c)^* ==_{\nu \mathcal{F}_{eq}} Success + \big(\Sigma_{e \in E} e; e\backslash c\big); (Failure + \Sigma_{e \in E} e; e\backslash c)^*$$

  By neutrality of Failure, this equation is also just equivalent to an instantiation of (35), finishing the proof.

Lemma 8.8 *For all contracts $c_0\ c_1$, $c_0 \sim c_1$ implies $c_0 ==_{\nu \mathcal{F}_{eq}} c_1$*

We must show $bisim \subseteq \nu\ \mathcal{F}_{eq}$. By coinduction it suffices to show: $bisim \subseteq \mathcal{F}_{eq}(bisim)$, that is, we must show for all c_0 and c_1, if $c_0 \sim c_1$ then

$$c_0 ==_{\nu \mathcal{F}_{bi}} c_1$$

We apply Lemma 8.7 on both sides yielding

$$o(c_0) + \Sigma_{e \in E} e; e\backslash c_0 ==_{\nu \mathcal{F}_{bi}} o(c_1) + \Sigma_{e \in E} e; e\backslash c_1$$

By assumption we have $nu\ c_0 = nu\ c_1$ and $\forall e.\ e\backslash c_0 (\nu\ \mathcal{F}_{bi}) e\backslash c_1$. The left most $o(\cdot)$ terms cancel out and we must show

$$\Sigma_{e \in E} e; e\backslash c_0 ==_{\nu \mathcal{F}_{bi}} \Sigma_{e \in E} e; e\backslash c_1$$

We now apply Sum-fix and must show its premise

$$\forall e. e\backslash c_0 (\nu\ \mathcal{F}_{bi}) e\backslash c_1$$

Which we have by assumption.

Theorem 8.9 (Completeness) *For all contracts c_0 and c_1, $\forall s.\ s : c_0 \iff s : c_1$ implies $c_0 ==_{\nu \mathcal{F}_{eq}} c_1$*

Immediate from Lemma 8.2 and 8.8.

9 Mechanizing CSL_*

9.1 The paco library

The coinductive sets $\nu \mathcal{F}_b i$ and $\nu \mathcal{F}_e q$ will be represented in Coq using the library paco, developed by Hur et al. [4]. Hur et al. showed how to reason about coinductive sets by *parameterized coinduction*, a compositional and incremental proof principle that simplifies coinductive proofs. We have seen that for coinductive set S with monotonic operator $\mathcal{F}$, to prove $x \in S$, one can show this by finding some X, where $x \in X$ and show $X \subseteq \mathcal{F}(X)$. For more complicated proofs, determining X upfront can be hard and paramterized coinduction is a principle that allows X to be constructed gradually. For our coinductive proofs determining X won't be hard as it is always either sat or the bisimilarity. What would have been hard, had we not used paco, would be constructing valid proof terms with automation because Coq's native support for coinduction (via keyword `CoInductive`) interacts poorly with tactics such as `auto`. Moreover using `CoInductive`, would not have allowed the flexibility of interpreting some rules coinductively (Sum-fix) and others inductively (all other rules). In the discussion alternative representations will be mentioned.

9.2 Representing bisimilarity

We represent $\mathcal{F}_{bi}$ by the inductive definition `bisimilarity_gen` which is parameterized over a relation `bisim:Contract -> Contract -> Prop`.

```
Inductive bisimilarity_gen bisim : Contract -> Contract -> Prop :=
bisimilarity_con c0 c1 (H0: forall e, bisim (e \ c0) (e \ c1) : Prop )
                        (H1: nu c0 = nu c1) :
                       bisimilarity_gen bisim c0 c1.
```

The bisimilarity relation itself is the greatest fixpoint of the operator, written as:

```
Definition Bisimilarity c0 c1 := paco2 bisimilarity_gen bot2 c0 c1.
```

paco2 is a function used to define paramterized coinductive predicates of arity 2. The paramterized greatest fixpoint of f is $G_f(X) := \nu(\lambda Y.\ f(Y \cup X))$. That is, the fixpoint is paramterized over X, allowing it to be constructed gradually during a proof. Note that $G_f(\{\}) = \nu(\lambda Y.\ f(Y)) = \nu f$ and with `bot2` representing the empty relation, `Bisimilarity` therefore represents $\nu\mathcal{F}_{bi}$.

9.3 Representing $==_{\nu\mathcal{F}_{eq}}$

The representation of $\mathcal{F}_{eq}$ (only showing a few of the constructurs) is:

```
Section axiomatization.
  Variable co_eq : Contract -> Contract -> Prop.

Inductive c_eq : Contract -> Contract -> Prop :=
| c_plus_assoc c0 c1 c2 : (c0 _+_ c1) _+_ c2 == c0 _+_ (c1 _+_ c2)
| c_plus_comm c0 c1: c0 _+_ c1 == c1 _+_ c0
| c_plus_neut c: c _+_ Failure == c
(...)
| c_co_sum es ps  (H: forall p, In p ps -> co_eq (fst p) (snd p) : Prop)
                  : (Σe es (map fst ps)) == (Σe es (map snd ps))
 where "c1 == c2" := (c_eq c1 c2).
End axiomatization.
```

Coq's Section mechanism (seen on the first line), allows us to paramterize the definition of `c_eq` by the variable `co_eq` implicitly, saving us from explicitly refering to co_eq in all constructors. co_eq is only used in c_co_sum (Sum-fix). The notation $c_1 == c_2$ is also local to the section and therefore solely used to make the definition more readable inside the Section. Outside the section the type of c_eq is:

```
c_eq : (Contract -> Contract -> Prop) -> Contract -> Contract -> Prop
```

It can be seen by the type that c_eq is paramterized over a relation. We introduce the notation c0 =(R)= c1 for c_eq R c0 c1 and as Coq supports rewriting parameterized equivalence relations most equivalence proofs from the mechanization of $CSL_{||}$ also work for our new equivalence. We copy those and define hint databases as usual. As an example neutrality of Failure (from the left) is stated as:

```
Lemma c_plus_neut_l : forall R c, Failure _+_ c =(R)= c.
```

Recall that Sum-fix is defined as:

$$\frac{\forall i.\ c_i R d_i}{\Sigma_{i=1}^{n} e_i; c_i ==_{\nu\mathcal{F}_{eq}} \Sigma_{i=1}^{n} e_i; d_i}\ \text{Sum-fix}$$

To represent this rule, we define the function Σe : list Event -> list Contract -> Contract, representing $\Sigma_{i=1}^{n} e_i; c_i$ by zipping its input lists with _;_.

```
Definition Σe es cs := Σ (combine es cs)
                         (fun x => Event (fst x) _;_ snd x).
```

We now represent the sum rule as:

```
Section axiomatization.
  Variable co_eq : Contract -> Contract -> Prop.

Inductive c_eq : Contract -> Contract -> Prop :=
(...)
| c_co_sum es ps  (H: forall p, In p ps -> co_eq (fst p) (snd p) : Prop)
                  : (Σe es (map fst ps)) == (Σe es (map snd ps))
 where "c1 == c2" := (c_eq c1 c2).
End axiomatization.
```

In the conclusion, the second argument of Σe (on both sides of the equality) is a projection of the list ps : list (Contract * Contract), projecting either the left components or the right, by mapping with fst/snd. The convenience of this definition is that it is implicit that map fst ps and map snd ps have the same length, saving us from the trouble of asserting this with an additional assumption. This simplifies some induction proofs.

We represent $\nu\mathcal{F}_{eq}$ as the greatest fixpoint of c_eq.

```
Definition co_eq c0 c1 := paco2 c_eq bot2 c0 c1.
Notation "c0 =C= c1" := (co_eq c0 c1)(at level 63).
```

Finally $==_{\nu\mathcal{F}_{eq}}$ is then represented by =(co_eq)=.

9.4 Soundness

The soundness proof wrt. to the bisimilarity is:

```
Lemma bisim_soundness : forall (c0 c1 : Contract),
c0 =C= c1 -> Bisimilarity c0 c1.
Proof.
pcofix CIH.
intros. pfold. constructor.
- intros. right. apply CIH.  apply co_eq_derive. auto.
- auto using co_eq_nu.
Qed.
```

Here pcofix is a use of the paramterized coinduction principle. We may assume for some r that $\nu\mathcal{F}_{eq} \subseteq r$. We must however now show the lemma, not for the normal greatest fixpoint $\nu\mathcal{F}_{bi} = G_{\mathcal{F}_{bi}}(\{\})$, but for the paramterized greatest fixpoint $G_{\mathcal{F}_{bi}}(r)$. Specifically we must show $\nu\mathcal{F}_{eq} \subseteq G_{\mathcal{F}_{bi}}(r)$. `pfold` is a paco tactic mechanizing the paramaterized variant of the strong coinduction principle, unfolding the goal `(c0,c1)` $\in G_{\mathcal{F}_{bi}}(r)$ to `(c0,c1)` $\in \mathcal{F}_{bi}(G_{\mathcal{F}_{bi}} \cup r)$ from where we apply the $\mathcal{F}_{bi}$ constructor (`bisimilarity_con`) and prove its two premises using `co_eq_derive` (Lemma 8.3) and `co_eq_nu` (Lemma 8.4).

9.5 Completeness

Several parts of the mechanized completneeess proof are useful to factor out into separate tactics. These can be reused later when we want to illustrate the mechanized axiomatization with example derivations. In this section we present three helper tactics and then show the mechanized completeness proof.

Helper tactics

In the paper-proof of completenesss wrt. to the bisimilarity, from $c_0 \sim c_1$ we had to show

$$c_0 ==_{\nu\mathcal{F}_{bi}} c_1$$

To do this we normalized both c_0 and c_1 in $c_0 ==_{\nu\mathcal{F}_{bi}} c_1$. This normalization step is carried out by the tactic `unfold_tac`.

```
Ltac unfold_tac :=
 match goal with
          | [ |-  ?c0 = (_) = ?c1  ] =>
          rewrite <- (derive_unfold _ c0) at 1;
          rewrite <- (derive_unfold _ c1) at 1;
          unfold o; eq_m_left; try solve [apply if_nu; simpl; btauto]
          end.
```

After normalizing c0 and c1, we try in line 6 to match up the left-most $o(\cdot)$ terms. If matching up these terms was successful we are left with a goal of the following shape (leaving the parameterized relation unspecificed by underscore):

```
Σ alphabet (fun a : EventType => Event a _;_ a \ c0) = (_)=
Σ alphabet (fun a : EventType => Event a _;_ a \ c1)
```

This goal must be written into a form that matches the conclusion of c_co_sum. This is handled by the tactic sum_reshape.

```
Ltac sum_reshape := repeat rewrite Σd_to_Σe;
                    apply Σe_to_pair;
                    repeat rewrite map_length; auto.
```

After applying sum_reshape, the goal becomes a quite large expression. To make this more readable we define ps to be a zip of the residual lists of c0 and c1.

```
ps = combine (map (fun e : EventType => e \ c0) alphabet)
             (map (fun e : EventType => e \ c1) alphabet)
```

Then sum_reshape rewrites the goal into the following shape:

```
Σ alphabet (map fst ps) = (_)= Σ alphabet (map snd ps)
```

This shape allows us to apply c_co_sum, exchanging the goal with the premise of c_co_sum, which has the following shape (where co_eq is some unspecified relation):

```
forall p : Contract * Contract, In p ps -> co\_eq (fst p) (snd p)
```

Recall that ps is a zip of the residual lists of c0 and c1. The i'th element of ps are residuals of c0 and c1 wrt. to the same event. Therefore if it is the case that In p ps, then there must exist an event e, such that fst p = e\c0 and snd p = e\c1. The tactic simp_premise mechanizes this argument.

```
Ltac simp_premise :=
 match goal with
 | [ H: In ?p (combine (map _ _) (map _ _)) |- _ ] =>
     destruct p; rewrite combine_map in H;
     rewrite in_map_iff in *;
     destruct_ctx;simpl;inversion H;subst;clear H
 end.
```

The completeness proof

The completeness proof wrt. to the bisimilarity is:

```
1  Lemma bisim_completeness : forall c0 c1,
2  Bisimilarity c0 c1 -> c0 =C= c1.
3  Proof.
4  pcofix CIH.
5  intros. punfold H0. inversion H0.
6  pfold.
7  unfold_tac.
8  - rewrite H2. reflexivity.
9  - sum_reshape.
10   apply c_co_sum. intros.
11   simp_premise.
12   right. apply CIH.
13   pclearbot.
14   unfold Bisimilarity. auto.
15 Qed.
```

At line 4 the parameterized coinduction principle is applied, allowing us to assume for some r that $bisim \subseteq r$ and must show $bisim \subseteq G_{\mathcal{F}_{eq}}(r)$. The `punfold` tactic on line 5 applied to `H0` is the analogue of `pfold` that unfolds the assumption `H0` from `(c0,c1)` $\in G_{\mathcal{F}_{bi}}(r)$ to `(c0,c1)` $\in \mathcal{F}_{bi}(G_{\mathcal{F}_{bi}} \cup r)$. After applying inversion on `H0` in line 5, the goal `(c0,c1)` $\in G_{\mathcal{F}_{eq}}(r)$ is unfolded to `(c0,c1)` $\in \mathcal{F}_{eq}(G_{\mathcal{F}_{eq}}(r) \cup r)$ in line 6, which corresponds to showing `c0` $==_{G_r \cup \mathcal{F}_{eq}}$ `c1`. We now use our helper tactics. `unfold_tac` performs the normalization. Line 8 show `o(c0) = o(c1)`. Line 9-14 rewrites the summations (`sum_reshape`), applies `c_co_sum` and simplifies the `In p ps` assumption (`simp_premise`). After this the goal is `(e\c0,e\ c1)` $\in G_{\mathcal{F}_{eq}}(r) \cup r$ for some event `e`. In line l12, `right` reduces the goal to `(e\c0,e\ c1)` $\in r$, from which we use the assumption `CIH` that shows $bisim \subseteq r$, to reduce the goal to `(e\c0,e\ c1)` $\in bisim$. From the additional hypotheses in the context that `inversion H0` produced in line 5, the goal becomes immediate.

10 Discussion

The design of the full-size CSL language is inspired by Jones et al. [7], who introduced the idea of specifyinig contracts compositonally. Their language contains analogoues to most CSL constructs except for $Failure$ and recursion (no notion of iteration). The language does unlike CSL support predicates that can be used on arbitrary contracts, not just the transmit construct. They also have an until operator, allowing one to successfully terminate a contract at some particular time. Hvitved et al. [8] introduced a compositional contract specification language, where they focused on blame assignment. They map contract specifications denotationally to functions from traces to verdicts, where a verdict either indicates that the contract is satisfied or if it is not, in which case the verdict contains the id of the agent who violated the contract and the time stamp at which it occurred.

10.1 Other formalizations

CSL_* is equivalent to the regular expressions extended with a parallel operator and the equivalence of this extension on regular expressions has a well-studied axiomatization known as the Concurrent Kleene Algebra. Concurrent Kleene Algebra, introduced by Hoare et al. [9] is an algebra that satisfies a set of axioms, used to study the behaviour of concurrent programs (of which parallel regular expressions is just one example). It extends the Kleene Algebra with a parallel operator. Kappé et al. [10] recently showed that the Concurrent Kleene Algebra is complete for an abstract model of concurrent programs. A complete axiomatization of CSL_* could therefore also have been given by showing that CSL_* is a Concurrent Kleene Algebra, with the significant drawback that the completeness proof of CKA is challenging, in part because it introduces many intermediate constructions. Alternatively, Salomaa (1966) [11] gave two sound and complete axiomatizations of regular expression equivalence (without parallel operator). We could have extended and mechanized either of these systems. A drawback with these axiomatizations is however that they each contain rules with nullability as a side condition, which makes substitution unsound. The coinductive/translation approach we have taken is simple in comparison CKA and unlike Salomaa's systems does not have any side conditions in any rules, making substitution a sound transformation.

On other coinductive axiomatizations, Grabmeyer gave a coinductive axiomatization of regular expression equivalence [2], which he also used to construct an efficient decision procedure based on building a finite bisimulation. Our Sum-fix rule is inspired by his rule Comp-fix. The largest difference between these rules is that Comp-fix is a semantic rule referring explicitly to nullaryness and residuals. Brandt and Henglein [3] gave a coinductive axiomatization for equality and subtyping of recursive types. Unlike the axiomatization of CSL_* that is defined as the greatest fixpoint of the inductively defined operator $\mathcal{F}_{eq}$ their axiomatization is inductively defined with contexts, allowing to reason coinductively by extending a premise's context with the conclusion. They use a fix-rule for the equality on function types.

$$\frac{A, \tau \to \tau' = \sigma \to \sigma' \vdash \tau = \sigma \qquad A, \tau \to \tau' = \sigma \to \sigma' \vdash \tau' = \sigma'}{A \vdash \tau \to \tau' = \sigma \to \sigma'}$$

It seems highly likely that the axiomatization of CSL_* could have been given completely in the style of Brandt and Henglein but this definition is harder to mechanize. The intuitive way of representing the premise of Sum-fix as a function expecting a proof of $\Sigma_{i=1}^{n} e_i; c_i == \Sigma_{i=1}^{n} e_i; d_i$ and returning a proof of $\forall i.\ c_i == d_i$ would introduce non-termination and is therefore not admissible in Coq. Danielsson et al. [12] demonstrates techniques for mixing inductive and coinductive definitions and demonstrates this in the proof-assistant Agda. As an example, they mechanize the Brandt and Henglein's coinductive axiomatization following their style. One line of future work is to apply their techniques to mechanize a simpler axiomatization of CSL_*.

It is known that CKA is decidable, that is one can construct a decision procedure for determining whether a statement is a theorem of CKA. It should equivalentely be possible to implement a decision procedure for CSL_*, based on normalization and applying Sum-fix. For it to be a terminating procedure, a set of previously visited equations would have to be maintained while solving the premise of Sum-fix. Almeida et al. [13] defined a decision procedure in this manner and it would be interesting future work to repurpose their procedure for CSL_* and show it sound and complete.

10.2 Other mechanizations and thoughts on new mechanizations in the future

To the best of my knowledge, the only other two mechanizations of sound and complete axiomatizations of Regular Expression Equivalence is Foster et al. [14] who mechanize four axiomatizations of regular expressions in Isabelle. Secondly its Pous [15], who mechanized Kleene Algebra with Tests (KAT) in Coq. KAT is an extension of KA with boolean tests, useful to model imperative programs. Pous's mechanization not only mechanizes KAT, but is actually a library mechanizing several related algebras (including KA). I believe a deeper study of this library would be beneficial in a future mechanization of the second generation CSL2 (that still is under development) that contains predicates. This I think for two reasons.

- Firstly, the library is designed with support for reflexive tactics in mind. Reflexive tactics embed propositional statements in a syntax that can be manipulated by computation, a powerful technique for proof automation. Pous gave a complete reflexive tactic, automating the proof of any theorem of KAT. Being able to achieve such a goal for CSL2 with predicates would be a much stronger property than a decision procedure, but would most likely require the alphabet to remain finite. For example, a payment event, Pay n, would need n to be restricted by some upper bound. Secondly the predicate language must of course be decidable itself.

- The second reason Pous library is interesting for our purposes is its abstract notion of equivalence. The many algebras share the same level-parameterized equivalence operator, allowing properties of lower level algebras (monoids) to propagate to higher level algebras (kleene algebra). This might be beneficial both for contracts and traces. For contracts, equivalences for CSL_0 might be resuable for $CSL_{||}$ and CSL_* so that we don't have to copy-paste proofs. To see the possible benefit for traces, consider how a formalization of CSL_* with predicates would look like. The constructor e, would be replaced with $P(x_0, x_1..., x_n)$. The satisfaction relation could be extended as $\delta \vdash s : (c, \delta')$, meaning a contract with environment δ is satisfied by s and transforms the state into δ'. Properties of traces and environments could then be proved more abstractly by considering them as distinct instances of monoids, where the corresponding append operation of environments would be map extension $\oplus$. This would include theorems about interleavings of monoids.

One additional insight that will be useful for a future development is to define summation with techniques from Bertot et al. [16]. They show how to represent summations in Coq elegantly, which if we had used, significantly could have reduced code complexity.

10.3 Experience with proving in Coq

It has been an interesting experience to mechanize proofs in Coq. To present a coherent story for the thesis, the mechanization is presented as the consequence of a prior formalization. In reality it was the other way around. I noticed that working too long on paper before mechanizing, often meant that I had missed an essential detail or more, either making the proof unsound or inconvenient to mechanize. On the other hand, working too little on paper one easily loses perspective and starts digging a deep hole of hacked lemmas leading nowhere. To avoid this I used the utility `Admit` heavily, instructing Coq to accept an unfinished proof. While constructing a challenging proof (such as the final completeness theorem) I would admit lemmas that seemed reasonable, finish the challenging proof and then go back and try to show the admitted proofs.

I also noticed that I could sit longer working on a Coq proof, than on a paper proof. I think one of the reasons for this is not having to worry about proof soundness. If Coq accepts the proof it is sound and this allows the coder to focus on constructing the proof instead of evaluating it. Evaluation then comes later as part of simplifying the proof.

11 Conclusion

We have seen that contracts can be represented in the specification language CSL and we showed mechanized formalization of a propositional calculus for the specifications in CSL_*. We defined compositional satisfaction semantics and operational monitoring semantics that we showed to be equivalent. We then defined the calculus and showed that derivations within it respected the semantics of contracts (soundnesss) and that all semantically equivalent contracts were derivable (complenetess). We gave the formalization and mechanization incrementally, starting with the further restricted variants CSL_0 and $CSL_{||}$. Soundness for both variants was straightforward to show. To show completeness of CSL_0 we took advantage of the language's finiteness, rewriting contracts to embeddings of their underlying trace sets. This result was extended to $CSL_{||}$ by eliminating the parallel operator by normalization. Finally, with the addition of iteration in CSL_* we made the axiomatization coinductive, which with the the rule Sum-fix allowed us to reuse the translation technique to show completeness.

References

[1] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen, "Compositional specification of commercial contracts," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 6, pp. 485–516, 2006.

[2] C. Grabmayer, "Using proofs by coinduction to find "traditional" proofs," in *Algebra and Coalgebra in Computer Science* (J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. Rutten, eds.), (Berlin, Heidelberg), pp. 175–193, Springer Berlin Heidelberg, 2005.

[3] M. Brandt and F. Henglein, "Coinductive axiomatization of recursive type equality and subtyping," *Fundamentae Informaticae*, no. Vol. 33, pp. 309–338, 1998.

[4] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis, "The power of parameterization in coinductive proof," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, (New York, NY, USA), p. 193–206, Association for Computing Machinery, 2013.

[5] C. Paulin-Mohring, "Introduction to the calculus of inductive constructions," 11 2014.

[6] A. D. Gordon, "Bisimilarity as a theory of functional programming," *Theor. Comput. Sci.*, vol. 228, p. 5–47, Oct. 1999.

[7] S. P. Jones, J.-M. Eber, J. Seward, and S. Peyton Jones, "Composing contracts: an adventure in financial engineering," in *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pp. 280–292, ACM Press, September 2000.

[8] T. Hvitved, F. Klaedtke, and E. Zălinescu, "A trace-based model for multiparty contracts," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 2, pp. 72–98, 2012. Formal Languages and Analysis of Contract-Oriented Software (FLACOS'10).

[9] C. Hoare, B. Möller, G. Struth, and I. Wehrman, "Concurrent kleene algebra," vol. 5710, pp. 399–414, 09 2009.

[10] T. Kappé, P. Brunet, A. Silva, and F. Zanasi, "Concurrent kleene algebra: Free model and completeness," in *Programming Languages and Systems* (A. Ahmed, ed.), (Cham), pp. 856–882, Springer International Publishing, 2018.

[11] A. Salomaa, “Two complete axiom systems for the algebra of regular events,” *J. ACM*, vol. 13, pp. 158–169, 01 1966.

[12] N. A. Danielsson and T. Altenkirch, “Mixing induction and coinduction,” 2009.

[13] M. Almeida, N. Moreira, and R. Reis, “Antimirov and mosses’s rewrite system revisited,” vol. 20, pp. 669–684, 08 2009.

[14] S. Foster and G. Struth, “On the fine-structure of regular algebra,” *J. Autom. Reason.*, vol. 54, p. 165–197, Feb. 2015.

[15] D. Pous, “Kleene algebra with tests and coq tools for while programs,” 2013.

[16] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca, “Canonical big operators,” in *Theorem Proving in Higher Order Logics* (O. A. Mohamed, C. Muñoz, and S. Tahar, eds.), (Berlin, Heidelberg), pp. 86–101, Springer Berlin Heidelberg, 2008.

12 Appendix A: Example derivation of the mechanized calculus

We now show an example derivation in the mechanized calculus.
We start with the goal:

```
forall c : Contract, Star c =C= Star (Star c)
```

The paco library provides the fold tactic to unfold the fixpoint. We apply the tactics:

```
intros.
pfold.
```

The proof state is now:

```
1 subgoal
c : Contract
________________________________________(1/1)
Star c = (upaco2 c_eq bot2)= Star (Star c)
```

We now, normalize on both sides, reshape to Sum-fix friendly shape, apply Sum-fix and simply its premise.

```
unfold_tac.
sum_reshape.
apply c_co_sum. intros.
simp_premise.
```

New proof state:

```
1 subgoal
c : Contract
x : EventType
H0 : In x alphabet
______________________________________(1/1)
upaco2 c_eq bot2 (x \ c _;_ Star c)
  (x \ c _;_ Star c _;_ Star (Star c))
```

Note that `upaco2 f r` is short for `paco2 f r ∪ r`, so we must show the contract pair lies in the union. Since bot2 is the empty relation, it must lie in the set to the left.
We apply the tactic

```
left.
```

New proof state:

```
1 subgoal
c : Contract
x : EventType
H0 : In x alphabet
______________________________________(1/1)
paco2 c_eq bot2 (x \ c _;_ Star c)
  (x \ c _;_ Star c _;_ Star (Star c))
```

Our goal is similar to our original goal, now just with the premise of Sum-fix. We will call this our first iteration.
We apply the tactic

```
pfold
```

New proof state:

```
1 subgoal
c : Contract
x : EventType
H0 : In x alphabet
______________________________________(1/1)
x \ c _;_ Star c = (upaco2 c_eq bot2)=
x \ c _;_ Star c _;_ Star (Star c)
```

The first component of each sequence can be matched up.
We apply the tactics

```
rewrite c_seq_assoc. apply c_seq_ctx. reflexivity.
```

New proof state:

```
1 subgoal
c : Contract
x : EventType
H0 : In x alphabet
______________________________________(1/1)
Star c = (upaco2 c_eq bot2)= Star c _;_ Star (Star c)
```

We finish our second iteration by again normalizing, reshaping, applying Sum-fix, simplifying the premise of Sum-fix and choosing the left set of the union.
We apply the tactics

```
unfold_tac.
sum_reshape.
apply c_co_sum. intros.
simp_premise.
left.
```

New proof state:

```
1 subgoal
c : Contract
x : EventType
H0 : In x alphabet
x0 : EventType
H1 : In x0 alphabet
______________________________________(1/1)
paco2 c_eq bot2 (x0 \ c _;_ Star c)
  (x0 \ c _;_ Star c _;_ Star (Star c)
   _+_ x0 \ c _;_ Star c _;_ Star (Star c))
```

This proof state will be the start of the third iteration. We will soon see that the fourth iteration will have a very similar proof state, only differing in the event that is residuated with.

We generalize x_0 and apply the paco tactic pcofix.

```
generalize x0. pcofix CIH2. intros.
```

New proof state:

```
1 subgoal
c : Contract
x : EventType
H0 : In x alphabet
x0 : EventType
H1 : In x0 alphabet
r : Contract -> Contract -> Prop
```

```
CIH2 : forall x3 : EventType,
       r (x3 \ c _;_ Star c)
         (x3 \ c _;_ Star c _;_ Star (Star c)
          _+_ x3 \ c _;_ Star c _;_ Star (Star c))
x3 : EventType
______________________________________________(1/1)
paco2 c_eq r (x3 \ c _;_ Star c)
  (x3 \ c _;_ Star c _;_ Star (Star c)
   _+_ x3 \ c _;_ Star c _;_ Star (Star c))
```

The effect of pcofix was to add CIH2 has been added to our proof-state [2]. We now unfold.

```
unfold.
```

New proof state (only showing goal)

```
x3 \ c _;_ Star c = (upaco2 c_eq r)=
x3 \ c _;_ Star c _;_ Star (Star c)
_+_ x3 \ c _;_ Star c _;_ Star (Star c)
```

We simply the equation.
We apply tactics:

```
rewrite c_plus_idemp.
rewrite c_seq_assoc. apply c_seq_ctx. reflexivity.
```

New proof state (only showing goal)

```
Star c = (upaco2 c_eq r)= Star c _;_ Star (Star c)
```

We finish the third iteration with the usual steps, this time choosing to show the contract pair lies in the set to the right of the union operator.

```
unfold_tac.
sum_reshape.
apply c_co_sum. intros.
simp_premise.
right.
```

New proof state: (not showing the whole context)

```
CIH2 : forall x3 : EventType,
       r (x3 \ c _;_ Star c)
         (x3 \ c _;_ Star c _;_ Star (Star c)
```

[2] pcofix also changed the relation parameter of the fixpoint from bot2 to r in the goal, to ensure the proof is semantically guarded.

```
          _+_ x3 \ c _;_ Star c _;_ Star (Star c))
x3, x1 : EventType
H2 : In x1 alphabet
______________________________________(1/1)
r (x1 \ c _;_ Star c)
  (x1 \ c _;_ Star c _;_ Star (Star c)
   _+_ x1 \ c _;_ Star c _;_ Star (Star c))
```

The assumption CI2 fits perfectly to our goal, so we apply the tactic

```
apply CIH2
```

After which the proof is finished.

```
No more subgoals.
```

13 Appendix B: More proofs

We show the cases of $+$ and ; of Theorem 6.6, i.e. for all contracts c, $c == |c|$

- Case $c = c_0 + c_1$.
  We must show

  $$c_0 + c_1 == o(c_0 + c_1) + \Sigma_{e \in E}\ event\ e; e \backslash c_0 + e \backslash c_1$$

  We know that $o(c_0 + c_1) = o(c_0) + o(c_1)$, and with distributivity of sequence, the summation is decomposed into

  $$c_0 + c_1 == o(c_0) + o(c_1) + \Sigma_{e \in E}\ event\ e; e \backslash c_0 + e \Sigma event\ e; e \backslash c_1$$

  Applying the IHs of c_0 and c_1 and reordering the terms then yields

  $$\begin{aligned} & o(c_0) + \Sigma_{e \in E}\ event\ e; e \backslash c_0 + o(c_1) + \Sigma_{e \in E}\ event\ e; e \backslash c_1 == \\ & o(c_0) + \Sigma_{e \in E}\ event\ e; e \backslash c_0 + o(c_1) + \Sigma_{e \in E}\ event\ e; e \backslash c_1 \end{aligned}$$

  Which is true by reflexivity.

- Case $c = c_0; c_1$.
  We must show

  $$c_0; c_1 == o(c_0; c_1) + \Sigma_{e \in E}\ event\ e; e \backslash (c_0; c_1)$$

  We apply the IHs on the left hand side, yielding

  $$\begin{aligned} \big(o(c_0) + \Sigma_{e \in E}\ event\ e; e \backslash c_0\big); \big(o(c_1) + \Sigma_{e \in E}\ event\ e; e \backslash c_1\big) == \\ o(c_0; c_1) + \Sigma_{e \in E}\ event\ e; e \backslash (c_0; c_1) \end{aligned}$$

$e\backslash(c_0; c_1)$ is derivably equivalent to $e\backslash c_0; c_1 + o(c_0); c_1$, which can be shown by case distinction on the nullaryness of c_0. We apply this fact along with distribution and sum decomposition on the right hand side to yield.

$$\big(o(c_0) + \Sigma_{e\in E}\ \ e; e\backslash c_0\big); \big(o(c_1) + \Sigma_{e\in E} e; e\backslash c_1\big) ==$$
$$o(c_0; c_1) + \big(\Sigma_{e\in E} e; e\backslash c_0; c_1\big) + \big(\Sigma_{e\in E} e; o(c_0); e\backslash c_1\big)$$

We now distribute on the left-hand-side:

$$o(c_0); o(c_1) + o(c_0); \Sigma_{e\in E} e; e\backslash c_1 + \big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); o(c_1) + \big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); \big(\Sigma_{e\in E} e; e\backslash c_1\big)$$
$$==$$
$$o(c_0; c_1) + \big(\Sigma_{e\in E} e; e\backslash c_0; c_1\big) + \big(\Sigma_{e\in E} e; o(c_0); e\backslash c_1\big)$$

From the fact that $o(c_0); o(c_1) == o(c_0; c_1)$, we match the two left-most terms on either side and must show:

$$o(c_0); \Sigma_{e\in E} e; e\backslash c_1 + \big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); o(c_1) + \big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); \big(\Sigma_{e\in E} e; e\backslash c_1\big)$$
$$==$$
$$\big(\Sigma_{e\in E} e; e\backslash c_0; c_1\big) + \big(\Sigma_{e\in E} e; o(c_0); e\backslash c_1\big)$$

We know that $\Sigma_{e\in E} e; o(c_0); e\backslash c_1 == o(c_0); \big(\Sigma_{e\in E} e; e\backslash c_1\big)$ because if $o(c_0) = Success$; by neutrality of Success it can be reduced away and if $o(c_0) = Failure$, both terms can be reduced to Failure. Matching these terms up and distributing c_1 on the right-hand-side, we are left with showing

$$\big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); o(c_1) + \big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); \big(\Sigma_{e\in E} e; e\backslash c_1\big)$$
$$==$$
$$\big(\Sigma_{e\in E} e; e\backslash c_0\big); c_1$$

We now apply the IH of c_1 on the right-hand-side.

$$\big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); o(c_1) + \big(\Sigma_{e\in E}\ \ e; e\backslash c_0\big); \big(\Sigma_{e\in E} e; e\backslash c_1\big)$$
$$==$$
$$\big(\Sigma_{e\in E} e; e\backslash c_0\big); \big(o(c_1) + \Sigma_{e\in E} e; e\backslash c_1\big)$$

It can now be seen that these terms identical after distributing sequence.

14 Appendix C: Code

14.1 Core.Contract.v

Definitions and semantic equivalence proof for CSL_0.

```
Require Import Lists.List.
Require Import FunInd.
Require Import Bool.Bool.
Require Import Bool.Sumbool.
Require Import Structures.GenericMinMax.
From Equations Require Import Equations.
Import ListNotations.
Require Import micromega.Lia.
Require Import Setoid.
Require Import Init.Tauto btauto.Btauto.
Require Import Logic.ClassicalFacts.

Inductive EventType : Type :=
| Transfer : EventType
| Notify : EventType.

Scheme Equality for EventType.

Definition Trace := List.list EventType % type.

Inductive Contract : Set :=
| Success : Contract
| Failure : Contract
| Event : EventType -> Contract
| CPlus : Contract -> Contract -> Contract
| CSeq : Contract -> Contract -> Contract.

Notation "e _._ c" := (CSeq (Event e) c)
                          (at level 51, right associativity).

Notation "c0 _;_ c1"  := (CSeq c0 c1)
                              (at level 52, left associativity).

Notation "c0 _+_ c1"  := (CPlus c0 c1)
                              (at level 53, left associativity).

Scheme Equality for Contract.


Fixpoint nu(c:Contract):bool :=
match c with
| Success => true
| Failure => false
| Event e => false
| c0 _;_ c1 => nu c0 && nu c1
| c0 _+_ c1 => nu c0 || nu c1
end.
```

```
Reserved Notation "e \ c" (at level 40, left associativity).
Fixpoint derive (e:EventType) (c:Contract) :Contract :=
match c with
| Success => Failure
| Failure => Failure
| Event e' => if (EventType_eq_dec e' e) then Success else Failure
| c0 _;_ c1 => if nu c0 then
                 ((derive e c0) _;_ c1) _+_ (derive e c1)
                else (derive e c0) _;_ c1
| c0 _+_ c1 => e \ c0 _+_ e \ c1
end
where "e \ c" := (derive e c).

Ltac destruct_ctx :=
  repeat match goal with
         | [ H: ?H0 /\ ?H1 |- _ ] => destruct H
         | [ H: exists _, _  |- _ ] => destruct H
         end.

Ltac autoIC := auto with cDB.


Reserved Notation "s \\ c" (at level 42, no associativity).
Fixpoint trace_derive (s : Trace) (c : Contract)  : Contract :=
match s with
| [] => c
| e::s' => s' \\ (e \ c)
end
where "s \\ c" := (trace_derive s c).

Definition matchesb (c : Contract)(s : Trace) := nu (s\\c).


Reserved Notation "s (:) re" (at level 63).

Inductive Matches_Comp : Trace -> Contract -> Prop :=
  | MSuccess : Matches_Comp [] Success
  | MEvent x : Matches_Comp [x] (Event x)
  | MSeq s1 c1 s2 c2
             (H1 : Matches_Comp s1 c1)
             (H2 : Matches_Comp s2 c2)
           : Matches_Comp (s1 ++ s2) (c1 _;_ c2)
  | MPlusL s1 c1 c2
                (H1 : Matches_Comp s1 c1)
              : Matches_Comp s1 (c1 _+_ c2)
  | MPlusR c1 s2 c2
                (H2 : Matches_Comp s2 c2)
              : Matches_Comp s2 (c1 _+_ c2).

Notation "s (:) c" := (Matches_Comp s c)(at level 63).


Hint Constructors Matches_Comp : cDB.
```

```
Ltac eq_event_destruct :=
  repeat match goal with
         | [ |- context[EventType_eq_dec ?e ?e0] ]
                => destruct (EventType_eq_dec e e0);try contradiction
         | [ _ : context[EventType_eq_dec ?e ?e0]  |- _ ]
                => destruct (EventType_eq_dec e e0);try contradiction
         end.

Lemma seq_Success : forall c s, s (:) Success _;_ c <-> s (:) c.
Proof.
split;intros. inversion H. inversion H3. subst. now simpl.
rewrite <- (app_nil_l s). autoIC.
Qed.

Lemma seq_Failure : forall c s, s (:) Failure _;_ c <-> s (:) Failure.
Proof.
split;intros. inversion H. inversion H3. inversion H.
Qed.

Hint Resolve seq_Success seq_Failure : cDB.

Lemma derive_distr_plus : forall (s : Trace)(c0 c1 : Contract),
s \\ (c0 _+_ c1) = s \\ c0 _+_ s \\ c1.
Proof.
induction s;intros;simpl;auto.
Qed.

Hint Rewrite derive_distr_plus : cDB.

Lemma nu_seq_derive : forall (e : EventType)(c0 c1 : Contract),
nu c0 = true -> nu (e \ c1) = true -> nu (e \ (c0 _;_ c1)) = true.
Proof.
intros. simpl. destruct (nu c0). simpl. auto with bool. discriminate.
Qed.


Lemma nu_Failure : forall (s : Trace)(c : Contract),
nu (s \\ (Failure _;_ c)) = false.
Proof.
induction s;intros. now simpl. simpl. auto.
Qed.

Hint Rewrite nu_Failure : cDB.

Lemma nu_Success : forall (s : Trace)(c : Contract),
nu (s \\ (Success _;_ c)) = nu (s \\ c).
Proof.
induction s;intros;simpl;auto.
autorewrite with cDB using simpl;auto.
Qed.

Hint Rewrite nu_Failure nu_Success : cDB.
```

```
Lemma nu_seq_trace_derive : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c0 _;_ c1)) = true.
Proof.
induction s;intros;simpl in *. intuition. destruct (nu c0).
rewrite derive_distr_plus. simpl. auto with bool. discriminate.
Qed.

Lemma matchesb_seq : forall (s0 s1 : Trace)(c0 c1 : Contract),
nu (s0\\c0) = true -> nu (s1\\c1) = true -> nu ((s0++s1)\\(c0 _;_c1)) = true.
Proof.
induction s0;intros;simpl in *.
- rewrite nu_seq_trace_derive; auto.
- destruct (nu c0); autorewrite with cDB; simpl; auto with bool.
Qed.

Hint Rewrite matchesb_seq : cDB.

Lemma Matches_Comp_i_matchesb : forall (c : Contract)(s : Trace),
s (:) c -> nu (s\\c) = true.
Proof.
intros; induction H;
solve [ autorewrite with cDB; simpl; auto with bool
      | simpl;eq_event_destruct;auto ].
Qed.

Lemma Matches_Comp_nil_nu : forall (c : Contract), nu c = true -> [] (:) c.
Proof.
intros;induction c; simpl in H ; try discriminate; autoIC.
apply orb_prop in H. destruct H; autoIC.
rewrite <- (app_nil_l []); autoIC.
Qed.


Lemma Matches_Comp_derive : forall (c : Contract)(e : EventType)(s : Trace),
s (:) e \ c-> (e::s) (:) c.
Proof.
induction c;intros; simpl in*; try solve [inversion H].
- eq_event_destruct. inversion H. subst. autoIC. inversion H.
- inversion H; autoIC.
- destruct (nu c1) eqn:Heqn.
  * inversion H.
    ** inversion H2. subst. rewrite app_comm_cons. auto with cDB.
    ** subst. rewrite <- (app_nil_l (e::s)).
       auto using Matches_Comp_nil_nu with cDB.
  * inversion H. subst. rewrite app_comm_cons. auto with cDB.
Qed.


Theorem Matches_Comp_iff_matchesb : forall (c : Contract)(s : Trace),
s (:) c <-> nu (s \\ c) = true.
Proof.
split;intros.
- auto using Matches_Comp_i_matchesb.
- generalize dependent c. induction s;intros.
```

```
  simpl in H. auto using Matches_Comp_nil_nu.
  auto using Matches_Comp_derive.
Qed.
```

14.2 Core.ContractEquations.v

Axiomatization for CSL_0 with soundness and completeness proof.

```
Require Import CSL.Core.Contract.
Require Import Lists.List Bool.Bool Bool.Sumbool Setoid Coq.Arith.PeanoNat.

Import ListNotations.

Set Implicit Arguments.

Reserved Notation "c0 == c1" (at level 63).

Inductive c_eq : Contract -> Contract -> Prop :=
  | c_plus_assoc c0 c1 c2 :
        (c0 _+_ c1) _+_ c2 == c0 _+_ (c1 _+_ c2)
  | c_plus_comm c0 c1:
        c0 _+_ c1 == c1 _+_ c0
  | c_plus_neut c: c _+_ Failure == c
  | c_plus_idemp c : c _+_ c == c
  | c_seq_assoc c0 c1 c2 :
        (c0 _;_ c1) _;_ c2 == c0 _;_ (c1 _;_ c2)
  | c_seq_neut_l c :
        (Success _;_ c) == c
  | c_seq_neut_r c :
        c _;_ Success == c
  | c_seq_failure_l c :
       Failure _;_ c == Failure
  | c_seq_failure_r c :
       c _;_ Failure == Failure
  | c_distr_l c0 c1 c2 :
     c0 _;_ (c1 _+_ c2) == (c0 _;_ c1) _+_ (c0 _;_ c2)
  | c_distr_r c0 c1 c2 :
     (c0 _+_ c1) _;_ c2 == (c0 _;_ c2) _+_ (c1 _;_ c2)
  | c_refl c : c == c
  | c_sym c0 c1 (H: c0 == c1) : c1 == c0
  | c_trans c0 c1 c2 (H1 : c0 == c1) (H2 : c1 == c2) : c0 == c2
  | c_plus_ctx c0 c0' c1 c1' (H1 : c0 == c0')
                             (H2 : c1 == c1') :
                             c0 _+_ c1 == c0' _+_ c1'
  | c_seq_ctx c0 c0' c1 c1' (H1 : c0 == c0')
                            (H2 : c1 == c1') :
                             c0 _;_ c1 == c0' _;_ c1'
  where "c1 == c2" := (c_eq c1 c2).

Hint Constructors c_eq : eqDB.

Add Parametric Relation : Contract c_eq
  reflexivity proved by c_refl
  symmetry proved by c_sym
  transitivity proved by c_trans
  as Contract_setoid.

Add Parametric Morphism : CPlus with
```

```
signature c_eq ==> c_eq ==> c_eq as c_eq_plus_morphism.
Proof.
  intros. auto with eqDB.
Qed.

Add Parametric Morphism : CSeq with
signature c_eq ==> c_eq ==> c_eq as c_eq_seq_morphism.
Proof.
  intros. auto with eqDB.
Qed.


Ltac c_inversion :=
  (repeat match goal with
         | [ H: _ (:) Failure |- _ ] => inversion H
         | [ H: ?s (:) _ _+_ _ |- _ ] => inversion H; clear H
         | [ H: ?s (:) _ _;_ _ |- _ ] => inversion H; clear H
         | [ H: [?x] (:) Event _ |- _ ] => fail
         | [ H: ?s (:) Event _ |- _ ] => inversion H; subst
         | [ H: [] (:) Success |- _ ] => fail
         | [ H: _ (:) Success |- _ ] => inversion H; clear H
         end);auto with cDB.


Lemma c_eq_soundness : forall (c0 c1 : Contract),
c0 == c1 -> (forall s : Trace, s (:) c0 <-> s (:) c1).
Proof.
intros c0 c1 H. induction H ;intros;
try solve [split;intros;c_inversion].
  * split;intros;c_inversion;
      [ rewrite <- app_assoc | rewrite app_assoc ];
      auto with cDB.
  * rewrite <- (app_nil_l s). split;intros;c_inversion.
  * rewrite <- (app_nil_r s) at 1. split;intros;c_inversion.
      subst. repeat rewrite app_nil_r in H1. now rewrite <- H1.
  * now symmetry.
  * eauto using iff_trans.
  * split;intros; inversion H1; [ rewrite IHc_eq1 in H4
                                | rewrite IHc_eq2 in H4
                                | rewrite <- IHc_eq1 in H4
                                | rewrite <- IHc_eq2 in H4];
                                auto with cDB.
  * split;intros; c_inversion; constructor;
                                [ rewrite <- IHc_eq1
                                | rewrite <- IHc_eq2
                                | rewrite IHc_eq1
                                | rewrite IHc_eq2];
                                auto.
Qed.

Lemma Matches_plus_comm : forall c0 c1 s,
s (:) c0 _+_ c1 <-> s (:) c1 _+_ c0.
Proof. auto using c_eq_soundness  with eqDB. Qed.
```

```
Lemma Matches_plus_neut_l : forall c s,
s (:) Failure _+_ c <-> s (:) c.
Proof.
intros. rewrite Matches_plus_comm.
auto using c_eq_soundness  with eqDB.
Qed.

Lemma Matches_plus_neut_r : forall c s,
s (:) c _+_ Failure <-> s (:) c.
Proof.
auto using c_eq_soundness  with eqDB.
Qed.

Lemma Matches_seq_neut_l : forall c s,
s (:) (Success _;_ c) <-> s (:) c.
Proof.
auto using c_eq_soundness  with eqDB.
Qed.

Lemma Matches_seq_neut_r : forall c s,
s (:) c _;_ Success <-> s (:) c.
Proof. auto using c_eq_soundness  with eqDB. Qed.

Lemma Matches_seq_assoc : forall c0 c1 c2 s,
s (:) (c0 _;_ c1) _;_ c2  <-> s (:) c0 _;_ (c1 _;_ c2).
Proof. auto using c_eq_soundness  with eqDB. Qed.

Hint Rewrite Matches_plus_neut_l
             Matches_plus_neut_r
             Matches_seq_neut_l
             Matches_seq_neut_r : eqDB.

Lemma c_plus_neut_l : forall c, Failure _+_ c == c.
Proof. intros. rewrite c_plus_comm. auto with eqDB.
Qed.

Hint Rewrite c_plus_neut_l
             c_plus_neut
             c_seq_neut_l
             c_seq_neut_r
             c_seq_failure_l
             c_seq_failure_r
             c_distr_l
             c_distr_r : eqDB.

Ltac auto_rwd_eqDB := autorewrite with eqDB;auto with eqDB.


Fixpoint L (c : Contract) : list Trace :=
match c with
| Success => [[]]
| Failure => []
| Event e => [[e]]
| c0 _+_ c1 => (L c0) ++ (L c1)
```

```
| c0 _;_ c1 => map (fun p => (fst p)++(snd p))
                    (list_prod (L c0) (L c1))
end.


Lemma Matches_member : forall (s : Trace)(c : Contract),
s (:) c -> In s (L c).
Proof.
intros. induction H ; simpl ; try solve [auto using in_or_app ||
                                         auto using in_or_app ].
rewrite in_map_iff. exists (s1,s2). rewrite in_prod_iff. split;auto.
Qed.

Lemma member_Matches : forall (c : Contract)(s : Trace),
In s (L c) -> s (:) c.
Proof.
induction c;intros;simpl in*;
  try solve [ destruct H;try contradiction; subst; constructor].
- apply in_app_or in H. destruct H; auto with cDB.
- rewrite in_map_iff in H. destruct_ctx. destruct x.
  rewrite in_prod_iff in H0. destruct H0. simpl in H.
  subst. auto with cDB.
Qed.

Theorem Matches_iff_member : forall s c, s (:) c <-> In s (L c).
Proof.
split; auto using Matches_member,member_Matches.
Qed.

Lemma Matches_incl : forall (c0 c1 : Contract),
(forall (s : Trace), s (:) c0 -> s (:) c1) ->
incl (L c0) (L c1).
Proof.
intros. unfold incl. intros. rewrite <- Matches_iff_member in *. auto.
Qed.

Lemma comp_equiv_destruct : forall (c0 c1: Contract),
(forall s : Trace, s (:) c0 <-> s (:) c1) <->
(forall s : Trace, s (:) c0 -> s (:) c1) /\
(forall s : Trace, s (:) c1 -> s (:) c0).
Proof.
 split ; intros.
- split;intros; specialize H with s; destruct H; auto.
- destruct H. split;intros;auto.
Qed.


Theorem Matches_eq_i_incl_and : forall (c0 c1 : Contract),
(forall (s : Trace), s (:) c0 <-> s (:) c1) ->
incl (L c0) (L c1) /\ incl (L c1) (L c0) .
Proof.
intros. apply comp_equiv_destruct in H.
destruct H. split; auto using Matches_incl.
Qed.
```

```
Fixpoint Σ (l : list Contract) : Contract :=
match l with
| [] => Failure
| c ::l => c _+_ (Σ l)
end.

Lemma Σ_app : forall (l1 l2 : list Contract),
Σ (l1 ++ l2) == (Σ l1) _+_ (Σ l2).
Proof.
induction l1;intros;simpl;auto_rwd_eqDB.
rewrite IHl1. now rewrite c_plus_assoc.
Qed.

Lemma in_Σ : forall (l : list Contract)(s : Trace), s (:) Σ l <->
(exists c, In c l /\ s (:) c).
Proof.
induction l;intros;simpl.
- split;intros. c_inversion. destruct_ctx. contradiction.
- split;intros. c_inversion. exists a. split;auto.
  rewrite IHl in H1. destruct_ctx. exists x. split;auto.
  destruct_ctx. inversion H. subst. auto with cDB.
  apply MPlusR. rewrite IHl. exists x. split;auto.
Qed.

Lemma in_Σ_idemp : forall l c, In c l -> c _+_ Σ l == Σ l.
Proof.
induction l;intros;simpl; auto_rwd_eqDB.
simpl in H;contradiction.
simpl in H. destruct H. subst. all: rewrite <- c_plus_assoc.
auto_rwd_eqDB. rewrite (c_plus_comm c). rewrite c_plus_assoc.
apply c_plus_ctx;auto_rwd_eqDB.
Qed.

Lemma incl_Σ_idemp : forall (l1 l2 : list Contract),
incl l1 l2 -> Σ l2 == Σ (l1++l2).
Proof.
induction l1;intros;simpl;auto_rwd_eqDB.
apply incl_cons_inv in H as [H0 H1].
rewrite <- IHl1;auto. now rewrite in_Σ_idemp;auto.
Qed.

Lemma Σ_app_comm : forall (l1 l2 : list Contract), Σ (l1++l2) == Σ (l2++l1).
Proof.
induction l1;intros;simpl. now rewrite app_nil_r.
repeat rewrite Σ_app. rewrite <- c_plus_assoc.
rewrite c_plus_comm. apply c_plus_ctx;auto_rwd_eqDB.
Qed.

Lemma incl_Σ_c_eq : forall (l1 l2 : list Contract),
incl l1 l2 -> incl l2 l1-> Σ l1 == Σ l2.
Proof.
intros. rewrite (incl_Σ_idemp H).
```

```
rewrite (incl_Σ_idemp H0). apply Σ_app_comm.
Qed.


Fixpoint ∏ (s : Trace) :=
match s with
| [] => Success
| e::s' => (Event e) _;_ (∏ s')
end.

Lemma ∏_app : forall (l1 l2 : Trace), ∏ l1 _;_ ∏ l2 == ∏ (l1++l2).
Proof.
induction l1;intros;simpl; auto_rwd_eqDB.
rewrite <- IHl1. auto_rwd_eqDB.
Qed.

Lemma ∏_distr_aux : forall (ss : list Trace) (s : Trace),
∏ s _;_ (Σ (map ∏ ss)) ==
 Σ (map (fun x => ∏ (fst x ++ snd x))
        (map (fun y : list EventType => (s, y)) ss)).
Proof.
induction ss;intros;simpl;auto_rwd_eqDB.
apply c_plus_ctx;auto using ∏_app.
Qed.


Lemma ∏_distr : forall l0 l1,  Σ (map ∏ l0) _;_ Σ (map ∏ l1) ==
 Σ (map (fun x => ∏ (fst x ++ snd x)) (list_prod l0 l1)).
Proof.
induction l0;intros;simpl. auto_rwd_eqDB.
repeat rewrite map_app. rewrite Σ_app. rewrite <- IHl0.
auto_rwd_eqDB.
apply c_plus_ctx; auto using ∏_distr_aux with eqDB.
Qed.



Theorem ∏_L_ceq : forall (c : Contract), Σ (map ∏ (L c)) == c.
Proof.
induction c; simpl; try solve [auto_rwd_eqDB].
- rewrite map_app. rewrite Σ_app.
  auto using c_plus_ctx.
- rewrite map_map.
  rewrite <- IHc1 at 2. rewrite <- IHc2 at 2.
  symmetry. apply ∏_distr.
Qed.



Lemma c_eq_completeness : forall (c0 c1 : Contract),
(forall s : Trace, s (:) c0 <-> s (:) c1) -> c0 == c1.
Proof.
intros. rewrite <- ∏_L_ceq. rewrite <- (∏_L_ceq c1).
apply Matches_eq_i_incl_and in H.
```

```
destruct H. auto using incl_map, incl_Σ_c_eq.
Qed.



Theorem Matches_iff_c_eq : forall c0 c1,
(forall s, s (:) c0 <-> s (:) c1) <-> c0 == c1.
Proof.
split; auto using c_eq_completeness, c_eq_soundness.
Qed.

Lemma L_Σ : forall l, L (Σ l) = flat_map L l.
Proof.
induction l;intros;simpl;auto. now rewrite IHl.
Qed.
```

14.3 Parallel.Contract.v

Definitions and semantic equivalence proof for $CSL_{||}$.

```
Require Import Lists.List.
Require Import FunInd.
Require Import Bool.Bool.
Require Import Bool.Sumbool.
Require Import Structures.GenericMinMax.
From Equations Require Import Equations.
Import ListNotations.
Require Import micromega.Lia.
Require Import Setoid.
Require Import Init.Tauto btauto.Btauto.
Require Import Logic.ClassicalFacts.


Set Implicit Arguments.

Require CSL.Core.Contract.

Module CSLC := CSL.Core.Contract.
Definition Trace := CSLC.Trace.
Definition EventType := CSLC.EventType.
Definition EventType_eq_dec := CSLC.EventType_eq_dec.
Definition Transfer := CSLC.Transfer.
Definition Notify := CSLC.Notify.

Inductive Contract : Set :=
| Success : Contract
| Failure : Contract
| Event : EventType -> Contract
| CPlus : Contract -> Contract -> Contract
| CSeq : Contract -> Contract -> Contract
| Par : Contract -> Contract -> Contract.



Notation "c0 _;_ c1"  := (CSeq c0 c1)
                         (at level 50, left associativity).

Notation "c0 _||_ c1"  := (Par c0 c1)
                         (at level 52, left associativity).

Notation "c0 _+_ c1"  := (CPlus c0 c1)
                         (at level 53, left associativity).

Scheme Equality for Contract.


Fixpoint nu(c:Contract):bool :=
match c with
| Success => true
| Failure => false
```

```
| Event e => false
| c0 _;_ c1 => nu c0 && nu c1
| c0 _+_ c1 => nu c0 || nu c1
| c0 _||_ c1 => nu c0 && nu c1
end.

Reserved Notation "e \ c" (at level 40, left associativity).
Fixpoint derive (e:EventType) (c:Contract) :Contract :=
match c with
| Success => Failure
| Failure => Failure
| Event e' => if (EventType_eq_dec e' e) then Success else Failure
| c0 _;_ c1 => if nu c0 then
                ((derive e c0) _;_ c1) _+_ (derive e c1)
               else (derive e c0) _;_ c1
| c0 _+_ c1 => e \ c0 _+_ e \ c1
| c0 _||_ c1 => (derive e c0) _||_ c1 _+_ c0 _||_ (derive e c1)
end
where "e \ c" := (derive e c).

Ltac destruct_ctx :=
  repeat match goal with
         | [ H: ?H0 /\ ?H1 |- _ ] => destruct H
         | [ H: exists _, _  |- _ ] => destruct H
         end.



Ltac autoIC := auto with cDB.


Reserved Notation "s \\ c" (at level 42, no associativity).
Fixpoint trace_derive (s : Trace) (c : Contract)  : Contract :=
match s with
| [] => c
| e::s' => s' \\ (e \ c)
end
where "s \\ c" := (trace_derive s c).


Inductive interleave (A : Set) : list A -> list A -> list A -> Prop :=
| IntLeftNil t : interleave nil t t
| IntRightNil t : interleave t nil t
| IntLeftCons t1 t2 t3 e (H: interleave t1 t2 t3) :
                          interleave (e :: t1) t2 (e :: t3)
| IntRightCons t1 t2 t3 e (H: interleave t1 t2 t3) :
                          interleave t1 (e :: t2) (e :: t3).
Hint Constructors interleave : cDB.

Fixpoint interleave_fun (A : Set) (l0 l1 l2 : list A ) : Prop :=
match l2 with
| [] => l0 = [] /\ l1 = []
| a2::l2' => match l0 with
             | [] => l1 = l2
```

```
            | a0::l0' => a2=a0 /\ interleave_fun l0' l1 l2'
             \/ match l1 with
                | [] => l0 = l2
                | a1::l1' => a2=a1 /\ interleave_fun l0 l1' l2'
               end
             end
end.

Lemma interl_fun_nil : forall (A:Set), @interleave_fun A [] [] [].
Proof. intros. unfold interleave_fun. split;auto. Qed.

Hint Resolve interl_fun_nil : cDB.

Lemma interl_fun_l : forall (A:Set) (l : list A), interleave_fun l [] l.
Proof.
induction l;intros; auto with cDB. simpl. now right.
Qed.

Lemma interl_fun_r : forall (A:Set) (l : list A), interleave_fun [] l  l.
Proof.
induction l;intros; auto with cDB. now simpl.
Qed.

Hint Resolve interl_fun_l interl_fun_r : cDB.



Lemma interl_eq_l : forall (A: Set) (l0 l1 : list A),
interleave [] l0 l1 -> l0 = l1.
Proof.
induction l0;intros;simpl.
- inversion H;auto.
- inversion H; subst; auto. f_equal. auto.
Qed.

Lemma interl_comm : forall (A: Set) (l0 l1 l2 : list A),
interleave l0 l1 l2 -> interleave l1 l0 l2.
Proof.
intros. induction H;auto with cDB.
Qed.

Lemma interl_eq_r : forall (A: Set) (l0 l1 : list A),
interleave  l0 [] l1 -> l0 = l1.
Proof. auto using interl_eq_l,interl_comm.
Qed.

Lemma interl_nil : forall (A: Set) (l0 l1 : list A),
interleave  l0 l1 [] -> l0 = [] /\ l1 = [].
Proof.
intros. inversion H;subst; split;auto.
Qed.

Lemma interl_or : forall (A:Set)(l2 l0 l1 :list A)(a0 a1 a2:A),
interleave (a0::l0) (a1::l1) (a2 :: l2) -> a0 = a2 \/ a1 = a2.
```

```
Proof.
intros. inversion H;subst; auto||auto.
Qed.

Lemma interl_i_fun : forall (A:Set)(l0 l1 l2 : list A),
interleave l0 l1 l2 -> interleave_fun l0 l1 l2.
Proof.
intros. induction H;auto with cDB.
- simpl. left. split;auto.
- simpl. destruct t1. apply interl_eq_l in H. now subst. right. split;auto.
Qed.

Lemma fun_i_interl : forall (A:Set)(l2 l0 l1 : list A),
interleave_fun l0 l1 l2 -> interleave l0 l1 l2.
Proof.
induction l2;intros.
- simpl in*. destruct H. subst. constructor.
- simpl in H. destruct l0. subst. auto with cDB.
  destruct H.
  * destruct H. subst. auto with cDB.
  * destruct l1.
    ** inversion H. auto with cDB.
    ** destruct H. subst. auto with cDB.
Qed.

Theorem interl_iff_fun : forall (A:Set)(l2 l0 l1 : list A),
interleave l0 l1 l2 <-> interleave_fun l0 l1 l2.
Proof.
split;auto using interl_i_fun,fun_i_interl.
Qed.

Lemma interl_eq_r_fun : forall (A: Set) (l0 l1 : list A),
interleave_fun  l0 [] l1 -> l0 = l1.
Proof.
intros. rewrite <- interl_iff_fun in H. auto using interl_eq_r.
Qed.

Lemma interl_eq_l_fun : forall (A: Set) (l0 l1 : list A),
interleave_fun  [] l0 l1 -> l0 = l1.
Proof.
intros. rewrite <- interl_iff_fun in H. auto using interl_eq_l.
Qed.

Lemma interl_fun_cons_l : forall (A: Set) (a:A) (l0 l1 l2 : list A),
interleave_fun l0 l1 l2 -> interleave_fun (a::l0) l1 (a::l2).
Proof.
intros. rewrite <- interl_iff_fun in *. auto with cDB.
Qed.

Lemma interl_fun_cons_r : forall (A: Set) (a:A) (l0 l1 l2 : list A),
interleave_fun l0 l1 l2 -> interleave_fun l0 (a::l1) (a::l2).
Proof.
intros. rewrite <- interl_iff_fun in *. auto with cDB.
Qed.
```

```
Hint Rewrite interl_eq_r interl_eq_l interl_eq_r_fun interl_eq_l_fun : cDB.

Hint Resolve interl_fun_cons_l interl_fun_cons_r : cDB.

Ltac interl_tac :=
        (repeat match goal with
         | [ H: _::_ = [] |- _ ] => discriminate
         | [ H: _ /\ _ |- _ ] => destruct H
         | [ H: _ \/ _ |- _ ] => destruct H
         | [ H: interleave_fun _ _ [] |- _ ] => simpl in H
         | [ H: interleave_fun _ _ (?e::?s) |- _ ] => simpl in H
         | [ H: interleave_fun _ _ ?s |- _ ] => destruct s;simpl in H
         | [ H: interleave _ _ _ |- _ ] => rewrite interl_iff_fun in H
         end);subst.

Lemma interl_fun_app : forall (l l0 l1 l_interl l2  : Trace),
interleave_fun l0 l1 l_interl -> interleave_fun l_interl l2 l ->
exists l_interl', interleave_fun l1 l2 l_interl' /\
                  interleave_fun l0 l_interl' l.
Proof.
induction l;intros.
- simpl in H0. destruct H0. subst. simpl in H. destruct H.
  subst.  exists []. split;auto with cDB.
- simpl in H0. destruct l_interl. simpl in H. destruct H. subst.
  exists (a::l). split;auto with cDB.
  destruct H0.
  * destruct H0. subst. simpl in H. destruct l0.
    ** subst. exists (e::l). split;auto with cDB.
    ** destruct H. destruct H. subst.
      *** eapply IHl in H1;eauto. destruct_ctx.
          exists x. split;auto with cDB.
      *** destruct l1.
        **** inversion H. subst. exists l2.
             split;auto with cDB.
        **** destruct H. subst. eapply IHl in H1;eauto. destruct_ctx.
             exists (e1::x). split;auto with cDB;
             apply interl_iff_fun; constructor;
         now rewrite interl_iff_fun.
  * destruct l2.
    ** inversion H0. subst. exists l1. split; auto with cDB.
    ** destruct H0. subst. eapply IHl in H1;eauto. destruct H1.
       exists (e0::x). split; apply interl_iff_fun; constructor;
       destruct H0; now rewrite interl_iff_fun.
Qed.

Lemma interl_app : forall (l l0 l1 l_interl l2  : Trace),
interleave l0 l1 l_interl -> interleave l_interl l2 l ->
exists l_interl', interleave l1 l2 l_interl' /\ interleave l0 l_interl' l.
Proof.
intros. rewrite interl_iff_fun in *.
eapply interl_fun_app in H0;eauto. destruct_ctx. exists x.
repeat rewrite interl_iff_fun. split;auto.
Qed.
```

```
Lemma event_interl : forall s (e0 e1 : EventType),
interleave_fun [e0] [e1] s -> s = [e0]++[e1] \/ s = [e1]++[e0].
Proof.
induction s;intros. simpl in H. destruct H. discriminate.
simpl in H. destruct H.
- destruct H. subst. apply interl_eq_l_fun in H0. subst.
  now left.
- destruct H. subst. apply interl_eq_r_fun in H0. subst.
  now right.
Qed.

Lemma interleave_app : forall (A:Set) (s0 s1: list A),
interleave s0 s1 (s0++s1).
Proof.
induction s0;intros;simpl;auto with cDB.
Qed.

Hint Resolve interleave_app : cDB.

Lemma interleave_app2 : forall (A:Set) (s1 s0: list A),
interleave s0 s1 (s1++s0).
Proof.
induction s1;intros;simpl;auto with cDB.
Qed.

Hint Resolve interleave_app interleave_app2 : cDB.

Lemma interl_extend_r : forall (l0 l1 l2 l3 : Trace),
interleave l0 l1 l2 -> interleave l0 (l1++l3) (l2++l3).
Proof.
intros. generalize dependent l3. induction H;intros;simpl;auto with cDB.
Qed.

Lemma interl_extend_l : forall (l0 l1 l2 l3 : Trace),
interleave l0 l1 l2 -> interleave (l0++l3) l1 (l2++l3).
Proof.
intros. generalize dependent l3. induction H;intros;simpl;auto with cDB.
Qed.


Reserved Notation "s (:) re" (at level 63).
Inductive Matches_Comp : Trace -> Contract -> Prop :=
  | MSuccess : [] (:) Success
  | MEvent x : [x] (:) (Event x)
  | MSeq s1 c1 s2 c2
             (H1 : s1 (:) c1)
             (H2 : s2 (:) c2)
           : (s1 ++ s2) (:) (c1 _;_ c2)
  | MPlusL s1 c1 c2
                (H1 : s1 (:) c1)
              : s1 (:) (c1 _+_ c2)
```

```
  | MPlusR c1 s2 c2
              (H2 : s2 (:) c2)
            : s2 (:) (c1 _+_ c2)
  | MPar s1 c1 s2 c2 s
           (H1 : s1 (:) c1)
           (H2 : s2 (:) c2)
           (H3 : interleave s1 s2 s)
         : s (:) (c1 _||_ c2)
  where "s (:) c" := (Matches_Comp s c).

(*Derive Signature for Matches_Comp.*)

Hint Constructors Matches_Comp : cDB.

Ltac eq_event_destruct :=
  repeat match goal with
         | [ |- context[EventType_eq_dec ?e ?e0] ]
                => destruct (EventType_eq_dec e e0);try contradiction
         | [ _ : context[EventType_eq_dec ?e ?e0]  |- _ ]
                => destruct (EventType_eq_dec e e0);try contradiction
         end.

Lemma seq_Success : forall c s, s (:) Success _;_ c <-> s (:) c.
Proof.
split;intros. inversion H. inversion H3. subst. now simpl.
rewrite <- (app_nil_l s). autoIC.
Qed.

Lemma seq_Failure : forall c s, s (:) Failure _;_ c <-> s (:) Failure.
Proof.
split;intros. inversion H. inversion H3. inversion H.
Qed.

Hint Resolve seq_Success seq_Failure : cDB.

Lemma derive_distr_plus : forall (s : Trace)(c0 c1 : Contract),
s \\ (c0 _+_ c1) = s \\ c0 _+_ s \\ c1.
Proof.
induction s;intros;simpl;auto.
Qed.

Hint Rewrite derive_distr_plus : cDB.

Lemma nu_seq_derive : forall (e : EventType)(c0 c1 : Contract),
nu c0 = true -> nu (e \ c1) = true -> nu (e \ (c0 _;_ c1)) = true.
Proof.
intros. simpl. destruct (nu c0). simpl. auto with bool. discriminate.
Qed.

Lemma nu_Failure : forall (s : Trace)(c : Contract),
nu (s \\ (Failure _;_ c)) = false.
Proof.
induction s;intros. now simpl. simpl. auto.
Qed.
```

```
Hint Rewrite nu_Failure : cDB.

Lemma nu_Success : forall (s : Trace)(c : Contract),
nu (s \\ (Success _;_ c)) = nu (s \\ c).
Proof.
induction s;intros;simpl;auto.
autorewrite with cDB using simpl;auto.
Qed.

Hint Rewrite nu_Failure nu_Success : cDB.

Lemma nu_seq_trace_derive : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c0 _;_ c1)) = true.
Proof.
induction s;intros;simpl in *. intuition. destruct (nu c0).
rewrite derive_distr_plus. simpl. auto with bool. discriminate.
Qed.

Lemma matchesb_seq : forall (s0 s1 : Trace)(c0 c1 : Contract),
nu (s0\\c0) = true -> nu (s1\\c1) = true -> nu ((s0++s1)\\(c0 _;_c1)) = true.
Proof.
induction s0;intros;simpl in *.
- rewrite nu_seq_trace_derive; auto.
- destruct (nu c0); autorewrite with cDB; simpl; auto with bool.
Qed.


Hint Rewrite matchesb_seq : cDB.

Lemma nu_par_trace_derive_r : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c0 _||_ c1)) = true.
Proof.
induction s;intros;simpl in *. intuition.
rewrite derive_distr_plus. simpl. rewrite (IHs c0);auto with bool.
Qed.



Lemma nu_par_trace_derive_l : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c1 _||_ c0)) = true.
Proof.
induction s;intros;simpl in *. intuition.
rewrite derive_distr_plus. simpl. rewrite (IHs c0);auto with bool.
Qed.

Hint Resolve nu_par_trace_derive_l nu_par_trace_derive_r : cDB.

Lemma matchesb_par : forall (s0 s1 s : Trace)(c0 c1 : Contract),
interleave s0 s1 s -> nu (s0\\c0) = true -> nu (s1\\c1) = true ->
nu (s\\(c0 _||_c1)) = true.
Proof.
intros. generalize dependent c1. generalize dependent c0.
induction H;intros;simpl in*; auto with cDB.
```

```
- rewrite derive_distr_plus. simpl. rewrite IHinterleave;auto.
- rewrite derive_distr_plus. simpl. rewrite (IHinterleave c0);auto with bool.
Qed.


Hint Resolve matchesb_par : cDB.


Lemma Matches_Comp_i_matchesb : forall (c : Contract)(s : Trace),
s (:) c -> nu (s\\c) = true.
Proof.
intros; induction H;
solve [ autorewrite with cDB; simpl; auto with bool
      | simpl;eq_event_destruct;eauto with cDB].
Qed.



Lemma Matches_Comp_nil_nu : forall (c : Contract), nu c = true -> [] (:) c.
Proof.
intros;induction c; simpl in H ; try discriminate; autoIC.
- apply orb_prop in H. destruct H; autoIC.
- rewrite <- (app_nil_l []); autoIC.
- apply andb_prop in H. destruct H. eauto with cDB.
Qed.


Lemma Matches_Comp_derive : forall (c : Contract)(e : EventType)(s : Trace),
s (:) e \ c-> (e::s) (:) c.
Proof.
induction c;intros; simpl in*; try solve [inversion H].
- eq_event_destruct. inversion H. subst. autoIC. inversion H.
- inversion H; autoIC.
- destruct (nu c1) eqn:Heqn.
  * inversion H.
    ** inversion H2. subst. rewrite app_comm_cons. auto with cDB.
    ** subst. rewrite <- (app_nil_l (e::s)).
       auto using Matches_Comp_nil_nu with cDB.
  * inversion H. subst. rewrite app_comm_cons. auto with cDB.
- inversion H.
  * inversion H2; subst; eauto with cDB.
  * inversion H1;subst; eauto with cDB.
Qed.

Theorem Matches_Comp_iff_matchesb : forall (c : Contract)(s : Trace),
s (:) c <-> nu (s \\ c) = true.
Proof.
split;intros.
- auto using Matches_Comp_i_matchesb.
- generalize dependent c. induction s;intros.
  simpl in H. auto using Matches_Comp_nil_nu.
  auto using Matches_Comp_derive.
Qed.
```

```
Lemma derive_spec_comp : forall (c : Contract)(e : EventType)(s : Trace),
e::s (:) c <-> s (:) e \ c.
Proof.
intros. repeat rewrite Matches_Comp_iff_matchesb. now simpl.
Qed.
```

14.4 Parallel.ContractEquations.v

Axiomatization for $CSL_{||}$ with soundness and completeness proof.

```
Require Import CSL.Parallel.Contract.
Require Import Lists.List Bool.Bool Bool.Sumbool Setoid Coq.Arith.PeanoNat.
Require Import micromega.Lia.
From Equations Require Import Equations.
Require Import Arith.
Require Import micromega.Lia.

Import ListNotations.

Set Implicit Arguments.

Reserved Notation "c0 =R= c1" (at level 63).

Inductive Sequential : Contract -> Prop :=
 | SeqFailure : Sequential Failure
 | SeqSuccess : Sequential Success
 | SeqEvent e  : Sequential (Event e)
 | SeqPlus c0 c1 (H0: Sequential c0)
                 (H1 : Sequential c1) : Sequential (c0 _+_ c1)
 | SeqSeq c0 c1 (H0: Sequential c0)
                (H1 : Sequential c1) : Sequential (c0 _;_ c1).
Hint Constructors Sequential : eqDB.

Definition bind {A B : Type} (a : option A) (f : A -> option B) : option B :=
  match a with
    | Some x => f x
    | None => None
  end.


Fixpoint translate_aux (c : Contract) : option CSLC.Contract :=
match c with
| Failure => Some CSLC.Failure
| Success => Some CSLC.Success
| Event e => Some (CSLC.Event e)
| c0 _+_ c1 => bind (translate_aux c0)
                    (fun c0' => bind (translate_aux c1)
                                     (fun c1' => Some (CSLC.CPlus c0' c1')))
| c0 _;_ c1 => bind (translate_aux c0)
                    (fun c0' => bind (translate_aux c1)
                                (fun c1' => Some (CSLC.CSeq c0' c1')))
| c0 _||_ c1 => None
end.

Lemma translate_aux_sequential : forall (c : Contract),
Sequential c -> exists c', translate_aux c = Some c'.
Proof.
intros. induction H.
- exists CSLC.Failure. reflexivity.
- exists CSLC.Success. reflexivity.
```

```
- exists (CSLC.Event e). reflexivity.
- destruct IHSequential1,IHSequential2. exists (CSLC.CPlus x x0).
  simpl. unfold bind. destruct (translate_aux c0).
  * destruct (translate_aux c1).
    ** inversion H1. inversion H2. reflexivity.
    ** inversion H2.
  * inversion H1.
- destruct IHSequential1,IHSequential2. exists (CSLC.CSeq x x0).
  simpl. unfold bind. destruct (translate_aux c0).
  * destruct (translate_aux c1).
    ** inversion H1. inversion H2. reflexivity.
    ** inversion H2.
  * inversion H1.
Qed.


Require CSL.Core.ContractEquations.
Module CSLEQ := CSL.Core.ContractEquations.

Ltac option_inversion :=
  (repeat match goal with
         | [ H: None = Some _ |- _ ] => discriminate
         | [ H: Some _ = None |- _ ] => discriminate
         | [ H: Some _ = Some _ |- _ ] => inversion H; clear H
          end);subst.

Ltac c_inversion :=
  (repeat match goal with
         | [ H: _ (:) Failure |- _ ] => inversion H
         | [ H: ?s (:) _ _+_ _ |- _ ] => inversion H; clear H
         | [ H: ?s (:) _ _;_ _ |- _ ] => inversion H; clear H
         | [ H: ?s (:) _ _||_ _ |- _ ] => inversion H; clear H
         | [ H: [?x] (:) Event _ |- _ ] => fail
         | [ H: ?s (:) Event _ |- _ ] => inversion H; subst
         | [ H: [] (:) Success |- _ ] => fail
         | [ H: _ (:) Success |- _ ] => inversion H; clear H

         end); option_inversion; auto with cDB.

Ltac core_inversion := option_inversion;CSLEQ.c_inversion.

Lemma translate_aux_spec : forall (c : Contract)(c' : CSLC.Contract),
translate_aux c = Some c' -> (forall s, s (:) c <-> CSLC.Matches_Comp s c').
Proof.
split. generalize dependent c'. generalize dependent s.
- induction c; intros;simpl in*;c_inversion.
  all: unfold bind in H; destruct (translate_aux c1);try c_inversion;
       destruct (translate_aux c2); c_inversion; c_inversion.
- generalize dependent c'.
  generalize dependent s; induction c; intros;simpl in*.
  * core_inversion.
  * core_inversion.
  * core_inversion.
  * unfold bind in H. destruct (translate_aux c1);try c_inversion.
```

```
    destruct (translate_aux c2);try c_inversion.
    core_inversion; eauto with cDB.
  * unfold bind in H. destruct (translate_aux c1);try c_inversion.
    destruct (translate_aux c2);try c_inversion.
    core_inversion; eauto with cDB.
  * discriminate.
Qed.


Inductive c_eq : Contract -> Contract -> Prop :=
| c_core p0 p1 c0 c1 (H0: translate_aux p0 = Some c0)
                     (H1:translate_aux p1 = Some c1)
                     (H2: CSLEQ.c_eq c0 c1) : p0 =R= p1

| c_plus_assoc c0 c1 c2 : (c0 _+_ c1) _+_ c2 =R= c0 _+_ (c1 _+_ c2)
| c_plus_comm c0 c1: c0 _+_ c1 =R= c1 _+_ c0
| c_plus_neut c: c _+_ Failure =R= c
| c_plus_idemp c : c _+_ c =R= c
| c_seq_assoc c0 c1 c2 : (c0 _;_ c1) _;_ c2 =R= c0 _;_ (c1 _;_ c2)
| c_seq_neut_l c : (Success _;_ c) =R= c
| c_seq_neut_r c : c _;_ Success =R= c
| c_seq_failure_l c : Failure _;_ c =R= Failure
| c_seq_failure_r c :  c _;_ Failure =R= Failure
| c_distr_l c0 c1 c2 : c0 _;_ (c1 _+_ c2) =R= (c0 _;_ c1) _+_ (c0 _;_ c2)
| c_distr_r c0 c1 c2 : (c0 _+_ c1) _;_ c2 =R= (c0 _;_ c2) _+_ (c1 _;_ c2)

| c_par_assoc c0 c1 c2 : (c0 _||_ c1) _||_ c2 =R= c0 _||_ (c1 _||_ c2)
| c_par_neut c : c _||_ Success =R= c
| c_par_comm c0 c1: c0 _||_ c1 =R= c1 _||_ c0
| c_par_failure c : c _||_ Failure =R= Failure
| c_par_distr_l c0 c1 c2 : c0 _||_ (c1 _+_ c2) =R=
                          (c0 _||_ c1) _+_ (c0 _||_ c2)

| c_par_event e0 e1 c0 c1 : Event e0 _;_ c0 _||_ Event e1 _;_ c1 =R=
                           Event e0 _;_ (c0 _||_ (Event e1 _;_ c1)) _+_
                           Event e1 _;_ ((Event e0 _;_ c0) _||_ c1)
| c_refl c : c =R= c
| c_sym c0 c1 (H: c0 =R= c1) : c1 =R= c0
| c_trans c0 c1 c2 (H1 : c0 =R= c1) (H2 : c1 =R= c2) : c0 =R= c2
| c_plus_ctx c0 c0' c1 c1' (H1 : c0 =R= c0')
                          (H2 : c1 =R= c1') : c0 _+_ c1 =R= c0' _+_ c1'
| c_seq_ctx c0 c0' c1 c1' (H1 : c0 =R= c0')
                         (H2 : c1 =R= c1') : c0 _;_ c1 =R= c0' _;_ c1'
| c_par_ctx c0 c0' c1 c1' (H1 : c0 =R= c0')
                         (H2 : c1 =R= c1') : c0 _||_ c1 =R= c0' _||_ c1'
  where "c1 =R= c2" := (c_eq c1 c2).


Hint Constructors c_eq : eqDB.

Add Parametric Relation : Contract c_eq
  reflexivity proved by c_refl
  symmetry proved by c_sym
  transitivity proved by c_trans
```

```
  as Contract_setoid.

Add Parametric Morphism : Par with
signature c_eq ==> c_eq ==> c_eq as c_eq_par_morphism.
Proof.
  intros. auto with eqDB.
Qed.

Add Parametric Morphism : CPlus with
signature c_eq ==> c_eq ==> c_eq as c_eq_plus_morphism.
Proof.
  intros. auto with eqDB.
Qed.

Add Parametric Morphism : CSeq with
signature c_eq ==> c_eq ==> c_eq as c_eq_seq_morphism.
Proof.
  intros. auto with eqDB.
Qed.




(*******************Soundness****************************)
Lemma cons_app : forall (A: Type) (a : A)(l : list A), a::l = [a]++l.
Proof. auto.
Qed.

Lemma event_seq : forall s e0 c0 e1 c1,
s (:) (Event e0 _;_ c0) _||_ (Event e1 _;_ c1) <->
s (:) Event e0 _;_ (c0 _||_ (Event e1 _;_ c1)) _+_
Event e1 _;_ ((Event e0 _;_ c0) _||_ c1).
Proof.
split;intros.
- c_inversion. inversion H5;subst. symmetry in H1. apply app_eq_nil in H1.
  destruct H1;subst;simpl. inversion H8.
  *  apply MPlusL. rewrite cons_app. constructor;auto.
     econstructor;eauto. auto with cDB.
  * inversion H8;subst. simpl in H. inversion H.
    apply MPlusR. rewrite cons_app. constructor;auto;subst.
     econstructor;eauto. eapply MSeq;eauto.
- c_inversion.
  * inversion H6;subst. econstructor. econstructor;eauto.
    econstructor;eauto. simpl in*;auto with cDB.
  * inversion H6;subst. econstructor. econstructor;eauto.
    econstructor;eauto. simpl in*;auto with cDB.
Qed.

Lemma c_eq_soundness : forall (c0 c1 : Contract),
c0 =R= c1 -> (forall s : Trace, s (:) c0 <-> s (:) c1).
Proof.
intros c0 c1 H. induction H ;intros; try solve [split;intros;c_inversion].
  * repeat rewrite translate_aux_spec;eauto. now apply CSLEQ.c_eq_soundness.
  * split;intros;c_inversion; [ rewrite <- app_assoc | rewrite app_assoc ];
```

```
      auto with cDB.
  * rewrite <- (app_nil_l s). split;intros;c_inversion.
  * rewrite <- (app_nil_r s) at 1. split;intros;c_inversion. subst.
    repeat rewrite app_nil_r in H1. now rewrite <- H1.
  * split;intros; inversion H; subst.
    ** inversion H3. subst. eapply interl_app in H5;eauto. destruct_ctx.
       eauto with cDB.
    ** inversion H4. subst. eapply interl_comm in H5.
       eapply interl_comm in H8. eapply interl_app in H5;eauto.
       destruct_ctx. econstructor;eauto. econstructor;eauto.
       all: eauto using interl_comm.
  * split;intros.
    ** inversion H. subst. inversion H4. subst.
       apply interl_eq_r in H5. subst;auto.
    ** eauto with cDB.
  * split;intros.
    ** inversion H. subst. econstructor;eauto using interl_comm.
    ** inversion H. subst. econstructor;eauto using interl_comm.
  * split;intros.
    ** inversion H. subst. inversion H4;  eauto with cDB.
    ** inversion H. subst.
      *** inversion H2. subst. econstructor;eauto with cDB.
      *** inversion H1. subst. econstructor;eauto with cDB.
  * apply event_seq.
  * now symmetry.
  * eauto using iff_trans.
  * split;intros; inversion H1; [ rewrite IHc_eq1 in H4
                                | rewrite IHc_eq2 in H4
                                | rewrite <- IHc_eq1 in H4
                                | rewrite <- IHc_eq2 in H4];
                                auto with cDB.
  * split;intros; c_inversion; constructor;
                                [ rewrite <- IHc_eq1
                                | rewrite <- IHc_eq2
                                | rewrite IHc_eq1
                                | rewrite IHc_eq2];
                                auto.
  * split;intros; c_inversion; econstructor;eauto;
                                [ rewrite <- IHc_eq1
                                | rewrite <- IHc_eq2
                                | rewrite IHc_eq1
                                | rewrite IHc_eq2];
                                auto.
Qed.


Lemma Matches_plus_comm : forall c0 c1 s,
s (:) c0 _+_ c1 <-> s (:) c1 _+_ c0.
Proof. auto using c_eq_soundness  with eqDB. Qed.

Lemma Matches_plus_neut_l : forall c s,
s (:) Failure _+_ c <-> s (:) c.
Proof.
intros. rewrite Matches_plus_comm.
```

```
auto using c_eq_soundness  with eqDB.
Qed.

Lemma Matches_plus_neut_r : forall c s,
s (:) c _+_ Failure <-> s (:) c.
Proof. auto using c_eq_soundness  with eqDB. Qed.

Lemma Matches_seq_neut_l : forall c s,
s (:) (Success _;_ c) <-> s (:) c.
Proof. auto using c_eq_soundness  with eqDB. Qed.

Lemma Matches_seq_neut_r : forall c s,
s (:) c _;_ Success <-> s (:) c.
Proof. auto using c_eq_soundness  with eqDB. Qed.

Lemma Matches_seq_assoc : forall c0 c1 c2 s,
s (:) (c0 _;_ c1) _;_ c2  <-> s (:) c0 _;_ (c1 _;_ c2).
Proof. auto using c_eq_soundness  with eqDB. Qed.

Hint Rewrite Matches_plus_neut_l
             Matches_plus_neut_r
             Matches_seq_neut_l
             Matches_seq_neut_r
             c_par_distr_l
             c_par_neut
             c_par_failure : eqDB.

Lemma c_plus_neut_l : forall c, Failure _+_ c =R= c.
Proof. intros. rewrite c_plus_comm. auto with eqDB.
Qed.

Lemma c_par_neut_l : forall c, Success _||_ c =R= c.
Proof. intros. rewrite c_par_comm. auto with eqDB.
Qed.

Lemma c_par_failure_l  : forall c, Failure _||_ c =R= Failure.
Proof. intros. rewrite c_par_comm. auto with eqDB.
Qed.

Lemma c_par_distr_r : forall c0 c1 c2,
(c0 _+_ c1) _||_ c2 =R= (c0 _||_ c2) _+_ (c1 _||_ c2).
Proof. intros. rewrite c_par_comm. rewrite c_par_distr_l. auto with eqDB.
Qed.


Hint Rewrite c_plus_neut_l
             c_plus_neut
             c_seq_neut_l
             c_seq_neut_r
             c_seq_failure_l
             c_seq_failure_r
             c_distr_l
             c_distr_r
             c_par_neut_l
```

```
            c_par_failure_l
            c_par_distr_r
            c_par_event : eqDB.

Ltac auto_rwd_eqDB := autorewrite with eqDB;auto with eqDB.

Definition alphabet := [Notify;Transfer].

Lemma in_alphabet : forall e, In e alphabet.
Proof.
destruct e ; repeat (try apply in_eq ; try apply in_cons).
Qed.

Hint Resolve in_alphabet : eqDB.
Opaque alphabet.

Fixpoint Σ (A:Type) (l : list A) (f : A -> Contract) : Contract :=
match l with
| [] => Failure
| c ::l => f c _+_ (Σ l f)
end.


Lemma in_Σ : forall (A:Type)(f : A -> Contract)(l : list A)(s : Trace),
s (:) Σ l f <-> (exists c, In c (map f l) /\ s (:) c).
Proof.
induction l;intros;simpl.
- split;intros. c_inversion. destruct_ctx. contradiction.
- split;intros. c_inversion. exists (f a). split;auto.
  rewrite IHl in H1. destruct_ctx. exists x. split;auto.
  destruct_ctx. inversion H. subst. auto with cDB.
  apply MPlusR. rewrite IHl. exists x. split;auto.
Qed.


Definition o c := if nu c then Success else Failure.

Lemma o_plus : forall c0 c1, o (c0 _+_ c1) =R= o c0 _+_ o c1.
Proof.
unfold o. intros. simpl.
destruct (nu c0);destruct (nu c1);simpl;auto_rwd_eqDB.
Qed.

Lemma o_seq : forall c0 c1, o (c0 _;_ c1) =R= o c0 _;_ o c1.
Proof.
unfold o. intros. simpl.
destruct (nu c0);destruct (nu c1);simpl;auto_rwd_eqDB.
Qed.

Lemma o_par : forall c0 c1, o (c0 _||_ c1) =R= o c0 _||_ o c1.
Proof.
unfold o. intros. simpl.
destruct (nu c0);destruct (nu c1);simpl;auto_rwd_eqDB.
Qed.
```

```
Lemma o_true : forall c, nu c = true -> o c = Success.
Proof.
intros. unfold o.
destruct (nu c);auto. discriminate.
Qed.

Lemma o_false : forall c, nu c = false -> o c = Failure.
Proof.
intros. unfold o.
destruct (nu c);auto. discriminate.
Qed.

Lemma o_destruct : forall c, o c = Success \/ o c = Failure.
Proof.
intros. unfold o.
destruct (nu c);auto || auto.
Qed.

Hint Rewrite o_plus o_seq o_par : eqDB.

Hint Rewrite o_true o_false : oDB.


(****************Translation**************)

Inductive Stuck : Contract -> Prop :=
 | STFailure : Stuck Failure
 | STPLus c0 c1 (H0 : Stuck c0) (H1 : Stuck c1) : Stuck (c0 _+_ c1)
 | STSeq c0 c1 (H0 : Stuck c0) : Stuck (c0 _;_ c1)
 | STParL c0 c1 (H0 : Stuck c0) : Stuck (c0 _||_ c1)
 | STParR c0 c1 (H1 : Stuck c1) : Stuck (c0 _||_ c1).
Hint Constructors Stuck : tDB.

Inductive NotStuck : Contract -> Prop :=
 | NSTSuccess : NotStuck Success
 | NSEvent e : NotStuck (Event e)
 | NSTPlusL c0 c1 (H0 : NotStuck c0) : NotStuck (c0 _+_ c1)
 | NSTPlusR c0 c1 (H1 : NotStuck c1) : NotStuck (c0 _+_ c1)
 | NSTSeq c0 c1 (H0 : NotStuck c0) : NotStuck (c0 _;_ c1)
 | NSTPar c0 c1 (H0 : NotStuck c0)(H1 : NotStuck c1) : NotStuck (c0 _||_ c1).

Hint Constructors NotStuck : tDB.

Fixpoint stuck (c : Contract) :=
match c with
| Failure => true
| c0 _+_ c1 => stuck c0 && stuck c1
| c0 _;_ _ => stuck c0
| c0 _||_ c1 => stuck c0 || stuck c1
| _ => false
end.
```

```
Lemma stuck_false : forall (c : Contract), stuck c = false -> NotStuck c.
Proof.
induction c; intros;simpl in*;auto with tDB bool; try discriminate.
apply andb_false_elim in H as [H | H]; auto with tDB.
apply orb_false_iff in H as [H1 H2]; auto with tDB.
Defined.

Lemma stuck_true : forall (c : Contract), stuck c = true -> (Stuck c).
Proof.
induction c; intros; simpl in *; auto with tDB; try discriminate.
apply orb_prop in H as [H | H]; auto with tDB.
Defined.


Definition stuck_dec (c : Contract) : {Stuck c}+{NotStuck c}.
Proof.
destruct (stuck c) eqn:Heqn;
auto using stuck_true || auto using stuck_false.
Defined.

Lemma NotStuck_negation : forall (c : Contract), NotStuck c -> ~(Stuck c).
Proof.
intros. induction H ; intro H2; inversion H2.
all : inversion H2; contradiction.
Qed.

Fixpoint con_size (c:Contract):nat :=
match c with
| Failure => 0
| Success => 1
| Event _ => 2
| c0 _+_ c1 => max (con_size c0) (con_size c1)
| c0 _;_ c1 => if stuck_dec c0 then 0 else (con_size c0) + (con_size c1)
| c0 _||_ c1 => if  sumbool_or _ _ _ _ (stuck_dec c0)
                                         (stuck_dec c1)
                  then 0
                  else (con_size c0) + (con_size c1)
end.

Ltac stuck_tac :=
  (repeat match goal with
         | [ H : _ /\ _ |- _ ] => destruct H
         | [ |- context[if ?a then _ else _] ] => destruct a
         | [ H: Stuck ?c0, H1: NotStuck ?c0  |- _ ]
                          => apply NotStuck_negation in H1; contradiction
  end);auto with tDB.



Lemma stuck_0 : forall (c : Contract), Stuck c -> con_size c = 0.
Proof.
intros. induction H;auto;simpl; try solve [ lia | stuck_tac].
```

```
Defined.

Lemma stuck_not_nullary : forall (c : Contract), Stuck c -> nu c = false.
Proof.
intros. induction H; simpl ;subst ;auto with bool.
all : rewrite IHStuck. all: auto with bool.
rewrite andb_comm. auto with bool.
Defined.

Lemma Stuck_derive : forall (c : Contract)(e : EventType),
Stuck c -> Stuck (e \ c).
Proof.
intros. induction H; simpl in *.
- constructor.
- constructor; auto.
- apply stuck_not_nullary in H. rewrite H. auto with tDB.
- auto with tDB.
- auto with tDB.
Qed.



Lemma Stuck_derive_0 : forall (c : Contract)(e:EventType),
Stuck c -> con_size (e \ c) = 0.
Proof.
intros. apply stuck_0. apply Stuck_derive. assumption.
Qed.

Ltac NotStuck_con H := apply NotStuck_negation in H; contradiction.

Lemma NotStuck_0lt : forall (c : Contract), NotStuck c -> 0 < con_size c.
Proof.
intros. induction H; simpl ; try lia.
- stuck_tac. lia.
- stuck_tac. destruct o0; stuck_tac. lia.
Defined.


Lemma not_stuck_derives : forall (c : Contract),
NotStuck c -> (forall (e : EventType), con_size (e \ c) < con_size c).
Proof.
intros. induction c.
- simpl. lia.
- inversion H.
- simpl. destruct (EventType_eq_dec e0 e) ; simpl ; lia.
- simpl. inversion H.
  * destruct (stuck_dec c2).
    ** apply stuck_0 in s as s2. rewrite (Stuck_derive_0 _ s).
       rewrite Max.max_comm. simpl. apply Max.max_case_strong.
      *** intros. auto.
      *** intros. rewrite s2 in H3. pose proof (NotStuck_0lt H1). lia.
    ** apply IHc1 in H1. apply IHc2 in n. lia.
  * destruct (stuck_dec c1).
    ** apply stuck_0 in s as s2. rewrite (Stuck_derive_0 _ s). simpl.
```

```
      apply Max.max_case_strong.
     *** intros. rewrite s2 in H3. pose proof (NotStuck_0lt H0). lia.
     *** intros. auto.
    ** apply IHc1 in n. apply IHc2 in H0. lia.
- inversion H. subst. simpl. destruct (nu c1) eqn:Heqn.
  * destruct (stuck_dec c1). apply NotStuck_negation in H1. contradiction.
    simpl. destruct (stuck_dec (e \ c1)).
    ** simpl. destruct (stuck_dec c2).
      *** rewrite Stuck_derive_0. pose proof (NotStuck_0lt H1).
          lia. assumption.
      *** rewrite <- (plus_O_n (con_size (e \ c2))). apply IHc2 in n0. lia.
    ** apply IHc1 in H1. destruct (stuck_dec c2).
      *** rewrite (Stuck_derive_0 _ s). rewrite Max.max_comm.
          simpl. apply plus_lt_compat_r. assumption.
      *** apply IHc1 in n. apply IHc2 in n1. lia.
  * destruct (stuck_dec c1).
    ** apply NotStuck_negation in H1. contradiction.
    ** simpl. destruct (stuck_dec (e \ c1)).
      *** pose proof (NotStuck_0lt H1). lia.
      *** apply Plus.plus_lt_compat_r. auto.
- inversion H. subst. simpl.
  destruct (sumbool_or (Stuck (e \ c1)) (NotStuck (e \ c1))
                       (Stuck c2) (NotStuck c2) (stuck_dec (e \ c1))
                       (stuck_dec c2)) as [[o | o] | o].
  * destruct (sumbool_or (Stuck c1) (NotStuck c1)
      (Stuck (e \ c2)) (NotStuck (e \ c2))
      (stuck_dec c1) (stuck_dec (e \ c2))) as [[o0 | o0] | o0].
    ** NotStuck_con H2.
    ** simpl. destruct (sumbool_or (Stuck c1) (NotStuck c1) (Stuck c2)
    (NotStuck c2) (stuck_dec c1) (stuck_dec c2)) as [[o1 | o1] | o1].
      *** NotStuck_con H2.
      *** NotStuck_con H3.
      *** pose proof (NotStuck_0lt H2). lia.
    ** destruct (sumbool_or (Stuck c1) (NotStuck c1) (Stuck c2)
    (NotStuck c2) (stuck_dec c1) (stuck_dec c2)) as [[o1 | o1] | o1].
      *** NotStuck_con H2.
      *** NotStuck_con H3.
      *** simpl. apply plus_lt_compat_l. auto.
  * NotStuck_con H3.
  * destruct o. destruct (sumbool_or (Stuck c1) (NotStuck c1)
      (Stuck (e \ c2)) (NotStuck (e \ c2))
      (stuck_dec c1) (stuck_dec (e \ c2))) as [[o0 | o0] | o0].
    ** NotStuck_con H2.
    ** destruct (sumbool_or (Stuck c1) (NotStuck c1) (Stuck c2)
    (NotStuck c2) (stuck_dec c1) (stuck_dec c2)) as [[o | o] | o].
      *** NotStuck_con H2.
      *** NotStuck_con H3.
      *** rewrite Max.max_comm. simpl. apply plus_lt_compat_r. auto.
    ** destruct (sumbool_or (Stuck c1) (NotStuck c1) (Stuck c2)
    (NotStuck c2) (stuck_dec c1) (stuck_dec c2)) as [[o | o] | o].
      *** NotStuck_con H2.
      *** NotStuck_con H3.
      *** apply Max.max_case_strong.
```

```
        **** intros. apply plus_lt_compat_r. auto.
        **** intros. apply plus_lt_compat_l. auto.
Qed.

Lemma Stuck_failure : forall (c : Contract),
Stuck c -> (forall s, s (:) c <-> s (:) Failure).
Proof.
intros. split. 2: { intros. inversion H0. }
generalize dependent s.  induction c; intros.
- inversion H.
- assumption.
- inversion H.
- inversion H. inversion H0; auto.
- inversion H. inversion H0. apply IHc1 in H7. inversion H7. assumption.
- inversion H0. inversion H.
  * eapply IHc1 in H8. inversion H8. eauto.
  * eapply IHc2 in H8. inversion H8. eauto.
Qed.


Equations plus_norm (c : Contract) : (Contract) by  wf (con_size c) :=
plus_norm  c := if stuck_dec c then Failure
               else (o c) _+_ Σ alphabet
                                 (fun e => (Event e) _;_ (plus_norm (e \ c))).

Next Obligation. auto using not_stuck_derives. Defined.

Global Transparent plus_norm.

Lemma Σ_derive : forall (A:Type)(l : list A)(f : A -> Contract)(e : EventType),
e \ (Σ l f) = Σ l (fun c => e \ f c).
Proof.
induction l;auto;simpl;intros;rewrite IHl;auto.
Qed.

Lemma plus_norm_cons : forall (e:EventType)(s:Trace)(c:Contract),
(forall (e : EventType) (s : Trace), s (:) e \ c <-> s (:) plus_norm (e \ c)) ->
e :: s (:) c <->
e :: s (:) Σ alphabet
             (fun e0 : EventType => Event e0 _;_ plus_norm (e0 \ c)).
Proof.
intros. repeat rewrite derive_spec_comp.
rewrite Σ_derive. rewrite in_Σ.
rewrite H. split;intros.
- exists (Success _;_ (plus_norm (e \ c))). split.
  * rewrite in_map_iff. exists e. split;auto with eqDB.
    simpl. destruct (EventType_eq_dec e e);[ reflexivity | contradiction ].
  * rewrite <- (app_nil_l s). constructor; auto with cDB.
- destruct_ctx. rewrite in_map_iff in H0. destruct_ctx.
  subst. simpl in H1. destruct (EventType_eq_dec x0 e).
  * inversion H1. inversion H5. subst. simpl. assumption.
  * inversion H1. inversion H5.
Qed.
```

```
Lemma plus_norm_nil : forall (c : Contract),
~([] (:) Σ alphabet
           (fun e0 : EventType => Event e0 _;_ plus_norm (e0 \ c))).
Proof.
intros. intro H. apply in_Σ in H as [c0 [P0 P1]].
apply in_map_iff in P0 as [e [P P3]].
subst. inversion P1. apply app_eq_nil in H0 as [H0 H00].
subst. inversion H1.
Qed.



Lemma cons_Success : forall (c : Contract) e s,
e::s (:) Success _+_ c <-> e::s (:) c.
Proof.
split; intros. inversion H. inversion H2. all: auto with cDB.
Qed.

Lemma plus_Failure : forall (c : Contract) s,
s (:) Failure _+_ c <-> s (:) c.
Proof.
intro c. apply c_eq_soundness. auto_rwd_eqDB.
Qed.

Theorem plus_norm_spec : forall (c : Contract)(s : Trace),
s (:) c <-> s (:) plus_norm c.
Proof.
intros. funelim (plus_norm c). destruct (stuck_dec c).
- apply Stuck_failure. assumption.
- destruct s.
  * unfold o. destruct (nu c) eqn:Heqn.
    ** split;intros;auto using Matches_Comp_nil_nu with cDB.
    ** split;intros.
      *** rewrite Matches_Comp_iff_matchesb in H0. simpl in *.
          rewrite Heqn in H0. discriminate.
      *** c_inversion. apply plus_norm_nil in H3 as [].
  * unfold o. destruct (nu c) eqn:Heqn.
    ** rewrite cons_Success. auto using plus_norm_cons.
    ** rewrite plus_Failure. auto using plus_norm_cons.
Qed.



(**********plus_norm respects axiomatization *******)



Lemma Stuck_eq_Failure : forall c, Stuck c -> c =R= Failure.
Proof.
intros. induction H;auto with eqDB.
- rewrite IHStuck1. rewrite IHStuck2. auto_rwd_eqDB.
- rewrite IHStuck. auto_rwd_eqDB.
```

```
- rewrite IHStuck. rewrite c_par_comm. auto_rwd_eqDB.
- rewrite IHStuck. auto_rwd_eqDB.
Qed.



Lemma plus_norm_Failure : plus_norm Failure =R= Failure.
Proof.
simp plus_norm. stuck_tac;auto_rwd_eqDB. inversion n.
Qed.

Lemma Σ_Seq_Failure : forall es,
Σ es (fun e : EventType => Event e _;_ plus_norm Failure) =R= Failure.
Proof.
induction es. simpl. reflexivity.
simpl. rewrite IHes. auto_rwd_eqDB.
Qed.

Lemma plus_norm_Success : plus_norm Success =R= Success.
Proof.
simp plus_norm. stuck_tac. symmetry. auto using Stuck_eq_Failure.
simpl. rewrite Σ_Seq_Failure. auto_rwd_eqDB.
Qed.

Hint Rewrite plus_norm_Failure plus_norm_Success : eqDB.

Ltac eq_m_left := repeat rewrite c_plus_assoc; apply c_plus_ctx;
                  auto_rwd_eqDB.

Ltac eq_m_right := repeat rewrite <- c_plus_assoc; apply c_plus_ctx;
                  auto_rwd_eqDB.


Lemma Σ_alphabet_or : forall alphabet0 e ,
Σ alphabet0
   (fun a : CSLC.EventType => if EventType_eq_dec e a then Success else Failure)
   =R=
   Success /\ In e alphabet0
   \/
Σ alphabet0
  (fun a : CSLC.EventType => if EventType_eq_dec e a then Success else Failure)
  =R= Failure /\ ~(In e alphabet0).
Proof.
induction alphabet0;intros.
- simpl. now right.
- simpl. eq_event_destruct.
  * subst. edestruct IHalphabet0.
    ** destruct H. left. split.
       rewrite H. auto_rwd_eqDB. now left.
    ** destruct H. rewrite H.
       auto_rwd_eqDB.
  * edestruct IHalphabet0; destruct H; rewrite H; auto_rwd_eqDB.
    right. split;auto with eqDB. intro H2. destruct H2.
    symmetry in H1. contradiction. contradiction.
```

```
Qed.

(************Summation rules used in showing
             normalization respects axiomatization*****)
Lemma Σ_alphabet : forall e,
Σ alphabet
  (fun a => if EventType_eq_dec e a then Success else Failure) =R= Success.
Proof.
intros. destruct (Σ_alphabet_or alphabet e).
- destruct H. auto.
- destruct H. pose proof (in_alphabet e). contradiction.
Qed.

Definition fun_eq (f0 f1 : EventType -> Contract) := (forall a, f0 a =R= f1 a).

Add Parametric Morphism l : (Σ l)  with
signature fun_eq  ==> c_eq as c_eq_Σ_morphism.
Proof.
induction l;intros; simpl; auto with eqDB.
Qed.

Notation "f0 =F= f1" := (fun_eq f0 f1)(at level 63).

Lemma fun_eq_refl : forall f, f =F= f.
Proof.
intros. unfold fun_eq. auto with eqDB.
Qed.

Lemma fun_eq_sym : forall f0 f1,f0 =F= f1 -> f1 =F= f0.
Proof.
intros. unfold fun_eq. auto with eqDB.
Qed.

Lemma fun_eq_trans : forall f0 f1 f2,f0 =F= f1 -> f1 =F= f2 -> f0 =F= f2.
Proof.
intros. unfold fun_eq. eauto with eqDB.
Qed.

Add Parametric Relation : (EventType -> Contract) fun_eq
  reflexivity proved by fun_eq_refl
  symmetry proved by fun_eq_sym
  transitivity proved by fun_eq_trans
  as fun_Contract_setoid.



Lemma seq_derive_o : forall e c0 c1, e \ (c0 _;_ c1) =R= e \ c0 _;_ c1 _+_ o (c0) _;_ e \
Proof.
intros;simpl.  destruct (nu c0) eqn:Heqn.
- destruct (o_destruct c0). rewrite H. auto_rwd_eqDB.
  unfold o in H. rewrite Heqn in H. discriminate.
- destruct (o_destruct c0). unfold o in H. rewrite Heqn in H. discriminate.
  rewrite H. auto_rwd_eqDB.
Qed.
```

```
Lemma seq_derive_o_fun : forall c0 c1,
(fun e0 => e0 \ (c0 _;_ c1)) =F=
(fun e0 => e0 \ c0 _;_ c1 _+_ o (c0) _;_ e0 \ c1).
Proof.
intros. unfold fun_eq.  pose proof seq_derive_o. simpl in *. auto.
Qed.

Hint Rewrite seq_derive_o_fun : funDB.

Definition seq_fun (f0 f1 : EventType -> Contract) := fun a => f0 a _;_ f1 a.
Notation "f0 !$\lambda$!;!$\lambda$! f1" := (seq_fun f0 f1)(at level 59).

Lemma to_seq_fun : forall f0 f1, (fun a => f0 a _;_ f1 a) =F= f0 λ;λ f1.
Proof.
intros. unfold seq_fun. reflexivity.
Qed.

Opaque seq_fun.

Add Parametric Morphism : (seq_fun) with
signature fun_eq ==> fun_eq ==> fun_eq as fun_eq_seq_morphism.
Proof.
intros. repeat rewrite <- to_seq_fun.
unfold fun_eq in *. intros. auto with eqDB.
Qed.


Definition plus_fun (f0 f1 : EventType -> Contract) :=
  fun a => f0 a _+_ f1 a.

Notation "f0 !$\lambda$!+!$\lambda$! f1" := (plus_fun f0 f1)(at level 61).
Lemma to_plus_fun : forall f0 f1, (fun a => f0 a _+_ f1 a) =F= f0 λ+λ f1.
Proof.
intros. unfold plus_fun. reflexivity.
Qed.

Opaque plus_fun.

Add Parametric Morphism : (plus_fun) with
signature fun_eq ==> fun_eq ==> fun_eq as fun_eq_plus_morphism.
Proof.
intros. repeat rewrite <- to_plus_fun. unfold fun_eq in *.
intros. auto with eqDB.
Qed.


Definition par_fun (f0 f1 : EventType -> Contract) :=
  fun a => f0 a _||_ f1 a.
Notation "f0 !$\lambda$!||!$\lambda$! f1" := (par_fun f0 f1)(at level 60).
Lemma to_par_fun : forall f0 f1, (fun a => f0 a _||_ f1 a) =F= f0 λ||λ f1.
Proof.
intros. unfold par_fun. reflexivity.
Qed.
```

```
Opaque plus_fun.

Add Parametric Morphism : (par_fun) with
signature fun_eq ==> fun_eq ==> fun_eq as fun_eq_par_morphism.
Proof.
intros. repeat rewrite <- to_par_fun. unfold fun_eq in *.
intros. auto with eqDB.
Qed.

Hint Rewrite to_seq_fun to_plus_fun to_par_fun : funDB.



Lemma Σ_split_plus : forall (A: Type) l (P P' : A -> Contract),
Σ l (fun a : A => P a _+_ P' a) =R=
Σ l (fun a : A => P a) _+_  Σ l (fun a : A => P' a).
Proof.
intros.
induction l;intros.
- simpl. auto_rwd_eqDB.
- simpl. rewrite IHl. eq_m_left. rewrite c_plus_comm. eq_m_left.
Qed.


Lemma Σ_factor_seq_r : forall l (P: EventType -> Contract) c,
Σ l (fun a  => P a _;_ c) =R= Σ l (fun a  => P a) _;_ c.
Proof.
induction l;intros.
- simpl. auto_rwd_eqDB.
- simpl. auto_rwd_eqDB.
Qed.

Lemma Σ_factor_seq_l : forall l (P: EventType -> Contract) c,
Σ l (fun a  => c _;_ P a) =R= c _;_ Σ l (fun a  => P a).
Proof.
induction l;intros.
- simpl. auto_rwd_eqDB.
- simpl. auto_rwd_eqDB.
Qed.


Lemma Σ_factor_par_l : forall l1 c (P' : EventType -> Contract),
Σ l1 (fun a' : EventType => c _||_ P' a') =R=
c _||_ Σ l1 (fun a0 : EventType => P' a0).
Proof.
induction l1;intros.
- simpl. auto_rwd_eqDB.
- simpl. rewrite IHl1. auto_rwd_eqDB.
Qed.

Lemma Σ_factor_par_r : forall l1 c (P' : EventType -> Contract),
Σ l1 (fun a0 : EventType => P' a0) _||_ c =R=
Σ l1 (fun a' : EventType => P' a' _||_ c).
```

```
Proof.
induction l1;intros.
- simpl. auto_rwd_eqDB.
- simpl. rewrite <- IHl1. auto_rwd_eqDB.
Qed.

Lemma Σ_par_ΣΣ : forall l0 l1 (P0 P1 : EventType -> Contract),
Σ l0 (fun a0  => P0 a0) _||_ Σ l1 (fun a1 => P1 a1) =R=
Σ l0 (fun a0  => Σ l1 (fun a1  => (P0 a0) _||_ (P1 a1))).
Proof.
induction l0;intros.
- simpl. auto_rwd_eqDB.
- simpl. auto_rwd_eqDB.
  rewrite Σ_factor_par_l. rewrite IHl0.  reflexivity.
Qed.


Lemma ΣΣ_prod_swap : forall l0 l1 (P : EventType -> EventType -> Contract),
Σ l0 (fun a0 => Σ l1 (fun a1 => P a0 a1)) =R=
Σ l1 (fun a0 => Σ l0 (fun a1 => P a1 a0)).
Proof.
induction l0;intros.
- simpl. induction l1;intros;simpl;auto with eqDB. rewrite IHl1.
  auto with eqDB.
- simpl. rewrite Σ_split_plus. eq_m_left.
Qed.

Lemma fold_Failure : forall l,
Σ l (fun _ : EventType => Failure) =R= Failure.
Proof.
induction l;intros. simpl. reflexivity.
simpl. rewrite IHl. autorewrite with eqDB. reflexivity.
Qed.

Hint Resolve fold_Failure : eqDB.

(*Duplicate some of the rules to the function level*)

Lemma Σ_plus_decomp_fun : forall l f0 f1,
Σ l (f0 λ+λ f1) =R= Σ l f0 _+_  Σ l f1.
Proof.
intros. rewrite <- to_plus_fun. apply Σ_split_plus.
Qed.

Lemma Σ_factor_seq_l_fun : forall l f c,
Σ l ((fun _ => c ) λ;λ f) =R= c _;_ Σ l f.
Proof.
intros. rewrite <- to_seq_fun. apply Σ_factor_seq_l.
Qed.

Lemma Σ_factor_seq_r_fun : forall l f0 c,
Σ l (f0 λ;λ (fun _ => c )) =R= Σ l f0 _;_ c.
Proof.
intros. rewrite <- to_seq_fun. apply Σ_factor_seq_r.
```

```
Qed.


(*Rules for rewriting functions*)
Lemma Σ_distr_l_fun : forall f0 f1 f2,
f0  λ;λ (f1 λ+λ f2) =F= f0 λ;λ f1 λ+λ f0 λ;λ f2.
Proof.
intros. rewrite <- to_plus_fun. rewrite <- to_seq_fun.
symmetry. repeat rewrite <- to_seq_fun. rewrite <- to_plus_fun.
unfold fun_eq. intros. auto_rwd_eqDB.
Qed.


Lemma Σ_distr_par_l_fun : forall f0 f1 f2,
f0  λ||λ (f1 λ+λ f2) =F= f0 λ||λ f1 λ+λ f0 λ||λ f2.
Proof.
intros. rewrite <- to_plus_fun. repeat rewrite <- to_par_fun.
rewrite <- to_plus_fun. unfold fun_eq.  auto with eqDB.
Qed.

Lemma Σ_distr_par_r_fun : forall f0 f1 f2,
(f0 λ+λ f1)  λ||λ f2 =F= f0 λ||λ f2 λ+λ f1 λ||λ f2.
Proof.
intros. rewrite <- to_plus_fun. repeat rewrite <- to_par_fun.
rewrite <- to_plus_fun. unfold fun_eq. intros.  rewrite c_par_distr_r. reflexivity.
Qed.

Lemma Σ_seq_assoc_left_fun : forall f0 f1 f2 ,
f0 λ;λ (f1 λ;λ f2) =F= (f0 λ;λ f1) λ;λ f2.
Proof.
intros. symmetry. rewrite <- (to_seq_fun f0). rewrite <- to_seq_fun.
rewrite <- (to_seq_fun f1). rewrite <- to_seq_fun. unfold fun_eq.
auto with eqDB.
Qed.

Lemma Σ_seq_assoc_right_fun : forall f0 f1 f2 ,
(f0 λ;λ f1) λ;λ f2 =F= f0 λ;λ (f1 λ;λ f2).
Proof.
intros. symmetry. apply Σ_seq_assoc_left_fun.
Qed.

Lemma o_seq_comm_fun : forall c f,
(f λ;λ (fun _ : EventType => o c)) =F= (fun _ : EventType => o c) λ;λ f.
Proof.
intros. repeat rewrite <- to_seq_fun. unfold fun_eq.
intros. destruct (o_destruct c); rewrite H; auto_rwd_eqDB.
Qed.

Hint Rewrite Σ_distr_l_fun Σ_plus_decomp_fun Σ_factor_seq_l_fun
             Σ_factor_seq_r_fun Σ_seq_assoc_left_fun Σ_distr_par_l_fun
             Σ_distr_par_r_fun o_seq_comm_fun : funDB.


Lemma derive_unfold_seq : forall c1 c2,
```

```
o c1 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c1) =R= c1 ->
o c2 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c2) =R= c2 ->
o (c1 _;_ c2) _+_
Σ alphabet (fun a : EventType => Event a _;_ a \ (c1 _;_ c2)) =R=
c1 _;_ c2.
Proof.
intros. rewrite <- H at 2. rewrite <- H0 at 2.
autorewrite with funDB eqDB.
eq_m_left.
rewrite Σ_seq_assoc_right_fun. rewrite Σ_factor_seq_l_fun.
rewrite <- H0 at 1.
autorewrite with eqDB funDB.
rewrite c_plus_assoc.
rewrite (c_plus_comm (Σ _ _ _;_  Σ _ _)).
eq_m_right.
Qed.

Lemma rewrite_in_fun : forall f0 f1,
f0 =F= f1 -> (fun a => f0 a) =F= (fun a => f1 a).
Proof.
intros. unfold fun_eq in*. auto.
Qed.

Lemma rewrite_c_in_fun : forall (c c' : Contract),
c =R= c' -> (fun _ : EventType => c) =F= (fun _ : EventType => c').
Proof.
intros. unfold fun_eq. intros. auto.
Qed.

Lemma fun_neut_r : forall f, f λ||λ (fun _ => Success) =F= f.
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Lemma fun_neut_l : forall f, (fun _ => Success) λ||λ f =F= f.
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Lemma fun_Failure_r : forall f,
f λ||λ (fun _ => Failure) =F= (fun _ => Failure).
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Lemma fun_Failure_l : forall f,
(fun _ => Failure) λ||λ f =F= (fun _ => Failure).
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.
```

```
Hint Rewrite fun_neut_r fun_neut_l fun_Failure_r fun_Failure_l : funDB.

Lemma o_seq_comm_fun3: forall c1 c2,
Σ alphabet
  (Event λ;λ ((fun a : EventType => a \ c1) λ||λ
    (fun _ : EventType => o c2)))
=R=
Σ alphabet
  (Event λ;λ ((fun a : EventType => a \ c1))) _||_ o c2.
Proof.
intros. destruct (o_destruct c2);
rewrite H; autorewrite with funDB eqDB; reflexivity.
Qed.

Lemma o_seq_comm_fun4: forall c1 c2,
Σ alphabet
  (Event λ;λ ((fun _ : EventType => o c1) λ||λ
    (fun a : EventType => a \ c2)))
=R=
o c1 _||_ Σ alphabet (Event λ;λ (fun a : EventType => a \ c2)).
Proof.
intros. destruct (o_destruct c1);
rewrite H; autorewrite with funDB eqDB; reflexivity.
Qed.


Hint Rewrite to_seq_fun to_plus_fun to_par_fun : funDB.


Definition Σ_fun (f : EventType -> EventType -> Contract) :=
  fun a  => Σ alphabet (f a).

Lemma to_Σ_fun : forall f, (fun a : EventType => Σ alphabet (f a)) =F= Σ_fun f.
Proof.
intros. unfold Σ_fun. reflexivity.
Qed.


Definition app a (f : EventType -> Contract) := f a.

Lemma to_app : forall f a, f a = app a f.
Proof.
intros. unfold app. reflexivity.
Qed.

Opaque app.

Add Parametric Morphism a : (app a) with
signature fun_eq ==>  c_eq as afun_eq_par_morphism.
Proof.
intros. repeat rewrite <- to_app. unfold fun_eq in *. intros. auto with eqDB.
Qed.
```

```
Lemma o_seq_comm_fun_fun : forall c1 c2 a,
(fun a1 : EventType => (Event λ;λ (fun a0 : EventType => a0 \ c1)) a _||_
(Event λ;λ (fun a0 : EventType => a0 \ c2)) a1)
=F=
(fun a1 : EventType => (Event a _;_ a \ c1) _||_ Event a1 _;_ a1 \ c2).
Proof.
intros. unfold fun_eq. intros. repeat rewrite to_app.
repeat rewrite <- to_seq_fun.
 apply c_par_ctx; now rewrite <- to_app.
Qed.

Lemma o_seq_comm_fun_fun2 : forall c1 c2 a,
(fun a1 : EventType => (Event a _;_ a \ c1) _||_ Event a1 _;_ a1 \ c2)
=F=
(fun a1 : EventType => (Event a _;_ (a \ c1 _||_ Event a1 _;_ a1 \ c2)))
λ+λ
(fun a1 => Event a1 _;_ (Event a _;_ a \ c1 _||_ a1 \ c2)).
Proof.
intros. rewrite <- to_plus_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.


Lemma derive_unfold_par : forall c1 c2,
o c1 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c1) =R= c1 ->
o c2 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c2) =R= c2 ->
o (c1 _||_ c2) _+_
Σ alphabet (fun a : EventType => Event a _;_ a \ (c1 _||_ c2)) =R=
c1 _||_ c2.
Proof.
intros;simpl.
rewrite <- H at 2. rewrite <- H0 at 2.
rewrite to_seq_fun in *. autorewrite with funDB eqDB.
eq_m_left.
rewrite <- (rewrite_c_in_fun H). rewrite <- (rewrite_c_in_fun H0).
autorewrite with funDB eqDB.
rewrite o_seq_comm_fun3.
rewrite o_seq_comm_fun4.
repeat rewrite <- c_plus_assoc.
rewrite (c_plus_comm _ (_ _||_ o c2)) .
eq_m_left. rewrite (c_plus_comm).
eq_m_left. rewrite Σ_par_ΣΣ.
symmetry.
rewrite rewrite_in_fun. 2: { unfold fun_eq. intros.
                             rewrite o_seq_comm_fun_fun.
                             rewrite o_seq_comm_fun_fun2.
    rewrite Σ_plus_decomp_fun at 1. eapply c_refl. }
rewrite Σ_split_plus. rewrite c_plus_comm.
apply c_plus_ctx.
- rewrite ΣΣ_prod_swap. apply c_eq_Σ_morphism.

rewrite <- to_par_fun.
```

```
repeat rewrite <- to_seq_fun.
unfold fun_eq. intros.
rewrite Σ_factor_seq_l. apply c_seq_ctx. reflexivity.
repeat rewrite <- Σ_factor_par_r.
apply c_par_ctx; auto with eqDB.
- apply c_eq_Σ_morphism. rewrite <- to_par_fun.
repeat rewrite <- to_seq_fun. unfold fun_eq. intros.
rewrite Σ_factor_seq_l. apply c_seq_ctx;auto with eqDB.
repeat rewrite Σ_factor_par_l.
apply c_par_ctx; auto with eqDB.

Qed.

Lemma derive_unfold : forall c,
o c _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c) =R= c.
Proof.
induction c;intros.
- unfold o; simpl. autorewrite with funDB eqDB using reflexivity.
- unfold o. simpl. autorewrite with funDB eqDB. reflexivity.
- unfold o;simpl. autorewrite with funDB eqDB.
  rewrite rewrite_in_fun.
  2: { instantiate (1:= (fun _ => Event e) λ;λ
                         (fun a : EventType => if EventType_eq_dec e a
                                               then Success
                                               else Failure)).
  repeat rewrite <- to_seq_fun. unfold fun_eq. intros. eq_event_destruct.
  subst. reflexivity. auto_rwd_eqDB. }
  rewrite Σ_factor_seq_l_fun. rewrite Σ_alphabet. auto_rwd_eqDB.
- simpl;auto_rwd_eqDB.
  rewrite <- IHc1 at 2. rewrite <- IHc2 at 2. autorewrite with funDB eqDB.
  repeat rewrite <- c_plus_assoc. eq_m_right. eq_m_left.
- auto using derive_unfold_seq.
- auto using derive_unfold_par.
Qed.


Lemma plus_norm_c_eq : forall c, plus_norm c =R= c.
Proof.
intros. funelim (plus_norm c). stuck_tac.
- symmetry. auto using Stuck_eq_Failure.
- rewrite <- (derive_unfold c) at 2. eq_m_left.
  apply c_eq_Σ_morphism. unfold fun_eq. intros.
  rewrite H;auto. reflexivity.
Qed.

Lemma Sequential_Σ : forall (A:Type) (l : list A) f,
(forall a, In a l -> Sequential (f a)) -> Sequential (Σ l f).
Proof.
induction l;intros; auto with eqDB.
simpl. constructor. auto using in_eq.
apply IHl. intros. apply H. simpl. now right.
Qed.

(************Completeness = rewrite to normal form + appeal to CSL_core ***********)
```

```
Lemma plus_norm_Sequential : forall c, Sequential (plus_norm c).
Proof.
intros. funelim (plus_norm c). stuck_tac.
- constructor.
- constructor.
  * destruct (o_destruct c); rewrite H0; auto with eqDB.
  * apply Sequential_Σ. intros. constructor. constructor. auto.
Qed.

Lemma c_eq_completeness : forall (c0 c1 : Contract),
(forall s : Trace, s (:) c0 <-> s (:) c1) ->  c0 =R= c1.
Proof.
intros. rewrite <- plus_norm_c_eq. rewrite <- (plus_norm_c_eq c1).
pose proof (plus_norm_Sequential c0). pose proof (plus_norm_Sequential c1).
apply translate_aux_sequential in H0.
apply translate_aux_sequential in H1. destruct_ctx.
pose proof c_eq_soundness (plus_norm_c_eq c0).
setoid_rewrite <- H2 in H.
pose proof c_eq_soundness (plus_norm_c_eq c1).
setoid_rewrite <- H3 in H.
eapply c_core;eauto. apply CSLEQ.c_eq_completeness.
setoid_rewrite translate_aux_spec in H; eauto.
Qed.
```

14.5 Iteration.Contract.v

Definitions and semantic equivalence proof for CSL_*.

```
Require Import Lists.List.
Require Import FunInd.
Require Import Bool.Bool.
Require Import Bool.Sumbool.
Require Import Structures.GenericMinMax.
From Equations Require Import Equations.
Import ListNotations.
Require Import micromega.Lia.
Require Import Setoid.
Require Import Init.Tauto btauto.Btauto.
Require Import Logic.ClassicalFacts.

Set Implicit Arguments.

Require CSL.Core.Contract.

Module CSLC := CSL.Core.Contract.
Definition Trace := CSLC.Trace.
Definition EventType := CSLC.EventType.
Definition EventType_eq_dec := CSLC.EventType_eq_dec.
Definition EventType_beq := CSLC.EventType_beq.
Definition Transfer := CSLC.Transfer.
Definition Notify := CSLC.Notify.

Inductive Contract : Set :=
| Success : Contract
| Failure : Contract
| Event : EventType -> Contract
| CPlus : Contract -> Contract -> Contract
| CSeq : Contract -> Contract -> Contract
| Par : Contract -> Contract -> Contract
| Star : Contract -> Contract.


Notation "c0 _;_ c1"  := (CSeq c0 c1)
                         (at level 50, left associativity).

Notation "c0 _*_ c1"  := (Par c0 c1)
                         (at level 52, left associativity).

Notation "c0 _+_ c1"  := (CPlus c0 c1)
                         (at level 53, left associativity).

Scheme Equality for Contract.


Fixpoint nu(c:Contract):bool :=
match c with
| Success => true
| Failure => false
```

```
| Event e => false
| c0 _;_ c1 => nu c0 && nu c1
| c0 _+_ c1 => nu c0 || nu c1
| c0 _*_ c1 => nu c0 && nu c1
| Star c => true
end.


Reserved Notation "e \ c" (at level 40, left associativity).
Fixpoint derive (e:EventType) (c:Contract) :Contract :=
match c with
| Success => Failure
| Failure => Failure
| Event e' => if (EventType_eq_dec e' e) then Success else Failure
| c0 _;_ c1 => if nu c0 then
                ((e \ c0) _;_ c1) _+_ (e \ c1)
                else (e \ c0) _;_ c1
| c0 _+_ c1 => e \ c0 _+_ e \ c1
| c0 _*_ c1 => (e \ c0) _*_ c1 _+_ c0 _*_ (e \ c1)
| Star c => e \ c _;_ (Star c)
end
where "e \ c" := (derive e c).

Ltac destruct_ctx :=
  repeat match goal with
         | [ H: ?H0 /\ ?H1 |- _ ] => destruct H
         | [ H: exists _, _  |- _ ] => destruct H
         end.



Ltac autoIC := auto with cDB.


Reserved Notation "s \\ c" (at level 42, no associativity).
Fixpoint trace_derive (s : Trace) (c : Contract)  : Contract :=
match s with
| [] => c
| e::s' => s' \\ (e \ c)
end
where "s \\ c" := (trace_derive s c).


Inductive interleave (A : Set) : list A -> list A -> list A -> Prop :=
| IntLeftNil t : interleave nil t t
| IntRightNil t : interleave t nil t
| IntLeftCons t1 t2 t3 e (H: interleave t1 t2 t3) :
                         interleave (e :: t1) t2 (e :: t3)
| IntRightCons t1 t2 t3 e (H: interleave t1 t2 t3) :
                           interleave t1 (e :: t2) (e :: t3).
Hint Constructors interleave : cDB.

Fixpoint interleave_fun (A : Set) (l0 l1 l2 : list A ) : Prop :=
```

```
match l2 with
| [] => l0 = [] /\ l1 = []
| a2::l2' => match l0 with
             | [] => l1 = l2
             | a0::l0' => a2=a0 /\ interleave_fun l0' l1 l2'
              \/ match l1 with
                 | [] => l0 = l2
                 | a1::l1' => a2=a1 /\ interleave_fun l0 l1' l2'
                 end
             end
end.


Lemma interl_fun_nil : forall (A:Set), @interleave_fun A [] [] [].
Proof. intros. unfold interleave_fun. split;auto. Qed.

Hint Resolve interl_fun_nil : cDB.

Lemma interl_fun_l : forall (A:Set) (l : list A), interleave_fun l [] l.
Proof.
induction l;intros; auto with cDB. simpl. now right.
Qed.

Lemma interl_fun_r : forall (A:Set) (l : list A), interleave_fun [] l  l.
Proof.
induction l;intros; auto with cDB. now simpl.
Qed.

Hint Resolve interl_fun_l interl_fun_r : cDB.



Lemma interl_eq_l : forall (A: Set) (l0 l1 : list A),
interleave [] l0 l1 -> l0 = l1.
Proof.
induction l0;intros;simpl.
- inversion H;auto.
- inversion H; subst; auto. f_equal. auto.
Qed.

Lemma interl_comm : forall (A: Set) (l0 l1 l2 : list A),
interleave l0 l1 l2 -> interleave l1 l0 l2.
Proof.
intros. induction H;auto with cDB.
Qed.

Lemma interl_eq_r : forall (A: Set) (l0 l1 : list A),
interleave  l0 [] l1 -> l0 = l1.
Proof. auto using interl_eq_l,interl_comm.
Qed.

Lemma interl_nil : forall (A: Set) (l0 l1 : list A),
interleave  l0 l1 [] -> l0 = [] /\ l1 = [].
Proof.
```

```
intros. inversion H;subst; split;auto.
Qed.

Lemma interl_or : forall (A:Set)(l2 l0 l1 :list A)(a0 a1 a2:A),
interleave (a0::l0) (a1::l1) (a2 :: l2) -> a0 = a2 \/ a1 = a2.
Proof.
intros. inversion H;subst; auto||auto.
Qed.

Lemma interl_i_fun : forall (A:Set)(l0 l1 l2 : list A),
interleave l0 l1 l2 -> interleave_fun l0 l1 l2.
Proof.
intros. induction H;auto with cDB.
- simpl. left. split;auto.
- simpl. destruct t1. apply interl_eq_l in H. now subst. right. split;auto.
Qed.

Lemma fun_i_interl : forall (A:Set)(l2 l0 l1 : list A),
interleave_fun l0 l1 l2 -> interleave l0 l1 l2.
Proof.
induction l2;intros.
- simpl in*. destruct H. subst. constructor.
- simpl in H. destruct l0. subst. auto with cDB.
  destruct H.
  * destruct H. subst. auto with cDB.
  * destruct l1.
    ** inversion H. auto with cDB.
    ** destruct H. subst. auto with cDB.
Qed.

Theorem interl_iff_fun : forall (A:Set)(l2 l0 l1 : list A),
interleave l0 l1 l2 <-> interleave_fun l0 l1 l2.
Proof.
split;auto using interl_i_fun,fun_i_interl.
Qed.

Lemma interl_eq_r_fun : forall (A: Set) (l0 l1 : list A),
interleave_fun  l0 [] l1 -> l0 = l1.
Proof.
intros. rewrite <- interl_iff_fun in H. auto using interl_eq_r.
Qed.

Lemma interl_eq_l_fun : forall (A: Set) (l0 l1 : list A),
interleave_fun  [] l0 l1 -> l0 = l1.
Proof.
intros. rewrite <- interl_iff_fun in H. auto using interl_eq_l.
Qed.

Lemma interl_fun_cons_l : forall (A: Set) (a:A) (l0 l1 l2 : list A),
interleave_fun l0 l1 l2 -> interleave_fun (a::l0) l1 (a::l2).
Proof.
intros. rewrite <- interl_iff_fun in *. auto with cDB.
Qed.
```

```
Lemma interl_fun_cons_r : forall (A: Set) (a:A) (l0 l1 l2 : list A),
interleave_fun l0 l1 l2 -> interleave_fun l0 (a::l1) (a::l2).
Proof.
intros. rewrite <- interl_iff_fun in *. auto with cDB.
Qed.

Hint Rewrite interl_eq_r interl_eq_l interl_eq_r_fun interl_eq_l_fun : cDB.

Hint Resolve interl_fun_cons_l interl_fun_cons_r : cDB.

Ltac interl_tac :=
        (repeat match goal with
         | [ H: _::_ = [] |- _ ] => discriminate
         | [ H: _ /\ _ |- _ ] => destruct H
         | [ H: _ \/ _ |- _ ] => destruct H
         | [ H: interleave_fun _ _ [] |- _ ] => simpl in H
         | [ H: interleave_fun _ _ (?e::?s) |- _ ] => simpl in H
         | [ H: interleave_fun _ _ ?s |- _ ] => destruct s;simpl in H
         | [ H: interleave _ _ _ |- _ ] => rewrite interl_iff_fun in H
         end);subst.

Lemma interl_fun_app : forall (l l0 l1 l_interl l2  : Trace),
interleave_fun l0 l1 l_interl -> interleave_fun l_interl l2 l ->
exists l_interl', interleave_fun l1 l2 l_interl' /\
                  interleave_fun l0 l_interl' l.
Proof.
induction l;intros.
- simpl in H0. destruct H0. subst. simpl in H. destruct H.
  subst.  exists []. split;auto with cDB.
- simpl in H0. destruct l_interl. simpl in H. destruct H. subst.
  exists (a::l). split;auto with cDB.
  destruct H0.
  * destruct H0. subst. simpl in H. destruct l0.
    ** subst. exists (e::l). split;auto with cDB.
    ** destruct H. destruct H. subst.
      *** eapply IHl in H1;eauto. destruct_ctx.
          exists x. split;auto with cDB.
      *** destruct l1.
        **** inversion H. subst. exists l2.
             split;auto with cDB.
        **** destruct H. subst. eapply IHl in H1;eauto. destruct_ctx.
             exists (e1::x). split;auto with cDB;
             apply interl_iff_fun; constructor;
         now rewrite interl_iff_fun.
  * destruct l2.
    ** inversion H0. subst. exists l1. split; auto with cDB.
    ** destruct H0. subst. eapply IHl in H1;eauto. destruct H1.
       exists (e0::x). split; apply interl_iff_fun; constructor;
       destruct H0; now rewrite interl_iff_fun.
Qed.

Lemma interl_app : forall (l l0 l1 l_interl l2  : Trace),
interleave l0 l1 l_interl -> interleave l_interl l2 l ->
exists l_interl', interleave l1 l2 l_interl' /\ interleave l0 l_interl' l.
```

```
Proof.
intros. rewrite interl_iff_fun in *.
eapply interl_fun_app in H0;eauto. destruct_ctx. exists x.
repeat rewrite interl_iff_fun. split;auto.
Qed.


Lemma event_interl : forall s (e0 e1 : EventType),
interleave_fun [e0] [e1] s -> s = [e0]++[e1] \/ s = [e1]++[e0].
Proof.
induction s;intros. simpl in H. destruct H. discriminate.
simpl in H. destruct H.
- destruct H. subst. apply interl_eq_l_fun in H0. subst.
  now left.
- destruct H. subst. apply interl_eq_r_fun in H0. subst.
  now right.
Qed.

Lemma interleave_app : forall (A:Set) (s0 s1: list A),
interleave s0 s1 (s0++s1).
Proof.
induction s0;intros;simpl;auto with cDB.
Qed.

Hint Resolve interleave_app : cDB.

Lemma interleave_app2 : forall (A:Set) (s1 s0: list A),
interleave s0 s1 (s1++s0).
Proof.
induction s1;intros;simpl;auto with cDB.
Qed.

Hint Resolve interleave_app interleave_app2 : cDB.

Lemma interl_extend_r : forall (l0 l1 l2 l3 : Trace),
interleave l0 l1 l2 -> interleave l0 (l1++l3) (l2++l3).
Proof.
intros. generalize dependent l3. induction H;intros;simpl;auto with cDB.
Qed.

Lemma interl_extend_l : forall (l0 l1 l2 l3 : Trace),
interleave l0 l1 l2 -> interleave (l0++l3) l1 (l2++l3).
Proof.
intros. generalize dependent l3. induction H;intros;simpl;auto with cDB.
Qed.


Reserved Notation "s (:) re" (at level 63).
Inductive Matches_Comp : Trace -> Contract -> Prop :=
  | MSuccess : [] (:) Success
  | MEvent x : [x] (:) (Event x)
  | MSeq s1 c1 s2 c2
             (H1 : s1 (:) c1)
```

```
              (H2 : s2 (:) c2)
            : (s1 ++ s2) (:) (c1 _;_ c2)
| MPlusL s1 c1 c2
                (H1 : s1 (:) c1)
              : s1 (:) (c1 _+_ c2)
| MPlusR c1 s2 c2
                (H2 : s2 (:) c2)
              : s2 (:) (c1 _+_ c2)
| MPar s1 c1 s2 c2 s
              (H1 : s1 (:) c1)
              (H2 : s2 (:) c2)
              (H3 : interleave s1 s2 s)
            : s (:) (c1 _*_ c2)
| MStar0 c
              : [] (:) Star c
| MStarSeq c s1 s2 (H1: s1 (:) c)
                   (H2: s2 (:) Star c)
              : s1 ++ s2 (:) Star c
where "s (:) c" := (Matches_Comp s c).

(*Derive Signature for Matches_Comp.*)

Hint Constructors Matches_Comp : cDB.

Ltac eq_event_destruct :=
  repeat match goal with
         | [ |- context[EventType_eq_dec ?e ?e0] ]
                => destruct (EventType_eq_dec e e0);try contradiction
         | [ _ : context[EventType_eq_dec ?e ?e0]  |- _ ]
                => destruct (EventType_eq_dec e e0);try contradiction
         end.

Lemma seq_Success : forall c s, s (:) Success _;_ c <-> s (:) c.
Proof.
split;intros. inversion H. inversion H3. subst. now simpl.
rewrite <- (app_nil_l s). autoIC.
Qed.

Lemma seq_Failure : forall c s, s (:) Failure _;_ c <-> s (:) Failure.
Proof.
split;intros. inversion H. inversion H3. inversion H.
Qed.

Hint Resolve seq_Success seq_Failure : cDB.


Lemma derive_distr_plus : forall (s : Trace)(c0 c1 : Contract),
 s \\ (c0 _+_ c1) = s \\ c0 _+_ s \\ c1.
Proof.
induction s;intros;simpl;auto.
Qed.

Hint Rewrite derive_distr_plus : cDB.
```

```
Lemma nu_seq_derive : forall (e : EventType)(c0 c1 : Contract),
nu c0 = true -> nu (e \ c1) = true -> nu (e \ (c0 _;_ c1)) = true.
Proof.
intros. simpl. destruct (nu c0). simpl. auto with bool. discriminate.
Qed.


Lemma nu_Failure : forall (s : Trace)(c : Contract),
nu (s \\ (Failure _;_ c)) = false.
Proof.
induction s;intros. now simpl. simpl. auto.
Qed.

Hint Rewrite nu_Failure : cDB.

Lemma nu_Success : forall (s : Trace)(c : Contract),
nu (s \\ (Success _;_ c)) = nu (s \\ c).
Proof.
induction s;intros;simpl;auto.
autorewrite with cDB using simpl;auto.
Qed.

Hint Rewrite nu_Failure nu_Success : cDB.

Lemma nu_seq_trace_derive : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c0 _;_ c1)) = true.
Proof.
induction s;intros;simpl in *. intuition. destruct (nu c0).
rewrite derive_distr_plus. simpl. auto with bool. discriminate.
Qed.

Lemma matchesb_seq : forall (s0 s1 : Trace)(c0 c1 : Contract),
nu (s0\\c0) = true -> nu (s1\\c1) = true -> nu ((s0++s1)\\(c0 _;_c1)) = true.
Proof.
induction s0;intros;simpl in *.
- rewrite nu_seq_trace_derive; auto.
- destruct (nu c0); autorewrite with cDB; simpl; auto with bool.
Qed.


Hint Rewrite matchesb_seq : cDB.

Lemma nu_par_trace_derive_r : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c0 _*_ c1)) = true.
Proof.
induction s;intros;simpl in *. intuition.
rewrite derive_distr_plus. simpl. rewrite (IHs c0);auto with bool.
Qed.



Lemma nu_par_trace_derive_l : forall (s : Trace)(c0 c1 : Contract),
nu c0 = true -> nu (s \\ c1) = true -> nu (s \\ (c1 _*_ c0)) = true.
Proof.
```

```
induction s;intros;simpl in *. intuition.
rewrite derive_distr_plus. simpl. rewrite (IHs c0);auto with bool.
Qed.

Hint Resolve nu_par_trace_derive_l nu_par_trace_derive_r : cDB.

Lemma matchesb_par : forall (s0 s1 s : Trace)(c0 c1 : Contract),
interleave s0 s1 s -> nu (s0\\c0) = true -> nu (s1\\c1) = true ->
nu (s\\(c0 _*_c1)) = true.
Proof.
intros. generalize dependent c1. generalize dependent c0.
induction H;intros;simpl in*; auto with cDB.
- rewrite derive_distr_plus. simpl. rewrite IHinterleave;auto.
- rewrite derive_distr_plus. simpl.
  rewrite (IHinterleave c0);auto with bool.
Qed.


Hint Resolve matchesb_par matchesb_seq : cDB.


Lemma Matches_Comp_i_matchesb : forall (c : Contract)(s : Trace),
s (:) c -> nu (s\\c) = true.
Proof.
intros; induction H;
solve [ autorewrite with cDB; simpl; auto with bool
      | simpl;eq_event_destruct;eauto with cDB
      | destruct s1; simpl in*; auto with cDB].
Qed.



Lemma Matches_Comp_nil_nu : forall (c : Contract), nu c = true -> [] (:) c.
Proof.
intros;induction c; simpl in H ; try discriminate; autoIC.
- apply orb_prop in H. destruct H; autoIC.
- rewrite <- (app_nil_l []); autoIC.
- apply andb_prop in H. destruct H. eauto with cDB.
Qed.

Lemma Matches_Comp_derive : forall (c : Contract)(e : EventType)(s : Trace),
s (:) e \ c-> (e::s) (:) c.
Proof.
induction c;intros; simpl in*; try solve [inversion H].
- eq_event_destruct. inversion H. subst. autoIC. inversion H.
- inversion H; autoIC.
- destruct (nu c1) eqn:Heqn.
  * inversion H.
    ** inversion H2. subst. rewrite app_comm_cons. auto with cDB.
    ** subst. rewrite <- (app_nil_l (e::s)).
       auto using Matches_Comp_nil_nu with cDB.
  * inversion H. subst. rewrite app_comm_cons. auto with cDB.
- inversion H.
  * inversion H2; subst; eauto with cDB.
```

```
  * inversion H1;subst; eauto with cDB.
- inversion H. rewrite app_comm_cons. auto with cDB.
Qed.

Theorem Matches_Comp_iff_matchesb : forall (c : Contract)(s : Trace),
s (:) c <-> nu (s \\ c) = true.
Proof.
split;intros.
- auto using Matches_Comp_i_matchesb.
- generalize dependent c. induction s;intros.
  simpl in H. auto using Matches_Comp_nil_nu.
  auto using Matches_Comp_derive.
Qed.

Lemma derive_spec_comp : forall (c : Contract)(e : EventType)(s : Trace),
 e::s (:) c <-> s (:) e \ c.
Proof.
intros. repeat rewrite Matches_Comp_iff_matchesb. now simpl.
Qed.
```

14.6 Iteration.ContractEquations.v

Axiomatization for CSL_* with soundness and completeness proof.

```
Require Import CSL.Iteration.Contract.
Require Import Lists.List Bool.Bool Bool.Sumbool Setoid Coq.Arith.PeanoNat.
Require Import micromega.Lia.
From Equations Require Import Equations.
Require Import Arith.
Require Import micromega.Lia.

Require Import Paco.paco.


Import ListNotations.

Set Implicit Arguments.
Inductive bisimilarity_gen bisim : Contract -> Contract -> Prop :=
 bisimilarity_con c0 c1 (H0: forall e, bisim (e \ c0) (e \ c1) : Prop )
                        (H1: nu c0 = nu c1) : bisimilarity_gen bisim c0 c1.

Definition Bisimilarity c0 c1 := paco2 bisimilarity_gen bot2 c0 c1.
Hint Unfold  Bisimilarity : core.

Lemma bisimilarity_gen_mon: monotone2 bisimilarity_gen.
Proof.
unfold monotone2. intros.  constructor. inversion IN. intros.
auto. inversion IN. auto.
Qed.
Hint Resolve bisimilarity_gen_mon : paco.


Theorem matches_eq_i_bisimilarity : forall c0 c1,
(forall s, s(:) c0 <-> s(:)c1) -> Bisimilarity c0 c1.
Proof.
pcofix CIH. intros. pfold. constructor.
- intros. right. apply CIH.  setoid_rewrite <- derive_spec_comp. auto.
- apply eq_true_iff_eq. setoid_rewrite Matches_Comp_iff_matchesb in H0.
  specialize H0 with []. simpl in*. auto.
Qed.

Theorem bisimilarity_i_matches_eq : forall c0 c1,
Bisimilarity c0 c1 -> (forall s, s(:) c0 <-> s(:)c1).
Proof.
intros. generalize dependent c1. generalize dependent c0.
induction s;intros.
- repeat rewrite Matches_Comp_iff_matchesb. simpl.
  rewrite <- eq_iff_eq_true.  punfold H. inversion H. auto.
- repeat rewrite derive_spec_comp. apply IHs. punfold H.
  inversion_clear H. specialize H0 with a. pclearbot. auto.
Qed.

Theorem matches_eq_iff_bisimilarity : forall c0 c1,
(forall s, s(:) c0 <-> s(:)c1) <-> Bisimilarity c0 c1.
```

```
Proof.
split;auto using matches_eq_i_bisimilarity, bisimilarity_i_matches_eq.
Qed.



Definition alphabet := [Notify;Transfer].

Lemma in_alphabet : forall e, In e alphabet.
Proof.
destruct e ; repeat (try apply in_eq ; try apply in_cons).
Qed.

Hint Resolve in_alphabet : eqDB.
Opaque alphabet.
(*
Fixpoint !Σ! (l : list Contract) : Contract :=
match l with
| [] => Failure
| c ::l => c _+_ (!Σ! l)
end.
*)

Fixpoint Σ (A:Type) (l : list A) (f : A -> Contract) : Contract :=
match l with
| [] => Failure
| c ::l => f c _+_ (Σ l f)
end.

Definition Σe es cs := Σ (combine es cs) (fun x => Event (fst x) _;_ snd x).


Definition Σed c := (Σ alphabet (fun a : EventType => Event a _;_ a \ c)).
Notation "!$\Sigma$!e\ c" := (Σed c)
                    (at level 30, no associativity).

Reserved Notation "c0 =R= c1" (at level 63).

Section axiomatization.
  Variable co_eq : Contract -> Contract -> Prop.

Inductive c_eq : Contract -> Contract -> Prop :=
| c_plus_assoc c0 c1 c2 : (c0 _+_ c1) _+_ c2 =R= c0 _+_ (c1 _+_ c2)
| c_plus_comm c0 c1: c0 _+_ c1 =R= c1 _+_ c0
| c_plus_neut c: c _+_ Failure =R= c
| c_plus_idemp c : c _+_ c =R= c
| c_seq_assoc c0 c1 c2 : (c0 _;_ c1) _;_ c2 =R= c0 _;_ (c1 _;_ c2)
| c_seq_neut_l c : (Success _;_ c) =R= c
| c_seq_neut_r c : c _;_ Success =R= c
| c_seq_failure_l c : Failure _;_ c =R= Failure
| c_seq_failure_r c :  c _;_ Failure =R= Failure
| c_distr_l c0 c1 c2 : c0 _;_ (c1 _+_ c2) =R= (c0 _;_ c1) _+_ (c0 _;_ c2)
| c_distr_r c0 c1 c2 : (c0 _+_ c1) _;_ c2 =R= (c0 _;_ c2) _+_ (c1 _;_ c2)
```

```
| c_par_assoc c0 c1 c2 : (c0 _*_ c1) _*_ c2 =R= c0 _*_ (c1 _*_ c2)
| c_par_neut c : c _*_ Success =R= c
| c_par_comm c0 c1: c0 _*_ c1 =R= c1 _*_ c0
| c_par_failure c : c _*_ Failure =R= Failure
| c_par_distr_l c0 c1 c2 : c0 _*_ (c1 _+_ c2) =R= (c0 _*_ c1) _+_ (c0 _*_ c2)

| c_par_event e0 e1 c0 c1 : Event e0 _;_ c0 _*_ Event e1 _;_ c1 =R=
                            Event e0 _;_ (c0 _*_ (Event e1 _;_ c1)) _+_
                            Event e1 _;_ ((Event e0 _;_ c0) _*_ c1)

| c_unfold c :  Success _+_ (c _;_ Star c) =R= Star c
| c_star_plus c :  Star (Success _+_ c) =R= Star c
| c_refl c : c =R= c
| c_sym c0 c1 (H: c0 =R=c1) : c1 =R=c0
| c_trans c0 c1 c2 (H1 : c0 =R=c1) (H2 : c1 =R=c2) : c0 =R=c2
| c_plus_ctx c0 c0' c1 c1' (H1 : c0 =R=c0')
                           (H2 : c1 =R=c1') : c0 _+_ c1 =R=c0' _+_ c1'
| c_seq_ctx c0 c0' c1 c1' (H1 : c0 =R=c0')
                          (H2 : c1 =R=c1') : c0 _;_ c1 =R=c0' _;_ c1'
| c_par_ctx c0 c0' c1 c1' (H1 : c0 =R=c0')
                          (H2 : c1 =R=c1') : c0 _*_ c1 =R=c0' _*_ c1'
| c_star_ctx c0 c1 (H : c0 =R=c1) : Star c0 =R= Star c1
| c_co_sum es ps  (H: forall p, In p ps -> co_eq (fst p) (snd p) : Prop)
                : (Σe es (map fst ps)) =R= (Σe es (map snd ps))
 where "c1 =R= c2" := (c_eq c1 c2).

End axiomatization.

Notation "c0 = ( R ) = c1" := (c_eq R c0 c1)(at level 63).

Hint Constructors c_eq : eqDB.


Add Parametric Relation R : Contract (@c_eq R)
  reflexivity proved by (c_refl R)
  symmetry proved by (@c_sym R)
  transitivity proved by (@c_trans R)
  as Contract_setoid.

Add Parametric Morphism R : Par with
signature (c_eq R) ==> (c_eq R) ==> (c_eq R) as c_eq_par_morphism.
Proof.
intros. eauto with eqDB.
Qed.

Add Parametric Morphism R : CPlus with
signature (c_eq R) ==> (c_eq R) ==> (c_eq R) as c_eq_plus_morphism.
Proof.
intros. eauto with eqDB.
Qed.

Add Parametric Morphism R : CSeq with
signature (c_eq R) ==> (c_eq R) ==> (c_eq R) as co_eq_seq_morphism.
Proof.
```

```
intros. eauto with eqDB.
Qed.

Add Parametric Morphism R : Star with
signature (c_eq R) ==> (c_eq R) as c_eq_star_morphism.
Proof.
intros. eauto with eqDB.
Qed.




Lemma c_plus_neut_l : forall R c, Failure _+_ c =(R)= c.
Proof. intros. rewrite c_plus_comm. auto with eqDB.
Qed.

Lemma c_par_neut_l : forall R c, Success _*_ c =(R)= c.
Proof. intros. rewrite c_par_comm. auto with eqDB.
Qed.

Lemma c_par_failure_l  : forall R c, Failure _*_ c =(R)= Failure.
Proof. intros. rewrite c_par_comm. auto with eqDB.
Qed.

Lemma c_par_distr_r : forall R c0 c1 c2,
(c0 _+_ c1) _*_ c2 =(R)= (c0 _*_ c2) _+_ (c1 _*_ c2).
Proof.
intros. rewrite c_par_comm. rewrite c_par_distr_l. auto with eqDB.
Qed.

Hint Rewrite c_plus_neut_l
             c_plus_neut
             c_seq_neut_l
             c_seq_neut_r
             c_seq_failure_l
             c_seq_failure_r
             c_distr_l
             c_distr_r
             c_par_neut_l
             c_par_failure_l c_par_distr_r c_par_event
             c_par_neut c_par_failure c_par_distr_l : eqDB.

Ltac auto_rwd_eqDB := autorewrite with eqDB;auto with eqDB.

Definition co_eq c0 c1 := paco2 c_eq bot2 c0 c1.

Notation "c0 =C= c1" := (co_eq c0 c1)(at level 63).

Lemma c_eq_gen_mon: monotone2 c_eq.
Proof.
unfold monotone2.
intros. induction IN; eauto with eqDB.
Qed.
Hint Resolve c_eq_gen_mon : paco.
```

```
Ltac eq_m_left := repeat rewrite c_plus_assoc; apply c_plus_ctx;
                  auto_rwd_eqDB.

Ltac eq_m_right := repeat rewrite <- c_plus_assoc; apply c_plus_ctx;
                  auto_rwd_eqDB.

Lemma Σe_not_nu : forall es l0, nu (Σe es l0) = false.
Proof.
unfold Σe.
intros. induction ((combine es l0)).
- simpl. auto.
- simpl. rewrite IHl. auto.
Qed.

Ltac finish H := simpl; right; apply H; pfold; auto_rwd_eqDB.

Require Import Coq.btauto.Btauto.
Lemma c_eq_nu : forall R (c0 c1 : Contract) , c0 =(R)= c1 -> nu c0 = nu c1.
Proof.
intros. induction H; simpl; auto with bool; try btauto.
all : try (rewrite IHc_eq1; rewrite IHc_eq2; auto).
repeat rewrite Σe_not_nu. auto.
Qed.


Lemma co_eq_nu : forall (c0 c1 : Contract) , c0 =C= c1 -> nu c0 = nu c1.
Proof.
intros. eapply c_eq_nu. punfold H.
Qed.


Lemma Σederive_eq : forall es ps R e,
(forall p : Contract * Contract, In p ps ->
   (fst p) =(R)= (snd p)) -> e \ (Σe es (map fst ps)) =(R)=
                             e \ (Σe es (map snd ps)).
Proof.
induction es;intros;unfold Σe.
- simpl. reflexivity.
- simpl. destruct ps.
  * simpl. reflexivity.
  * simpl. eq_event_destruct;subst.
    ** auto_rwd_eqDB. rewrite H; auto using in_eq.
       eq_m_left. unfold Σe in IHes. apply IHes.
       intros. apply H. simpl. right. auto.
    ** auto_rwd_eqDB.  unfold Σe in IHes. apply IHes.
       intros. apply H. simpl. right. auto.
Qed.

Lemma co_eq_derive : forall (c0 c1 : Contract)
e, c0 =C= c1 -> e \ c0 =C= e \ c1.
Proof.
intros. pfold. punfold H.
induction H; try solve [ simpl; auto_rwd_eqDB] .
```

```
- simpl. destruct (nu c0) eqn:Heqn;simpl.
    ** destruct (nu c1).
      *** auto_rwd_eqDB. repeat rewrite <- c_plus_assoc.
          auto with eqDB.
      *** auto_rwd_eqDB.
    ** auto_rwd_eqDB.
- simpl;destruct (nu c);auto_rwd_eqDB.
- simpl. destruct (nu c); auto_rwd_eqDB.
- simpl. destruct (nu c0); auto_rwd_eqDB.
    eq_m_left. eq_m_right.
- simpl.
  destruct (nu c0); destruct (nu c1);simpl; auto_rwd_eqDB;
  repeat rewrite c_plus_assoc ; rewrite (c_plus_comm _ (e \ c2)).
  eq_m_left.  auto_rwd_eqDB.
- simpl. auto_rwd_eqDB. eq_m_right.
- simpl. rewrite c_plus_comm. eq_m_right.
- simpl. auto_rwd_eqDB. eq_m_left. eq_m_right.
- simpl. auto_rwd_eqDB. eq_event_destruct;subst;auto_rwd_eqDB.
- simpl. auto_rwd_eqDB. destruct (nu c);auto_rwd_eqDB.
- eauto with eqDB.
- simpl. destruct (nu c0) eqn:Heqn; destruct (nu c0') eqn:Heqn2;simpl.
  rewrite IHc_eq1. rewrite IHc_eq2.
  rewrite H0. reflexivity. apply c_eq_nu in H.
   rewrite Heqn in H. rewrite Heqn2 in H. discriminate.
   apply c_eq_nu in H.
   rewrite Heqn in H. rewrite Heqn2 in H. discriminate.
  rewrite IHc_eq1. rewrite H0. reflexivity.
- apply Σederive_eq;auto. intros. apply H in H0.
  pclearbot. punfold H0.
Qed.

Lemma bisim_soundness : forall (c0 c1 : Contract),
c0 =C= c1 -> Bisimilarity c0 c1.
Proof.
pcofix CIH.
intros. pfold. constructor.
- intros. right. apply CIH.  apply co_eq_derive. auto.
- auto using co_eq_nu.
Qed.


(**************Completeness***********)


Definition o c := if nu c then Success else Failure.
Transparent o.
Lemma o_plus : forall c0 c1 R, o (c0 _+_ c1) =(R)= o c0 _+_ o c1.
Proof.
unfold o. intros. simpl.
destruct (nu c0);destruct (nu c1);simpl;auto_rwd_eqDB.
Qed.

Lemma o_seq : forall c0 c1 R, o (c0 _;_ c1) =(R)= o c0 _;_ o c1.
Proof.
```

```
unfold o. intros. simpl.
destruct (nu c0);destruct (nu c1);simpl;auto_rwd_eqDB.
Qed.

Lemma o_par : forall c0 c1 R, o (c0 _*_ c1) =(R)= o c0 _*_ o c1.
Proof.
unfold o. intros. simpl.
destruct (nu c0);destruct (nu c1);simpl;auto_rwd_eqDB.
Qed.


Lemma o_true : forall c, nu c = true -> o c = Success.
Proof.
intros. unfold o.
destruct (nu c);auto. discriminate.
Qed.

Lemma o_false : forall c, nu c = false -> o c = Failure.
Proof.
intros. unfold o.
destruct (nu c);auto. discriminate.
Qed.

Lemma o_destruct : forall c, o c = Success \/ o c = Failure.
Proof.
intros. unfold o.
destruct (nu c);auto || auto.
Qed.

Hint Rewrite o_plus o_seq o_par : eqDB.

Hint Rewrite o_true o_false : oDB.


(***************New*****************)


Lemma Σ_alphabet_or : forall R alphabet0 e,
 Σ alphabet0
   (fun a : CSLC.EventType => if EventType_eq_dec e a
                               then Success else Failure)
   =(R)= Success /\ In e alphabet0 \/
 Σ alphabet0
   (fun a : CSLC.EventType => if EventType_eq_dec e a
                               then Success else Failure)
   =(R)= Failure /\ ~(In e alphabet0).
Proof.
induction alphabet0;intros.
- simpl. now right.
- simpl. eq_event_destruct.
  * subst. edestruct IHalphabet0.
    ** destruct H. left. split.
       rewrite H. auto_rwd_eqDB. now left.
    ** destruct H. rewrite H.
```

```
       auto_rwd_eqDB.
  * edestruct IHalphabet0; destruct H; rewrite H; auto_rwd_eqDB.
    right. split;auto with eqDB. intro H2. destruct H2.
    symmetry in H1. contradiction. contradiction.
Qed.

(************Summation rules used in showing
             normalization respects axiomatization*****)
Lemma Σ_alphabet : forall e R,
Σ alphabet (fun a => if EventType_eq_dec e a
                        then Success else Failure) =(R)= Success.
Proof.
intros. destruct (Σ_alphabet_or R alphabet e).
- destruct H. auto.
- destruct H. pose proof (in_alphabet e). contradiction.
Qed.

Definition fun_eq R (f0 f1 : EventType -> Contract) :=
 (forall a, f0 a =(R)= f1 a).

Add Parametric Morphism R l : (Σ l)  with
signature (@fun_eq R)  ==> (@ c_eq R) as c_eq_Σ_morphism.
Proof.
induction l;intros; simpl; auto with eqDB.
Qed.


Notation "f0 =(!$\lambda$! R )= f1" := (fun_eq R f0 f1)(at level 63).

Lemma fun_eq_refl : forall R f, f =(λ R)= f.
Proof.
intros. unfold fun_eq. auto with eqDB.
Qed.

Lemma fun_eq_sym : forall R f0 f1,f0 =(λ R)= f1 -> f1 =(λ R)= f0.
Proof.
intros. unfold fun_eq. auto with eqDB.
Qed.

Lemma fun_eq_trans : forall R f0 f1 f2,
f0 =(λ R)= f1 -> f1 =(λ R)= f2 -> f0 =(λ R)= f2.
Proof.
intros. unfold fun_eq. eauto with eqDB.
Qed.

Add Parametric Relation R : (EventType -> Contract) (@fun_eq R)
  reflexivity proved by (@fun_eq_refl R)
  symmetry proved by (@fun_eq_sym R)
  transitivity proved by (@fun_eq_trans R)
  as fun_Contract_setoid.



Lemma seq_derive_o : forall R e c0 c1,
```

```
e \ (c0 _;_ c1) = (R) = e \ c0 _;_ c1 _+_ o (c0) _;_ e \ c1.
Proof.
intros;simpl.  destruct (nu c0) eqn:Heqn.
- destruct (o_destruct c0). rewrite H. auto_rwd_eqDB.
  unfold o in H. rewrite Heqn in H. discriminate.
- destruct (o_destruct c0). unfold o in H.
  rewrite Heqn in H. discriminate.
  rewrite H. auto_rwd_eqDB.
Qed.

Lemma seq_derive_o_fun : forall R c0 c1,
(fun e0 => e0 \ (c0 _;_ c1)) =(λ R)=
(fun e0 => e0 \ c0 _;_ c1 _+_ o (c0) _;_ e0 \ c1).
Proof.
intros. unfold fun_eq. pose proof seq_derive_o. simpl in *. auto.
Qed.


Hint Rewrite seq_derive_o_fun : funDB.

Definition seq_fun (f0 f1 : EventType -> Contract) :=
  fun a => f0 a _;_ f1 a.
Notation "f0 !$\lambda$!;!$\lambda$! f1" := (seq_fun f0 f1)(at level 59).

Lemma to_seq_fun : forall R f0 f1,
(fun a => f0 a _;_ f1 a) =(λ R)= f0 λ;λ f1.
Proof.
intros. unfold seq_fun. reflexivity.
Qed.

Opaque seq_fun.

Add Parametric Morphism R : (seq_fun) with
signature (@fun_eq R) ==> (@fun_eq R) ==> (@fun_eq R)
  as fun_eq_seq_morphism.
Proof.
intros. repeat rewrite <- to_seq_fun.
unfold fun_eq in *. intros. auto with eqDB.
Qed.


Definition plus_fun (f0 f1 : EventType -> Contract) :=
 fun a => f0 a _+_ f1 a.

Notation "f0 !$\lambda$!+!$\lambda$! f1" := (plus_fun f0 f1)(at level 61).
Lemma to_plus_fun : forall R f0 f1,
(fun a => f0 a _+_ f1 a) =(λ R)= f0 λ+λ f1.
Proof.
intros. unfold plus_fun. reflexivity.
Qed.

Opaque plus_fun.

Add Parametric Morphism R : (plus_fun) with
```

```
signature (@fun_eq R) ==> (@fun_eq R) ==> (@fun_eq R)
 as fun_eq_plus_morphism.
Proof.
intros. repeat rewrite <- to_plus_fun.
unfold fun_eq in *. intros. auto with eqDB.
Qed.


Definition par_fun (f0 f1 : EventType -> Contract) :=
 fun a => f0 a _*_ f1 a.

Notation "f0 !$\lambda$!||!$\lambda$! f1" := (par_fun f0 f1)(at level 60).
Lemma to_par_fun : forall R f0 f1,
(fun a => f0 a _*_ f1 a) =(λ R)= f0 λ||λ f1.
Proof.
intros. unfold par_fun. reflexivity.
Qed.

Opaque plus_fun.

Add Parametric Morphism R : (par_fun) with
signature (@fun_eq R) ==> (@fun_eq R) ==> (@fun_eq R)
 as fun_eq_par_morphism.
Proof.
intros. repeat rewrite <- to_par_fun.
unfold fun_eq in *. intros. auto with eqDB.
Qed.

Hint Rewrite to_seq_fun to_plus_fun to_par_fun : funDB.



Lemma Σ_split_plus : forall R (A: Type) l (P P' : A -> Contract),
Σ l (fun a : A => P a _+_ P' a) = (R) =
Σ l (fun a : A => P a) _+_  Σ l (fun a : A => P' a).
Proof.
intros.
induction l;intros.
- simpl. auto_rwd_eqDB.
- simpl. rewrite IHl. eq_m_left. rewrite c_plus_comm. eq_m_left.
Qed.


Lemma Σ_factor_seq_r : forall R l (P: EventType -> Contract) c,
Σ l (fun a  => P a _;_ c) = (R) = Σ l (fun a  => P a) _;_ c.
Proof.
induction l;intros.
- simpl. auto_rwd_eqDB.
- simpl. auto_rwd_eqDB.
Qed.

Lemma Σ_factor_seq_l : forall R l (P: EventType -> Contract) c,
Σ l (fun a  => c _;_ P a) = (R) = c _;_ Σ l (fun a  => P a).
Proof.
```

```
induction l;intros.
- simpl. auto_rwd_eqDB.
- simpl. auto_rwd_eqDB.
Qed.


Lemma Σ_factor_par_l : forall R l1 c (P' : EventType -> Contract),
Σ l1 (fun a' : EventType => c _*_ P' a') = (R) =
c _*_ Σ l1 (fun a0 : EventType => P' a0).
Proof.
induction l1;intros.
- simpl. auto_rwd_eqDB.
- simpl. rewrite IHl1. auto_rwd_eqDB.
Qed.

Lemma Σ_factor_par_r : forall R l1 c (P' : EventType -> Contract),
Σ l1 (fun a0 : EventType => P' a0) _*_ c = (R) =
Σ l1 (fun a' : EventType => P' a' _*_ c).
Proof.
induction l1;intros.
- simpl. auto_rwd_eqDB.
- simpl. rewrite <- IHl1. auto_rwd_eqDB.
Qed.

Lemma Σ_par_ΣΣ : forall R l0 l1 (P0 P1 : EventType -> Contract),
Σ l0 (fun a0  => P0 a0) _*_ Σ l1 (fun a1 => P1 a1) = (R) =
Σ l0 (fun a0  => Σ l1 (fun a1  => (P0 a0) _*_ (P1 a1))).
Proof.
induction l0;intros.
- simpl. auto_rwd_eqDB.
- simpl. auto_rwd_eqDB.
  rewrite Σ_factor_par_l. rewrite IHl0.  reflexivity.
Qed.


Lemma ΣΣ_prod_swap : forall R l0 l1
(P : EventType -> EventType -> Contract),
Σ l0 (fun a0 => Σ l1 (fun a1 => P a0 a1)) = (R) =
Σ l1 (fun a0 => Σ l0 (fun a1 => P a1 a0)).
Proof.
induction l0;intros.
- simpl. induction l1;intros;simpl;auto with eqDB.
  rewrite IHl1. auto with eqDB.
- simpl. rewrite Σ_split_plus. eq_m_left.
Qed.

Lemma fold_Failure : forall R l,
Σ l (fun _ : EventType => Failure) = (R) = Failure.
Proof.
induction l;intros. simpl. reflexivity.
simpl. rewrite IHl. autorewrite with eqDB. reflexivity.
Qed.

Hint Resolve fold_Failure : eqDB.
```

```
(*Duplicate some of the rules to the function level*)

Lemma Σ_plus_decomp_fun : forall R l f0 f1,
Σ l (f0 λ+λ f1) = (R) = Σ l f0 _+_  Σ l f1.
Proof.
intros. rewrite <- to_plus_fun. apply Σ_split_plus.
Qed.

Lemma Σ_factor_seq_l_fun : forall R l f c,
Σ l ((fun _ => c ) λ;λ f) = (R) = c _;_ Σ l f.
Proof.
intros. rewrite <- to_seq_fun. apply Σ_factor_seq_l.
Qed.

Lemma Σ_factor_seq_r_fun : forall R l f0 c,
Σ l (f0 λ;λ (fun _ => c )) = (R) = Σ l f0 _;_ c.
Proof.
intros. rewrite <- to_seq_fun. apply Σ_factor_seq_r.
Qed.


(*Rules for rewriting functions*)
Lemma Σ_distr_l_fun : forall R f0 f1 f2,
f0  λ;λ (f1 λ+λ f2) =(λ R)= f0 λ;λ f1 λ+λ f0 λ;λ f2.
Proof.
intros. rewrite <- to_plus_fun. rewrite <- to_seq_fun.
symmetry. repeat rewrite <- to_seq_fun. rewrite <- to_plus_fun.
unfold fun_eq. intros. auto_rwd_eqDB.
Qed.


Lemma Σ_distr_par_l_fun : forall R f0 f1 f2,
f0 λ||λ (f1 λ+λ f2) =(λ R)= f0 λ||λ f1 λ+λ f0 λ||λ f2.
Proof.
intros. rewrite <- to_plus_fun. repeat rewrite <- to_par_fun.
rewrite <- to_plus_fun. unfold fun_eq.  auto with eqDB.
Qed.

Lemma Σ_distr_par_r_fun : forall R f0 f1 f2,
(f0 λ+λ f1)  λ||λ f2 =(λ R)= f0 λ||λ f2 λ+λ f1 λ||λ f2.
Proof.
intros. rewrite <- to_plus_fun. repeat rewrite <- to_par_fun.
rewrite <- to_plus_fun. unfold fun_eq. intros.  rewrite c_par_distr_r. reflexivity.
Qed.



Lemma Σ_seq_assoc_left_fun : forall R f0 f1 f2 ,
f0 λ;λ (f1 λ;λ f2) =(λ R)= (f0 λ;λ f1) λ;λ f2.
Proof.
intros. symmetry. rewrite <- (to_seq_fun _ f0). rewrite <- to_seq_fun.
rewrite <- (to_seq_fun _ f1). rewrite <- to_seq_fun. unfold fun_eq.
```

```
auto with eqDB.
Qed.

Lemma Σ_seq_assoc_right_fun : forall R f0 f1 f2 ,
(f0 λ;λ f1) λ;λ f2 =(λ R)= f0 λ;λ (f1 λ;λ f2).
Proof.
intros. symmetry. apply Σ_seq_assoc_left_fun.
Qed.

Lemma o_seq_comm_fun : forall R c f,
(f λ;λ (fun _ : EventType => o c)) =(λ R)=
(fun _ : EventType => o c) λ;λ f.
Proof.
intros. repeat rewrite <- to_seq_fun. unfold fun_eq.
intros. destruct (o_destruct c); rewrite H; auto_rwd_eqDB.
Qed.



Hint Rewrite Σ_distr_l_fun Σ_plus_decomp_fun Σ_factor_seq_l_fun
             Σ_factor_seq_r_fun Σ_seq_assoc_left_fun
             Σ_distr_par_l_fun Σ_distr_par_r_fun : funDB.


Lemma derive_unfold_seq : forall R c1 c2,
o c1 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c1) = (R) = c1 ->
o c2 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c2) = (R) = c2 ->
o (c1 _;_ c2) _+_
Σ alphabet (fun a : EventType => Event a _;_ a \ (c1 _;_ c2)) = (R) =
c1 _;_ c2.
Proof.
intros. rewrite <- H at 2. rewrite <- H0 at 2. autorewrite with funDB eqDB.
repeat rewrite c_plus_assoc; apply c_plus_ctx;
                  auto_rwd_eqDB.
rewrite o_seq_comm_fun.
autorewrite with funDB. rewrite Σ_seq_assoc_right_fun.
rewrite Σ_factor_seq_l_fun.
rewrite <- H0 at 1. autorewrite with eqDB funDB.
rewrite c_plus_assoc.
rewrite (c_plus_comm _ (Σ _ _ _;_  Σ _ _)).
eq_m_right.
Qed.

Lemma rewrite_in_fun : forall R f0 f1,
f0 =(λ R)= f1 -> (fun a => f0 a) =(λ R)= (fun a => f1 a).
Proof.
intros. unfold fun_eq in*. auto.
Qed.

Lemma rewrite_c_in_fun : forall R (c c' : Contract) ,
c = (R) = c' -> (fun _ : EventType => c) =(λ R)= (fun _ : EventType => c').
Proof.
intros. unfold fun_eq. intros. auto.
Qed.
```

```
Lemma fun_neut_r : forall R f, f λ||λ (fun _ => Success) =(λ R)= f.
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Lemma fun_neut_l : forall R f, (fun _ => Success) λ||λ f =(λ R)= f.
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Lemma fun_Failure_r : forall R f,
f λ||λ (fun _ => Failure) =(λ R)= (fun _ => Failure).
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Lemma fun_Failure_l : forall R f,
(fun _ => Failure) λ||λ f =(λ R)= (fun _ => Failure).
Proof.
intros. rewrite <- to_par_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.

Hint Rewrite fun_neut_r fun_neut_l fun_Failure_r fun_Failure_l : funDB.

Lemma o_seq_comm_fun3: forall R c1 c2,
Σ alphabet (Event λ;λ ((fun a : EventType => a \ c1) λ||λ
           (fun _ : EventType => o c2)))
= (R) =
Σ alphabet (Event λ;λ ((fun a : EventType => a \ c1))) _*_ o c2.
Proof.
intros. destruct (o_destruct c2);
rewrite H; autorewrite with funDB eqDB; reflexivity.
Qed.

Lemma o_seq_comm_fun4: forall R c1 c2,
Σ alphabet (Event λ;λ ((fun _ : EventType => o c1) λ||λ
           (fun a : EventType => a \ c2)))
= (R) =
o c1 _*_ Σ alphabet (Event λ;λ (fun a : EventType => a \ c2)).
Proof.
intros. destruct (o_destruct c1);
rewrite H; autorewrite with funDB eqDB; reflexivity.
Qed.

Hint Rewrite to_seq_fun to_plus_fun to_par_fun : funDB.

Definition app a (f : EventType -> Contract) := f a.

Lemma to_app : forall f a, f a = app a f.
```

```
Proof.
intros. unfold app. reflexivity.
Qed.

Opaque app.

Add Parametric Morphism R a : (app a) with
signature (@fun_eq R) ==>  (@c_eq R) as afun_eq_par_morphism.
Proof.
intros. repeat rewrite <- to_app. unfold fun_eq in *. intros. auto with eqDB.
Qed.

Lemma o_seq_comm_fun_fun : forall R c1 c2 a,
(fun a1 : EventType => (Event λ;λ (fun a0 : EventType => a0 \ c1)) a
                         _*_ (Event λ;λ (fun a0 : EventType => a0 \ c2)) a1)
=(λ R)=
(fun a1 : EventType => (Event a _;_ a \ c1) _*_ Event a1 _;_ a1 \ c2).
Proof.
intros. unfold fun_eq. intros. repeat rewrite to_app.
repeat rewrite <- to_seq_fun.
apply c_par_ctx; now rewrite <- to_app.
Qed.

Lemma o_seq_comm_fun_fun2 : forall R c1 c2 a,
(fun a1 : EventType => (Event a _;_ a \ c1) _*_ Event a1 _;_ a1 \ c2)
=(λ R)=
(fun a1 : EventType => (Event a _;_ (a \ c1 _*_ Event a1 _;_ a1 \ c2)))
λ+λ (fun a1 => Event a1 _;_ (Event a _;_ a \ c1 _*_ a1 \ c2)).
Proof.
intros. rewrite <- to_plus_fun. unfold fun_eq. intros.
auto_rwd_eqDB.
Qed.




Lemma derive_unfold_par : forall R c1 c2,
o c1 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c1) = (R) = c1 ->
o c2 _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c2) = (R) = c2 ->
o (c1 _*_ c2) _+_
Σ alphabet (fun a : EventType => Event a _;_ a \ (c1 _*_ c2)) = (R) =
c1 _*_ c2.
Proof.
intros;simpl.
rewrite <- H at 2. rewrite <- H0 at 2.
rewrite to_seq_fun in *. autorewrite with funDB eqDB.
eq_m_left.
rewrite <- (rewrite_c_in_fun H). rewrite <- (rewrite_c_in_fun H0).
autorewrite with funDB eqDB.
rewrite o_seq_comm_fun3.
rewrite o_seq_comm_fun4.
repeat rewrite <- c_plus_assoc.
eq_m_left.
```

```
rewrite c_plus_comm.
eq_m_left. rewrite Σ_par_ΣΣ.
symmetry.
rewrite rewrite_in_fun.
2: { unfold fun_eq. intros. rewrite o_seq_comm_fun_fun.
    rewrite o_seq_comm_fun_fun2.
    rewrite Σ_plus_decomp_fun at 1. eapply c_refl. }
rewrite Σ_split_plus. rewrite c_plus_comm.
apply c_plus_ctx.
- rewrite ΣΣ_prod_swap. apply c_eq_Σ_morphism.

rewrite <- to_par_fun.
repeat rewrite <- to_seq_fun.
unfold fun_eq. intros.
rewrite Σ_factor_seq_l. apply c_seq_ctx. reflexivity.
repeat rewrite <- Σ_factor_par_r.
apply c_par_ctx; auto with eqDB.
- apply c_eq_Σ_morphism. rewrite <- to_par_fun.
repeat rewrite <- to_seq_fun. unfold fun_eq. intros.
rewrite Σ_factor_seq_l. apply c_seq_ctx;auto with eqDB.
repeat rewrite Σ_factor_par_l.
apply c_par_ctx; auto with eqDB.
Qed.

Lemma derive_unfold : forall R c,
o c _+_ Σ alphabet (fun a : EventType => Event a _;_ a \ c) = (R) = c.
Proof.
induction c;intros.
- unfold o; simpl. autorewrite with funDB eqDB. reflexivity.
- unfold o. simpl. autorewrite with funDB eqDB. reflexivity.
- unfold o;simpl. autorewrite with funDB eqDB.
  rewrite rewrite_in_fun.
  2: { instantiate (1:= (fun _ => Event e) λ;λ
                          (fun a : EventType => if EventType_eq_dec e a
                                                 then Success else Failure)).
  repeat rewrite <- to_seq_fun. unfold fun_eq. intros. eq_event_destruct.
  subst. reflexivity. auto_rwd_eqDB. }
  rewrite Σ_factor_seq_l_fun. rewrite Σ_alphabet. auto_rwd_eqDB.
- simpl;auto_rwd_eqDB.
  rewrite <- IHc1 at 2. rewrite <- IHc2 at 2. autorewrite with funDB eqDB.
  repeat rewrite <- c_plus_assoc. eq_m_right. eq_m_left.
- auto using derive_unfold_seq.
- auto using derive_unfold_par.
- unfold o. simpl. rewrite <- IHc. autorewrite with funDB eqDB.
  rewrite <- IHc at 1.
  destruct (o_destruct c);rewrite H in *.
  * repeat rewrite c_star_plus. apply c_unfold.
  * auto_rwd_eqDB.
Qed.



Lemma Σd_to_Σe : forall c es,
Σ es (fun a : EventType => Event a _;_ a \ c) =
```

```
Σe es (map (fun e => e \ c) es).
Proof.
induction es;intros;simpl;auto.
unfold Σe in *. simpl. rewrite IHes. auto.
Qed.

Lemma map_fst_combine : forall (A: Type)(l0 l1 : list A),
length l0 = length l1 -> map fst (combine l0 l1) = l0.
Proof.
induction l0;intros;simpl;auto.
destruct l1 eqn:Heqn. simpl in H. discriminate.
simpl. f_equal. rewrite IHl0;auto.
Qed.

Lemma map_snd_combine : forall (A: Type)(l0 l1 : list A),
length l0 = length l1 -> map snd (combine l0 l1) = l1.
Proof.
induction l0;intros;simpl;auto.
- destruct l1. auto. simpl in H. discriminate.
- destruct l1.  simpl in H.  discriminate.
  simpl. f_equal. auto.
Qed.


Lemma Σe_to_pair : forall R es l0 l1, length l0 = length l1 ->
Σe es (map fst (combine l0 l1)) = (R) = Σe es (map snd (combine l0 l1)) ->
Σe es l0 = (R) = Σe es l1.
Proof.
intros. rewrite map_fst_combine in H0; auto.
rewrite  map_snd_combine in H0;auto.
Qed.

Lemma combine_map : forall (A B : Type) (l : list A) (f f' : A -> B),
combine (map f l) (map f' l) = map (fun c => (f c, f' c)) l.
Proof.
induction l;intros.
- simpl. auto.
- simpl. rewrite IHl. auto.
Qed.

Ltac sum_reshape := repeat rewrite Σd_to_Σe; apply Σe_to_pair;
                    repeat rewrite map_length; auto.

Lemma if_nu : forall R (b0 b1 : bool), b0 = b1 ->
(if b0 then Success else Failure) = (R) =
(if b1 then Success else Failure).
Proof.
intros. rewrite H. reflexivity.
Qed.

Ltac unfold_tac :=
 match goal with
         | [ |-  ?c0 = (_) = ?c1  ] =>
            rewrite <- (derive_unfold _ c0) at 1;
```

```
            rewrite <- (derive_unfold _ c1) at 1;
            unfold o; eq_m_left; try solve [apply if_nu; simpl; btauto]
         end.

Ltac simp_premise :=
 match goal with
         | [ H: In ?p (combine (map _ _) (map _ _)) |- _ ] =>
             destruct p; rewrite combine_map in H;
             rewrite in_map_iff in *;
             destruct_ctx;simpl;inversion H;subst;clear H
         end.

Lemma bisim_completeness : forall c0 c1,
Bisimilarity c0 c1 -> c0 =C= c1.
Proof.
pcofix CIH.
intros. punfold H0. inversion H0.
pfold.
unfold_tac.
- rewrite H2. reflexivity.
- sum_reshape.
  apply c_co_sum. intros.
  simp_premise.
  right. apply CIH.
  pclearbot.
  unfold Bisimilarity. auto.
Qed.


Theorem soundness : forall c0 c1,
c0 =C= c1 -> (forall s, s(:)c0 <-> s(:)c1).
Proof.
intros c0 c1 H. rewrite matches_eq_iff_bisimilarity. auto using bisim_soundness.
Qed.

Theorem completeness : forall c0 c1,
(forall s, s(:)c0 <-> s(:)c1) -> c0 =C= c1.
Proof.
intros. apply bisim_completeness. rewrite <- matches_eq_iff_bisimilarity. auto.
Qed.



Lemma test : forall c, Star c =C= Star (Star c).
Proof.
intros.
pfold.
unfold_tac.
sum_reshape.
```

```
apply c_co_sum. intros.
simp_premise.
left.

pfold.
rewrite c_seq_assoc. apply c_seq_ctx. reflexivity. (*match first sequence*)
unfold_tac.
sum_reshape.
apply c_co_sum. intros.
simp_premise.
left.

generalize x0. pcofix CIH2. intros. (*Coinduction principle*)
pfold.
rewrite c_plus_idemp.
rewrite c_seq_assoc. apply c_seq_ctx. reflexivity.
unfold_tac.
sum_reshape.
apply c_co_sum. intros.
simp_premise.
right. apply CIH2.
Qed.
```