Citation for published version


DOI

Link to record in KAR

http://kar.kent.ac.uk/13695/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact: researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
ArcAngel: a Tactic Language For Refinement and its Tool Support
by
Marcel Vinicius Medeiros Oliveira

MSc. Thesis

Recife, 4th December, 2002
ArcAngel: a Tactic Language For Refinement and its Tool Support

This work was presented to the Msc programme in Computer Science, Centro de Informática, Universidade Federal de Pernambuco as a requirement for the award of the degree.

Supervisor: Profa. Ana Lúcia Caneca Cavalcanti
Acknowledgements

Thanks God, for helping me and for making me able to realize this work. Thanks to my parents for growing me up and for investing in my studies. To Lauana for the support and the incentive in the most difficult times.

To my supervisor, Ana Cavalcanti, for guiding me in this research, for the valuable insights, for the detailed revisions and for supporting my work since graduation, trying always to transfer your knowledge to me.

To IPAD - Instituto de Planejamento e Apoio ao Desenvolvimento Tecnológico e Científico for the financial, technical and personal support.

To Adeline de Souza Silva for helping me with the implementation of Gabriel’s parser. To Adnan Sherif for giving me hints about \LaTeX{} and its tools for Windows.

To Lindsay Groves for very detailed comments and insights. To Andrew Martin, and Ray Nickson for valuable discussions about refinement, mechanization, and tactics. Jim Woodcock collaborated in the presentation of the semantics of ArcAngel. The name of the language itself is due to him.

To all my friends and to my family for being part of my life and for have being always for me and with me.

Thanks.
Abstract

Morgan’s refinement calculus is a successful technique to develop and implement software in a precise, complete, and consistent way. From a formal specification we produce a program which correctly implements the specification by repeatedly applying transformation rules, which are called refinement laws. Using the refinement calculus, however, can be a hard task, as program developments may prove to be long and repetitive.

Frequently used strategies of development are reflected in sequences of law applications that are over and over applied in different developments or even in different points of a single development. A lot is to be gained from identifying these tactics of development, documenting them, and using them in program developments as a single transformation rule.

In this work we present ArcAngel, a language for the definition of refinement tactics based on Angel, and formalize its semantics. Angel is a general-purpose tactic language that assumes only that rules transform proof goals. The semantics of ArcAngel is similar to Angel’s semantics, but is elaborated to take into account the particularities of the refinement calculus.

Most of Angel’s algebraic laws are not proved. In this work we present their proofs based on the ArcAngel semantics. A normal form is also presented in this work; it is similar to that presented for Angel tactics. Our contribution in this respect is to give more details on the proofs of the lemmas and theorems involved in the strategy of reduction to this normal form.

The constructs of ArcAngel are similar to those of Angel, but are adapted to deal with refinement laws and programs. Moreover, ArcAngel provides structural combinators that are suitable to apply refinement laws to components of programs. Using ArcAngel, we define refinement tactics that embody common development and programming strategies.

Finally, we present Gabriel, a tool support for ArcAngel. Gabriel works as a plug-in to Refine, a tool that semi-automatizes transformations from formal specifications to correct programs with successive refinement laws applications. Gabriel allows its users to create tactics and use them in a program development.
Resumo

O cálculo de refinamentos é uma técnica moderna para o desenvolvimento e implementação de programas de uma maneira precisa, completa e consistente. A partir de uma especificação formal, nós produzimos um programa que implementa corretamente a especificação através de repetidas aplicações de regras de transformação, também chamadas de leis de refinamento. Entretanto, o uso do cálculo de refinamentos pode ser uma tarefa difícil, pois o desenvolvimento de programas pode vir a ser longo e repetitivo.

Estratégias de desenvolvimento são refletidas em sequências de aplicações de leis que são aplicadas repetidamente em desenvolvimentos distintos, ou até mesmo, em pontos diferentes de um mesmo desenvolvimento. A identificação destas táticas de desenvolvimento, documentação, e uso das mesmas em desenvolvimentos de programas como uma simples regra de transformação trazem um grande ganho de tempo e esforço.

Neste trabalho nós apresentamos ArcAngel, uma linguagem para definição de táticas de refinamento baseada em Angel, e formalizamos a sua semântica. Angel é uma linguagem de táticas de propósito geral que assume apenas que regras transformam objetivos de prova. A semântica de ArcAngel é similar a de Angel, mas é elaborada de maneira a levar em consideração as particularidades do cálculo de refinamentos.

A maioria das leis algébricas de Angel não são provadas. Neste trabalho, nós apresentamos suas provas baseadas na semântica de ArcAngel. Também apresentamos neste trabalho uma forma normal; ela é similar àquela apresentada para táticas em Angel. Neste respeito, nossa contribuição é dar mais detalhes nas provas de lemas e teoremas envolvidos na estratégia de redução para esta forma normal.

Os construtores de ArcAngel são similares aos de Angel, mas são adaptados para tratar com leis de refinamento e programas. Além disso, ArcAngel provê combina- dores estruturais que são apropriados para aplicar leis de refinamento a componentes de programas. Usando ArcAngel, nós definimos táticas de refinamento que refletem estratégias comuns de desenvolvimento de programas.

Finalmente, nós apresentamos Gabriel, um suporte ferramental para ArcAngel. Gabriel trabalha como um componente de Refine, uma ferramenta que semi-automatiza transformações de especificações formais para programas corretos através de sucessivas aplicações de leis de refinamento. Gabriel permite aos seus usuários criar táticas e usá-las em desenvolvimento de programas.
# Contents

1 Introduction ........................................... 1  
  1.1 Motivation ........................................ 2  
  1.2 Refinement Calculus ............................... 3  
  1.3 Overview .......................................... 11  

2 ArcAngel .............................................. 13  
  2.1 ArcAngel’s Syntax ................................ 14  
     2.1.1 Basic Tactics ................................ 14  
     2.1.2 Tacticals ...................................... 14  
     2.1.3 Structural Combinators ....................... 17  
  2.2 Examples .......................................... 19  
     2.2.1 Following Assignment ......................... 20  
     2.2.2 Leading Assignment ............................ 21  
     2.2.3 TakeConjAsInv ................................ 22  
     2.2.4 ReplConsByVar ................................ 25  
     2.2.5 StrengInv ...................................... 28  
     2.2.6 TailInvariant .................................. 38  
     2.2.7 ProcNoArgs .................................. 43  
     2.2.8 ProcCalls ...................................... 45  
     2.2.9 ProcArgs ....................................... 45  
     2.2.10 RecProcArgs ................................ 47  
  2.3 ArcAngel’s Semantics .............................. 51  
     2.3.1 Tactics ........................................ 52  
     2.3.2 Structural Combinators ....................... 54  
     2.3.3 Tactic Declarations and Programs ............ 62  
     2.3.4 Tacticals ...................................... 63  

3 Algebraic Laws ....................................... 64  
  3.1 Simple Laws ....................................... 65  
     3.1.1 Basic properties of composition .............. 65  
     3.1.2 Laws involving Cut ........................... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>Laws involving <em>succs</em> and <em>fails</em></td>
<td>66</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Laws involving Structural Combinators</td>
<td>69</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Laws of <em>con</em></td>
<td>69</td>
</tr>
<tr>
<td>3.2</td>
<td><em>ArcAngel</em>’s Normal Form</td>
<td>70</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Cut-free Normal Form</td>
<td>70</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Pre-Normal Form</td>
<td>73</td>
</tr>
<tr>
<td>3.2.3</td>
<td>General Normal Form</td>
<td>77</td>
</tr>
<tr>
<td>4</td>
<td><em>Gabriel</em>: a tool for <em>ArcAngel</em></td>
<td>86</td>
</tr>
<tr>
<td>4.1</td>
<td><em>Gabriel</em> Concepts</td>
<td>87</td>
</tr>
<tr>
<td>4.2</td>
<td><em>Gabriel</em>’s User Interface</td>
<td>87</td>
</tr>
<tr>
<td>4.3</td>
<td>Constructs of <em>Gabriel</em></td>
<td>89</td>
</tr>
<tr>
<td>4.4</td>
<td>Using <em>Gabriel</em></td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>Conclusions</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Contributions and Conclusions</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Related Work</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>Future Work</td>
<td>100</td>
</tr>
<tr>
<td>A</td>
<td><em>ArcAngel</em>’s Novel Constructs</td>
<td>102</td>
</tr>
<tr>
<td>B</td>
<td>Infinite Lists</td>
<td>103</td>
</tr>
<tr>
<td>C</td>
<td>Refinement Laws</td>
<td>107</td>
</tr>
<tr>
<td>D</td>
<td>Proofs of the Laws</td>
<td>111</td>
</tr>
<tr>
<td>D.1</td>
<td>Basic properties of composition</td>
<td>111</td>
</tr>
<tr>
<td>D.2</td>
<td>Laws involving Cut</td>
<td>115</td>
</tr>
<tr>
<td>D.3</td>
<td>Laws involving <em>succs</em> and <em>fails</em></td>
<td>121</td>
</tr>
<tr>
<td>D.4</td>
<td>Laws involving Structural Combinators</td>
<td>156</td>
</tr>
<tr>
<td>D.5</td>
<td>Laws on <em>con</em></td>
<td>161</td>
</tr>
<tr>
<td>D.6</td>
<td>Lemmas</td>
<td>162</td>
</tr>
<tr>
<td>D.6.1</td>
<td>Lemma 3</td>
<td>162</td>
</tr>
<tr>
<td>D.6.2</td>
<td>Lemma 7</td>
<td>167</td>
</tr>
<tr>
<td>D.6.3</td>
<td>Lemma 8</td>
<td>168</td>
</tr>
<tr>
<td>D.6.4</td>
<td>Lemma 9</td>
<td>169</td>
</tr>
<tr>
<td>D.6.5</td>
<td>Lemma 10</td>
<td>169</td>
</tr>
<tr>
<td>D.6.6</td>
<td>Lemma 11</td>
<td>171</td>
</tr>
<tr>
<td>D.6.7</td>
<td>Lemma 12</td>
<td>172</td>
</tr>
<tr>
<td>D.6.8</td>
<td>Lemma 13</td>
<td>173</td>
</tr>
<tr>
<td>D.6.9</td>
<td>Lemma 14</td>
<td>174</td>
</tr>
<tr>
<td>D.6.10</td>
<td>Lemma 15a</td>
<td>182</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Abstract Syntax of the Language of Morgan’s Refinement Calculus 4
2.1 Abstract Syntax of ArcAngel ................................. 15
4.1 Gabriel’s User Interface ........................................ 88
4.2 Refine’s Tactics List ............................................. 89
4.3 takeConjAsInv written in ASCII ArcAngel ................. 90
4.4 Refine’s New Development Window .......................... 91
4.5 Gabriel’s Arguments Window ................................. 92
4.6 Refine’s Development Window ............................... 92
4.7 Refine’s Collected Code Window ............................ 93
A.1 ArcAngel’s Novel Constructs ................................. 102
E.1 Integration Gabriel – Refine ................................. 193
E.2 Tactic’s Hierarchy .............................................. 194
E.3 Tactic Generation .............................................. 195
E.4 Tactic Application .............................................. 196
## List of Tables

4.1 Argument’s Values .................................................. 93  
F.1 ArcAngel’s constructs in Gabriel ................................ 198  
F.2 Gabriel’s Laws Names ............................................. 199  
F.3 Continuation of Gabriel’s Laws Names ......................... 200  
F.4 Argument’s Types in Gabriel ..................................... 201
Chapter 1

Introduction

In this chapter we motivate the creation of a tactic language for the refinement calculus of Morgan. The motivation for implementing a tool that provides support for the use of this language is also presented. Furthermore, we give a brief introduction to the refinement calculus, and, finally, provide an overview of the whole dissertation.
1.1 Motivation

Software development should include theories and formalisms in order to become software engineering. Most of software developments, however, do not use the already existing theories. This leads to difficulties in developing a relatively low cost trustworthy software, where the time of development is controllable. In [22], Milner affirms that software development theories are as important as computing theories. The experience with the informal techniques is the main reason for using formal methods in the development processes.

The refinement calculus of Morgan [23] is a modern technique for formal program development. By repeatedly applying correctness-preserving transformation rules to an initial specification, we produce a program that implements it. However, applying the refinement calculus may be a hard task, since developments are often long and repetitive. Some development strategies may be captured as sequences of rule applications, and used in different developments, or even several times within a single development. Identifying these strategies [17], documenting them as refinement tactics, and using them as single transformation rules brings a profit in time and effort. Also, a notation for describing derivations can be used for modifying and analyzing formal derivations.

In this work we present a refinement-tactic language called ArcAngel, derived from the more general tactic language, Angel [21, 20]. The flexibility of Angel and its theoretical basis were the reasons for choosing Angel as our basis language. Angel is a general-purpose tactic language that is not tailored to any particular proof tool; it assumes only that rules transform proof goals. A refinement-tactic language must take into account the fact that, when applying refinement laws to a program, we get not only a program, but proof obligations as well. So, the result of applying a tactic is a program and a list of all the proof obligations generated by the individual law applications. We give an informal description of ArcAngel and use it to formalize commonly used refinement strategies. We also provide its formal semantics. Based on these definitions, we have proven over seventy laws of reasoning, which support a strategy of reduction to a normal form of ArcAngel. ArcAngel’s constructs are similar to Angel’s, but are adapted to deal with the application of refinement laws to programs. In particular, ArcAngel’s structural combinators are used to apply tactics to program components.

Other tactic languages can be found in the literature [11, 33, 34, 31, 32, 12, 13, 4, 35, 36, 6, 14]. However, as far as we know, none of them present a formal semantics and a normal form, as ArcAngel does. Furthermore, some of these languages do not present some operators (i.e. recursion and alternation) as ArcAngel does. This limits the power of expression of these languages.

The use of ArcAngel, without a tool support, would still be a hard task. Implementing such a tool would bring further profit in time and effort. We present
Gabriel, a tool support for ArcAngel. Gabriel works as a plug-in to Refine [8] and allows its users to create tactics and use them in a program development.

Some tools for refinement support use existing languages for tactic definition [11, 12, 13, 4, 35, 36, 6, 14]. For this reason, the user must learn complex languages in order to formalize tactics using these tools. Furthermore, some of them have a goal-oriented approach [14]. These tools use existing theorem provers. The refinement consists of proving that the final program implements the initial formal specification. This, however, is not a good approach for program refinement since we do not know the final program from the beginning of the refinement. Finally, some tools do not provide the reuse of previous derivations [31, 32] and do not support procedure constructs [4, 35, 36] as Refine does.

1.2 Refinement Calculus

In this dissertation we assume a previous knowledge of refinement calculus. However, a short description is given in order to establish the notation used in this work.

The refinement calculus is based on a unified language of specification, design and implementation. Its syntax is presented in Figure 1.1. Developing programs consists of repeatedly applying refinement laws to a specification until an adequate program is obtained. All these laws are listed in Appendix C.

A specification has the form \( w : [\text{pre}, \text{post}] \) where \( w \), the frame, lists the variables whose values may change, \( \text{pre} \) is the precondition, and \( \text{post} \) is the postcondition. The language used to define the precondition and the postcondition is the predicate calculus. The execution of a specification statement in a state that satisfies the precondition changes the variables listed in the frame so that the final state satisfies the postcondition. If the initial state does not satisfy the precondition, the result cannot be predicted. A precondition \( \text{true} \) can be omitted. For example, the specification statement \( x : [x > 0, x^2 = 1] \) specifies a program which, given \( x \) such that \( x > 0 \), sets \( x \) so that \( x^2 = 1 \).

In the postcondition, 0-subscripted variables can be used to represent the initial value of the corresponding variable. As an example of its use we have \( x : [x = x_0 + 1] \). After the execution of this program, the variable \( x \) has its value in the initial state incremented by one.

As a development example, we use the following specification statement:

\[
q, r : [a \geq 0 \land b > 0, q = a \div b \land r = a \mod b]
\]

It specifies a program which, given \( a \) and \( b \), such that \( a \geq 0 \) and \( b > 0 \), sets \( q \) to the quotient of \( a \) divided by \( b \), and sets \( r \) to the remaining of this quotient. The
program ::= name*: [predicate, predicate]                    [specification statement]
    | name*: := expression*                                      [multiple-assignment]
    | program; program                                           [sequence]
    | if [] i • predicate → program fi                          [conditional]
    | do [] i • predicate → program od                           [loop]
      | [ var varDec • program ]                                  [variable block]
      | [ con varDec • program ]                                  [constant block]
      | [ proc name ≡ procBody • program ]                        [procedure block]
      | [ proc name ≡ procBody variant name is expression • program ] [variant block]

procBody ::= program
            | (parDec • program)

parDec ::= val varDec
         | res varDec
         | val-res varDec
         | parDec; parDec

varDec ::= name+ : Type
          | varDec; varDec

Figure 1.1: Abstract Syntax of the Language of Morgan’s Refinement Calculus
symbol $\sqsubseteq$ represents the refinement relation such that

\[ a \sqsubseteq l\text{name}(\text{args}) \sqsubseteq b \]

represents that, using the law $\text{name}$ with arguments $\text{args}$, $b$ refines $a$.

We begin the development by strengthening the postcondition in order to insert the mathematical definition of quotient and of quotient’s remaining.

\[ q, r : [a \geq 0 \land b > 0, q = a \text{ div } b \land r = a \text{ mod } b] \]

\[ \sqsubseteq \text{strPost}((a = q \ast b + r \land 0 \leq r) \land \neg r \geq b) \]

\[ q, r : [a \geq 0 \land b > 0, (a = q \ast b + r \land 0 \leq r) \land \neg r \geq b] \]

This law generates the following proof obligation

\[ (a = q \ast b + r \land 0 \leq r) \land \neg r \geq b \Rightarrow (q = a \text{ div } b \land r = a \text{ mod } b). \]

Then, we strengthen again the postcondition in order to introduce the bound limits of the invariant $b > 0$.

\[ \sqsubseteq \text{strPost}(b > 0 \land a = q \ast b + r \land 0 \leq r \land \neg r \geq b) \]

\[ q, r : [a \geq 0 \land b > 0, b > 0 \land a = q \ast b + r \land 0 \leq r \land \neg r \geq b] \]

This law generates the following proof obligation

\[ (b > 0 \land a = q \ast b + r \land 0 \leq r \land \neg r \geq b) \Rightarrow (a = q \ast b + r \land 0 \leq r) \land \neg r \geq b. \]

We now split the specification into two. We intend to use the first one to initialize the variables used in the iteration generated from the second one.

\[ \sqsubseteq \text{seqComp}(b > 0 \land a = q \ast b + r \land 0 \leq r) \]

\[ q, r : [a \geq 0 \land b > 0, b > 0 \land a = q \ast b + r \land 0 \leq r]; \]

\[ q, r : \begin{cases} b > 0 \land a = q \ast b + r \land 0 \leq r \land r \geq b \end{cases} \]

\[ \sqsubseteq \text{assign}(q, r := 0, a) \]

\[ q, r := 0, a \]

This law generates the following proof obligation

\[ a \geq 0 \land b > 0 \Rightarrow b > 0 \land a = 0 \ast b + a \land 0 \leq a. \]

Then, we introduce the iteration.
\[(i) \subseteq \text{iter}(\langle r \geq b \rangle, r) \]
\[
\begin{array}{c}
\text{do}
\end{array}
\begin{array}{c}
r \geq b \\
q, r:
\begin{cases}
q > 0 \land a = q \ast b + r \\
0 \leq r \land r \geq b \\
b > 0 \land a = q \ast b + r \\
0 \leq r \land 0 \leq r < r_0
\end{cases} \end{array}
\begin{array}{c}
\od
\end{array}
\]

Finally, we introduce the assignment in the body of the iteration. The quotient is incremented by one, and the quotient’s remaining is decremented by \(b\).

\[
\subseteq \text{assignIV}(q, r := q + 1, r - b)
\]
\[
q, r := q + 1, r - b
\]

This law generates the following proof obligation

\[
(q = q_0) \land (r = r_0) \land b > 0 \land a = q \ast b + r \land 0 \leq r \land r \geq b \Rightarrow
b > 0 \land a = (q + 1) \ast b + (r - b) \land 0 \leq (r - b) \land 0 \leq (r - b) < r_0.
\]

So, by applying a sequence of refinement laws to the initial specification we get the following executable program.

\[
q, r := 0, a;
\begin{array}{c}
\text{do}
\end{array}
\begin{array}{c}
r \geq b \\
q, r := q + 1, r - b
\end{array}
\begin{array}{c}
\od
\end{array}
\]

Besides the specification statement, the language of the refinement calculus includes all the constructors of Dijkstra’s language [10]. There are also block constructs to declare local variables, logical constants, and procedures. Variable blocks have the form \([\text{var } x : T \bullet p]\), where \(x\) is a variable name of type \(T\) whose scope is restricted to \(p\). Similarly, logical constants \(c\) are declared in blocks of the form \([\text{con } c \bullet p]\).

Procedure blocks are very simple: they begin with the keyword \textproc, the name of the procedure and a program fragment, called the body of the procedure. This body can have argument declarations, if the procedure has arguments. Then, after a \(\bullet\), the main program is declared. The general form is \([\textproc name \equiv body \bullet prog]\).

As an example, we have the program which, given three numbers \(p\), \(q\), and \(r\), exchanges their values such that \(p \leq q \leq r\). The specification of this program is
presented below.

\[ p, q, r : [p \leq q \leq r \land [p, q, r] = [p_0, q_0, r_0]] \]

The expression \([p, q, r] = [p_0, q_0, r_0]\) assures that the bag containing the values of \(p, q,\) and \(r\) cannot be changed.

We start the derivation by breaking the program in three assignments. The last assignment sorts \(p\) and \(q\). The expression \(p \cap q\) returns \(q\) if \(p > q\) and \(p\) if \(p \leq q\). The expression \(p \cup q\) returns \(p\) if \(p > q\) and \(q\) if \(p \leq q\).

\[
\begin{align*}
& \text{assignIV}(p, q := (p \cap q), (p \cup q)) \\
& p, q, r : [(p \cap q) \leq (p \cup q) \leq r \land [(p \cap q), (p \cup q), r] = [p_0, q_0, r_0]] \\
& p, q := (p \cap q), (p \cup q)
\end{align*}
\]

The second assignment sorts \(q\) and \(r\). Finally, the first assignment also sorts \(p\) and \(q\).

\[
\begin{align*}
& \text{assignIV}(q, r := (q \cap r), (q \cup r)) \\
& p, q, r : [(p \cap (q \cap r)) \leq (p \cup (q \cap r)) \leq (q \cap r) \\
& [(p \cap (q \cap r)), (p \cup (q \cap r)), (q \cap r)] = [p_0, q_0, r_0]] \\
& q, r := (q \cap r), (q \cup r)
\end{align*}
\]

This law generates the following proof obligation.

\[
p = p_0 \land q = q_0 \land r = r_0 \Rightarrow \\
((p \cap q) \cap ((p \cap q) \cap r)) \leq ((p \cap q) \cup ((p \cap q) \cap r)) \leq ((p \cup q) \cup r) \\
[((p \cap q) \cap ((p \cap q) \cap r)), ((p \cap q) \cup ((p \cap q) \cap r)), ((p \cup q) \cup r)] = [p_0, q_0, r_0]
\]

Finally, we introduce the procedure \textit{sort} and, then, we use this procedure in the body of the program.

\[
\begin{align*}
& \text{procNoArgsIntro}(\textit{sort}, (p, q := p \cap q, p \cup q)) \\
& \begin{array}{l}
\text{proc } \textit{sort} \equiv p, q := p \cap q, p \cup q \\
\text{p, q := (p \cap q), (p \cup q)};
\text{q, r := (q \cap r), (q \cup r)};
\text{p, q := (p \cap q), (p \cup q)}
\end{array}
\end{align*}
\]

So, our derivation results in the following program. This program defines the
procedure \textit{sort} and use it to sort \( p, q, \) and \( r \).

\begin{verbatim}
[ [ \text{ proc }\text{ sort } \triangleq p, q := p \sqcap q, p \sqcup q \bullet 
  \text{ sort } ; q, r := (q \sqcap r), (q \sqcup r) ; \text{ sort } ] ]
\end{verbatim}

The arguments can be passed by value, using the keyword \texttt{val}, by result, using
the keyword \texttt{res}, or by value-result, using the keyword \texttt{val-res}. An example of
a parameterized procedure is \[ \text{ proc } \text { inc } \triangleq (\text{val-res } n : \mathbb{N} \bullet n := n + 1) \bullet \text{ inc } (n) \].
The body of the procedure in this case is a parameterized command [2].

For example, we have the square root program. Its specification is
\[ x : [0 \leq x, x^2 = x_0] \]

We start the development by introducing the procedure \texttt{sqrts}, which returns the
square root of a given real number.

\begin{verbatim}
\begin{array}{c}
\sqsubseteq \text{procArgsIntro}(\texttt{sqrts}, b : [0 \leq a, b^2 = a], (\texttt{val a : R; res b : R}))
\hline
\text{ proc }\text{ sqrts } \triangleq (\text{val a : R; res b : R} \bullet b : [0 \leq a, b^2 = a]) \bullet 
  x : [0 \leq x, x^2 = x_0] \\
\end{array}
\end{verbatim}

The next step is to introduce a substitution by value and contract the frame to
remove the variable \( a \).

\begin{verbatim}
\begin{array}{c}
\sqsubseteq \text{callByValueIV}(x, a)
\hline
(\texttt{val a : R} \bullet x, a : [0 \leq a, x^2 = a_0])(x) \\
\end{array}
\end{verbatim}

\begin{verbatim}
\begin{array}{c}
\sqsubseteq \text{contractFrame}(a)
\hline
(\texttt{val a : R} \bullet x : [0 \leq a, x^2 = a])(x) \\
\end{array}
\end{verbatim}

Then, we introduce a substitution by result and use the law \texttt{multiArgs} to put
our two kinds of substitution together.

\begin{verbatim}
\begin{array}{c}
\sqsubseteq \text{callByResult}(x, b)
\hline
(\texttt{res b : R} \bullet b : [0 \leq a, b^2 = a])(x) \\
\end{array}
\end{verbatim}

\begin{verbatim}
\begin{array}{c}
\sqsubseteq \text{multiArgs}()
\hline
(\texttt{val a : R, res b : R} \bullet b : [0 \leq a, b^2 = a])(x, x) \\
\end{array}
\end{verbatim}

Finally, we introduce the procedure call.

\begin{verbatim}
\sqsubseteq \text{procArgsCall}()
\hline
\texttt{sqrts}(x, x) \\
\end{verbatim}

Our resulting program introduces the procedure \texttt{sqrts} and uses it to calculate
the square root of a given number \( x \).

\[
\begin{align*}
\text{proc} & \quad \text{sqrt} \equiv (\text{val} \ a : \mathbb{R}, \text{res} \ b : \mathbb{R} \bullet \ b : [0 \leq a, b^2 = a]) \bullet \\
& \quad \text{sqrt}(x, x)
\end{align*}
\]

Variant blocks are used to develop recursive procedures. Besides declaring the procedure and the main program, a variant block declares a variant \( v \) used to develop a recursive implementation for the procedure. The general form is

\[
\begin{align*}
\text{proc} & \quad \text{name} \equiv \text{body} \quad \text{variant} \ v \quad \text{is} \ e \bullet \ prog
\end{align*}
\]

We present the factorial program as an example of a recursive procedure development. We have the following specification as the initial specification.

\[
f : [f = n! * 1]
\]

The variant introduction is the first step of the development. We define the value of the variable \( m \) as the variant \( V \).

\[
\begin{align*}
\text{variantIntro} & \quad (\text{fact}, V, m, (f : [f = m! * k]), \text{val} \ m, k : \mathbb{N})
\end{align*}
\]

\[
\begin{align*}
\text{proc} & \quad \text{fact} \equiv (\text{val} \ m, k : \mathbb{N} \bullet f : [V = m, f = m! * k])
\end{align*}
\]

\[
\begin{align*}
\text{variant} \ V \quad \text{is} \ m \bullet \\
& \quad f : [f = n! * 1]
\end{align*}
\]

Then we introduce a substitution by value of the variables \( m \) and \( k \).

\[
\begin{align*}
\text{callByValue} & \quad (m, k : \mathbb{N}, (n, 1))
\end{align*}
\]

\[
\begin{align*}
\text{proc} & \quad \text{fact} \equiv (\text{val} \ m, k : \mathbb{N} \bullet f : [V = m, f = m! * k])
\end{align*}
\]

\[
\begin{align*}
\text{variant} \ V \quad \text{is} \ m \bullet \\
& \quad (\text{val} \ m, k : \mathbb{N} \bullet f : [f = m! * k])(n, 1)
\end{align*}
\]

The next step is to introduce the procedure call in the variant block.

\[
\begin{align*}
\text{procVariantBlockCall}()
\end{align*}
\]

\[
\begin{align*}
\text{proc} & \quad \text{fact} \equiv (\text{val} \ m, k : \mathbb{N} \bullet f : [V = m, f = m! * k])
\end{align*}
\]

\[
\begin{align*}
\text{variant} \ V \quad \text{is} \ m \bullet \\
& \quad \text{fact}(n, 1)
\end{align*}
\]

Now, we start to develop the body of the procedure \( \text{fact} \). First, we introduce an
alternation in order to check if we must stop the recursion ($m = 0$) or not ($m > 0$).

\[\text{law } \text{alt}((m = 0, m > 0))\]

\[
\begin{array}{ll}
\text{if } m = 0 & \rightarrow f : [V = m \land m = 0, f = m! * k] \\
\text{if } m > 0 & \rightarrow f : [V = m \land m > 0, f = m! * k]
\end{array}
\]

This law application generates the following proof obligation

\[V = m \Rightarrow m = 0 \lor m > 0\]

The base case is when we have $m$ equals to 0. In such case, $k$ must receive the value of the variable $f$.

\[\text{assign}(f := k)\]

\[f := k\]

The following proof obligation is generated in this case

\[V = m \land m = 0 \Rightarrow k = 0! * k\]

If $m > 0$ we should make a recursive call. We first strengthen the postcondition in order to make a recursive call of procedure $\text{fact}$.

\[\begin{array}{ll}
\text{strPost}(f = (m - 1)! * m * k) & \leftarrow \\
\text{val } m, k : \mathbb{N} \bullet f : [V = m + 1 \land m + 1 > 0, f = m! * k] & \leftarrow \\
\end{array}
\]

We have the following proof obligation

\[f = (m - 1)! * m * k \Rightarrow f = m! * k\]

Then, we introduce a substitution by value and weaken the pre condition.

\[\begin{array}{ll}
\text{callByValue}((m, k), (m - 1, m * k)) & \leftarrow \\
\text{weakPre}(0 \leq m < V) & \leftarrow \\
\end{array}
\]

This law generates the following proof obligation

\[V = m + 1 \land m + 1 > 0 \Rightarrow 0 \leq m < V\]

At this point of the program development, we have the following collected pro-
gram.

\[
\begin{array}{l}
\begin{array}{l}
\text{proc } fact \equiv (\text{val } m, k : \mathbb{N} \cdot \\
\text{if } m = 0 \rightarrow f := k \\
\quad m > 0 \rightarrow f : [0 \leq m < V, f = m! \ast k]
\end{array}
\end{array}
\]

\text{variant } V \text{ is } m

\text{fact}(n, 1)\]

Finally, we refine this program by introducing a recursive call to the procedure \text{fact} as seen below.

\[
\begin{array}{l}
\begin{array}{l}
\text{proc } fact \equiv (\text{val } m, k : \mathbb{N} \cdot \\
\text{if } m = 0 \rightarrow f := k \\
\quad m > 0 \rightarrow \text{fact}(m - 1, m \ast k)
\end{array}
\end{array}
\]

\text{variant } V \text{ is } m

\text{fact}(n, 1)\]

This law application generates the following proof obligation

\[
\begin{array}{l}
f : [V = n, f = m! \ast k]
\end{array}
\]

\[
\begin{array}{l}
\text{if } m = 0 \rightarrow f := k \\
\quad m > 0 \rightarrow f : [0 \leq m < V, f = m! \ast k]
\end{array}
\]

This finishes the development of the factorial program.

The development of some examples seen here make use of development strategies which can be found in the literature. For instance, the first example introduces an initialized iteration using a conjunction as its invariant. The sequence of law applications to introduce such an iteration can be formalized as a tactic, reducing time and effort in program developments. Next chapter presents ArcAngel, a language for tactic formalization.

1.3 Overview

Chapter 2 introduces ArcAngel. First, we present ArcAngel’s Syntax and give an informal description of the language. Then, we present some commonly used refinement strategies (tactics) written in ArcAngel. Finally, we present ArcAngel’s formal semantics.
Based on this semantics, Chapter 3 presents laws of reasoning. These laws are used to support a strategy of reduction to a normal form for ArcAngel. All the laws used in this work are proved in Appendix D.

In Chapter 4 we discuss the implementation of Gabriel, a tool that supports the use of ArcAngel. We present the concepts of the tool. Then we discuss briefly its user interface, presenting the constructs of ArcAngel available in Gabriel, and an example of a tactic creation and usage.

Finally, in Chapter 5 we summarize the results obtained and make some final considerations. Related and future works are also discussed in this chapter.
Chapter 2

ArcAngel

In this chapter we present the language ArcAngel, a refinement-tactic language based on Angel. Most of basic tactics in ArcAngel are inherited from Angel. However, almost all ArcAngel’s structural combinators are not defined in Angel. ArcAngel’s extension for Angel can be found in Appendix A. All the program development strategies in the literature are formalized as tactics written in ArcAngel and their use are also presented. Finally, we present ArcAngel’s formal semantics which is totally different from Angel’s formal semantics.
2.1 ArcAngel’s Syntax

The syntax of ArcAngel is displayed in Figure 2.1. The syntactic category args is the set of (possibly empty) comma-separated lists of arguments enclosed in parentheses. An argument is a predicate; an expression; a variable or constant declaration; a list of predicates, expressions, variables, or arguments declarations; or a program. The category pars is the set of possibly empty comma-separated lists of (parameter) names enclosed in parentheses. The definitions of these syntactic categories are standard and are omitted. Finally, the notation tactic+ is used to represent a non-empty list of tactics; in examples, we number the elements of this list. A pair of square brackets represents optional clauses.

There are three distinct kinds of tactics: basic tactics are the simplest tactics; tacticals are combination of tactics; and structural combinators are tactics used to handle parts of a program. Some of the basic tactics and most of the tacticals are original to ArcAngel; most of the structural combinators in ArcAngel do not exist in Angel. ArcAngel’s extension for Angel can be found in Appendix A.

2.1.1 Basic Tactics

The most basic tactic is a simple law application: law n(a). The application of this tactic to a program has two possible outcomes. If the law n with arguments a is applicable to the program, then the application actually occurs and the program is changed, possibly generating proof obligations. If the law is not applicable to the program, then the application of the tactic fails.

The construct tactic n(a) applies a previously defined tactic as though it were a single law. The trivial tactic skip always succeeds, and the tactic fail always fails; neither generates any proof obligations. The tactic abort neither succeeds nor fails, but runs indefinitely.

2.1.2 Tacticals

In ArcAngel, tactics may be sequentially composed: t1; t2. This tactic first applies t1 to the program, and then applies t2 to the outcome of the application of t1. If either t1 or t2 fails, then so does the whole tactic. When it succeeds, the proof obligations generated by the application of this tactic are those resulting from the application of t1 and t2.

For example, consider the program x : [ x ≥ 1 ]. We could implement this by first strengthening the postcondition to x = 1, and then replacing it with a simple assignment. The necessary tactic is law strPost(x = 1); law assign(x := 1). After the application of the first tactic, the resulting program is x : [ x = 1 ] and the proof obligation is x = 1 ⇒ x ≥ 1. After the application of the assignment introduction
\[ \text{tactic ::= law name args \quad [law application]} \]

\[ \text{\quad tactic name args \quad [tactic application]} \]

\[ \text{\quad skip \quad fail \quad abort} \]

\[ \text{\quad tactic; tactic \quad [sequence]} \]

\[ \text{\quad tactic \mid tactic \quad [alternation]} \]

\[ \text{\quad ! tactic \quad [cut]} \]

\[ \mu \text{ name} \bullet \text{ tactic \quad [recursion]} \]

\[ \text{\quad succs tactic \mid fails tactic \quad [assertions]} \]

\[ \text{\quad tactic tactic \quad [structural combinators]} \]

\[ \text{\quad if tactic} [\text{If}] \mid \text{do tactic} [\text{Do}] \od \]

\[ \text{\quad var tactic} [\text{Var}] \mid \text{con tactic} [\text{Con}] \]

\[ \text{\quad pmain tactic} [\text{Pmain}] \]

\[ \text{\quad pmainvariant tactic} [\text{Pmainvariant}] \]

\[ \text{\quad pbody tactic} [\text{Pbody}] \]

\[ \text{\quad pbodyvariant tactic} [\text{Pbodyvariant}] \]

\[ \text{\quad pbodymain tactic tactic} [\text{Pbodymain}] \]

\[ \text{\quad pmainvariantbody tactic tactic} [\text{Pmainvariantbody}] \]

\[ \text{\quad val tactic \quad [constants]} \]

\[ \text{\quad res tactic \quad [patterns]} \]

\[ \text{\quad val-res tactic \quad [patterns]} \]

\[ \text{\quad parcommand tactic \quad [patterns]} \]

\[ \text{\quad con v \bullet tactic} \]

\[ \text{\quad applies to program do tactic \quad [patterns]} \]

\[ \text{tacDec ::= Tactic name pars tactic} \]

\[ \text{\quad \quad [proof obligations predicate*]} \]

\[ \text{\quad \quad [program generated program] end} \]

\[ \text{tacProg ::= tacDec* tactic} \]

Figure 2.1: Abstract Syntax of ArcAngel
law, we get the program $x := 1$ with the two proof obligations $x = 1 \Rightarrow x \geq 1$ and true $\Rightarrow 1 = 1$. The discharge of proof obligations is not tackled by ArcAngel. In our examples, and in Gabriel, the generated proof obligations are listed, but not proved. Support for their discharge is left as future work.

In examples, for clarity, sometimes we use assignments as arguments as a syntactic sugar. For instance, above, we write `law assign(x := 1)`, but the arguments are actually $x$ and 1. We use the same sort of notation for arguments and parameters of tactics.

Tactics may also be combined in alternation: $t_1 | t_2$. First $t_1$ is applied to the program. If the application of $t_1$ leads to success, then the composite tactic succeeds; otherwise $t_2$ is applied to the program. If the application of $t_2$ leads to success then the composite tactic succeeds; otherwise the composite tactic fails. If one of the tactics aborts, the whole tactic aborts. When a tactic contains many choices, the first choice that leads to success is selected. It is the angelic nature of this nondeterminism, in which alternative tactics that lead to success are chosen over those that lead to failure, that earned Angel and ArcAngel (A Refinement Calculus for Angel) their names.

For example, suppose that we have a useful tactic $t$ that relies on the frame containing only the variable $x$, and that we want to generalize it to $t'$, which applies to specification statements with frame $x, y$. We could define $t'$ to be `law contractFrame(y); t`. Unfortunately, the resulting tactic no longer applies where we found $t$ to be useful, since contracting the frame will surely fail in such cases. Instead, we can use the compound tactic `(law contractFrame(y) | skip); t`. Now, if contracting the frame works, we do it; if it does not, then we ignore it; either way, we apply $t$ next. The tactic `(skip | law contractFrame(y)); t` has the same effect. In this case, the tactic $t$ is applied without any prior change to the specification statement. If this application does not succeed, the law `contractFrame` is applied before a new attempt to apply $t$.

The angelic nondeterminism can be implemented through backtracking: in the case of failure, law applications are undone to go back to the last point where further alternatives are available and can be explored. This, however, may result in inefficient searches. Some control over this is given to the programmer through the cut operator. The cut tactic `! t` behaves like $t$, except that it returns the first successful application of $t$. If a subsequent tactic application fails, then the whole tactic fails. Consider our previous example once more, supposing that `law contractFrame(y)` succeeds, but that $t$ subsequently does not. There is no point in applying the trivial tactic `skip` and then trying $t$ again. Instead, we should cut [7] the search: `!(law contractFrame(y) | skip); t`.

Suppose that we have a tactic $u$ that performs some simple task, like identifying an equation in a specification postcondition and then applying the rule of `following-assignment`. Such a simple tactic may be made more useful by applying
it repeatedly, until it can be applied no more. ArcAngel has a fixed-point operator that allows us to define recursive tactics. Using this operator, we can define a tactic that applies \( u \) exhaustively: the tactic \( \mu X \cdot (u; X \mid \text{skip}) \) applies \( u \) as many times as possible, terminating with success when the application of \( u \) fails. Recursive application of a tactic may lead to nontermination, in which case the result is the same as the trivial tactic \text{abort}.

The tactic \text{con} \( v \cdot t \) introduces \( v \) as a set of free variables ranging over appropriate syntactic classes in \( t \), angelically chosen so that as many choices as possible succeed. The tactic \text{applies to} \( p \) \text{do} \( t \) is used to define a pattern for the programs to which the tactic \( t \) can be applied. It introduces a meta-program \( p \) that characterizes the programs to which this tactic is applicable; the meta-variables used in \( p \) can then be used in \( t \). For example, the meta-program \( w : \{\text{pre}, \text{post}_1 \lor \text{post}_2\} \) characterizes those specifications whose postcondition is a disjunction; here, \( \text{pre} \), \( \text{post}_1 \), and \( \text{post}_2 \) are the meta-variables. Consider as an example a commonly used refinement tactic: strengthening a postcondition by dropping a disjunct. This tactic is formalized as \text{applies to} \( w : \{\text{pre}, \text{post}_1 \lor \text{post}_2\} \text{do law strPost(post}_1) \).

Two tactics are used to make tactic assertions that check the outcome of applying a tactic. The tactic \text{succs} \( t \) fails whenever \( t \) fails, and behaves like \text{skip} whenever \( t \) succeeds. On the other hand, \text{fails} \( t \) behaves like \text{skip} if \( t \) fails, and fails if \( t \) succeeds. If the application of \( t \) runs indefinitely, then these tacticals behave like \text{abort}. A simple example is a test to see whether a program is a specification statement. We know that it is always possible (but seldom desirable) to strengthen a specification statement’s postcondition to \text{false}; however, the tactic applies only to specification statements. So, our test may be coded as \text{succs} \( \text{law strPost(false)} \).

2.1.3 Structural Combinators

Very often, we want to apply individual tactics to subprograms. The tactic \( t_1; t_2 \) applies to programs of the form \( p_1; p_2 \). It returns the sequential composition of the programs obtained by applying \( t_1 \) to \( p_1 \) and \( t_2 \) to \( p_2 \); the proof obligations generated are those arising from both tactic applications. Combinators like \( ; \) are called structural combinators. These combinators correspond to the syntactic structures in the programming language. Essentially, there is one combinator for each syntactic construct.

For alternation, there is the structural combinator \( \text{if} t_1 [\ldots] t_n \text{fi} \), which applies to an alternation \( \text{if} g_1 \rightarrow p_1 [\ldots] g_n \rightarrow p_n \text{fi} \). It returns the result of applying each tactic \( t_i \) to the corresponding program \( p_i \). For example, if we have
the program

if \( a < b \rightarrow x : [ x < 0 ] \) \( a = b \rightarrow x : [ x = 0 ] \) \( a > b \rightarrow x : [ x > 0 ] \) fi

and tactic

\[
\text{if } \text{law assign}(x := -1) \text{ law assign}(x := 0) \text{ law assign}(x := 1) \text{ fi}
\]

we obtain three proof obligations \( \text{true} \Rightarrow -1 < 0 \) and \( \text{true} \Rightarrow 0 = 0 \), and \( \text{true} \Rightarrow 1 > 0 \), and if \( a < b \rightarrow x := -1 \) \( a = b \rightarrow x := 0 \) \( a > b \rightarrow x := 1 \) fi as the resulting program. For iterations do \( y_1 \rightarrow p_1 \) \( \ldots \) \( y_n \rightarrow p_n \) od, we have a similar structural combinator \( \text{do} t_1 \) \( \ldots \) \( t_n \text{ od} \).

The structural combinator \( \text{var } t \) applies to a variable block, and \( \text{con } t \) to a logical constant block; each applies its tactic \( t \) to the body of the block. For example, if we apply to \( \text{var } x : \mathbb{N} \bullet x : [x \geq 0] \) the structural combinator \( \text{var } \text{law assign}(x := 10) \), we get \( \text{var } x : \mathbb{N} \bullet x := 10 \) and the proof obligation \( \text{true} \Rightarrow 10 \geq 0 \).

In the case of procedure blocks and variant blocks, the structural combiners \( \text{pmain } t \) and \( \text{pmainvariant } t \) are used, respectively; they apply \( t \) to the main program of the blocks. For example, applying \( \text{pmain } \text{law assign}(x := 10) \) to the procedure block

\[
\text{proc nonNeg } \overset{=} x : [ x > 0 ] \bullet x : [ x \geq 0 ]
\]

we get \( \text{proc nonNeg } \overset{=} x : [ x > 0 ] \bullet x := 10 \) and the proof obligation \( \text{true} \Rightarrow 10 \geq 0 \).

To apply a tactic to a procedure body, we use the structural combiners \( \text{pbody } t \) and \( \text{pbodyvariant } t \), which apply to procedure and variant blocks, respectively. For example, if we apply the tactic

\[
\text{pbody } \text{law assign}(x := 10)
\]

to the program

\[
\text{proc nonNeg } \overset{=} x : [ x \geq 0 ] \bullet \text{nonNeg}
\]

we get

\[
\text{proc nonNeg } \overset{=} x := 10 \bullet \text{nonNeg}
\]

and the proof obligation \( \text{true} \Rightarrow 10 \geq 0 \).

It is also possible to apply tactics to a procedure body and to the main program of a procedure block, or a variant block, at the same time. For this, we use the structural combiners \( \text{pbodymain } t_b \ t_m \) and \( \text{pmainvariantbody } t_b \ t_m \), which apply to procedure blocks and variant blocks, respectively. They apply \( t_b \) to the body of the procedure, and \( t_m \) to the main program.

For argument declaration, the combiners \( \text{val } t \), \( \text{res } t \), and \( \text{val-res } t \) are used, depending on whether the arguments are passed by value, result, or value-result.
For example, when the following tactic
\[
\text{pbody} \text{val-res law assign}(x := 10) \]
is applied to the procedure block
\[
\begin{array}{c}
\text{proc nonNegArg} \equiv (\text{val-res } x : \mathbb{N} \cdot x : [x \geq 0]) \cdot x : [x \geq 0] \\
\end{array}
\]
it returns the proof obligation \(\text{true} \Rightarrow 10 \geq 0\) and the program
\[
\begin{array}{c}
\text{proc nonNegArg} \equiv (\text{val-res } x : \mathbb{N} \cdot x := 10 \cdot x : [x \geq 0]) \\
\end{array}
\]
Sometimes, however, we are not concerned with the type of argument declaration. In these cases, we use the structural combinator \text{parcommand} \(t\). For example, if we apply the tactic
\[
\text{pmain parcommand law assign}(x := 10) \]
to the program
\[
\begin{array}{c}
\text{proc nonNegArg} \equiv \text{body} \cdot (\text{val-res } x : \mathbb{N}; \text{val } y : \mathbb{N} \cdot x : [x \geq 0]) \\
\end{array}
\]
it returns the proof obligation \(\text{true} \Rightarrow 10 \geq 0\) and the program
\[
\begin{array}{c}
\text{proc nonNegArg} \equiv \text{body} \cdot (\text{val-res } x : \mathbb{N}; \text{val } y : \mathbb{N} \cdot x := 10) \\
\end{array}
\]
We may declare a named tactic with arguments using \text{Tactic n(a) t end}. For documentation purposes, we may include the clause \text{proof obligations} and the clause \text{program generated}; the former lists the proof obligations generated by the application of \(t\), and the latter shows the program generated. These two clauses are optional as this information can be inferred from the tactic itself. The effect of \text{Tactic n(a) t end} is that of \(t\), which is named \(n\) and uses the arguments \(a\); the presence of the optional clauses does not affect the behavior.

A tactic program consists of a sequence of tactic declarations followed by a tactic that usually makes use of the declared tactics.

### 2.2 Examples

In this section we give a few examples of ArcAngel programs. The first two, \text{followingAssign} and \text{leadingAssign}, implement two derived rules from Morgan’s calculus. The other tactics formalize all the refinement strategies we found in the literature. Some of these examples were first presented in [28]. More examples can also be found in [25].
2.2.1 Following Assignment

In [23, p.32], the derived rule following assignment is presented as a useful combination of the assignment and sequential composition laws; in other words, it is rather like a tactic for using the more basic laws of the calculus. Of course, there is a big difference between a tactic and a derived rule. The former is a program that applies rules; proof obligations arise from the law applications. A derived rule is a law itself, that is proved in terms of applications of other laws. On the other hand, tactics are much more flexible than derived rules, since they can make choices about the form of the resulting program; derived rules cannot.

Following assignment splits a specification statement into two pieces, the second of which is implemented by the assignment $x := E$. If we apply to $w, x : [pre, post]$ the tactic law seqComp($post \{ x \setminus E \}$), we get the program

$$w, x : [pre, post \{ x \setminus E \}]; \quad w, x : [post \{ x \setminus E \}, post]$$

All the refinement laws are defined in Appendix C.

If we now apply assign($x := E$) to the second statement, we get our assignment $x := E$ and the proof obligation $post \{ x \setminus E \} \Rightarrow post \{ x \setminus E \}$. This is a simple tautology, but remains part of the documentation of the tactic, as the proof obligation is raised every time the tactic is applied.

**Tactic** followingAssign($x, E$)

**applies to** $w, x : [pre, post]$ do

  law seqComp($post \{ x \setminus E \}$); (skip[] law assign($x := E$))

**proof obligations**

  $post \{ x \setminus E \} \Rightarrow post \{ x \setminus E \}$

**program generated**

  $w, x : [pre, post \{ x \setminus E \}];$

  $x := E$

end

There is a further restriction on the use of this tactic: it uses a simple form of the law of sequential composition that forbids initial variables in post conditions. If applied to a specification statement that does not satisfy this restrictions, followingAssign fails.
2.2.2 Leading Assignment

In [23, p.71], the presentation of the derived rule for leading assignment is a little more complicated than that of the following assignment:

Law 13.1 Leading assignment For any expression $E$,

$$w, x : [ \text{pre} [x \setminus E], \text{post} [x_0 \setminus E_0] ]$$

$$\sqsubseteq x := E;$$

$$w, x : [\text{pre}, \text{post}]$$

The expression $E_0$ is that obtained from $E$ by replacing all free occurrences of $x$ and $w$ with $x_0$ and $w_0$, respectively. The complication arises from the need to find predicates $\text{pre}$ and $\text{post}$, such that, when appropriate substitutions are made, they match the current goal.

Tactic $\text{leadingAssign} (x, X, E)$

applies to $w, x : [\text{pre} [x \setminus E], \text{post} [x_0 \setminus E_0]]$ do

law $\text{seqCompCon (pre } \land x = E_0, X, x )$;

con law $\text{assignIV (x := E)}$;

law $\text{strPostIV (post)}$; law $\text{weak (pre)}$;

law $\text{removeCon (X)}$

proof obligations

$$(x = x_0) \land \text{pre [x \setminus E]} \Rightarrow (\text{pre } \land x = E [x \setminus x_0] [x \setminus E]),$$

$$(\text{pre } \land x = E [x \setminus X]) [x, w \setminus x_0, w_0] \land \text{post} \Rightarrow \text{post} [x_0 \setminus E_0] [x_0 \setminus X],$$

$$(\text{pre } \land x = E [x \setminus X]) \Rightarrow \text{pre}$$

program generated

$$x := E; \ w, x : [\text{pre}, \text{post}]$$

end

If applied to a specification statement $w, x; [\text{pre} [x \setminus E], \text{post} [x_0 \setminus E_0]]$, this tactic first splits it using the law $\text{seqCompCon}$. As $\text{seqComp}$, this law introduces sequences, but it applies to specification statements that make use of initial variables; it declares a logical constant to record the initial value of $x$. This gives:

$$\parallel \text{con X} \bullet$$

$$x : [\text{pre} [x \setminus E], \text{pre } \land x = E [x \setminus x_0]];$$

$$w, x : [\text{pre } \land x = E [x \setminus X], \text{post} [x_0 \setminus E_0] [x_0 \setminus X]]$$

$$\parallel$$

The first specification statement is refined to $x := E$, using the $\text{assignIV}$ law: a
law for introducing assignments that considers the presence of initial variables. The precondition and postcondition of the second specification statement are changed to \( pre \) and \( post \), respectively, and finally the logical constant is removed. The law \( \text{strPostIV} \) is the strengthening postcondition law that takes initial variables into account. The proof obligations are generated by the applications of the laws \( \text{assignIV} \), \( \text{strPostIV} \), and \( \text{weakPre} \), respectively; they always hold.

### 2.2.3 TakeConjAsInv

This refinement strategy that we formalize in this section aims at the development of an initialized iteration. It consists of taking a conjunct of the postcondition of the program specification as the main part of the invariant. The tactic \( \text{takeConjAsInv} \) defined below follows this strategy to transform a specification \( w : [pre, \text{invConj} \land \neg \text{guard}] \) into an initialized iteration.

**Tactic** \( \text{takeConjAsInv} (\text{invBound}, (ivar := ival), \text{variant}) \)

**applies to** \( w : [pre, \text{invConj} \land \neg \text{guard}] \)

\[
\begin{align*}
&\text{law} \ \text{strPost}(\text{invBound} \land \text{invConj} \land \neg \text{guard}); \\
&\text{law} \ \text{seqComp}(\text{invBound} \land \text{invConj}); \\
&(\text{law} \ \text{assign}(ivar := ival) \ [\text{law} \ \text{iter}\langle\text{guard}\rangle, \text{variant})])
\end{align*}
\]

**proof obligations**

\[
\begin{align*}
&\text{invBound} \land \text{invConj} \land \neg \text{guard} \Rightarrow \text{invConj} \land \neg \text{guard}, \\
&pre \Rightarrow (\text{invBound} \land \text{invConj})[ivar \setminus ival]
\end{align*}
\]

**program generated**

\[
\begin{align*}
ivar &:= ival; \\
\text{do} \ \text{guard} \rightarrow \\
&w : [\text{invBound} \land \text{invConj} \land \text{guard}, \\
&\text{invBound} \land \text{invConj} \land 0 \leq \text{variant} < \text{variant}_0]
\end{align*}
\]

end

This tactic has three arguments: a predicate \( \text{invBound} \), an assignment \( ivar := ival \), where \( ivar \) is a list of variables and \( ival \) is an equal-length list of values, and an integer expression \( \text{variant} \). This tactic applies to a specification statement \( w : [pre, \text{invConj} \land \neg \text{guard}] \) to introduce an initialized iteration whose invariant is \( \text{invConj} \land \text{invBound} \) and whose variant is \( \text{variant} \). The initialization is \( ivar := ival \). Typically, the predicate \( \text{invBound} \) states the range limits of indexing variables of the iteration. The conjunction \( \text{invConj} \land \text{invBound} \) is used as invariant of the iteration.
The tactic \textit{takeConjAsInv} first strengthens the postcondition (law \textit{strPost}) using the argument \textit{invBound}, then it introduces a sequential composition (law \textit{seqComp}). Afterwards, the law \textit{assign} is applied to the first program of the composition to derive the initialization, and the law \textit{iter} is applied to the second program in order to introduce the iteration. The first proof obligation is generated by the application of the law \textit{strPost}, and the second by the application of the law \textit{assign}.

Example. We now present the use of a tactic in the first example presented in Section 1.2. This example develops a program which, given \(a\) and \(b\), such that \(a \geq 0\) and \(b > 0\), sets \(q\) to the quotient of \(a\) divided by \(b\), and sets \(r\) to the remaining of this quotient.

As already presented before, we begin the development by strengthening the postcondition in order to insert the mathematical definition of quotient and of a quotient’s remaining.

\[
q, r : [a \geq 0 \land b > 0, q = a \div b \land r = a \mod b]
\]

\[
\models \text{strPost}((a = q \ast b + r \land 0 \leq r) \land \neg r \geq b)
\]

\[
q, r : [a \geq 0 \land b > 0, (a = q \ast b + r \land 0 \leq r) \land \neg r \geq b]
\]

This law generates the following proof obligation.

\[
(a = q \ast b + r \land 0 \leq r) \land \neg r \geq b \Rightarrow (q = a \div b \land r = a \mod b).
\]

Now, we apply the tactic \textit{takeConjAsInv} using the bound limits of the invariant \(b > 0\), the initialization of the variables \(q, r := 0, a\) and the variant \(r\) of the iteration.

\[
\models \text{takeConjAsInv}(b > 0, (q, r := 0, a), r)
\]

This application represents a simple application and should give a program as result. However, for better understanding, we present the details of tactics applications. We follow with the application of the tactic \textit{takeConjAsInv}.

\[
\models \text{strPost}(b > 0 \land a = q \ast b + r \land 0 \leq r \land \neg r \geq b)
\]

\[
q, r : [a \geq 0 \land b > 0, b > 0 \land a = q \ast b + r \land 0 \leq r \land \neg r \geq b]
\]

This law generates the following proof obligation.

\[
(b > 0 \land a = q \ast b + r \land 0 \leq r \land \neg r \geq b) \Rightarrow (a = q \ast b + r \land 0 \leq r) \land \neg r \geq b.
\]

Most of the proof obligations generated in the examples are very simple like this one; they follow directly from the properties of the predicate calculus and of the data type involved, mainly numbers. Then, the tactic splits the specification into
two parts. The first one is refined to the initialization of the iteration.

\[ \sqsubseteq \text{seqComp}(b > 0 \land a = q \ast b + r \land 0 \leq r) \]
\[ q, r : [a \geq 0 \land b > 0, b > 0 \land a = q \ast b + r \land 0 \leq r]; \]
\[ q, r : \left[ \begin{array}{lcl}
q > 0 \land a = q \ast b + r \land 0 \leq r \\
0 \leq r \land r \geq b
\end{array} \right] \]
\[ \sqsubseteq \text{assign}(q, r := 0, a) \]
\[ q, r := 0, a \]

This law generates the following proof obligation.

\[ a \geq 0 \land b > 0 \Rightarrow b > 0 \land a = 0 \ast b + a \land 0 \leq a. \]

Finally, the tactic introduces the iteration.

(1) \[ \sqsubseteq \text{iter}(\{r \geq b\}, r) \]
\[ q, r : \left[ \begin{array}{lcl}
q > 0 \land a = q \ast b + r \\
0 \leq r \land r \geq b
\end{array} \right] \]
\[ \text{do } r \geq b \rightarrow \]
\[ q, r : \left[ \begin{array}{lcl}
q > 0 \land a = q \ast b + r \\
0 \leq r \land r \geq b
\end{array} \right] \]
\[ \text{od} \]

This finishes the tactic application. The output of its application is

\[ q, r := 0, a; \]
\[ \text{do } r \geq b \rightarrow \]
\[ q, r : \left[ \begin{array}{lcl}
q > 0 \land a = q \ast b + r \\
0 \leq r \land r \geq b
\end{array} \right] \]
\[ \text{od} \]

To finish the development we introduce the assignments in the body of the iteration. At each step of the iteration we must increment \( q \) by one and decrement
assignIV(q, r := q + 1, r − b)
q, r := q + 1, r − b

This law generates the following proof obligation.

\[(q = q_0) \land (r = r_0) \land b > 0 \land a = q \cdot b + r \land 0 \leq r \land r \geq b \Rightarrow b > 0 \land a = (q + 1) \cdot b + (r - b) \land 0 \leq (r - b) \land 0 \leq (r - b) < r_0).\]

So, by applying the following tactic to the original specification

- law strPost((a = q \cdot b + r \land 0 \leq r) \land \neg r \geq b);
- tactic takeConjAsInv(b > 0, (q, r := 0, a), r);
- law skipInt
- (do law assignIV(q, r := q + 1, r − b) od)

we get the following program

\[
q, r := 0, a;
\text{do } r \geq b \rightarrow \text{ q, r := q + 1, r − b } \text{ od}
\]

which correct implements the initial specification.

### 2.2.4 ReplConsByVar

Another common way of choosing an iteration invariant is replacing a constant in the specification postcondition by a variable. This strategy is captured by the tactic replConsByVar.

The tactic replConsByVar has five arguments: the declaration newV : T of the fresh loop-index variable; the constant cons to be replaced; an invariant invBound on the loop indexes; the loop initialization ivar := ival; and an integer-valued
Tactic replConsByVar (newV : T, cons, invBound, (ivar := ival), variant)
applies to w : [pre, post] do
  law varInt(newV : T);
  var law strPost(post[cons \ newV] \ → (newV \≠ cons));
  tactic takeConjAsInv(invBound, (ivar := ival), variant);
proof obligations
  post[cons \ newV] \ → (newV \≠ cons) \→ post,
  (invBound \∧ post[cons \ newV] \→ (newV \≠ cons)) \→
  post[cons \ newV] \→ (newV \≠ cons),
  pre \→ (invBound \∧ post[cons \ newV] \∧ invBound)[ivar \→ ival]
program generated
[ [ var newV : T •
  var: ival;
  do newV \≠ cons →
    newV, w : \{ \{ invBound \∧ post[cons \ newV]
       \{ newV \≠ cons
       \{ invBound \∧ post[cons \ newV]
       \{ 0 ≤ variant < variant_0
       \} \} \} \}
  od ] ]

This tactic first introduces the new variable, then it strengthens the postcondition to replace the constant by the new variable. Afterwards it calls the tactic takeConjAsInv to introduce the iteration. The first proof obligation is generated by the law strPost, and the others are generated by the tactic takeConjAsInv. The first and the second proof obligations always hold.

Example. The following example presents a program to calculate, given two non-negative numbers a and b, the exponentiation $a^b$. Its formal specification is

$$r : [a \geq 0 \land b \geq 0, r = a^b]$$

The first step of the refinement is to apply the tactic replConsByVar with the loop index $x : \mathbb{Z}$, the constant to be replaced $b$, the invariant of the loop indexes $0 \leq x \leq b$, the variables initialization $(x, r := 0, 1)$, and the variant $b - x$ of the
iteration as arguments. First, the tactic introduces the new variable $x$ which is used in the iteration. Then, the tactic strengthens the postcondition in order to make a relation between the new variable $x$ and the constant $b$ to be replaced.

```
⊑ replConsByVar(x : Z, b, 0 ≤ x ≤ b, (x, r := 0, 1), b − x)
    ⊑ varInt(x : Z)
        [ var x : Z •
            r, x : [a ≥ 0 ∧ b ≥ 0, r = a^b]
        ] ⊑ strPost(r = a^x ∧ ¬ x ≠ b)
            r, x : [a ≥ 0 ∧ b ≥ 0, 0 ≤ x ≤ b ∧ r = a^x ∧ ¬ x ≠ b]
        ⊲
    ⊲

This law generates the following proof obligation.

$$(r = a^x ∧ ¬ x ≠ b) \Rightarrow (r = a^b)$$

Then, the tactic calls the tactic `takeConjAsInv`, which first strengthens the postcondition in order to introduce the bound limits of the invariant.

```
⊑ takeConjAsInv(0 ≤ x ≤ b, (x, r := 0, 1), b − x)
    ⊑ strPost(0 ≤ x ≤ b ∧ r = a^x ∧ ¬ x ≠ b)
        r, x : [a ≥ 0 ∧ b ≥ 0, 0 ≤ x ≤ b ∧ r = a^x ∧ ¬ x ≠ b]
    ⊲
    ⊲

This law generates the following proof obligation.

$$(0 ≤ x ≤ b ∧ r = a^x ∧ ¬ x ≠ b) \Rightarrow (r = a^b ∧ ¬ x ≠ b)$$

Then, the tactic `takeConjAsInv` introduces a sequential composition. The first part is refined to the initialization of the iteration.

```
⊑ seqComp(0 ≤ x ≤ b ∧ r = a^x)
    r, x : [a ≥ 0 ∧ b ≥ 0, 0 ≤ x ≤ b ∧ r = a^x]
    ⊲
    ⊲ assign(x, r := 0, 1)
        x, r := 0, 1;
```

This law generates the following proof obligation.

$$(a ≥ 0 ∧ b ≥ 0) \Rightarrow (0 ≤ 0 ≤ b ∧ 1 = a^0)$$

Finally, the second part of the sequential composition is refined by the tactic
takeConjAsInv to an iteration.

(i) \( \subseteq \text{iter}((x \neq b), x - b) \)
\[\text{do } x \neq b \rightarrow \]
\[r, x : [\begin{align*}
0 \leq x \leq b, \\
r = a^x \land x \neq b, \\
0 \leq x \leq b, \\
r = a^x \land 0 \leq b - x \leq b - x_0
\end{align*}] \]
\[\od \]

To finish this development we should introduce the assignment in the body of the iteration as seen below.

\( \subseteq \text{assignIV}(r, x := r \ast a, x + 1) \)
\[r, x := r \ast a, x + 1; \]

This law generates the following proof obligation.

\[(r = r_0 \land x = x_0 \land 0 \leq x \leq b \land r = a^x \land x \neq b) \implies \\
(0 \leq x + 1 \leq b \land r \ast a = a^{x+1} \land 0 \leq b - (x + 1) \leq b - x_0)\]

So, by applying the following tactic to the original specification.

\texttt{tactic replConsByVar}(x : Z, b, 0 \leq x \leq b, (x, r := 0, 1), b - x);
\[\begin{array}{l}
\texttt{var skip} \\
\texttt{do law assignIV}(r, x := r \ast a, x + 1) \od
\end{array}\]

We get the following program.

\[\begin{array}{l}
\begin{array}{l}
\texttt{var} x : Z \bullet \\
x, r := 0, 1; \\
\texttt{do} x \neq b \rightarrow r, x := r \ast a, x + 1 \texttt{ od}
\end{array}
\end{array}\]

This program uses a variable \( x \), which is initialized with 0, in the iteration to count the number of times \( r \), which is initialized with 1, is multiplied by \( a \). In the end of the loop we have that \( r = a^b \).

2.2.5 StrengInv

Sometimes we cannot simply replace a constant by a variable in the postcondition of the specification to determine the invariant. First, we have to strengthen the postcondition.

The tactic \texttt{strengInv} has seven arguments: a variable declaration \( \texttt{newV}_1 : T_1 \), a predicate \( \texttt{streng} \) and the rest of the arguments are the same as those taken by the
Tactic replConsByVar. This tactic applies to a specification statement \( w : [pre, post] \) to introduce an initialized iteration with \((post \land streng)[cons \setminus newV_2] \land invBound\) as invariant and \(variant\) as variant. The initialization is \(ivar := ival\). In this tactic the predicate \(invBound\) states the range limits (typically 0 and \(cons\)) of the new variable \(newV_2\) which is used as an indexing variable of the iteration. The new variable \(newV_1\) is used as an auxiliary variable, and the predicate \(streng\) is used to make a link between the new variable introduced and the data used in the specification.

**Tactic strengInv**

\[
(\text{newV}_1 : T_1, \text{streng}, \text{newV}_2 : T_2, \text{cons}, \text{invBound}, (ivar := ival), \text{variant})
\]

applies to \(w : [pre, post]\) do

law \text{varInt} (\text{newV}_1 : T_1);

\[
\begin{align*}
\text{variant} & \text{ law} \text{ strPost}(post \land streng); \\
\text{tactic} & \text{ replConsByVar} \\
& (\text{newV}_2 : T_2, \text{cons}, \text{invBound}, (ivar := ival), \text{variant}) \\
\end{align*}
\]

proof obligations

\[
\begin{align*}
(pos \land streng) & \Rightarrow post, \\
(pos \land streng)[cons \setminus \text{newV}_2] \land \neg (\text{newV}_2 \neq \text{cons}) & \Rightarrow (pos \land streng), \\
(invBound \land (pos \land streng)[cons \setminus \text{newV}_2] \land \neg (\text{newV}_2 \neq \text{cons})) & \Rightarrow \\
& (pos \land streng)[cons \setminus \text{newV}_2] \land \neg (\text{newV}_2 \neq \text{cons}), \\
\text{pre} & \Rightarrow (invBound \land (pos \land streng)[cons \setminus \text{newV}_2] \land \\
& \text{invBound})[ivar \setminus ival]
\end{align*}
\]

program generated

\[
\begin{align*}
\text{newV}_1 & : T_1 \bullet \text{newV}_2 : T_2 \bullet \\
ivar & := ival; \\
\text{do} \text{newV}_2 \neq \text{cons} \rightarrow \\
\text{newV}_1, \\
\text{newV}_2, : \text{w} & \left[ \begin{array}{c}
\text{invBound} \\
(pos \land streng)[cons \setminus \text{newV}_2] \\
\text{newV}_2 \neq \text{cons} \\
\end{array} \right], \\
\text{newV}_1, \\
\text{newV}_2, : \text{w} & \left[ \begin{array}{c}
\text{invBound} \\
(pos \land streng)[cons \setminus \text{newV}_2] \\
0 \leq \text{variant} < \text{variant}_0 \\
\end{array} \right]
\end{align*}
\]

end

The first step of this tactic is to introduce a new variable which is used to strengthen
the postcondition. Then, this tactic strengthens the postcondition and calls the tactic `replConsByVar`.

**Example.** The following example derives a program which computes, given an array $f[0..n]$, the number of pairs $(i, j)$ for which

$$0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0$$

The formal specification of this program is given below.

$$r : [n \geq 0, r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\}]$$

The principle of the program is to use a variable $m$ to access each element of the array $f$. So, the range of $m$ is $0 \leq m < n$. We use the variable $s$ to count the number of non-positive elements in $f$, and the variable $r$ to count the number of non-negative elements which are found after a non-positive element. The variables $m$, $s$, and $r$ are initialized with 0. We use the tactic `strengInv` as seen below.

$$\begin{array}{c}
\lceil strengInv \\
( s : \mathbb{Z}, s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\}, \\
 m : \mathbb{Z}, n, 0 \leq m < n, (m, r, s := 0, 0, 0), n - m) \\
\lceil \text{varInt}(s : \mathbb{Z}) \\
\lceil \text{var} \ s : \mathbb{Z} \bullet \\
r, s : \big[( n \geq 0, \\
r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\} \big) \bigg] \end{array}$$

$$\begin{array}{c}
\lceil \text{strPost}(r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\} \\
\land s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\}) \\
r, s : \bigg[ \bigg( r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\} \bigg) \bigg] \\
\land \bigg( s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\} \bigg) \end{array}$$

This law generates the following proof obligation.

$$(r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
 s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\}) \Rightarrow \\
r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\}$$

Then, the tactic `strengInv` calls the tactic `replConsByVar` in order to introduce
the initialized iteration. The index of this iteration is the new variable $m$.

\[
\begin{align*}
\text{replConsByVar}(m : \mathbb{Z}, n, 0 \leq m < n, (m, r, s), (0, 0, 0))
\end{align*}
\]

\[
\begin{align*}
\text{varInt}(m : \mathbb{Z})
\end{align*}
\]

\[
\begin{align*}
\left[ \begin{array}{l}
\text{var } m : \mathbb{Z} \\
\text{r, s, m : }
\end{array} \right]
\end{align*}
\]

\[
\begin{align*}
\text{strPost}(r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\} \land
\hspace{1cm}
\text{s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\})[n \setminus m] \land \neg m \neq n)
\end{align*}
\]

This law generates the following proof obligation.

\[
\begin{align*}
(r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < n \land f[i] \leq 0 \land f[j] \geq 0\} \land
\hspace{1cm}
\text{s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\} \land \neg m \neq n)
\end{align*}
\]

The tactic replConsByVar introduces the iteration using the tactic takeConjAsInv, as follows.

\[
\begin{align*}
\text{takeConjAsInv}(0 \leq m \leq n, (m, r, s := 0, 0, 0), n - m)
\end{align*}
\]

\[
\begin{align*}
\text{strPost}(0 \leq m \leq n \land
\hspace{1cm}
\text{r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \land
\hspace{1cm}
\text{s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land \neg m \neq n)
\end{align*}
\]

This law generates the following proof obligation.

\[
\begin{align*}
(0 \leq m \leq n \land r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \land
\hspace{1cm}
\text{s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land \neg m \neq n)
\end{align*}
\]

First, the tactic takeConjAsInv introduces a sequential composition and the
initialization of the iteration.

\[
\begin{align*}
\lnot \text{seqComp}(0 \leq m \leq n & \land \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} & \land \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\rangle, \\
\langle r, s, m : & \begin{cases} \\
\qquad n \geq 0, \\
\qquad 0 \leq m \leq n \\
\qquad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} \\
\qquad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\rangle; \triangle \\
\langle r, s, m : & \begin{cases} \\
\qquad 0 \leq m \leq n \\
\qquad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} \\
\qquad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\quad \neg m \neq n \\
\rangle (i) \\
\rangle \text{assign}(m, r, s := 0, 0, 0) \\
\rangle m, s, r := 0, 0, 0
\end{cases}
\end{align*}
\]

This law generates the following proof obligation.

\[
n \geq 0 \Rightarrow \\
(0 \leq 0 \leq n \land 0 = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} \\
\land 0 = \#\{i : \mathbb{N} \mid 0 \leq i < 0 \land f[i] \leq 0\})
\]

Finally, the tactic \texttt{takeConjAsInv} introduces the iteration.

\[
\begin{align*}
(i) \quad \lnot \text{iter}((m \neq n), n - m) \\
\textbf{do} \quad m \neq n \rightarrow \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\quad \neg m \neq n \\
\rangle, \\
\langle r, s, m : & \begin{cases} \\
\qquad 0 \leq m \leq n \\
\qquad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} \\
\qquad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\rangle \triangle \\
\langle r, s, m : & \begin{cases} \\
\qquad 0 \leq m \leq n \\
\qquad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j \land f[i] \leq 0 \land f[j] \geq 0\} \\
\qquad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\quad 0 \leq n - m < n - m_0 \\
\rangle \textbf{od}
\end{cases}
\end{align*}
\]

We should now refine the body of the iteration in order to get the resulting program. First, we introduce the increment of the iteration index \( m \) to the end of
each iteration, and then, we split the specification into two parts.

\[
\begin{align*}
\forall fassign (m &:= m + 1) \\
& \begin{cases}
0 \leq m \leq n \\
r = \# \{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \\
s = \# \{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
m \neq n \\
r, s, m : \begin{cases}
0 \leq m + 1 \leq n \\
r = \# \{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
s = \# \{i : \mathbb{N} \mid 0 \leq i < m + 1 \land f[i] \leq 0\} \\
0 \leq n - (m + 1) < n - m_0
\end{cases}
\end{cases}; \triangleright
\end{align*}
\]

\[
\begin{align*}
\forall \text{segComp} (0 &\leq m \leq n \land \\
r = \# \{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
s = \# \{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n \\
& \begin{cases}
0 \leq m \leq n \\
r = \# \{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \\
s = \# \{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
m \neq n \\
r, s, m : \begin{cases}
0 \leq m \leq n \\
r = \# \{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
s = \# \{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
m \neq n \\
r, s, m : \begin{cases}
0 \leq m + 1 \leq n \\
r = \# \{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
s = \# \{i : \mathbb{N} \mid 0 \leq i < m + 1 \land f[i] \leq 0\} \\
0 \leq n - (m + 1) < n - m_0
\end{cases}
\end{cases}
\end{cases}; \triangleright
\end{align*}
\]

The first program of the composition is refined to an alternation which counts the number of pairs that satisfies the conditions seen in the description of this
example.

\[ \begin{array}{l}
\text{alt}((f[m] < 0, f[m] \geq 0)) \\
\text{if} \ f[m] < 0 \rightarrow \\
\left( \begin{array}{l}
f[m] < 0 \\
0 \leq m \leq n \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\}
\end{array} \right)
\right)
\]

\[ r, s, m : \\
\left( \begin{array}{l}
0 \leq m \leq n \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\quad m \neq n
\end{array} \right)
\]

\[ \left( f[m] \geq 0 \rightarrow \\
\left( \begin{array}{l}
f[m] \geq 0 \\
0 \leq m \leq n \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\}
\end{array} \right)
\right)
\]

\[ r, s, m : \\
\left( \begin{array}{l}
0 \leq m \leq n \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
\quad m \neq n
\end{array} \right)
\]

\[ \left( f[m] < 0 \lor f[m] \geq 0 \right)
\]

This law generates the following proof obligation.

\[
(0 \leq m \leq n \land \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n) \Rightarrow \\
(f[m] < 0 \lor f[m] \geq 0)
\]

If \( f[m] < 0 \) we must skip.

\[ \text{skipIntro()} \]

\[ \text{skip} \]

This law generates the following proof obligation.

\[
(f[m] < 0 \land 0 \leq m \leq n \land \\
\quad r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n) \Rightarrow \\
(0 \leq m \leq n \land r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n)
\]

If \( f[m] \geq 0 \) we must increment the number of pairs \( r \) by \( s \), the number of
non-positive number already found.

(iii) \( \sqsubseteq \) \text{assign}(r := r + s)

\[
\begin{align*}
  r := r + s;
\end{align*}
\]

This law generates the following proof obligation.

\[
\begin{align*}
  (f[m] \geq 0 \land 0 \leq m \leq n \land \\
  r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n) \Rightarrow \\
  (0 \leq m \leq n \land \\
  r + s = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n)
\end{align*}
\]

The second program of the sequential composition is refined to an alternation that counts the number of non-positive elements.

(ii) \( \sqsubseteq \) \text{alt}((f[m] > 0, f[m] \leq 0))

\[
\begin{align*}
  \text{if } f[m] > 0 & \rightarrow \\
  r, s, m : & \begin{cases} 
    f[m] > 0 \\
    0 \leq m \leq n \\
    r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
    s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
    m \neq n \end{cases} & (iv) \\
\end{align*}
\]

\[
\begin{align*}
  \text{if } f[m] \leq 0 & \rightarrow \\
  r, s, m : & \begin{cases} 
    f[m] \geq 0 \\
    0 \leq m \leq n \\
    r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \\
    s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \\
    m \neq n \end{cases} \\
\end{align*}
\]

\[
\begin{align*}
  \text{fi}
\end{align*}
\]

This application of law \text{alt} generates the proof obligation which can be seen
below.

\[(0 \leq m \leq n \land \quad \begin{aligned} 
  r &= \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s &= \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n\} \Rightarrow \\
  (f[m] < 0 \lor f[m] \geq 0) \end{aligned} \]

If \( f[m] > 0 \) we must skip.

\[\sqsubseteq \text{skipIntroIV()}\]

This law generates the following proof obligation.

\[(m = m_0 \land f[m] > 0 \land 0 \leq m \leq n \land \quad \begin{aligned} 
  r &= \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s &= \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n\} \Rightarrow \\
  0 \leq m + 1 \leq n \land \\
  r &= \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s &= \#\{i : \mathbb{N} \mid 0 \leq i < m + 1 \land f[i] \leq 0\} \land 0 \leq n - (m + 1) < n - m_0\} \end{aligned} \]

If \( f[m] \leq 0 \) we must increment the number of non-positive numbers \( s \) by one.

\[(iv) \sqsubseteq \text{assignIV}(s := s + 1)\]

\[s := s + 1\]

This law generates the following proof obligation.

\[(m = m_0 \land f[m] \leq 0 \land 0 \leq m \leq n \land \quad \begin{aligned} 
  r &= \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s &= \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land m \neq n\} \Rightarrow \\
  (0 \leq m + 1 \leq n \land \\
  r &= \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land \\
  s + 1 &= \#\{i : \mathbb{N} \mid 0 \leq i < m + 1 \land f[i] \leq 0\} \land \\
  0 \leq n - (m + 1) < n - m_0\} \end{aligned} \]

So, in this example, we can see that by applying the following tactic to the
original specification.

\textbf{tactic} \textit{strengInv}(s : \mathbb{Z}, s = \#\{i : \mathbb{N} \mid 0 \leq i < n \land f[i] \leq 0\},

m : \mathbb{Z}, n, 0 \leq m < n, (m, r, s := 0, 0, 0), n - m);

\textbf{var}\n
\textbf{var} skip
\textbf{do}
\textbf{law} fassign(m := m + 1);
\textbf{(}
\textbf{(law} seqComp
\quad (r = \#\{i, j : \mathbb{N} \mid 0 \leq i < j < m + 1 \land f[i] \leq 0 \land f[j] \geq 0\} \land
\quad s = \#\{i : \mathbb{N} \mid 0 \leq i < m \land f[i] \leq 0\} \land 0 \leq m \leq n \land m \neq n);\n\textbf{(law} alt((f[m] < 0, f[m] \geq 0));
\textbf{if} \textbf{law} skipIntro() \textbf{[] law} assign(r := r + s)\textbf{[]}
\textbf{(law} alt((f[m] > 0, f[m] \leq 0));
\textbf{if} \textbf{law} skipIntroIV() \textbf{[] law} assignIV(s := s + 1)\textbf{[]}
\textbf{)}
\textbf{skip}
\textbf{od})
\textbf{[]}

we get the following program.

\textbf{[var} s : \mathbb{Z} \textbf{] [var} m : \mathbb{Z} \textbf{]}
\textbf{m, r, s := 0, 0, 0;}
\textbf{do} m \neq n \rightarrow
\textbf{if} f[m] < 0 \rightarrow \textbf{skip;}
\quad \textbf{[} f[m] \geq 0 \rightarrow r := r + s\textbf{]}
\textbf{fi;}
\textbf{if} f[m] > 0 \rightarrow \textbf{skip;}
\quad \textbf{[} f[m] \leq 0 \rightarrow s := s + 1\textbf{]}
\textbf{fi;}
\quad m := m + 1
\textbf{od}\textbf{[]}

This program introduces two variables \(s\) and \(m\). The first one is used to count the number of non-positive numbers and the second one is used as an index in the
iteration. The variables $m$, $r$, and $s$ are initialized with 0. Finally, the program checks each element of the array and increments $r$ by $s$, if the element is non-negative, and $s$ by one if the element is non-positive.

### 2.2.6 TailInvariant

This strategy is used when we want to develop an algorithm involving an iteration whose invariant is based on a function defined using tail recursion.

This tactic applies to a specification statement $w : [pre, post]$ to introduce an initialized iteration whose invariant is $\text{invConj} \land \text{invBound}$ and variant is $\text{variant}$, and a final assignment $\text{ivar}_2 := \text{ival}_2$, which typically is an assignment of an element of a sequence identified by an index found in the iteration body, to a variable. The initialization of the iteration is $\text{ivar}_1 := \text{ival}_1$.

Tactic \texttt{tailInvariant}( newV\_1 : T\_1, newV\_2 : T\_2, invConj, guard, invBound, (ivar\_1 := ival\_1), variant, (ivar\_2 := ival\_2) )

applies to $w : [pre, post]$ do

\begin{verbatim}
  law varInt( newV\_1 : T\_1; newV\_2 : T\_2 );
  var law seqComp( invConj \land \neg guard );
  tactic takeConjAsInv( invBound, (ivar\_1 := ival\_1), variant )
  \[ law assign(ivar\_2 := ival\_2) \]
\end{verbatim}

proof obligations

$\text{invBound} \land \text{invConj} \land \neg \text{guard} \Rightarrow \text{invConj} \land \neg \text{guard}$,
(pre $\Rightarrow$ ( invBound $\land$ invConj )[ivar\_1 \ \text{ival\_1}]),
invConj $\land$ guard $\Rightarrow$ post[ivar\_2 \ \text{ival\_2}]

program generated

\begin{verbatim}
[ var newV\_1 : T\_1; newV\_2 : T\_2 •
  ivar\_1 := ival\_1;
  do guard →
    newV\_1, newV\_2, w : [invBound $\land$ invConj $\land$ guard, 
      invBound $\land$ invConj $\land$ 0 $\leq$ variant $<$ variant\_0]]
  od;
  ivar\_2 := ival\_2 ]
\end{verbatim}

end

This tactic has eight arguments: the first two arguments are variable declarations, which are used as indexing variables in the iteration. The next argument is a predicate $\text{invConj}$. It is used to make a conjunction with the next argument, $\text{guard}$, which is also a predicate and represents the guard of the iteration. The next
argument, \textit{invBound}, states the range limits of the invariant of the iteration. The assignment \texttt{ivar}_1 := \texttt{ival}_1 is the initialization of the iteration. An integer expression \textit{variant} is the next argument and represents the variant of the iteration. Finally, the final assignment \texttt{ivar}_2 := \texttt{ival}_2 is used after the end of the loop.

The tactic \texttt{tailInvariant} first introduces two variables (arguments \texttt{newV}_1 : \texttt{T}_1 and \texttt{newV}_2 : \texttt{T}_2). Afterwards, it splits the body of the variable block into a sequential composition of two other specifications. The first defines the initialized iteration and the second the final assignment. In sequence, the tactic \texttt{take ConjAsInv} is applied to the first program using \textit{invBound}, (\texttt{ivar}_1 := \texttt{ival}_1) and \textit{variant} as arguments. The tactic also applies the law \texttt{assign} to the second program using the argument (\texttt{ivar}_2 := \texttt{ival}_2). The first and the second proof obligations are generated by the tactic \texttt{take ConjAsInv}. The third proof obligation is generated by the law \texttt{assign}.

\textbf{Example.} In this example, we develop a program which identifies the maximum element of an array. The function below has such behavior and is used in the development.

\[
F(x,y:A) = \begin{cases} 
  A.x & \text{if } x = y \\
  F.(x+1).y & \text{if } A.x \leq A.y \\
  F.x.(y-1) & \text{if } A.x > A.y 
\end{cases}
\]

It is a recursive function which, given two integers \( x \) and \( y \), and an array \( A \), returns the maximum element of the set \( \{ i : \mathbb{N} | x \leq i \leq y \cdot A.i \} \).

The specification of our program is

\[
r : [n \geq 0, r = \text{max}\{ i : \mathbb{N} \mid 0 \leq i \leq n \cdot A.i \}]
\]

We start our development by strengthening the post condition in order to use the function definition presented above.

\[
\subseteq \text{strPost}(r = F.0.n) \\
r = [n \geq 0, r = F.0.n] \tag{1}
\]

This law generates the following proof obligation

\[
r = F.0.n \Rightarrow r = \text{max}\{ i : \mathbb{N} \mid 0 \leq i \leq n \cdot A.i \}
\]

Then, we use the tactic \texttt{tailInvariant}, which starts by introducing the variables
\(x\) and \(y\) used to calculate the function.

\[
\text{tailInvariant}(x : \mathbb{N}, y : \mathbb{N}, F.x.y = F.0.n \land 0 \leq x \leq y \leq n, x \neq y, \text{true},
(x, y := 0, n), y - x)
\]

\[
\text{varInt}(x : \mathbb{N}; y : \mathbb{N})
\]

\[
\begin{array}{ll}
\text{y, x, r : } & [n \geq 0, r = F.0.n] \\
\end{array}
\]

The next step is to split the specification into two programs.

\[
\text{seqComp}(F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y)
\]

\[
y, x, r : [n \geq 0, F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y] \quad <
\]

\[
y, x, r : [F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y, r = F.0.n] \quad (i)
\]

The first part of the program is refined to the initialized iteration using the tactic \textit{takeConjAsInv}.

\[
\text{takeConjAsInv(true, (x, y := 0, n), y - x)}
\]

\[
\text{strPost(true \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y)}
\]

\[
y, x, r : \begin{bmatrix}
    n \geq 0, \\
    \text{true} \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y
\end{bmatrix} \quad <
\]

This law generates the following proof obligation

\[(\text{true} \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y) \Rightarrow
(F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y)\]

The tactic \textit{takeConjAsInv} splits the specification into two parts. The first part is refined to the initialization of the variables used in the iteration to which the second part is refined.

\[
\text{seqComp(true} \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n)\]

\[
y, x, r : [n \geq 0, \text{true} \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n] \quad <
\]

\[
y, x, r : \begin{bmatrix}
    \text{true} \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \\
    \text{true} \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \lnot x \neq y
\end{bmatrix} \quad (ii)
\]

\[
\text{assign(x, y := 0, n)}
\]

\[
x, y := 0, n;
\]

This law generates the following proof obligation

\[n \geq 0 \Rightarrow (\text{true} \land F.0.n = F.0.n \land 0 \leq 0 \leq n \land n)\]

Then, in order to introduce the iteration, we use the tactic \textit{takeConjAsInv} as
seen below.

(ii) ⊑ \( \text{iter}(x \neq i, y - x) \)
    \[
    \begin{array}{l}
    \text{do } x \neq y \rightarrow \\
    y, x, r : [ \begin{align*}
    & (\text{true} \land F.x.y = F.0.n) \\
    & 0 \leq x \leq y \leq n \land x \neq y \\
    & (\text{true} \land F.x.y = F.0.n) \\
    & 0 \leq x \leq y \\
    & 0 \leq y - x < y_0 - x_0 \\
    \end{align*} ] \\
    \end{array}
    \]
    \[(iii)\]

\( \text{od} \)

Finally, the tactic \textit{tailInvariant} refines the second part of the program to the assignment \( r := A.x \).

(i) ⊑ \( \text{assign}(r := A.x) \)
    \[
    r := A.x
    \]

This law generates the following proof obligation

\( (F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land \neg x \neq y) \Rightarrow A.x = F.0.n \)

At this point, we finish the application of tactic \textit{tailInvariant}. Now, we refine the body of the iteration to an alternation as seen below.

(iii) ⊑ \( \text{alt}((A.x \leq A.y, A.x > A.y)) \)
    \[
    \begin{array}{l}
    \text{if } A.x \leq A.y \rightarrow \\
    y, x, r : [ \begin{align*}
    & A.x \leq A.y \\
    & \text{true} \land F.x.y = F.0.n \\
    & 0 \leq x \leq y \leq n \\
    & x \neq y \\
    & \text{true} \land F.x.y = F.0.n \\
    & 0 \leq x \leq y \\
    & 0 \leq y - x < y_0 - x_0 \\
    \end{align*} ] \\
    \end{array}
    \]
    \[(iv)\]

\[\text{fi}\]
This law generates the following proof obligation
\[
(true \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land x \neq y) \Rightarrow
(A.x \leq A.y \lor A.x > A.y)
\]

If \(A.x \leq A.y\) we should increment the auxiliary variable \(x\) by one. Otherwise, we should decrement the auxiliary variable \(y\) by one. We refine the first guarded command.

\[
\subseteq assignIV(x := x + 1)
\]
\[
x := x + 1
\]

This law generates the following proof obligation
\[
(A.x \leq A.y \land true \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land x \neq y) \Rightarrow
(true \land F.(x + 1).y = F.0.n \land
0 \leq x + 1 \leq y \leq n \land 0 \leq y - (x + 1) \leq y_0 - x_0)
\]

Now, we refine the second guarded command.

(iv) \(\subseteq assignIV(y := y - 1)\)
\[
y := y - 1
\]

This law generates the following proof obligation.
\[
(A.x > A.y \land true \land F.x.y = F.0.n \land 0 \leq x \leq y \leq n \land x \neq y) \Rightarrow
(true \land F.x.(y - 1) = F.0.n \land 0 \leq (y - 1) - x \leq y_0 - x_0)
\]

We get the following program.

\[
\begin{array}{l}
\text{var } x : \mathbb{N}; \ y : \mathbb{N} \bullet \\
x, y := 0, n; \\
\text{do } x \neq y \rightarrow \\
\quad \text{if } A.x \leq A.y \rightarrow x := x + 1 \\
\quad \quad \text{if } A.x > A.y \rightarrow y := y - 1 \\
\text{fi} \\
\text{od} \\
r := A.x;
\end{array}
\]

The strategies already presented do not have included procedure concepts. The next sections introduce tactics which work with procedures.
2.2.7 ProcNoArgs

Most of the laws in the refinement calculus are very basic. It is very common to use a set of them together. It is the case of the tactic procNoArgs which introduces a procedure block that declares a procedure with no parameters. It has two arguments: the name of the procedure which is introduced and its body. This tactic applies to a program \( p \), introduces the procedure, and makes one or more calls to it, depending on the program \( p \).

\[
\text{Tactic} \quad \text{procNoArgs}( \text{procName}, \text{procBody})
\]

applies to \( p \) do

law \( \text{procNoArgsIntro}( \text{procName}, \text{procBody}); \)

law \( \text{procNoArgsCall}( \) \)

program generated

\[
[p \text{procName} \overset{\text{\textbullet}}{=} \text{procBody} \cdot p[\text{procBody} \setminus \text{procName}]]
\]

end

The program \( p[\text{procBody} \setminus \text{procName}] \) is that obtained by replacing all occurrences of \( \text{procBody} \) in \( p \) with \( \text{procName} \).

Example. As an example of the use of the tactic procNoArgs we present again the development of the second example of Section 1.2. This example, derives a program which, given three numbers \( p, q, \) and \( r \), exchanges their values such that \( p \leq q \leq r \). The specification of this program is

\[
p, q, r : [p \leq q \leq r \land [p, q, r] = [p_0, q_0, r_0]]
\]

We have started the derivation by splitting the program in three assignments. The last assignment sorts \( p \) and \( q \).

\[
\sqsubseteq \text{fassign}(p, q := (p \cap q), (p \cup q))
\]

\[
p, q, r : [(p \cap q) \leq (p \cup q) \leq r \land [(p \cap q), (p \cup q), r] = [p_0, q_0, r_0]]; \quad \sqsubset
\]

\[
p, q := (p \cap q), (p \cup q)
\]

The second assignment sorts \( q \) and \( r \), and finally, the first assignment also sorts
\(p\) and \(q\).

\[\begin{align*}
\text{fassign}(q, r &:= (q \cap r), (q \cup r)) \\
q, r &:= (q \cap r), (q \cup r)
\end{align*}\]

\[\begin{align*}
\text{assignIV}(p, q &:= (p \cap q), (p \cup q)) \\
p, q &:= (p \cap q), (p \cup q)
\end{align*}\]

This law generates the following proof obligation.

\[p = p_0 \land q = q_0 \land r = r_0 \Rightarrow ((p \cap q) \cap ((p \cup q) \cap r)) \leq ((p \cap q) \cup ((p \cup q) \cap r)) \leq ((p \cup q) \cup r)\]

\[[(p \cap q) \cap ((p \cup q) \cap r)), ((p \cap q) \cup ((p \cup q) \cap r)), ((p \cup q) \cup r)] = [p_0, q_0, r_0]
\]

Now, we use the tactic \textit{procNoArgs} to introduce the procedure and use it in the body of the program.

\[\begin{align*}
\text{procNoArgs}(sort, (p, q &:= p \cap q, p \cup q)) \\
\text{procNoArgsIntro}(sort, (p, q &:= p \cap q, p \cup q)) \\
\text{procNoArgsCall}()
\end{align*}\]

So, by applying the following tactic to the initial specification.

\[\text{law fassign}(p, q := (p \cap q), (p \cup q));\]
\[\text{law fassign}(q, r := (q \cap r), (q \cup r))\]
\[\text{law assignIV}(p, q := (p \cap q), (p \cup q))\]
\[\text{tactic procNoArgsIntro}(sort, (p, q := p \cap q, p \cup q));\]
\[\text{tactic procNoArgsCall}()
\]

we get the following program.

\[\begin{align*}
\text{proc sort} &\equiv p, q := p \cap q, p \cup q \bullet \\
&\text{sort}; q, r := (q \cap r), (q \cup r); \text{sort}
\end{align*}\]

This program first calls the procedure \textit{sort} to sort \(p\) and \(q\). Then, it sorts \(q\) and \(r\). Finally, it calls again the procedure \textit{sort} to sort \(p\) and \(q\) again.
2.2.8 ProcCalls

Before describing a tactic which introduces procedures with arguments, it is useful to define a tactic that introduces parameterized commands. The tactic procCalls takes as arguments two lists. The first is a list of parameter declarations and the second is a list of arguments. The declarations have the form \texttt{pasMech v : T} where \texttt{pasMech} defines how the arguments are passed, by value(\texttt{val}), by result(\texttt{res}) or by value-result(\texttt{val-res}), \texttt{v} is the name of the argument, and \texttt{T} its type.

This tactic tries to derive an application of a procedure call with the given parameters to the given arguments. With this purpose, it tries to apply each of the laws that introduces parameterized commands. If one of them succeeds, the tactic goes on with the tail of the list, else, the tactic behaves like \texttt{skip} and finishes.

\begin{verbatim}
Tactic procCalls (pars, args)
appliesto w : [pre, post] do
  law callByValue(head' args, head' pars);
    \texttt{val}(tactic procCalls(tail pars, tail args)) |
  law callByValueIV(head' args, head' pars);
    \texttt{val}(law contractFrame(head' pars));
    \texttt{tactic procCalls(tail pars, tail args)) |
  law callByResult(head' args, head' pars);
    \texttt{res}(tactic procCalls(tail pars, tail args)) |
  law callByValueResult(head' args, head' pars);
    \texttt{val-res}(tactic procCalls(tail pars, tail args)) |
  skip
end
\end{verbatim}

The function \texttt{head'} applies to a list, and gives another list that contains just the head of the given list, or is empty if the given list is empty. The function \texttt{tail} returns the given list removing its first element. These functions \texttt{head'} and \texttt{tail} are not defined in \texttt{ArcAngel}'s semantics and are not supported by \texttt{Gabriel}. However, implementing these functions in \texttt{Gabriel} is an interesting task that is left as future work.

This tactic is recursive and can generate applications of parameterized commands whose bodies can include further applications. An example of the use of this tactic is presented in the next section.

2.2.9 ProcArgs

Now we can define the tactic \texttt{procArgs} which introduces a parameterized procedure in the scope of the program and makes calls to this procedure. Here the argu-
ment pars is a parameter declaration. We make use of the function seqToList to convert args, a semicolon-separated sequence of argument declarations, to a list of declarations.

Tactic procArgs ( procName, body, pars, args )
  applies to p do
    law  progArgsIntro(procName, body, pars);
    pmain
      tactic procCalls(seqToList pars, args);
      exhaust(law multiArgs())
  end
  law procArgsCall()

This tactic first introduces a procedure with arguments using the law procArgsIntro. Then the tactic uses procCalls to introduce applications of parameterized commands with parameters pars to arguments args. Finally, the tactic uses the law multiArgs to nested applications of parameterized commands. The definition of exhaust can be found in Section 2.3.4.

The function seqToList is also not defined in ArcAngel’s semantics. However, implementing this function in Gabriel is another interesting task that is left as future work.

Example. We use the square root program, presented in Section 1.2, as an example. Its specification is

\[ x : [0 \leq x, x^2 = x_0] \]

Our development is simplified by the use of tactic procArgs, which starts with the introduction of the sqrts procedure.

\[
x : [0 \leq x, x^2 = x_0]
\]
\[
\subseteq \text{procArgs}(\text{sqrts}, b : [0 \leq a, b^2 = a], (\text{val } a : \mathbb{R}; \text{res } b : \mathbb{R}), (x, x))
\subseteq \text{procArgsIntro}(\text{sqrts}, b : [0 \leq a, b^2 = a], (\text{val } a : \mathbb{R}; \text{res } b : \mathbb{R}))
\]
\[
[ \text{proc } \text{sqrts} \equiv (\text{val } a : \mathbb{R}; \text{res } b : \mathbb{R} \bullet b : [0 \leq a, b^2 = a]) \bullet
  x : [0 \leq x, x^2 = x_0] ] \quad <
\]

Then, using the tactic procCalls, this tactic attempts to introduce the substitu-
tions of the variable \( x \).

\[
\begin{align*}
\text{procCalls}(\langle \text{val } a : \mathbb{R}, \text{res } b : \mathbb{R} \rangle, \langle x, x \rangle) \\
\text{callByValueIV}(x, a) \\
\quad (\text{val } a : \mathbb{R} \cdot x, a : [0 \leq a, x^2 = a_0])(x) \quad \triangleleft \\
\text{contractFrame}(a) \\
\quad (\text{val } a : \mathbb{R} \cdot x : [0 \leq a, x^2 = a])(x) \quad \triangleleft \\
\text{procCalls}(\langle \text{res } b : \mathbb{R} \rangle, \langle x \rangle) \\
\quad \text{callByResult}(x, b) \\
\quad (\text{res } b : \mathbb{R} \cdot b : [0 \leq a, b^2 = a])(x) \quad \triangleleft \\
\end{align*}
\]

Then it uses the tactical \textit{exhaust} to put our two kinds of substitution together.

\[
\begin{align*}
\text{exhaust}(\text{law multiArgs()}) \\
\quad \text{multiArgs()} \\
\quad (\text{val } a : \mathbb{R}, \text{res } b : \mathbb{R} \cdot b : [0 \leq a, b^2 = a])(x, x) \quad \triangleleft \\
\end{align*}
\]

Finally, the tactic introduces the procedure call.

\[
\begin{align*}
\text{procArgsCall()} \\
\quad \text{sqrts}(x, x) \\
\end{align*}
\]

So, by applying the following tactic to the initial specification.

\[
\text{tactic procArgs(sqrts, (val } a : \mathbb{R}; \text{ res } b : \mathbb{R}, b : [0 \leq a, b^2 = a], \langle x, x \rangle) \\
\text{ we get the following program.}
\]

\[
\begin{align*}
\text{proc } \text{sqrts} \equiv (\text{val } a : \mathbb{R}, \text{res } b : \mathbb{R} \cdot b : [0 \leq a, b^2 = a]) \cdot \\
\text{sqrts}(x, x)
\end{align*}
\]

This program uses the procedure \textit{sqrts} to calculate the square root of a given number.

\subsection{RecProcArgs}

This tactic is useful when we want to develop a recursive procedure and have to
introduce a variant block with a parameterized procedure. Its definition is

\textbf{Tactic} \textit{recProcArgs}

\begin{verbatim}
( procName, body, variantName, variantExp, args, varsProcF )
\end{verbatim}

\textbf{applies to} \textit{p} do

\begin{verbatim}
  law  \text{variantIntro}(procName, body, variantName, variantExp, args);
  \text{pmainvariant}

  \textbf{tactic} \text{procCalls}(seqToList args, varsProcF);
  exhaust(law multiArgs())

  \text{p}

  law \text{procArgsVariantBlockCall}();
\end{verbatim}

end

This tactic first introduces a variant block using the law \textit{variantIntro}. Then, as in the previous tactic, it uses the law \textit{procCalls}. Finally, it uses the tactic \textit{exhaust(law multiArgs())} to introduce an application of a parameterized command. Afterwards, the tactic uses the law \textit{procArgsVariantBlockCall} to introduce a procedure call in the main program of the variant block.

\textbf{Example.}

We present the factorial program seen in Section 1.2 as an example. We have presented the following specification as the initial specification.

\begin{verbatim}
f : [f = n! * 1]
\end{verbatim}

We start by using the tactic \textit{recProcArgs}. The variant introduction is the first step of this tactic.

\begin{verbatim}
\{\textit{recProcArgs}
  \{\textit{fact}, V, m, (f : [f = m! * k]), \textit{val} m, k : \text{N}, \langle n, 1 \rangle\}
\} \textit{variantIntro} (\textit{fact}, V, m, (f : [f = m! * k]), \textit{val} m, k : \text{N})
  \[\begin{array}{l}
  \textit{proc} \textit{fact} \equiv (\textit{val} m, k : \text{N} \bullet f : \text{[V = m, f = m! * k]})
  \textit{variant} V \textit{is} m \bullet f : [f = n! * 1]
  \end{array}\]
\end{verbatim}

Then, using the tactic \textit{procCalls}, this tactic attempts to introduce a substitution
by value of the variables $m$ and $k$.

\[ \boxminus \text{procCalls}\langle \langle m, k \rangle, \langle n, 1 \rangle \rangle \]
\[ \boxminus \text{callByValue}\langle m, k : \mathbb{N}, \langle n, 1 \rangle \rangle \]
\[ \begin{array}{ll}
\text{[ proc } \text{ fact } \equiv (\text{val } m, k : \mathbb{N} \bullet f : [V = m, f = m! * k]) \\
\text{variant } V \text{ is } m \bullet \\
\text{(val } m, k : \mathbb{N} \bullet f : [f = m! * k])(n, 1) \\
\end{array} \]
\[ \triangle \]

The next step is to introduce the procedure call in the variant block.

\[ \boxminus \text{procArgsVariantBlockCall}() \]
\[ \begin{array}{ll}
\text{[ proc } \text{ fact } \equiv (\text{val } m, k : \mathbb{N} \bullet \{ V = m \} f : [f = m! * k]) \\
\text{variant } V \text{ is } m \bullet \\
\text{fact}(n, 1) \\
\end{array} \]

This finishes the tactic application. Now, we start to develop the body of the procedure fact. First, we introduce an alternation in order to check if we must stop the recursion ($m = 0$) or not ($m > 0$).

\[ \boxminus \text{alt}\langle \langle m = 0, m > 0 \rangle \rangle \]
\[ \begin{array}{ll}
\text{if } m = 0 \rightarrow f : [V = m \wedge m = 0, f = m! * k] \\
\text{[ } m > 0 \rightarrow f : [V = m \wedge m > 0, f = m! * k] \text{ (i)} \\
\text{fi} \\
\end{array} \]

This law application generates the following proof obligation.

\[ V = m \Rightarrow m = 0 \vee m > 0 \]

The base case is when we have $m$ equals to 0. In such case, $k$ must receive the value of the variable $f$.

\[ \boxminus \text{assign}(f := k) \]
\[ f := k \]

The following proof obligation is generated in this case.

\[ V = m \wedge m = 0 \Rightarrow k = 0! * k \]

If $m > 0$ we should make a recursive call. We first strengthen the postcondition.
in order to make a recursive call of procedure \textit{fact}.

(i) \begin{align*}
\text{strPost}(f &= (m - 1)! \ast m \ast k) \\
& f : [V = m \land m > 0, f = (m - 1)! \ast m \ast k] \text{ } \triangleright
\end{align*}

We have the following proof obligation

$$f = (m - 1)! \ast m \ast k \Rightarrow f = m! \ast k$$

Then we introduce a substitution by value and weaken the precondition.

\begin{align*}
\text{callByValue}((m, k), (m - 1, m \ast k)) \\
& (\text{val } m, k : \mathbb{N} \bullet f : [V = m + 1 \land m + 1 > 0, f = m! \ast k]) \\
& (m - 1, m \ast k) \text{ } \triangleright
\end{align*}

\begin{align*}
\text{weakPre}(0 \leq m < V) \\
& f : [0 \leq m < V, f = m! \ast k]) \text{ } \triangleright
\end{align*}

This law generates the following proof obligation

$$V = m + 1 \land m + 1 > 0 \Rightarrow 0 \leq m < V$$

Now we have the following program.

\begin{align*}
\begin{array}{c}
\text{proc } fact \\= (\text{val } m, k : \mathbb{N} \bullet) \\
& \text{if } m = 0 \rightarrow f := k \\
& \text{m} > 0 \rightarrow f : [0 \leq m < V, f = m! \ast k] \\
& \text{fi}
\end{array} \\
\text{variant } V \text{ is } m \\
\text{fact}(n, 1)\]
\end{align*}

Finally, we refine this program by introducing a recursive call to the procedure \textit{fact} as follows.

\begin{align*}
\text{recursiveCall}() \\
\begin{array}{c}
\text{proc } fact \\= (\text{val } m, k : \mathbb{N} \bullet) \\
& \text{if } m = 0 \rightarrow f := k \\
& \text{m} > 0 \rightarrow \text{fact}(m - 1, m \ast k) \\
& \text{fi}
\end{array} \\
\text{variant } V \text{ is } m \\
\text{fact}(n, 1)\]
\end{align*}

The last application generates the following proof obligation.

$$f : [V = n, f = m! \ast k]$$

\begin{align*}
\begin{array}{c}
\text{if } m = 0 \rightarrow f := k \\
& \text{m} > 0 \rightarrow f : [0 \leq m < V, f = m! \ast k] \\
& \text{fi}
\end{array}
\end{align*}

The program obtained calculates the factorial of a given number.
2.3 ArcAngel’s Semantics

In this section we present ArcAngel’s semantics which is based in Angel semantics. It, however, takes into account the particularities of the refinement calculus, in which a transformation rule consists of refinement laws. The application of a refinement law to a program and returns a new program, but also proof obligations.

Tactics are applied to a pair: the first element of this pair is a program to which the tactic is applied; the second element is the set of proof obligations generated to obtain this program. This pair is called \( RCell \) (refinement cell), and is defined as follows:

\[
RCell == \text{program} \times \mathbb{P}\text{predicate}
\]

The result of a tactic application is a possibly infinite list of \( RCells \) that contains all possible outcomes of its application: every program it can generate, together with the corresponding proof obligations (existing obligations and those generated by the tactic application). Different possibilities arise from the use of alternation, and the list can be infinite, since the application of a tactic may run indefinitely. If the application of some tactic fails, then the empty list is returned.

\[
Tactic == RCell \rightarrow pfiseq RCell
\]

The type \( pfiseq RCell \) is that of possibly infinite lists of \( RCells \). We use the model for infinite lists proposed in [19]. This is summarized in Appendix B.

In order to give semantics to named laws and tactics, we need to maintain two appropriate environments.

\[
LEnv == \text{name} \rightarrow \text{seq argument} \rightarrow \text{program} \rightarrow \text{RCell}
\]

\[
TEnv == \text{name} \rightarrow \text{seq argument} \rightarrow \text{Tactic}
\]

A law environment records the set of known laws; it is a partial function whose domain is the set of the names of these laws. For a law environment \( \Gamma_L \) and a given law name \( n \), we have that \( \Gamma_L n \) is also a partial function: it relates all valid arguments of \( n \) to yet another function. For a valid argument \( a \), we have that \( \Gamma_L n a \) relates all the programs to which \( n \) can be applied when given arguments \( a \); the result is a refinement cell.

For example, let \( \Gamma_L \) be an environment that records the law \( strPost \). For a predicate \( post' \) and a program \( w : [pre, post] \) we have

\[
\Gamma_L strPost post' ( w : [pre, post]) = ( w : [pre, post'], \{ post' \Rightarrow post \})
\]

This means that if we apply \( strPost \) with argument \( post' \) to \( w : [pre, post] \), we change its postcondition to \( post' \), but we must prove that \( post' \Rightarrow post \). We are
interested in environments that record at least the laws of Morgan’s refinement calculus in [23] and in Appendix C.

Similarly, a tactic environment is a function that takes a tactic name and a list of arguments, and returns a Tactic.

### 2.3.1 Tactics

We define the semantic function for tactics inductively; it has the type

\[
\llbracket - \rrbracket : \text{tactic} \rightarrow \text{LEnv} \rightarrow \text{TEnv} \rightarrow \text{Tactic}
\]

The basic tactic law \( n(a) \) is that which applies a simple law to an RCell.

\[
\llbracket \text{law } n(a) \rrbracket \Gamma_L \Gamma_T (p, pobs) =
\]

\[
\text{if } n \in \text{dom } \Gamma_L \land a \in \text{dom } (\Gamma_L n) \land p \in \text{dom } (\Gamma_L n a)
\]

\[
\text{then } \left\{ \begin{array}{l}
\text{let } (\text{newp}, \text{npobs}) = \Gamma_L n a p \text{ in } [(\text{newp}, \text{pobs} \cup \text{npobs})]
\end{array} \right.
\]

\[
\text{else } []
\]

We check if the law name \( n \) is in the law environment \( \Gamma_L \), and if the arguments \( a \) and program \( p \) are appropriate. If these conditions hold, then the tactic succeeds, and returns a list with a new RCell. The program is transformed by applying the law to the program \( p \); the new proof obligations are added to the proof obligations \( pobs \) of the original RCell. Otherwise, the tactic fails with the empty list as result.

We use angle brackets to delimit finite lists; possibly infinite lists are delimited by square brackets.

In this work, we use a simple approach for expression arguments. For us, they are used as they were already evaluated. However, they should be evaluated before being used. This is left as future work.

The semantics of tactic \( n(a) \) is similar to that of the law construct. Its definition is

\[
\llbracket \text{tactic } n(a) \rrbracket \Gamma_L \Gamma_T r =
\]

\[
\text{if } n \in \text{dom } \Gamma_T \land a \in \text{dom } \Gamma_T n
\]

\[
\text{then } \Gamma_T n a r
\]

\[
\text{else } []
\]

If the tactic is in the tactic environment, and if the arguments are valid, then the tactic succeeds; it returns the result of applying the tactic to the arguments. Otherwise, the tactic fails.
The tactic skip returns its argument unchanged; the tactic fail always fails.

\[
\begin{align*}
\text{\texttt{skip}} & \Gamma_L \Gamma_T r = [r] \\
\text{\texttt{fail}} & \Gamma_L \Gamma_T r = []
\end{align*}
\]

The sequence operator applies its first tactic to its argument, producing a list of cells; it then applies the second tactic to each member of this list; finally, this list-of-lists is flattened to produce the result.

\[
\begin{align*}
\text{\texttt{[ [ t_1 ; t_2 ]]} \Gamma_L \Gamma_T} & = \infty/ \cdot (\text{\texttt{[ [ t_2 ]]} \Gamma_L \Gamma_T})^* \cdot (\text{\texttt{[ [ t_1 ]]} \Gamma_L \Gamma_T})
\end{align*}
\]

For a total function \(f : A \rightarrow B\), \(f^* : \text{pfiseq} A \rightarrow \text{pfiseq} B\) is the map function that operates on a list by applying \(f\) to each of its elements; the operator \(\cdot\) is used to compose functions; and \(\infty/\) is the distributed concatenation operation. Formal definitions of these operators and others to follow can be found in Appendix B.

The semantics of the alternation operator is given by concatenation: the possible outcomes of each individual tactic are joined to give the list of possible outcomes of the alternation.

\[
\begin{align*}
\text{\texttt{[ [ t_1 \mid t_2 ]]} \Gamma_L \Gamma_T} & = \infty/ \cdot ((\text{\texttt{[ [ t_1 ]]} \Gamma_L \Gamma_T}), (\text{\texttt{[ [ t_2 ]]} \Gamma_L \Gamma_T}))^o
\end{align*}
\]

The function \(^o\) applies a list of functions to a single argument and returns a list containing the results of the applications.

The cut operator applies its tactic to its argument, taking the first result (if it exists) and discarding the rest; if there is no first result, then the cut tactic fails.

\[
\begin{align*}
\text{\texttt{[ ! t ]]} \Gamma_L \Gamma_T & = \text{head}' \cdot (\text{\texttt{[ [ t ]]} \Gamma_L \Gamma_T})
\end{align*}
\]

The recursion operator \(\mu\) has a standard definition [9] as a least fixed point. For a continuous function \(f\) from tactics to tactics, we have that

\[
\begin{align*}
(\mu X \bullet f(X)) & = \bigsqcup\{ i : \mathbb{N} \bullet f^i(\text{abort}) \}
\end{align*}
\]

where \(f^i\) represents \(i\) applications of \(f\).

This definition makes sense if the set of tactics is a complete lattice. First, we define a partial order for lists: the completely-undefined list, denoted \(\bot\), is the least-defined element in the set. It is a partial list, as is any list that ends with \(\bot\). One list is less than another, \(s_1 \sqsubseteq s_2\), whenever they are equal, or the first is a partial list that forms an initial subsequence of the second. In the model we adopt, an infinite list is a limit of a directed set of partial lists.

For tactics, as they are partial functions, we may lift the ordering on lists of
Cells to an ordering on tactics. We can say that
\[ t_1 \sqsubseteq_T t_2 \iff (\forall r : RCell \bullet t_1 r \sqsubseteq \infty t_2 r) \]

As the set of lists of RCells is a complete lattice, the set of tactics is also a complete lattice, using the order above [9].

The bottom element is used in the semantics of abort, which runs indefinitely.

\[ \llbracket \text{abort} \rrbracket \Gamma_L \Gamma_T = \bot \]

The tactics succs t and fails t are defined as follows

\[
\begin{align*}
\text{succs } t r &= (\text{if } t r = \bot \text{ then abort else } (\text{if } t r = [] \text{ then fail else skip}) r) \\
\text{fails } t r &= (\text{if } t r = \bot \text{ then abort else } (\text{if } t r = [] \text{ then skip else fail}) r)
\end{align*}
\]

If the application of t aborts, so do succs t and fails t. If it fails, then succs t fails and fails t skips. Finally, if it succeeds, succs t skips and fails t fails.

### 2.3.2 Structural Combinators

The structural combinators apply tactics to components of a program independently (and so can be thought of as in parallel), and then reassemble the results in all possible ways. There is one combinator for each construct in the programming language.

As already explained, the structural combinator \( t_1 \parallel t_2 \) applies to a sequential composition \( p_1; p_2 \). Independently, \( t_1 \) is applied to \( p_1 \) and \( t_2 \) is applied to \( p_2 \); the resulting alternatives are assembled into pairs, so that the first is an alternative outcome from \( t_1 \) and the second is an alternative outcome from \( t_2 \); finally, these pairs are combined with the sequential composition tactical.

\[
\llbracket t_1 \parallel t_2 \rrbracket \Gamma_L \Gamma_T (p_1; p_2, pobs) = \Omega_* \left( \Pi (\langle \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, pobs), (\llbracket t_2 \rrbracket \Gamma_L \Gamma_T (p_2, pobs)) \rangle) \right)
\]

The distributed cartesian product operator \( \Pi \) is defined in Appendix B. The sequential combination function \( \Omega_* \) sequentially composes the programs and unites the proof obligations. The function \( \Omega_* \) is a partial function because it is defined only for lists with two elements.

\[
\Omega_* : RCell^* \rightarrow RCell
\]

\[
\Omega_* ((p_1, pobs_1), (p_2, pobs_2)) = (p_1; p_2, pobs_1 \cup pobs_2)
\]

The semantics of the structural combinators \( \text{if } - \text{fi} \) and \( \text{do } - \text{od} \) use a few functions
which are explained and defined as they are needed; we start by giving a simple example. Consider the $RCell$

\[
\text{if } x \geq 0 \rightarrow x : [x \geq 0, x > y] \| x < 0 \rightarrow x : [x < 0, x < z] \text{ fi, } \{x \geq 1\}
\]

and the tactic

\[
\text{if } \text{law weakPre(true)} \| \text{law assign}(x := x + 1) \| \text{law assign}(x := x - 1) \text{fi}
\]

First of all, we extract the commands from each branch of the conditional; the function $extractP$ does this for us.

\[
extractP :: \text{GuardedCommand} \rightarrow \text{pfseq program}
\]

\[
extractP G \rightarrow P = \langle P \rangle
\]

\[
extractP G \rightarrow P [] GC = P : (extractP GC)
\]

In our example we have

\[
extractP(x \geq 0 \rightarrow x : [x \geq 0, x > y] \| x < 0 \rightarrow x : [x < 0, x < z])
\]

\[
= \langle x : [x \geq 0, x > y], x : [x < 0, x < z] \rangle
\]

The function $\Phi_r$ constructs an $RCell$ with a given program $p$ and an empty set of proof obligations. We map $\Phi_r$ over our list of commands.

\[
\Phi_r :: \text{program} \rightarrow \text{RCell}
\]

\[
\Phi_r p = (p, \emptyset)
\]

In our example we have

\[
\Phi_r \ast (\langle x : [x \geq 0, x > y], x : [x < 0, x < z] \rangle) =
\]

\[
\langle (x : [x \geq 0, x > y], \emptyset), (x : [x < 0, x < z], \emptyset) \rangle
\]

Now, we need to apply each tactic to its corresponding command. We use the function $apply$ that takes two lists: the first is a list of functions, and the second is a list of elements to which the functions are applied; it returns the list of the results of applying each function in the first list to the corresponding element in the second list. In our example we have

\[
apply(\text{law weakPre(true)} \| \text{law assign}(x := y + 1), \text{law assign}(x := z - 1))
\]

\[
\langle (x : [x \geq 0, x > y], \emptyset), (x : [x < 0, x < z], \emptyset) \rangle
\]

\[
= \langle \text{law weakPre(true)} \| \text{law assign}(x := y + 1) (x : [x \geq 0, x > y], \emptyset),
\]

\[
\text{law assign}(x := z - 1) (x : [x < 0, x < z], \emptyset) \rangle
\]

\[
= \langle [(x : [x > y], \{x \geq 0 \Rightarrow true\}), (x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\})],
\]

\[
[(x := z - 1, \{x < 0 \Rightarrow z - 1 < z\})] \rangle
\]

Now, we have to take the distributed cartesian product of this list to get all the
possible combinations of the cells of the first list with the cells of the second list.

$$\Pi([((x : [x > y], \{x \geq 0 \Rightarrow true\}), (x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\})),
[(x := z - 1, \{x < 0 \Rightarrow z - 1 < z\})])]
= [(x : [x > y], \{x \geq 0 \Rightarrow true\}), (x := z - 1, \{x < 0 \Rightarrow z - 1 < z\})),
((x := y + 1, \{x \geq 0 \Rightarrow y + 1 > y\})), (x := z - 1, \{x < 0 \Rightarrow z - 1 < z\}))]$$

The function $extractG$ makes a list of guards of the guarded command.

$$extractG :: GuardsComands \rightarrow pfseq\ Guard
extractG\ G \rightarrow P = \langle G \rangle
extractG\ G \rightarrow P \parallel GC = G \cdot (extractG\ GC)$$

In the example we have

$$extractG\ (x \geq 0 \rightarrow x : [x \geq 0, x = x_0 + 1] \parallel x < 0 \rightarrow x : [x < 0, x = x_0 - 1]) =
\langle x \geq 0, x < 0 \rangle$$

We combine the list of guards with each element of the list of $RCells$ we get from the
distributed cartesian product above. The function $insertG$ takes a list of guards $g_i$, a list of $RCells$ ($p_i, pobs_i$), and returns a pair, where the first element is a guarded
command and the second is a set of proof obligations. The guarded command
associates each guard $g_i$ to the program $p_i$: it is $g_1 \rightarrow p_1 \parallel g_2 \rightarrow p_2 \parallel ...$. The set of
proof obligations is the union of the $pobs_i$.

$$insertG :: pfseq\ Guard \rightarrow pfseq\ RCell \rightarrow pfiseq(GuardedCommand, \mathbb{P}\ predicate)
insertG\ \langle \rangle\ (rs) = [\]
insertG\ gs\ [\] = [\]
insertG\ (g : gs)\ ((p, pobs) : rs) = tg\ (g \rightarrow (fst\ r),\ snd\ r)\ (insertG\ gs\ rs)$$

where

$$tg\ (g_1 \rightarrow p_1, pr_1)\ (g_2 \rightarrow p_2, pr_2) = (g_1 \rightarrow p_1 \parallel g_2 \rightarrow p_2, pr_1 \cup pr_2)$$

We use the function $mkGC$ to apply $insertG\ \langle x \geq 0, x < 0 \rangle$ to each element of
the cartesian product above.

\[ mkGC :: \text{pfseq Guard} \rightarrow \text{pfseq(pfiseq RCell)} \rightarrow \text{pfiseq(GuardedCommand, P predicate)} \]

\[ mkGC \ gs = (\text{insertG gs})* \]

In our example we have

\[ mkGC \langle x \geq 0, x < 0 \rangle \]

\[ = \langle x \geq 0 \rightarrow x : [x \geq 1] \] \[ ] x < 0 \rightarrow x := z - 1, \]

\[ \{ x \geq 0 \Rightarrow true, x < 0 \Rightarrow z - 1 < z \} \rangle, \]

\[ (x \geq 0 \rightarrow x := y + 1 ] x < 0 \rightarrow x := z - 1, \]

\[ \{ x \geq 0 \Rightarrow y + 1 > y, x < 0 \Rightarrow z - 1 < z \} \rangle \]

The last step, which rebuilds the RCells, uses the function \( \Omega_i \). The arguments of this function are the original set \( \text{pobs} \) of proof obligations and the list of RCells \( (gc_i, pobs_i) \) generated in the previous step. The result is the list of RCells \( (\text{if } gc_i \ text{fi}, \text{pobs} \cup \text{pobs}_i) \); each guarded command in the argument list is turned into a conditional, and the set of original proof obligations is added to those generated by the tactic.

\[ \Omega_i :: \text{P predicate} \rightarrow \text{pfiseq(GuardedCommands, P predicate)} \rightarrow \text{pfiseq(RCell)} \]

\[ \Omega_i \ ops = (\text{if } \ ops)* \]

where

\[ \text{if} :: \text{P predicate} \rightarrow (\text{GuardedCommands, P predicate}) \rightarrow \text{RCell} \]

\[ \text{if} \ ops \ (gc, nps) = (\text{if } gc \ \text{fi}, \text{pobs} \cup \text{nps}) \]

In our example we have

\[ \Omega_i \{ x \geq 1 \}

\[ = \langle (\text{if } x \geq 0 \rightarrow x : [x \geq 1] \] \[ ] x < 0 \rightarrow x := z - 1 \text{ fi}, \]

\[ \{ x \geq 0 \Rightarrow true, x < 0 \Rightarrow z - 1 < z, x \geq 1 \} \rangle, \]

\[ (\text{if } x \geq 0 \rightarrow x := y + 1 ] x < 0 \rightarrow x := z - 1 \text{ fi}, \]

\[ \{ x \geq 0 \Rightarrow y + 1 > y, x < 0 \Rightarrow z - 1 < z, x \geq 1 \} \rangle \]

With this example as motivation, we present the definition of the combinator for
the conditional.

\[
\begin{align*}
( \llbracket \text{tacs} \rrbracket \Gamma_L \Gamma_T ) ( \text{if } gc \ \text{fi}, pobs ) &= \\
\quad \Omega_g \ pobs \ (mkGC \ (extractG \ gc) \\
\quad (\Pi (apply (\llbracket - \rrbracket \Gamma_L \Gamma_T ) * \text{tacs}) (\Phi_r * (extractP \ gc))))
\end{align*}
\]

Firstly, we extract the commands from the branches \( gc \) of the conditional \( (extractP) \). Secondly, we construct a list of \( RCells \) with these commands as their programs, and an empty set of proof obligations \( (\Phi_r, \text{?}) \). We apply each element of the list \( \text{tacs} \) of tactics to the corresponding element in the list of \( RCells \) we constructed \( (apply (\llbracket - \rrbracket \Gamma_L \Gamma_T ) * \text{tacs}) \). The next step is to take the distributed cartesian product of the resulting list \( (\Pi) \). Finally, we rebuild the conditionals with the resulting commands \( (mkGC \ (extractG \ gc)) \) and add the original proof obligations \( pobs \) to the new sets of proof obligations \( (\Omega_g \ pobs) \).

The similar definition of the structural combinator \( \text{do} \) is

\[
\begin{align*}
( \llbracket \text{do} \text{tacs} \text{od} \rrbracket \Gamma_L \Gamma_T ) ( \text{do } gc \ \text{od}, pobs ) &= \\
\quad \Omega_d \ pobs \ (mkGC \ (extractG \ gc) \\
\quad (\Pi (apply (\llbracket - \rrbracket \Gamma_L \Gamma_T ) * \text{tacs}) (\Phi_r * (extractP \ gc))))
\end{align*}
\]

The function which generates the list of \( RCells \) for the structural combinator \( \text{do} \) is \( \Omega_d \). Its definition is

\[
\Omega_d :: \text{P predicate} \rightarrow \text{pfiseq(GuardedCommands, P predicate)} \rightarrow \text{pfiseq(RCell)}
\]

\[
\Omega_d \ ops = (\text{do} \ ops) *
\]

where

\[
\text{do} :: \text{P predicate} \rightarrow (\text{GuardedCommands, P predicate}) \rightarrow \text{RCell}
\]

\[
\text{do} \ ops \ (gc, nps) = (\text{do } gc \ \text{od}, \text{ops} \cup nps)
\]

The structural combinator \( \text{var} \) applies a tactic to the program in a variable block and rebuilds the variable blocks. Its formal definition is

\[
\begin{align*}
( \llbracket \text{var} \rrbracket \Gamma_L \Gamma_T ) ( \llbracket \text{var} \ d \bullet p \rrbracket , pobs ) &= \\
\quad (\text{var} \ d) * (\llbracket t \rrbracket \Gamma_L \Gamma_T (p, pobs))
\end{align*}
\]

The function \( \text{var} \) rebuilds the variable block after the application of the tactic to its body. It takes as arguments a variable declaration and an \( RCell \). The result is a new \( RCell \) containing a variable block built from the declaration and the program. The proof obligations are not changed.

\[
\text{var} : \text{declaration} \rightarrow \text{RCell} \rightarrow \text{RCell}
\]

\[
\text{var} \ d \ (p, pobs) = (\llbracket \text{var} \ d \bullet p \rrbracket , pobs)
\]

The structural combinator \( \text{con} \) is defined similarly as are all others struc-
tural combinators that handle procedure and variant blocks and parameters. Its definition is

\[
\text{con} \colon \text{declaration} \rightarrow \text{RCell} \rightarrow \text{RCell}
\]

\[
\text{con} \ d \ (p, \text{proofs}) = (\text{con} \ d) \ast (\text{con} \ d \bullet p, \text{proofs})
\]

where

\[
\text{con} \colon \text{declaration} \rightarrow \text{RCell} \rightarrow \text{RCell}
\]

\[
\text{con} \ d \ (p, \text{proofs}) = (\text{con} \ d \bullet p, \text{proofs})
\]

This function is similar to \text{var}, as are those below.

The structural combinator \text{pmain} is used to apply a tactic to the main program of a procedure block. Its definition is as follows.

\[
(\Gamma \ L \ T (\text{pobs})) = (\text{procm} \ n \ p_1) \ast (\text{procm} \ n \ p_2, \text{proofs})
\]

where

\[
\text{procm} : \text{name} \rightarrow \text{program} \rightarrow \text{RCell} \rightarrow \text{RCell}
\]

\[
\text{procm} \ n \ p_1 \ (p_2, \text{proofs}) = (\text{procm} \ n \ p_1 \bullet p_2, \text{proofs})
\]

The structural combinator \text{pmainvariant} is used to apply a tactic to the main program of a variant block. Its definition is shown below

\[
(\Gamma \ L \ T (\text{pobs})) = (\text{variant} \ p \ n \ v \ e \ p, \text{proofs})
\]

where

\[
\text{variant} : \text{name} \rightarrow \text{program} \rightarrow \text{variable} \rightarrow \text{expression} \rightarrow \text{RCell} \rightarrow \text{RCell}
\]

\[
\text{variant} \ p \ n \ v \ e \ p, \text{proofs} = (\text{variant} \ p \ n \ v \ e \ p, \text{proofs})
\]

The structural combinator \text{pbody} is used to apply a tactic to the procedure body of a procedure block. Its definition is as follows.

\[
(\Gamma \ L \ T (\text{body})) = (\text{procb} \ n \ p, \text{proofs})
\]

where

\[
\text{procb} : \text{name} \rightarrow \text{program} \rightarrow \text{RCell} \rightarrow \text{RCell}
\]

\[
\text{procb} \ n \ p, \text{proofs} = (\text{procb} \ n \ p, \text{proofs})
\]

The structural combinator \text{pbodyvariant} is used to apply a tactic to the
procedure body of a variant block. Its definition is shown below.

\[
\begin{align*}
\text{pbodyvariant} &\colon t \Gamma_L \Gamma_T \\
\text{proc } &\text{name } \equiv \text{body } \text{variant } \text{vis } e \bullet p, \text{proofs} = \\
& (\text{variantb } \text{name } v e p) \ast (\Gamma t \Gamma_L \Gamma_T (\text{body}, \text{proofs}))
\end{align*}
\]

where

\[
\begin{align*}
\text{variantb} &\colon \text{name } \to \text{variable } \to \text{expression } \to \text{program } \to \text{RCell } \to \text{RCell} \\
\text{variantb } \text{name } v e \text{main} (\text{body}, \text{proofs}) &= \\
& (\text{proc } \text{name } \equiv \text{body } \text{variant } v \text{is } e \bullet \text{main}, \text{proofs})
\end{align*}
\]

The structural combinator \text{pbodymain} and the structural combinator \text{pmainvariantbody} apply to procedure blocks and variant blocks, respectively. They apply \text{t} to the body of the procedure, and \text{t} to the main program. Their definitions are

\[
\begin{align*}
\text{pbodymain} &\colon t_b \text{ t} \Gamma_L \Gamma_T \\
\text{pmainvariantbody} &\colon t_b \text{ t} \Gamma_L \Gamma_T
\end{align*}
\]

and

\[
\begin{align*}
\text{pbodymain} (\text{t} \text{ t} \Gamma_L \Gamma_T) &= \text{pbody} \text{ t} \text{ t} \Gamma_L \Gamma_T \\
\text{pmainvariantbody} (\text{t} \text{ t} \Gamma_L \Gamma_T) &= \text{pbodyvariant} \text{ t} \text{ t} \Gamma_L \Gamma_T
\end{align*}
\]

When we are not concerned with the type of argument declaration we use the structural combinator \text{parcommand}. Its definition is

\[
\begin{align*}
\text{parcommand} &\colon t \Gamma_L \Gamma_T (\text{pars} \bullet p, \text{proofs}) = \\
& (\text{parcommand } \text{pars}) \ast (\Gamma t \Gamma_L \Gamma_T (p, \text{proofs}))
\end{align*}
\]

where

\[
\begin{align*}
\text{parcommand} &\colon \text{declaration } \to \text{RCell } \to \text{RCell} \\
\text{parcommand } \text{pars} (p, \text{proofs}) &= ((\text{pars} \bullet p), \text{proofs})
\end{align*}
\]

For parameter passing, we have first the semantics of \text{val}. Its definition is

\[
\begin{align*}
\text{val} &\colon \text{name } \to \text{Type } \to \text{args } \to \text{RCell } \to \text{RCell} \\
\text{val } v T a (p, \text{proofs}) &= ((\text{val } v T \bullet p)(a), \text{proofs})
\end{align*}
\]

where

\[
\begin{align*}
\text{val} &\colon \text{name } \to \text{Type } \to \text{args } \to \text{RCell } \to \text{RCell} \\
\text{val } v T a (p, \text{proofs}) &= ((\text{val } v T \bullet p)(a), \text{proofs})
\end{align*}
\]

This structural combinator applies the tactic to the body of the parameterized
command and then rebuilds it, using the function \( \text{val} \).

The structural combinator \( \text{res} \) is used when we have an argument passed by result in the program of an \( \text{RCell} \). Its definition is shown below.

\[
\text{res} : \text{variable} \rightarrow \text{type} \rightarrow \text{pfseq} \text{variable} \rightarrow \text{RCell} \\
\text{res} v T \text{pars} (p, \text{proofs}) = ((\text{res} v : T \bullet p)(\text{pars}), \text{proofs})
\]

The tactic \( \text{con} \) introduces \( v \) as a set of variables whose values are taken from an appropriate syntactic class denoted \( \text{TERM} \) below. These variables are used in \( t \) as elements of its syntactic class. Their values are angelically chosen so that the tactic succeeds. The semantics is 

\[
\| \text{con} v \bullet t \|_{\Gamma_L \Gamma_T} = \| v \in \text{TERM} t(v) \|_{\Gamma_L \Gamma_T}
\]

where \( v \in \text{TERM} t(v) \) is an alternation of all tactics that can be obtained from \( t \), with \( v \) ranging over all values in \( \text{TERM} \). This is an infinite alternation, which can be defined as follows.

\[
\big|_{i=0}^{\infty} f(i) = \mu X \bullet F(X_0)
\]

where \( F(X_i) = f(i) | F(X_{i+1}) \). This definition generates the following alternation.

\[
f(0) | f(1) | \ldots | f(n)
\]

The tactic \( \text{applysto} \ p \ \text{do} \ t \) needs to consider all ways in which the given program can match \( p \). Its semantics uses the function \( \text{equals} \), that yields a tactic
that succeeds only if it is presented with a \( RCell \) matching the given program.

\[
equals : RCell \rightarrow Tactic
\]

\[
equals r r = [r]
\]

\[
equals r h = [], \text{ if } h \neq r
\]

The definition of the tactic \texttt{applies to } \( p \) do \( t \) is

\[
\llbracket \text{applies to } p \text{ do } t \rrbracket \Gamma_L \Gamma_T (p_1, pobs) =
\llbracket \text{con } p \bullet \text{equals } (p, pobs); \ t \rrbracket \Gamma_L \Gamma_T (p_1, pobs)
\]

For simplicity, we use \( p \), which is a meta-program, as a variable itself (the alternative is to consider the individual variables of \( p \)). In \texttt{con } \( p \bullet \text{equals } (p, pobs); \ t \), an instantiation of \( p \) is angelically chosen to ensure the success of \texttt{equals } \( (p, pobs); \ t \). The tactic \texttt{equals } \( (p, pobs) \) succeeds only for those instantiations that match its argument. Above, its argument is \( (p_1, pobs) \), so \texttt{equals } \( (p, pobs) \) is a filter for the instantiations of \( p \): only those that match \( p_1 \) are considered. If there is none, the tactic fails. If a successful instantiation can be chosen, the instantiated values are used in \( t \), which is applied to \( (p_1, pobs) \).

### 2.3.3 Tactic Declarations and Programs

The effect of a tactic declaration is to include a tactic name \( n \) in the domain of the tactics environment. This element is mapped to a new function that, for each possible argument \( v \in \text{TERM} \), gives the semantics of the tactic when the arguments of the tactic are replaced by \( v \). The presence of the clauses \texttt{proof obligations} and \texttt{program generated} does not change the semantics of a \textit{tacDec}, which is

\[
\llbracket \_ \rrbracket : tacDec \rightarrow LEnv \rightarrow TEnv \rightarrow TEnv
\]

\[
\llbracket \text{Tactic } n(a) \text{ t end } \rrbracket \Gamma_L \Gamma_T =
\Gamma_T \oplus \{n \mapsto \{v \in \text{TERM} \bullet v \rightarrow \llbracket t[a \backslash v] \rrbracket \} \Gamma_L \Gamma_T \}
\]

where \( \oplus \) is the overriding operator.

A tactic program is a sequence \( tds \) of tactic declarations followed by a main tactic \( t \). The semantics is that of \( t \), when evaluated in the environment determined by the tactic declarations and the given laws environment. The definition is as
follows.

\[
\begin{align*}
\llbracket-\rrbracket : & \text{tacProg} \rightarrow \text{LEnv} \rightarrow \text{TEnv} \rightarrow \text{TEnv} \\
\llbracket tds\ t \rrbracket \Gamma_L \Gamma_T = & \llbracket\ t\rrbracket \Gamma_L (\text{decl} \ tds \Gamma_L \emptyset)
\end{align*}
\]

This environment is defined by the function \text{decl}.

\[
\begin{align*}
decl : & \text{seq tacDec} \rightarrow \text{LEnv} \rightarrow \text{TEnv} \rightarrow \text{TEnv} \\
decl(\langle\rrangle \ tds \Gamma_L \Gamma_T = & \Gamma_T \\
decl(\langle td_1 \ tds \rrangle \Gamma_L \Gamma_T = & (\text{decl} \ tds \Gamma_L (\llbracket\ tds \rrbracket \Gamma_L \Gamma_T ))
\end{align*}
\]

In the case that we have an empty tactic declaration, this function returns the tactics environment given as argument. Otherwise, it uses the tactic environment resulting from the first tactic declaration to evaluate the rest of the declarations.

### 2.3.4 Tacticals

Some tactic languages include a definition of a tactical which makes a robust application of a tactic \texttt{t}. This is called \texttt{try} and is defined as

\[
\text{try } t = !(t \mid \text{skip})
\]

Another tactical makes an exhaustive application of a tactic \texttt{t}. This tactic is defined below.

\[
\text{exhaust } t = (\mu \ Y \bullet (t; Y \mid \text{skip}))
\]

This tactic applies \texttt{t} as many times as possible, terminating (with success) when \texttt{t} fails to apply. An example of its use can be found in Section 2.2.8.

Using the formal semantics presented in this section we are able to prove a set of algebraic laws which can be used for checking tactics equivalence. Furthermore, this set can be used for tactics transformations and optimizations. Next chapter, presents this set of algebraic laws.
Chapter 3

Algebraic Laws

In this chapter we present laws of reasoning for ArcAngel. These laws can be used for checking tactics equivalence. Furthermore, they can be used for tactic transformations and optimizations. In order to prove the completeness of the set of laws presented in this chapter, we use them to support a strategy of reduction to a normal form.
3.1 Simple Laws

In this section we present laws proposed in the literature for Angel. In Appendix D we prove their soundness in the context of the ArcAngel’s semantics. The laws marked with (*) do not hold in the presence of recursive tactics.

3.1.1 Basic properties of composition

The three following laws guarantee that the skip is a unit of a sequential composition, and that fail is a unit of alternation, and a zero of sequential composition.

Law 1(a). skip; t = t
Law 1(b). t = skip; t
Law 2(a). t | fail = t
Law 2(b). t = fail | t
Law 3(a)*. t; fail = fail
Law 3(b). fail = fail; t

Now, we have that sequential composition and alternation are associative, and sequential composition distributes over alternation, on the right only.

Law 4. t₁ | (t₂ | t₃) = (t₁ | t₂) | t₃
Law 5. t₁; (t₂; t₃) = (t₁; t₂) | t₃
Law 6. (t₁ | t₂); t₃ = (t₁; t₂) | (t₂; t₃)

The distributive law on the left succeeds only for sequential tactics, which we characterize as follows.

Definition 1. (Sequential Tactics) A tactic is sequential if it is skip, fail, law l for some law l, or it has the form t₁; t₂ where t₁ is in one of these forms and t₂ is a sequential tactic.

Law 7. t₁; (t₂ | t₃) = (t₁; t₂) | (t₁; t₃) for any sequential tactic t₁.

This law establishes that sequential composition distributes over alternation, on the left, if the left tactic is a sequential tactic.

3.1.2 Laws involving Cut

Here, we present some laws involving the cut operator, and how this operator interacts with other tactic combinators.

First, we have that atomic law applications and tactics are unchanged by appli-
The use of the cut operator on the left side of a sequential composition allows it to distribute over an alternation on the right.

Law 11. \(!t_1; (t_2 | t_3) = (!t_1; t_2) | (!t_1; t_3)\)

The cut operator partially distributes over sequential composition and over alternation.

Law 12. \(!t_1; !t_2 = (!t_1; !t_2)\)
Law 13. \(!t_1; t_2) = (!t_1; !t_2)\)
Law 14(a). \(!t_1 | t_2) = (!t_1 | t_2)\)
Law 14(b). \(!t_1 | t_2) = (!t_1 | !t_2)\)

It also has two absorption rules:

Law 15. \(!t_1; t_2) = !t_1\)
Law 16. \(!t_1 | t_2 | t_1) = !t_1 | t_2)\)

In the presence of a cut, the skip becomes a left-zero for alternation, and alternation becomes idempotent.

Law 17. \(!(!t_1; t) = \text{skip}\)
Law 18. \(!t | t) = !t\)

The cut itself is also idempotent.

Law 19. \(!!t = !t\)

### 3.1.3 Laws involving \textit{succs} and \textit{fails}

When we interact fails and succs with each other and with the other basic tactics, a large set of laws can be proved.

For example, verifying if a tactic \(t\) succeeds and then applying it is the same as only applying it. Also, verifying if a tactic \(t\) fails and then applying it is the same
as fail.

Law 20. succeeds \( t; t = t \)
Law 21*. fails \( t; t = \text{fail} \)

In the presence of fails, cut becomes identity, as well as succeeds.

Law 22. fails \( t = \text{fails} !t = \text{fails} t \)
Law 23. fails(succeeds \( t) = \text{fails} t \)

The operator fails is also associative.

Law 24*. fails \( t_1; \text{fails} t_2 = \text{fails} t_2; \text{fails} t_1 \)

Applying cut to an alternation or to a sequential composition can be represented using fails and succeeds.

Law 25. \(! (t_1 | t_2) = !t_1 | (\text{fails} t_1; !t_2) \)
Law 26. \(! (t_1; t_2) = ! (t_1; \text{succeeds} t_2); !t_2 t_2 \)

Applying succeeds and fails to an alternation can also be expanded.

Law 27. succeeds \( (t_1 | t_2) = ! (\text{succeeds} t_1 | \text{succeeds} t_2) \)
Law 28. fails \( (t_1 | t_2) = \text{fails} t_1; \text{fails} t_2 \)

Verifying the success of part of a tactic application is unnecessary if we verify the success of the whole tactic application. However, for failures we should verify each part of the tactic application.

Law 29. succeeds \( s; \text{succeeds} (s; t) = \text{succeeds} (s; t) \)
Law 30. fails \( s = \text{fails} s; \text{fails} (s; t) \)

We present below another property of the cut operator over sequential composition and fails.

Law 31. \(!s; \text{fails} t = \text{fails}(!s; t); !s \)

Nested applications of fails can be removed as follows.

Law 32. fails(fails \( s; t \)) = succeeds \( s | (\text{fails} s; \text{fails} t) \)
Law 33. fails \( s; \text{fails} t = \text{fails} s | \text{succeeds} (s; t) \)

Applying fails to the first tactic of a sequential composition and then applying
succs to the whole sequential composition does not succeed.

**Law 34**: fails $s$; succs$(s; t) = \text{fail}$

Nested applications of fails and succs can be removed as follows.

**Law 35**: fails$(t; d) = \text{fails}(t; \text{succs } d)$
**Law 36**: $!s$; succs $t = \text{succs}(!s; t); !s$
**Law 37**: fails(fails $t) = \text{succs } t$

The fails and succs operators are associative.

**Law 38**: fails $t_1; \text{succs } t_2 = \text{succs } t_2; \text{fails } t_1$
**Law 39**: succs $t_1; \text{succs } t_2 = \text{succs } t_2; \text{succs } t_1$

It is unnecessary nested applications of succs. However, succs has no effect when applied to fails.

**Law 40**: succs(succs $t) = \text{succs } t$
**Law 41**: succs(fails $t) = \text{fails } t$

The following two properties shows, respectively, that the application of succs to a sequential composition $t; d$ is the same as an application of succs to a sequential composition $t; \text{succs } d$, and that, in the presence of succs the cut becomes identity.

**Law 42**: succs$(t; d) = \text{succs}(t; \text{succs } d)$
**Law 43**: succs $t = \text{succs } !t = !\text{succs } t$

We present below some properties of composition of succs and fails with skip and fail.

**Law 44**: succs skip = skip
**Law 45**: succs fail = fail
**Law 46**: fails skip = fail
**Law 47**: fails fail = skip

Now, we present some properties of composition and alternation of fails and succs.

**Law 48**: fails $t; \text{fails } t = \text{fails } t$
**Law 49**: succs $t; \text{succs } t = \text{succs } t$
**Law 50**: succs $t; \text{fails } t = \text{fails } t; \text{succs } t = \text{fail}$
**Law 51**: fails $t \mid \text{succs } t = \text{succs } t \mid \text{fails } t = \text{skip}$

Finally, we present some more expanding properties of nested application of fails
and succs.

Law 52. \( \text{succs}(t \mid u) = \text{succs } t \mid (\text{fails } t; \text{succs } u) \)

Law 53. \( \text{succs}(\text{fails } s; t)) = \text{fails } s; \text{succs } t \)

Law 54. \( \text{succs}(\text{succs } s; t) = \text{succs } s; \text{succs } t \)

Law 55. \( \text{succs}(s; \text{fails } t) = \text{succs } s; \text{fails}(s; t) \)

Law 56. \( \text{succs}(s; \text{succs } t) = \text{succs } s; \text{succs } (s; t) \)

Law 57. \( \text{fails}(\text{succs } s; t)) = \text{fails } s \mid (\text{succs } s; \text{fails } t) \)

Law 58. \( \text{fails}(s; \text{succs } t) = \text{fails } s \mid (\text{succs } s; \text{fails } (s; t)) \)

The laws presented in this section are very useful in Section 3.2.

3.1.4 Laws involving Structural Combinators

Here, we have the laws involving the composition of the structural combinator \( \boxplus \) with sequence, alternation and cut.

Law 59. \( (t_1 \boxplus t_2); (t_3 \boxplus t_4) = (t_1; t_3) \boxplus (t_2; t_4) \)

Law 60. \( t_1 \boxplus (t_2 \mid t_3) = (t_1 \boxplus t_2) \mid (t_1 \boxplus t_3) \)

Law 61. \( (t_1 \mid t_2) \boxplus t_3 = (t_1 \boxplus t_3) \mid (t_2 \boxplus t_3) \)

Law 62. \( !(t_1 \boxplus t_2) = (!t_1 \boxplus !t_2) \)

3.1.5 Laws of con

The side-conditions of these laws use the function \( \phi \), which extracts the set of free-(meta)variables of the tactic to which it is applied. The first property shows that, in a sequential composition \( t_1; t_2 \), if \( v \) is not a free-(meta)variable of \( t_2 \), we may only use the variables of \( v \) in \( t_1 \), and vice-versa. However, the last is true only if \( t_1 = !t_1 \).

Law 63. \( (\text{con } v \bullet t_1; t_2) = (\text{con } v \bullet t_1); t_2 \) provided \( v \notin \phi t_2 \)

Law 64. \( (\text{con } v \bullet t_1; t_2) = t_1; (\text{con } v \bullet t_2) \) provided \( v \notin \phi t_1 \) and \( !t_1 = t_1 \)

If \( v \) is a not a free-(meta)variable in \( t \), the use of \( \text{con } \) is unnecessary.

Law 65. \( (\text{con } v \bullet t) = t \) provided \( v \notin \phi t \)

We can also change the use of a meta-variable \( v \) in \( t \) for a new meta-variable \( u \) if the new meta-variable is not a free-variable in \( t \).

Law 66. \( (\text{con } v \bullet t) = (\text{con } u \bullet t[v \backslash u]) \) provided \( u \notin \phi t \)

In order to prove the completeness of the set of laws presented in this section, we use then to support a strategy of reduction to a normal form in the next section.

69
3.2 ArcAngel’s Normal Form

In this section we prove that tactics written in a subset of ArcAngel that excludes recursion, abort, con, and applies to . . . do have a unique normal form. This guarantees the completeness of the set of laws presented in the previous section.

First, we present the cut-free normal form for the subset of ArcAngel that includes the operator law, sequential composition, alternation, skip, fail, and the structural combinators. Then, we present the pre-normal form, which expands this subset of ArcAngel and includes the cut(!) operator, succs, and fails. However, the pre-normal form does not guarantee uniqueness. For this reason, we present the general normal form, which guarantees this property. The idea is to define a normal form for tactics, to show that every tactic can be transformed into a unique normal form using the laws presented in Section 3.1, and proved in Appendix D, and finally to show that if two tactics are equivalent then they have the same normal form.

ArcAngel’s normal form provides a complete test whether two tactics texts denotes the same tactic. Furthermore, the laws presented can be used for correctness-preserving transformations, or as justification for automatic tactic optimization [15]. The transformation of tactics to a restricted syntax for special applications can also be obtained [30].

In the following proofs we shall use the notation

\[ ; i:I t_i \]

to represent the alternate composition \( t_{i_1} \mid \ldots \mid t_{i_n} \), for \( I = \{i_1, \ldots, i_n\} \), with the alternation combinators associated to the right. The index sequence \( I \) is finite. In the case it has one element only, then the alternation is vacuous and consists only of one instance on the tactic \( t_i \), and in the case \( I \) is empty, the expression denotes fail.

In the same way, we use the notation

\[ ; i:I t_i \]

to denote the composition \( t_{i_1}; \ldots; t_{i_n} \). The empty sequential composition denotes skip.

3.2.1 Cut-free Normal Form

We say that a tactic is in the cut-free normal form if it has the form

\[ ; i:I ( ; j:J M_j) \]

where the \( M_j \) are either law \( l_j \), where \( l_j \) are names of laws in the domain of the environment of laws, or structural combinators of ArcAngel containing tactics in cut-free normal form.
Lemma 1. Any tactic expressed using basic laws, alternation, sequential composition, \texttt{skip}, \texttt{fail}, and any structural combinator of ArcAngel can be transformed into a unique tactic in cut-free normal form using the laws presented in Section 3.1. Proof. The proof is by structural induction over the possible forms of the tactics.

Base cases:
- \texttt{skip} = (; \emptyset)
- \texttt{fail} = | \emptyset
- \texttt{law} l = | _i I (law l)

Inductive step: We assume that
\begin{align*}
t_1 &= ( | _i I (; _j J M)) \text{ and } \\
t_2 &= ( | _k K (; _l L M))
\end{align*}
We must prove that
\begin{enumerate}
\item $t_1 \mid t_2$ can be put in cut-free normal form
\end{enumerate}
Using the associative Laws 4 and 5 we can easily transform the tactic
\[( | _i I (; _j J M)) \mid ( | _k K (; _l L M))\]
to the cut-free normal form.

b) $t_1; t_2$ can be put in cut-free normal form
\[( | _j J (; _i I M_i)); ( | _k K (; _l L M_l))\]
In order to proceed, we use the generalization of the Laws 6 and 7 presented below.

\begin{align*}
\textbf{Law 6'}. & ( | _i I t_i); t_3 = ( | _i I t_i; t_3) \\
\textbf{Law 7'}. & t_1; ( | _i I t_i) = ( | _i I t_i; t_i)
\end{align*}
And we proceed with the proof
\begin{align*}
( | _j J (; _i I M_i)); ( | _k K (; _l L M_l)) \\
= | _j J ((; _i I M_i); ( | _k K (; _l L M_l)))) & \quad \text{[Law 6']} \\
= | _j J ( | _k K ((; _i I M_i); (; _l L M_l)))) & \quad \text{[Law 7']} \\
\end{align*}
This last can be put in the cut-free normal form using the associative Laws 4 and 5.
c) The proof for the structural combinators is direct from the definition of the cut-free normal form. For example, \( \text{var} \{ t_1 \} \) and \( t_1 \{ t_2 \} \) are already in the cut-free normal form since \( t_1 \) and \( t_2 \) are in cut-free normal form by assumption.

Now, we define when two tactics can be considered equivalent: when they have the same behavior for all \( RCells \).

**Definition 2. (Tactic Equivalence)** Two tactics \( t_1 \) and \( t_2 \) are equivalent \( (t_1 \equiv t_2) \) if for all environment of laws \( \Gamma_L \), for all environment of tactics \( \Gamma_T \), and for all \( RCell \ r \), we have that \( \llbracket t_1 \rrbracket \Gamma_L \Gamma_T r = \llbracket t_2 \rrbracket \Gamma_L \Gamma_T r \).

We prove that if two tactics in normal form are equivalent, then they are syntactically equal.

**Lemma 2.** If two tactics in normal form are equivalent, then they are identical.

**Proof.** We prove that there is no point where the tactics differ. We now use the following notation: let the laws which are used in the tactics be \( l_1, l_2, \ldots \), and assume that the \( RCells \) are indexed by sequences of law numbers and applications of structural combinators to such sequences, so that:

\[
\text{law } l_n \ r_0 = [r_{(n)}]
\]

\[
\text{law } l_n \ r_i = [r_{^\sim(n)}]
\]

For alternations we have that:

\[
(\text{law } l_n \mid \text{law } l_m) \ r_i = [r_{^\sim(n)}, r_{^\sim(m)}]
\]

We use the structural combinators to represent themselves. For example, we have that:

\[
(\text{var} \text{ law } l_1; \text{ law } l_2; \text{ law } l_3) \ r_0 = [r \text{ var}^{(1,2,3)} \ r_{(4)}]
\]

Using this indexing, a tactic applied to \( r_0 \) will produce an account of the tactic’s normal form (with the alternation of the sequential composition of the laws in the respective \( RCells \) in the result list) and so the result is immediate.

\[\square\]

Conversely, we have that equivalent tactics have a unique normal form.
Theorem 1. Two tactics are equivalent if, and only if, they have the same normal form.

Proof. Let $t_1$ and $t_2$ be the normal form of $t_1$ and $t_2$, respectively. First we must prove that $t_1 \equiv t_2 \Rightarrow t_1$ is identical to $t_2$. Since $t_1 \equiv t_2$ and $t_1$, $t_2$ are the normal forms of $t_1$ and $t_2$ respectively, we can say that $t_1 \equiv t_1$ and $t_2 \equiv t_2$. This is valid because the laws used in the reduction to the normal form are already proved. So, we can also say that $t_1 \equiv t_2$, and then, using Lemma 2, that $t_1$ is identical to $t_2$.

Then we prove that $t_1$ is identical to $t_2 \Rightarrow t_1 \equiv t_2$. Since $t_1$, $t_2$ are the normal forms of $t_1$ and $t_2$ respectively, we can say that $t_1 \equiv t_1$ and that $t_2 \equiv t_2$. But, we know that $t_1$ is identical to $t_2$, and so, $t_1 \equiv t_2$.

A set of laws can be said to be complete when tactics which are observationally equivalent (i.e. they behave identically on all goals) are provably so (using the laws of this set).

Corollary 1. The laws in Section 3.1 are complete for the cut-free finite (non-recursive) ArcAngel.

Proof. We have shown that, using the laws given above, we can transform a tactic to its normal form (Lemma 1). Using Theorem 1, we can say that two tactics are equivalent if, and only if, they have the same normal form.

3.2.2 Pre-Normal Form

The pre-normal form is for the subset of ArcAngel that includes the subset of the cut-free normal form, the cut (!) operator, and the tactic assertions succs and fails.

First, we revisit the Definition 1 of sequential tactics, so that it also contains the structural combinators.

Definition 3. Sequential Tactics A tactic is sequential if it is skip, fail, law $l$ for some law $l$, !$t$, fails $t$ or succs $t$, for some sequential tactic $t$, a structural combinator applied to a sequential tactic, or it has the form $t_1; t_2$, where $t_1$ is in one of these forms and $t_2$ is a sequential tactic.

The next Lemma is very useful.

Lemma 3. If a tactic $t$ is a sequential tactic then $t = !t$.

Proof. The proof of this Lemma is presented in Appendix D.6.1.
Now it is possible to prove a completeness result for the finite (non-recursive) ArcAngel, including the cut operator. The proof is similar to the one used in the cut-free normal form, but now we have a two-stages process. First, we define a pre-normal form and then we define the general normal form.

The cut-free normal form shows the possible sequences of law applications resulting from a tactic execution. When we include the cut operator in the language, the possibilities depend on the success or failure of sequences of law applications. For this reason we need to include guards which check the validity of the options.

**Definition 4. (Pre-normal form)** A tactic is in pre-normal form if it has the form

\[
\left| \begin{array}{c}
\text{j}; \ g_j ; (; \ i; k M_i) \\
\end{array} \right|
\]

where \( g_j \) are guards of the form \( \text{succs}(; k M_k) \) or \( \text{fails}(; k M_k) \), or a possibly empty sequential composition of such guards. The \( M_i \) are either \text{law} \( l_j \), where \( l_j \) are names of laws in the domain of the environment of laws, or structural combinators of ArcAngel containing tactics in pre-normal form.

Now, we want to prove that all the tactics written in the subset of ArcAngel considered can be put in pre-normal form.

**Lemma 4.** Any tactic written in the language above can be put into pre-normal form using the laws given above.

**Proof.** In this proof we assume the use of the associative Laws 4 and 5 where necessary. The proof proceeds like the one for the cut-free normal form.

**Base cases:**

\text{skip} = \text{succs} \text{ skip}; (; i; \emptyset)\]

\text{fail} = \left| \begin{array}{c}
\text{j}; \emptyset
\end{array} \right|

\text{law} l = \text{succs} (\text{law} l); \text{law} l \quad \text{[Law 20]}

**Inductive step:** We assume that

\[ t_1 = \left( \begin{array}{c}
\text{i}; j; g_j ; (; i; k M_i) \\
\end{array} \right) \]

\[ t_2 = \left( \begin{array}{c}
\text{k}; k; g_k ; (; i; L_k M_i) \\
\end{array} \right) \]

We must prove that

a) \( t_1 \mid t_2 \) can be put in pre-normal form
Using the associative Laws 4 and 5 we can easily transform the tactic

\[(\mid j: g_j; (\mid \iota_j M_i)) \mid (\mid k: k; (\mid \iota_k M_i))\]

to the cut-free normal form.

b) \(t_1; t_2\) can be put in pre-normal form

\[(\mid j: g_j; (\mid \iota_j M_i)) \mid (\mid k: k; (\mid \iota_k M_i))\]

\[= (\mid j: g_j; (\mid \iota_j M_i)) \mid (\mid k: k; (\mid \iota_k M_i))\]  \[\text{[Law 6']}\]

We can generalize the Law 11 as

**Law 11'.** \(!t_1; (\mid \iota; t_2) = (\mid \iota; t_1; t_2)\)

And so we have

\[= (\mid j: (!g_j; (\mid \iota_j M_i)); (\mid k: k; (\mid \iota_k M_i)))\]

\[= (\mid j: (\mid k: (!g_j; (\mid \iota_j M_i)); (\mid k: k; (\mid \iota_k M_i))))\]  \[\text{[Law 11']}\]

\[= (\mid j: (\mid k: (g_j; (\mid \iota_j M_i); (\mid k: k; (\mid \iota_k M_i))))\]  \[\text{[Lemma 3]}\]

This tactic has the form

\[\mid \iota; (g_1; t_1; g_2; t_2)\]

Since \(t_1\) are sequences of laws and structural combinators, we have that \(t_1 = !t_1\) (Lemma 3). Now we can apply the laws 31 and/or 36 to assemble the guard components at the beginning of each alternation branch, and remove the cut operators by applying again the Lemma 3. For example, consider the tactic

\[\text{succs (law } l_1); \text{ law } l_1; \text{ succs (law } l_2 \parallel \text{ law } l_3); \text{ law } l_2 \parallel \text{ law } l_3\]

\[\mid \text{ succs (law } l_4); \text{ law } l_4; \text{ succs (law } l_5; \text{ law } l_6); \text{ law } l_5; \text{ law } l_6\]

\[= \text{ succs (law } l_1); \text{ ! law } l_1; \text{ succs (law } l_2 \parallel \text{ law } l_3); \text{ ! law } l_2 \parallel \text{ law } l_3\]

\[\mid \text{ succs (law } l_4); \text{ ! law } l_4; \text{ succs (law } l_5; \text{ law } l_6); \text{ !}(\text{law } l_5; \text{ law } l_6)\]  \[\text{[Lemma 3]}\]

\[= \text{ succs (law } l_1); \text{ Succs(}!\text{ law } l_1; \text{ law } l_2 \parallel \text{ law } l_3); \text{ ! law } l_1; \text{ !}(\text{law } l_2 \parallel \text{ law } l_3)\]

\[\mid \text{ succs (law } l_4); \text{ Succs(}!\text{ law } l_4; \text{ law } l_5; \text{ law } l_6); \text{ ! law } l_4; \text{ !}(\text{law } l_5; \text{ law } l_6)\]  \[\text{[Law 36]}\]
\[\text{succs}(\text{law } l_4); \text{succs}(\text{law } l_4; \text{law } l_5; \text{law } l_6) \]

\[\text{succs}(\text{law } l_4; \text{succs}(\text{law } l_5; \text{law } l_6); \text{law } l_4; \text{law } l_5; \text{law } l_6)\ [\text{Lemma 3}]\]

The resulting tactic will be in pre-normal form.

c) \(!t\) can be put in pre-normal form.

The tactic \(!t\) can be normalized via repeated (if needed) uses of Law 25. As \(t\) is in the pre-normal formal (inductive hypothesis) this repeated application of this law will distribute the cuts onto the sequential components, from where they can be removed, using Lemma 3, as we can see in

\[!(\mid j: j \ g_i; (i: j M_i))\]

\[= !(g_i; (i: j M_i)) \mid \]

\[\text{fails}(g_i; (i: j M_i)); !(\mid j: j \setminus \{1\} \ g_i; (i: j M_i)) \ [\text{Law 25}]\]

\[= (g_i; (i: j M_i)) \mid \]

\[\text{fails}(g_i; (i: j M_i)); !(\mid j: j \setminus \{1\} \ g_i; (i: j M_i)) \ [\text{Lemma 3}]\]

This distribution may result in the creation of nested instances of \text{succs} and \text{fails}. Laws 32, 33, and 53-58 can be used to remove those nested instances. At this point, applying the distributive Laws 6 and 11, together with the Lemma 3 will put the tactic in pre-normal form.

For example, consider the tactic

\[!(\text{succs}(\text{law } l_4); \text{law } l_4)\]

\[\mid \text{succs}(\text{law } l_4; \text{law } l_4; \text{law } l_4; \text{law } l_4)\]

\[= !(\text{succs}(\text{law } l_4); \text{law } l_4)\]

\[\mid (\text{fails}(\text{succs}(\text{law } l_4); \text{law } l_4); !(\text{succs}(\text{law } l_4; \text{law } l_4); \text{law } l_4; \text{law } l_4)) \ [\text{Law 25}]\]

\[= (\text{succs}(\text{law } l_4); \text{law } l_4)\]

\[\mid (\text{fails } (\text{law } l_4) \mid \]

\[\text{succs}(\text{law } l_4); \text{fails}(\text{law } l_4)); (\text{succs}(\text{law } l_4; \text{law } l_4); \text{law } l_4; \text{law } l_4))\ [\text{Lemma 3}]\]

\[= (\text{succs}(\text{law } l_4); \text{law } l_4)\]

\[\mid (\text{fails } (\text{law } l_4); \text{succs}(\text{law } l_4; \text{law } l_4); \text{law } l_4; \text{law } l_4)\]

\[\mid (\text{succs}(\text{law } l_4); \text{fails}(\text{law } l_4)); \text{succs}(\text{law } l_4; \text{law } l_4); \text{law } l_4; \text{law } l_4) \ [\text{Law 6}]\]

d) \text{fails } t can be put in pre-normal form (inductive hypothesis), and so it is an alternation. We can apply a generalized version of Law 28 to the tactic \text{fails } t to distribute the \text{fails} through the alternation.
Generalizing Law 28 we have that

\textbf{Law 28}. \texttt{fails}( \mid i; t_i) = ; i:\texttt{fails}(t_i)

And so we have

\[
\texttt{fails}( \mid j; g_j; (i; l_i M_i)) = ; j:\texttt{fails}(g_j; (i; l_i M_i)) \tag{Law 28'}
\]

Again, the nested instances of \texttt{succs} and \texttt{fails} can be removed using the Laws 32, 33 and 53-58. In this way, we may introduce alternations as above, and then distributive laws can be used to transform the resulting tactics into pre-normal form.

e) Similar arguments are used to the case \texttt{succs} \texttt{t}. The only difference is to change the application of the Law 28 by the application of the Law 52, and to proceed like we did for the \texttt{fails} \texttt{t} case.

f) The proofs for the structural combinators are a direct application of the definition of the pre-normal form.

In the next subsection we extend the pre-normal form to the general normal form.

\subsection{3.2.3 General Normal Form}

The pre-normal form does not guarantee uniqueness. For example, let \texttt{s} and \texttt{t} be sequence of laws and structural combinators, then the tactics

\[
\texttt{s}; \texttt{t} \\
(\texttt{succs} \texttt{s}); \texttt{s}; \texttt{t} \\
(\texttt{succs} \texttt{s}; \texttt{t}); \texttt{s}; \texttt{t} \\
(\texttt{succs} \texttt{s}); (\texttt{succs} \texttt{s}; \texttt{t}); \texttt{s}; \texttt{t}
\]

are all equivalent, but they are not in the same pre-normal form.

In order to define the general normal form, it is important to introduce some new concepts. Law and structural combinator sequences are sequences of law applications and structural combinators in the form \( M_1; \ldots; M_n \), where \( M_i \) are law applications or structural combinators applied to law and structural combinator sequences. For instance, the sequence \texttt{law} \texttt{l}_1; (\texttt{law} \texttt{l}_2 \texttt{], law} \texttt{l}_4) \) represents such a sequence. A set \( T \) of such sequences is prefix-closed if for any sequence \( t \) in \( T \), \( T \)
also contains all initial subsequences of \( t \). For example, the set

\[
T = \{ \text{law } l_1, \text{law } l_3; \ (\text{law } l_2 \boxtimes \text{law } l_4), \text{law } l_1; \ ((\text{law } l_2; \ \text{law } l_3) \boxtimes \text{law } l_4) \}
\]

is prefix-closed, but the set

\[
T = \{ \text{law } l_3, \text{law } l_4; \ (\text{law } l_2; \ \text{law } l_3) \boxtimes \text{law } l_4) \}
\]

is not, since the element \( \text{law } l_1; \ (\text{law } l_2; \ \text{law } l_4) \) is missing.

In the sequel we present the General Normal Form, which is based in the cut-free normal form. The pre-normal form is used in the proof of the lemmas.

**Definition 5. (General Normal Form)** A tactic \( t \) is in general normal form, relative to a law and structural combinator sequence \( T \), if it has the form

\[
\mid j: g_j; v_j
\]

where \( v_j \) are tactics in the cut-free normal form and \( g_j \) are guards as in the pre-normal form, with certain provisos:

- **(Consistency)** for each guard \( g_j \), if for some law and structural combinator sequence \( t \), \( g_j \) contains \( \text{succs } t \), it must not contain \( \text{fails } s \), for any prefix \( s \) of \( t \) (or \( t \) itself);

- **(Maximality)** for each \( j \), for all \( t \) in \( T \), either \( \text{succs } t \) or \( \text{fails } t \) must be present in \( g_j \);

- **(Sufficiency)** for each \( j \), the success of \( g_j \) must be sufficient to guarantee the success of all the alternate clauses in \( v_j \), i.e., if \( v_j = \mid k: v_k \), then for all \( k \), \( \text{succs } v_k \) must be present in \( g_j \);

- **(Mutual Exclusivity)** the guards are mutually exclusive; that is, for \( i \) and \( j \), with \( i \neq j \), there must be some law and structural combinator sequence \( t \) for which \( g_i \) contains \( \text{succs } t \) and \( g_j \) contains \( \text{fails } t \) (or vice-versa).

We extend the pre-normal form, imposing the above restrictions on the guards, in order to guarantee uniqueness.

If the conditions above are satisfied we have the following properties.

1. \( g_i \neq \text{fail} \): Since the maximality property guarantees that for all \( t \) in \( T \), either \( \text{succs } t \) or \( \text{fails } t \) must be present in \( g_i \). So, it can not be \( \text{fail} \).
2. **suc**s \( t; g_j = g_j \) or **fails** \( t; g_j = g_j \): Again the maximality property guarantees that all sequence of laws and structural combinators are already guarded in \( g_i \), either with suc**c**s or fails. The consistency property guarantees that if there is suc**c**s \( t \), there must not be fails \( s \), for \( s \) some prefix of \( t \). In both cases, using the Laws 24, 39, 38, 48, and 49 we can eliminate the first guard suc**c**s \( t \) or fails \( t \) and maintain only the guards \( g_j \).

3. **suc**c**c**s \( g_j = suc**c**s g_j; suc**c**s \( v_h \): Since the sufficiency property affirms that suc**c**s \( v_h \) is a consequence of suc**c**s \( g_j \).

4. \( g_i; g_j = fail \ (i \neq j) \): Since the mutual exclusivity property says that for two different guards \( g_i \) and \( g_j \ (i \neq j) \) there is some law and structural combinator sequence \( t \) for which \( g_j \) contains suc**c**s \( t \) and \( g_j \) contains fails \( t \) (or vice-versa). So, if suc**c**s \( t \) skips in \( g_i \), then fails \( t \) fails (or vice-versa), then we use the Laws 1, 2 and 3.

For example, consider two mutual excludents guards

\[
\text{suc**c**s(law } l_1\text{); fails(law } l_2\text{)} \\
\text{fails(law } l_1\text{); fails(law } l_2\text{)}
\]

if we compose these two guards, we have

\[
\text{suc**c**s(law } l_1\text{); fails(law } l_2\text{); fails(law } l_1\text{); fails(law } l_2\text{)}
\]

\[= \text{suc**c**s(law } l_1\text{); fails(law } l_1\text{); fails(law } l_2\text{); fails(law } l_2\text{)} \quad \text{[Law 24]} \]

\[= \text{suc**c**s(law } l_1\text{); fails(law } l_1\text{); fails(law } l_2\text{)} \quad \text{[Law 48]} \]

\[= \text{fail; fails(law } l_2\text{)} \quad \text{[suc**c**s } t = \text{skip } \Rightarrow \text{ fails } t = \text{fail]} \]

\[= \text{fail} \quad \text{[suc**c**s } t = \text{fail } \Rightarrow \text{ fails } t = \text{skip]} \quad \text{[Law 3]} \]

Using the conditions seen above, we can say that a tactic in the general normal form has, for example, the form

\[
\ldots | \ldots; \text{suc**c**s } t; \ldots; v_n \ldots \\
\ldots | \ldots; \text{fails } t; \ldots; v_m \ldots
\]

Using the Laws 24, 39, and 38 we can re-order the components of the guards of
these tactics to the form

\[
\begin{align*}
&| \text{succs } t; \ldots; v_n \\
&| \text{fails } t; \ldots; v_m
\end{align*}
\]

This property, together with the mutual exclusion property, means that the outermost alternation can also be re-ordered.

\[
\begin{align*}
&| \text{succs } t; \ldots; v_n \\
&| \text{fails } t; \ldots; v_m
\end{align*}
\]

Therefore, an arbitrary pair of adjacent alternation branches of a tactic in general normal form can be considered to have the form

\[
\text{succs } t; s_1 | \text{fails } t; s_2
\]

\[
\begin{align*}
&= (\text{fails } t | \text{succs } t); (\text{succs } t; s_1 | \text{fails } t; s_2) \quad \text{[Laws 51 and 1]} \\
&= (\text{fails } t; \text{succs } t; s_1) \\
&\quad | (\text{fails } t; \text{fails } t; s_2) \\
&\quad | (\text{succs } t; \text{succs } t; s_1) \\
&\quad | (\text{succs } t; \text{fails } t; s_2) \quad \text{[Laws 6 and 11, Lemma 3]} \\
&= \text{fails } t; s_2 | \text{succs } t; s_1 \quad \text{[Laws 50, 2, 48 and 49]}
\end{align*}
\]

So, we can say that the ordering of the tactic may be changed arbitrarily. This means that the normal forms achieved below is unique only modulo these two forms of reordering. This relative definition is important since the guards \(g_t\) of two tactics, for example, \(t_1\) and \(t_2\), can be syntactically different, but equivalent, given the commutative laws on \text{succs} and \text{fails}, and so, the tactics are equivalent.

\textbf{Lemma 5}. Given a sufficiently large prefix-closed set of law and structural combinator sequences, \(T\), any tactic in pre-normal form can be put into a unique general normal form, relative to \(T\) (unique modulo reordering of the guards), using laws drawn from the set given above. \(T\) is sufficiently large if contains at least the minimal set of law and structural combinator sequences determined by consideration of the tactic \(t\) in pre-normal form: \(T\) must contain all the instances of law and structural combinator sequences.

\textit{Proof}. The idea behind this proof is to sequentially compose an alternation of
guards which is equivalent to skip with the tactic in pre-normal form (since it is equivalent to the tactic itself by Law 1), and then to transform it into a tactic in general normal form.

First, consider the tactic formed from all possible guards for law and structural combinator sequences in \( T \):

\[
\tau \colon T \text{(suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t} | \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t})
\]

This is equivalent to skip since one of the branches will succeed and behave like skip. Using the distributive Laws 6 and 11 (applying Lemma 3 when necessary), we can transform this expression into

\[
\tau \coloneqq \bigvee_{i : I} g_i
\]

where \( g_i = \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t} \) or \( g_i = \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t} \) for all \( t : T \).

It is still equal to skip because the laws preserve the functionality of the tactics. Using Law 34 and the commutative Laws 24, 39, and 38, we can remove all those \( i \) from \( I \) for which \( g_i \) is equivalent to fail. Let us name this new \( I \) as \( I' \).

To illustrate these first steps let us have \( T = \{l_1; l_1; l_2; l_1; l_2; l_2; l_2; l_4\} \). We have the tactic formed from all possible guards for law and structural combinator sequences in \( T \) as seen below:

\[
\begin{align*}
(\text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t} | \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) &; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) \\
(\text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) &; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t})
\end{align*}
\]

Using the sequential composition and alternation distributive laws we get the following alternation.

\[
\begin{align*}
\text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) &; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) \\
(\text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) &; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) \\
(\text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}) &; \text{suc} \begin{array}{c} \text{c} \end{array}\text{c} \begin{array}{c} \text{s} \end{array}\begin{array}{c} \text{t} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t}; \text{f} \begin{array}{c} \text{a} \end{array}\begin{array}{c} \text{i} \end{array}\text{a} \begin{array}{c} \text{i} \end{array}\text{t})
\end{align*}
\]

The branches [2],[3],[4] and [7] can be removed using Laws 34, 24, 39, and 38.
Finally, we get

\[
\begin{align*}
\text{succs } l_1; \text{succs } (l_1; l_2); \text{succs}(l_1; l_2; l_3[l_4]) & \quad [1] \\
\text{succs } l_1; \text{succs } (l_1; l_2); \text{fails}(l_1; l_2; l_3[l_4]) & \quad [5] \\
\text{succs } l_1; \text{fails } (l_1; l_2); \text{fails}(l_1; l_2; l_3[l_4]) & \quad [6] \\
\text{fails } l_1; \text{fails } (l_1; l_2); \text{fails}(l_1; l_2; l_3[l_4]) & \quad [8]
\end{align*}
\]

This final tactic has the mutual exclusion property mentioned in the definition of the general normal form. It also has the consistency property, since those branches which were equal to \textit{fail} were removed, and the maximality property, in that for each \( t \) in \( T \), for each \( i \), either \( g_i \) contains \textit{fails } \( t \) or \textit{succs } \( t \).

Proceeding with the proof, if we sequentially compose the tactic we worked out above with the tactic in pre-normal form we have

\[
( \mid i.; g_i) ; ( \mid j.; h_j ; (; k.; K_j M_k))
\]

Applying the distributive laws we have

\[
\mid i.; ( \mid j.; g_i) ; h_j ; (; k.; K_j M_k)
\]

This tactic is still equivalent to the original tactic in pre-normal form.

By the maximality property, and the sufficient size of \( T \), for each \( h_j \), \( g_i \) pair, we have either that each component of \( h_j \) is present in \( g_i \), and so, by the commutative laws and Laws 48 and 49, \( g_i = h_j \), or that \( h_j \) has a \textit{succs } \( t \) for which \( g_i \) has a \textit{fails } \( t \), and so, by the commutative laws and Law 34, we have that \( g_i; h_j = \text{fail} \). Using the commutative laws and Law 2, we can rewrite the tactic, removing the branches from \( J_i \) (\( J_i' \) replaces \( J_i \)) which are equal to \textit{fail}.

\[
\mid i.; ( \mid j.; g_i) ; (; k.; K_j M_k)
\]

The next step is to omit (using the commutative laws and Law 2) the inner alternation branches whose guards contains an instance of \textit{fails } (\; k.; M_k), since this branch will fail (\( J_i'' \) replaces \( J_i' \)). In the case \( J_i'' \) becomes empty, and so denotes \textit{fail}, this \( i \) can be omitted from \( I' \) using Law 2.

Finally, the distributive law can be used to transform the tactic into the required normal form:

\[
\mid i.; (g_i; \mid j.; j'' ; (; k.; K_j M_k))
\]

This is in normal form, since the \( g_i \) remain maximal, consistent, and mutually exclusive. Their sufficiency arises as a result of the maximality and the omission (in the last step, above) of clauses which must fail.

\[\square\]
Lemma 6. Two tactics in general normal form (relative to some sufficiently large prefix-closed set of rule and structural combinator sequences $T$) are equivalent if, and only if, they are identical modulo reordering of the guards.

Proof. The first part of this proof is to show that if two tactics are identical modulo reordering of the guards, then they are equivalent. This is trivial since the Laws 24, 38, and 39 make reordering possible and the laws are sound.

The second part of the proof is to show that, if two tactics in general normal form are equivalent, then they are identical modulo reordering of the guards. We assume that the laws and structural combinators we refer to are $M_1, M_2, \ldots, M_n$ and the $RCells$ are decorated with pairs. The first element of this pair is a prefix-closed set of law and structural combinator sequences. These sequences are those which can succeed when applied to that $RCell$. The second element is the law and structural combinator sequence which contains the laws and the structural combinators which have been already successfully applied.

This is

$$M_i \ r_s \ t \not= \ [\ ] \iff \exists \ xs : s \mid \ head \ xs = M_i,$$

and

$$M_i \ r_s \ t = r_{s', t} \cap (r_i) \ where \ s' = xs \mid r_i : xs \in s$$

otherwise the laws application fails.

Analyzing the behavior of a tactic applied to different $RCells$, we can determine the guard/body (in cut-free normal form) pairs of its normal form. The application of a tactic to an $RCell \ r_{s,()}$ will either result in the empty sequence or in a sequence of the form $r_{s_1, t_1}, \ldots, r_{s_n, t_n}$. The former case does not interest us since it will not help us to reconstruct the tactic in general normal form.

In the latter case, the body part of an alternation can be reconstructed as the alternation of $t_1, \ldots, t_n$. The guard for that branch can be also reconstructed from $\text{succs}$ applied to each member of $s$ and $\text{fails}$ applied to each member of $T \setminus s$.

If we take all the initials $RCells \ r_{s,()}$ (with $s$ any prefix-closed subset of $T$) for which the outcome is not the empty sequence, it is possible to reconstruct the whole tactic in normal form.

For example, let us consider an hypothetical case where the set

$$T = \{ l_1, l_1; l_2, l_1; l_3, l_1; l_2; l_5[l_4] \}$$

and there are only two possible initial $RCell$ which are

$$R_{l_1, l_1; l_2, l_1; l_3, l_1; l_2; l_5[l_4],()}$$

$$R_{l_1, l_1; l_2, l_1; l_3; l_5[l_4],()}$$

Now, we consider that when we apply a tactic $t_1$ to these two $RCells$ we get,
respectively,

\[
\langle R_{\langle l_1, l_2, l_3, l_4 \rangle}, R_{\langle l_2, l_3, l_4, \langle h, l_4 \rangle \rangle} \rangle
\]

If we rebuild this tactic as described before, we have, for the first branch (from the first \textit{RCell})

\[
\text{succe}(l_1; \text{succe}(l_1; l_2); \text{succe}(l_1; l_3); \text{succe}(l_1; l_2; l_2 \lceil l_4); (l_1; l_2; l_3 \lceil l_4 | l_1; l_4)
\]

\textit{fails} does not appear because \( T \setminus s \) is empty for the first \textit{RCell}. For the second \textit{RCell} we have

\[
\text{succe}(l_1); \text{succe}(l_1; l_2); \text{succe}(l_1; l_2; l_3 \lceil l_4); \text{fails}(l_1; l_3); (l_1; l_2; l_3 \lceil l_4)
\]

Now, we make an alternation of these to branches, and we have the tactic in the normal form.

\[
\text{succe}(l_1); \text{succe}(l_1; l_2); \text{succe}(l_1; l_3); \text{succe}(l_1; l_2; l_2 \lceil l_4); (l_1; l_2; l_3 \lceil l_4 | l_1; l_3)
\]

We guarantee the maximality property by ensuring that every member of \( T \) is present in each guard, since we apply \textit{succe} to each member of \( s \) and \textit{fails} to each member of \( T \setminus s \). The mutual exclusivity is ensured by the fact that the sets \( s \) we consider in each initial \textit{RCell} \( r_i,0 \) are different, and so, the set of sequences with \textit{succe} are different. The consistency property is ensured by the fact that the guards either contains \textit{succe} \( x \) if \( x \in s \) or \textit{fails} \( x \) if \( x \in T \setminus s \). The last property, the sufficiency, is guaranteed by the fact that the law and structural combinators sequences \( t_1, ..., t_n \) are those which succeeds when applied to the initial \textit{RCell}. So, these law and structural combinator sequences are in \( s \), and consequently, \textit{succe} \( t_i \) is in the guard \( g_j \).

In this way, a tactic in normal form can be completely characterized (modulo reordering) by the set of goals on which it succeeds and the outcomes when it does so, and the result follows immediately.

\[ \Box \]
Theorem 2. Two tactics are equivalent under all law and structural combinators instantiations if, and only if, they have the same general normal form, modulo reordering of the guards.

Proof. Let $t_{1n}$ and $t_{2n}$ be the normal form of $t_1$ and $t_2$, respectively. First we must prove that $t_1 \equiv t_2 \Rightarrow t_{1n}$ is identical to $t_{2n}$, modulo reordering of the guards. Since $t_1 \equiv t_2$ and $t_{1n}$ and $t_{2n}$ are the normal forms of $t_1$ and $t_2$ respectively, we can say that $t_1 \equiv t_{1n}$ and $t_2 \equiv t_{2n}$. This is valid because all the laws used in the reduction to the normal form are already proved. So, we can also say that $t_{1n} \equiv t_{2n}$, and then, using Lemma 6, that $t_{1n}$ is identical to $t_{2n}$, modulo reordering of the guards.

Then we prove that $t_{1n}$ is identical to $t_{2n}$, modulo reordering of the guards, implies $t_1 \equiv t_2$. Since $t_{1n}$ and $t_{2n}$ are the normal forms of $t_1$ and $t_2$ respectively, we can say that, $t_{1n} \equiv t_1$ and that $t_{2n} \equiv t_2$. But, we know that, $t_{1n}$ is identical to $t_{2n}$, modulo reordering of the guards, and so, $t_1 \equiv t_2$.

\[\square\]

Corollary 2. The set of laws is complete for the language described above.

Proof. The proof of this corollary is similar to the proof of the Corollary 1.

\[\square\]

In order to put tactics which involve tactic $t$ in the general normal form, we can replace the tactic call by the tactic body itself. It must be in the general normal form, or it must be possible to reduce it to the general normal form.

We are yet to consider tactics involving recursion, abort, con, and applies to ... do. This is left as further work.
Chapter 4

Gabriel: a tool for ArcAngel

In this chapter we present Gabriel. We present its concepts, discuss its user interface, and present the constructs of ArcAngel available in Gabriel. Finally, we present an example of writing and using a tactic.
4.1 Gabriel Concepts

Gabriel [27] is a tactic editor plugged-in to Refine [8], a tool that supports program development using Morgan’s refinement calculus. Using Refine, the user starts with a formal specification, and repeatedly applies laws of refinement in order to obtain the program which correctly implements the initial formal specification. After each law application, Refine shows to the user the program development, the collected code and the proof obligations generated with the law applications. Refine also allows users to save a program development in order to continue it later. Furthermore, Refine also has the undo and redo operations.

Gabriel is activated from Refine. Using Gabriel, the user can:

- Create a tactic: the user writes a tactic in ASCII ArcAngel (see Section 4.3);
- Edit a tactic;
- Generate a tactic: the system verifies the syntax correctness, and includes the tactic to the list of tactics of Refine. Afterwards, the user can apply the generated tactic in program developments;
- Remove a tactic;
- Apply a tactic.

The generation of a tactic automatically from a program development in Refine is also a required functionality. However, this is not implemented in the current version of Gabriel and is left as future work.

Some diagrams of Gabriel implementation can be found in Appendix E. We present the class diagrams of the Gabriel-Refine integration and tactics hierarchy, and the sequence diagrams for tactic generation and application.

4.2 Gabriel’s User Interface

Gabriel has a simple user interface. This interface is presented in Figure 4.1. The buttons descriptions are (from left to right):

- Start new tactic;
- Open existing tactic;
- Save tactic;
- Generate tactic and insert it into the tactics list of Refine;
Refine’s user interface was modified in order to use Gabriel. Basically, the changes are: the creation of the tactics list and the addition of a button that opens Gabriel. Figure 4.2 shows the tactics list window.

Gabriel’s user interface is based [24] on the LUCID, the User-Centered Interface Design [16]. First, we made an action-object analysis; this activity consists of building a directed-graph describing the activities of program refinements, and building an object-tree containing the objects of Refine-Gabriel’s user interface. Using this technique we analyzed the correspondence of program refinement activities and the user-interface of Refine and Gabriel.

Finally, a test was made with Refine-Gabriel’s potential users. This test consisted of a tactic creation and application using Gabriel and Refine, respectively. After the tests, we interviewed the users, identifying usage difficulties and the reasons for them. Based on the whole test, the users identified positive points of Gabriel and made suggestions for improvement. The positive aspects pointed were:

- Open symbols keyboard, which is used to insert the ASCII representation of ArcAngel’s constructs.
- Easy to open the tactic editor (Gabriel);
- Easy to see the result of a tactic creation;
- Easy application of laws and tactics;
- Nice integration Refine-Gabriel

Figure 4.2: Refine’s Tactics List

The suggestions made were the inclusion of:

- A symbol keyboard for Gabriel;
- ArcAngel documentation in the help;
- A Generate tactic button in Gabriel;
- A Show tactic/law details facility with a double click on the tactic/law in the tactic/law list in Refine;
- A tactic template to start new tactics.

Most of these suggestions were applied to Refine and Gabriel.

4.3 Constructs of Gabriel

Gabriel uses ArcAngel as a tactic language. However, some constructs of ArcAngel, as structural combinators, cannot be written in ASCII. For this reason, Gabriel uses an ASCII representation of ArcAngel which is presented in Appendix F.1. The
tactics \texttt{con}, \(\mu\), \texttt{val}, \texttt{res}, and \texttt{val-res} are not implemented in the current version of \textit{Gabriel}. These are left as future work.

Furthermore, \textit{Gabriel} supports most of the laws of Morgan’s refinement calculus. The names of the laws implemented in Refine and their usage template are presented in Appendix F.2. The law usage template defines the name of the law and its arguments. For example, the strengthening post-condition law must be used as \texttt{strPost(PRED(newPost))}, where \texttt{strPost} is the name of the law and \texttt{PRED(newPost)} is the argument of the law.

All the arguments are typed. The types that can be used in \textit{Gabriel} are presented in Appendix F.3. The list of tactics as an argument is not implemented in this version of \textit{Gabriel} and is left as future work.

### 4.4 Using \textit{Gabriel}

In this section we present an example of a tactic creation and application. We use

![Figure 4.3: \texttt{takeConjAsInv} written in ASCII ArcAngel](image)

90
the tactic \textit{takeConjAsInv}(Section 2.2.3) and the example presented in page 23.

In order to create a tactic we must open \texttt{Gabriel} pressing \texttt{Gabriel}'s button in Refine. Afterwards, we must write the tactic \textit{takeConjAsInv} in ASCII \texttt{ArcAngel} as seen in Figure 4.3.

After writing the tactic, we must save it before its generation. We can save a tactic pressing the save button or choosing the save option in the file menu of \texttt{Gabriel}.

After saving the tactic we can generate it pressing the generate button or choosing the generate option in the file menu of \texttt{Gabriel}. If the generation succeeds, the tactic is inserted in the list of tactics of Refine. However, if some syntax error is found, the system shows an error message indicating the line of the error.

Once the tactic is generated we can apply it in Refine. In our example, the program \( q, r : [a >= 0 \& b > 0, (a = q * b + r \& 0 <= r) \& not r >= b] \) is the initial program. We start the program by pressing the start new development button. Figure 4.4 presents Refine’s window used to insert the new start specification.

![Figure 4.4: Refine’s New Development Window](image)

Then, we select the program in the development window and the tactic \textit{takeConjAsInv} in the tactics list window and press the apply button in this window. The system
will require the parameters of the tactic application, as presented in Figure 4.5.

![Arguments Window](image)

**Figure 4.5: Gabriel’s Arguments Window**

We insert the values presented in Table 4.1.

![Refine’s Development Window](image)

**Figure 4.6: Refine’s Development Window**

After the insertion of the last argument value, Refine actually applies the tactic. The program development window, the proof obligations window and the collected code window are actualized. Figure 4.6 presents the resulting development window.

Finally, we select the guarded program in the iteration by clicking the left-button of the mouse on it, select the assignment with initial variables law in the list of laws window, and apply it by pressing the apply button of this window. We use the
arguments $q, r$ and $q + 1, r - b$.

<table>
<thead>
<tr>
<th>Argument Name</th>
<th>Argument’s Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>invBound</td>
<td>$b &gt; 0$</td>
</tr>
<tr>
<td>lstVar</td>
<td>$q, r$</td>
</tr>
<tr>
<td>lstVal</td>
<td>$0, a$</td>
</tr>
<tr>
<td>variante</td>
<td>$r$</td>
</tr>
</tbody>
</table>

Table 4.1: Argument’s Values

At the end of the development, the resulting collected code window is that

![Figure 4.7: Refine’s Collected Code Window](image)

presented in Figure 4.7.
Chapter 5

Conclusions

In this chapter we give an overview of the results of our work. Related works are also presented in this chapter. Finally, we suggest topics for future work.
5.1 Contributions and Conclusions

We have presented ArcAngel, a refinement-tactic language. Using this language, it is possible to specify commonly used strategies of program development. Tactics can be used as transformation rules; this shortens developments and improves their readability. We have identified some strategies of program development, documented them as tactics using ArcAngel, and used them in some program developments as single transformation rules.

We have defined the semantics of ArcAngel based on the Angel semantics. The main difference is that Angel is a very general language and its goals have no defined structure. For us, a goal is called an RCell, which is a pair with a program as its first element and a set of proof obligations as its second element. The application of a tactic to an RCell returns a list of RCells containing the possible output programs, with their corresponding set of proof obligations.

We have shown the soundness of algebraic laws for reasoning about ArcAngel tactics. We have covered most of the laws that have been proposed for Angel. However, most of them have no available proof in the literature. These proofs are provided here in the context of ArcAngel. Since these laws are valid, the strategy proposed to reduce finite Angel tactics to a normal form has been applied to ArcAngel tactics.

Finally, we have presented Gabriel, a tool to support the use of ArcAngel. We have presented the concepts of the tool and briefly discussed its user interface. We have presented the constructs of ArcAngel available in Gabriel, the names of the laws of Morgan’s refinement calculus available in ArcAngel, and the types of arguments which can be used in the tool. An example of a tactic creation and application has also been presented.

5.2 Related Work

Several other tools that support program refinement are available in the literature. Some of them provide facilities for use of tactics, but, as far as we know, none of them has a special purpose language, like ArcAngel.

The aim of the work and the tool described in [11] is also to find a way of encoding developing strategies and reusing them in other developments. As Gabriel their system helps the user to develop programs using the refinement calculus. It supports the specification and programming constructs described in [23]. The user loads an initial program (or types it), and then, repeatedly uses commands displayed in a menu. Basically, the user can apply a refinement law or a tactic (refine command), undo the last refinement step, elide a part of the program, or expand to see hidden parts again. The expand and print command produces a \LaTeX description of the refinement step. The rerun command is also supported.
by this system. The interaction between the user and the system is menu-mouse based.

This system supports data refinement of variable blocks. The user specifies the abstract variables, the concrete variables, and the coupling invariant that defines the relationship between them. It uses an assisted, theory-driven rewriter to discharge proof obligations. This is possible only if they can be simplified to true, otherwise the user is asked to prove it. Also, trivial predicates are automatically removed from specification statements. This facility is still not supported by Refine, nevertheless their system does not display rules and proof obligations as Refine does.

Tactics in their system can be written in a language that is embedded in Prolog, which has a formal semantics [18, 1]. As far as we know, however, the extension of Prolog used as a tactic language does not have a well defined semantics. This language has operators for sequential, alternate, and conditional composition. The sequential and the alternate composition are as in ArcAngel, and the conditional composition corresponds to ArcAngel's cut operator. Constructs for tackling parts of the programs are also provided. Nevertheless, their tactic language does not have a recursion operator.

Their tactic language has a different style if compared with ArcAngel's. At each step of the tactic, the user defines the part of the program that is being refined, the refinement law, and the structure of the resulting program. In ArcAngel, the user must only define the refinement law and to which point of the program it applies. The structure of the resulting program is contained in the law definition. Although the former gives an easier-to-read tactic, the latter allows tactic programs that can be implemented using parallel application of tactics. For instance, to implement \texttt{if } t_1 t_2 \texttt{ fi }, we can apply \texttt{t}_1 and \texttt{t}_2 in parallel.

Primitive refinement rules can also be defined using the same language. This brings to the user the possibility to include new refinement laws in the system. Nevertheless, it is improbable that new laws beyond those in Morgan’s refinement calculus are needed in any development.

The Refinement Editor [33, 34] has an embedded tactic language. A fundamental feature of the editor is a record of law applications in a program development, and its reuse in later developments. The Refinement Editor is a multiple windows system containing a main window for the overall development; a window for the refinement of a sub-program; a window for the proof obligations; a window for the list of available commands; and a window that displays the record of law applications. The main window accepts commands and displays the transformations. The user can save and load a development, cut and paste parts of a development, modify existing law applications, and label segments of the development. The proof obligations can be discharged using an theorem prover that needs to be provided by the user, but may be invoked directly from the editor. The support for data refinement is still to be done.
The record of law applications uses a tactic language similar to ArcAngel. It has constructs for sequential composition and structural combinators. Nevertheless, the Refinement Editor tactic language does not have operators for alternation, recursion and backtracking. The tactic language also does not have a formal semantics. The library of basic laws contains the laws of Morgan’s refinement calculus. This library can be extended with user-defined laws. Their definitions use pre-defined laws and must provide the provisos and the parameters of the laws. This brings advantages and disadvantages as already discussed.

The Proxac system [31, 32] is a very general transformation editor. Its goal is to support the application of a sequence of transformation steps based on algebraic rules of one or more theories. As such, Proxac supports both refinement by calculation and theorem proving. Proxac offers a reduce option, which applies a predefined set of transformations exhaustively. Some of the main functions are: apply a rule (possibly with a hint to the system), delete the last step, and manipulate expressions.

Its user-interface is based on windows, and uses mouse and keyboard for the interaction with the user. By selecting the line to which the rule is going to be applied, selecting a rule in the rules window, and pressing the a-key, the rule is applied to the selected line. The editor attempts to match it against the left-hand side or the right-hand side of the rule. If they do not match, then the editor proceeds by attempting to match a sub-expression of the selected line. The system attempts to verify the side conditions. If it fails, then the editor questions their validity. The user must decide whether to proceed or not.

Proxac theories are organized in modules. A module has a name, and declares a list of items. These items are declarations of identifiers, operators, imports of other modules, and statements of a property or a rule. Those are written using a specific language described in [31]. A transformation session occurs in the context of the definitions in one or more modules. Proxac contains a module with all the laws of Morgan’s refinement calculus, as well as modules with pre-defined operators and properties.

Proxac does not have a tactic language, and also does not provide the reuse of previous derivations. The user can only save the derivation to edit it later.

The aim of the tool presented in [12, 13] is to support more flexible styles of program developments than the strict top-down. In particular, program construction by incremental enhancement and program reuse is considered.

The tool has two windows: the program window, where the user can select components of the program to be transformed by clicking and dragging with the mouse; and the script window, which is divided in three parts. The center part displays the refinement step that is currently being assembled or ready to be applied. The top part contains the history of steps that have been applied (script). The bottom part contains the steps that are available to be applied.
At each step of the derivation, the user can perform several operations on the script window: insert a new refinement step; step backwards through previous steps; jump to any earlier step; apply the next step from the script, possibly modifying a step; delete the next step of a script; insert an existing script into the script window; and apply remaining steps from the script automatically, until a step fails. The modifications can be propagated to later steps in a script. Adaptations use an editor which has only a query-replace facility. The refinement tool automatically constructs a script describing the derivation. These derivation scripts can be constructed, saved, copied, and edited.

The script language used to record the derivations is an easy to learn tactic language also based in Prolog terms. Its only construct is the sequential composition. It has the same style as [11]. We have not found in the literature a definition of the script language formal semantics.

The tool also attempts to discharge proof obligations using a library of rewrite rules. Nevertheless, the tool allows the derivations to proceed if it cannot discharge the proof obligations. Data refinement rules are available. Procedures can be introduced extracting a fragment of the script, inserting a corresponding procedure declaration, and replacing the removed fragment by a procedure call.

In [4, 35, 36] the HOL System, a theorem-proving assistant that can be used to formalize theories and verify proofs within these theories, is used in program refinements. The user interacts with the HOL System through ML, a functional programming language, making definitions and evaluating expressions. Definitions and expressions are written in a combination of predicate calculus and typed lambda calculus. All this requires quite an amount of ML programming.

Using this language, the user can program tactics that can be used in later developments. The combinators available are THEN, a sequential composition; REPEAT, which repeats a tactic as long as it is applicable; and ORELSE, which tries a second tactic if the first one fails.

The formalization of the refinement concepts includes a theory of commands. A number of algorithmic and data refinement rules have also been proved. The specification language formalized is more restricted than the refinement calculus language of Back and Wright [3]. For example, multiple assignments and recursion cannot be expressed in this specification language. Beyond the sequential, assignment, alternation, iteration commands, and variable and logical constants blocks, Back and Wright refinement calculus language presents also non-deterministic assignment (angelical and demonic), and inverse commands. Procedures and parameters are not supported by their language, although their language can be extended to support them.

In order to prove a refinement, the programmer must express the problem in their formalization. The proof mainly consists of rewriting using basic definitions.

The Program Refinement Tool (PRT) [6] is built as an extension of the Ergo
theorem prover [5]. It supports a language similar to Morgan’s refinement calculus.

PRT uses the window inference proof paradigm, which is based on term rewriting [29]. In this paradigm, the proof is conducted with a stack of windows. Each window has a focus, which is the expression to be transformed; a set of assumptions from the context of the window; a relation between the focus and the expression to which it is transformed; and a theorem which relates the focus of the window to that of the window that originated it (the parent window). In order to transform a subexpression, the user opens a subwindow using special inference rules, called window rules. This pushes a new window onto the stack of windows. When the transformation of a window is finished, the user closes the window using a window rule. This pops the window from the stack and uses its theorem to infer how to transform the parent window.

The program window inference theory of PRT has support for dealing with program variables, applying refinement rules, and managing the kind of context that arises during refinement. This support includes a definition of the refinement relation, the refinement rules, window opening and closing rules, a mechanism for handling program variables and their substitution, a tactic support for applying and instantiating refinement rules, and a proof interface. Using PRT, each refinement step is an application of an instance of a theorem schema in the theory inserted in the tool, with the meta variables instantiated to correspond to the current context.

The interface of PRT is a window divided in several frames. The most important frames are the proof frame, which contains the panes that display the current state of the refinement; the proof script frame, where the commands required to perform the refinement are recorded; the rules frame, which controls the display and selection of matching rules; and the help frame. To select a subterm of the proof or a law to be applied, the user can point and click with the mouse. This opens a frame on the subterm, the proof browser, or applies the selected rule to the highlighted term.

Some proof obligations are discharged using the Ergo window inference theory, which includes logic, set theory, arithmetic, and structures such as sequences. The remaining proof obligations are handled in one of three ways. The default is to record them as conjectures, which must be discharged before closing the current window. Another possibility is to record them as postulates in the theory, which can be discharged later as separated proofs. Finally, they can be discharged as they are generated. The theorem and its proof are stored and can be viewed subsequently using the browser and applied in subsequent developments.

Tactics support is in the form of proof scripts written in a language based on Prolog. It includes commands for all the rule applications and the proofs of their side-conditions as sub-proofs. The user can copy, combine, and edit scripts. As mentioned above, PRT records proof scripts as developments. We can also load a script and replay it line by line to recreate the proof. We have not found a detailed description of this language.
New refinement rules can be postulated and proved with the definition of refinement and the weakest precondition semantics of the program constructs used. Application theories can be defined building upon an existing library that includes first-order predicate logic, arithmetic, and set theory.

Jim Grundy [14] uses the HOL System to formalize a refinement support system. Instead of formalizing the refinement calculus based on weakest pre-conditions, he formalizes programs as predicates. This has a goal-oriented reasoning approach, which is a popular method of proving theorems. However, this is not usual when we are refining programs, since we do not know the final implementation of the specification before finishing its refinement.

In his formalization, refinements preserve only partial correctness with respect to a specification. This means that a program implements correctly a specification if it stops. Otherwise, it may run indefinitely.

Specifications are written using logic, with a extension to support initial variables. The executable constructs are skip, multiple assignment, alternation, sequential composition, and recursion. These constructs and the support for initial variables are a syntactic sugaring of standard notations supplied by an user-interface. The laws formalized in the system are skip introduction, assignment, alternation, iteration, and sequential composition.

The user creates tactics as sets of commands that automate, or partially automate, common program refinement steps. Users can add these tactics in order to use them in later developments.

5.3 Future Work

Generalizing the normal form to encompass the rest of the language is not trivial and is still to be done. Recursion, abort, con, and applies to \ldots do are still to be inserted in the subset of ArcAngel that has a defined unique normal form. Moreover, the simple approach for expression arguments must also be extended. These should be evaluated before being used.

The automatic tactic generation from a program development in Refine is not implemented in the current version of Gabriel. Besides, Gabriel does not implement a subset of ArcAngel. This subset includes recursion (\(\mu\)), con \(v \bullet t\), and the structural combinators [val], [res], and [val-res]. The inclusion of the structural combinators in Gabriel is very simple. However, the inclusion of recursion and con \(v \bullet t\), and the automatic tactic generation needs more complex analysis and implementation.

Furthermore, in Gabriel tactics cannot be passed as argument. The implementation of tactics as arguments allows the implementation of some tactics which are not yet implementable in Gabriel [25]. However, this needs a theoretical analysis which is still to be done. Moreover, the functions tail, head', and seqToList, used
to define the tactic procCalls(Section 2.2.8) and the tactic procArgs(Section 2.2.9), are not implemented in Gabriel. Their implementation makes it possible their use in Gabriel and, consequently, the use of the tactic recProcArgs(Section 2.2.10). The implementation of these functions in Gabriel is very simple.

Gabriel’s error messages in tactic applications do not express precisely the reasons of the error. This happens to allow backtracking in these applications. In the current implementation of Gabriel, if we display a tactic application failure, further applications are not possible anymore. For this reason, when a tactic application in Gabriel fails, an error message is given to the user indicating only that an error has occurred. However, the reasons of the failure are not displayed. In order to give the user some feedback about the application errors, an application log is generated and indicated by the error message. Implementing a way to display the application log is very simple and is left as future work.

The discharge of trivial proof obligations is not implemented in Refine. Using an existing theorem-prover to discharge them is also required. Finally, generating a \LaTeX document describing a program development and a tactic can also be inserted in Refine and Gabriel, respectively.
Appendix A

ArcAngel’s Novel Constructs

Here, we distinguish the tactics of ArcAngel which are inherited from Angel and the tactics of ArcAngel which are not defined in Angel. In the figure below, we reproduce part of the syntax of ArcAngel and indicate on the right the constructs that we already available in Angel.

| tactic ::= law name args  
|          | tactic name args  
|          | skip | fail | abort  
|          | tactic; tactic | tactic | tactic  
|          | ! tactic | µ name • tactic  
|          | succ tactic | fails tactic  
|          | tactic tack | tack | tactic  
|          | if tack + fi | do tack + od  
|          | var tack | con tack  
|          | pmain tack | pmainvariant tack  
|          | pbbody tack | pbbodyvariant tack  
|          | pbbodymain tack | tack tack  
|          | pmainvariantbody tack tack  
|          | val tack | res tack | val-res tack  
|          | parcommand tack  
|          | con v • tack  
|          | applies to | program do tactic  

Figure A.1: ArcAngel’s Novel Constructs
Appendix B

Infinite Lists

We present the model for infinite lists adopted here [19]. The set of the finite and partial sequences of members of \( X \) is defined as

\[
pfseq X ::= \text{partial} \langle \text{seq} X \rangle | \text{finite} \langle \text{seq} X \rangle
\]

We define an order \( \sqsubseteq \) on these pairs such that for \( a, b : pfseq X \), if \( a \) is finite, then \( a \sqsubseteq b \) if, and only if, \( b \) is also finite and equal to \( a \). If \( a \) is partial, then \( a \sqsubseteq b \) if, and only if, \( a \) is a prefix of \( b \).

\[
\sqsubseteq : pfseq X \leftrightarrow pfseq X
\]

\[
\forall gs, hs : \text{seq} X \bullet
\]

\[
\text{finite} gs \sqsubseteq \text{finite} hs \iff gs = hs
\]

\[
\text{finite} gs \sqsubseteq \text{partial} hs \iff \text{false}
\]

\[
\text{partial} gs \sqsubseteq \text{finite} hs \iff gs \text{ prefix } hs
\]

\[
\text{partial} gs \sqsubseteq \text{partial} hs \iff gs \text{ prefix } hs
\]

A chain of sequences is a set whose elements are pairwise related.

\[
chain : \mathcal{P}(\mathcal{P}(pfseq X))
\]

\[
\forall c : \mathcal{P}(pfseq X) \bullet c \in chain \iff (\forall x, y : c \bullet x \sqsubseteq y \lor y \sqsubseteq x)
\]

The set pfiseq contains partial, finite, and infinite list of elements of \( X \), which are
prefixed-closed chains of elements in pfseq $X$.

$$\text{pfiseq } X \equiv \text{pchain}[X]$$

The idea is that $\bot = \text{partial } \langle \rangle$, the empty list $[\ ] = \text{finite } \langle \rangle$, and the finite list $[e_1, e_2, \ldots, e_n]$ is represented by the set containing $\text{finite } \langle e_1, e_2, \ldots, e_n \rangle$ and all approximations to it. An infinite list is represented by an infinite set of partial approximations to it. The infinite list itself is the least upper bound of such a set.

The definitions of the functions used in this thesis are as follows.

1. The map function $\ast$ maps a function $f$ to each element of a possibly infinite list.

$$\ast : (X \rightarrow Y) \rightarrow \text{pfseq } X \rightarrow \text{pfseq } Y$$

$$\forall c : \text{pfiseq } X; f : X \rightarrow Y \bullet$$

$$f \ast \bot = \bot$$

$$f \ast c = \{ x : c \bullet \text{pfmap } f \ x \}$$

The function $\text{pfmap}$ maps the function $f$ to the second element of $x$.

$$\text{pfmap} : (X \rightarrow Y) \rightarrow \text{pfseq } X \rightarrow \text{pfseq } Y$$

$$\forall xs : \text{seq } X; f : X \rightarrow Y \bullet$$

$$\text{pfmap } f (\text{finite } xs) = \text{finite } (f \circ xs) \land$$

$$\text{pfmap } f (\text{partial } xs) = \text{partial } (f \circ xs)$$

2. The distributed concatenation returns the concatenation of all the elements of a possibly infinite list of possibly infinite lists.

$$\infty/ : \text{pfiseq}(\text{pfiseq } X) \rightarrow \text{pfiseq } X$$

$$\forall s : \text{pfiseq}(\text{pfiseq } X) \bullet$$

$$\infty/ s = \bigsqcup_{\infty} \{ c : s \bullet \infty/ c \}$$

It uses the function $\infty/$, which is the distributed concatenation for $\text{pfiseq}(\text{pfiseq } X)$. 

104
The function $\textit{cat}$ is the standard concatenation function for $X^*$.  

\[
\infty/ : \text{pfseq}(\text{pfseq} X) \to \text{pfseq} X
\]

\[
\infty/(f, \langle \rangle) = \emptyset
\]

\[
\infty/(p, \langle \rangle) = (p, \langle \rangle)
\]

\[
\infty/(f, \langle g \rangle) = [g]
\]

\[
\infty/(p, \langle g \rangle) = [g] \uparrow \{ (p, \langle \rangle) \}
\]

\[
\infty/(f, gs \upshs hs) = (\infty/(f, gs)) \infty (\infty/(f, hs))
\]

\[
\infty/(f, p \upshs hs) = (\infty/(f, gs)) \infty (\infty/(p, hs))
\]

The function $\infty$ is the concatenation function for possibly infinite lists. Its definition is

\[
\infty_\downarrow : \text{pfseq} X \times \text{pfseq} X \to \text{pfseq} X
\]

\[
\forall a, b : \text{pfseq} X \bullet \ a \infty b = \{ x : a; \ y : b \cdot x \uparrow y \}
\]

where the function $\uparrow$ is the concatenation function for $\text{pfseq} X$ defined as

\[
\infty_\downarrow : \text{pfseq} X \times \text{pfseq} X \to \text{pfseq} X
\]

\[
\forall gs, hs : \text{seq} X; \ s : \text{pfseq} X \bullet
\]

\[
\text{finite} gs \uparrow \text{finite} hs = \text{finite} (gs \upshs hs)
\]

\[
\text{finite} gs \uparrow \text{partial} hs = \text{partial}(gs \upshs hs)
\]

\[
\text{partial} gs \uparrow s = \text{partial} gs
\]

3. The function $\textit{head}'$ returns a list containing the first element of a possibly infinite list.

\[
\textit{head}' : \text{pfiseq} X \to \text{pfiseq} X
\]

\[
\textit{head}' \hs = \text{take} 1 \hs
\]

It uses the function $\textit{take}$ that returns a list containing the first $n$ elements of a possibly infinite list.

\[
\textit{take} : \mathbb{N} \to \text{pfiseq} X \to \text{pfiseq} X
\]

\[
\textit{take} n \bot = \bot
\]

\[
\textit{take} 0 \hs = []
\]

\[
\textit{take} n [] = []
\]

\[
\textit{take} n \hs = [\textit{head} \hs] \upshs (\textit{take} (n - 1) (\textit{tail} \hs))
\]

For a list $(x : \hs)$, the function $\textit{head}$ returns $x$ and the function $\textit{tail}$ returns $xs$. For a pair $(a, b)$, the function $\textit{first}$ returns $a$ and the function $\textit{second}$ returns $b$.  

105
4. The function $\circ$ applies a possibly infinite list of functions to a single argument.

$$\circ : \text{pfiseq}(X \rightarrow Y) \rightarrow X \rightarrow \text{pfiseq} Y$$

$$\bot^\circ x = []$$
$$[]^\circ x = []$$
$$[f]^\circ x = [f \, x]$$
$$(fs \leftarrow gs)^\circ x = fs^\circ x \leftarrow gs^\circ x$$

5. The function $\Pi$ is the distributed cartesian product for possibly infinite lists.

$$\Pi : \text{seq}(\text{pfiseq} X) \rightarrow \text{pfiseq}(\text{seq} X)$$

$$\Pi(xs) = e2l \ast xs$$
$$\Pi(xs : xss) = [(a : as) | a \leftarrow xs, as \leftarrow \Pi xss]$$

where

$$e2l \, x = [x]$$
Appendix C

Refinement Laws

We present the refinement laws of Morgan’s refinement calculus. For each law, we present its name, the arguments used for its application, the transformation of the program to which it is applied, and the provisos of its application.

**Law** \( \text{strPost}(\text{post}_2) \).
\[ w : [\text{pre}, \text{post}_1] \sqsubseteq w : [\text{pre}, \text{post}_2] \]
provided \( \text{post}_2 \Rightarrow \text{post}_1 \)

**Law** \( \text{weakPre}(\text{pre}_2) \).
\[ w : [\text{pre}_1, \text{post}] \sqsubseteq w : [\text{pre}_2, \text{post}] \]
provided \( \text{pre}_1 \Rightarrow \text{pre}_2 \)

**Law** \( \text{assign}(w := E) \).
\[ w : [\text{pre}, \text{post}] \sqsubseteq w := E \]
provided \( \text{pre} \Rightarrow \text{post}[w \setminus E] \)

**Law** \( \text{simpleEspec}() \).
\[ w : [w = E] = w := E \]
provided \( E \) does not contain \( w \)

**Law** \( \text{absAssump}(). \{ \text{pre'} \} w : [\text{pre}, \text{post}] = w : [\text{pre'} \land \text{pre}, \text{post}] \)

**Law** \( \text{skipIntro}(). w, x : [\text{pre}, \text{post}] \sqsubseteq \text{skip} \) provided \( w = w_0 \land \text{pre} \Rightarrow \text{post} \)

**Law** \( \text{seqComp}(\text{mid}) \).
\[ w : [\text{pre}, \text{post}] \sqsubseteq w : [\text{pre}, \text{mid}] ; w : [\text{mid}, \text{post}] \]
provided \( \text{mid} \) and \( \text{post} \) have no free initial variables.

**Law** \( \text{skipComp}() \).
For any program \( p \)
\[ \text{skip}; p = p; \text{skip} = p \]

**Law** \( \text{fassign}(x := E) \).
\[ w, x : [\text{pre}, \text{post}] \sqsubseteq w, x : [\text{pre}, \text{post}[x \setminus E]]; x := E \]
Law \textit{alt}(\langle G_0, \ldots, G_n \rangle). \quad w : [\text{pre}, \text{post}] \subseteq \textbf{if} \ (\ ? \ i. G_i \rightarrow w : [G_i \land \text{pre}, \text{post}]) \ \textbf{fi} \\
\text{provided} \ \text{pre} \Rightarrow G_0 \lor \ldots \lor G_n

Law \textit{altGuards}(\langle H_0, \ldots, H_n \rangle).
\text{Let} \ \GG = G_0 \lor \ldots \lor G_n \ \text{and} \ \HH = H_0 \lor \ldots \lor H_n.
\textbf{if} \ (\ ? \ i. G_i \rightarrow w : [G_i \land \text{pre}, \text{post}]) \ \textbf{fi} \subseteq \\
\textbf{if} \ (\ ? \ i. H_i \rightarrow w : [H_i \land \text{pre}, \text{post}]) \ \textbf{fi} \\
\text{provided} \ \GG \Rightarrow \HH \ \text{and for each} \ i, \ \GG \Rightarrow (H_i \Rightarrow G_i)

Law \textit{strPostIV}(\text{post}_2).
\text{w : [pre, post] \subseteq w : [pre, post_2]} \\
\text{provided} \ w \setminus w_0 \land \text{post}_2 \Rightarrow \text{post}_1

Law \textit{assignIV}(w := E).
\text{w, x : [pre, post] \subseteq w := E} \\
\text{provided} \ (w = w_0) \land \text{pre} \Rightarrow \text{post}[w \setminus E]

Law \textit{skipIntroIV}().
\text{w : [pre, post] \subseteq \textbf{skip}} \\
\text{provided} \ (w = w_0) \land \text{pre} \Rightarrow \text{post}

Law \textit{contractFrame}(x).
\text{w, x : [pre, post] \subseteq w : [pre, post[x_0 \setminus x]]}

Law \textit{iter}(\langle G_1, \ldots, G_n \rangle, V).
\text{Let} \ \textit{inv}, \ \text{the invariant}, \ \text{be any formula}; \ \text{Let} \ V, \ \text{the variant}, \ \text{be any} \ \text{integer} – \text{valued expression}. \ \text{Then, if} \ \GG, \ \text{is the disjunction of} \ \text{the guards,}
\text{w : [inv, inv \land \neg GG] \subseteq}
\textbf{do} \ (\ ? \ i. G_i \rightarrow w : [\text{inv \land G_i, inv \land 0} \leq V \leq V[w \setminus w_0]]) \ \textbf{od}
\text{Neither} \ \textit{inv} \ \text{nor} \ G_i \ \text{may contain} \ \text{initial variables.}

Law \textit{varInt}(n : T).
\text{w : [pre, post] \subseteq \llbracket \ \textbf{var} x : T \bullet w, x : [pre, post] \rrbracket} \\
\text{provided} \ x \ \text{does not occurs in} \ w, \ \text{pre}, \ \text{and} \ \text{post.}

Law \textit{conInt}(c : T, pre').
\text{w : [pre, post] \subseteq \llbracket \ \textbf{con} c : T \bullet w : [pre', post] \rrbracket} \\
\text{provided} \ \text{pre} \Rightarrow (\exists c : T \bullet \text{pre'}) \ \text{and} \ c \ \text{does not occurs in} \ w, \text{pre} \ \text{and} \ \text{post.}

Law \textit{fixInitValue}(c : T, E).
\text{For any term} \ E \text{ }
\text{w : [pre, post] \subseteq \llbracket \ \textbf{con} c : T \bullet w : [pre \land c = E, post] \rrbracket} \\
\text{provided} \ \text{pre} \Rightarrow E \in T \ \text{and} \ c \ \text{is a new name.}
Law \( \text{removeCon}(x) \).
\[ \begin{array}{c}
[ \text{con } c : T \bullet p ] \subseteq p \\
\text{provided } c \text{ does not occur in } p.
\end{array} \]

Law \( \text{expandFrame}(x) \).
\[ w : [\text{pre}, \text{post}] \subseteq w, x : [\text{pre}, \text{post} \land x = x_0] \]

Law \( \text{seqCompCon}(\text{mid}, X, x) \).
\[ w, x : [\text{pre}, \text{post}] \subseteq \\
[ \text{con } X \bullet x : [\text{pre}, \text{mid}]; w, x : [\text{mid}[x_0 \setminus X], \text{post}[x_0 \setminus X]] \\
\text{provided } \text{mid} \text{ does not contain other initial variables beyond } x_0. \]

Law \( \text{seqCompIV}(\text{mid}, x) \).
\[ w, x : [\text{pre}, \text{post}] \subseteq x : [\text{pre}, \text{mid}]; \ w, x : [\text{mid}, \text{post}] \\
\text{provided } \text{mid} \text{ does not contain any initial variables and } \text{post} \text{ does not contain } x_0. \]

Law \( \text{procNoArgsIntro}(\text{pn}, p_1). \) \( p_2 = [ \text{proc } \text{pn} = p_1 \bullet p_2] \)
\[ \text{provided } \text{pn} \text{ is not free in } p_2 \]

Law \( \text{procNoArgsCall}() \). \[ [ \text{proc } \text{pn} = p_1 \bullet p_2[p_1]] = [ \text{proc } \text{pn} = p_1 \bullet p_2[\text{pn}]] \]

Law \( \text{procArgsIntro}(\text{pn}, p_1, \text{par}). \) \( p_2 = [ \text{proc } \text{pn} = (\text{par } p_1) \bullet p_2] \)
\[ \text{provided } \text{pn} \text{ is not free in } p_2 \]

Law \( \text{procArgsCall}() \).
\[ [ \text{proc } \text{pn} = \\
(\text{par } p_1) \bullet p_2[(\text{par } p_1)(a))]] = [ \text{proc } \text{pn} = (\text{par } p_1) \bullet p_2[\text{pn}(a)]]] \]

Law \( \text{callByValue}(f, a) \).
\[ w : [\text{pre}[f \setminus a], \text{post}[f, f_0 \setminus a, a_0]] = \\
(\text{val } f \bullet w : [\text{pre}, \text{post}](a) \\
\text{provided } f \text{ is not in } w \text{ and } w \text{ is not free in } a \]

Law \( \text{callByValueIV}(f, a) \).
\[ w : [\text{pre}[f \setminus a], \text{post}[f_0 \setminus a_0]] = \\
(\text{val } f \bullet w, f : [\text{pre}, \text{post}](a) \\
\text{provided } f \text{ is not in } w \text{ and } w \text{ is not free in } \text{post} \]

Law \( \text{callByResult}(f, a) \). \( w, a : [\text{pre}, \text{post}] = (\text{res } f \bullet w, f : [\text{pre}, \text{post}[a \setminus f]])(a) \)
\[ \text{provided } f \text{ is not in } w, \text{ and is not free in } \text{pre or } \text{post} \text{ and, } f_0 \text{ is not free in } \text{post} \]

Law \( \text{multiArgs}(). \) \( (\text{par}_1 f_1 \bullet (\text{par}_2 f_2)(a_2))(a_1) = (\text{par}_1 f_1; \text{par}_2 f_2)(a_1, a_2) \)

Law \( \text{callByValueResult}(f, a) \).
\[ w, a : [\text{pre}[f \setminus a], \text{post}[f_0 \setminus a_0]] = (\text{val-res } f \bullet w, f : [\text{pre}, \text{post}[a \setminus f]])(a) \]
\[ \text{provided } f \text{ is not in } w, \text{ and is not free in } \text{post} \]

109
Law \( \text{variantIntro}(pr, p_1, n, e, \text{pars}) \).
\[
p_2 = \begin{bmatrix}
\text{proc } pr = (\text{par} \cdot w : [n = e \land \text{pre}, \text{post}])
\end{bmatrix}
\text{variant } n \text{ is } e \cdot p_2
\]
provided \( pr \) and \( n \) are not free in \( e \) and \( p_2 \)

Law \( \text{procVariantBlockCall}() \).
\[
[\begin{bmatrix}
\text{proc } pr = (\text{par} \cdot p_1) \text{ variant } n \text{ is } e \cdot p_2[[\text{par} \cdot p_3(a)]]
\end{bmatrix}]
\]
\[
= [\begin{bmatrix}
\text{proc } pr = (\text{par} \cdot p_1) \text{ variant } n \text{ is } e \cdot p_2[(pr)(a)]
\end{bmatrix}]
\]
provided \( pr \) is not recursive, \( n \) is not free in \( e \) and \( p_3 \), and \( \{n = e\} p_3 \sqsubseteq p_1 \)

Law \( \text{recursiveCall}() \).
\[
[\begin{bmatrix}
\text{proc } pr = (\text{par} \cdot p_1[[\text{par} \cdot \{0 \leq e < n\} p_3(a)]]) \text{ variant } n \text{ is } e \cdot p_2
\end{bmatrix}]
\]
\[
= [\begin{bmatrix}
\text{proc } pr = (\text{par} \cdot p_1[pr(a)]) \text{ variant } n \text{ is } e \cdot p_2
\end{bmatrix}]
\]
provided \( n \) is not free in \( p_3 \) and \( p_1[pr(a)] \), and \( \{n = e\} p_3 \sqsubseteq p_1[[\text{par} \cdot \{0 \leq e < n\} p_3(a)]] \)

Law \( \text{variantNoArgsIntro}(pr, p_1, n, e) \).
\[
p_2 = [\begin{bmatrix}
\text{proc } pr = w : [n = e \land \text{pre}, \text{post}] \text{ variant } n \text{ is } e \cdot p_2
\end{bmatrix}]
\]
provided \( pr \) and \( n \) are not free in \( e \) and \( p_2 \)

Law \( \text{procVariantBlockCallNoArgs}() \).
\[
[\begin{bmatrix}
\text{proc } pr = w : [n = e \land \text{pre}, \text{post}] \text{ variant } n \text{ is } e \cdot p_2[pr]
\end{bmatrix}]
\]
\[
\sqsubseteq [\begin{bmatrix}
\text{proc } pr = w : [n = e \land \text{pre}, \text{post}] \text{ variant } n \text{ is } e \cdot p_2[pr]
\end{bmatrix}]
\]
provided \( nm \) is not recursive, \( n \) is not free in \( w \) and \( p_1[pr] \) and \( w : [n = e \land \text{pre}, \text{post}] \sqsubseteq p_1 \).

Law \( \text{recursiveCallNoArgs}() \).
\[
[\begin{bmatrix}
\text{proc } pr = p_1[w : [0 \leq e < n \land \text{pre}, \text{post}]] \text{ variant } n \text{ is } e \cdot p_2
\end{bmatrix}]
\]
\[
\sqsubseteq [\begin{bmatrix}
\text{proc } pr = p_1[pr(a)] \text{ variant } n \text{ is } e \cdot p_2
\end{bmatrix}]
\]
provided \( n \) is not free in \( w : [\text{pre}, \text{post}] \) and \( p_1[pr] \) and \( w : [n = e \land \text{pre}, \text{post}] \sqsubseteq p_1 \).

Law \( \text{coercion}() \).
\[
[\begin{bmatrix}
\text{post} = : [\text{true}, \text{post}]
\end{bmatrix}]
\]
provided \( \text{post} \) does not contain any initial variables.

Law \( \text{absCoercion}() \).
\[
\begin{bmatrix}
\text{post} = : [\text{true}, \text{post}]
\end{bmatrix}
\]
\[
\sqsubseteq [\begin{bmatrix}
\text{post} = : [\text{true}, \text{post}]
\end{bmatrix}]
\]

Law \( \text{intCoercion}() \).
\[
\begin{bmatrix}
\text{skip} \sqsubseteq [\text{post}]
\end{bmatrix}
\]
Appendix D

Proofs of the Laws

In this Appendix we prove the laws presented in this thesis. The laws marked with * are not valid with infinite lists and are proved only for finite lists. The lemmas used are presented and proved in the Appendix D.6.

D.1 Basic properties of composition

Law 1(a). \( \text{skip} \; ; \; t = t \)

\[
\begin{align*}
\llbracket \text{skip} \; ; \; t \rrbracket & = \infty/ \bullet (\llbracket t \rrbracket \Gamma_L \Gamma_T) \bullet \llbracket \text{skip} \rrbracket \Gamma_L \Gamma_T r \\
& = \infty/(((\llbracket t \rrbracket \Gamma_L \Gamma_T) \bullet (\llbracket \text{skip} \rrbracket \Gamma_L \Gamma_T r)) \\
& = \infty/((\llbracket t \rrbracket \Gamma_L \Gamma_T) \bullet [r]) \\
& = \infty/[[t]] \Gamma_L \Gamma_T r \\
& = [t] \Gamma_L \Gamma_T r
\end{align*}
\]

[Definition of ; ]
[Functional Composition]
[Definition of skip]
[Definition of *]
[Property of \( \infty/ \)]

\[\square\]

Law 1(b). \( t = \text{skip} \; ; \; t \)

\[
\begin{align*}
\llbracket \text{skip} \; ; \; t \rrbracket & = \infty/ \bullet (\llbracket t \rrbracket \Gamma_L \Gamma_T) \bullet \llbracket \text{skip} \rrbracket \Gamma_L \Gamma_T r \\
& = \infty/(((\llbracket t \rrbracket \Gamma_L \Gamma_T) \bullet (\llbracket \text{skip} \rrbracket \Gamma_L \Gamma_T r)) \\
& = \infty/((\llbracket t \rrbracket \Gamma_L \Gamma_T) \bullet [r]) \\
& = \infty/[[t]] \Gamma_L \Gamma_T r \\
& = [t] \Gamma_L \Gamma_T r
\end{align*}
\]

[Definition of ; ]
[Functional Composition]
[Definition of skip]
[Definition of *]
[Property of \( \infty/ \)]

\[\square\]
Law 2(a). \( t \mid \text{fail} = t \)

\[
\begin{align*}
\text{[(} t \mid \text{fail} \text{)]}_L \Gamma_T r &= (\infty / (\text{[(} t \text{)]}_L \Gamma_T), (\text{[(} \text{fail} \text{)]}_L \Gamma_T)^\circ) r \\
&= \infty / (\text{[(} t \text{)]}_L \Gamma_T r), (\text{[(} \text{fail} \text{)]}_L \Gamma_T r)
\end{align*}
\]

[Definition of [( ]]
[Definition of \( \circ \)]
[Functional Composition]
[Definition of fail]
[Definition of \( \infty / \)]

\[
\begin{align*}
&= \infty / (\text{[(} t \text{)]}_L \Gamma_T r, [ ])
&= \bigcup_{\infty} \{ c : (\text{[(} t \text{)]}_L \Gamma_T r, [ ] \bullet \infty / c \}
&= \bigcup_{\infty} \{ \infty / (f, < [ ] \Gamma_T r, [ ] >) \}
&= \bigcup_{\infty} \{ \infty / (f, < [ ] \Gamma_T r > \infty / (f, < [ ] >) \}
&= \bigcup_{\infty} \{ [ ] \Gamma_T r \}
&= \bigcup_{\infty} (\text{[(} t \text{)]}_L \Gamma_T)
&= \text{[(} t \text{)]}_L \Gamma_T r
\end{align*}
\]

\[\square\]

Law 2(b). \( t = \text{fail} \mid t \)

\[
\begin{align*}
\text{[(} t \text{)]}_L \Gamma_T r &= \bigcup_{\infty} (\text{[(} t \text{)]}_L \Gamma_T r)
&= \bigcup_{\infty} \{ (\text{[(} t \text{)]}_L \Gamma_T r, [ ] \bullet (s, l) \}
&= \bigcup_{\infty} \{ (s, l) : (\text{[(} t \text{)]}_L \Gamma_T r \bullet (s, l) ]^< \infty (s, l))\}
&= \bigcup_{\infty} \{ x : (\text{[(} t \text{)]}_L \Gamma_T r \bullet (f, >)^^< x) \}
&= \bigcup_{\infty} \{ x : (\text{[(} t \text{)]}_L \Gamma_T r ; y : [ ] \bullet y^< x) \}
&= \bigcup_{\infty} \{ [ ] \Gamma_T r \}
&= \bigcup_{\infty} \{ \infty / (f, < [ ] ) \infty / (f, < [ ] \Gamma_T r ) \}
&= \bigcup_{\infty} \{ c : [ ], (\text{[(} t \text{)]}_L \Gamma_T r) \bullet \infty / c \}
&= \infty / (\text{[(} \text{fail} \text{)]}_L \Gamma_T r, (\text{[(} t \text{)]}_L \Gamma_T r) r)
&= \infty / (\text{[(} \text{fail} \text{)]}_L \Gamma_T r, (\text{[(} \text{fail} \text{)]}_L \Gamma_T)^\circ r)
&= \text{[(} t \text{)]}_L \Gamma_T r
\end{align*}
\]

\[\square\]
Law 3(a)∗. \( t; \, \text{fail} = \text{fail} \)

\[
\left\lbrack t; \, \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T} r
= \infty \cdot (\left\lbrack \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T}) \ast (\left\lbrack t \right\rbrack_{\Gamma_L \ \Gamma_T}) \ r
\]

[Definition of ; ]

[Functional Composition]

[Definition of fail and *]

where

\[
\left\lbrack \text{lst} \right\rbrack^0 = [ ]
\]

\[
\left\lbrack \text{lst} \right\rbrack^n = [\text{lst}] \infty ([\text{lst}]^{n-1})
\]

As \( \left\lbrack \left\lbrack t \right\rbrack_{\Gamma_L \ \Gamma_T} \right\rbrack r \) is a finite list, we follow as

\[
= [ ]
\]

\[
= (\left\lbrack \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T}) \ r
\]

[Definition of \( \infty / \)]

[Definition of fail]

\[
\square
\]

Law 3(b). \( \text{fail} = t; \, \text{fail} \)

\[
\left\lbrack \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T} r = [ ]
\]

[Definition of fail]

[Property of \( \infty / \)]

[Definition of *]

\[
= \infty / (\left\lbrack \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T}) \ast [ ]
\]

[Definition of fail]

[Definition of ; ]

\[
= \infty / \cdot (\left\lbrack \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T}) \ast \cdot (\left\lbrack \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T}) \ r
\]

\[
= \left\lbrack t; \, \text{fail} \right\rbrack_{\Gamma_L \ \Gamma_T} r
\]

[Definition of \( \infty / \)]

\[
\square
\]

Law 4. \( t_1 \mid (t_2 \mid t_3) = (t_1 \mid t_2) \mid t_3 \)

\[
\left\lbrack t_1 \mid (t_2 \mid t_3) \right\rbrack_{\Gamma_L \ \Gamma_T} r
= \infty / (\left\lbrack t_3 \right\rbrack_{\Gamma_L \ \Gamma_T} ; ((\left\lbrack t_2 \mid t_3 \right\rbrack_{\Gamma_L \ \Gamma_T} \ast \left\lbrack t_3 \right\rbrack_{\Gamma_L \ \Gamma_T}) \circ r)
\]

[Definition of | ]

[Functional Composition]

[Lemma D.6.17]

[Definition of | ]

[Functional Composition]

[Lemma D.6.17]

[Lemma D.6.24]

[Definition of \( \infty / \) and \( \circ \)]

[Definition of | ]

[Functional Composition]

[Definition of \( \infty / \) and \( \circ \)]

[Definition of | ]

\[
\square
\]

113
Law 5. \( t_1; (t_2; t_3) = (t_1; t_2); t_3 \)

\[
\begin{align*}
(\langle t_1; (t_2; t_3) \rangle)_{\mathcal{L}} \Gamma_T & = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Property of \( \ast \)]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Lemma D.6.2]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Property of \( \ast \) and \( \ast \)]} \\
& = (\langle t_1; (t_2; t_3) \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]}
\end{align*}
\]

Law 6. \( (t_1 \mid t_2); t_3 = (t_1; t_3) \mid (t_2; t_3) \)

\[
\begin{align*}
(\langle t_1 \mid t_2 \rangle; t_3)_{\mathcal{L}} \Gamma_T & = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Property of \( \ast \)]} \\
& = \infty \cdot (\langle t_2; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle t_1; (t_2; t_3) \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1; t_2 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle t_1; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle t_1; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle t_1; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]}
\end{align*}
\]

Law 6'. \( (t_1 \mid i; t_3); t_3 = (t_1 \mid i; t_3) \mid (t_1 \mid i; t_3) \)

\[
\begin{align*}
(\langle t_1 \mid i; t_3 \rangle; t_3)_{\mathcal{L}} \Gamma_T & = \infty \cdot (\langle t_3; t_1 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = \infty \cdot (\langle t_3; t_1 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = \infty \cdot (\langle t_3; t_1 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Property of \( \ast \)]} \\
& = \infty \cdot (\langle t_3; t_1 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T \ast \cdot (\langle t_1 \mid i; t_3 \rangle)_{\mathcal{L}} \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle (t_3; t_1) \rangle_{\mathcal{L}} \Gamma_T) \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle (t_3; t_1) \rangle_{\mathcal{L}} \Gamma_T) \Gamma_T & \text{[Definition of \( \ast \)]} \\
& = (\langle (t_3; t_1) \rangle_{\mathcal{L}} \Gamma_T) \Gamma_T & \text{[Definition of \( \ast \)]}
\end{align*}
\]
Law 7. \( t_1; (t_2 \mid t_3) = (t_1; t_2) \mid (t_1; t_3) \) for any sequential tactic.

\[
\begin{align*}
[[t_1; (t_2 \mid t_3)][\Gamma_L \Gamma_T r] &= \infty/ \cdot ([[t_2 \mid t_3][\Gamma_L \Gamma_T r]) \star ([[]t_1][\Gamma_L \Gamma_T r]) & \text{[Definition of ;]} \\
&= \infty/ \cdot (\infty/ \cdot ([[t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r])) \star ([[]t_1][\Gamma_L \Gamma_T r]) & \text{[Definition of !]} \\
&= \infty/ \cdot ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) \star ([[]t_1][\Gamma_L \Gamma_T r]) & \text{[Definition of \( \circ \) and \( * \)]} \\
&= \infty/ \cdot ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) \star ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) & \text{[Definition of \( \circ \)} \\
&= [[(t_1; t_2) \mid (t_1; t_3)][\Gamma_L \Gamma_T r] & \text{[Definition of !]}
\end{align*}
\]

\( \square \)

Law 7'. \( t_1; (\mid_{\nu, I} t_1) = (\mid_{\nu, I} t_1; t_4) \) for any sequential tactic.

\[
\begin{align*}
[[t_1; (\mid_{\nu, I} t_1)][\Gamma_L \Gamma_T r] &= \infty/ \cdot (\infty/ \cdot ([[]t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r])) \star ([[]t_1][\Gamma_L \Gamma_T r]) & \text{[Definition of ;]} \\
&= \infty/ \cdot (\infty/ \cdot ([[]t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r])) \star ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) & \text{[Definition of !]} \\
&= \infty/ \cdot ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) \star ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) & \text{[Definition of \( \circ \) and \( * \)]} \\
&= \infty/ \cdot ([[]t_1; t_2][\Gamma_L \Gamma_T r, [t_3][\Gamma_L \Gamma_T r]) \star ([[]t_1; t_2; t_4][\Gamma_L \Gamma_T r]) & \text{[Definition of \( \circ \)} \\
&= [[(\mid_{\nu, I} t_1; t_4)][\Gamma_L \Gamma_T r] & \text{[Definition of !]}
\end{align*}
\]

\( \square \)

D.2 Laws involving Cut

Law 8. \( !\text{skip} = \text{skip} \)

\[
\begin{align*}
[[!\text{skip}][\Gamma_L \Gamma_T r] &= head'' \cdot [[\text{skip}][\Gamma_L \Gamma_T r] & \text{[Definition of !]} \\
&= head''[r] & \text{[Definition of \text{skip}]} \\
&= [r] & \text{[Definition of head'']} \\
&= [[\text{skip}][\Gamma_L \Gamma_T r] & \text{[Definition of \text{skip}]}
\end{align*}
\]

\( \square \)
Law 9. \( \text{!fail} = \text{fail} \)

\[
\begin{align*}
[\text{!fail}] &\Gamma_L \Gamma_T r \\
= &\text{head'} \cdot [\text{fail}] \Gamma_L \Gamma_T r \\
= &\text{head'}[1] \\
= &[1] \\
= &\text{!fail}] \Gamma_L \Gamma_T r
\end{align*}
\]

[Definition of !] [Definition of fail] [Definition of head'] [Definition of fail]

\[\square\]

Law 10. \( \text{!(law } l \ (\text{args})) = \text{law } l \ (\text{args}) \)

\[
\begin{align*}
[\text{! law } l \ (\text{args})] &\Gamma_L \Gamma_T r \\
= &\text{head'} \cdot ([\text{law } l \ (\text{args})] \Gamma_L \Gamma_T r) \\
= &\text{head'}([\text{law } l \ (\text{args})] \Gamma_L \Gamma_T r) \\
= &\text{law } l \ (\text{args})] \Gamma_L \Gamma_T r
\end{align*}
\]

[Definition of !] [Functional Composition] [Lemma D.6.20] [Definition of head']

\[\square\]

Law 11. \( \text{!t}_1; (t_2 \mid t_3) = (\text{!t}_1; t_2) \mid (\text{!t}_1; t_3) \)

\[
\begin{align*}
[\text{!t}_1; \ (t_2 \mid t_3)] &\Gamma_L \Gamma_T r \\
= &([[(t_2 \mid t_3)] \Gamma_L \Gamma_T r] \circ \cdot [\text{!t}_1]] \Gamma_L \Gamma_T r \\
= &([[(t_2 \mid t_3)] \Gamma_L \Gamma_T r] \circ \cdot [\text{!t}_1]] \Gamma_L \Gamma_T r \\
= &([[(t_2 \mid t_3)] \Gamma_L \Gamma_T r] \circ \cdot [\text{!t}_1]] \Gamma_L \Gamma_T r)
\end{align*}
\]

[Definition of ;] [Definition of ] [Definition of \( \circ \) and \( \cdot \)] [Definition of ;] [Definition of \( \circ \) and \( \cdot \)] [Definition of ]

\[\square\]

Law 11'. \( \text{!t}_1; (\mid i : t_i) = \mid i : t_i; t_i \)

\[
\begin{align*}
[\text{!t}_1; \ (\mid i : t_i)] &\Gamma_L \Gamma_T r \\
= &([[(\mid i : t_i)] \Gamma_L \Gamma_T r] \circ \cdot [\text{!t}_1]] \Gamma_L \Gamma_T r \\
= &([[(\mid i : t_i)] \Gamma_L \Gamma_T r] \circ \cdot [\text{!t}_1]] \Gamma_L \Gamma_T r)
\end{align*}
\]

[Definition of ;] [Definition of ]

116
\[ = (\infty / \bullet (\llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} \cdot (\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} r)) \infty \ldots \infty (\llbracket t_n \rrbracket_{\Gamma_L \Gamma_T} \cdot (\llbracket t_r \rrbracket_{\Gamma_L \Gamma_T} r)) \]

[Definition of \( \circ \) and \( \ast \)]

\[ = (\infty / \bullet (\llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} \cdot (\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} r)) \infty \ldots \infty (\infty / \bullet (\llbracket t_n \rrbracket_{\Gamma_L \Gamma_T} \cdot (\llbracket t_r \rrbracket_{\Gamma_L \Gamma_T} r))) \]

[Lemma D.6.5]

\[ = (\mathcal{G}t_1; 2 \llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T}) \infty \ldots \infty (\mathcal{G}t_1; 2 \llbracket t_n \rrbracket_{\Gamma_L \Gamma_T} r) \]

[Definition of \( ; \)]

\[ = \mathcal{G}t_1; (\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T}) \cdot \ldots \cdot (\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} r) \]

[Definition of \( \infty / \) and \( \circ \)]

\[ = \mathcal{G}t_1; \mathcal{G}t_r \llbracket \Gamma_L \Gamma_T r \]

[Definition of \( \| \)]

\[ \]

\[ \square \]

**Law 12.** \( !t_1; !t_2 = !(\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} \cdot !t_2) \)

\[ \llbracket !t_1; !t_2 \rrbracket_{\Gamma_L \Gamma_T} r \]

\[ = \infty / \bullet (\llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} \cdot !t_1) \llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} r \]

[Definition of \( ; \)]

\[ = \infty / \bullet (\llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} \cdot !t_1) \llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} r \]

[Definition of \( \mid \)]

\[ = \infty / \bullet (\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} \cdot \llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} r) \]

[Definition of \( \circ \)]

\[ = \infty / \bullet (\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} \cdot \llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} r) \]

[Definition of \( \ast \)]

\[ = \llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} \cdot \llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T} r \]

[Definition of \( \mathcal{G} \)]

\[ = \mathcal{G}t_1; !t_2 \llbracket \Gamma_L \Gamma_T r \]

[Definition of \( \| \)]

\[ \square \]
**Law 13.** \( ! (t_1; t_2) = ! (t_1; ! t_2) \)

\[
[![t_1; t_2)] \Gamma_L \Gamma_T r
= \text{head}'([![t_1; t_2)] \Gamma_L \Gamma_T r)
= \text{head}'(\infty/ \bullet ([![t_2)] \Gamma_L \Gamma_T) \ast [t_1)] \Gamma_L \Gamma_T r)
= [\text{head}(\text{head}'([![t_2)] \Gamma_L \Gamma_T) \ast [t_1)] \Gamma_L \Gamma_T r)]
= \text{head}'(\text{head}'([![t_2)] \Gamma_L \Gamma_T) \ast [t_1)] \Gamma_L \Gamma_T r)]
= \text{head}'([![t_1; ! t_2)] \Gamma_L \Gamma_T r)
= [![t_1; ! t_2)] \Gamma_L \Gamma_T r
\]

[Definition of !]
[Definition of ;]
[Definition of head']
[Definition of \( \infty/ \)]
[Definition of head']
[Definition of \( \infty/ \)]
[Definition of head']
[Definition of !]
[Definition of ;]
[Definition of !]

\[\square\]

**Law 14(a).** \( ! (t_1 | t_2) = ! (t_1 | t_2) \)

\[
[![t_1 | t_2)] \Gamma_L \Gamma_T r
= \text{head}'([![t_1 | t_2)] \Gamma_L \Gamma_T r)
= \text{head}'(\infty/ \bullet ([![t_1)] \Gamma_L \Gamma_T) \ast [t_2)] \Gamma_L \Gamma_T r^{\circ} r)
= \text{head}'(\infty/([![t_1)] \Gamma_L \Gamma_T r) \ast [t_2)] \Gamma_L \Gamma_T r)
= \text{head}'([![t_1)] \Gamma_L \Gamma_T r) \ast [t_2)] \Gamma_L \Gamma_T r)
= [\text{head}'([![t_1)] \Gamma_L \Gamma_T r) \ast [t_2)] \Gamma_L \Gamma_T r)]
= \text{head}'(\infty/ [![t_1)] \Gamma_L \Gamma_T r) \ast [t_2)] \Gamma_L \Gamma_T r)
= [\text{head}'([![t_1)] \Gamma_L \Gamma_T r) \ast [t_2)] \Gamma_L \Gamma_T r)]
\]

[Definition of !]
[Definition of head', \( \infty/ \)]
[Assumption]
[Definition of \( \infty/ \)]
[Definition of head']
[Definition of !, \( \infty/ \)]

Case \([![t_1)] \Gamma_L \Gamma_T r = [ \]

\[
= [\text{head} ([![t_1)] \Gamma_L \Gamma_T r)]
= [\text{head} ((\text{head}'([![t_1)] \Gamma_L \Gamma_T r)) \ast [t_2)] \Gamma_L \Gamma_T r)]
= \text{head}'(\infty/ [\text{head}'([![t_1)] \Gamma_L \Gamma_T r)] \ast [t_2)] \Gamma_L \Gamma_T r)
= \text{head}'(\infty/ [\text{head}'([![t_1)] \Gamma_L \Gamma_T r)] \ast [t_2)] \Gamma_L \Gamma_T r)
= [\text{head}'([![t_1)] \Gamma_L \Gamma_T r) \ast [t_2)] \Gamma_L \Gamma_T r]
\]

[Definition of !]
[Definition of head']
[Definition of \( \infty/ \)]
[Definition of head']
[Definition of !]
[Definition of !]
[Definition of !]

Case \([![t_1)] \Gamma_L \Gamma_T r \neq [ \]

\[
= [\text{head} ([\text{head}'([![t_1)] \Gamma_L \Gamma_T r)]) \ast [t_2)] \Gamma_L \Gamma_T r)]
= \text{head}'([\text{head}'([![t_1)] \Gamma_L \Gamma_T r)]) \ast [t_2)] \Gamma_L \Gamma_T r)]
= \text{head}'(\infty/ [\text{head}'([![t_1)] \Gamma_L \Gamma_T r)]) \ast [t_2)] \Gamma_L \Gamma_T r)
= \text{head}'(\infty/ \bullet [\text{head}'([![t_1)] \Gamma_L \Gamma_T r)]) \ast [t_2)] \Gamma_L \Gamma_T r)
= \text{head}'([![t_1)] \Gamma_L \Gamma_T r)
= [\text{head}'([![t_1)] \Gamma_L \Gamma_T r)]
\]

[Lemma D.6.22]
[Definition of head']
[Definition of \( \infty/ \)]
[Definition of head']
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]

\[\square\]

118
Law 14(b). \( ! (t_1 \mid t_2) = ! (t_1 \mid ! t_2) \)

\[
\begin{align*}
[! (t_1 \mid t_2)] & \Gamma L \Gamma T r \\
& = head'([t_1 \mid t_2]) \Gamma L \Gamma T r \\
& = head'(\infty \cdot ([t_1]) \Gamma L \Gamma T, [t_2]) \Gamma L \Gamma T r^o r) \\
& = head'(\infty / ([t_1]) \Gamma L \Gamma T r, [t_2]) \Gamma L \Gamma T r \\
& = head'(([t_1]) \Gamma L \Gamma T r \infty ([t_2]) \Gamma L \Gamma T r) \\
& = [head((t_1)] \Gamma L \Gamma T r) \infty ([t_2]) \Gamma L \Gamma T r)) \\
\end{align*}
\]

Case \([t_2] \Gamma L \Gamma T r \neq [] \)

\[
\begin{align*}
& = [\text{head}([t_1]) \Gamma L \Gamma T r) \infty [\text{head}([t_2]) \Gamma L \Gamma T r)]) \\
& = head'(\infty / ([t_1]) \Gamma L \Gamma T r, head'([t_2]) \Gamma L \Gamma T r)) \\
& = head'(\infty / ([t_1]) \Gamma L \Gamma T r, head'([t_2]) \Gamma L \Gamma T) r^o r) \\
& = [!!(t_1 \mid t_2)] \Gamma L \Gamma T r \\
\end{align*}
\]

Law 15. \( ! (t_1 \mid t_1; t_2) = ! t_1 \)

\[
\begin{align*}
[! (t_1 \mid t_1; t_2)] & \Gamma L \Gamma T r \\
& = head'([t_1 \mid t_1; t_2]) \Gamma L \Gamma T r \\
& = head'(\infty / \cdot ([t_1]) \Gamma L \Gamma T, [t_1; t_2]) \Gamma L \Gamma T) r^o r) \\
& = head'(\infty / \cdot ([t_1]) \Gamma L \Gamma T, \infty / \cdot ([t_2]) \Gamma L \Gamma T) r^\circ r) \\
& = head'(\infty / \cdot ([t_1]) \Gamma L \Gamma T, \infty / \cdot ([t_2]) \Gamma L \Gamma T r, [t_1]) \Gamma L \Gamma T r) \\
& = head'(\infty / \cdot ([t_1]) \Gamma L \Gamma T) r^o ([t_2]) \Gamma L \Gamma T) r, [t_1]) \Gamma L \Gamma T r) \\
& = head'(\infty / \cdot ([t_1]) \Gamma L \Gamma T) r^\circ ([t_2]) \Gamma L \Gamma T) r, [t_1]) \Gamma L \Gamma T r) \\
& = head'(([t_1]) \Gamma L \Gamma T) r^\circ ([t_2]) \Gamma L \Gamma T) r, [t_1]) \Gamma L \Gamma T r) \\
& = [t_1]) \Gamma L \Gamma T r \\
\end{align*}
\]

\( \square \)

119
Law 16. \( !(t_1 \mid t_2 \mid t_1) = !(t_1 \mid t_2) \)

\[
\llbracket !(t_1 \mid t_2 \mid t_1) \rrbracket_{\Gamma_L \Gamma_T} = head'(\llbracket (t_1 \mid t_2 \mid t_1) \rrbracket_{\Gamma_L \Gamma_T}) = head'((\star\star\star / \bullet [\llbracket t_1 \mid t_2 \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r)) = head'((\star\star\star / [\star\star\star / \bullet [\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r, [\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T}]) \rrbracket_{\Gamma_L \Gamma_T} r) = head'((\star\star\star / [\star\star\star / \bullet [\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r, [\llbracket t_1 \rrbracket_{\Gamma_L \Gamma_T}]) \rrbracket_{\Gamma_L \Gamma_T} r) = head'((\llbracket t_1 \mid t_2 \rrbracket_{\Gamma_L \Gamma_T} r) = \llbracket !(t_1 \mid t_2) \rrbracket_{\Gamma_L \Gamma_T} \]

[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]

\[\square\]

Law 17. \( !(\text{skip} \mid t) = \text{skip} \)

\[
\llbracket !(\text{skip} \mid t) \rrbracket_{\Gamma_L \Gamma_T} = head'((\llbracket \text{skip} \mid t \rrbracket_{\Gamma_L \Gamma_T}) = head'((\star\star\star / \bullet [\llbracket \text{skip} \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r)) = head'((\star\star\star / [\star\star\star / \bullet [\llbracket \text{skip} \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r, [\llbracket \text{skip} \rrbracket_{\Gamma_L \Gamma_T}]) \rrbracket_{\Gamma_L \Gamma_T} r) = head'((\llbracket \text{skip} \rrbracket_{\Gamma_L \Gamma_T} \rrbracket_{\Gamma_L \Gamma_T} r) = \llbracket \text{skip} \rrbracket_{\Gamma_L \Gamma_T} \]

[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]

\[\square\]

Law 18. \( !(t \mid t) = t \)

\[
\llbracket !(t \mid t) \rrbracket_{\Gamma_L \Gamma_T} = head'((\llbracket t \mid t \rrbracket_{\Gamma_L \Gamma_T}) = head'((\star\star\star / \bullet [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r)) = head'((\star\star\star / [\star\star\star / \bullet [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}, [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}]^\circ, r, [\llbracket t \rrbracket_{\Gamma_L \Gamma_T}]) \rrbracket_{\Gamma_L \Gamma_T} r) = head'([\llbracket t \rrbracket_{\Gamma_L \Gamma_T} \rrbracket_{\Gamma_L \Gamma_T} r) = \llbracket t \rrbracket_{\Gamma_L \Gamma_T} \]

[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
[Definition of !]
\[ \text{Definition of head} \]
\[ \text{Definition of head'} \]
\[ \text{Definition of !} \]
\[ \text{Definition of !} \]

Law 19. \[ \text{!!t =!t} \]

\[ \text{!!t r =!t r} \]
\[ \text{Definition of !} \]
\[ \text{Lemma D.6.15} \]
\[ \text{Definition of !} \]

\[ \text{Assumption} \]

D.3 Laws involving succs and fails

In this section we use the notation

\[ t_1 \triangleleft \text{condition} \triangleright t_2 = \text{if condition then } t_1 \text{ else } t_2 \]

Law 20. succs \( t; t = t \)

\[ \text{succs } t; t r \]
\[ = \infty / \bullet t * \bullet \text{succs } t r \]
\[ \text{Definition of ; } \]

Case \( t r = \bot \)
\[ = \infty / \bullet t * \bot \]
\[ = \bot \]
\[ = t r \]
\[ \text{Definition of succs} \]
\[ \text{Definition of fail} \]
\[ \text{Assumption} \]

Case \( t r = [ ] \)
\[ = \infty / \bullet t * [ ] \]
\[ = \bot / [ ] \]
\[ = [ ] \]
\[ = t r \]
\[ \text{Definition of succs} \]
\[ \text{Definition of succs} \]
\[ \text{Definition of *} \]
\[ \text{Definition of } \infty / \}
\[ \text{Assumption} \]

Case \( t r \neq [ ] \)
\[ = \infty / \bullet t * \bullet \text{skip } r \]
\[ \text{Definition of succs} \]
\[= \infty/ \bullet t * [r] \quad \text{[Definition of \texttt{skip}]}\]
\[= \infty/[t r] \quad \text{[Definition of \texttt{*}]}\]
\[= t r \quad \text{[Definition of \texttt{\infty/}]}\]

Law 21. \* fails \(t\); \(t = \text{fail}\)

\texttt{fails} \(t\); \(t r\)
\[= \infty/ \bullet t * \bullet \texttt{fails} t r \quad \text{[Definition of \texttt{; \}}\]

Case \(t r =<>\)
\[= \infty/ \bullet t * \bullet \texttt{skip} r \quad \text{[Definition of \texttt{succs}]}\]
\[= \infty/ \bullet t * [r] \quad \text{[Definition of \texttt{skip}]}\]
\[= \infty/ \bullet [t r] \quad \text{[Definition of \texttt{*}]}\]
\[= t r \quad \text{[Definition of \texttt{\infty/}]}\]
\[= [ ] \quad \text{[Assumption]}\]
\[= \texttt{fail} r \quad \text{[Definition of \texttt{fail}]}\]

Case \(t r \neq [ ]\)
\[= \infty/ \bullet t * \bullet \texttt{fail} r \quad \text{[Definition of \texttt{fails}]}\]
\[= \infty/[ ] \quad \text{[Definition of \texttt{*}]}\]
\[= \texttt{fail} r \quad \text{[Definition of \texttt{\infty/}]}\]

\[\rlap{\Box}\]

Law 22. \texttt{fails} \(t\) = \texttt{fails} \(!t\) = \texttt{fails} \(t\)

1. \texttt{fails} \(t\) = \texttt{fails} \(!t\)
\[\texttt{fails} \(t r\)\]

Case \(t r = [ ]\)
\[= \texttt{skip} r \quad \text{[Definition of \texttt{fails}]}\]
\[= \texttt{fails}(\texttt{head'}(t r)) \quad \text{[Definition of \texttt{fails}]}\]
\[= \texttt{fails}(!t r) \quad \text{[Assumption]}\]
\[= \texttt{fails}(!t r) \quad \text{[Definition of \texttt{!}]}\]

Case \(t r \neq [ ]\)
\[= \texttt{fail} r \quad \text{[Definition of \texttt{fails}]}\]
= \textit{fails}(\textit{head}'(t \ r))

= \textit{fails}(!t \ r)

Case $t \ r = \bot$

= $\bot$

= $\textit{fails}(!t \ r)$

2. $\textit{fails} \ !t = \textit{!fails} \ t \ r$

$\textit{fails} \ !t$

Case $t \ r = [ ]$

= $(\text{skip} < !t \ r = [ ] \triangleright \text{fail}) \ r$

= $(\text{skip} < \textit{head}'(t \ r) = [ ] \triangleright \text{fail}) \ r$

= $(\text{skip} < \textit{head}'([ ])) = [ ] \triangleright \text{fail}) \ r$

= $\text{skip} \ r$

= $\text{!skip} \ r$

= $!(\text{fails} \ t \ r)$

Case $t \ r = \text{lst} \neq [ ]$

= $(\text{skip} < !t \ r = [ ] \triangleright \text{fail}) \ r$

= $(\text{skip} < \textit{head}'(\text{lst}) = [ ] \triangleright \text{fail}) \ r$

= $\text{fail} \ r$

= $\text{!fail} \ r$

= $!(\text{fails} \ t \ r)$

Case $t \ r = \bot$

= $\bot$

= $!(\text{fails} \ t \ r)$
Law 23. \( \text{fails}(\text{succs } t) = \text{fails } t \)

\[ \text{fails}(\text{succs } t) r \]

Case \( t r = [ ] \)
\[ \text{fails}(\text{succs } t) r \]
\[ = (\text{skip } < \text{succs } t r = [ ] > \text{fail}) r \]
\[ = (\text{skip } < \text{fail } r = [ ] > \text{fail}) r \]
\[ = (\text{skip } < [ ] = [ ] > \text{fail}) r \]
\[ = \text{skip } r \]
\[ = \text{fails } t r \]

Case \( t r \neq [ ] \)
\[ = (\text{skip } < \text{succs } t r = [ ] > \text{fail}) r \]
\[ = (\text{skip } < \text{skip } r = [ ] > \text{fail}) r \]
\[ = (\text{skip } < [r] = [ ] > \text{fail}) r \]
\[ = \text{fail } r \]
\[ = \text{fails } t r \]

Case \( t r = \perp \)

Case \( t r = \perp \)
\[ = \perp \]
\[ = \text{fails } t \]

Law 24*. \( \text{fails } t_1; \text{fails } t_2 = \text{fails } t_2; \text{fails } t_1 \)

\[ \text{fails } t_1; \text{fails } t_2 r \]

Case \( t_1 r = [ ] \) and \( t_2 r = [ ] \)
\[ = \infty/ \bullet (\text{fails } t_2) \ast \bullet \text{fails } t_1 r \]
\[ = \infty/ \bullet (\text{fails } t_2) \ast \bullet \text{skip } r \]
\[ = \infty/ \bullet (\text{fails } t_2) \ast [r] \]
\[ = \infty/[\text{fails } t_2 r] \]
\[ = \text{fails } t_2 r \]
\[ = \text{skip } r \]
\[ = \text{fails } t_1 r \]
\[ = \infty/[\text{fails } t_1 r] \]

\[ \square \]

124
\[
\infty/(\text{fails } t_1) * [r]
\]
\[
\infty/(\text{fails } t_1) * \text{ skip } r
\]
\[
\infty/(\text{fails } t_1) * \text{ fails } t_2 \ r
\]
\[
\text{fails } t_2; \text{ fails } t_1 \ r
\]
\[
\text{Case } t_1 \ r = [\ ] \text{ and } t_2 \ r \neq [\ ]
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_2 \ r
\]
\[
\text{fails } t_2; \text{ fails } t_1 \ r
\]
\[
\text{Case } t_1 \ r \neq [\ ] \text{ and } t_2 \ r = [\ ]
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\infty/\text{ fails } t_2) * \text{ fail } t_2 \ r
\]
\[
\text{fails } t_2; \text{ fails } t_1 \ r
\]
\[
\text{Case } t_1 \ r \neq [\ ] \text{ and } t_2 \ r \neq [\ ]
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\infty/\text{ fails } t_2) * \text{ fails } t_1 \ r
\]
\[
\text{fails } t_2; \text{ fails } t_1 \ r
\]

[Definition of *]
[Definition of skip]
[Definition of fails]
[Definition of ; ]
= fails $t_2$; fails $t_1 r$  

[Definition of $;$]

Case $t_1 r = []$  

$= head'(t_2 r)$  

[Property of $head'$]

$= \Xi/[[], head'(t_2 r)]$  

[Definition of $\Xi/$]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* [r]]$  

[Definition of $head'$]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* skip $r$]$  

[Definition of skip]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* fails $t_1$ r]$  

[Definition of fails]

$= \Xi/ • [t_1, fails $t_1$; $t_1)]^o r$  

[Definition of $^o; ; , !$]

$= !t_2 | (fails $t_1$; !$t_2$) r$  

[Definition of $||$]

Case $t_2 r \neq [] = head'(t_1 r \Xi [])$  

[Assumption]

$= \Xi/[head'(t_1 r), [[]]$  

[Definition of $\Xi/$]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* [[]]]$  

[Definition of $\Xi/,*$]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* fails $t_1$ r]$  

[Definition of $^o; ; , !$]

$= \Xi/ • [t_1, fails $t_1$; !$t_1)]^o r$  

[Definition of $||$]

Case $t_1 r = \bot$ 

= $\bot/[[\bot, head'(t_2 r)]$  

[Definition of $\Xi/$]

$= \Xi/[head'(\bot), head'(t_2 r)]$  

[Assumption]

$= \Xi/[head'(t_1 r), head'(t_2 r)]$  

[Definition of $head'$]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* [r]]$  

[Definition of $\Xi/,*$]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* skip $r$]$  

[Definition of skip]

$= \Xi/[head'(t_1 r), \Xi/ • (head' $t_2$)* fails $t_1$ r]$  

[Definition of fails]

$= \Xi/ • [t_1, fails $t_1$; !$t_2)]^o r$  

[Definition of $^o; ; , !$]

$= !t_1 | (fails $t_1$; !$t_2$) r$  

[Definition of $||$]
Law 26.  \( !(t_1; t_2) = !(t_1; \text{succs} \ t_2); !t_2 \)

This Law is proved by induction as seen below.

\[
!(t_1; t_2) \ r \\
= \text{head'} \cdot \infty/ \bullet t_2 \ast \bullet t_1 \ r \\
\text{[Definition of !, ;, ;]} \\
\]

Base Case \( t_1 \ r = [ \] \)
\[
= \text{head'} \cdot \infty/ \bullet t_2 \ast [ \] \\
= \text{head'} [ \] \\
\text{[Assumption]} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast [ \] \\
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'} [ \] \\
\text{[Definition of head']} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'}(\infty/ \bullet (\text{succs} \ t_2) \ast [ \]) \\
\text{[Definition of head']} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'}(\infty/ \bullet (\text{succs} \ t_2) \ast \bullet t_1 \ r) \\
\text{[Assumption]} \\
\]

\[
= !(t_1; \text{succs} \ t_2); !t_2 \ r \\
\text{[Definition of ; ; !]} \\
\]

Base Case \( t_1 \ r = \bot \)
\[
= \bot \\
\text{[Definition of \( \ast, \infty/ \)]} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'} \bot \\
\text{[Definition of head']} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'}(\infty/ \bullet (\text{succs} \ t_2) \ast \bot) \\
\text{[Definition of head']} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'}(\infty/ \bullet (\text{succs} \ t_2) \ast \bullet t_1 \ r) \\
\text{[Assumption]} \\
\]

\[
= !(t_1; \text{succs} \ t_2); !t_2 \ r \\
\text{[Definition of ; ; !]} \\
\]

Inductive Step on \( t_1 \ r \):

Suppose
\[
\text{head'} \cdot \infty/ \bullet t_2 \ast xs = \infty/ \bullet (\text{head'} \ t_2) \ast \bullet (\text{head'}(\infty/ \bullet (\text{succs} \ t_2) \ast xs) \\
\]

Prove that
\[
\text{head'} \cdot \infty/ \bullet t_2 \ast [x] \infty xs = \infty/ \bullet (\text{head'} \ t_2) \ast \bullet (\text{head'}(\infty/ \bullet (\text{succs} \ t_2) \ast [x] \infty xs) \\
\]

\[
\text{head'} \cdot \infty/ \bullet t_2 \ast [x] \infty xs \\
= \text{head'} \cdot \infty/ \bullet [t_2 \ x] \infty t_2 \ast xs \\
\text{[Definition of \( \ast \)]} \\
\]

\[
= \text{head'}(t_2 \ x \infty \infty/ \bullet t_2 \ast xs) \\
\text{[Definition of \( \infty/ \)]} \\
\]

Case \( t_2 x \neq [ \] \) and \( t_2 x \neq \bot \)
\[
= \text{head'}(t_2 \ x) \\
\text{[Property of head']} \\
\]

\[
= \infty/ \bullet \text{head'}(t_2 \ x) \\
\text{[Definition of \( \infty/ \)]} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast [x] \\
\text{[Definition of \( \ast \)]} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'}([x]) \\
\text{[Definition of head']} \\
\]

\[
= \infty/ \bullet (\text{head'} \ t_2) \ast \text{head'}(\text{skip} \ x) \\
\text{[Definition of skip]} \\
\]

127
= \infty/ \bullet (\text{head}' t_2) \ast \bullet \text{head}'(\infty/ \bullet (\text{succs } t_2) \ast [x] \infty xs) \quad \text{[Definition of } \infty/\text{]} \\
= !(t_1; \text{succs } t_2); !t_2 \ r

Case \( t_2 x = [\ ] \)
\(= \infty/ \bullet \text{head}'(\bot) \quad \text{[Functional Composition]} \)
\(= \infty/ \bullet (\text{head}' t_2) \ast \bot \quad \text{[Definition of } \ast, \text{head']}}
\(= \infty/ \bullet (\text{head}' t_2) \ast \bullet \text{head}'(\bot) \quad \text{[Definition of } \text{head']}}
\(= \infty/ \bullet (\text{head}' t_2) \ast \bullet \text{head}'(\infty/ \bullet (\text{succs } t_2) \ast [x] \infty xs) \quad \text{[Assumption]} \)
\(= !(t_1; \text{succs } t_2); !t_2 \ r \quad \text{[Definition of } ;, !\text{]} \)

Case \( t_2 x = \bot \)
\(= \bot \quad \text{[Definition of } \infty, \text{head']}}
\(= \infty/ \bullet \text{head}'(\bot) \quad \text{[Definition of } \ast, \text{head']}}
\(= \infty/ \bullet (\text{head}' t_2) \ast \bot \quad \text{[Definition of } \ast]\}
\(= \infty/ \bullet (\text{head}' t_2) \ast \bullet \text{head}'(\bot) \quad \text{[Definition of } \text{head']}}
\(= \infty/ \bullet (\text{head}' t_2) \ast \bullet \text{head}'(\infty/ \bullet (\text{succs } t_2) \ast [x] \infty xs) \quad \text{[Assumption]} \)
\(= !(t_1; \text{succs } t_2); !t_2 \ r \quad \text{[Definition of } ;, !\text{]} \)

Law 27. \( \text{succs}(t_1 | t_2) = !(\text{succs } t_1 | \text{succs } t_2) \)

\(\text{succs}(t_1 | t_2) \ r \)

Case \( t_1 r = [\ ] \) and \( t_2 r = [\ ] \)
\(= \text{fail } r \quad \text{[Definition of } \text{succs]} \)
\(= [\ ] \quad \text{[Definition of } \text{fail]} \)
\(= \text{head}'([\ ])] \quad \text{[Definition of } \text{head']}}
\(= \text{head}'(\infty/ [\ ])] \quad \text{[Definition of } \infty/\text{]} \)
\(= \text{head}'(\infty/ [\ ]], [\ ])] \quad \text{[Definition of } \infty/\text{]} \)
\(= \text{head}'(\infty/ [\text{fail } r, \text{fail } r]) \quad \text{[Definition of } \text{fail]} \)
\(= \text{head}'(\infty/ [(\text{fail } \triangleleft t_1 r = [\ ] \triangleright \text{skip}) \ r, \quad (\text{fail } \triangleleft t_2 r = [\ ] \triangleright \text{skip}) \ r)] \quad \text{[Definition of } \triangleleft \triangleright\text{]} \)
\(= \text{head}'(\infty/ \bullet [(\text{fail } \triangleleft t_1 = [\ ] \triangleright \text{skip}), \quad (\text{fail } \triangleleft t_2 = [\ ] \triangleright \text{skip}) \ r] \ r) \quad \text{[Definition of } \circ\text{]} \)
\(= !(\text{succs } t_1 | \text{succs } t_2) \quad \text{[Definition of } \text{succs]} \)

Case \( t_1 r = \bot \) or \( t_2 r = \bot \)
\(= \bot \quad \text{[Definition of } \text{succs]} \)
\(= \text{head}'(\bot) \quad \text{[Definition of } \text{head']}} \)
\[ = !\left(\text{succs } t_1 \mid \text{succs } t_2\right) r \]

[Definition of succs, |]

Other Cases

\[ = \text{skip } r \]

[Definition of succs]

\[ = [r] \]

[Definition of skip]

\[ = \text{head'}([r]) \]

[Definition of head']

Case \( t_1 r \neq [] \) and \( t_2 r = [] \)

\[ = \text{head'}(\infty/[[r],[r]]) \]

[Definition of \( \infty/ \)]

\[ = \text{head'}(\infty/[\text{skip } r, \text{fail } r]) \]

[Definition of fail, skip]

Case \( t_1 r = [] \) and \( t_2 r \neq [] \)

\[ = \text{head'}(\infty/[[r],[r]]) \]

[Definition of \( \infty/ \)]

\[ = \text{head'}(\infty/\text{fail } r, \text{skip } r) \]

[Definition of fail, skip]

Case \( t_1 r \neq [] \) and \( t_2 r \neq [] \)

\[ = \text{head'}(\infty/[[r],[r]]) \]

[Definition of \( \infty/ \)]

\[ = \text{head'}(\infty/[\text{skip } r, \text{skip } r]) \]

[Definition of fail, skip]

In the three cases above, we have that

\[ = \text{head'}(\infty/ [(\text{fail } \triangleleft t_1 r = [] \triangleright \text{skip } r), \]

[Definition of \( \triangleleft \triangleright \)]

\( \text{fail } \triangleleft t_2 r = [] \triangleright \text{skip } r)] \]

[Assumption]

\[ = \text{head'}(\infty/ \cdot [(\text{fail } \triangleleft t_1 r = [] \triangleright \text{skip } r), \]

[Definition of \( \cdot \)]

\( \text{fail } \triangleleft t_2 r = [] \triangleright \text{skip } r)]^* r) \]

[Definition of succs, |]

\[ = !(\text{succs } t_1 \mid \text{succs } t_2) \]

[Definition of head']

\[ \square \]

Law 28. \( \text{fails}(t_1 \mid t_2) = \text{fails } t_1; \text{ fails } t_2 \)

\[ \text{fails}(t_1 \mid t_2) r \]

Case \( t_1 r = [] \) and \( t_2 r = [] \)

\[ = \text{fail } r \]

[Definition of fail]

\[ = [] \]

[Definition of \( \infty/ \)]

\[ = \infty/\cdot \text{fail } * [] \]

[Definition of *]

\[ = \infty/\cdot \text{fail} + \cdot \text{fail } r \]

[Definition of fail]
\[ \star / r (\text{skip} \triangleright t_2 = [ ] \triangleright \text{fail}) \star / r (\text{skip} \triangleright t_1 = [ ] \triangleright \text{fail}) \]

[Definition of \( \triangleright / \) and \( \star / \)]

\[ = \text{fails } t_1; \text{fails } t_2 \]

[Definition of \text{fails}]

Case \( t_1 \) \( r = \bot \) or \( t_2 \) \( r = \bot \)

\[ = \bot \]

[Definition of \( \triangleright \) and \text{fails}]

\[ = \text{fails } t_1; \text{fails } t_2 \]

[Definition of \text{fails}]

Case \( t_1 \) \( r \neq [ ] \) and \( t_2 \) \( r = [ ] \)

\[ = \text{skip } r \]

[Assumption]

\[ = [r] \]

[Definition of \text{skip}]

\[ = \infty / [r] \]

[Definition of \( \infty / \)]

\[ = \infty / \star / r \]

[Definition of \( \star / \)]

\[ = \infty / \star / r \]

[Definition of \text{skip}]

\[ = \infty / (\text{skip} \triangleright t_2 = [ ] \triangleright \text{fail}) \star / (\text{skip} \triangleright t_1 = [ ] \triangleright \text{fail}) \]

[Definition of \( \triangleright / \) and \( \star / \) and \text{fails}]

\[ = \text{fails } t_1; \text{fails } t_2 \]

[Definition of \text{fails}]

Case \( t_1 \) \( r \neq [ ] \) and \( t_2 \) \( r \neq [ ] \)

\[ = \text{skip } r \]

[Assumption]

\[ = [r] \]

[Definition of \text{skip}]

\[ = \infty / [r] \]

[Definition of \( \infty / \)]

\[ = \infty / \star / \star / r \]

[Definition of \( \star / \)]

\[ = \infty / \star / \star / r \]

[Definition of \text{skip}]

\[ = \infty / (\text{skip} \triangleright t_2 = [ ] \triangleright \text{fail}) \star / (\text{skip} \triangleright t_1 = [ ] \triangleright \text{fail}) \]

[Definition of \( \triangleright / \) and \( \star / \) and \text{fails}]

\[ = \text{fails } t_1; \text{fails } t_2 \]

[Definition of \text{fails}]

Case \( t_1 \) \( r \neq [ ] \) and \( t_2 \) \( r \neq [ ] \)

\[ = \text{skip } r \]

[Assumption]

\[ = [r] \]

[Definition of \text{skip}]

\[ = \infty / [r] \]

[Definition of \( \infty / \)]

\[ = \infty / \star / \star / r \]

[Definition of \( \star / \)]

\[ = \infty / \star / \star / r \]

[Definition of \text{skip}]

\[ = \infty / (\text{skip} \triangleright t_2 = [ ] \triangleright \text{fail}) \star / (\text{skip} \triangleright t_1 = [ ] \triangleright \text{fail}) \]

[Definition of \( \triangleright / \) and \( \star / \) and \text{fails}]

\[ = \text{fails } t_1; \text{fails } t_2 \]

[Definition of \text{fails}]

\[ \square \]

Law 29. \( \text{succs } s; \text{succs } (s; t) = \text{succs } (s; t) \)

\[ \text{succs } s; \text{succs } (s; t) \]

\[ = \infty / (\text{succs } (s; t)) \star \text{succs } s \]

[Definition of \( \star \)]

\[ \square \]
Case $s \ r = [ ]$
\[
= \infty / \bullet (\operatorname{succs} (s; t)) \ast [ ]
\]
\[
= [ ]
\]
\[
= \operatorname{fail} \ r
\]
\[
= (\operatorname{fail} \triangleleft [ ] = [ ] \triangleright \operatorname{skip}) \ r
\]
\[
= (\operatorname{fail} \triangleleft t \ast [ ] = [ ] \triangleright \operatorname{skip}) \ r
\]
\[
= (\operatorname{fail} \triangleleft t \ast \bullet s \ r = [ ] \triangleright \operatorname{skip}) \ r
\]
\[
= \operatorname{succs} (s; t) \ r
\]

Case $s \ r \neq [ ]$
\[
= \infty / \bullet (\operatorname{succs} (s; t)) \ast [r]
\]
\[
= \operatorname{succs} (s; t) \ [r]
\]

Case $s \ r = \bot$
\[
= \bot
\]
\[
= \operatorname{succs} (s; t) \ r
\]

\begin{proof}

Law 30. \quad \text{fails} \ s = \operatorname{fails} \ s; \ \operatorname{fails}(s; t)

\quad \text{fails} \ s \ r

\[
= (\operatorname{skip} \triangleleft s \ r = [ ] \triangleright \operatorname{fail}) \ r
\]
\end{proof}
= ∞/ • (skip < ∞/ • t* • s r = [] ▷ fail)* • (skip < ∞/ • s r = [] ▷ fail) r

= fails s; fails(s; t)
Case s r ≠ []
= fail r
= []
= ∞/ []
Let f = skip or f = fail
= ∞/ • f * [] >
= ∞/ • f * • fail r
= ∞/ • (skip < ∞/ • t* • s r = [] ▷ fail)* • (skip < ∞/ • s r = [] ▷ fail) r
= ∞/ (fails (s; t))* • fails s
= fails s; fails(s; t) r

Case s r = ⊥
= ⊥
= fails s; fails(s; t) r

Law 31. !s; fails t = fails(!s; t); !s

!s; fails t r
= !s; fails t r
= ∞/ • (fails t)* • head′(s r) [Definition of !, head′]

Base Case s r = []
= ∞/ • (fails t)* • head′([], ) [Assumption]
= ∞/ • (fails t) * []
= ∞/ []
= ∞/ • head′ []
= ∞/ • head′(s r)
= ∞/ • (head′ s)* [r] [Definition of *]
= ∞/ • (head′ s)* • skip r
= ∞/ • (head′ s)* • (skip < [] ▷ fail) r
= ∞/ • (head′ s)* • (skip < ∞/ • t* • head′(s r) = [] ▷ fail) r [Definition of ∞/ , head′]
= fails(!s; t); !s r [Assumption]
[Definition of ; , !]
Base Case $s \cdot r = \bot$
\[
= \infty/ \cdot (\text{fails } t) \ast \cdot \text{head'}(\bot)
\]
\[
= \infty/ \cdot \bot
\]
\[
= \bot
\]
\[
= \text{fails}(ls; t); !s \cdot r
\]  
[Definition of \text{fails}]

Inductive Step on $s \cdot r$:

Suppose
\[
\infty/ \cdot (\text{fails } t) \ast \cdot \text{head'}([\cdot] \infty xs) = \infty/ \cdot (\text{head'} s) \ast \cdot (\text{skip} < \infty/ \cdot t \ast \cdot \text{head'}([\cdot] \infty xs) = [\cdot] \triangleright \text{fail})
\]

Prove that
\[
\infty/ \cdot (\text{fails } t) \ast \cdot \text{head'}([\cdot] \infty xs)
\]
\[
= \\infty/ \cdot \text{fails } t \ast [\cdot]
\]  
[Definition of \text{head'}]

Case $t \cdot x = [\cdot]$
\[
\infty/\text{fails } t \cdot x
\]
\[
\infty/\text{head'}([\cdot] \infty xs)
\]
\[
\infty/\text{head'}(s \cdot r)
\]  
[Definition of \text{head'}, \infty/]

Case $t \cdot x = \bot$
\[
= \bot
\]  
[Definition of \text{fails}, \infty/]

Case $t \cdot x = ys$
\[
= \infty/\text{fails } t \cdot x
\]
\[
\infty/\text{head'}([\cdot])
\]
\[
\infty/\text{head'}(s \cdot r)
\]  
[Definition of \text{head'}, \infty/]
\[
\begin{align*}
\&= \ast/(\text{head'} s) \ast \mathcal{[} \mathcal{]} \\
\&= \ast \cdot (\text{head'} s) \ast \cdot \text{fail } r \\
\&= \ast \cdot (\text{head'} s) \ast \cdot (\text{skip} \leftarrow \ast/ \hspace{1em} \mathcal{y} s = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} \\
\&= \ast \cdot (\text{head'} s) \ast \cdot (\text{skip} \leftarrow \ast/ \hspace{1em} t \ast \cdot \text{head'}([x] \ast/ \hspace{1em} \mathcal{x} s) = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} \\
\end{align*}
\]

[Definition of *]

[Definition of skip]

[Definition of \(\triangleright \triangleright\)]

[Definition of head', *]

\[\square\]

**Law 32.** \(\text{fails}(\text{fails}(s; t)) = \text{succs } s \mid (\text{fails } s; \text{fails } t)\)

\[
\begin{align*}
\text{fails}(\text{fails}(s; t)) \mathcal{r} \\
\&= (\text{abort} \leftarrow \text{fails}(\text{fails}(s; t)) \mathcal{r} = \bot \triangleright (\text{skip} \leftarrow \text{fails } s; t \mathcal{r} = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} \\
\&= (\text{abort} \leftarrow \text{fails}(\text{fails}(s; t)) \mathcal{r} = \bot \triangleright (\text{skip} \leftarrow (t \ast \cdot (\text{skip} \leftarrow \text{fails } s \mathcal{r} = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} = \mathcal{[} \mathcal{]} \triangleright \text{fail})) \mathcal{r} \\
\end{align*}
\]

[Definition of \(\triangleright \triangleright\)]

[Definition of fails]

[Definition of fails]

[Definition of \text{succs, [\mathcal{]}}]

Case \(s \mathcal{r} = \bot\)
\[
\begin{align*}
\&= \bot \\
\&= \text{succs } s \mid (\text{fails } s; \text{fails } t) \mathcal{r} \\
\end{align*}
\]

[Definition of \(\triangleright \triangleright\)]

[Definition of fails]

[Definition of \text{succs, [\mathcal{]}}]

Case \(s \mathcal{r} = \mathcal{[} \mathcal{]}\)
\[
\begin{align*}
\&= (\text{skip} \leftarrow t \ast \mathcal{r} = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} \\
\&= (\text{skip} \leftarrow t \mathcal{r} = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} \\
\&= \text{fails } t \mathcal{r} \\
\&= \ast/(\mathcal{[} \mathcal{]}, \text{fails } t \mathcal{r}) \\
\&= \ast/\text{succs } s \mathcal{r}, \text{fails } t \mathcal{r} \\
\&= \ast/\text{succs } s \mathcal{r}, \ast/ \hspace{1em} (t \ast \cdot (\text{skip} \leftarrow \text{fails } t) \ast \cdot \text{fails } s \mathcal{r}) \\
\&= \ast/\text{succs } s, \ast/ \hspace{1em} (t \ast \cdot \text{fails } t) \ast \cdot \text{fails } s \mathcal{r} \circ \\
\&= \text{succs } s \mid (\text{fails } t) \ast \cdot \text{fails } s \mathcal{r} \\
\end{align*}
\]

[Definition of \(\triangleright \triangleright\)]

[Definition of \text{succs, [\mathcal{]}}]

[Definition of \ast/]

[Definition of \text{fails}]

[Assumption]

[Definition of \ast/]

[Definition of \circ]

[Definition of \text{succs, [\mathcal{]}}]

Case \(s \mathcal{r} \neq \mathcal{[} \mathcal{]}\)
\[
\begin{align*}
\&= (\text{skip} \leftarrow t \ast \cdot \text{fails } r = \mathcal{[} \mathcal{]} \triangleright \text{fail}) \mathcal{r} \\
\&= \text{skip } r \\
\&= \mathcal{[} \mathcal{]} \\
\&= \ast/\mathcal{[} \mathcal{]}, \mathcal{[} \mathcal{]} \\
\&= \ast/\text{skip } r, \ast/ \hspace{1em} (t \ast \cdot \text{fails } t) \ast \mathcal{[} \mathcal{]} \\
\end{align*}
\]

[Definition of \(\triangleright \triangleright\)]

[Definition of \text{fail, \ast, \triangleright \triangleright}]

[Definition of \text{skip}]

[Definition of \ast/]

[Definition of \text{skip, \ast}]

134
\[ = \infty \] (fails s r \triangleleft s r = \infty \) (fails t) \triangleleft s r
\]

\[ = \infty \] (s r, \infty \) (fails s r = \infty \) (fails t) \\triangleleft \infty
\]

\[ = \infty \] (s r, \infty \) (fails s r) \infty
\]

**Law 33.** fails(s; fails t) = fails s | succs(s; t)

fails(s; fails t) r

\[ = (abort \triangleleft \text{fails}(s; \text{fails } t) r = \bot \triangleleft (\text{skip} \triangleleft s; \text{fails } t r = \infty \triangleleft \text{fail})) r
\]

\[ = (abort \triangleleft \text{fails}(s; \text{fails } t) r = \bot \triangleleft (\text{skip} \triangleleft \infty \triangleleft \text{fails } t) \infty \) \triangleleft \infty
\]

\[ = \text{fails } s | \text{succs}(s; t) r
\]

Case s r = \bot

\[ = \bot
\]

\[ = \text{fails } s | \text{succs}(s; t) r
\]

Case s r = \emptyset

\[ = (\text{skip} \triangleleft \emptyset) r
\]

\[ = \text{skip } r
\]

\[ = \emptyset
\]

\[ = \infty \] (s r, \infty \) (fails s r, \infty \)

\[ = \infty \] (s r, \emptyset \) \\triangleleft \emptyset
\]

\[ = \text{fails } s | \text{succs}(s; t) r
\]

Case s r \neq \emptyset

\[ = (\text{skip} \triangleleft \infty \triangleleft \text{fails } t) \infty \) \triangleleft s r = \emptyset \triangleleft \text{fail}) r
\]

135
We have that
\[
\begin{align*}
&= (\text{fail} \triangleleft \infty/ \bullet t \bullet s \ r = [\ ] \triangleright \text{skip} ) \ r \\
&= \infty/ [\ ], (\text{fail} \triangleleft \infty/ \bullet t \bullet s \ r = [\ ] \triangleright \text{skip} ) \ r \\
&= \infty/ [\text{fails} s \ r, \text{succs} (s; t) \ r] \\
&= [\ ] [\text{Definition of } \infty/ ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of succs} ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of fails, fail} ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of *} ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of *} ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of fails} ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of ends} ] \\
&= \text{fails} s \ r \\
&= [\ ] [\text{Definition of fails} ] \\
\end{align*}
\]

Law 34*. \ fails s; \ succs(s; t) = fail

fails s; succs(s; t) r
\[
\begin{align*}
&= \infty/ \bullet (\text{succs(s; t)})^* \bullet \text{fails} s \ r \\
&= \text{fail} r \\
&= [\ ] [\text{Definition of fail} ] \\
\end{align*}
\]

Law 35. \ fails(t; d) = fails(t; succ d)

fails (t; d) r
\[
\begin{align*}
&= \text{fail} r \\
&= [\ ] [\text{Definition of fail} ] \\
\end{align*}
\]
Case \( s r = [ ] \)
\[
= (\text{skip} < [ ] = [ ] \triangleright \text{fail}) r \\
= (\text{skip} < \infty / \bullet (\text{succs } d) \bullet t \ r = [ ] \triangleright \text{fail}) r \\
= \text{fails } (t; \text{succs } d) \ r
\]
[Definition of \( \infty, * \)]
[Definition of *]
[Assumption]
[Definition of fails]

Case \( s r \neq [ ] \)
\[
= (\text{skip} < \infty / \bullet (\text{succs } d) \bullet t \ r = [ ] \triangleright \text{fail}) r \\
= \text{fails } (t; \text{succs } d) \ r
\]
[Lemma D.6.16]
[Definition of fails]

\[ \square \]

Law 36. \( !s; \text{succs } t = \text{succs } (!s; t); !s \)

\( !s; \text{succs } t \ r \)
\[
= \infty / \bullet (\text{succs } t) \ast \bullet \text{head'}(s \ r) \\
\text{[Definition of ; ]}
\]

Base Case \( s r = [ ] \)
\[
= [ ] \\
= \infty / \bullet (\text{head'} s) \ast [ ] \\
= \infty / \bullet (\text{head'} s) \ast \text{fail } r[ ] \\
= \infty / \bullet (\text{head'} s) \ast (\text{fail} < [ ] = [ ] \triangleright \text{skip}) \ r \\
= \infty / \bullet (\text{head'} s) \ast (\text{fail} < \infty / \bullet t \ast \bullet \text{head'}(s \ r) = [ ] \triangleright \text{skip}) \ r \\
\text{[Definition of head']}
[Definition of *; \infty/]
[Definition of head']
[Definition of *; \infty/]
[Definition of fail]
[Definition of < \triangleright]
[Definition of head']
[Definition of *; \infty/]
[Definition of ; , !, succs]

Base Case \( s r = \bot \)
\[
= \bot \\
= \infty / \bullet (\text{head'} s) \ast \bot \\
= \text{succs}(!s; t); !s \\
\text{[Definition of head']}
[Definition of *; \infty/]
[Definition of head']
[Definition of *; \infty/]
[Definition of ; , !, succs]

Inductive step on \( s r \):
Suppose
\[
\infty / \bullet (\text{succs } t) \ast \bullet \text{head'}(xs) = \infty / \bullet (\text{head'} s) \ast \bullet \text{fail} < \infty / \bullet t \ast \bullet \text{head'}(xs) = [ ] \triangleright \text{skip}) \ r
\]
Prove that

137
\[\\infty/ \bullet (\text{succs } t) \star \bullet \text{head}'([x] \infty xs)\]
\[= \infty/ \bullet (\text{head}' s) \star \bullet (\text{fail} \triangleleft \infty/ \bullet t \star \bullet \text{head}'([x] \infty xs)) = [ ] \triangleright \text{skip}) r\]
\[\infty/ \bullet (\text{succs } t) \star \bullet \text{head}'([x] \infty xs)\]
\[= \infty/ \bullet (\text{succs } t) \star [x] \quad \text{[Definition of } \text{succs}]\]
\[= \infty/ \bullet \text{succs } t \ x \quad \text{[Definition of } \bullet \text{]}\]

Case \(t x = \bot\)
\[= \bot \quad \text{[Definition of succs]}\]
\[= \bot \quad \text{[Definition of skip, } \infty/ \text{]}\]
\[= \bot \quad \text{[Definition of } \star \text{]}\]
\[= \bot \quad \text{[Definition of } \triangleleft \text{]}\]
\[= \bot \quad \text{[Definition of } \triangleright \text{]}\]
\[= \bot \quad \text{[Assumption]}\]

\[= \bot \quad \text{[Definition of } \text{succs}, \star \text{]}\]
\[= \bot \quad \text{[Definition of } \text{skip}, \infty/ \text{]}\]
\[= \bot \quad \text{[Definition of } \triangleleft \text{]}\]
\[= \bot \quad \text{[Definition of } \triangleright \text{]}\]
\[= \bot \quad \text{[Definition of } \text{succs, } \star \text{]}\]

Case \(t x = [y]\)
\[= [y] \quad \text{[Definition of succs]}\]
\[= [y] \quad \text{[Definition of skip, } \infty/ \text{]}\]
\[= [y] \quad \text{[Definition of } \triangleleft \text{]}\]
\[= [y] \quad \text{[Definition of } \triangleright \text{]}\]
\[= [y] \quad \text{[Assumption]}\]
\[= [y] \quad \text{[Definition of } \text{succs, } \star \text{]}\]

Law 37. \(\text{fails(fails } t) = \text{succs } t\)

\text{fails(fails } t) r
Case \( t \ r = \bot \)
\[ = \bot \]  & \hspace{1cm} \text{[Definition of fails]} \\
\[ = \text{succs} \ t \ r \]  & \hspace{1cm} \text{[Definition of succs]} \\

Otherwise
\[ = (\text{skip} < \text{fails} \ t \ r = [\ ] \triangleright \text{fail}) \ r \]  & \hspace{1cm} \text{[Definition of fails]} \\
\[ = (\text{skip} < (\text{skip} \ t \ r = [\ ] \triangleright \text{fail}) \ r = [\ ] \triangleright \text{fail}) \ r \]  & \hspace{1cm} \text{[Definition of fails]} \\

Case \( t \ r = [\ ] \)
\[ = (\text{skip} < [r] \ r = [\ ] \triangleright \text{fail}) \ r \]  & \hspace{1cm} \text{[Definition of \( < \triangleright \), skip]} \\
\[ = \text{fail} \ r \]  & \hspace{1cm} \text{[Definition of \( < \triangleright \)]} \\
\[ = (\text{fail} < t \ r = [\ ] \triangleright \text{skip}) \ r \]  & \hspace{1cm} \text{[Definition of \( < \triangleright \)]} \\
\[ \quad \text{Assumption} \]  & \hspace{1cm} \text{[Definition of succs]} \\
\[ = \text{succs} \ t \ r \]  & \hspace{1cm} \text{[Definition of succs]} \\

Case \( t \ r \neq [\ ] \)
\[ = (\text{skip} < [\ ] \ r = [\ ] \triangleright \text{fail}) \ r \]  & \hspace{1cm} \text{[Definition of \( < \triangleright \), fail]} \\
\[ = \text{skip} \ r \]  & \hspace{1cm} \text{[Definition of \( < \triangleright \)]} \\
\[ = (\text{fail} < t \ r = [\ ] \triangleright \text{skip}) \ r \]  & \hspace{1cm} \text{[Definition of \( < \triangleright \)]} \\
\[ \quad \text{Assumption} \]  & \hspace{1cm} \text{[Definition of succs]} \\
\[ = \text{succs} \ t \ r \]  & \hspace{1cm} \text{[Definition of succs]} \\

\[ \square \]

Law 38\(^*\). \( \text{fails} \ t_1; \text{succs} \ t_2 = \text{succs} \ t_2; \text{fails} \ t_1 \)

\( \text{fails} \ t_1; \text{succs} \ t_2 \ r \)

\[ = \infty / \bullet (\text{succs} \ t_2) * \bullet \text{fails} \ t_1 \ r \]  & \hspace{1cm} \text{[Definition of fails]} \\
\[ \quad \text{Case} \ t_1 \ r = [\ ] \quad \text{and} \quad t_2 \ r = [\ ] \]  & \hspace{1cm} \text{[Definition of skip]} \\
\[ = \infty / \bullet (\text{succs} \ t_2) * [r] \]  & \hspace{1cm} \text{[Definition of *]} \\
\[ = \infty / [\ ] \]  & \hspace{1cm} \text{[Definition of succs]} \\
\[ = \infty / [\ ] \]  & \hspace{1cm} \text{[Definition of \( \infty / \)]} \\
\[ = \infty / \bullet (\text{fails} \ t_1) * [\ ] \]  & \hspace{1cm} \text{[Definition of *]} \\
\[ = \infty / \bullet (\text{fails} \ t_1) * \bullet \text{succs} \ t_2 \ r \]  & \hspace{1cm} \text{[Definition of fails]} \\
\[ \quad \text{Definition of succs} \]  & \hspace{1cm} \text{[Definition of \( \infty / \)]} \\
\[ = \text{succs} \ t_2; \text{fails} \ t_1 \ r \]  & \hspace{1cm} \text{[Definition of \( ; \)]}
Case $t_1 r = []$ and $t_2 r \neq []$
\[= \mathcal{Q}/ \bullet (\text{succs } t_2) \ast [r]\]
\[= \mathcal{Q}/ \bullet [\text{succs } t_2 r]\]
\[= \mathcal{Q}/[(r)]\]
\[= \mathcal{Q}/[\text{fails } t_1 r]\]
\[= \mathcal{Q}/ \bullet (\text{fails } t_1) \ast [r]\]
\[= \mathcal{Q}/ \bullet (\text{fails } t_1) \ast \bullet \text{succs } t_2 r\]
\[= \text{succs } t_2; \text{fails } t_1 r\]

Case $t_1 r \neq []$ and $t_2 r = []$
\[= \mathcal{Q}/ \bullet (\text{succs } t_2) \ast [r]\]
\[= \mathcal{Q}/ \bullet (\text{fails } t_1) \ast [r]\]
\[= \mathcal{Q}/ \bullet (\text{fails } t_1) \ast \bullet \text{succs } t_2 r\]
\[= \text{succs } t_2; \text{fails } t_1 r\]

Case $t_1 r \neq []$ and $t_2 r \neq []$
\[= \mathcal{Q}/ \bullet (\text{succs } t_2) \ast [r]\]
\[= [r]\]
\[= \mathcal{Q}/[\text{fails } t_1 r]\]
\[= \mathcal{Q}/ \bullet (\text{fails } t_1) \ast \bullet \text{succs } t_2 r\]
\[= \text{succs } t_2; \text{fails } t_1 r\]

\[\square\]

Law 39*: $\text{succs } t_1; \text{succs } t_2 = \text{succs } t_2; \text{succs } t_1$

$\text{succs } t_1; \text{succs } t_2 r$
\[= \mathcal{Q}/ \bullet (\text{succs } t_2) \ast \bullet \text{succs } t_1 r\]

Case $t_1 r = []$ and $t_2 r = []$
\[\begin{align*}
\&= \infty \cdot \text{succs } t_2 \ast [] \\
\&= [\] \\
\&= \infty \cdot \text{succs } t_1 \ast [] \\
\&= \infty \cdot \text{succs } t_1 \ast \ast \text{succs } t_2 \ r

\&= \text{succs } t_2 ; \text{succs } t_1 \ r \\
\text{Case } t_1 \ r = [ ] \text{ and } t_2 \ r \neq [ ] \\
\&= \infty \cdot \text{succs } t_2 \ast [ ] \\
\&= [ ] \\
\&= \infty \text{/ (succs } t_1 \ r) \\
\&= \infty \cdot \text{succs } t_1 \ast [r] \\
\&= \infty \cdot \text{succs } t_1 \ast \ast \text{succs } t_2 \ r

\&= \text{succs } t_2 ; \text{succs } t_1 \ r \\
\text{Case } t_1 \ r \neq [ ] \text{ and } t_2 \ r = [ ] \\
\&= \infty \cdot \text{succs } t_2 \ast [r] \\
\&= \infty / [succs t_2 \ r] \\
\&= \infty/ [[[]]] \\
\&= [ ] \\
\&= \infty / [ ] \\
\&= \infty \cdot \text{succs } t_1 \ast [ ] \\
\&= \infty \cdot \text{succs } t_1 \ast \ast \text{succs } t_2 \ r

\&= \text{succs } t_2 ; \text{succs } t_1 \ r \\
\text{Case } t_1 \ r \neq [ ] \text{ and } t_2 \ r \neq [ ] \\
\&= \infty \cdot \text{succs } t_2 \ast [r] \\
\&= \infty / [succs t_2 \ r] \\
\&= \infty/ [[r]] \\
\&= \infty / [succs t_1 \ r] \\
\&= \infty \cdot \text{succs } t_1 \ast [r]
\end{align*}\]
\[
\begin{align*}
\text{succs}(\text{succs } t_1) \star \bullet \text{succs } t_2 \ r & \quad \text{[Definition of succs]} \\
= \text{succs } t_2; \text{succs } t_1 \ r & \quad \text{[Definition of \(\star\)]}
\end{align*}
\]

Law 40. \(\text{succs}(\text{succs } t) = \text{succs } t\)

\[
\text{succs} (\text{succs } t) \ r
\]
Case \(t \ r = \bot\)
= \(\bot\) \quad \text{[Definition of succs]}

Case \(t \ r = [\ ]\)
= (\text{fail} \triangleright \text{succs } t \ r = [\ ] \triangleright \text{skip}) \ r \quad \text{[Definition of succs]}
= \text{fail } \ r \quad \text{[Definition of \(<\triangleright\)>]}
= \text{succs } t \ r \quad \text{[Definition of succs]}

Case \(t_1 \ r \neq [\ ]\)
= (\text{fail} \triangleright \text{succs } t \ r = [\ ] \triangleright \text{skip}) \ r \quad \text{[Definition of succs]}
= \text{skip } \ r \quad \text{[Definition of \(<\triangleright\)>]}
= \text{succs } t \ r \quad \text{[Definition of succs]}

Law 41. \(\text{succs}(\text{fails } t) = \text{fails } t\)

\[
\text{succs} (\text{fails } t) \ r
\]
Case \(t \ r = \bot\)
= \(\bot\) \quad \text{[Definition of succs]}

Otherwise
= (\text{fail} \triangleright \text{fails } t \ r = [\ ] \triangleright \text{skip}) \ r \quad \text{[Definition of succs]}

Case \(t \ r = [\ ]\)
\[= \text{skip} \; r\]
\[= \text{fails} \; t \; r\]

\textbf{Case} \(t_1 \; r \neq \; \bot\)
\[= \text{fail} \; r\]
\[= \text{fails} \; t \; r\]

\textbf{Law 42.} \[\text{succs}(t; \; d) = \text{succs}(t; \; \text{succs} \; d)\]

\[\text{succs}(t; \; d) \; r\]
\textbf{Case} \(t \; r = \; \bot\)
\[= \; \bot\]
\[= \text{succs}(t; \; \text{succs} \; d) \; r\]

\textbf{Otherwise}
\[= (\text{fail} \triangleleft \infty \; \bullet \; d \ast \bullet \; t \; r = \; \bot \triangleright \; \text{skip}) \; r\]

\[\text{Case} \; t \; r = \; \bot\]
\[= (\text{fail} \triangleleft \infty \; \bullet \; d \ast \bullet \; t \; r = \; \bot \triangleright \; \text{skip}) \; r\]

We can say that this is the same as
\[= (\text{fail} \triangleleft \infty \; \bullet \; (\text{succs} \; d) \ast \bullet \; t \; r = \; \bot \triangleright \; \text{skip}) \; r\]

Since the existence of a \(r_n\) in \(t \; r\) such that \(d \; r_n = \; \bot\) implies that, in the same way,
\[= (\text{fail} \triangleleft t; \; \text{succs} \; d \; r = \; \bot \triangleright \; \text{skip}) \; r\]
\[= \text{succs} \; (t; \; \text{succs} \; d) \; r\]

\textbf{Case} \(t_1 \; r \neq \; \bot\)
\[= (\text{fail} \triangleleft \infty \; \bullet \; d \ast \bullet \; t \; r = \; \bot \triangleright \; \text{skip}) \; r\]

We can say that this is the same as
\[= (\text{fail} \triangleleft \infty \; \bullet \; (\text{succs} \; d) \ast \bullet \; t \; r \neq \; \bot \triangleright \; \text{skip}) \; r\]

Since the existence of a \(r_n\) in \(t \; r\) such that \(d \; r_n \neq \; \bot\) implies that, in the same way,
\[= (\text{fail} \triangleleft t; \; \text{succs} \; d \; r = \; \bot \triangleright \; \text{skip}) \; r\]
\[= \text{succs} \; (t; \; \text{succs} \; d) \; r\]

\[\square\]
Law 43. \( \text{succs } t = \text{succs } !t =!\text{succs } t \)

First, let us prove that \( \text{succs } t = \text{succs } !t \). We know that \( t \sim r = [ ] \Leftrightarrow \text{head}'(t \sim r) = [ ] \), this is \( !t = [ ] \). In the same way \( t \sim r \neq [ ] \Leftrightarrow \text{head}'(t \sim r) \neq [ ] \), this os \( !t \neq [ ] \). Also, if \( t \sim r = \bot \) we have that \( !t \sim r = \bot \). This finishes this part of the proof.

Now let us prove that \( \text{succs } !t =!\text{succs } t \)

\[ \begin{align*}
\text{succs } !t \sim r \\
\text{Case } t \sim r = [ ] \\
&= [ ] \\
&= \text{head}' [ ] \\
&= \text{head}'(\text{succs } t \sim r) \\
&= !\text{succs } t \sim r \\
\text{Case } t \sim r = \bot \\
&= \bot \\
&= \text{head}' \bot \\
&= \text{head}'(\text{succs } t \sim r) \\
&= !\text{succs } t \sim r \\
\text{Case } t \sim r \neq [ ] \\
&= < r > \\
&= \text{head}' [r] \\
&= \text{head}'(\text{succs } t \sim r) \\
&= !\text{succs } t \sim r
\end{align*} \]

\[ \square \]

Law 44. \( \text{succs } \text{skip} = \text{skip} \)

\[ \begin{align*}
\text{succs } \text{skip } r \\
&= (\text{fail } \triangleleft \text{skip } r = [ ] \triangleright \text{skip} ) r \\
&= (\text{fail } \triangleleft [r] = [ ] \triangleright \text{skip} ) r \\
&= \text{skip}
\end{align*} \]

\[ \square \]
Law 45.  succs fail = fail

succs fail r
= (fail \triangleleft fail r = [ ] \triangleright skip) r  \quad [\text{Definition of succs}]
= (fail \triangleleft [ ] = [ ] \triangleright skip) r  \quad [\text{Definition of fail}]
= fail  \quad [\text{Definition of } \triangleleft \triangleright]

\square

Law 46.  fails skip = fail

fails skip r
= (skip \triangleleft skip r = [ ] \triangleright fail) r  \quad [\text{Definition of fails}]
= (skip \triangleleft [r] = [ ] \triangleright fail) r  \quad [\text{Definition of skip}]
= fail  \quad [\text{Definition of } \triangleleft \triangleright]

\square

Law 47.  fails fail = skip

fails fail r
= (skip \triangleleft fail r = [ ] \triangleright fail) r  \quad [\text{Definition of fails}]
= (skip \triangleleft [ ] = [ ] \triangleright fail) r  \quad [\text{Definition of fail}]
= skip  \quad [\text{Definition of } \triangleleft \triangleright]

\square

Law 48.  fails t; fails t = fails t

fails t; fails t r
= \infty/ \bullet (fails t) * \bullet fails t r

Case t r = \perp
= \infty/ \bullet (fails t) * \perp  \quad [\text{Definition of fails}]
= \infty/ \perp  \quad [\text{Definition of } *]
= \perp  \quad [\text{Definition of } \infty/]
= fails t r  \quad [\text{Definition of fails}]

Case t r = [ ]
= \infty/ \bullet (fails t) * [r]  \quad [\text{Definition of fails}]
= \infty/([fails t] r)  \quad [\text{Definition of skip}]

\square
\[= [r]\]
\[= \text{fails } t \ r\]

Case \(t \ r \neq [[]]\)
\[= \infty/ \bullet (\text{fails } t) * [[]]\]
\[= [[]]\]
\[= \text{fails } t \ r\]

\[\text{Law 49. succs } t; \text{ succs } t' = \text{ succs } t\]

\[\text{succs } t; \text{ succs } t' r\]
\[= \infty/ \bullet (\text{succs } t) * \bullet \text{ succs } t' r\]

Case \(t \ r = \bot\)
\[= \infty/ \bullet (\text{succs } t) * \bot\]
\[= \bot\]
\[= \text{succs } t \ r\]

Case \(t \ r = [[]]\)
\[= \infty/ \bullet (\text{succs } t) * [[]]\]
\[= [[]]\]
\[= \text{succs } t \ r\]

Case \(t \ r \neq [[]]\)
\[= \infty/ \bullet (\text{fails } t) * [r]\]
\[= \infty/ \bullet [\text{succ } t \ r]\]
\[= [r]\]
\[= \text{succs } t \ r\]

\[\square\]
Law 50*. \( \text{succs } t; \text{ fails } t = \text{ fails } t ; \text{ succs } t = \text{ fail } \)

\[
\text{succs } t; \text{ fails } t r \\
= \infty/ \bullet (\text{fails } t) \& \bullet \text{succs } t r \\
\text{[Definition of ; ]}
\]

Case \( t r = [ ] \)
\[
= \infty/ \bullet (\text{fails } t) * [ ] \\
= [ ] \\
= \text{fail } r \\
= [ ] \\
= \infty/ \bullet [ ] \\
= \infty/ \bullet [\text{succs } t r] \\
= \infty/ \bullet (\text{succs } t) \& [r] \\
= \infty/ \bullet (\text{succs } t) \& \bullet \text{fails } t r \\
= \text{fails } t; \text{ succs } t r \\
\text{[Definition of ; ]}
\]

Case \( t r \neq [ ] \)
\[
= \infty/ \bullet (\text{fails } t) * [r] \\
= \infty/ [\text{fails } t r] \\
= \infty/ [\bullet ] \\
= [ ] \\
= \text{fail } r \\
= [ ] \\
= \infty/ \bullet (\text{succs } t) \& [ ] \\
= \infty/ \bullet (\text{succs } t) \& \bullet \text{fails } t r \\
= \text{fails } t; \text{ succs } t r \\
\text{[Definition of ; ]}
\]

\( \square \)

Law 51*. \( \text{fails } t \mid \text{succs } t = \text{succs } t \mid \text{fails } t = \text{skip } \)

\[
\text{fails } t \mid \text{succs } t r \\
= \infty/ \bullet <\text{fails } t r, \text{succs } t r > \\
\text{[Definition of } \mid , \& \text{]}
\]
Case $t \cdot r = \emptyset$
\[= \infty/[[r], [\emptyset]]\]
\[= \infty/[[\emptyset], [r]]\]
\[= [r]\]
\[= \text{skip } r\]
\[= [r]\]
\[= \infty/[[r], [\emptyset]]\]
\[= \infty/[[\emptyset], [r]]\]
\[= \text{succs } t \cdot r, \text{fails } t \cdot r\]

\[= \text{succs } t | \text{fails } t \cdot r\]

Case $t \cdot r \neq \emptyset$
\[= \infty/[[\emptyset], [r]]\]
\[= \infty/[[r], [\emptyset]]\]
\[= [r]\]
\[= \text{skip } r\]
\[= [r]\]
\[= \infty/[[r], [\emptyset]]\]
\[= \infty/[[\emptyset], [r]]\]
\[= \text{succs } t \cdot r, \text{fails } t \cdot r\]

\[= \text{succs } t | \text{fails } t \cdot r\]

Law 52. \(\text{succs}(t | u) = \text{succs } t | (\text{fails } t; \text{succs } u)\)

\(\text{succs}(t | u) \cdot r\)

Case $t \cdot r = \bot$ or $u \cdot r = \bot$
\[= \bot\]

148
\[
= \text{succs } t \mid (\text{fails } t; \text{succs } u) \ r
\]

Otherwise
\[
= (\text{fail } \triangleleft \infty/ \bullet < t, u >^\circ \ r =<> \triangleright \text{skip}) \ r
\]
\[
= (\text{fail } \triangleleft \infty/ \bullet t \ r \infty \ u \ r =<> \triangleright \text{skip}) \ r
\]

Case \( t \ r = [ ] \), \( u \ r = [ ] \)
\[
= <>
\]
\[
= \infty/[[ ]], [ ]
\]
\[
= \infty/\text{succs } t \ r, \infty/[[ ]]\]
\[
= \infty/\text{succs } t \ r, \infty/ < \text{succs } u \ r] \]
\[
= \infty/\text{succs } t \ r, \infty/ \bullet (\text{succs } u) \ast [r] \]
\[
= \infty/\text{succs } t \ r, \infty/ \bullet (\text{succs } u) \ast \bullet \text{fails } t \ r
\]
\[
= \text{succs } t \mid (\text{fails } t; \text{succs } u) \ r
\]

Case \( t \ r \neq [ ] \)
\[
= [r]
\]
\[
= \infty/[[ ], [ ]]
\]
\[
= \infty/\text{succs } t \ r, \infty/[[r]]\]
\[
= \infty/\text{succs } t \ r, \infty/ [\text{succs } u \ r] \]
\[
= \infty/\text{succs } t \ r, \infty/ \bullet (\text{succs } u) \ast [r] \]
\[
= \infty/\text{succs } t \ r, \infty/ \bullet (\text{succs } u) \ast \bullet \text{fails } t \ r
\]
\[
= \text{succs } t \mid (\text{fails } t; \text{succs } u) \ r
\]

Case \( t \ r \neq [ ] \)
\[
= [r]
\]
\[
= \infty/[[r], [ ]]
\]
\[
= \infty/\text{succs } t \ r, \infty/ \bullet (\text{succs } u) \ast [ ]\]
= \infty/\text{succs } t \ r, \infty/ \cdot \text{succs } u \ast \cdot \text{fails } t \ r \quad \text{[Definition of fails]}
= \text{succs } t \mid (\text{fails } t; \text{succs } u) \ r \quad \text{[Definition of } \circ, ][

\textbf{Law 53. } \text{succs}(\text{fails } s; \ t)) = \text{fails } s; \text{succs } t
\text{succs}(\text{fails } s; \ t)) \ r
\text{Case } s \ r = \bot \text{ or } t \ r = \bot
\quad = \bot 
\quad = \text{fails } s; \text{succs } t \ r
\text{Case } s \ r = [], t \ r = []
\quad = (\text{fail } \lhd \infty/ \cdot t \ast [r] = [ ] \rhd \text{skip}) \ r
\quad = (\text{fail } \lhd t \ r = [ ] \rhd \text{skip}) \ r 
\quad = [ ]
\quad = \infty/[[ ]] 
\quad = \infty/\text{succs } t \ r 
\quad = \infty/(\text{succs } t) \ast [r] 
\quad = \infty/(\text{succs } t) \ast \cdot \text{fails } s \ r 
\quad = \text{fails } s; \text{succs } t \ r 
\text{Case } s \ r = [], t \ r \neq []
\quad = (\text{fail } \lhd \infty/ \cdot t \ast [r] = [ ] \rhd \text{skip}) \ r 
\quad = (\text{fail } \lhd t \ r = [ ] \rhd \text{skip}) \ r 
\quad = [r] 
\quad = \infty/[[r]] 
\quad = \infty/\text{succs } t \ r 
\quad = \infty/(\text{succs } t) \ast [r] 

150
\( = \infty/(\text{succs } t) \ast \bullet \text{fails } s \ r \)  
\( = \text{fails } s; \text{succs } t \ r \)  
\[ \text{[Definition of fails]} \]
\[ \text{[Definition of skip]} \]
\[ \text{[Definition of ; ]} \]

Case \( s \ r \neq [] \)
\( = (\text{fail} \triangleleft \infty/ \bullet t \ast [ ] = [ ] \triangleright \text{skip}) \ r \)
\( = [] \)
\[ \text{[Definition of fails]} \]
\[ \text{[Definition of fail]} \]
\[ \text{[Definition of } \ast, \infty/\text{]} \]
\[ \text{[Definition of fail]} \]
\[ \text{[Definition of } \ast \text{]} \]
\[ \text{[Definition of fails]} \]
\[ \text{[Definition of fail]} \]
\[ \text{[Definition of ; ]} \]

\[ = \text{fails } s; \text{succs } t \ r \]

\[ \text{Law 54. succs(succs } s; \ t) = \text{succs } s; \text{succs } t \]

\[ \text{succs(succs } s; \ t)) \ r \]

Case \( s \ r = \bot \) or \( t \ r = \bot \)
\( = \bot \)
\( = \text{succs } s; \text{succs } t \ r \)

[Definition of succs,,; ]
[Definition of succs,,; ]

Otherwise
\( = (\text{fail} \triangleleft \text{succs } s; \ t \ r = [ ] \triangleright \text{skip}) \ r \)
\( = (\text{fail} \triangleleft \infty/ \bullet t \ast \bullet \text{succs } s \ r = [ ] \triangleright \text{skip}) \ r \)

[Definition of succs]
[Definition of ; ]

Case \( s \ r = [ ] \)
\( = (\text{fail} \triangleleft \infty/ \bullet t \ast [ ] = [ ] \triangleright \text{skip}) \ r \)
\( = [] \)

[Definition of succs]
[Definition of fail]
[Definition of } \ast, \infty/\text{]
[Definition of fail]
[Definition of } \ast \text{]
[Definition of succs]
[Definition of fail]
[Definition of ; ]

\[ = \text{succs } s; \text{succs } t \ r \]

Case \( s \ r \neq [] \), \( t \ r = [ ] \)
\( = (\text{fail} \triangleleft \infty/ \bullet t \ast [ r] = [ ] \triangleright \text{skip}) \ r \)

[Definition of succs]
[Definition of skip]
\[(\text{fail} \triangleright t \ r = [ ] \triangleright \text{skip}) \ r = [ ]\]

\[= \bigcirc/([ ]]
\[= \bigcirc/([\text{succs} \ t \ r])\]

\[= \bigcirc/(\text{succs} \ t) \ast [r]
\[= \bigcirc/(\text{succs} \ t) \ast \bullet \text{succs} \ s \ r\]

\[= \text{succs} \ s; \text{succs} \ t \ r \]

Case \(s \ r \neq [ ], \ t \ r \neq [ ]\)
\[= (\text{fail} \triangleright \bigcirc/ \bullet \ t \ast [r] = [ ] \triangleright \text{skip}) \ r\]
\[= (\text{fail} \triangleright t \ r = [ ] \triangleright \text{skip}) \ r = [r]\]
\[= \bigcirc/([r])
\[= \bigcirc/([\text{succs} \ t \ r])\]

\[= \bigcirc/(\text{succs} \ t) \ast [r]
\[= \bigcirc/(\text{succs} \ t) \ast \bullet \text{succs} \ s \ r\]

\[= \text{succs} \ s; \text{succs} \ t \ r\]

\[\square\]

**Law 55.** \(
\text{succs}(s; \text{fails} \ t) = \text{succs} \ s; \text{fails}(s; \ t)
\)

\[\text{succs}(s; \text{fails} \ t) \ r\]

Case \(s \ r = \bot \text{ or } t \ r = \bot\)
\[= \bot\]
\[= \text{succs} \ s; \text{fails}(s; \ t) \ r\]

Otherwise
\[= (\text{fail} \triangleright \bigcirc/ \bullet (\text{fails} \ t) \ast \bullet s \ r =<> \triangleright \text{skip}) \ r\]

Case \(s \ r =<>\)
\[= (\text{fail} \triangleright <> =<> <> \triangleright \text{skip}) \ r\]
= \langle \rangle

= \bigotimes \cdot (\text{fails}(s; t)) \ast \langle \rangle

= \bigotimes \cdot (\text{fails}(s; t)) \ast \cdot \text{succs} \ s \ r

= \text{succs} \ s; \ \text{fails}(s; t) \ r

\text{Case} \ s \ r \neq \langle \rangle

= (\text{skip} < \bigotimes \cdot (\text{succs} \ t) \ast \cdot \ s \ r = \langle \rangle \triangleright \text{fail}) \ r

= (\text{skip} < \bigotimes \cdot \ t \ast \cdot \ s \ r = \langle \rangle \triangleright \text{fail}) \ r

= \text{fails}(s; t) \ r

= \bigotimes \cdot (\text{fails}(s; t)) \ast < \ r >

= \bigotimes \cdot (\text{fails}(s; t)) \ast \cdot \text{succs} \ s \ r

= \text{succs} \ s; \ \text{fails}(s; t) \ r

\text{Law 56.} \ \text{succs}(s; \text{succs} \ t) = \text{succs} \ s; \ \text{succs} \ (s; t)

\text{succs}(s; \text{succs} \ t)

= \text{succs} \ (s; t)

= \text{succs} \ s; \ \text{succs} \ (s; t)

\text{Law 57.} \ \text{fails} \ (\text{succs} \ s; t) = \text{fails} \ s \mid (\text{succs} \ s; \ \text{fails} \ t)

\text{fails} \ (\text{succs} \ s; t) \ r

\text{Case} \ s \ r = \bot \ \text{or} \ t \ r = \bot

= \bot

= \text{fails} \ s \mid (\text{succs} \ s; \ \text{fails} \ t) \ r

\text{Otherwise}

= \text{fails} \ (\text{succs} \ (\text{succs} \ s); \ \text{succs} \ t) \ r

= \text{fails} \ (\text{succs} \ s; \ \text{succs} \ t) \ r

= (\text{skip} < \bigotimes \cdot (\text{succs} \ t) \ast \cdot \text{succs} \ s \ r = [ \ ] \triangleright \text{fail}) \ r

\text{Law 42}

\text{Definition of ; ,succs}

\text{Definition of ; ,fails}

\text{Definition of succs}

\text{Law 35}

\text{Law 40}

\text{Law 29}

\text{Law 34}
Case \( s \cdot r = [] \)
\[ (\text{skip} < \infty/ \bullet (\text{succs } t) \star [ ] = [ ] \triangleright \text{fail}) \quad r \]
\[ = [r] \]
\[ = \infty/[[r], [ ]] \]
\[ = \infty/\text{fails } s \cdot r, \infty/ \bullet (\text{fails } t) \star [ ] \]
\[ = \infty/\text{fails } s \cdot r, \infty/ \bullet (\text{fails } t) \star \bullet \text{succs } s \cdot r \]
\[ = \text{fails } s \mid (\text{succs } s; \text{fails } t) \quad r \]

Case \( s \cdot r \neq [ ] \), \( t \cdot r = [ ] \)
\[ (\text{skip} < \infty/ \bullet (\text{succs } t) \star [r] = [ ] \triangleright \text{fail}) \quad r \]
\[ = (\text{skip} < \text{succs } t \cdot r = [ ] \triangleright \text{fail}) \quad r \]
\[ = [r] \]
\[ = \infty/[[], [r]] \]
\[ = \infty/\text{fails } s \cdot r, \infty/\text{fails } t \cdot r \]
\[ = \infty/\text{fails } s \cdot r, \infty/ \bullet (\text{fails } t) \star [r] \]
\[ = \infty/\text{fails } s \cdot r, \infty/ \bullet (\text{fails } t) \star \bullet \text{succs } s \cdot r \]
\[ = \text{fails } s \mid (\text{succs } s; \text{fails } t) \quad r \]

Case \( s \cdot r \neq [ ] \), \( t \cdot r \neq [ ] \)
\[ (\text{skip} < \infty/ \bullet (\text{succs } t) \star [r] = [ ] \triangleright \text{fail}) \quad r \]
\[ = (\text{skip} < \text{succs } t \cdot r = [ ] \triangleright \text{fail}) \quad r \]
\[ = [ ] \]
\[ = \infty/[[], [ ]] \]
\[ = \infty/\text{fails } s \cdot r, \infty/\text{fails } t \cdot r \]
\[ = \infty/\text{fails } s \cdot r, \infty/ \bullet (\text{fails } t) \star [r] \]
\[ = \mathcal{K}/[\text{fails } s \ r, \mathcal{K}/ \bullet (\text{fails } t) \ast \bullet \text{succs } s \ r] \quad \text{[Definition of succs]} \]
\[ = \text{fails } s \ | (\text{succs } s; \text{fails } t) \ r \quad \text{[Definition of } ; \text{, } \circ \text{, } \bullet \text{]} \]

**Law 58.** \( \text{fails}(s; \text{succs } t) = \text{fails } s \ | (\text{succs } s; \text{fails } (s; t)) \)

This law is proved by induction, as follows:

\( \text{fails}(s; \text{succs } t) \ r \)

**Base Case:** \( s \ r = \bot \)
\[ = \bot \quad \text{[Definition of fails]} \]
\[ = \mathcal{K}/\bot \quad \text{[Definition of } \mathcal{K}/\text{]} \]
\[ = \mathcal{K}/[\bot, \mathcal{K}/ \bullet (\text{fails } (s; t)) \ast [ ]] \quad \text{[Definition of } \mathcal{K}/\text{]} \]
\[ = \mathcal{K}/[\text{fails } s \ r, \mathcal{K}/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r] \quad \text{[Definition of succs]} \]
\[ = \text{fails } s \ | (\text{succs } s; \text{fails } (s; t)) \ r \quad \text{[Definition of } ; \text{, } \circ \text{, } \bullet \text{]} \]

**Base Case:** \( s \ r = [ ] \)
\[ = (\text{skip} < \mathcal{K}/ \bullet (\text{succs } t) \ast \bullet s \ r = [ ] \triangleright \text{fail}) \ r \quad \text{[Definition of fails, ; ]} \]
\[ = (\text{skip} < [ ] = [ ] \triangleright \text{fail}) \ r \quad \text{[Definition of } < \triangleright \text{]} \]
\[ = [r] \quad \text{[Definition of skip]} \]
\[ = \mathcal{K}/[[r],[ ]] \quad \text{[Definition of } \mathcal{K}/\text{]} \]
\[ = \mathcal{K}/[\text{fails } s \ r, \mathcal{K}/ \bullet (\text{fails } (s; t)) \ast [ ]] \quad \text{[Definition of fails]} \]
\[ = \mathcal{K}/[\text{fails } s \ r, \mathcal{K}/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r] \quad \text{[Definition of succs]} \]
\[ = \text{fails } s \ | (\text{succs } s; \text{fails } (s; t)) \ r \quad \text{[Definition of ; , } \circ \text{, } \bullet \text{]} \]

**Inductive step on** \( s \ r \):

We suppose
\[ (\text{skip} < \mathcal{K}/ \bullet (\text{succs } t) \ast xs = \]
\[ [ ] \triangleright \text{fail}) \ r = \mathcal{K}/[\text{fails } s \ r, \mathcal{K}/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r] \]

We must prove that
\[ (\text{skip} < \mathcal{K}/ \bullet (\text{succs } t) \ast [xs] \mathcal{K}/ \ast xs = \]
\[ [ ] \triangleright \text{fail}) \ r = \mathcal{K}/[\text{fails } s \ r, \mathcal{K}/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r] \]

\[ (\text{skip} < \mathcal{K}/ \bullet (\text{succs } t) \ast [xs] \mathcal{K}/ = [ ] \triangleright \text{fail}) \ r \]
\[ = (\text{skip} < \mathcal{K}/ \bullet ((\text{succs } t \ x) \mathcal{K}/ \ast xs = [ ] \triangleright \text{fail}) \ r \quad \text{[Definition of } \ast \text{]} \]
Case $t \ x = []$

$= (\text{skip} < \infty/ \bullet ([] \infty (\text{succs } t) \ast xs = []) > \text{fail}) \ r$ [Assumption]

$= (\text{skip} < \infty/ \bullet (\text{succs } t) \ast xs = []) > \text{fail}) \ r$ [Definition of $\infty$]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r]$ [Assumption]

Case $t \ x = \bot$

$= \bot$ [Definition of $\text{succs}$]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r]$ [Definition of $\text{fails}$, $\infty/\bullet$]

Case $t \ x = xs$

$= (\text{skip} < \infty/ \bullet ([x] \infty (\text{succs } t) \ast xs = []) > \text{fail}) \ r$ [Assumption]

$= (\text{skip} < [x] \infty/ \bullet (\text{succs } t) \ast xs = []) > \text{fail}) \ r$ [Definition of $\infty/\bullet$]

$= [r]$ [Definition of $\bullet \bot$]

$= \infty/ [ [\text{fails } s \ r], [r] ]$ [Definition of $\infty/\bullet$]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet \text{fails } (\infty/ \bullet t \ast \bullet s) \ r]$ [Assumption]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet (\text{fails } (\infty/ \bullet t \ast \bullet s)) \ast [r]]$ [Definition of $\bullet \bot$]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet (\text{fails } (s; t)) \ast [r]]$ [Definition of $\bullet \bot$]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet (\text{fails } (s; t)) \ast \text{skip } r]$ [Definition of $\text{skip}$]

$= \infty/ [\text{fails } s \ r, \infty/ \bullet (\text{fails } (s; t)) \ast \bullet \text{succs } s \ r]$ [Definition of $\text{succs}$]

\[\square\]

### D.4 Laws involving Structural Combinators

**Law 59.** $(t_1 \triangleright t_2); (t_3 \triangleright t_4) = (t_1; t_3) \triangleright (t_2; t_4)$

\[
(\llbracket (t_1 \triangleright t_2); (t_3 \triangleright t_4) \rrbracket)_{\Gamma_L \Gamma_T} (p_1; p_2, \text{proofs})
\]

$= \infty/ \bullet (\llbracket (t_3 \triangleright t_4) \rrbracket)_{\Gamma_L \Gamma_T} \ast \bullet (\llbracket (t_1 \triangleright t_2) \rrbracket)_{\Gamma_L \Gamma_T} (p_1; p_2, \text{proofs})$ [Definition of $\llbracket \bullet \rrbracket$]

$= \infty/ \bullet (\llbracket (t_3 \triangleright t_4) \rrbracket)_{\Gamma_L \Gamma_T} \ast \bullet \Omega\ast (\llbracket (t_1 \triangleright t_2) \rrbracket)_{\Gamma_L \Gamma_T} (p_1; p_2, \text{proofs}) >)$ [Definition of $\llbracket \Pi \rrbracket$]

$= \infty/ \bullet (\llbracket (t_3 \triangleright t_4) \rrbracket)_{\Gamma_L \Gamma_T} \ast \bullet \Omega\ast ((p'_1, p'_{1}) : \text{as} \mid (p'_1, p'_{1}) \leftarrow \llbracket t_4 \rrbracket_{\Gamma_L \Gamma_T} (p_1, \text{proofs}),$

$\text{as} \leftarrow \Pi < \llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} (p_2, \text{proofs}) >]$ [Definition of $\Pi$]

$= \infty/ \bullet (\llbracket (t_3 \triangleright t_4) \rrbracket)_{\Gamma_L \Gamma_T} \ast \bullet \Omega\ast ((p'_1, p'_{1}) : \text{as} \mid (p'_1, p'_{1}) \leftarrow \llbracket t_4 \rrbracket_{\Gamma_L \Gamma_T} (p_1, \text{proofs}),$

$\text{as} \leftarrow e2l \text{as} < \llbracket t_2 \rrbracket_{\Gamma_L \Gamma_T} (p_2, \text{proofs}) >]$ [Definition of $\Pi$]
= ∞/ • ((t_3 \overline{t}_4)[Γ_L Γ_T]) [Definition of Ω] * • (p_1, pr_1), (p_2, pr_2) > | (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)]

= ∞/ • (||t_3\overline{t}_4)[Γ_L Γ_T] * [Definition of *]
| (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)]

= ∞/ (\llbracket t_3 \overline{t}_4 \rrbracket [Γ_L Γ_T] *
) [(p_1', p_2', pr_1' \cup pr_2') | (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)]

= ∞/ [Ω] * (Π < [t_3][Γ_L Γ_T (p_1', pr_1' \cup pr_2'), [t_4][Γ_L Γ_T (p_2', pr_1' \cup pr_2') > ]
| (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)]

= ∞/ [Ω] * (p_1, pr_1') : cs
| (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)], (p_3', pr_3') ← [t_3][Γ_L Γ_T (p_1', pr_1' \cup pr_2'), cs ← e2l * [t_4][Γ_L Γ_T (p_2', pr_1' \cup pr_2')]]

= ∞/ [Ω] * (p_1, pr_1') : cs
| (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)], (p_3', pr_3') ← [t_3][Γ_L Γ_T (p_1', pr_1' \cup pr_2'), cs ← e2l * [t_4][Γ_L Γ_T (p_2', pr_1' \cup pr_2')]]

= ∞/ [Ω] * (< (p_5', pr_3'), (p_4', pr_4') > 
| (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)], (p_3', pr_3') ← [t_3][Γ_L Γ_T (p_1', pr_1' \cup pr_2'), (p_4', pr_4') ← [t_4][Γ_L Γ_T (p_2', pr_1' \cup pr_2')]

= ∞/ [Ω] < (p_5', pr_3'), (p_4', pr_4') > 
| (p_1', pr_1') ← [t_1][Γ_L Γ_T (p_1, proofs)], (p_2', pr_2') ← [t_2][Γ_L Γ_T (p_2, proofs)], (p_3', pr_3') ← [t_3][Γ_L Γ_T (p_1', pr_1' \cup pr_2'), (p_4', pr_4') ← [t_4][Γ_L Γ_T (p_2', pr_1' \cup pr_2')]

[Definition of *]
\[ \Omega = \infty / (p'_3, p'_4, pr'_3 \cup pr'_4) \]
\[ | (p'_1, pr'_1) \leftarrow [t_1] \Gamma_L \Gamma_T (p_1, proofs), \]
\[ (p'_2, pr'_2) \leftarrow [t_2] \Gamma_L \Gamma_T (p_2, proofs), \]
\[ (p'_3, pr'_3) \leftarrow [t_3] \Gamma_L \Gamma_T (p'_1, pr'_1 \cup pr'_2), \]
\[ (p'_4, pr'_4) \leftarrow [t_4] \Gamma_L \Gamma_T (p'_2, pr'_1 \cup pr'_2) \]

[Definition of \( \Omega \)]

\[ [ (p'_3, p'_4, pr'_3 \cup pr'_4) \]
\[ | (p'_1, pr'_1) \leftarrow [t_1] \Gamma_L \Gamma_T (p_1, proofs), \]
\[ (p'_2, pr'_2) \leftarrow [t_2] \Gamma_L \Gamma_T (p_2, proofs), \]
\[ (p'_3, pr'_3) \leftarrow (add pr'_2) \ast [t_3] \Gamma_L \Gamma_T (p'_1, pr'_1), \]
\[ (p'_4, pr'_4) \leftarrow (add pr'_1) \ast [t_4] \Gamma_L \Gamma_T (p'_2, pr'_2) \]

[Definition of \( \infty / \)]

\[ [ (p'_3, p'_4, pr'_3 \cup pr'_4) \]
\[ | (p'_1, pr'_1) \leftarrow [t_1] \Gamma_L \Gamma_T (p_1, proofs), \]
\[ (p'_2, pr'_2) \leftarrow [t_2] \Gamma_L \Gamma_T (p_2, proofs), \]
\[ (p'_3, pr'_3) \leftarrow [t_3] \Gamma_L \Gamma_T (p'_1, pr'_1), \]
\[ (p'_4, pr'_4) \leftarrow [t_4] \Gamma_L \Gamma_T (p'_2, pr'_2), \]
\[ pr'_3 = pr'_3 \cup pr'_2, pr'_4 = pr'_4 \cup pr'_1 \]

[Definition of \( \text{add} \)]

\[ [ (p'_3, p'_4, pr''_3 \cup pr'_4 \cup pr''_2 \cup pr'_1) \]
\[ | (p'_1, pr'_1) \leftarrow [t_1] \Gamma_L \Gamma_T (p_1, proofs), \]
\[ (p'_2, pr'_2) \leftarrow [t_2] \Gamma_L \Gamma_T (p_2, proofs), \]
\[ (p'_3, pr''_3) \leftarrow [t_3] \Gamma_L \Gamma_T (p'_1, pr'_1), \]
\[ (p'_4, pr''_4) \leftarrow [t_4] \Gamma_L \Gamma_T (p'_2, pr'_2), \]
\[ pr''_3 = pr''_3 \cup pr'_2, pr''_4 = pr''_4 \cup pr'_1 \]

[Property of \( \text{add} \)]

\[ [ (p'_3, p'_4, pr''_3 \cup pr''_4) \]
\[ | (p'_1, pr'_1) \leftarrow [t_1] \Gamma_L \Gamma_T (p_1, proofs), \]
\[ (p'_2, pr'_2) \leftarrow [t_2] \Gamma_L \Gamma_T (p_2, proofs), \]
\[ (p'_3, pr''_3) \leftarrow [t_3] \Gamma_L \Gamma_T (p'_1, pr'_1), \]
\[ (p'_4, pr''_4) \leftarrow [t_4] \Gamma_L \Gamma_T (p'_2, pr'_2) \]

[Lemma D.6.9]

\[ [ \Omega, < (p''_3, p'_3), (p'_4, pr''_4) > \]
\[ | (p'_1, pr'_1) \leftarrow [t_1] \Gamma_L \Gamma_T (p_1, proofs), \]
\[ (p'_2, pr'_2) \leftarrow [t_2] \Gamma_L \Gamma_T (p_2, proofs), \]
\[ (p'_3, pr''_3) \leftarrow [t_3] \Gamma_L \Gamma_T (p'_1, pr'_1), \]
\[ (p'_4, pr''_4) \leftarrow [t_4] \Gamma_L \Gamma_T (p'_2, pr'_2) \]

[Definition of \( \Omega \)]
\[
\begin{align*}
= \Omega, \ \star \cdot \left< (p_3', pr_3''), (p_4', pr_4'') \right>
& \mid (p_1', pr_1') \leftarrow \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
(p_2', pr_2') \leftarrow \llbracket t_2 \rrbracket \Gamma_L \Gamma_T (p_2, proofs), \\
(p_3', pr_3'') \leftarrow \llbracket t_3 \rrbracket \Gamma_L \Gamma_T (p_1', pr_1'), \\
(p_4', pr_4'') \leftarrow \llbracket t_4 \rrbracket \Gamma_L \Gamma_T (p_2', pr_2')
\end{align*}
\]
[Definition of \(\star\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi < \llbracket p_3', pr_3' \rrbracket \right)
& \mid (p_1', pr_1') \leftarrow \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
(p_3', pr_3') \leftarrow \llbracket t_3 \rrbracket \Gamma_L \Gamma_T (p_1', pr_1'), \\
\infty/\left( p_4', pr_4' \right)
& \mid (p_2', pr_2') \leftarrow \llbracket t_2 \rrbracket \Gamma_L \Gamma_T (p_2, proofs), \\
(p_4', pr_4') \leftarrow \llbracket t_4 \rrbracket \Gamma_L \Gamma_T (p_2', pr_2')
\end{align*}
\]
[Definition of \(\infty/\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi \llbracket \infty/ \bullet \llbracket t_3 \rrbracket \Gamma_L \Gamma_T * \left< (p_1', pr_1') \mid (p_3', pr_3') \right> \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\infty/ \bullet \llbracket t_4 \rrbracket \Gamma_L \Gamma_T * \left< (p_2', pr_2') \mid (p_4', pr_4') \right> \llbracket t_2 \rrbracket \Gamma_L \Gamma_T (p_2, proofs)
\end{align*}
\]
[Definition of \(\llbracket \) and \(\star\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi \llbracket \infty/ \bullet \llbracket t_3 \rrbracket \Gamma_L \Gamma_T * \bullet \llbracket t_3 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\infty/ \bullet \llbracket t_4 \rrbracket \Gamma_L \Gamma_T * \bullet \llbracket t_4 \rrbracket \Gamma_L \Gamma_T (p_2, proofs)
\end{align*}
\]
[Definition of \(\infty/\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left< \llbracket t_1; t_2 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\llbracket t_2; t_3 \rrbracket \Gamma_L \Gamma_T (p_2, proofs) \right>
\end{align*}
\]
[Definition of \(\bullet\)]

\[
\begin{align*}
= \llbracket t_1; t_3 \rrbracket \llbracket t_2; t_4 \rrbracket \Gamma_L \Gamma_T (p_1; p_2, proofs)
\end{align*}
\]
[Definition of \(\llbracket \) and \(\rrbracket\)]

\[\square\]

**Law 60.** \( t_1 \llbracket t_2 \mid t_3 \rrbracket = (t_1 \llbracket t_2 \rrbracket) \mid (t_1 \llbracket t_3 \rrbracket) \)

\[
\begin{align*}
\llbracket t_1 \rrbracket \llbracket t_2 \mid t_3 \rrbracket \Gamma_L \Gamma_T (p_1; p_2, proofs)
= \Omega, \ \star \cdot \left( \Pi < \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\llbracket t_2 \mid t_3 \rrbracket \Gamma_L \Gamma_T (p_2, proofs) \right)
\end{align*}
\]
[Definition of \(\llbracket \) and \(\star\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi < \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\llbracket t_2 \mid t_3 \rrbracket \Gamma_L \Gamma_T (p_2, proofs) \right)^{\circ} (p_2, proofs) \]
\]
[Definition of \(\circ\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi < \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\llbracket t_2 \mid t_3 \rrbracket \Gamma_L \Gamma_T (p_2, proofs) \right)^{\circ} (p_2, proofs) \]
\]
[Definition of \(\circ\)]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi < \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\llbracket t_2 \mid t_3 \rrbracket \Gamma_L \Gamma_T (p_2, proofs) \right)^{\circ} (p_2, proofs) \]
\]
[Lemma D.6.10]

\[
\begin{align*}
= \Omega, \ \star \cdot \left( \Pi < \llbracket t_1 \rrbracket \Gamma_L \Gamma_T (p_1, proofs), \\
\llbracket t_2 \mid t_3 \rrbracket \Gamma_L \Gamma_T (p_2, proofs) \right)^{\circ} (p_2, proofs) \]
\]
[Lemma D.6.12]

159
\[
\begin{align*}
&= \infty/(\Omega; \ast (\Pi < (\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >) ) \\
&= \infty/(\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; p_2; proofs)) \\
&= \infty/(\llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T, (\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T)^\circ (p_1; p_2; proofs)) \\
&= \llbracket (t_1 \llbracket t_2 \rrbracket) \mid (t_2 \llbracket t_3 \rrbracket) \rrbracket_{\Gamma_L} \Gamma_T (p_1; p_2; proofs))
\end{align*}
\]

Law 61. \((t_1 \mid t_2)\llbracket t_3 \rrbracket = (t_1 \llbracket t_3 \rrbracket) \mid (t_2 \llbracket t_3 \rrbracket)

\[
\begin{align*}
&= \llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs) \\
&= \Omega; \ast (\Pi < (\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >) \\
&= \Omega; \ast (\Pi < (\llbracket \llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >)) \\
&= \Omega; \ast (\Pi < (\llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >)) \\
&= \Omega; \ast (\Pi < (\llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >)) \\
&= \Omega; \ast (\Pi < (\llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >)) \\
&= \Omega; \ast (\Pi < (\llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_3 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >)) \\
&= \llbracket (t_1 \llbracket t_2 \rrbracket \mid (t_2 \llbracket t_3 \rrbracket) \rrbracket_{\Gamma_L} \Gamma_T (p_1; p_2; proofs))
\end{align*}
\]

Law 62. \(! (t_1 \llbracket t_2 \rrbracket) = !t_1 \llbracket t_2 \rrbracket

\[
\begin{align*}
&= head'(\Omega; \ast (\Pi < (\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >))) \\
&= \Omega; \ast (head'(\Pi < (\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs), \llbracket t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_2; proofs) >)) \\
&= \Omega; \ast (\Pi < head'(\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs))) \\
&= \Omega; \ast (\Pi < head'(\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs))) \\
&= \Omega; \ast (\Pi < head'(\llbracket t_1 \rrbracket_{\Gamma_L} \Gamma_T (p_1; proofs))) \\
&= \llbracket !t_1 \mid !t_2 \rrbracket_{\Gamma_L} \Gamma_T (p_1; p_2; proofs))
\end{align*}
\]
D.5 Laws on con

The side-conditions of these laws use the function $\phi$ which extracts the set of free-variables of the tactic to which it is applied.

**Law 63.** \((\text{con } v \bullet t_1; t_2) = (\text{con } v \bullet t_1); t_2\) provided \(v \notin \phi t_2\)

\[
\begin{align*}
&\llbracket \text{con } v \bullet t_1; t_2 \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1; t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1(v); t_2(v))) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1(v); t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1(v); t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\\text{con } v \bullet t_1; t_2) \rrbracket_{L \Gamma T}
\end{align*}
\]

[Definition of con]

[Distribution of ;]

[provided \(v \notin \phi t_2\)]

[Law 6]

[Definition of con]

\[
\square
\]

**Law 64.** \((\text{con } v \bullet t_1; t_2) = t_1; (\text{con } v \bullet t_2)\) provided \(v \notin \phi t_1\) and \(t_1 = t_1\)

\[
\begin{align*}
&\llbracket (\text{con } v \bullet t_1; t_2) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1; t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1(v); t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1; t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t_1; t_2)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\text{con } v \bullet t_1; t_2) \rrbracket_{L \Gamma T}
\end{align*}
\]

[Definition of con]

[Distribution of ;]

[provided \(v \notin \phi t_2\)]

[Law 7]

[Definition of con]

\[
\square
\]

**Law 65.** \((\text{con } v \bullet t) = t\) provided \(v \notin \phi t\)

\[
\begin{align*}
&\llbracket \text{con } v \bullet t \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t)) \rrbracket_{L \Gamma T} \\
&= \llbracket \emptyset \rrbracket_{L \Gamma T} \\
&= \llbracket t \rrbracket_{L \Gamma T}
\end{align*}
\]

[Definition of con]

[Provided \(v \notin \phi t\)]

[t \mid t = t]

\[
\square
\]

**Law 66.** \((\text{con } v \bullet t) = (\text{con } u \bullet t[v \backslash u])\) provided \(u \notin \phi t\)

\[
\begin{align*}
&\llbracket \text{con } v \bullet t \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall v \in \text{TERM}(t)) \rrbracket_{L \Gamma T} \\
&= \llbracket (\:\forall u \in \text{TERM}(t)) \rrbracket_{L \Gamma T} \\
&= \llbracket \text{con } u \bullet t[v \backslash u] \rrbracket_{L \Gamma T}
\end{align*}
\]

[Definition of con]

[provided \(u \notin \phi t\)]

[Definition of con]

\[
\square
\]

161
D.6 Lemmas

D.6.1 Lemma 3

We prove this Lemma by structural induction, using definition 3 above.

**Base cases:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip =!skip</td>
<td>8</td>
</tr>
<tr>
<td>fail =!fail</td>
<td>9</td>
</tr>
<tr>
<td>law l =!(law l)</td>
<td>10</td>
</tr>
</tbody>
</table>

**Inductive step:** We assume that \( t_i \) is sequential, this means \( t_i =!t_i \)

We must consider the following cases

a) \( !t \)

\[ \begin{align*}
= & ! (t \mid \text{fail}) \quad \text{[Law 2]} \\
= & !(! (t \mid \text{fail}) \quad \text{[Law 14]} \\
= & !(!t) \quad \text{[Law 2]} \\
\end{align*} \]

b) \( \text{fails } t =!\text{fails } t \)

\[ \text{[Law 22]} \]

c) \( \text{succs } t =!\text{succs } t \)

\[ \text{[Law 43]} \]

d) \( t_1; t_2 \)

\[ \begin{align*}
= & ! (t_1; t_2) \quad \text{[Inductive Hypothesis]} \\
= & !(! (t_1; t_2)) \quad \text{[Law 12]} \\
= & ! (t_1; t_2) \quad \text{[Inductive Hypothesis]} \\
\end{align*} \]

In the following proofs we use the notation \( \text{list}_i \) to denote the \( i \)th-element of the list \( \text{list} \).

e) \( \boxed{\text{tactics}} \boxed{\text{if } GC \text{ fi, proofs}} \)

\[ \begin{align*}
= & \Omega_f \text{ (proofs) } (mkGC (extractG GC)) \\
& \quad (\Pi(apply ((([\text{tactics}]_L \Gamma_L \Gamma_{T_r}) \ast tactics) (\Phi_r \ast (extractP GC)))))) \\
& \quad \text{[Definition of } \Pi \text{]} \\
= & \Omega_f \text{ (proofs) } (mkGC (extractG GC)) \\
& \quad (\Pi([\text{tactics},_L \Gamma_L \Gamma_{T_r}) (\Phi_r \ast extractP GC))) \\
& \quad \text{[Definition of apply and *]} \\
\end{align*} \]
\[
\omega_g \text{ (proofs)} \ (mkGC \ (extractG \ GC) \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Assumption]}
\]

\[
\omega_g \text{ (proofs)} \ (mkGC \ (extractG \ GC) \\
(\prod(\text{head}'([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),))) \\
\text{[Definition of !]}
\]

\[
\omega_g \text{ (proofs)} \ (mkGC \ (extractG \ GC) \\
\text{head}'(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Lemma 18]}
\]

\[
\omega_g \text{ (proofs)} * \ (mkGC \ (extractG \ GC) \\
\text{head}'(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Definition of } \omega_g \text{]} \\
\]

\[
\omega_g \text{ (proofs)} * \ ((\text{insertG} \ (extractG \ GC)) * \\
\text{head}'(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Definition of mkGC]}
\]

\[
\omega_g \text{ (proofs)} * \ (\text{head}'((\text{insertG} \ (extractG \ GC))) * \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Lemma 17]}
\]

\[
\omega_g \text{ (proofs)} * \ (\text{head}'(\text{mkGC}(\text{extractG} \ GC) \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),))) \\
\text{[Definition of mkGC]}
\]

\[
\omega_g \text{ (proofs)} * \ (\text{head}'(\omega_g \text{ (proofs)} \ (mkGC \ (extractG \ GC) \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),))) \\
\text{[Lemma 17]}
\]

\[
\omega_g \text{ (proofs)} * \ (\text{mkGC} \ (extractG \ GC) \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Definition of } \omega_g \text{ and mkGC]}
\]

\[
\omega_g \text{ (proofs)} * \ (\text{mkGC} \ (extractG \ GC) \\
(\prod(\text{apply} \ (((\!\!\!\!tactics,\Gamma_L \ \Gamma_T) * \ \text{tactics}) \ (\Phi_r * (\text{extractP} \ GC)))))) \\
\text{[Definition of apply and *]}
\]

\[
\omega_g \text{ (proofs)} * \ (\text{mkGC} \ (extractG \ GC) \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Definition of if } \omega_g \text{ and } \text{head'}]
\]

f) \[
\text{do} \ \text{tactics} \ \text{od} \ (\text{do} \ GC \ \text{od}, \ \text{proofs}) \\
\omega_d \text{ (proofs)} \ (mkGC \ (extractG \ GC) \\
(\prod(\text{apply} \ (((\!\!\!\!tactics,\Gamma_L \ \Gamma_T) * \ \text{tactics}) \ (\Phi_r * (\text{extractP} \ GC)))))) \\
\text{[Definition of } \text{do} \ \text{od} \text{]} \\
\]

\[
\omega_d \text{ (proofs)} \ (mkGC \ (extractG \ GC) \\
(\prod([\!\!\!\!tactics,]\Gamma_L \ \Gamma_T) \ (\Phi_r * \ \text{extractP} \ GC),)) \\
\text{[Definition of apply and *]}
\]

163
\[
= \Omega_{\text{do}} (\text{proofs}) \ (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad (\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC}))) \hspace{1cm} \text{[Assumption]}
\]

\[
= \Omega_{\text{do}} (\text{proofs}) \ (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad (\Pi(\text{head'}(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC})))) \hspace{1cm} \text{[Definition of !]}
\]

\[
= \Omega_{\text{do}} (\text{proofs}) \ (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad \text{head'}(\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC}))) \hspace{1cm} \text{[Lemma 18]}
\]

\[
= \text{do} (\text{proofs}) * (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad \text{head'}(\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC}))) \hspace{1cm} \text{[Definition of \(\Omega_{\text{do}}\)]}
\]

\[
= \text{do} (\text{proofs}) * ((\text{insertG} \ (\text{extractG} \ \text{GC})) * \ \\
\quad \text{head'}(\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC}))) \hspace{1cm} \text{[Definition of \text{mkGC}]}
\]

\[
= \text{do} (\text{proofs}) * (\text{head'}((\text{insertG} \ (\text{extractG} \ \text{GC}))) * \ \\
\quad (\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC}))) \hspace{1cm} \text{[Lemma 17]}
\]

\[
= \text{do} (\text{proofs}) * \ (\text{head'}(\text{mkGC}(\text{extractG} \ \text{GC}) \ \\
\quad (\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC})))) \hspace{1cm} \text{[Definition of \text{mkGC}]}
\]

\[
= \text{head'}(\text{do} (\text{proofs}) * \ (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad (\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC})))) \hspace{1cm} \text{[Lemma 17]}
\]

\[
= \text{head'}(\Omega_{\text{do}} (\text{proofs}) \ (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad (\Pi(\llbracket \text{tactics} \rrbracket) (\Phi_r * \text{extractP} \ \text{GC})))) \hspace{1cm} \text{[Definition of \(\Omega_{\text{do}}\) and \text{mkGC}]}
\]

\[
= \text{head'}(\Omega_{\text{do}} (\text{proofs}) \ (\text{mkGC} \ (\text{extractG} \ \text{GC}) \ \\
\quad (\Pi(\text{apply} (\llbracket \llbracket \text{tactics} \rrbracket) (\Phi_r * (\text{extractP} \ \text{GC})))))) \hspace{1cm} \text{[Definition of \text{apply} and *]}
\]

\[
= \text{do} \text{tactics \ od} (\text{do} \ \text{GC} \ \text{od}, \text{proofs}) \hspace{1cm} \text{[Definition of \text{do \ od} and \text{head}']}
\]

\[
g) \quad \text{var} \ t_1 \llbracket(p, \ pobs) \hspace{1cm} \text{[Definition of \text{var} \ \llbracket]}
\]

\[
= (\text{var} \ d) * (\llbracket t \rrbracket \Gamma_L \Gamma_T \ (p, \ pobs)) \hspace{1cm} \text{[Assumption]}
\]

\[
= (\text{var} \ d) * (\llbracket t \rrbracket \Gamma_L \Gamma_T \ (p, \ pobs)) \hspace{1cm} \text{[Definition of !]}
\]

\[
= \text{head'}((\text{var} \ d) * (\llbracket t \rrbracket \Gamma_L \Gamma_T \ (p, \ pobs))) \hspace{1cm} \text{[Lemma 17]}
\]

\[
= !\text{var} \ t_1 \llbracket(p, \ pobs) \hspace{1cm} \text{[Definition of ! and \text{var} \ \llbracket]}
\]

\[
h) \quad \text{con} \ t_1 \llbracket(p, \ pobs)
\]
= (cons $d$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{cons} \]
= (cons $d$) * ($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Assumption} \]
= (cons $d$) * ($\text{head}'([t]\Gamma_L\Gamma_T (p, pobs))$)  \[\text{Definition of } \text{!} \]
= head'((cons $d$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$))  \[\text{Lemma 17} \]
= head'($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{!} \text{ and } \text{cons} \]

i) $\text{pmain} t_1 \square (p, pobs)$
= (procm $n$ $p$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{pmain} \]
= (procm $n$ $p$) * ($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Assumption} \]
= (procm $n$ $p$) * ($\text{head}'([t]\Gamma_L\Gamma_T (p, pobs))$)  \[\text{Definition of } \text{!} \]
= head'((procm $n$ $p$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$))  \[\text{Lemma 17} \]
= head'($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{!} \text{ and } \text{pmain} \]

j) $\text{pmainvariant} t_1 \square (p, pobs)$
= (variant $n$ $b$ $v$ $e$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{pmainvariant} \]
= (variant $n$ $b$ $v$ $e$) * ($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Assumption} \]
= (variant $n$ $b$ $v$ $e$) * ($\text{head}'([t]\Gamma_L\Gamma_T (p, pobs))$)  \[\text{Definition of } \text{!} \]
= head'((variant $n$ $b$ $v$ $e$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$))  \[\text{Lemma 17} \]
= head'($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{!} \text{ and } \text{pmainvariant} \]

k) $\text{pbody} t_1 \square (p, pobs)$
= (procb $n$ $p$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{pbody} \]
= (procb $n$ $p$) * ($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Assumption} \]
= (procb $n$ $p$) * ($\text{head}'([t]\Gamma_L\Gamma_T (p, pobs))$)  \[\text{Definition of } \text{!} \]
= head'((procb $n$ $p$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$))  \[\text{Lemma 17} \]
= head'($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{!} \text{ and } \text{pbody} \]

l) $\text{pbodyvariant} t_1 \square (p, pobs)$
= (variantb $n$ $b$ $v$ $e$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{pbodyvariant} \]
= (variantb $n$ $b$ $v$ $e$) * ($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Assumption} \]
= (variantb $n$ $b$ $v$ $e$) * ($\text{head}'([t]\Gamma_L\Gamma_T (p, pobs))$)  \[\text{Definition of } \text{!} \]
= head'((variantb $n$ $b$ $v$ $e$) * ($[t]\Gamma_L\Gamma_T (p, pobs)$))  \[\text{Lemma 17} \]
= head'($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Definition of } \text{!} \text{ and } \text{pbodyvariant} \]

m) $\text{pbodymain} t_b \ t_m \square (p, pobs)$
= $\text{pbody} t_b \square ; \text{pmain} t_m \square (p, pobs)$  \[\text{Definition of } \text{pbodymain} \]
= $\text{pbody} t_b \square ; \text{pmain} t_m \square (p, pobs)$  \[\text{Items (i) and (k)} \]
= head'($\!\Gamma_L\Gamma_T (p, pobs)$)  \[\text{Law 12} \]

165
\[
\begin{align*}
&= !\text{\textbf{pbody}} t_b \parallel \text{\textbf{pbody}} t_m \parallel (p, \text{pobs}) \quad \text{(Items (i) and (k))} \\
&= !\text{\textbf{pbody}} \text{\textbf{main}} t_b \parallel \text{\textbf{main}} t_m \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{pbody}} \text{\textbf{main}} \parallel) \\
&= !\text{\textbf{pbody}} \text{\textbf{main}} t_b \parallel \text{\textbf{main}} t_m \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{pbody}} \text{\textbf{main}} \parallel) \\
&= !\text{\textbf{pbody}} \text{\textbf{variant}} t_b \parallel \text{\textbf{variant}} t_m \parallel (p, \text{pobs}) \quad \text{(Law 12)} \\
&= !\text{\textbf{pbody}} \text{\textbf{variant}} t_b \parallel \text{\textbf{variant}} t_m \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{pbody}} \text{\textbf{variant}} \parallel) \\
&= !\text{\textbf{variant}} t_b \parallel \text{\textbf{variant}} t_m \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{variant}} \parallel) \\
&= !\text{\textbf{main}} t_b \parallel \text{\textbf{main}} t_m \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{main}} \parallel) \\
&= !\text{\textbf{main}} t_b \parallel \text{\textbf{main}} t_m \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{main}} \parallel) \\
&= !\text{\textbf{val}} t_1 \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{val}} \parallel) \\
&= (\text{\textbf{val}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs})) \quad \text{(Definition of } \text{\textbf{val}} \parallel) \\
&= (\text{\textbf{val}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs})) \quad \text{(Assumption)} \\
&= (\text{\textbf{val}} t v t a) * (head'([t] \Gamma_{L \Gamma_T} (p, \text{pobs}))) \quad \text{(Definition of } !) \\
&= head'((\text{\textbf{val}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs}))) \quad \text{(Lemma 17)} \\
&= !\text{\textbf{val}} t_1 \parallel (p, \text{pobs}) \quad \text{(Definition of } ! \text{ and } \text{\textbf{val}} \parallel) \\
&= !\text{\textbf{res}} t_1 \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{res}} \parallel) \\
&= (\text{\textbf{res}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs})) \quad \text{(Definition of } \text{\textbf{res}} \parallel) \\
&= (\text{\textbf{res}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs})) \quad \text{(Assumption)} \\
&= (\text{\textbf{res}} t v t a) * (head'([t] \Gamma_{L \Gamma_T} (p, \text{pobs}))) \quad \text{(Definition of } !) \\
&= head'((\text{\textbf{res}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs}))) \quad \text{(Lemma 17)} \\
&= !\text{\textbf{res}} t_1 \parallel (p, \text{pobs}) \quad \text{(Definition of } ! \text{ and } \text{\textbf{res}} \parallel) \\
&= !\text{\textbf{val-res}} t_1 \parallel (p, \text{pobs}) \quad \text{(Definition of } \text{\textbf{val-res}} \parallel) \\
&= (\text{\textbf{val-res}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs})) \quad \text{(Definition of } \text{\textbf{val-res}} \parallel) \\
&= (\text{\textbf{val-res}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs})) \quad \text{(Assumption)} \\
&= (\text{\textbf{val-res}} t v t a) * (head'([t] \Gamma_{L \Gamma_T} (p, \text{pobs}))) \quad \text{(Definition of } !) \\
&= head'((\text{\textbf{val-res}} t v t a) * ([t] \Gamma_{L \Gamma_T} (p, \text{pobs}))) \quad \text{(Lemma 17)} \\
&= !\text{\textbf{val-res}} t_1 \parallel (p, \text{pobs}) \quad \text{(Definition of } ! \text{ and } \text{\textbf{val-res}} \parallel) \\
&= !\text{\textbf{par-command}} t_1 \parallel (p, \text{post}) \quad \text{(Definition of } \text{\textbf{par-command}} \parallel) \\
&= (\text{\textbf{par-command}} d) * ([t] \Gamma_{L \Gamma_T} (p, \text{post})) \quad \text{(Definition of } \text{\textbf{par-command}} \parallel) \\
&= (\text{\textbf{par-command}} d) * ([t] \Gamma_{L \Gamma_T} (p, \text{post})) \quad \text{(Assumption)} \\
&= (\text{\textbf{par-command}} d) * (head'([t] \Gamma_{L \Gamma_T} (p, \text{post}))) \quad \text{(Definition of } !) \\
&= head'((\text{\textbf{par-command}} d) * ([t] \Gamma_{L \Gamma_T} (p, \text{post}))) \quad \text{(Lemma 17)} 
\end{align*}
\]
= !\text{parcommand}t_1[]} (p, post) \quad [\text{Definition of !}]

\[\text{parcommand}[]\] \quad [\text{Definition of \text{parcommand}[]}]

\textbf{D.6.2 Lemma 7}

\[f \bullet \infty/ = \infty/ \bullet f \ast\ast\]

This Lemma can be proved by induction as follows:

**Base Case:** \(\text{lst} = []\)

\[f \bullet \infty/[]\]

\[= f \ast \infty/[]\]

\[= f \ast []\]

\[= []\]

\[= \infty/[]\]

\[= \infty/ f \ast\ast[]\]

**Base Case:** \(\text{lst} = \bot\)

\[f \bullet \infty/[]\]

\[= f \ast \bot\]

\[= \bot\]

\[= \infty/ \bot\]

\[= \infty/ f \ast\ast \bot\]

**Base Case:** \(\text{lst} = [x]\)

\[f \bullet \infty/[[x]]\]

\[= f \ast [x]\]

\[= [f \ast x]\]

\[= \infty/[[f \ast x]]\]

\[= \infty/ f \ast\ast [[x]]t\]

**Inductive step:**

\(fs : \text{finseq}(\text{pfiseq } X)\)

\(fs : \text{pfiseq}(\text{pfiseq } X)\)

We assume that

\[f \bullet \infty/ fs = \infty/ f \ast\ast fs.\]

\[f \bullet \infty/ is = \infty/ f \ast\ast is.\]

We must prove that \(f \bullet \infty/ fs \infty/ is = \infty/ f \ast\ast fs \infty/ is.\)
\( f \ast \infty / fs \infty \ast is \)  
\[ = (f \ast \infty / fs) \infty (f \ast \infty \ast is) \]  
\[ = (\infty / f \ast fs) \infty (\infty / f \ast fs \ast is) \]  
\[ = \infty / (f \ast fs) \infty (f \ast is) \]  
\[ = \infty / f \ast is \]  
[Lemma D.6.21]  
[Assumption]  
[Lemma D.6.5]  
[Lemma D.6.12]

\[ D.6.3 \text{ Lemma 8} \]

\( \infty / \bullet \infty / \infty / \ast \)

This lemma is also proved by induction as seen below:

\( \infty / \bullet \infty / \infty / \ast \)

**Base Case:** lst = [ ]
\[ \infty / \bullet \infty / [ ] \]  
\[ = \infty / \bullet \infty / \ast [ ] \]  
[Definition of \( \ast \)]

**Base Case:** lst = ⊥
\[ \infty / \bullet \infty / [ ] \]  
\[ = \bot \]  
[Definition of \( \infty / \)]  
\[ = \infty / \bullet \infty / \ast \bot \]  
[Definition of \( \infty / \ast \)]

**Base Case:** lst = [[[x]]]
\[ \infty / \bullet \infty / [[[x]]] \]  
\[ = [x] \]  
[Definition of \( \ast \)]  
\[ = \infty / [x] \]  
[Definition of \( \ast \)]  
\[ = \infty / \bullet \infty / \ast [[[x]]] \]  
[Definition of \( \ast \)]

**Inductive step:**
\( fs : \text{finseq(pfiSeq } X) \)  
\( fs : \text{pfiSeq(pfiSeq } X) \)

We assume that
\( \infty / \bullet \infty / fs = \infty / \bullet \infty / \ast fs \)  
\( \infty / \bullet \infty / is = \infty / \bullet \infty / \ast is \)

We must prove that \( \infty / \bullet \infty / fs \infty is = \infty / \bullet \infty / \ast fs \infty is \)


\[
\begin{align*}
\exists/ \bullet \exists/ f \exists/ \exists \text{is} \\
= \exists/ (\exists/ f \exists/ \exists/ \exists \text{is}) & \quad \text{[Lemma D.6.5]} \\
= (\exists/ \bullet \exists/ f) \exists/ (\exists/ \bullet \exists/ \exists \text{is}) & \quad \text{[Lemma D.6.5]} \\
= (\exists/ \bullet \exists/ f) \exists/ ((\exists/ \bullet \exists/ \exists \text{is})) & \quad \text{[Assumption]} \\
= \exists/ (\exists/ f) \exists/ (\exists/ \exists \text{is}) & \quad \text{[Lemma D.6.5]} \\
= \exists/ \bullet \exists/ (\exists/ f) \exists/ \exists \text{is} & \quad \text{[Lemma D.6.12]}
\end{align*}
\]

\[\square\]

### D.6.4 Lemma 9

\[\text{head'} l = l \text{ if } l = [\ ] \text{ or } \# l = 1\]

See Lemma D.6.20.

### D.6.5 Lemma 10

\[\exists/ (a \exists/ b) = \exists/ a \exists/ \exists/ b\]

This lemma is proved by induction as follows:

\[\exists/ (a \exists/ b) = \exists/ a \exists/ \exists/ b\]

**Base Case:** \(a = [\ ]\)

\[\exists/ (a \exists/ b) = \exists/ ([\ ] \exists/ b) = \exists/ b = [\ ] \exists/ \exists/ b = \exists/ [\ ] \exists/ b\]

**Base Case:** \(a = \bot\)

\[\exists/ (a \exists/ b) = \bot \exists/ \exists/ b = \exists/ \bot \exists/ \exists/ (a \exists/ b)\]

**Base Case:** \(a = [e]\)

\[\exists/ (a \exists/ b) = \exists/ ([e] \exists/ b) = \bigcup_{\exists/ [e]} \{c : [e] \exists/ b \circ \wedge c\} = \bigcup_{\exists/ [e]} \{c : [e] ; y : b \circ x \wedge y \circ \wedge c\} = \{c : [e] ; y : b \circ x \wedge y \circ \wedge c\}\]

169
We need to make induction also on the list \( b \).

**Base Case: \( b = [\] \)**

\[
\begin{align*}
\{ c : \{ x : [e] ; y : [ ] \} \cdot x^\wedge y \} \cdot c = \{ c : \{ x : [e] \} \cdot x \} \cdot ^\wedge c \quad &\text{[Definition of } ^\wedge \text{]} \\
\{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : [ ] \} \cdot x^\wedge y \} \quad &\text{[Set theory]} \\
\{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : b \cdot ^\wedge c \} \cdot x^\wedge y \} \quad &\text{[Set theory]} \\
\{ x : \infty ; c ; y : \infty / b \cdot x^\wedge y \} \quad &\text{[Definition of } \infty \text{]} \\
\infty / [e] \infty / b \quad &\text{[Definition of } \infty \text{]} \\
\end{align*}
\]

**Base Case: \( b = \bot \)**

\[
\begin{align*}
\{ c : \{ x : [e] ; y : \bot \cdot x^\wedge y \} \} \cdot c = \bot \\
\{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \bot \cdot x^\wedge y \} \quad &\text{[Definition of } ^\wedge \text{]} \\
\{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : \bot \} \cdot x^\wedge y \} \quad &\text{[Set theory]} \\
\{ x : \infty ; c ; y : \infty / b \cdot x^\wedge y \} \quad &\text{[Definition of } \infty \text{]} \\
\infty / [e] \infty / b \quad &\text{[Definition of } \infty \text{]} \\
\end{align*}
\]

**Base Case: \( b = [v] \)**

\[
\begin{align*}
\{ c : \{ x : [e] ; y : [v] \cdot x^\wedge y \} \} \cdot c = \{ c : e^\wedge v \cdot ^\wedge c \} \\
\{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : [v] \} \cdot x^\wedge y \} \quad &\text{[Set theory]} \\
\{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : b \} \cdot ^\wedge c \} \cdot x^\wedge y \} \quad &\text{[Set theory]} \\
\{ x : \infty ; c ; y : \infty / b \cdot x^\wedge y \} \quad &\text{[Definition of } \infty \text{]} \\
\infty / [e] \infty / b \quad &\text{[Definition of } \infty \text{]} \\
\end{align*}
\]

**Inductive Step on \( b \):**

\( hs : \text{finseq } X \)

\( is : \text{pfisseq } X \)

We assume that
\[
\begin{align*}
\{ c : \{ x : [e] ; y : hs \cdot x^\wedge y \} \} \cdot ^\wedge c = \{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : hs \} \cdot x^\wedge y \} \\
\{ c : \{ x : [e] ; y : is \cdot x^\wedge y \} \} \cdot ^\wedge c = \{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : is \} \cdot x^\wedge y \} \\
\end{align*}
\]

We must prove that
\[
\begin{align*}
\{ c : \{ x : [e] ; y : hs \infty is \cdot x^\wedge y \} \} \cdot ^\wedge c = \{ x : \{ c : [e] \} \cdot ^\wedge c \}; y : \{ c : hs \infty is \} \cdot ^\wedge c \} \cdot x^\wedge y \} \\
\{ c : \{ x : [e] ; y : hs \infty is \cdot x^\wedge y \} \} \cdot ^\wedge c \\
= \{ c : \{ x : [e] ; y : hs \cdot x^\wedge y \} \cup \{ x : [e] ; y : is \cdot x^\wedge y \} \} \cdot ^\wedge c \quad &\text{[Set Comprehension]} \\
\end{align*}
\]
We must prove that
\[ \{ c : \{ x : [e] ; y : hs \cdot x^\wedge y \} \cdot ^\wedge c \} \cup \{ c : \{ x : [e] ; y : hs \cdot x^\wedge y \} \cdot ^\wedge c \} \]

\[ = \{ x : \{ c : [e] \cdot ^\wedge c \} ; y : \{ c : hs \cdot ^\wedge c \cdot x^\wedge y \} \cup \{ x : \{ c : [e] \cdot ^\wedge c \} ; y : \{ c : is \cdot ^\wedge c \cdot x^\wedge y \} \} \]

\[ = \{ x : \{ c : [e] \cdot ^\wedge c \} ; y : \{ c : hs \cdot is \cdot ^\wedge c \cdot x^\wedge y \} \}

We assume that
\[ \{ c : \{ x : fs ; y : b \cdot x^\wedge y \} \cdot ^\wedge c \} = \{ x : \{ c : fs \cdot ^\wedge c \} ; y : b \cdot x^\wedge y \} \]
\[ \{ c : \{ x : gs ; y : b \cdot x^\wedge y \} \cdot ^\wedge c \} = \{ x ; \{ c : gs \cdot ^\wedge c \} ; y : b \cdot x^\wedge y \} \]

We must prove that
\[ \{ c : \{ x : fs \cdot \infty gs ; y : b \cdot x^\wedge y \} \cdot ^\wedge c \} = \{ x ; \{ c : fs \cdot \infty gs \cdot ^\wedge c \} ; y : b \cdot x^\wedge y \} \]

\[ = \{ c : \{ x : fs ; y : b \cdot x^\wedge y \} \cup \{ x : gs ; y : b \cdot x^\wedge y \} \cdot ^\wedge c \}
= \{ c : \{ x : fs ; y : b \cdot x^\wedge y \} \cdot ^\wedge c \} \cup \{ c : \{ x : gs ; y : b \cdot x^\wedge y \} \cdot ^\wedge c \} \]
\[ = \{ x ; \{ c : \{ x : fs \cdot ^\wedge c \} ; y : b \cdot \times x^\wedge y \} \cup \{ x : \{ c : gs \cdot ^\wedge c \} ; y : b \cdot x^\wedge y \} \}
[Assumption]
\[ = \{ x : \{ c : \{ x : fs \cdot ^\wedge c \} ; y : b \cdot x^\wedge y \} \}
[Set Comprehension]

\[ \square \]

**D.6.6 Lemma 11**

\[ head'(l \cdot \infty ((\infty/ \cdot f*) l)) = head' l \]

This lemma is also proved by induction as seen below:

\[ head'(l \cdot \infty ((\infty/ \cdot f*) l)) = head' l \]

**Base Case:** \( l = [ ] \)
\[ head'([ ] \cdot \infty ((\infty/ \cdot f*) [ ])) \]
\[ = head'([ ] \cdot \infty [ ] \cdot [ ])) \]
\[ = head'([ ] \cdot \infty [ ]) \]
\[ = head'([ ]) \]
[Definition of \( \infty \)]
[Definition of \( f* \)]
[Definition of \( \infty \)]

**Base Case:** \( l = \perp \)
\[ head'((\perp \cdot \infty ((\infty/ \cdot f*) \perp)) \]

171
\[ \text{Base Case: } \{x\} = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ \text{Inductive Step on } a: \]

\[ \text{fs : finseq } X \]

\[ \text{is : pfiseq } X \]

\[ \text{Inductive step:} \]

\[ \text{We assume that} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}' \text{ fs} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}' \text{ is} \]

\[ \text{We must prove that} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}'(\{x\} \cdot f \ast ) \]

\[ = \text{head}'(\{x\} \cdot f \ast ) \]

\[ \text{D.6.7 Lemma 12} \]

\[ \text{This lemma is proved by induction as follows:} \]

\[ \text{Base Case:} \]

\[ \{x\} = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ \text{Inductive step:} \]

\[ \text{fs : finseq } X \]

\[ \text{is : pfiseq } X \]

\[ \text{Inductive step:} \]

\[ \text{We assume that} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}' \text{ fs} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}' \text{ is} \]

\[ \text{We must prove that} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}'(\{x\} \cdot f \ast ) \]

\[ = \text{head}'(\{x\} \cdot f \ast ) \]

\[ \text{D.6.7 Lemma 12} \]

\[ \text{This lemma is proved by induction as follows:} \]

\[ \text{Base Case:} \]

\[ \{x\} = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ = \text{head}'(\{x\}) \]

\[ \text{Inductive step:} \]

\[ \text{fs : finseq } X \]

\[ \text{is : pfiseq } X \]

\[ \text{Inductive step:} \]

\[ \text{We assume that} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}' \text{ fs} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}' \text{ is} \]

\[ \text{We must prove that} \]

\[ \text{head}'(\{x\} \cdot f \ast ) = \text{head}'(\{x\} \cdot f \ast ) \]

\[ = \text{head}'(\{x\} \cdot f \ast ) \]
\[fs : \text{finseq } X\]
\[is : \text{pfiseq } X\]

We assume that
\[\text{head}'(\infty \leftarrow l_2 \infty fs) = \text{head}'(\infty/[fs, l_2])\]
\[\text{head}'(\infty \leftarrow l_2 \infty is) = \text{head}'(\infty/[is, l_2])\]

We must prove that
\[\text{head}'((fs \leftarrow l_2 \infty is) \infty \leftarrow l_2 \infty (fs \leftarrow is)) = \text{head}'(\infty/[\infty \leftarrow \text{is}, l_2])\]

\[= \text{head}'((fs \leftarrow is) \infty \leftarrow l_2 \infty (fs \leftarrow is))\]
\[= \text{head}'(fs \leftarrow is)\]
\[= \text{head}'(\text{is} \infty \leftarrow l_2)\]
\[= \text{head}'(\infty/[\text{is}, l_2])\]

[D.6.8 Lemma 13]

\[\begin{array}{l}
\text{[a : as | a \leftarrow A, as \leftarrow B] \land \forall l : B \bullet \#l = 1} \\
\quad = \langle a, b \rangle | a \leftarrow A, b \leftarrow B
\end{array}\]

This lemma is proved by induction as follows:

Base Case:
\[B = [\ ]\]
\[\text{[a : as | a \leftarrow A, as \leftarrow B] = [a : as | a \leftarrow A, as \leftarrow \rangle}
\[= [\ ]\]
\[= \langle a, b \rangle | a \leftarrow A, b \leftarrow [\ ]\]
\[= \langle a, b \rangle | a \leftarrow A, b \leftarrow B\]

Inductive step: We assume that
\[\text{[a : as | a \leftarrow A, as \leftarrow B] \land \forall l : B \bullet \#l = 1} = \langle a, b \rangle | a \leftarrow A, b \leftarrow B >
\]

We must prove that
\[\text{[a : as | a \leftarrow A, as \leftarrow \{< x >\} \cup B} \land \forall l : \{< x >\} \cup B
\quad \bullet \#l = 1 = \langle a, b \rangle | a \leftarrow A, b \leftarrow \{< x >\} \cup B\]
\[= \text{[a : as | a \leftarrow A, as \leftarrow \{< x >\} \cup B}\]
\[\infty [a : as | a \leftarrow A, as \leftarrow B]̂\]

[Property of \infty]
\[ \langle a, x \rangle | a \leftarrow A \] \{ a : as | a \leftarrow A, as \leftarrow B \} \[ \langle a, b \rangle | a \leftarrow A, b \leftarrow \{ \langle x \rangle \} \] \{ a : as | a \leftarrow A, b \leftarrow B \} \[ \langle a, b \rangle | a \leftarrow A, b \leftarrow \{ \langle x \rangle \} \cup B \] \]

\[ \text{List Comprehension} \]
\[ \text{List Comprehension} \]
\[ \text{Assumption} \]
\[ \text{Property of } \infty \]

D.6.9 Lemma 14

\[ \forall t : \text{Tactic}, \forall (p, pr) \in [t] \Gamma_L \Gamma_T (p', pr') \bullet pr' \subseteq pr \]

This lemma is proved by induction as seen below:

Base Cases:
\[ t = \text{law } lname (args) \]
\[ [[ \text{law } l (args)]] \Gamma_L \Gamma_T (p', pr') \]

This is defined as

\[ \text{if } lname \in \text{dom } \Gamma_L \land \text{args} \in \text{dom } \Gamma_L lname \land p' \in \text{dom } \Gamma_L lname \text{ args then } \]
\[ \text{let } \text{newp, proofs} = \Gamma_L lname \text{ args } p' \in \]
\[ [(\text{newp, proofs} \cup nproofs)] \]
\[ \text{else } [] \]

As \( pr' \subseteq pr' \cup proofs \) we can conclude that
\[ \forall (p, pr) \in [[ \text{law } l (args)]] \Gamma_L \Gamma_T (p', pr') \bullet pr' \subseteq pr \]

\[ t = \text{skip} \]
\[ [\text{skip}] \Gamma_L \Gamma_T (p', pr') = [(p', pr')] \]
So \( \forall (p, pr) \in [\text{skip}] \Gamma_L \Gamma_T (p', pr') \bullet pr' \subseteq pr \)

\[ t = \text{fail} \]
\[ [\text{fail}] \Gamma_L \Gamma_T (p', pr') = [ ] \]
So \( \forall (p, pr) \in [\text{fail}] \Gamma_L \Gamma_T (p', pr') \bullet pr' \subseteq pr \)

\[ t = \text{abort} \]
\[ [\text{abort}] \Gamma_L \Gamma_T (p', pr') = \bot \]
So \( \forall (p, pr) \in [\text{abort}] \Gamma_L \Gamma_T (p', pr') \bullet pr' \subseteq pr \)
Inductive step: We assume that
∀ t₁, t₂ : Tactic and
∀ (p, pr) ∈ [[t₁] pr L] Γ T (p', pr') • pr' ⊆ pr and
∀ (p, pr) ∈ [[t₂] pr L] Γ T (p', pr') • pr' ⊆ pr
We must prove that
∀ (p, pr) ∈ [[t₁; t₂] pr L] Γ T (p', pr') • pr' ⊆ pr

\[ [[t₁; t₂] pr L] Γ T (p', pr') = \infty/ (\infty/ [[t₂] pr L] Γ T) \ast \ast \infty/[[t₁] pr L] Γ T (p', pr') \] [Definition of ; ]

let \[ [[t₁] pr L] Γ T (p', pr') = list_A \]
By Assumption we have that
∀ (p'', pr'') ∈ list_A • pr'' ⊆ pr''
Now, let \[ \infty/ (\infty/ [[t₂] pr L] Γ T) \ast list_A = list_B \]
We have by Assumption and transitivity \( (pr' \subseteq pr'' \wedge pr'' \subseteq pr''' \Rightarrow pr' \subseteq pr''') \)
that ∀ (p'', pr''') ∈ list_B • pr'' ⊆ pr'''

We must also prove that
∀ (p, pr) ∈ [[t₁ | t₂] pr L] Γ T (p', pr') • pr' ⊆ pr

\[ [[t₁ | t₂] pr L] Γ T (p', pr') = \infty/[[t₁] pr L] Γ T, [[t₁] pr L] Γ T] \circ (p', pr') \]
= (\infty/[[t₁] pr L] Γ T (p', pr')) \ast \ast \infty/ [[t₂] pr L] Γ T (p', pr') \] [Definition of \( ; \) ]
If, by assumption,
∀ (p, pr) ∈ [[t₁] pr L] Γ T (p', pr') • pr' ⊆ pr and
∀ (p, pr) ∈ [[t₂] pr L] Γ T (p', pr') • pr' ⊆ pr
so we can say that
∀ (p, pr) ∈ [[t₁] pr L] Γ T (p', pr') \ast \ast \infty/ [[t₁] pr L] Γ T (p', pr') • pr' ⊆ pr

We must also prove that
∀ (p, pr) ∈ [[!t₁] pr L] Γ T (p', pr') • pr' ⊆ pr

\[ [[!t₁] pr L] Γ T (p', pr') = \text{head}'([[t₁] pr L] Γ T (p', pr')) \]
= [head([[t₁] pr L] Γ T (p', pr'))] \] [Definition of head']
If, by assumption, we have that
∀ (p, pr) ∈ [[t₁] pr L] Γ T (p', pr') • pr' ⊆ pr and
we also can say that
∀ (p, pr) ∈ [head([[t₁] pr L] Γ T (p', pr'))] • pr' ⊆ pr and
We must also prove that
\[ \forall (p, pr) \in [\square_1 \square_2] \Gamma L \Gamma T (p_1'; p_2', pr') \bullet pr' \subseteq pr \]

This structural combinator is defined as below
\[ \[\square_1 \square_2\] \Gamma L \Gamma T (p_1'; p_2', proofs) = \Omega_{sc} \circ (\Pi(<([\square_1] \Gamma L \Gamma T(p_1', proofs), ([\square_2] \Gamma L \Gamma T(p_2', proofs)>)\])\]

We know that
\[ \Pi < [\square_1] \Gamma L \Gamma T(p_1', proofs), [\square_2] \Gamma L \Gamma T(p_2', proofs) > = [<(p_1, pr_1), (p_2, pr_2) >
\[ | (p_1, pr_1) \leftarrow [\square_1] \Gamma L \Gamma T(p_1', proofs),
(p_2, pr_2) \leftarrow [\square_1] \Gamma L \Gamma T(p_2', proofs)] \]

[Definition of \(\Pi\)]

and that
\[ \Omega_{sc} \circ list_p = <(p_1; p_2, pr_1 \circ pr_2)
\[ | (p_1, pr_1) \leftarrow [\square_1] \Gamma L \Gamma T(p_1', proofs),
(p_2, pr_2) \leftarrow [\square_1] \Gamma L \Gamma T(p_2', proofs) > \]

[Definition of \(\Omega_{sc}\) and \(\circ\)]

By assumption we have that
\[ \forall (p_1, pr_1) \in [\square_1] \Gamma L \Gamma T (p_1', pr') \bullet pr' \subseteq pr_1 \]
and
\[ \forall (p_2, pr_2) \in [\square_2] \Gamma L \Gamma T (p_2', pr') \bullet pr' \subseteq pr_2 \]

So we can say that \(\forall (p_1; p_2, pr_1 \circ pr_2) \in list_\Omega \bullet pr' \subseteq pr_1 \circ pr_2\)

We must also prove that
\[ \forall (p, pr) \in [\square] \Gamma L \Gamma T (if \ GC \ fi, pr') \bullet pr' \subseteq pr \]
\[ = [\square] \Gamma L \Gamma T (if \ GC \ fi, pr')
\[ = \Omega_{if} (pr')
\[ (mkGC (brG GC)
\[ (\Pi(apply (([\square] \Gamma L \Gamma T) * tactics) (\Phi_r * (brP GC)))))) \]
[Definition of \(\square\) and \(\Pi\)]

\[ = (if \ pr')\]

\[ (mkGC (brG GC)
\[ (\Pi(apply (([\square] \Gamma L \Gamma T) * tactics) (\Phi_r * (brP GC)))))) \]
[Definition of \(\Omega_{if}\)]

\[ = [(if \ gc, pr' \cup nps)
\[ | (gc, nps) \leftarrow mkGC (brG GC)
\[ (\Pi(apply (([\square] \Gamma L \Gamma T) * tactics) (\Phi_r * (brP GC))))))] \]
[Definition of if]
We can see that
\[ \forall (p, pr) \in [(\text{if } gc \text{ fi}, pr' \cup nps) \mid (gc, nps) \leftarrow mkGC (brG GC) \quad (\Pi(apply (([_]_L \quad \Gamma_T) * tactics) (\Phi_r \ast (brP \quad GC)))]) \]
\[ \quad \bullet \quad pr' \subseteq pr' \cup nps \]

We must also prove that
\[ \forall (p, pr) \in \Omega_{do} (pr') \]
\[ \begin{align*}
& = (do \quad pr')^* \quad mkGC (brG GC) \\
& \quad (\Pi(apply (([_]_L \quad \Gamma_T) * tactics) (\Phi_r \ast (brP \quad GC)))) \quad \text{[Definition of do]} \]
\[ = \Omega_{do} (pr') \quad \text{[Definition of } \Omega_{do}] \]
\[ = [(do \quad gc \quad od, pr' \cup nps) \mid (gc, nps) \leftarrow mkGC (brG GC) \quad (\Pi(apply (([_]_L \quad \Gamma_T) * tactics) (\Phi_r \ast (brP \quad GC))))] \quad \text{[Definition of } \Omega_{do}] \]

We can see that
\[ \forall (p, pr) \in [(do \quad gc \quad od, pr' \cup nps) \mid (gc, nps) \leftarrow mkGC (brG GC) \quad (\Pi(apply (([_]_L \quad \Gamma_T) * tactics) (\Phi_r \ast (brP \quad GC))))] \]
\[ \quad \bullet \quad pr' \subseteq pr' \cup nps \]

We must also prove that
\[ \forall (p, pr) \in (\begin{align*}
& \{ \text{var} \quad t_1 \}_{L} \quad \Gamma_T (\{ \text{var} \quad d \ast p', pr') \ast pr' \subseteq pr \\
& = (\text{var} \quad d) \ast (\{t_1\}_{L} \quad \Gamma_T (p', pr')) \quad \text{[Definition of } \text{var} \quad \text{]} \]
\[ = (\begin{align*}
& \{ \text{var} \ast d \ast p', pr\}_{L} \quad \Gamma_T (p', pr')) \quad \text{[Definition of } \text{var}] \]

So, we can say that
\[ \forall (p, pr) \in (\begin{align*}
& \{ \text{var} \ast d \ast p', pr\}_{L} \quad \Gamma_T (p', pr')) \ast pr' \subseteq pr \quad \text{[Assumption]} \]

We must also prove that
∀(p, pr) ∈ [[\text{con} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{con} \ d \bullet p'], pr')] \bullet pr' ⊆ pr

[[\text{con} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{con} \ d \bullet p'], pr')
= (\text{cons} \ d) * ([[t_1] \Gamma_L \Gamma_T (p', pr'))
= ([[[\text{con} \ d \bullet p] , pr]) | (p, pr) ← (\{t_1\} \Gamma_L \Gamma_T (p', pr'))\]  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ [\text{Definition of con} \ ]

So, we can say that
∀(p, pr) ∈ [[[\text{con} \ d \bullet p] , pr]) | (p, pr) ← (\{t_1\} \Gamma_L \Gamma_T (p', pr'))\] \bullet pr' ⊆ pr  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ [\text{Assumption}]

We must also prove that
∀(p, pr) ∈ [[[\text{pmain} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{proc} \ p\text{name} = body \bullet p'], pr')] \bullet pr' ⊆ pr

[[\text{pmain} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{proc} \ p\text{name} = body \bullet p'], pr')
= (\text{proc}\ p\text{name} body) * ([\{t_1\} \Gamma_L \Gamma_T (p', pr'))
= [[[\text{proc} \ p\text{name} = body \bullet p] , pr]) | (p, pr) ← (\{t_1\} \Gamma_L \Gamma_T (p', pr'))\] \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ [\text{Definition of pmain} \ ]

So, we can say that
∀(p, pr) ∈ [[[\text{proc} \ p\text{name} = body \bullet p] , pr]) | (p, pr) ← (\{t_1\} \Gamma_L \Gamma_T (p', pr'))\] \bullet pr' ⊆ pr  \ \ \ \ \ \ \ [\text{Assumption}]

We must also prove that
∀(p, pr) ∈ [[[\text{pmainvariant} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{proc} \ p\text{name} = body \text{variant} \ vis e \bullet p'], pr')] \bullet pr' ⊆ pr

[[\text{pmainvariant} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{proc} \ p\text{name} = body \text{variant} \ vis e \bullet p'], pr')
= (\text{variant} \ p\text{name} body v e) * ([\{t_1\} \Gamma_L \Gamma_T (p', pr'))
= [[[\text{proc} \ p\text{name} = body \text{variant} \ vis e \bullet p] , pr]) | (p, pr) ← (\{t_1\} \Gamma_L \Gamma_T (p', pr'))\] \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ [\text{Definition of pmainvariant} \ ]

So, we can say that
∀(p, pr) ∈ [[[\text{proc} \ p\text{name} = body \text{variant} \ vis e \bullet p] , pr]) | (p, pr) ← (\{t_1\} \Gamma_L \Gamma_T (p', pr'))\] \bullet pr' ⊆ pr  \ \ \ \ \ \ [\text{Assumption}]

We must also prove that
∀(p, pr) ∈ [[[\text{pbody} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{proc} \ p\text{name} = body \bullet p'], pr')] \bullet pr' ⊆ pr

[[\text{pbody} \ t_1 \ []] \Gamma_L \Gamma_T ([[\text{proc} \ p\text{name} = body \bullet p'], pr')
= (\text{proc}\ p\text{name} p') * ([\{t_1\} \Gamma_L \Gamma_T (body, pr'))\] \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ [\text{Definition of pbody} \ ]
∀(p, pr) ∈ \[\[\\text{proc } pname = \text{body'} \cdot p'[\], pr]\]
| (body', pr) ← \([t_1][\Gamma_L \Gamma_T (p', pr')]\)] [Definition of \text{proc}] 

So, we can say that
∀(p, pr) ∈ \[\[\\text{proc } pname = \text{body'} \cdot p'[\], pr]\]
| (body', pr) ← \([t_1][\Gamma_L \Gamma_T (p', pr')]\)] • pr’’ [Assumption] 

We must also prove that
∀(p, pr) ∈ \[\[\text{bodyvariant} t_1 t_m \Gamma_L \Gamma_T (\[\[\\text{proc } pname = \text{body variant } \text{vis e } \cdot p'[\], pr]\]
• p'[\], pr') • pr'’ ⊆ pr

\[\text{bodyvariant} t_1 t_m \Gamma_L \Gamma_T (\[\[\text{proc } pname = \text{body variant } \text{vis e } \cdot p'[\], pr]\]
• (body', pr) ← \([t_1][\Gamma_L \Gamma_T (\text{body}, pr')]\)] [Definition of \text{bodyvariant}]

So, we can say that
∀(p, pr) ∈ \[\[\text{bodyvariant} t_1 t_m \Gamma_L \Gamma_T (\[\[\text{proc } pname = \text{body variant } \text{vis e } \cdot p'[\], pr]\]
• p'[\], pr') • pr'’ ⊆ pr

\[\text{bodyvariant} t_1 t_m \Gamma_L \Gamma_T (\[\[\text{proc } pname = \text{body variant } \text{vis e } \cdot p'[\], pr]\]
| (body', pr) ← \([t_1][\Gamma_L \Gamma_T (\text{body}, pr')]\)] [Definition of \text{bodyvariant}]

So, as we have already proved for sequential composition and for both structural combinators, we have proved already for this structural combinator.

We must also prove that
∀(p, pr) ∈ \[\[\text{mainvariantbody} t_1 t_m \Gamma_L \Gamma_T (\[\[\text{proc } pname = \text{body variant } \text{vis e } \cdot p'[\], pr]\]
• p'[\], pr') • pr'’ ⊆ pr

\[\text{mainvariantbody} t_1 t_m \Gamma_L \Gamma_T (\[\[\text{proc } pname = \text{body variant } \text{vis e } \cdot p'[\], pr]\]
| (body', pr) ← \([t_1][\Gamma_L \Gamma_T (\text{body}, pr')]\)] [Definition of \text{mainvariantbody}]

So, as we have already proved for sequential composition and for both structural combinators, we have proved already for this structural combinator.

We must also prove that
∀(p, pr) ∈ \[\[\text{val} t_1 \Gamma_L \Gamma_T ((\text{val } v : t \cdot p') \text{pars}, pr') • pr'’ ⊆ pr

\[\text{val} t_1 \Gamma_L \Gamma_T ((\text{val } v : t \cdot p') \text{pars}, pr') • pr'’ ⊆ pr

179
So, we can say that
\[
\forall (p, pr) \in \left[\left((\text{val} \ v \ t \ (pars), pr) \mid (p, pr) \leftarrow (\Gamma [\Gamma_L \Gamma_T (p', pr')])\right]\right] \bullet pr' \subseteq pr
\]
[Assumption]

We must also prove that
\[
\forall (p, pr) \in \left[\left((\text{val} \ v \ t \ (pars), pr) \mid (p, pr) \leftarrow (\Gamma [\Gamma_L \Gamma_T (p', pr')])\right]\right] \bullet pr' \subseteq pr
\]
[Definition of val]

So, we can say that
\[
\forall (p, pr) \in \left[\left((\text{res} \ v \ t \ (pars), pr) \mid (p, pr) \leftarrow (\Gamma [\Gamma_L \Gamma_T (p', pr')])\right]\right] \bullet pr' \subseteq pr
\]
[Definition of res]

We must also prove that
\[
\forall (p, pr) \in \left[\left((\text{res} \ v \ t \ (pars), pr) \mid (p, pr) \leftarrow (\Gamma [\Gamma_L \Gamma_T (p', pr')])\right]\right] \bullet pr' \subseteq pr
\]
[Definition of valres]

So, we can say that
\[
\forall (p, pr) \in \left[\left((\text{val-res} \ v \ t \ (pars), pr) \mid (p, pr) \leftarrow (\Gamma [\Gamma_L \Gamma_T (p', pr')])\right]\right] \bullet pr' \subseteq pr
\]
[Definition of parcommand]
∀(p, pr) ∈ [(pars • p), pr] | (p, pr) ← ([t1]Γ L Γ T (p′, pr′))] • pr′ ⊆ pr \text{[Assumption]}

We must also prove that
∀(p, pr) ∈ [[\textbf{tactic name}({\textit{argument}})*]]Γ L Γ T (p′, pr′) • pr′ ⊆ pr

\[
[[\textbf{tactic name}(args)]Γ L Γ T (p′, pr′) = 
\begin{align*}
\text{if } & \text{name }\in \text{dom }Γ T ∧ \text{args }\in \text{dom }Γ T \text{ then} \\
& Γ T \text{ tname args } (p′, pr′) \\
\text{ else} & []
\end{align*}
\]

We know that Γ T : name → {argument}* → Tactic.
So, let Γ T name arguments = t1
We have then
= [[t1]]Γ L Γ T (p′, pr′) We can say that
∀(p, pr) ∈ [[t1]]Γ L Γ T (p′, pr′) • pr′ ⊆ pr \text{[Assumption]}

We must also prove that
∀(p, pr) ∈ µ variable • tactic (p′, pr′) • pr′ ⊆ pr

We know, by definition that this is defined as the least upper bound of a set of tactics, and the least upper bound of a set of tactics is a tactic. Let t1 be this tactic. So, by assumption, we finish this proof.

We must also prove that
∀(p, pr) ∈ [[\textbf{applies to } program do t2]]Γ L Γ T (p′, pr′) • pr′ ⊆ pr

\[
[[\textbf{applies to } program do t2]]Γ L Γ T (p′, pr′) = 
\begin{align*}
[[\textbf{con} program • equals (program, pr′)]t2]Γ L Γ T (p′, pr′) & \text{[Definition of } \textbf{applies to } \textbf{ – do } \textbf{ –]} \\
= [[v \in \text{TERM} (equals (program, pr′); t2)(v)] Γ L Γ T (p′, pr′) & \text{[Definition of } \textbf{con}] \\
\end{align*}
\]

We can say that
∀(p, pr) ∈ [[t1]]Γ L Γ T (p′, pr′) • pr′ ⊆ pr \text{[Assumption]}

\[\Box\]
D.6.10 Lemma 15a

\[ \Pi < \text{lst}_a, \text{lst}_b > \approx \Pi < \text{lst}_a, \text{lst}_c > = \Pi < \text{lst}_a, \text{lst}_b \times \text{lst}_c > \]

\[ \Pi < \text{lst}_a, \text{lst}_b > \approx \Pi < \text{lst}_a, \text{lst}_c > \]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * \text{lst}_b >\]
\[ \approx \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * \text{lst}_c > \]  
[Definition of \( \Pi \)]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow (e2l * \text{lst}_b) \times (e2l * \text{lst}_c) > \]  
[Property of \( \times \)]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * (\text{lst}_b \times \text{lst}_c) > \]  
[Property of \( * \)]
\[ = \Pi < \text{lst}_a, \text{lst}_b \times \text{lst}_c > \]  
[Definition of \( \Pi \)]

\[ = \Pi < \text{lst}_a, \text{lst}_b > \times \Pi < \text{lst}_c > \]
[Definition of \( \times \)]
\[ = \Pi < \text{lst}_a, \text{lst}_b \times \text{lst}_c > \]  
[Definition of \( \Pi \)]

\[ = \Pi < \text{lst}_a, \text{lst}_c > \times \Pi < \text{lst}_b, \text{lst}_c > \]
[Definition of \( \times \)]
\[ = \Pi < \text{lst}_a, \text{lst}_b \times \text{lst}_c > \]  
[Definition of \( \Pi \)]

\[ \Pi < \text{lst}_a, \text{lst}_b > \approx \Pi < \text{lst}_a, \text{lst}_c > \]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * \text{lst}_b >\]
\[ \approx \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * \text{lst}_c > \]  
[Definition of \( \Pi \)]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * (\text{lst}_b \times \text{lst}_c) > \]  
[Property of \( \times \)]
\[ = \Pi < \text{lst}_a, \text{lst}_b \times \text{lst}_c > \]  
[Definition of \( \Pi \)]

D.6.11 Lemma 15b

\[ \Pi < \text{lst}_a, \text{lst}_c > \approx \Pi < \text{lst}_b, \text{lst}_c > = \Pi < \text{lst}_a \times \text{lst}_b, \text{lst}_c > \]

\[ \Pi < \text{lst}_a, \text{lst}_c > \approx \Pi < \text{lst}_b, \text{lst}_c > \]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * \text{lst}_c >\]
\[ \approx \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * \text{lst}_c > \]  
[Definition of \( \Pi \)]
\[ = \langle (a : as) | a \leftarrow \text{lst}_a, as \leftarrow e2l * (\text{lst}_b \times \text{lst}_c) > \]  
[Property of \( \times \)]
\[ = \Pi < \text{lst}_a \times \text{lst}_b, \text{lst}_c > \]  
[Definition of \( \Pi \)]

D.6.12 Lemma 16

\[ f * a \times f * b = f * (a \times b) \]

This lemma is proved by induction as follows:

Base Case: \( a = [ ] \)
\[ f * [ ] \times f * b \]
\[ = f * b \]  
[Definition of \( \times \)]
\[ = f * ([ ] \times b) \]  
[Definition of \( \times \)]

Base Case: \( a = \bot \)
\[ f * \bot \times f * b \]
\[ = \bot \times f * b \]  
[Definition of \( \times \)]
\[ = \bot \]  
[Definition of \( \times \)]
\[ = f * (\bot) \]  
[Definition of \( \times \)]
\[ = f * (\bot \times b) \]  
[Definition of \( \times \)]

Base Case: \( a = [x] \)
\[ f * [x] \times f * b \]
\( (f \ x)^\infty f^* b \) \hspace{1cm} [Definition of *]
\( = f^*(\{x\}^\infty b) \) \hspace{1cm} [Definition of *]

Inductive step:
\( fs : \text{finseq } X \)
\( is : \text{pfiseq } X \)

We assume that
\( f^* fs^\infty f^* b = f^*(a^\infty b) \)
\( f^* is^\infty f^* b = f^*(a^\infty b) \)

We must prove that
\( f^*(fs^\infty is)^\infty f^* b = f^*((fs^\infty is)^\infty b) \)

\[
\begin{align*}
    f^*(fs^\infty is)^\infty f^* b &= \{x : f^*(fs^\infty is) ; y : f^* b \bullet x^\land y\} & [\text{Definition of } \infty] \\
    &= \{x : f \bullet \{u : fs ; v : is \bullet u^\land v\} ; y : f^* b \bullet x^\land y\} & [\text{Definition of } \infty] \\
    &= \{x : f^* fs ; y : f^* b \bullet x^\land y\} \cup \{x : f^* is ; y : f^* b \bullet x^\land y\} & [\text{Set Comprehension}] \\
    &= f^* \{x : f^* fs ; y : b \bullet x^\land y\} \cup f^* \{x : f^* is ; y : b \bullet x^\land y\} & [\text{Set Comprehension}] \\
    &= f^* \{x : \{u : fs ; v : is \bullet u^\land v\} ; y : b \bullet x^\land y\} & [\text{Definition of } \infty] \\
    &= f^* ((fs^\infty is)^\infty b) & [\text{Set Comprehension}]
\end{align*}
\]

\( \square \)

D.6.13 Lemma 17

\( \text{head}'(f^* \text{lst}) = f^*(\text{head}' \text{lst}) \)

This lemma is proved by induction as seen below:

\( \text{head}'(f^* \text{lst}) = f^*(\text{head}' \text{lst}) \)

Base Case: \( \text{lst} = [ ] \)
\( \text{head}'(f^* [ ] ) = \text{head}'([ ] ) \) \hspace{1cm} [Definition of *]
\( = [ ] \) \hspace{1cm} [Definition of head']
\( = f^* [ ] \) \hspace{1cm} [Definition of *]
\( = f^*(\text{head}'([ ] )) \) \hspace{1cm} [Definition of head']

Base Case: \( \text{lst} = \perp \)
\( \text{head}'(f^* \perp ) = \text{head}'(\perp ) \) \hspace{1cm} [Definition of *]
\( = \perp \) \hspace{1cm} [Definition of head']
Base Case: \( \text{lst} = [x] \)

\[
\begin{align*}
\text{head}'(f \ast [x]) &= \text{head}'([f \ x]) \\
&= [f \ x] \\
&= f \ast [x] \\
&= f \ast (\text{head}'([x]))
\end{align*}
\]

Inductive step:

\( fs : \text{finseq} \ X \)

\( is : \text{pfiseq} \ X \)

We assume that

\[
\begin{align*}
\text{head}'(f \ast fs) &= f \ast (\text{head}' \ fs) \\
\text{head}'(f \ast is) &= f \ast (\text{head}' \ is)
\end{align*}
\]

We must prove that

\[
\text{head}'(f \ast (fs \triangleright is)) = f \ast (\text{head}' \ (fs \triangleright is))
\]

\[
\begin{align*}
\text{head}'(f \ast (fs \triangleright is)) &= \text{head}'(f \ast fs \triangleright is) \\
&= \text{head}'(f \ast fs) \\
&= [\text{head}(f \ast fs)] \\
&= [f \ \text{head}(fs)] \\
&= f \ast [\text{head}(fs)] \\
&= f \ast \text{head}'(fs) \\
&= f \ast (\text{head}' \ (fs \triangleright is))
\end{align*}
\]

\[\square\]

**D.6.14 Lemma 18**

\[\text{head}'(\Pi(\text{llst})) = \Pi(\text{head}' \circ \text{llst})\]

This lemma is also proved by induction as follows:

Base Case:

\( \text{llst} = <> \)

\[\text{head}'(\Pi(\text{llst}))\]
\[ = head'(\Pi(\langle\rangle)) \]
\[ = head'([\]) \quad \text{[Definition of} \ \Pi \text{]} \]
\[ = [\] \quad \text{[Definition of} \ head' \text{]} \]
\[ = \Pi(\langle\rangle) \quad \text{[Definition of} \ \Pi \text{]} \]
\[ = \Pi(head' \circ \langle\rangle) \quad \text{[Definition of} \ \odot \text{]} \]

Inductive step: We assume that \( head'(\Pi(ls)) = \Pi(head' \ast ls) \)
We must prove that \( head'(\Pi(l : ls)) = \Pi(head' \ast (l : ls)) \)

\[
\begin{align*}
head'(\Pi(l : ls)) &= head'([a : as \mid a \leftarrow l, as \leftarrow \Pi ls]) \quad \text{[Definition of} \ \Pi \text{]} \\
&= [a : as \mid a \leftarrow head' l, as \leftarrow \Pi(head' \circ ls)] \quad \text{[Property of List Comprehension]} \\
&= \Pi(head' l : head' \circ ls) \quad \text{[Definition of} \ \Pi \text{]} \\
&= \Pi(head' \circ l : ls) \quad \text{[Definition of} \ \circ \text{]} \\
\end{align*}
\]

\( \square \)

**D.6.15 Lemma 19**

\[ head'(head'(lst)) = head'(lst) \]

Case: \( lst = [\] 
\[ head'(head' [\]) = head'(\Pi(\langle\rangle)) \quad \text{[Definition of} \ head' \text{]} \]

Case: \( lst = \bot \) 
\[ head'(head' \bot) = head'(\bot) \quad \text{[Definition of} \ head' \text{]} \]

Case: \( lst = [x] \infty xs \) 
\[ head'(head' ([x] \infty xs)) = head'([x]) \quad \text{[Definition of} \ head' \text{]} \]

\( \square \)
D.6.16 Lemma 20

\[ \exists \cdot t \star \text{lst} \neq [] \Rightarrow \exists \cdot (\text{succs } t) \star \text{lst} \neq [] \]

\[ \exists \cdot t \star \text{lst} \neq [] \]
\[ \Rightarrow \exists r : \text{RMCell} \cdot \exists i : \mathbb{N} \cdot \text{lst}[i] = r \land t r \neq [] \]
\[ \Rightarrow \exists p : \text{pfseq(}\text{RMCell}\text{)} \cdot \exists r : \text{RMCell} \cdot \exists i : \mathbb{N} \cdot \text{snd}(p)[i] = r \land t r \neq [] \]
\[ \Rightarrow \text{succs } t r \neq [] \]
\[ \Rightarrow \exists \cdot (\text{succs } t) \star \text{lst} \neq [] \]

\(\square\)

D.6.17 Lemma 21

\[ \forall f, g : X \rightarrow \text{pfseq} X \]
\[ \exists [f, g] r = f r \bowtie g r \]

\[ \bowtie [f, g] r \]
\[ = \bowtie [f r, g r] \quad \text{[Definition of } ^{\bowtie} \text{]} \]
\[ = \bigcup_{\infty} \{ c : [f r, g r] \bowtie c \} \quad \text{[Definition of } \bowtie \text{]} \]
\[ = \bigcup_{\infty} \{ ^{\bowtie} (f, < f r, g r>) \} \quad \text{[Set theory]} \]
\[ = \bigcup_{\infty} \{ ^{\bowtie} (f, < f r>) \bowtie (^{\bowtie} (f, < g r>)) \} \quad \text{[Definition of } ^{\bowtie} \text{]} \]
\[ = \bigcup_{\infty} \{ f r \bowtie g r \} \quad \text{[Definition of } \bigcup_{\infty} \text{]} \]
\[ = f r \bowtie g r \quad \text{[Definition of } \bigcup \text{]} \]

\(\square\)

D.6.18 Lemma 22

\[ \forall a, b, c : \text{pfseq} X \cdot a ^{\bowtie} (b ^{\bowtie} c) = (a ^{\bowtie} b) ^{\bowtie} c \]

\[ a ^{\bowtie} (b ^{\bowtie} c) \]

Case \( a = (f, x), b = (f, y), c = (f, z) \)
\[ = (f, x) ^{\bowtie} ((f, y) ^{\bowtie} (f, z)) \quad \text{[Definition of } ^{\bowtie} \text{]} \]
\[ = (f, x) ^{\bowtie} (f, y \text{ cat } z) \quad \text{[Definition of } ^{\bowtie} \text{]} \]
\[ = (f, x \text{ cat } (y \text{ cat } z)) \quad \text{[Property of } \bowtie \text{]} \]
\[ = (f, x ^{\bowtie} y) ^{\bowtie} (f, z) \quad \text{[Definition of } ^{\bowtie} \text{]} \]
\[ ((f, x) \wedge (f, y)) \wedge (f, z) \]  
\[ \text{[Definition of } \wedge \text{]} \]

Case \(a = (f, x), b = (f, y), c = (p, z)\)
\[ = (f, x) \wedge ((f, y) \wedge (p, z)) \]  
\[ = (f, x) \wedge (p, y \text{ cat } z) \]  
\[ = (p, x \text{ cat } (y \text{ cat } z)) \]  
\[ = (p, (x \text{ cat } y) \text{ cat } z) \]  
\[ = (f, x) \wedge (p, z) \]  
\[ = ((f, x) \wedge (f, y)) \wedge (p, z) \]  
\[ \text{[Definition of } \wedge \text{]} \]

Case \(a = (f, x), b = (p, y), c = (x : \{f, p\}, z)\)
\[ = (f, x) \wedge ((p, y) \wedge c) \]  
\[ = (f, x) \wedge (p, y) \]  
\[ = (p, x \text{ cat } y) \]  
\[ = (p, x \text{ cat } y) \text{ cat } pf \ c \]  
\[ = (f, x) \wedge (p, z) \]  
\[ = ((f, x) \wedge (p, y)) \wedge c \]  
\[ \text{[Definition of } \wedge \text{]} \]

Case \(a = (p, x), b = (x : \{f, p\}, y), c = (y : \{f, p\}, z)\)
\[ = (p, x) \wedge (b \wedge c) \]  
\[ = (p, x) \]  
\[ = ((p, x) \wedge b) \wedge c \]  
\[ \text{[Definition of } \wedge \text{]} \]

\[ \text{D.6.19 Lemma 23} \]
\[ \forall e_1, e_2 : \text{pfseq } X \bullet \infty/ \bullet f \ast \bullet \infty/[e_1, e_2] = (\infty/ \bullet f \ast e_1) \infty/ (\infty/ \bullet f \ast e_2) \]

\[ \infty/ \bullet f \ast \bullet \infty/[e_1, e_2] \]
\[ = \infty/ \bullet f \ast \bigcup_{e_1, e_2} \{c : [e_1, e_2] \bullet \wedge c\} \]
\[ = \infty/ \bullet f \ast \bigcup_{e_1, e_2} \{\wedge (f, < e_1, e_2 >)\} \]
\[ = \infty/ \bullet f \ast \bigcup_{e_1, e_2} \{e_1 \infty/ e_2\} \]
\[ = \infty/ \bullet f \ast (e_1 \infty/ e_2) \}
\[ = \infty/ \{x : e_1 \infty/ e_2 \bullet p\text{map } f \ x\} \]  
\[ \text{[Definition of } \wedge \text{]} \]
\[ \text{[Set Theory]} \]
\[ \text{[Definition of } \wedge \text{]} \]
\[ \text{[Definition of } \bigcup_{\infty} \text{]} \]
\[ \text{[Definition of } \ast \text{]} \]
\[= \infty / \{x : \{a : e_1; b : e_2 \bullet a ^ {\uparrow} b\} \bullet \text{pfmap } f\} x\]

\[= \infty / \{x : \{a : e_1 \bullet \text{pfmap } f\ a\}; y : \{b : e_2 \bullet \text{pfmap } f\ b\} \bullet x \infty \ y\}\]

\[= \{x : \infty / \{a : e_1 \bullet \text{pfmap } f\ a\}; y : \infty / \{b : e_2 \bullet \text{pfmap } f\ b\} \bullet x \infty \ y\}\]

\[= \{x : \infty / \bullet f * e_1; y : \infty / \bullet f * e_2 \bullet x \infty \ y\}\]

\[= (\infty / \bullet f * e_1) \infty (\infty / \bullet f * e_2)\]

**D.6.20 Lemma 24**

\[\text{take } 1 l = l \text{ if } l = [\ ] \text{ or } (\text{size } l = 1 \land \forall x \in L \bullet \text{size } (\text{snd } x) = 1)\]

Case \(l = [\ ]\)

\[\text{take } 1 [\ ] = [\ ]\] [Definition of \text{take}]

Case \(l = [e]\)

\[\text{take } 1 [e] = [e]\] [Definition of \text{take}]

**D.6.21 Lemma 25**

\[(f * * \infty / a) \infty (f * \infty / b) = f * \infty / (a \infty b)\]

\[(f * \bullet \infty / a) \infty (f * \bullet \infty / b)\]

\[= (\infty / \bullet f * * a) \infty (\infty / \bullet f * * b)\] [Lemma D.6.2]

\[= \infty / (f * * a \infty f * * b)\] [Lemma D.6.5]

\[= \infty / \bullet f * * (a \infty b)\] [Lemma D.6.12]

\[= f * \bullet \infty / (a \infty b)\] [Lemma D.6.2]

**D.6.22 Lemma 26**

\[\text{head}(x \infty y) = \text{head}([\text{head} x] \infty y), x \neq [\ ]\]

This lemma is proved by induction as follows
Base Case: $x = \bot$
\[
\text{head}(\bot \infty y) = \text{head}([\text{head } \bot] \infty y) \quad \text{[Definition of head]}
\]

Base Case: $x = [e]$
\[
\text{head}([e] \infty y) = \text{head}([e]) = \text{head}([\text{head } [e]]) = \text{head}([\text{head } [e]] \infty y) \quad \text{[Property of head] [Definition of head]}
\]

Inductive step:
\[
fs : \text{finseq } X
is : \text{pfiseq } X
\]

We assume that
\[
\text{head}(fs \infty y) = \text{head}([\text{head } fs] \infty y), fs \neq []
\]
\[
\text{head}(is \infty y) = \text{head}([\text{head } is] \infty y), is \neq []
\]

We must prove that
\[
\text{head}(fs \infty is \infty y) = \text{head}([\text{head } fs \infty is] \infty y), fs \infty is \neq []
\]
\[
\text{head}(fs \infty is \infty y) = \text{head}(fs) = \text{head}([\text{head } fs]) = \text{head}([\text{head } fs \infty is]) = \text{head}([\text{head } fs \infty is] \infty y) \quad \text{[Property of head] [Lemma D.6.15] [Property of head]}
\]

$\square$

D.6.23 Lemma 27

\[
\text{head}(x \infty y) = \text{head}(x \infty [\text{head } y]), y \neq []
\]

This lemma is proved by induction as follows

Base Case: $y = \bot$
\[
\text{head}(x \infty \bot) = \text{head}(x \infty [\text{head } \bot]) \quad \text{[Definition of head]}
\]

Base Case: $y = [e]$
\[
\text{head}(x \infty [e]) = \text{head}([e]) \quad \text{[Property of head]}
\]

Case $x = []$
\[
= \text{head}([e]) \quad \text{[Property of head]}
\]
= head([head [e]])          [Definition of head]
= head(x ⊇ [head [e]])      [Property of head]
Case x = ⊥
= head(⊥)                   [Property of head]
= head(⊥ ⊇ [head [e]])      [Property of head]

Inductive step on x:
fs : finseq X
is : pfiseq X

We assume that
head(fs ⊇ y) = head(fs ⊇ [head y]), y ≠ []
head(is ⊇ y) = head(is ⊇ [head y]), y ≠ []

We must prove that
head((fs ⊇ is) ⊇ y) = head((fs ⊇ is) ⊇ [head y]), y ≠ []  [Property of head]

Inductive step on y:
gs : finseq X
hs : pfiseq X

We assume that
head(x ⊇ gs) = head(x ⊇ [head gs]), gs ≠ []
head(x ⊇ hs) = head(x ⊇ [head hs]), hs ≠ []

We must prove that
head(x ⊇ (gs ⊇ hs)) = head(x ⊇ [head(gs ⊇ hs)]), gs ≠ []

Case x = []
= head((gs ⊇ hs))          [Property of head]
= head([ ] ⊇ [head (gs ⊇ hs)])  [Property of head]
Case x = ⊥
= head(⊥)                   [Definition of ⊇/]
= head(⊥ ⊇ [head (gs ⊇ hs)])  [Definition of ⊇/]

Inductive step on x:
fs : finseq X
is : pfiseq X

We assume that
head(fs ⊇ (gs ⊇ hs)) = head(fs ⊇ [head(gs ⊇ hs)]), (gs ⊇ hs) ≠ []
head(is ⊇ (gs ⊇ hs)) = head(is ⊇ [head(gs ⊇ hs)]), (gs ⊇ hs) ≠ []
We must prove that
\[ \text{head}( (f \otimes s) \otimes (g \otimes h s)) = \text{head}( (f \otimes s) \otimes [\text{head}(g \otimes h s)]), (g \otimes h s) \neq [ ] \] [Property of head]

\[ \Box \]

D.6.24 Lemma 28
\[ l_1 \otimes (l_2 \otimes l_3) = (l_1 \otimes l_2) \otimes l_3 \]

\[ l_1 \otimes (l_2 \otimes l_3) \]
\[ = \{ x : l_1; y : l_2 \otimes l_3 \bullet x \land y \} \] [Definition of \( \otimes \)]
\[ = \{ x : l_1; y : \{ w : l_2; z : l_3 \bullet w \land z \} \bullet x \land y \} \] [Definition of \( \otimes \)]
\[ = \{ x : l_1; w : l_2; z : l_3 \bullet x \land (w \land z) \} \] [Set Comprehension]
\[ = \{ x : l_1; w : l_2; z : l_3 \bullet (x \land (w \land z)) \} \] [Set Comprehension]
\[ = \{ v : \{ x : l_1; w : l_2 \bullet (x \land w) \}; z : l_3 \bullet v \land z \} \] [Definition of \( \otimes \)]
\[ = \{ v : l_1 \otimes l_2; z : l_3 \bullet v \land z \} \] [Definition of \( \otimes \)]
\[ = (l_1 \otimes l_2) \otimes l_3 \] [Definition of \( \otimes \)]

\[ \Box \]
Appendix E

Gabriel’s Architecture

We present now some diagrams that describes Gabriel’s architecture. The whole document can be found in [26].
E.1 Class Diagrams

E.1.1 Integration Gabriel – Refine

This class diagram presents the main classes of Gabriel and Refine, and their integration.

Figure E.1: Integration Gabriel – Refine
E.1.2 Tactic’s Hierarchy

This class diagram presents Gabriel’s tactics hierarchy.

---

Figure E.2: Tactic’s Hierarchy
E.2 Sequence Diagrams

E.2.1 Tactic Generation

This sequence diagram describes the tactic generation in Gabriel.

Figure E.3: Tactic Generation
E.2.2 Tactic Application

This sequence diagram describes the tactic application in Gabriel.

Figure E.4: Tactic Application
Appendix F

Constructs of Gabriel

We present now the constructs of Gabriel for ArcAngel.
### F.1 ASCII ArcAngel

Table F.1 presents the constructs of ArcAngel written in a ASCII representation.

<table>
<thead>
<tr>
<th>ArcAngel</th>
<th>Gabriel</th>
</tr>
</thead>
<tbody>
<tr>
<td>law</td>
<td>law</td>
</tr>
<tr>
<td>tactic</td>
<td>tactic</td>
</tr>
<tr>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>abort</td>
<td>abort</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>!</td>
</tr>
<tr>
<td>succ</td>
<td>succ</td>
</tr>
<tr>
<td>fails</td>
<td>fails</td>
</tr>
<tr>
<td>;;</td>
<td>;;</td>
</tr>
<tr>
<td>if</td>
<td>if</td>
</tr>
<tr>
<td>fi</td>
<td>fi</td>
</tr>
<tr>
<td>do</td>
<td>do</td>
</tr>
<tr>
<td>od</td>
<td>od</td>
</tr>
<tr>
<td>var</td>
<td>var</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>con</td>
<td>con</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>pmain</td>
<td>pmain</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>pmainvariant</td>
<td>pmainvariant</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>pbody</td>
<td>pbody</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>pbodyvariant</td>
<td>pbodyvariant</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>pbodymain</td>
<td>pbodymain</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>pmainvariantbody</td>
<td>pmainvariantbody</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>parcommand</td>
<td>parcommand</td>
</tr>
<tr>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>applies to</td>
<td>applies to</td>
</tr>
<tr>
<td>do</td>
<td>do</td>
</tr>
</tbody>
</table>

Table F.1: ArcAngel’s constructs in Gabriel
## F.2 Laws Names and Templates

Tables F.2 and F.3 present the law names and their usage templates in Gabriel. The usage template defines the name of the laws and their arguments.

<table>
<thead>
<tr>
<th>Morgan’s Refinement Calculus</th>
<th>Gabriel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Law 1.1</td>
<td>strPost(PRED(newPost))</td>
</tr>
<tr>
<td>Law 1.2</td>
<td>weakPre(PRED(newPre))</td>
</tr>
<tr>
<td>Law 1.3</td>
<td>assign(IDS(lstVar), EXPS(lstVal))</td>
</tr>
<tr>
<td>Law 1.7</td>
<td>simpleEspec()</td>
</tr>
<tr>
<td>Law 1.8</td>
<td>absAssump()</td>
</tr>
<tr>
<td>Law 3.2</td>
<td>skipIntro()</td>
</tr>
<tr>
<td>Law 3.3</td>
<td>seqComp(PRED(mid))</td>
</tr>
<tr>
<td>Law 3.4</td>
<td>skipComp()</td>
</tr>
<tr>
<td>Law 3.5</td>
<td>fassign(IDS(lstVar), EXPS(lstVal))</td>
</tr>
<tr>
<td>Law 4.1</td>
<td>alt(PREDS(guards))</td>
</tr>
<tr>
<td>Law 4.3</td>
<td>altGuards(PREDS(guards))</td>
</tr>
<tr>
<td>Law 5.1</td>
<td>strPostIV(PRED(newPost))</td>
</tr>
<tr>
<td>Law 5.2</td>
<td>assigIV(IDS(lstVar), EXPS(lstVal))</td>
</tr>
<tr>
<td>Law 5.3</td>
<td>skipIntroIV()</td>
</tr>
<tr>
<td>Law 5.4</td>
<td>contractFrame(IDS(vars))</td>
</tr>
<tr>
<td>Law 5.5</td>
<td>iter(PREDS(guard), EXP(variant))</td>
</tr>
<tr>
<td>Law 6.1</td>
<td>varInt(DECS(newV : T))</td>
</tr>
<tr>
<td>Law 6.2</td>
<td>conInt(DECS(newC), PRED(pred))</td>
</tr>
<tr>
<td>Law 6.3</td>
<td>fixInitValue(DECS(newC), EXPS(exps))</td>
</tr>
<tr>
<td>Law 6.4</td>
<td>removeCon(IDS(cons))</td>
</tr>
<tr>
<td>Law 8.3</td>
<td>expandFrame(IDS(newVars))</td>
</tr>
<tr>
<td>Law 8.4</td>
<td>seqCompCon(PRED(mid), DECS(conDecs), IDS(lstVar))</td>
</tr>
<tr>
<td>Law B.2</td>
<td>seqCompIV(PRED(mid), IDS(lstVar))</td>
</tr>
</tbody>
</table>

Table F.2: Gabriel’s Laws Names
<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>ProcNoArgsIntro(^1) ((\text{EXP}(\text{name}), \text{PROG}(\text{body})))</td>
</tr>
<tr>
<td>11.2</td>
<td>ProcNoArgsCall()</td>
</tr>
<tr>
<td>11.3</td>
<td>ProcArgsIntro(^2) ((\text{EXP}(\text{name}), \text{PROG}(\text{body}), \text{PARAMS}(\text{resx} : \text{Z})))</td>
</tr>
<tr>
<td>11.4</td>
<td>ProcArgsCall()</td>
</tr>
<tr>
<td>11.5</td>
<td>CallByValue()</td>
</tr>
<tr>
<td>11.6</td>
<td>CallByValueIV()</td>
</tr>
<tr>
<td>11.7</td>
<td>CallByResult(^3) ((\text{DECS}(\text{newVar}), \text{IDS}(\text{subsVar})))</td>
</tr>
<tr>
<td>11.8</td>
<td>MultiArgs()</td>
</tr>
<tr>
<td>11.9</td>
<td>CallByValueResult()</td>
</tr>
<tr>
<td>13.1</td>
<td>VariantIntro(^4) ((\text{EXP}(\text{name}), \text{PROG}(\text{body}), \text{IDS}(\text{varName}), \text{EXP}(\text{varExp}), \text{PARAMS}(\text{resx} : \text{Z})))</td>
</tr>
<tr>
<td>13.2</td>
<td>ProcVariantBlockCall()</td>
</tr>
<tr>
<td>13.3</td>
<td>RecursiveCall()</td>
</tr>
<tr>
<td>13.4</td>
<td>VariantNoArgsIntro(^5) ((\text{EXP}(\text{name}), \text{PROG}(\text{body}), \text{IDS}(\text{varName}), \text{EXP}(\text{varExp})))</td>
</tr>
<tr>
<td>13.5</td>
<td>ProcVariantBlockCallNoArgs()</td>
</tr>
<tr>
<td>13.6</td>
<td>RecursiveCallNoArgs()</td>
</tr>
<tr>
<td>17.1</td>
<td>Coercion()</td>
</tr>
<tr>
<td>17.2</td>
<td>AbsCoercion()</td>
</tr>
<tr>
<td>17.3</td>
<td>IntCoercion(^6) ((\text{PRED}(\text{post})))</td>
</tr>
</tbody>
</table>

\(^1\) Table F.3: Continuation of Gabriel’s Laws Names

\(^2\) \(\text{EXP}\), \(\text{PROG}\), \(\text{PARAMS}\)

\(^3\) \(\text{DECS}\), \(\text{IDS}\)

\(^4\) \(\text{EXP}\), \(\text{PROG}\), \(\text{IDS}\), \(\text{PARAMS}\)

\(^5\) \(\text{EXP}\), \(\text{PROG}\), \(\text{IDS}\)

\(^6\) \(\text{PRED}\)
F.3 Argument Types

Table F.4 presents argument types in Gabriel.

<table>
<thead>
<tr>
<th>Type Description</th>
<th>Gabriel’s Representation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td>PRED(pred)</td>
<td>PRED(x &gt; 0)</td>
</tr>
<tr>
<td>Expression</td>
<td>EXP(exp)</td>
<td>EXP(x + 1)</td>
</tr>
<tr>
<td>Program</td>
<td>PROG(prog)</td>
<td>PROG(x : [x &gt;= 0, x = x + 1])</td>
</tr>
<tr>
<td>List of Declarations</td>
<td>DECS(dec₁; ...; decₙ)</td>
<td>DECS(x : T; y : S)</td>
</tr>
<tr>
<td>List of Predicates</td>
<td>PREDs(pred₁, ..., predₙ)</td>
<td>PREDs(x &gt; 0, x &lt;= 0)</td>
</tr>
<tr>
<td>List of Expressions</td>
<td>EXPS(exp₁, ..., expₙ)</td>
<td>EXPS(x + 1, x - 1)</td>
</tr>
<tr>
<td>List of Identifiers</td>
<td>IDS(var₁, ..., varₙ)</td>
<td>IDS(x, y, z)</td>
</tr>
<tr>
<td>List of Parameters</td>
<td>PARAMS(par₁; ...; parₙ)</td>
<td>PARAMS(res a; val b)</td>
</tr>
</tbody>
</table>

Table F.4: Argument’s Types in Gabriel
Bibliography


