NUMMSQUARED 2006A0 EXPLAINED, 
INCLUDING A NEW WELL-FOUNDED FUNCTIONAL FOUNDATION FOR LOGIC, 
MATHEMATICS AND COMPUTER SCIENCE

by

Samuel Howse

Submitted in partial fulfillment of the 
requirements for the degree of 
Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
October 2006

© Copyright by Samuel Howse, 2006
The signature page goes here.
For the inspirational Dr. L. S. River, and Nummists everywhere.

# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LIST OF TABLES</td>
<td>xx</td>
</tr>
<tr>
<td></td>
<td>LIST OF FIGURES</td>
<td>xxi</td>
</tr>
<tr>
<td></td>
<td>ABSTRACT</td>
<td>xxii</td>
</tr>
<tr>
<td></td>
<td>ACKNOWLEDGMENTS</td>
<td>xxiii</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 2 NUMMSQUARED OVERVIEW AND COMPARISON</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>UNTYPED LAMBDA CALCULUS AND IMPROVEMENTS</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>SET THEORY, VON NEUMANN AND JONES</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>FUNDAMENTAL CONCEPTS</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>SMALL AND LARGE FUNCTIONS</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>WELL-FOUNDEDNESS AND COERCION</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>VARIABLE-FREE</td>
<td>9</td>
</tr>
<tr>
<td>2.7</td>
<td>REFLECTION</td>
<td>10</td>
</tr>
<tr>
<td>2.8</td>
<td>EQUALITY</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>NSGO</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 3 FORMAL AND INFORMAL</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 4 WHERE TO FIND THE FORMAL PART</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 5 NOTATION IN THE INFORMAL PART</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>CHAPTER 6 DATA IN THE INFORMAL PART</td>
<td>17</td>
</tr>
<tr>
<td>6.1</td>
<td>EQUALS</td>
<td>17</td>
</tr>
<tr>
<td>6.2</td>
<td>NULL</td>
<td>17</td>
</tr>
<tr>
<td>6.3</td>
<td>BOOLEANS</td>
<td>17</td>
</tr>
</tbody>
</table>
7.19 SOME COMPUTATIONAL LARGE FUNCTION EXTENSIONS 67

7.20 SOME COMPUTATIONAL COMBINATIONS OF LARGE FUNCTION EXTENSIONS 68

7.21 SOME NON-COMPUTATIONAL LARGE FUNCTION EXTENSIONS AND COMBINATIONS 71

CHAPTER 8 NUMMSQUARED SYNTAX 73

8.1 NORMALIZED LARGE FUNCTIONS 74

8.2 EXTENSION AND TRUTH OF A NORMALIZED LARGE FUNCTION 78

8.3 REDUCTION: COMPUTED OF A NORMALIZED LARGE FUNCTION 79

8.4 NORMAL FORM OF A NATURAL NUMBER 84

8.5 QUOTED OF A NORMALIZED LARGE FUNCTION 84

8.6 UNQUOTED OF A NORMALIZED LARGE FUNCTION 86

8.7 MACRO EXPANDED 87

8.8 SUBSTITUTION AND SUBSTITUTION THEOREM 87

8.9 COMMENTS 88

8.10 IDENTIFIERS 88

8.11 LARGE FUNCTIONS 89

8.12 DEFINITIONS, DEFINITION LISTS, MODULES AND ABSTRACT PROGRAMS 99

8.13 CONTEXTS 102

8.14 NORMAL FORM OF A PRIMITIVE 104

8.15 NORMAL FORM OF A NORMALIZED CONSTANT 104

8.16 NORMAL FORM OF A GLOBAL NAME 104

8.17 PSEUDO-NUMMSQUARED 105

8.18 NORMAL FORM OF A LOCAL NAME 105

8.19 LOCAL TUPLE ACCESSOR CHECK 107

8.20 NORMAL FORM OF A COMPUTATIONAL NON-NORMALIZED CONSTANT OR COMPUTATIONAL COMBINATION 108

8.20.1 CONFIRMATION WITH NULL 108
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.20.2</td>
<td>NEGATION WITH NULL</td>
<td>109</td>
</tr>
<tr>
<td>8.20.3</td>
<td>NULL TO ZERO</td>
<td>109</td>
</tr>
<tr>
<td>8.20.4</td>
<td>KIND PREDICATES</td>
<td>110</td>
</tr>
<tr>
<td>8.20.5</td>
<td>TREE PREDICATE</td>
<td>111</td>
</tr>
<tr>
<td>8.20.6</td>
<td>RESULT</td>
<td>114</td>
</tr>
<tr>
<td>8.20.7</td>
<td>RESTRICT</td>
<td>114</td>
</tr>
<tr>
<td>8.20.8</td>
<td>RESTRICT TO RANGE</td>
<td>114</td>
</tr>
<tr>
<td>8.20.9</td>
<td>NURO SET RESULT</td>
<td>115</td>
</tr>
<tr>
<td>8.20.10</td>
<td>TREE SET RESULT</td>
<td>115</td>
</tr>
<tr>
<td>8.20.11</td>
<td>DEPENDENT SUM RESULT</td>
<td>116</td>
</tr>
<tr>
<td>8.20.12</td>
<td>DEPENDENT PRODUCT RESULT</td>
<td>117</td>
</tr>
<tr>
<td>8.20.13</td>
<td>CURRY AUGMENTED</td>
<td>118</td>
</tr>
<tr>
<td>8.20.14</td>
<td>CURRY RESULT</td>
<td>119</td>
</tr>
<tr>
<td>8.20.15</td>
<td>RECURSION RIGHT-HAND-SIDE</td>
<td>119</td>
</tr>
<tr>
<td>8.20.16</td>
<td>NEGATION</td>
<td>120</td>
</tr>
<tr>
<td>8.20.17</td>
<td>IMPLICATION WITH NULL</td>
<td>120</td>
</tr>
<tr>
<td>8.20.18</td>
<td>IMPLICATION</td>
<td>121</td>
</tr>
<tr>
<td>8.21</td>
<td>NORMAL FORM OF A NON-COMPUTATIONAL NON-NORMALIZED</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>CONSTANT OR NON-COMPUTATIONAL COMBINATION</td>
<td></td>
</tr>
<tr>
<td>8.21.1</td>
<td>EXISTENTIAL QUANTIFICATION</td>
<td>122</td>
</tr>
<tr>
<td>8.21.2</td>
<td>UNIVERSAL QUANTIFICATION</td>
<td>122</td>
</tr>
<tr>
<td>8.21.3</td>
<td>UNARY UNIVERSAL QUANTIFICATION</td>
<td>123</td>
</tr>
<tr>
<td>8.21.4</td>
<td>SMALL UNIVERSAL QUANTIFICATION</td>
<td>123</td>
</tr>
<tr>
<td>8.21.5</td>
<td>EQUALS RIGHT-HAND-SIDE</td>
<td>124</td>
</tr>
<tr>
<td>8.21.6</td>
<td>NOT EQUALS</td>
<td>126</td>
</tr>
<tr>
<td>8.21.7</td>
<td>INDUCTIVE CASE</td>
<td>127</td>
</tr>
<tr>
<td>8.22</td>
<td>NORMAL FORM AND VALIDITY OF A LARGE FUNCTION</td>
<td>128</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.23</td>
<td>NORMAL FORM AND VALIDITY OF A DEFINITION,</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>DEFINITION LIST OR ABSTRACT PROGRAM</td>
<td></td>
</tr>
<tr>
<td>8.24</td>
<td>PSEUDO-NUMMSQUARED COMPLETE</td>
<td>131</td>
</tr>
<tr>
<td>8.25</td>
<td>SOME TRUE LARGE FUNCTION EXTENSIONS</td>
<td>131</td>
</tr>
<tr>
<td>8.25.1</td>
<td>IDENTITY</td>
<td>131</td>
</tr>
<tr>
<td>8.25.2</td>
<td>NULL</td>
<td>132</td>
</tr>
<tr>
<td>8.25.3</td>
<td>ZERO</td>
<td>133</td>
</tr>
<tr>
<td>8.25.4</td>
<td>ONE</td>
<td>134</td>
</tr>
<tr>
<td>8.25.5</td>
<td>NULL SET</td>
<td>136</td>
</tr>
<tr>
<td>8.25.6</td>
<td>NURO SET</td>
<td>137</td>
</tr>
<tr>
<td>8.25.7</td>
<td>LEAF SET</td>
<td>138</td>
</tr>
<tr>
<td>8.25.8</td>
<td>TREE SET</td>
<td>140</td>
</tr>
<tr>
<td>8.25.9</td>
<td>LARGE COMPOSITION</td>
<td>141</td>
</tr>
<tr>
<td>8.25.10</td>
<td>SMALL COMPOSITION</td>
<td>142</td>
</tr>
<tr>
<td>8.25.11</td>
<td>PAIR</td>
<td>142</td>
</tr>
<tr>
<td>8.25.12</td>
<td>DEPENDENT SUM</td>
<td>144</td>
</tr>
<tr>
<td>8.25.13</td>
<td>DEPENDENT PRODUCT</td>
<td>145</td>
</tr>
<tr>
<td>8.25.14</td>
<td>CURRY</td>
<td>147</td>
</tr>
<tr>
<td>8.25.15</td>
<td>IF-THEN-ELSE</td>
<td>149</td>
</tr>
<tr>
<td>8.25.16</td>
<td>RECURSION</td>
<td>150</td>
</tr>
<tr>
<td>8.25.17</td>
<td>PROPOSITIONAL LOGIC</td>
<td>150</td>
</tr>
<tr>
<td>8.25.18</td>
<td>TRUTH</td>
<td>151</td>
</tr>
<tr>
<td>8.25.19</td>
<td>EQUALS</td>
<td>152</td>
</tr>
<tr>
<td>8.25.20</td>
<td>HILBERT</td>
<td>152</td>
</tr>
<tr>
<td>8.25.21</td>
<td>INDUCTION</td>
<td>153</td>
</tr>
<tr>
<td>8.25.22</td>
<td>LEFTOVERS</td>
<td>154</td>
</tr>
<tr>
<td>8.26</td>
<td>SOME INFERENCES FROM TRUE LARGE FUNCTION EXTENSIONS</td>
<td>156</td>
</tr>
<tr>
<td>8.26.1</td>
<td>MODUS PONENS</td>
<td>156</td>
</tr>
</tbody>
</table>
8.26.2 SPECIALIZATION ................................................. 156
8.27 SOME TRUE NORMALIZED LARGE FUNCTIONS ................. 157
8.28 SOME INFERENCES FROM TRUE NORMALIZED LARGE FUNCTIONS 157
  8.28.1 MODUS PONENTES ............................................. 157
  8.28.2 SPECIALIZATION .............................................. 158
  8.28.3 SUBSTITUTION ................................................ 158
8.29 PROOFS ............................................................... 158
8.30 PROPOSITION AND VALIDITY OF A PROOF, AND
   SOUNDNESS THEOREM .............................................. 159
8.31 QUOTED OF A PROOF ............................................... 161
8.32 PROOF UNQUOTED OF A NORMALIZED LARGE FUNCTION ....... 162
8.33 RUSSELL’S PARADOX AVERTED ..................................... 162

CHAPTER 9 THE FORMAL PART ........................................ 164
9.1 PREFACE TO THE FORMAL PART .................................. 164
  9.1.1 THE FORMAL PART ............................................... 164
  9.1.2 A QUICK SURVEY OF COQ ..................................... 164
    9.1.2.1 COQ TERMS, CONTEXTS, ENVIRONMENTS, TYPE-CHECKING,
              REDUCTION, NORMAL FORMS AND CONVERTIBILITY ............. 164
    9.1.2.2 COQ SORTS .................................................. 165
    9.1.2.3 COQ PROOFS ................................................ 166
    9.1.2.4 COQ DEPENDENT PRODUCTS,
              FUNCTIONS AND APPLICATIONS ............................. 166
    9.1.2.5 COQ TYPE CASTS .......................................... 166
    9.1.2.6 COQ MODULES, COMMANDS AND
              GLOBAL DECLARATIONS ..................................... 167
    9.1.2.7 NAMING OF COQ MODULES AND
              GLOBAL DECLARATIONS ..................................... 167
  9.1.3 NUMMSQUARED FORMALLY STYLE ................................ 167
9.1.3.1 MAKE DESIRED TYPES EXPLICIT USING TYPE CASTS .................................. 168
9.1.3.2 USE TYPE, NOT SET ................................................................. 168
9.1.3.3 MAKE REUSABLE TERMS INTO SEPARATE GLOBAL DECLARATIONS ............ 168
9.1.3.4 USE UNDERSCORE FOR HIERARCHICAL NAMING .................................. 168

9.2 FUNDAMENTALS: OPERATORS: MAIN ....................................................... 168

9.2.1 OPERATORS .......................................................... 169
9.2.2 THE CONSTANT OPERATOR ...................................................... 169
9.2.3 SIMPLE OPERATORS .......................................................... 169
9.2.4 THE IDENTITY SIMPLE OPERATOR ......................................... 169
9.2.5 BINARY OPERATORS ......................................................... 169
9.2.6 CONNECTIVE BINARY OPERATORS .............................................. 170
9.2.7 SIMPLE BINARY OPERATORS .................................................. 170
9.2.8 TRINARY OPERATORS ......................................................... 170
9.2.9 CONNECTIVE TRINARY OPERATORS ............................................ 170
9.2.10 SIMPLE TRINARY OPERATORS ................................................. 171
9.2.11 QUATERNARY OPERATORS ..................................................... 171
9.2.12 CONNECTIVE QUATERNARY OPERATORS ........................................ 171
9.2.13 SIMPLE QUATERNARY OPERATORS ........................................... 171
9.2.14 QUINARY OPERATORS .......................................................... 171
9.2.15 CONNECTIVE QUINARY OPERATORS ............................................ 172
9.2.16 SIMPLE QUINARY OPERATORS .................................................. 172

9.3 FUNDAMENTALS: PROPOSITIONS: MAIN ............................................... 172

9.3.1 DEPENDENCIES ....................................................... 173
9.3.2 PROPOSITIONAL PREDICATES ................................................... 173
9.3.3 THE CONSTANT PROPOSITIONAL PREDICATE ................................ 173
9.3.4 BINARY PROPOSITIONAL PREDICATES ......................................... 173
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.10.1</td>
<td>DEPENDENCIES</td>
<td>193</td>
</tr>
<tr>
<td>9.10.2</td>
<td>CHOICES</td>
<td>193</td>
</tr>
<tr>
<td>9.10.3</td>
<td>CHOICE RELATED TO</td>
<td>194</td>
</tr>
<tr>
<td>9.10.4</td>
<td>CHOICE RELATED TO, CONNECTIVE</td>
<td>194</td>
</tr>
<tr>
<td>9.10.5</td>
<td>CHOICE TO OPTIONAL</td>
<td>195</td>
</tr>
<tr>
<td>9.10.6</td>
<td>CHOICE MERGE</td>
<td>195</td>
</tr>
<tr>
<td>9.11</td>
<td>FUNDAMENTALS: PAIRS: MAIN</td>
<td>196</td>
</tr>
<tr>
<td>9.11.1</td>
<td>DEPENDENCIES</td>
<td>196</td>
</tr>
<tr>
<td>9.11.2</td>
<td>PAIRS</td>
<td>196</td>
</tr>
<tr>
<td>9.11.3</td>
<td>PAIR RELATED TO</td>
<td>197</td>
</tr>
<tr>
<td>9.11.4</td>
<td>PAIR RELATED TO, CONNECTIVE</td>
<td>197</td>
</tr>
<tr>
<td>9.11.5</td>
<td>TRIPLES</td>
<td>198</td>
</tr>
<tr>
<td>9.11.6</td>
<td>TRIPLE LEFT 0</td>
<td>198</td>
</tr>
<tr>
<td>9.11.7</td>
<td>TRIPLE RIGHT 0</td>
<td>198</td>
</tr>
<tr>
<td>9.11.8</td>
<td>TRIPLE LEFT 1</td>
<td>199</td>
</tr>
<tr>
<td>9.11.9</td>
<td>TRIPLE RIGHT 1</td>
<td>199</td>
</tr>
<tr>
<td>9.11.10</td>
<td>QUADRUPLES</td>
<td>199</td>
</tr>
<tr>
<td>9.11.11</td>
<td>QUADRUPLE LEFT 0</td>
<td>200</td>
</tr>
<tr>
<td>9.11.12</td>
<td>QUADRUPLE RIGHT 0</td>
<td>200</td>
</tr>
<tr>
<td>9.11.13</td>
<td>QUADRUPLE LEFT 1</td>
<td>201</td>
</tr>
<tr>
<td>9.11.14</td>
<td>QUADRUPLE LEFT 2</td>
<td>201</td>
</tr>
<tr>
<td>9.11.15</td>
<td>QUADRUPLE RIGHT 2</td>
<td>202</td>
</tr>
<tr>
<td>9.12</td>
<td>FUNDAMENTALS: LISTS: MAIN</td>
<td>202</td>
</tr>
<tr>
<td>9.12.1</td>
<td>DEPENDENCIES</td>
<td>202</td>
</tr>
<tr>
<td>9.12.2</td>
<td>LISTS</td>
<td>202</td>
</tr>
<tr>
<td>9.12.3</td>
<td>LIST NOTATION</td>
<td>203</td>
</tr>
<tr>
<td>9.12.4</td>
<td>LIST RELATED TO</td>
<td>203</td>
</tr>
<tr>
<td>9.12.5</td>
<td>LIST RELATED TO, CONNECTIVE</td>
<td>204</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.21.24</td>
<td>NUMMSQUARED NON-COMPUTATIONAL NORMALIZED CONSTANTS</td>
<td>249</td>
</tr>
<tr>
<td>9.21.25</td>
<td>NUMMSQUARED NORMALIZED CONSTANTS</td>
<td>249</td>
</tr>
<tr>
<td>9.21.26</td>
<td>NUMMSQUARED COMPUTATIONAL NON-NORMALIZED CONSTANTS</td>
<td>249</td>
</tr>
<tr>
<td>9.21.27</td>
<td>NUMMSQUARED NON-COMPUTATIONAL NON-NORMALIZED CONSTANTS</td>
<td>252</td>
</tr>
<tr>
<td>9.21.28</td>
<td>NUMMSQUARED NON-NORMALIZED CONSTANTS</td>
<td>253</td>
</tr>
<tr>
<td>9.21.29</td>
<td>NUMMSQUARED CONSTANTS</td>
<td>253</td>
</tr>
<tr>
<td>9.21.30</td>
<td>NUMMSQUARED LARGE FUNCTIONS</td>
<td>254</td>
</tr>
<tr>
<td>9.21.31</td>
<td>NUMMSQUARED LOCAL TUPLE ACCESSOR LISTS</td>
<td>265</td>
</tr>
<tr>
<td>9.21.32</td>
<td>NUMMSQUARED LOCAL CONTEXTS</td>
<td>265</td>
</tr>
<tr>
<td>9.21.33</td>
<td>NUMMSQUARED DEFINITIONS</td>
<td>266</td>
</tr>
<tr>
<td>9.21.34</td>
<td>NUMMSQUARED GLOBAL CONTEXTS</td>
<td>266</td>
</tr>
<tr>
<td>9.21.35</td>
<td>NUMMSQUARED MODULES</td>
<td>266</td>
</tr>
<tr>
<td>9.21.36</td>
<td>NUMMSQUARED ABSTRACT PROGRAMS</td>
<td>266</td>
</tr>
</tbody>
</table>

**CHAPTER 10 CONCLUSION** ....................................................... 267

**BIBLIOGRAPHY** ........................................................................... 269

**INDEX** ..................................................................................... 273
LIST OF TABLES
2.1

VON NEUMANN’S AXIOMATIZATION AND COMBINATORY LOGIC ROUGHLY COMPARED . . . . . . . . . . . . . . . . . . . . .

xx

7


LIST OF FIGURES

7.1 SMALL FUNCTION EXTENSIONS ................................................. 27
ABSTRACT

NummSquared Explained is the thesis version of the comprehensive formal document NummSquared 2006a0 Done Formally, which is available at http://nummist.com/poohbist/.

Set theory is the standard foundation for mathematics, but often does not include rules of reduction for function calls. Therefore, for computer science, the untyped lambda calculus or type theory is usually preferred. The untyped lambda calculus (and several improvements on it) make functions fundamental, but suffer from non-terminating reductions and have partially non-classical logics. Type theory is a good foundation for logic, mathematics and computer science, except that, by making both types and functions fundamental, it is more complex than either set theory or the untyped lambda calculus. This document proposes a new foundational formal language called NummSquared that makes only functions fundamental, while simultaneously ensuring that reduction terminates, having a classical logic, and attempting to follow set theory as much as possible. NummSquared builds on earlier works by John von Neumann in 1925 and Roger Bishop Jones in 1998 that have perhaps not received sufficient attention in computer science.

A soundness theorem for NummSquared is proved.

Usual set theory, the work of Jones, and NummSquared are all well-founded. NummSquared improves upon the works of von Neumann and Jones by having reduction and proof, by supporting computation and reflection, and by having an interpreter called NsGo (work in progress) so the language can be practically used. NummSquared is variable-free.

For enhanced reliability, NsGo is an F#/C# .NET assembly that is mostly automatically extracted from a program of the Coq proof assistant.

As a possible step toward making formal methods appealing to a wider audience, NummSquared minimizes constraints on the logician, mathematician or programmer. Because of coercion, there are no types, and functions are defined and called without proof, yet reduction terminates. NummSquared supports proofs as desired, but not required.
ACKNOWLEDGMENTS

Many thanks to Dr. Malcolm Heywood, my PhD supervisor at Dalhousie University, for unbounded good ideas, patience and support throughout the lengthy PhD process. Thanks to Dr. Peter Hitchcock for insights into software engineering and program correctness. Thanks to Dr. Anthony Cox for discussions about programming languages, and for suggesting many useful improvements to the thesis. Thanks to Dr. Paul Gilmore for discussions about his Intensional Type Theory, and for suggesting many useful improvements to the thesis. Thanks to Hugo Herbelin for discussions about Coq and type theory, and for suggesting many useful improvements to the thesis. My PhD work would not have been possible without funding from Dalhousie University, the Killam Memorial Scholarship and the National Research Council Canada.

Thanks to Jan, Bob, Joe and Dr. L. S. River for many helpful conversations and editing. Thanks to Joe for graphic design services. Thanks to Mopsy for lots of stuff. Thanks to Dame P. P. Paws for suggesting a clean approach to programming. Thanks to the Rt. Hon. Leo L. Lion, my manager at Poohbist Technology, for providing the necessary impetus to complete this work. Thanks to Miss Plasma Tigerlilly Zoya for helping me digest the literature.
CHAPTER 1

INTRODUCTION

The modern personal computer comes bundled with an impressive assortment of software, and much more software and content is available on the Web (often at no additional cost). For typical use, disk space for storing software and documents is practically unlimited. Powerful CPUs sit idle most of the time.

Unrestricted functionality comes at low initial monetary cost, but at a high cost in complexity and security. Most installed software has almost unrestricted access to all data and other software on the computer. Even for Web content that is not explicitly installed by the user, security loopholes are frequently exploited. And even trusted software may contain errors that interfere with other software, damage data, or impact system stability.

For the most part, programmers are aware of these issues, and want to write secure software that has minimal impact on the remainder of the system. Languages with memory safety and automatic memory management (such as C#, Java and OCaml - see [31, chapter 1], [14, chapter 1] and [24]) offer substantial improvements by preventing memory corruption and memory leak errors. As a result, the programmer may take the convenient view that memory is a safe place to store data, and be mostly correct in this view. However, in the imperative paradigm, side-effects can still result in memory contents changing unexpectedly. The functional paradigm eliminates side-effects, thus presenting a view of memory that is both safe and mathematically elegant.

A substantial part of the complexity and security problem is the view of the computer (aside from memory) that the operating system and language present to the programmer. The typical view is easily summarized in two words: global state.

Because all processes share access to a single file system, any one process must view the state of the file system as being almost completely indeterminate. (Two notable
exceptions are that the operating system preserves certain structural properties, and that files may be locked while a process is running.) Bad software reacts to file system non-determinism non-deterministically. Good software will at least handle the errors, but still cannot always provide the desired functionality.

Interprocess communication is another source of complexity and security problems, since typically any process can send a message to any other. In the physical world, much is possible because agents can act independently and interact freely. The digital world we have created is a reflection of the physical one, in both its endless possibilities, and its occasional descent into chaos.

This document does not suggest that the complexity of the modern personal computer is unnecessary. But it does propose a way in which much is possible with very simple and mathematically elegant tools.

Set theory is the standard foundation for mathematics, but often does not include rules of reduction for function calls. Therefore, for computer science, the untyped lambda calculus or type theory is usually preferred. The untyped lambda calculus (and several improvements on it) make functions fundamental, but suffer from non-terminating reductions and have partially non-classical logics. Type theory is a good foundation for logic, mathematics and computer science, except that, by making both types and functions fundamental, it is more complex than either set theory or the untyped lambda calculus. This document proposes a new foundational formal language called NummSquared that makes only functions fundamental, while simultaneously ensuring that reduction terminates, having a classical logic, and attempting to follow set theory as much as possible. NummSquared builds on earlier works by John von Neumann in 1925 ([40]) and Roger Bishop Jones in 1998 ([26]) that have perhaps not received sufficient attention in computer science.

A soundness theorem for NummSquared is proved.

Usual set theory, the work of Jones, and NummSquared are all well-founded. NummSquared improves upon the works of von Neumann and Jones by having reduction and proof, by supporting computation and reflection, and by having an interpreter called NsGo (work in progress) so the language can be practically used. NummSquared is variable-free.

For enhanced reliability, NsGo is an F#/C# .NET assembly that is mostly automatically extracted from a program of the Coq proof assistant. (See [8] and [32].)

As a possible step toward making formal methods appealing to a wider audience, NummSquared minimizes constraints on the logician, mathematician or programmer.
Because of coercion, there are no types, and functions are defined and called without proof, yet reduction terminates. NummSquared supports proofs as desired, but not required.

NummSquared aims to hide much complexity from the programmer. The programmer sees only mathematical functions, and proofs of their properties. Because a NummSquared program may include propositions, computations and proofs, it may serve as specification, implementation, and proof that implementation satisfies specification. Side-effects and global state, including the file system and processes, are not part of the NummSquared view. Such a simplified view is ideal for the computational and logical tasks that are the core of almost any software. Mixing global state manipulation with these tasks would obscure their essentially mathematical nature.

A NummSquared program may be a component of a larger software project. Other components can handle interaction with the global state, while delegating the computational and logical tasks to NummSquared programs. Because NummSquared has a simple variable-free syntax and is untyped, it is easy for other components to generate and process NummSquared programs.

Much has already been accomplished with formal methods. For example, Praxis’s SPARK language is a subset of Ada that enables formal reasoning, and has been used for major industrial projects (see [33]). And [13] used Coq to check a proof of the Four Colour Theorem. The goal of NummSquared is to provide a foundation that is particularly simple, since it is based on untyped functions. Future research will apply and adapt NummSquared to large software projects, with the hypothesis that its simplicity is an asset.
CHAPTER 2

NUMMSQUARED OVERVIEW AND COMPARISON

NummSquared is a formal language, and a new well-founded functional foundation for logic, mathematics and computer science. A language \( L \) is well-founded iff \( L \) includes a well-founded relation on all \( L \) objects.

NummSquared meets all of the following goals:

- Functions are the only fundamental concept. There are no side-effects or global state.

- Include reduction and ensure that it always terminates.

- Minimize constraints on the logician, mathematician or programmer. In particular, because of coercion, there are no types, and functions are defined and called without proof, yet reduction terminates. NummSquared coercion is (loosely) a generalization to higher order functions of coercion (type conversion) found in many programming languages.

- Proofs as desired, but not required. Because a NummSquared program may include propositions, computations and proofs, it may serve as specification, implementation, and proof that implementation satisfies specification.

The motivation behind these goals is the idea that formal methods is more appealing when the language is simple, when proofs do not get in the way, and when termination of reduction is nonetheless ensured. It seems that many mathematicians have little interest in types, and many programmers have little interest in proofs. (Logicians, due to their focus on foundations, are often interested in both.) Perhaps by removing
types and delaying proofs, NummSquared will be a step toward making formal methods appealing to a wider audience.

NummSquared has a classical logic. Also, NummSquared attempts to follow set theory as much as possible, since set theory is the standard foundation for mathematics.

A soundness theorem for NummSquared is proved.

NummSquared is variable-free.

NummSquared supports reflection for extending the syntax of the language, and for manipulating NummSquared functions and proofs.

NummSquared has an interpreter, NsGo (work in progress), so the language can be practically used. For enhanced reliability, NsGo is an F#/C# .NET assembly that is mostly automatically extracted from a program of the Coq proof assistant. (See [8] and [32].) NsGo (and hence NummSquared programs) inherit memory safety and automatic memory management from .NET.

NummSquared is now overviewed and compared to existing foundations.

2.1 UNTYPED LAMBDA CALCULUS AND IMPROVEMENTS

The untyped lambda calculus (see [6, section 2]) suffers from non-terminating reductions. Letting \( f \) be \((\lambda x. (x \ x))\), consider \((f \ f)\), which reduces to itself.

The untyped lambda calculus, when augmented by negation for use as a logic, suffers from Russell’s paradox. Letting \( R \) be \((\lambda x. (\neg (x \ x)))\), consider \((R \ R)\), which reduces to \((\neg (R \ R))\). Thus \((R \ R)\) cannot be either true or false - a contradiction (see [35, p.3]). Also, the untyped lambda calculus augmented by implication results in Curry’s paradox (see [35, p.17]).

Church invented the untyped lambda calculus in 1932 and, in response to the paradox, Church’s type theory in 1940 (see [35, p.4,8]). However, Russell discovered in 1902 his paradox in Frege’s predicate calculus (see [41, section 2] and [25]). Russell’s paradox exploits Frege’s course-of-values notation (which is somewhat similar to lambda notation), together with Frege’s Basic Law V and Rule of Substitution. Course-of-values notation, together with Basic Law V, create a distinct object for each function, but there are more functions than objects. Russell’s solution to the paradox in 1903 was Russell’s theory of types. In summary, Frege’s predicate calculus and Russell’s theory of types
can be seen as precursors to the untyped lambda calculus and Church's type theory, respectively.

An improvement on the untyped lambda calculus in [2, section 2.2] resolves Russell's paradox, but some propositions are neither true nor false.

Gilmore's NaDSyL (see [12, abstract, section 2.4]) resolves Russell's paradox, and furthermore formulas are either true or false. However, the set of formulas is undecidable, and no internal predicate corresponding to the set of formulas is demonstrated.

Grue's map theory (see [16, p.13-14, section 8.6, chapter 11]) is an improvement on the untyped lambda calculus that includes ZFC set theory, but excluded middle is false in general, although excluded middle is true in an important special case.

[21, section 2.2] defines a programming language that includes the untyped lambda terms and also set-theoretic functions. Untyped lambda terms can be restricted to set domains, and thus used as arguments to set-theoretic functions.

None of the above improvements on the untyped lambda calculus eliminate non-terminating reductions, and each, except Howe, has a logic that is partially non-classical. (In the case of NaDSyL non-classicality appears differently: as undecidability of the set of formulas. In the case of Howe, the programming language is not itself a logic, although it is used to give semantics to Nuprl.)

2.2 SET THEORY, VON NEUMANN AND JONES

Zermelo's solution to Russell's paradox in Frege's predicate calculus, with extensions by Fraenkel, resulted in ZF set theory, which builds up sets from existing sets (see [17, p.156-157,180-181]). ZF does not use types to avoid paradox. Instead, ZF replaces Frege's course-of-values notation with more restricted abstraction: the axiom of replacement. ZF plus the axiom of choice is called ZFC (see [36, p.84,132-133]). In ZF, because of the axiom of regularity, membership is a well-founded relation on ZF sets - see [36, p.21]. Thus ZF is well-founded.

The axiomatization of functions by von Neumann ([40]) is conceptually related to ZFC, and has been adapted by others into a set theory called von Neumann-Bernays-Gödel (NBG) - see [30, p.176]. Since set theory is the standard mathematical foundation, it is understandable that von Neumann's work was adapted into a set theory for purposes of comparison with other set theories. But computer science is primarily about computable functions, and many set theories, including ZFC and NBG, do not include rules of reduction for function calls, or even rules of reduction for set member-
(Sometimes it is argued that NBG is simpler than von Neumann's original work. Actually, neither is simpler: they address different conventions. In mathematics, the convention is set theory in first order logic; in computer science, the convention is a theory of functions.)

Even though von Neumann's axiomatization lacks rules of reduction, it is conceptually somewhat similar (see table 2.1) to combinatory logic (see [37, section 3]), which is closely related to the untyped lambda calculus. But, while von Neumann's axiomatization is a good foundation for logic and mathematics, combinatory logic and the untyped lambda calculus are not (because, when augmented by negation and excluded middle, they suffer from Russell's paradox; and augmenting by implication results in Curry's paradox). So it is interesting that the most popular foundations for computer science are the untyped lambda calculus, and untyped (but partially non-classical) and typed improvements on it which eliminate the paradoxes, rather than von Neumann's axiomatization which is more closely related to set theory in classical logic.

<table>
<thead>
<tr>
<th>von Neumann</th>
<th>combinatory logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiom II.1</td>
<td>I combinator</td>
</tr>
<tr>
<td>axiom II.2</td>
<td>K combinator</td>
</tr>
<tr>
<td>axiom II.6</td>
<td>S combinator</td>
</tr>
</tbody>
</table>

Table 2.1: Von Neumann's axiomatization and combinatory logic roughly compared

Jones proposed Pure Functions ([26] as an axiomatization of functions that is related to ZFC. Pure Functions is defined using the formal language HOL (augmented with ZFC). However, Pure Functions lacks rules of reduction.

Farmer ([10]) proposed “STMM: A Set Theory for Mechanized Mathematics”. STMM is based on NBG and, in STMM, sets, not functions, are fundamental. However, STMM does have lambda notation for functions, and notation for function calls.

2.3 FUNDAMENTAL CONCEPTS

Like the untyped lambda calculus (and improvements), type theory, von Neumann's axiomatization and Pure Functions, NummSquared makes functions fundamental. As in the untyped lambda calculus and Pure Functions, in NummSquared, functions are the only fundamental concept.

Unlike set or type theory, NummSquared does not make sets or types fundamental.
2.4 SMALL AND LARGE FUNCTIONS

In von Neumann's axiomatization, there is a particular object representing false. A function can itself be used as an argument iff the result of the function does not too often differ from false (see [40, p.397], which includes a more precise definition). False might be considered as the default result of the function, and the default cannot too often be overridden. The criterion for being used as an argument is not computable, which is problematic from a practical perspective.

In von Neumann's axiomatization there are also functions that cannot be used as an argument or result. In Pure Functions there are functions that are external functions (taking the form of HOL functions - see [26, “Functional Abstraction”]). An external function can be restricted to the domain of an internal function, in order to obtain an internal function.

Somewhat similarly to von Neumann and Pure Functions, NummSquared distinguishes small and large functions. Like von Neumann, both small and large functions are defined over all small functions, and they always return small functions.

In NummSquared, for simplicity, only large functions appear directly in NummSquared programs, which differs from von Neumann and Pure Functions.

In NummSquared, a large function ‘f can be Curried. The partial call to ‘f is a small function, and is restricted using the domain of a small function.

Neither von Neumann nor Jones attempt to make functions computable. NummSquared improves upon von Neumann's axiomatization and Pure Functions in several ways:

• NummSquared has reduction and proof. Because Pure Functions is defined within HOL, Jones applies HOL's proofs at the metalevel.

• In NummSquared, coercion is used to define small functions over all small functions, while maintaining computability. This generalized definition of result is the basis for reduction.

• NummSquared supports reflection.

• NummSquared has an interpreter, NsGo (work in progress), so the language can be practically used.
2.5 WELL-FOUNDEDNESS AND COERCION

As already mentioned, ZF is well-founded. So is NBG when the axiom of regularity is included - see [30, p.216]. In Pure Functions, membership in the field of a Pure Function is a well-founded relation on Pure Functions. Thus Pure Functions is well-founded.

An important subset of map theory (called the classical maps) is well-founded - see [15, p.18]. The range of a classical map is built up from existing classical maps. However, classical maps are defined over all maps, so the inductive hypothesis involves an interesting complexity metric in place of assumptions about elements of the domain.

NummSquared, unlike map theory, is well-founded in a similar way to Pure Functions: membership in the field of a NummSquared non-null small function is a well-founded relation on small functions. However, NummSquared small functions, like map theory classical maps, are defined over all small functions (in keeping with the goal of minimizing constraints). This is accomplished as follows: a NummSquared small function ‘f has a domain (a small sub-language of the language of all small functions), but coercion (which is computable) is used to define ‘f over all small functions, even those outside the domain of ‘f. NummSquared coercion is somewhat related to the restriction of untyped lambda terms to set domains in [21, section 2.2]. Observational Type Theory in [1, section 2.2] has explicit coercion requiring proof of type equality, whereas NummSquared coercion is automatic and does not require the programmer to supply proof.

The well-foundedness of NummSquared strengthens the connection between NummSquared and set theory.

2.6 VARIABLE-FREE

In NummSquared, a combination is a large function that combines one or more large functions (somewhat similar in concept to the functional forms of Backus's FP - see [5, section 11.1]). Like FP, NummSquared is variable-free. Combinations make variables unnecessary. (Of course, variable syntactic sugar for NummSquared would be possible.)

Function calls do not appear in NummSquared. Sometimes it is said that variable-free languages are difficult to read. Actually, it is mostly a question of the notation to which one is accustomed. Therefore, although NummSquared is variable-free, NummSquared large and small composition combinations are written, in the concrete syntax,
using lambda calculus function call notation. So NummSquared looks, in the concrete syntax, somewhat like the corresponding lambda calculus notation with the variables removed. Furthermore, NummSquared has local tuple accessors as a replacement for argument variables.

\section{Reflection}

Programmers often find it useful to extend the syntax of a language. Macro languages can provide such functionality, but a macro language often lacks the nice features of the language being extended. Therefore, a better solution is reflection: For a language \( L \), \( L \) supports \textbf{reflection} iff \( L \) programs can manipulate (to some extent) \( L \) programs.

As pointed out by [19, section 7], a language \( L \) with terminating reduction (such as NummSquared) cannot express the \( L \) interpreter. There are several ways of dealing with Hoare's incomputability result:

- Common usage of macro languages involves syntactic manipulations, meaning operations that do not require calling the \( L \) interpreter. Expressing in \( L \) macros performing syntactic manipulations does not require expressing in \( L \) the \( L \) interpreter.

- Partial reflection, as proposed by [20, p.2-3]: For some part of \( L \), it may be possible to express in \( L \) the interpreter for that part of \( L \). Clearly, the chosen part of \( L \) cannot express the interpreter for that part.

- It may be possible to express in \( L \) the bounded interpreter for \( L \), meaning the function identical to the \( L \) interpreter, except that it halts with an error if interpretation does not complete in a pre-specified number of steps.

Gilmore's ITT supports a very useful implicit quotation facility by allowing certain terms of a predicate type to have a secondary type: the type of subjects (see[11, p.xii,74]). Subject terms may be “mentioned”, but not “used” (called).

Even without reflection, NummSquared's large functions allow abstraction over all small functions. Therefore, reflection in NummSquared is directed at allowing abstraction over all large functions, without resorting to introducing super-large functions, etc.

NummSquared reflection works as follows: In NummSquared, quotation converts from a large function to a tree representation that can be manipulated by functions
(small and large), and unquotation is the inverse process. Unquotation cannot be used within small or large functions - a necessary restriction since unquotation is effectively the interpreter for large functions. That restriction does not prevent syntactic manipulations, thus NummSquared reflection partly eliminates the need for a macro language.

NummSquared quotation and unquotation have some conceptual similarities with Howe's partial reflection and Gilmore's implicit quotation (although NummSquared quotation is explicit). NummSquared reflection is greatly simplified by the fact that NummSquared is variable-free.

In logic, reflection is also useful: For a language 'L, 'L supports **logical reflection** iff 'L programs can manipulate 'L proofs. For example:

- Artemov’s Explicit Reflection Principle allows one to infer a formula from an internal proof of that formula (see [3, section 7]).
- Because Coq is typed, Coq proofs are Coq terms according to the Curry-Howard isomorphism (see [8, “Introduction”, section 4.1.1]).

NummSquared proof reflection works as follows: In NummSquared, all proofs are in a tree representation that can be manipulated by functions (small and large).

### 2.8 EQUALITY

A relation 'R on functions is an **extensional equality** iff, for any two functions 'f and 'g, 'R relates 'f and 'g iff the domains of 'f and 'g are equal, and the results of 'f and 'g (for any program of the common domain) are equal. An extensional equality equates functions that implement different algorithms (see [18, question 35]). Furthermore, an extensional equality is not computable. Therefore, an extensional equality is somewhat problematic in computer science. In von Neumann’s axiomatization and Pure Functions, equality is extensional.

In NummSquared, rule small functions are represented by rules, whereas simple small functions are represented by simpler means. NummSquared has equality, which is extensional on rule small functions. Equality cannot be used in reduction because it is not computable, but equality is essential in propositions. However, equality deeply excluding rule small functions is computable and can be used in reduction.

Gilmore's Intensional Type Theory (ITT) includes an appealing Rule of Intensionality stating that the intensions of two predicates are Leibniz equal iff their names are
Leibniz equal. Gilmore avoids Russell’s paradox by treating a predicate term as a name only when the predicate term has no free predicate variable. (See [11, p.xii,85-86].) The concept of the Rule of Intensionality is important for equality in computer science.

HiLog equality ([7, p.2-3]) is based on names, and is computable.

In future, NummSquared equality on rule small functions may be adapted to include some aspects of ITT and HiLog. At present, an extensional equality on rule small functions is chosen for logical and mathematical simplicity, despite the problems for computer science. An extensional equality on rule small functions strengthens the connection between NummSquared and set theory (for example, the axiom of extensionality in ZF - see [36, p.8]).

2.9 NSGO

A NummSquared program must somehow interact with other software, albeit indirectly. NsGo supports two methods of interaction:

- When run as a process, NsGo receives a purported NummSquared program on standard input, and produces either program output or error messages on standard output (depending on whether the purported program actually is a NummSquared program). NsGo also returns an exit code. When NsGo is run as a process, these are the only ways in which NsGo (and hence NummSquared programs) interact with other software. Severely restricting interaction with other software isolates NsGo from global state changes, and makes security much simpler. Since NsGo does not affect global state, recovering from a crash (for example, power failure) simply involves re-running NsGo.

- Alternatively, because NsGo is a .NET assembly, NsGo can be used as a library (and called in various ways) from within .NET programs.

Progress towards NsGo can be found in [22].
CHAPTER 3

FORMAL AND INFORMAL

A language is an unordered collection of things without duplicates. For a language \( 'L \), a program of \( 'L \) is a thing belonging to \( 'L \). For languages \( 'L_0 \) and \( 'L_1 \), \( 'L_0 = 'L_1 \) iff, for each thing \( 'x \), \( 'x \) is an \( 'L_0 \) program iff \( 'x \) is an \( 'L_1 \) program.

A language \( 'L \) is formal iff \( 'L \) is defined precisely. A language \( 'L \) is informal iff \( 'L \) is not formal. Mathematical English is an example of an informal language.

A document (such as the one you are reading) comprises programs of one or more languages. For a document \( 'd \), the formal part of \( 'd \) is that part of \( 'd \) comprising programs of formal languages; and the informal part of \( 'd \) is that part of \( 'd \) comprising programs of informal languages. Informal comments written within the formal part are considered to belong to the informal part, not the formal part.

Here are some uses for the formal and informal parts of a document:

- Some practical aspects are best expressed in the informal part. For example, the informal part of the document you are reading is now being used to discuss the roles of the formal and informal parts of documents in general.

- Although it is preferable to define ideas in the formal part, the informal part is still useful for explaining ideas, and for relating ideas in the formal part to existing ideas in the informal part.

- The informal part is sometimes useful for defining a new formal language and relating it to existing languages (formal and informal). However, with the availability of good existing formal languages, it is preferable to use the formal part to define a new formal language and relate it to existing formal languages, using the informal part only when necessary to relate a new formal language to existing informal languages.
The **formal part** and **informal part** are the formal and informal parts, respectively, of the document you are reading.
CHAPTER 4

WHERE TO FIND THE FORMAL PART

The document you are reading consists firstly of the informal part, including detailed definitions, theorems and proofs in mathematical English of the NummSquared metatheory. At the end of this document, NummSquared metatheory is expressed in the formal language Coq - this is currently a work in progress.
CHAPTER 5

NOTATION IN THE INFORMAL PART

Some notation is used in the informal part.
Where a phrase is defined, the phrase is written like this.
Text is given emphasis by writing it like this.
When quoting sources, the text is written “like this”, as with the following pearl from Dr. L. S. River:

“LSR ⊢ T = F → TOTAL CLUELESS”

Informal identifiers are words beginning with grave accent (‘). Informal identifiers are case-sensitive, and may include periods (.). Here are four distinct informal identifiers: ‘x, ‘X, ‘X₀ and ‘A.x. Informal identifiers are distinct from identifiers in the formal part, and from identifiers of some language being discussed.

A natural number is one of the things 0, 1, 2, ... (each distinct from the others). Let ‘Nat be the language of all natural numbers.

A Unicode code point (see [39, section 2.4]) is a natural number in the range 0-1114111. Let ‘Unicode be the language of all Unicode code points.

A single isolated character in fixed-width font (the font distinguishes it from other text) represents a Unicode code point. Example: H.

Two or more adjacent characters in fixed-width font represent a list (see below) of ‘Unicode. Example:

"Hello, world!"
CHAPTER 6

DATA IN THE INFORMAL PART

Various kinds of data are now defined for use in the informal part. The language of the informal part is intended to provide approximately the same capabilities as NBG set theory (see [30, p.176]).

6.1 EQUALS

Let 'x = 'y iff 'x and 'y are equal (equals must be defined for various kinds of data). Let x \neq y iff not x = y.

6.2 NULL

The thing 'null is introduced. 'null should be interpreted as the absence of relevant information, like the null pointer in many programming languages.

6.3 BOOLEANS

A Boolean is either 0 or 1, which should be interpreted as false or true, respectively. Let 'Boo be the language of all Booleans.

For a Boolean 'b, the negation of 'b, denoted by 'not('b), is 0 if 'b = 1; and 1 otherwise.

6.4 LANGUAGES

To avoid confusion between the informal part and some language being discussed, the term language is preferred to the more conventional term set.

For languages 'L0 and 'L1, 'L0 is a sub-language of 'L1 iff each 'L0 program is an 'L1 program.
The **empty language**, denoted by `Lang.empty`, is the language that has no programs.

For a language `L`, `L` is **empty** iff `L = Lang.empty`.

For a thing `x`, the **singleton** of `x`, denoted by `sing(x)`, is the language whose only program is `x`.

For languages `L0` and `L1`, the **intersection** of `L0` and `L1`, denoted by `intersect(L0, L1)`, is the language of all things `x` such that `x` is an `L0` program and an `L1` program.

For languages `L0` and `L1`, the **union** of `L0` and `L1`, denoted by `union(L0, L1)`, is the language of all things `x` such that `x` is an `L0` program or an `L1` program (or both).

### 6.5 MODELS

A **model** is a language `S`, together with a mapping from each `S` program to a particular thing. For a model `m`, the **source** of `m`, denoted by `src(m)`, is the language part of `m`. For a model `m`, and a `src(m)` program `x`, the **interpretation** by `m` of `x`, denoted by `m(x)`, is the unique thing `m` assigns to `x`. For models `m0` and `m1`, `m0 = m1` iff `src(m0) = src(m1)` and, for each `src(m0)` program `x`, `m0(x) = m1(x)`.

To avoid confusion between the informal part and some language being discussed, the term model is preferred to the more conventional term function.

For a model `m`, the **destination** of `m`, denoted by `des(m)`, is the language of all `m(x)` such that `x` is a `src(m)` program.

For a model `m` and a language `S`, `m` is **from** `S` iff `src(m) = S`.

For a model `m` and a language `D`, `m` is **to** `D` iff `des(m)` is a sub-language of `D`.

For a language `S`, and a thing `y`, the **constant model** from `S` to `y`, denoted by `constant(S, y)`, is the model `m` from `S` such that, for each `S` program `x`, `m(x) = y`.

For a language `S`, the **identity model** on `S`, denoted by `identity(S)`, is the model `m` from `S` such that, for each `S` program `x`, `m(x) = x`.

### 6.6 PAIRS AND TUPLES

A **pair** is an ordered collection of two things, possibly with duplicates. For a pair `p`, the **left** and **right** of `p` are thing one and thing two of `p`, respectively. For a pair `p`, let `left(p)` and `right(p)` be the left and right of `p`, respectively. For pairs `p0` and `p1`, `p0 = p1` iff `left(p0) = left(p1)` and `right(p0) = right(p1)`. For things `x0` and `x1`, let `<x0, x1>` be the pair `p` such that `left(p) = x0` and `right(p) = x1`.

Pairs are used to represent tuples (in a manner similar to [36, p.16]).
For a natural number \( m \geq 2 \), and a thing \( t \), the property of \( t \) being an \( m \) tuple is defined by recursion on \( m \):

- If \( m = 2 \): \( t \) is an \( m \) tuple iff \( t \) is a pair.

- If \( m > 2 \): \( t \) is an \( m \) tuple iff \( t \) is a pair and \( \text{left}(t) \) is an \( m - 1 \) tuple.

For a natural number \( m \geq 2 \), and things \( x_0, x_1, \ldots, x_{m-2}, x_{m-1} \), let \( <> x_0, x_1, \ldots, x_{m-2}, x_{m-1} <> \) be the \( m \) tuple \( <> x_0, x_1, \ldots, x_{m-2}, x_{m-1} <> \).

For a pair \( p = <l, r> \), let \( \text{flip}(p) \) be \( <r, l> \).

### 6.7 LISTS

Pairs are used to represent lists (in a manner similar to [29]).

Lists are defined inductively. A list is exactly one of the following:

- 0
- \( <h, r> \) where \( r \) is a list

A list \( l \) is empty iff \( l = 0 \). The empty list is represented by 0, not ‘null. The empty list is often interpreted differently than the absence of relevant information.

For a non-empty list \( <h, r> \), the head of \( l \), denoted by \( \text{head}(l) \), is \( h \); and the rest of \( l \), denoted by \( \text{rest}(l) \), is \( r \).

For a list \( l \), the length of \( l \), denoted by \( \text{len}(l) \), is defined by recursion on \( l \):

- 0 if \( l = 0 \)
- \( \text{len}(r) + 1 \) if \( l = <h, r> \)

For a natural number \( m \), and things \( x_0, x_1, \ldots, x_{m-1} \), let \( l << x_0, x_1, \ldots, x_{m-1} << >> \) be the length \( m \) list \( << x_0, x_1, \ldots, x_{m-1}, 0 >> >> \).

For a list \( l = l << x_0, x_1, \ldots, x_{m-1} >> \), an element of \( l \) is one of \( x_0, x_1, \ldots, x_{m-1} \).

For a language \( L \), and a list \( l \), \( l \) is of \( L \) iff each element of \( l \) is an \( L \) program.

For a non-empty list \( l = l << x_0, x_1, \ldots, x_{m-1} >> \), the tail of \( l \), denoted by \( \text{tail}(l) \), is \( x_{m-1} \); and the pretail of \( l \), denoted by \( \text{pretail}(l) \), is \( l << x_0, x_1, \ldots, x_{m-2} >> \).

For lists \( l0 = l << x_0, x_1, \ldots, x_{m-1} >> \) and \( l1 = l << y_0, y_1, \ldots, y_{n-1} >> \), the concatenation of \( l0 \) and \( l1 \), denoted by \( l0 + l1 \), is \( l << x_0, x_1, \ldots, x_{m-1}, y_0, y_1, \ldots, y_{n-1} >> \).
For a property \( \mathcal{P} \), and a list \( l = (x_0, x_1, \ldots, x_{m-1}) \), the \textbf{search} for \( \mathcal{P} \) in \( l \) is the list of those \( <0, x_0>, <1, x_1>, \ldots, <m-1, x_{m-1}> \) whose right satisfies \( \mathcal{P} \) (in order).

For a property \( \mathcal{P} \), and a list \( l = (x_0, x_1, \ldots, x_{m-1}) \), the \textbf{search first} for \( \mathcal{P} \) in \( l \) is the head of the search for \( \mathcal{P} \) in \( l \) if the search for \( \mathcal{P} \) in \( l \) is non-empty; and \textit{null} otherwise.

For a property \( \mathcal{P} \), and a list \( l = (x_0, x_1, \ldots, x_{m-1}) \), the \textbf{search first index} for \( \mathcal{P} \) in \( l \) is \textit{null} if the search first for \( \mathcal{P} \) in \( l \) is \textit{null}; and the left of the search first for \( \mathcal{P} \) in \( l \) otherwise.

For a property \( \mathcal{P} \), and a list \( l = (x_0, x_1, \ldots, x_{m-1}) \), the \textbf{search first data} for \( \mathcal{P} \) in \( l \) is \textit{null} if the search first for \( \mathcal{P} \) in \( l \) is \textit{null}; and the right of the search first for \( \mathcal{P} \) in \( l \) otherwise.

For a property \( \mathcal{P} \), and a list \( l = (x_0, x_1, \ldots, x_{m-1}) \), the \textbf{search length} for \( \mathcal{P} \) in \( l \) is the length of the search for \( \mathcal{P} \) in \( l \).

For a property \( \mathcal{P} \), and a list \( l = (x_0, x_1, \ldots, x_{m-1}) \), \( \mathcal{P} \) is \textit{duplicitous} in \( l \) iff the search length for \( \mathcal{P} \) in \( l \) is \( > 1 \).

### 6.8 WELL-FOUNDED RELATIONS

For a property \( \mathcal{P} \), and a relation \( < \) on \( \mathcal{P} \), \( < \) is \textbf{well-founded} iff there is no model \( x \) from \( \text{Nat} \) such that, for each natural number \( m \), \( \mathcal{P}(x(m)) \) and \( x(m+1) < x(m) \). (See \cite[section 3]{34} for a definition of a well-founded relation, and equivalent statements.)

### 6.9 SMALL LANGUAGES

For a language \( \mathcal{L} \), \( \mathcal{L} \) is \textbf{small} iff there exists some ZFC set \( s \) (see \cite[p.84,132-133]{36}) and some model \( m \) from \( s \) such that \( \mathcal{L} \) is a sub-language of \( \text{des}(s) \).
CHAPTER 7

NUMMSQUARED SEMANTICS

NummSquared semantics are now defined. The semantics are to be used for both reduction and truth. The portion of the semantics used for reduction is computable, allowing reduction to be defined directly as a computable total function. Defining reduction in this way automatically ensures termination.

NummSquared semantics are developed as follows:

• Small function extensions, the core of NummSquared, are defined.

• For coercion and computational reasons, the domain of a rule small function extension is represented by a domain extension. A domain extension contains the same information as a type in type theory, but with a different purpose.

• The domain extension irrelevance theorem: domain extensions contain no more information than their domains.

• Tagged small function extensions are obtained by augmenting (tagging) rule small function extensions with domain extensions (tags).

• The tag irrelevance theorem: because of the domain extension irrelevance theorem, tagging adds no information.

• NummSquared coercion is (loosely) a generalization to higher order functions of coercion (type conversion) found in many programming languages. Numm-Squared coercion is defined by well-founded tango.

• The coercion stability theorem: coercion does not make unnecessary changes.
Coercion is used to define tagged small function extensions over all tagged small function extensions, while maintaining computability. This generalized definition of result is the basis for reduction.

The extensionality theorem characterizes equals on rule tagged small function extensions.

Large function extensions, the face of NummSquared, are defined. Truth of a tagged small function extension or large function extension is defined.

Some computational large function extensions and combinations are given. Among them are Curry and recursion.

Some non-computational large function extensions and combinations are given. Among them are are equals and Hilbert.

7.1 SMALL FUNCTION EXTENSIONS

Even though small function extensions never appear directly in NummSquared programs, they are the core of NummSquared. (The word extension means an object of the semantics.)

A null small function extension is exactly the null small function extension, ‘Func.Sm.Ext.null. ‘Func.Sm.Ext.null should be interpreted as the absence of relevant information, like the null pointer in many programming languages. ‘Func.Sm.Ext.null should not be interpreted as 0, false, undefined nor non-termination (since NummSquared reduction always terminates). Map theory includes a somewhat similar nil element, although nil is interpreted as true and 0 (see [16, p.15,40,43]).

A zero small function extension contains a null small function extension. (Containment means structural containment. For example, a record contains its fields. The purpose of the containment is to enable structural recursion and induction.) Let ‘Func.Sm.Ext.zero be the zero small function extension containing ‘Func.Sm.Ext.null. For any zero small function extension ‘x, then ‘x = ‘Func.Sm.Ext.zero. ‘Func.Sm.Ext.zero should be interpreted as false.

A one small function extension contains <‘n, ‘z> where ‘n is a null small function extension and ‘z is a zero small function extension. Let ‘Func.Sm.Ext.one be the one
small function extension containing `<‘Func.Sm.Ext.null, ‘Func.Sm.Ext.zero>`. For any one small function extension ‘x, then ‘x = ‘Func.Sm.Ext.one. ‘Func.Sm.Ext.one should be interpreted as true.

A **leaf small function extension** is exactly one of the following:

- a null small function extension
- a zero small function extension
- a one small function extension

For leaf small function extensions ‘x and ‘y, ‘x = ‘y iff exactly one of the following holds:

- ‘x = ‘Func.Sm.Ext.one and ‘y = ‘Func.Sm.Ext.one.

Small function extensions are defined inductively. Let ‘Func.Sm.Ext be the language of all small function extensions.

A **small function extension** is exactly one of the following:

- a simple small function extension
- a rule small function extension

A **simple small function extension** is exactly one of the following:

- a leaf small function extension
- a pair small function extension

A **pair small function extension** contains `<‘n, ‘z, ‘o, ‘left, ‘right>` where:

- ‘n is a null small function extension
- ‘z is a zero small function extension
- ‘o is a one small function extension
- ‘left and ‘right are small function extensions
A rule small function extension contains a model 'model to 'Func.Sm.Ext such that 'src('model) is a small sub-language of 'Func.Sm.Ext.

This concludes the inductive definition.

For a pair small function extension 'p containing <'n, 'z, 'o, 'left, 'right>, the left and right of 'p are 'left and 'right, respectively. For a pair small function extension 'p, let 'left('p) and 'right('p) be the left and right of 'p, respectively. For pair small function extensions 'p0 and 'p1, 'p0 = 'p1 iff 'left('p0) = 'left('p1) and 'right('p0) = 'right('p1). For small function extensions 'x0 and 'x1, let {'x0, 'x1} be the pair small function extension 'p such that 'left('p) = 'x0 and 'right('p) = 'x1.

For a natural number 'm ≥ 2, and a small function extension 't, the property of 't being an 'm tuple is defined by recursion on 'm:

- If 'm = 2: 't is an 'm tuple iff 't is a pair small function extension.
- If 'm > 2: 't is an 'm tuple iff 't is a pair small function extension and 'left('t) is an 'm - 1 tuple.

For a natural number 'm ≥ 2, and small function extensions 'x0, 'x1, ..., 'x_m-2, 'x_m-1, let {'x0, 'x1, ..., 'x_m-2, 'x_m-1} be the 'm tuple [[[{'x0, 'x1}, ..., 'x_m-2}, 'x_m-1]. Let 'Func.Sm.Ext.Null be the language of all null small function extensions.

For a small function extension 'f, 'f is a nuro iff 'f = 'Func.Sm.Ext.null or 'f = 'Func.Sm.Ext.zero.

Let 'Func.Sm.Ext.Nuro be the language of all nuro small function extensions.

For a small function extension 'f, 'f is a Boolean iff 'f = 'Func.Sm.Ext.zero or 'f = 'Func.Sm.Ext.one.

Let 'Func.Sm.Ext.Boo be the language of all Boolean small function extensions.

Let 'Func.Sm.Ext.Leaf be the language of all leaf small function extensions.

For a small function extension 'f, the property of 'f being a tree is defined by recursion on 'f:

- If 'f is a leaf small function extension: 'f is a tree.
- If 'f is a pair small function extension: 'f is a tree iff 'left('f) and 'right('f) are trees.
- If 'f is a rule small function extension: 'f is not a tree.

Let 'Func.Sm.Ext.Tree be the language of all tree small function extensions.
7.2 DOMAIN AND SPECIFIC RESULT OF A SMALL FUNCTION EXTENSION

For a small function extension 'f, the domain of 'f (a small sub-language of 'Func.Sm.Ext), denoted by 'dom('f), is given by one of the following mutually exclusive cases:

- 'Func.Sm.Ext.Null if 'f = 'Func.Sm.Ext.null
- 'Func.Sm.Ext.Null if 'f = 'Func.Sm.Ext.zero
- 'Func.Sm.Ext.Nuro if 'f = 'Func.Sm.Ext.one
- 'Func.Sm.Ext.Leaf if 'f is a pair small function extension
- 'src('model) if 'f is a rule small function extension containing 'model

'Func.Sm.Ext.null is a 'dom('Func.Sm.Ext.null) program. Thus 'Func.Sm.Ext.null is a program of its own domain.

For a nuro small function extension 'x, 'dom('x) = 'Func.Sm.Ext.Null.

For a leaf small function extension 'x, 'dom('x) is a sub-language of 'Func.Sm.Ext.Nuro.

For a tree small function extension 't, 'dom('t) is a sub-language of 'Func.Sm.Ext.Leaf.

For a small function extension 'f, and a 'dom('f) program 'x, the specific result of 'f at 'x, denoted by 'f<\langle 'x\rangle>, is given by one of the following mutually exclusive cases:

- 'x if 'f is a leaf small function extension
- 'Func.Sm.Ext.null if 'f is a pair small function extension and 'x = 'Func.Sm.Ext.null
- 'left('f) if 'f is a pair small function extension and 'x = 'Func.Sm.Ext.zero
- 'right('f) if 'f is a pair small function extension and 'x = 'Func.Sm.Ext.one
- 'model('x) if 'f is a rule small function extension containing 'model

For a small function extension 'f, the range of 'f (a small sub-language of 'Func.Sm.Ext), denoted by 'ran('f), is the language of all 'f<\langle 'x\rangle> such that 'x is a 'dom('f) program.
\[ \text{ran('Func.Sm.Ext.null)} = \text{Func.Sm.Ext.Null.} \]
\[ \text{ran('Func.Sm.Ext.zero)} = \text{Func.Sm.Ext.Null.} \]
\[ \text{ran('Func.Sm.Ext.one)} = \text{Func.Sm.Ext.Nuro.} \]

For a leaf small function extension \( \langle x \rangle \), \( \text{ran('x)} = \text{dom('x)}. \)

For a pair small function extension \( \langle p \rangle \), \( \text{ran('p)} \) is the language whose only programs are \( \text{Func.Sm.Ext.null, 'left('p)} \) and \( \text{right('p)} \).

For a rule small function extension \( \langle r \rangle \) containing \( \langle \text{model} \rangle \), \( \text{ran('r)} = \text{des('model).} \)

For a nuro small function extension \( \langle x \rangle \), \( \text{ran('x)} = \text{Func.Sm.Ext.Null.} \)

For a leaf small function extension \( \langle x \rangle \), \( \text{ran('x)} \) is a sub-language of \( \text{Func.Sm.Ext.Nuro.} \)

For a tree small function extension \( \langle t \rangle \), \( \text{ran('t)} \) is a sub-language of \( \text{Func.Sm.Ext.Tree.} \)

For a small function extension \( \langle f \rangle \), the field of \( \langle f \rangle \) (a small sub-language of \( \text{Func.Sm.Ext.} \)), denoted by \( \text{field('f)} \), is \( \text{union('dom('f)}, \text{ran('f)} \).

For a small function extension \( \langle f \neq \text{Func.Sm.Ext.null.} \) and a \( \text{field('f)} \) program \( \langle x \rangle \), \( \langle x \rangle \) is structurally smaller than \( \langle f \rangle \).

For small function extensions \( \langle f \rangle \) and \( \langle g \rangle \), \( \langle f \rangle \) = \( \langle g \rangle \) iff exactly one of the following holds:

- \( \langle f \rangle = \text{Func.Sm.Ext.null and 'g = 'Func.Sm.Ext.null.} \)
- \( \langle f \rangle = \text{Func.Sm.Ext.zero and 'g = 'Func.Sm.Ext.zero.} \)
- \( \langle f \rangle = \text{Func.Sm.Ext.one and 'g = 'Func.Sm.Ext.one.} \)
- \( \langle f \rangle \) and \( \langle g \rangle \) are pair small function extensions, and \( \text{left('f)} = \text{left('g)} \) and \( \text{right('f)} = \text{right('g)} \).
- \( \langle f \rangle \) and \( \langle g \rangle \) are rule small function extensions, and \( \text{dom('f)} = \text{dom('g)} \), and, for each \( \text{dom('f)} \) program \( \langle x \rangle \), \( \text{f'<x>}' = \text{g'<x>'}. \)

The small function extensions are illustrated in figure 7.1.

### 7.3 RANK OF A SMALL FUNCTION EXTENSION

Some concepts from set theory are found useful at this point: ordinals; the well-founded relation < on ordinals; and the smallest ordinal satisfying a given property (see [36, p.36,39,45-46]). The following definition of rank of a small function extension is similar to the definition of rank of a set (see [36, p.79]).
Figure 7.1: Small function extensions
For a small function extension ‘f, the **rank** of ‘f (an ordinal), denoted by ‘rank(‘f), is defined by recursion on ‘f: ‘rank(‘f) is 0 if ‘f = ‘Func.Sm.Ext.null or ‘field(‘f) is empty; and the smallest ordinal ‘a such that, for each ‘field(‘f) program ‘x, ‘rank(‘x) < ‘a otherwise.

For a sub-language ‘A of ‘Func.Sm.Ext, the **rank** of ‘A, denoted by ‘rank(‘A), is 0 if ‘A is empty; and the smallest ordinal ‘a such that, for each ‘A program ‘x, ‘rank(‘x) < ‘a otherwise.

### 7.4 IDENTITY SMALL FUNCTION EXTENSIONS

For a **small** sub-language ‘A of ‘Func.Sm.Ext, the **identity small function extension** on ‘A, denoted by ‘Func.Sm.Ext.identity(‘A), is the rule small function extension ‘f such that ‘dom(‘f) = ‘A and, for each ‘dom(‘f) program ‘x, ‘f ‘x = ‘x.

For a small function extension ‘f, the **domain small function extension** of ‘f, denoted by ‘domFuncExt(‘f), is ‘Func.Sm.Ext.identity(‘dom(‘f)). (Pure Functions also uses identity functions to represent sets - see [26].)

For a small function extension ‘f, ‘domFuncExt(‘f) is a rule small function extension.

For a small function extension ‘f, ‘dom(‘domFuncExt(‘f)) = ‘dom(‘f).

**Proof.** ‘domFuncExt(‘f) = ‘Func.Sm.Ext.identity(‘dom(‘f)).

‘dom(‘Func.Sm.Ext.identity(‘dom(‘f))) = ‘dom(‘f).

For small function extensions ‘f and ‘g, ‘domFuncExt(‘f) = ‘domFuncExt(‘g) iff ‘dom(‘f) = ‘dom(‘g).

**Proof.**


- If ‘domFuncExt(‘f) = ‘domFuncExt(‘g): ‘dom(‘domFuncExt(‘f)) = ‘dom(‘f). ‘dom(‘domFuncExt(‘g)) = ‘dom(‘g).

For **rule** small function extensions ‘f and ‘g, ‘f = ‘g iff ‘domFuncExt(‘f) = ‘domFuncExt(‘g) and, for each ‘dom(‘f) program ‘x, ‘f ‘x = ‘g ‘x.

For a small function extension ‘f, ‘f is an **identity** iff, for each ‘dom(‘f) program ‘x, ‘f ‘x = ‘x.

For a **small** sub-language ‘A of ‘Func.Sm.Ext, ‘Func.Sm.Ext.identity(‘A) is an identity.

For a small function extension ‘f, ‘domFuncExt(‘f) is an identity.
For rule small function extensions \( f \) and \( g \), if \( f \) and \( g \) are identities, then \( f = g \) iff \( \text{dom}(f) = \text{dom}(g) \).

\textbf{Proof.}

- Holds if \( f = g \).
- If \( \text{dom}(f) = \text{dom}(g) \): For each \( \text{dom}(f) \) program \( x \), \( f\langle x \rangle = x = g\langle x \rangle \).

For a rule small function extension \( f \), if \( f \) is an identity, then \( f = \text{domFuncExt}(f) \).

\textbf{Proof.} \( \text{dom}(f) = \text{dom}(\text{domFuncExt}(f)) \). \( \text{domFuncExt}(f) \) is a rule small function extension and an identity.

For a small function extension \( f \), \( \text{domFuncExt}(\text{domFuncExt}(f)) = \text{domFuncExt}(f) \).

\textbf{Proof.} \( \text{domFuncExt}(f) \) is a rule small function extension and an identity.

\section*{7.5 DOMAIN EXTENSIONS}

Often it is useful for the domain of a small function extension to be a function space. But membership of an arbitrary small function extension in a function space is not computable. Type theory partially solves the problem using compile-time type checking, although the requirement that type checking be computable imposes additional constraints on the programmer. Another option is to require proofs at function calls, but this would contradict the NummSquared goal of proofs as desired, but not required. Instead, NummSquared uses runtime coercion to restrict a function to a function space. NummSquared coercion is (loosely) a generalization to higher order functions of coercion (type conversion) found in many programming languages.

For coercion and computational reasons, the domain of a rule small function extension is represented by a domain extension. Not every small sub-language of \( \text{Func.Sm.Ext} \) can be represented by a domain extension, so representation of domains by domain extensions imposes a constraint on domains. A domain extension contains the same information as a type in type theory, but with a different purpose. Types in type theory are used for compile-time type checking, which is not present in NummSquared. (Full compile-time, or even runtime, type checking for NummSquared would not be computable.) Domain extensions in NummSquared are available at runtime (as
with runtime type information in many programming languages), and are used for coercion. Domain extensions never appear directly in NummSquared programs, but are available to the programmer as small function extensions (thus maintaining functions as the only fundamental concept).

A constant domain extension is exactly one of the following:

- the null domain extension, ‘Dom.Ext.Null
- the nuro domain extension, ‘Dom.Ext.Nuro
- the leaf domain extension, ‘Dom.Ext.Leaf
- the tree domain extension, ‘Dom.Ext.Tree

‘Dom.Ext.Tree is somewhat related to the axiom of infinity in ZF (see [36, p.133]).

Domain extensions and domain extension families are defined mutually inductively. Let ‘Dom.Ext be the language of all domain extensions.

A domain extension is exactly one of the following:

- a constant domain extension
- a combination domain extension

A combination domain extension is exactly one of the following:

- a dependent sum domain extension
- a dependent product domain extension

A dependent sum domain extension contains a domain extension family. Dependent sums in type theory (see [8, section 3.1.4]) are conceptually similar. The axiom of unions in ZF (see [36, p.132]) is also somewhat related.

A dependent product domain extension contains a domain extension family. Dependent products in type theory (see [8, sections 4.1.3, 4.2]) are conceptually similar. The axiom of powers in ZF (see [36, p.132]) is also somewhat related.

A domain extension family contains <‘model, ‘tag> where:

- ‘model is a model to ‘Dom.Ext such that ‘src(‘model) is a small sub-language of ‘Func.Sm.Ext.
- ‘tag is a domain extension

This concludes the mutually inductive definition.
7.6 DOMAIN, DOMAIN EXTENSION AND SPECIFIC RESULT OF A DOMAIN EXTENSION FAMILY

For a domain extension family ‘F containing <‘model, ‘tag>, the domain of ‘F (a small sub-language of ‘Func.Sm.Ext), denoted by ‘dom(‘F), is ‘src(‘model).

For a domain extension family ‘F containing <‘model, ‘tag>, the domain extension of ‘F, denoted by ‘domExt(‘F), is ‘tag.

For a domain extension family ‘F containing <‘model, ‘tag>, and a ‘dom(‘F) program ‘x, the specific result of ‘F at ‘x, denoted by ‘F<‘x>, is ‘model(‘x).

For domain extension families ‘F and ‘G, ‘F = ‘G iff all the following hold:
- ‘dom(‘F) = ‘dom(‘G).
- For each ‘dom(‘F) program ‘x, ‘F<‘x> = ‘G<‘x>.

7.7 DOMAIN, RANK AND VALIDITY OF A DOMAIN EXTENSION

For a domain extension ‘A, the domain of ‘A (a small sub-language of ‘Func.Sm.Ext), denoted by ‘dom(‘A), is defined by recursion on ‘A:
- If ‘A is a dependent sum domain extension containing ‘F: ‘dom(‘A) is the language of ‘Func.Sm.Ext.null and all pair small function extensions ‘p such that ‘left(‘p) is a ‘dom(‘F) program, and ‘right(‘p) is a ‘dom(‘F<‘left(‘p)>) program.
- If ‘A is a dependent product domain extension containing ‘F: ‘dom(‘A) is the language of ‘Func.Sm.Ext.null and all rule small function extensions ‘f such that ‘dom(‘f) = ‘dom(‘F) and, for each ‘dom(‘f) program ‘x, ‘f<‘x> is a ‘dom(‘F<‘x>) program.
For a domain extension 'A, 'Func.Sm.Ext.null is a 'dom('A) program, and 'dom('A) is non-empty. Empty domains are excluded for reasons of coercion.

For a small sub-language 'A of 'Func.Sm.Ext, the null rule small function extension on 'A, denoted by 'Func.Sm.Ext.Rule.null('A), is the rule small function extension 'f such that 'dom('f) = 'A and, for each 'dom('f) program 'x, 'f<\langle x\rangle> = 'Func.Sm.Ext.null.

For a dependent product domain extension 'A containing 'F, 'Func.Sm.Ext.Rule.null('dom('F)) is a 'dom('A) program.

Proof. Let 'f = 'Func.Sm.Ext.Rule.null('dom('F)). 'dom('f) = 'dom('F). For each 'dom('f) program 'x, 'f<\langle x\rangle> = 'Func.Sm.Ext.null is a 'dom('F<\langle x\rangle>) program. 'f is a 'dom('A) program.

For a domain extension 'A, the rank of 'A, denoted by 'rank('A), is 'rank('dom('A)).

The definition of domain extensions and domain extension families is too broad because there is no constraint between 'model and 'tag of a domain extension family containing <'model, 'tag>.

For a domain extension 'A or a domain extension family 'F, the property of 'A or 'F (respectively) being valid is defined by mutual recursion on 'A or 'F (respectively).

For a domain extension 'A, the property of 'A being valid is given by one of the following mutually exclusive cases:

- If 'A is a constant domain extension: 'A is valid.

- If 'A is a dependent sum domain extension containing 'F: 'A is valid iff 'F is valid.

- If 'A is a dependent product domain extension containing 'F: 'A is valid iff 'F is valid.

A domain extension family 'F is valid iff all the following hold:

- For each 'dom('F) program 'x, 'F<\langle x\rangle> is valid.

- 'domExt('F) is valid.

- 'dom('domExt('F)) = 'dom('F).

This concludes the mutually recursive definition.

For valid domain extension families 'F and 'G, if 'domExt('F) = 'domExt('G), then 'dom('F) = 'dom('G).
Proof. ‘dom(F) = ‘dom(domExt(F)). ‘dom(G) = ‘dom(domExt(G)).

For valid domain extension families F and G, F = G iff domExt(F) = domExt(G) and, for each dom(F) program x, F<x> = G<x>.

Proof.
• Holds if F = G.

• If domExt(F) = domExt(G) and, for each dom(F) program x, F<x> = G<x>:
  ‘dom(F) = ‘dom(G).


For a valid dependent sum domain extension A, ‘Func.Sm.Ext.Pair.null is a ‘dom(A) program.

Proof. Let A contain F. ‘dom(F) = ‘dom(domExt(F)). ‘Func.Sm.Ext.null is a ‘dom(F) program. ‘Func.Sm.Ext.null is a ‘dom(<‘Func.Sm.Ext.null>) program.

7.8 DOMAIN EXTENSION IRRELEVANCE THEOREM

Domain extensions are computationally useful. However, domain extensions contain no more information than their domains - this domain extension irrelevance theorem strengthens the connection between NummSquared and set theory, and is now proved.

For constant domain extensions A and B, if dom(A) = dom(B), then A = B.

Proof. By cases on A and B.

For a constant domain extension A and a valid dependent sum domain extension B, ‘dom(A) ≠ ‘dom(B).

Proof.
• If A = ‘Dom.Ext.Null: ‘Func.Sm.Ext.Pair.null is a ‘dom(B) program, but not a ‘dom(A) program.

• If A ≠ ‘Dom.Ext.Null: ‘Func.Sm.Ext.zero is a ‘dom(A) program, but not a ‘dom(B) program.

For a constant domain extension A and a dependent product domain extension B, ‘dom(A) ≠ ‘dom(B).
Proof. Let ‘B contain ‘F. ‘Func.Sm.Ext.Rule.null(‘dom(‘F)) is a ‘dom(‘B) program, but not a ‘dom(‘A) program.

For a constant domain extension ‘A and a valid combination domain extension ‘B, ‘dom(‘A) ≠ ‘dom(‘B).


Proof. Let ‘B contain ‘F. ‘Func.Sm.Ext.Rule.null(‘dom(‘F)) is a ‘dom(‘B) program, but not a ‘dom(‘A) program.

For a dependent sum domain extension ‘A containing ‘F, and a small function extension ‘l, ‘l is a ‘dom(‘F) program iff there exists some small function extension ‘r such that {‘l, ‘r} is a ‘dom(‘A) program.

Proof.

• If there exists some small function extension ‘r such that {‘l, ‘r} is a ‘dom(‘A) program: ‘l is a ‘dom(‘F) program.

• If ‘l is a ‘dom(‘F) program: {‘l, ‘Func.Sm.Ext.null} is a ‘dom(‘A) program.

For a dependent sum domain extension ‘A containing ‘F, ‘rank(‘dom(‘F)) ≤ ‘rank(‘A).

For a dependent sum domain extension ‘A containing ‘F, a ‘dom(‘F) program ‘l, and a small function extension ‘r, then ‘r is a ‘dom(‘F<‘l>) program iff {‘l, ‘r} is a ‘dom(‘A) program.

Proof.

• If {‘l, ‘r} is a ‘dom(‘A) program: ‘r is a ‘dom(‘F<‘l>) program.

• If ‘r is a ‘dom(‘F<‘l>) program: {‘l, ‘r} is a ‘dom(‘A) program.


For dependent sum domain extensions ‘A containing ‘FA and ‘B containing ‘FB, if ‘dom(‘A) = ‘dom(‘B), then ‘dom(‘FA) = ‘dom(‘FB) and, for each ‘dom(‘FA) program ‘l, ‘dom(‘FA<‘l>) = ‘dom(‘FB<‘l>).

Proof.
For each small function extension \( 'l \): \( 'l \) is a \( \text{dom}(\text{FA}) \) program iff there exists some small function extension \( 'r \) such that \( \{ 'l, 'r \} \) is a \( \text{dom}(\text{A}) \) program. \( 'l \) is a \( \text{dom}(\text{FB}) \) program iff there exists some small function extension \( 'r \) such that \( \{ 'l, 'r \} \) is a \( \text{dom}(\text{B}) \) program. \( 'l \) is a \( \text{dom}(\text{FA}) \) program iff \( 'l \) is a \( \text{dom}(\text{FB}) \) program.

\( \text{dom}(\text{FA}) = \text{dom}(\text{FB}) \).

For each \( \text{dom}(\text{FA}) \) program \( 'l \), and each small function extension \( 'r \): \( 'r \) is a \( \text{dom}(\text{FA}< 'l >) \) program iff \( \{ 'l, 'r \} \) is a \( \text{dom}(\text{A}) \) program. \( 'r \) is a \( \text{dom}(\text{FB}< 'l >) \) program iff \( \{ 'l, 'r \} \) is a \( \text{dom}(\text{B}) \) program. \( 'r \) is a \( \text{dom}(\text{FA}< 'l >) \) program iff \( 'r \) is a \( \text{dom}(\text{FB}< 'l >) \) program.

For each \( \text{dom}(\text{FA}) \) program \( 'l \), \( \text{dom}(\text{FA}< 'l >) = \text{dom}(\text{FB}< 'l >) \).

For a dependent product domain extension \( 'A \) containing \( 'F \), and a small function extension \( 'x \), \( 'x \) is a \( \text{dom}(\text{F}) \) program iff there exists some rule small function extension \( 'f \) such that \( 'f \) is a \( \text{dom}(\text{A}) \) program and \( 'x \) is a \( \text{dom}( 'f ) \) program.

\textbf{Proof.}

- If there exists some rule small function extension \( 'f \) such that \( 'f \) is a \( \text{dom}(\text{A}) \) program and \( 'x \) is a \( \text{dom}( 'f ) \) program: \( \text{dom}( 'f ) = \text{dom}(\text{F}) \). \( 'x \) is a \( \text{dom}(\text{F}) \) program.
- If \( 'x \) is a \( \text{dom}(\text{F}) \) program: Let \( 'f = \text{Func.Sm.Ext.Rule.null}(\text{dom}(\text{F})) \). \( 'f \) is a rule small function extension and a \( \text{dom}(\text{A}) \) program. \( \text{dom}( 'f ) = \text{dom}(\text{F}) \). \( 'x \) is a \( \text{dom}( 'f ) \) program.

For a dependent product domain extension \( 'A \) containing \( 'F \), \( \text{rank}(\text{dom}(\text{F})) \leq \text{rank}(\text{A}) \).

For a dependent product domain extension \( 'A \) containing \( 'F \), a \( \text{dom}(\text{F}) \) program \( 'x \), and a small function extension \( 'y \), then \( 'y \) is a \( \text{dom}(\text{F}< 'x >) \) program iff there exists some rule small function extension \( 'f \) such that \( 'f \) is a \( \text{dom}(\text{A}) \) program and \( 'f< 'x > = 'y \).

\textbf{Proof.}

- If there exists some rule small function extension \( 'f \) such that \( 'f \) is a \( \text{dom}(\text{A}) \) program and \( 'f< 'x > = 'y \): \( 'y \) is a \( \text{dom}(\text{F}< 'x >) \) program.
- If \( 'y \) is a \( \text{dom}(\text{F}< 'x >) \) program: Let \( 'f \) be the rule small function extension such that \( \text{dom}( 'f ) = \text{dom}(\text{F}) \) and, for each \( \text{dom}( 'f ) \) program \( 'z \), \( 'f< 'z > = 'y \) if \( 'z = 'x \); and \( \text{Func.Sm.Ext.null} \) otherwise. \( 'f< 'x > = 'y \). \( 'f \) is a \( \text{dom}(\text{A}) \) program.
For a dependent product domain extension 'A containing 'F, and a 'dom('F) program 'x, 'rank('F<x>) ≤ 'rank('A).

For dependent product domain extensions 'A containing 'FA and 'B containing 'FB, if 'dom('A) = 'dom('B), then 'dom('FA) = 'dom('FB) and, for each 'dom('FA) program 'x, 'dom('FA<x>) = 'dom('FB<x>).

Proof.

• For each small function extension 'x: 'x is a 'dom('FA) program iff there exists some rule small function extension 'f such that 'f is a 'dom('A) program and 'x is a 'dom('f) program. 'x is a 'dom('FB) program iff there exists some rule small function extension 'f such that 'f is a 'dom('B) program and 'x is a 'dom('f) program. 'x is a 'dom('FA) program iff 'x is a 'dom('FB) program.

• 'dom('FA) = 'dom('FB).

• For each 'dom('FA) program 'x, and each small function extension 'y: 'y is a 'dom('FA<x>) program iff there exists some rule small function extension 'f such that 'f is a 'dom('A) program and 'f<x> = 'y. 'y is a 'dom('FB<x>) program iff there exists some rule small function extension 'f such that 'f is a 'dom('B) program and 'f<x> = 'y. 'y is a 'dom('FA<x>) program iff 'y is a 'dom('FB<x>) program.

• For each 'dom('FA) program 'x, 'dom('FA<x>) = 'dom('FB<x>).

The domain extension irrelevance theorem: For valid domain extensions 'A and 'B, if 'dom('A) = 'dom('B), then 'A = 'B.

Proof.

• By induction on 'A.

• Holds if 'A and 'B are constant domain extensions.

• If 'A is a constant domain extension and 'B is a combination domain extension, or vice versa: 'dom('A) ≠ 'dom('B), a contradiction.

• If 'A is a dependent sum domain extension and 'B is a dependent product domain extension, or vice versa: 'dom('A) ≠ 'dom('B), a contradiction.
• If 'A and 'B are dependent sum domain extensions: Let 'A contain 'FA. Let 'B contain 'FB. 'dom('FA) = 'dom('FB) and 'dom('domExt('FA)) = 'dom('domExt('FB)), and 'domExt('FA) = 'domExt('FB) (by inductive hypothesis). For each 'dom('FA) program 'l, 'dom('FA<l>) = 'dom('FB<l>), and 'FA<l> = 'FB<l> (by inductive hypothesis). 'FA = 'FB.

• If 'A and 'B are dependent product domain extensions: Let 'A contain 'FA. Let 'B contain 'FB. 'dom('FA) = 'dom('FB) and 'dom('domExt('FA)) = 'dom('domExt('FB)), and 'domExt('FA) = 'domExt('FB) (by inductive hypothesis). For each 'dom('FA) program 'x, 'dom('FA<x>) = 'dom('FB<x>), and 'FA<x> = 'FB<x> (by inductive hypothesis). 'FA = 'FB.

For valid domain extension families 'F and 'G, 'dom('F) = 'dom('G) iff 'domExt('F) = 'domExt('G).

Proof.

• Holds if 'domExt('F) = 'domExt('G).

• If 'dom('F) = 'dom('G): 'dom('domExt('F)) = 'dom('domExt('G)). 'domExt('F) = 'domExt('G) (by domain extension irrelevance theorem).

For valid domain extension families 'F and 'G, 'F = 'G iff 'dom('F) = 'dom('G) and, for each 'dom('F) program 'x, 'F<x> = 'G<x>.

Proof.

• Holds if 'F = 'G.

• If 'dom('F) = 'dom('G) and, for each 'dom('F) program 'x, 'F<x> = 'G<x>: 'domExt('F) = 'domExt('G).

7.9 DOMAIN EXTENSION INFERENCE

For a valid domain extension 'A, and a 'dom('A) program 'f, it is possible to infer certain type information about 'f.

For a simple small function extension 'f, the domain extension of 'f, denoted by 'domExt('f), is given by one of the following mutually exclusive cases:

• 'Dom.Ext.Null if 'f = 'Func.Sm.Ext.null
• ‘Dom.Ext.Null if \( f = \text{Func.Sm.Ext.zero} \)

• ‘Dom.Ext.Nuro if \( f = \text{Func.Sm.Ext.one} \)

• ‘Dom.Ext.Leaf if \( f \) is a pair small function extension

For a simple small function extension \( f \), ‘domExt(\( f \)) is valid.
For a simple small function extension \( f \), ‘dom(‘domExt(\( f \))) = ‘dom(\( f \)).

Proof. By cases on \( f \).

For a domain extension \( A \), and a rule small function extension \( f \), if \( f \) is a ‘dom(\( A \)) program, then \( A \) is a dependent product domain extension.

For a domain extension \( A \), and a ‘dom(\( A \)) program \( f \), the inferred domain extension in \( A \) of \( f \), denoted by ‘inferDomExt(\( A \), \( f \)), is given by one of the following mutually exclusive cases:

• ‘domExt(\( f \)) if \( f \) is a simple small function extension

• ‘domExt(\( F \)) if \( f \) is a rule small function extension, and \( A \) is the dependent product domain extension containing \( F \)

For a valid domain extension \( A \), and a ‘dom(\( A \)) program \( f \), ‘inferDomExt(\( A \), \( f \)) is valid.
For a valid domain extension \( A \), and a ‘dom(\( A \)) program \( f \), ‘dom(‘inferDomExt(\( A \), \( f \))) = ‘dom(\( f \)).

Proof.

• If \( f \) is a simple small function extension: ‘inferDomExt(\( A \), \( f \)) = ‘domExt(\( f \)). ‘dom(‘domExt(\( f \))) = ‘dom(\( f \)).

• If \( f \) is a rule small function extension, and \( A \) is the dependent product domain extension containing \( F \): ‘inferDomExt(\( A \), \( f \)) = ‘domExt(\( F \)). ‘dom(‘domExt(\( F \))) = ‘dom(\( F \)). ‘dom(\( f \)) = ‘dom(\( F \)). □

For a domain extension \( A \), and a ‘dom(\( A \)) program \( f \), and a ‘dom(\( A \)) program \( x \), the inferred domain extension in \( A \) of \( f \) at \( x \), denoted by ‘inferDomExt(\( A \), \( f \), \( x \)), is given by one of the following mutually exclusive cases:

• ‘Dom.Ext.Tree if \( A \) is a constant domain extension
• ‘Dom.Ext.Null if 'A is a dependent sum domain extension, and 'f = 'Func.Sm.Ext.null

• ‘Dom.Ext.Null if 'A is a dependent sum domain extension, 'f is a pair small function extension, and 'x = 'Func.Sm.Ext.null

•.domExt('F) if 'A is a dependent sum domain extension containing 'F, 'f is a pair small function extension, and 'x = 'Func.Sm.Ext.zero

• ‘F<left('f)> if 'A is a dependent sum domain extension containing 'F, 'f is a pair small function extension, and 'x = 'Func.Sm.Ext.one. Note that 'left('f) is a 'dom('F) program.

• ‘Dom.Ext.Null if 'A is a dependent product domain extension, and 'f = 'Func.Sm.Ext.null

• 'F<x> if 'A is a dependent product domain extension containing 'F, and 'f is a rule small function extension. Note that 'dom('f) = 'dom('F).

For a valid domain extension 'A, and a 'dom('A) program 'f, and a 'dom('f) program 'x, 'inferDomExt('A, 'f, 'x) is valid.

For a constant domain extension 'A, and a 'dom('A) program 'f, 'f is a tree.

Proof. By cases on 'A.

For a valid domain extension 'A, and a 'dom('A) program 'f, and a 'dom('f) program 'x, ‘f<‘x> is a ‘dom(‘inferDomExt('A, 'f, ‘x)) program.

Proof.

• If 'A is a constant domain extension: 'f is a tree. 'ran('f) is a sub-language of 'Func.Sm.Ext.Tree. ‘f<‘x> is a tree. ‘dom(‘inferDomExt('A, 'f, ‘x)) = 'Func.Sm.Ext.Tree.


• If 'A is a dependent sum domain extension, 'f is a pair small function extension, and 'x = 'Func.Sm.Ext.null: ‘f<‘x> = 'Func.Sm.Ext.null. ‘dom(‘inferDomExt('A, 'f, ‘x)) = 'Func.Sm.Ext.Null.
• If $A$ is a dependent sum domain extension containing $F$, $f$ is a pair small function extension, and $x = \text{Func.Sm.Ext.zero}$: $f(x) = \text{left}(f)$ is a $\text{dom}(F)$ program. $\text{dom}(\text{inferDomExt}(A, f, x)) = \text{dom}(\text{domExt}(F)) = \text{dom}(F)$.

• If $A$ is a dependent sum domain extension containing $F$, $f$ is a pair small function extension, and $x = \text{Func.Sm.Ext.one}$: $f(x) = \text{right}(f)$ is a $\text{dom}(F \langle \text{left}(f) \rangle)$ program. $\text{dom}(\text{inferDomExt}(A, f, x)) = \text{dom}(F \langle \text{left}(f) \rangle)$.

• If $A$ is a dependent product domain extension, and $f = \text{Func.Sm.Ext.null}$: $f(x) = \text{Func.Sm.Ext.null}$ is a $\text{dom}(F \langle x \rangle)$ program. $\text{dom}(\text{inferDomExt}(A, f, x)) = \text{Func.Sm.Ext.Null}$.

• If $A$ is a dependent product domain extension containing $F$, and $f$ is a rule small function extension: $f(x)$ is a $\text{dom}(F \langle x \rangle)$ program. $\text{dom}(\text{inferDomExt}(A, f, x)) = \text{dom}(F \langle x \rangle)$.

\[ \square \]

### 7.10 TAGGED SMALL FUNCTION EXTENSIONS

Tagged small function extensions are obtained by augmenting (tagging) rule small function extensions with domain extensions (tags). Not every rule small function extension can be tagged, so tagging imposes a constraint on small function extensions.

Tagged small function extensions are defined inductively. Let $\text{Func.Sm.Ext.Tagged}$ be the language of all tagged small function extensions.

A **tagged small function extension** is exactly one of the following:

- a simple tagged small function extension

- a rule tagged small function extension

A **simple tagged small function extension** is exactly one of the following:

- a leaf small function extension

- a pair tagged small function extension

A **pair tagged small function extension** contains $\langle n, z, o, \text{left}, \text{right} \rangle$ where:

- $n$ is a null small function extension

- $z$ is a zero small function extension
• ‘o is a one small function extension

• ‘left and ‘right are tagged small function extensions

A **rule tagged small function extension** contains <‘model, ‘tag> where:

• ‘model is a model to ‘Func.Sm.Ext.Tagged such that ‘src(‘model) is a **small** sub-language of ‘Func.Sm.Ext.

• ‘tag is a **valid** domain extension such that ‘dom(‘tag) = ‘src(‘model). Note that the programs of ‘src(‘model) are small function extensions, not tagged small function extensions.

This concludes the inductive definition.

For a pair tagged small function extension ‘p containing <‘n, ‘z, ‘o, ‘left, ‘right>, the **left** and **right** of ‘p are ‘left and ‘right, respectively. For a pair tagged small function extension ‘p, let ‘left(‘p) and ‘right(‘p) be the left and right of ‘p, respectively. For pair tagged small function extensions ‘p0 and ‘p1, ‘p0 = ‘p1 iff ‘left(‘p0) = ‘left(‘p1) and ‘right(‘p0) = ‘right(‘p1). For tagged small function extensions ‘x0 and ‘x1, let {‘x0, ‘x1} be the pair tagged small function extension ‘p such that ‘left(‘p) = ‘x0 and ‘right(‘p) = ‘x1.

For a natural number ‘m ≥ 2, and a tagged small function extension ‘t, the property of ‘t being an ‘m tuple is defined by recursion on ‘m:

• If ‘m = 2: ‘t is an ‘m tuple iff ‘t is a pair tagged small function extension.

• If ‘m > 2: ‘t is an ‘m tuple iff ‘t is a pair tagged small function extension and ‘left(‘t) is an ‘m - 1 tuple.

For a natural number ‘m ≥ 2, and tagged small function extensions ‘x0, ‘x1, ..., ‘xm-2, ‘xm-1, let {‘x0, ‘x1, ..., ‘xm-2, ‘xm-1} be the ‘m tuple { { {‘x0, ‘x1}, ..., ‘xm-2}, ‘xm-1 }.

### 7.11 UNTAGGED, TAG IRRELEVANCE THEOREM, TAGGED AND TAGGABLE

For a tagged small function extension ‘f, the **untagged** of ‘f (a small function extension), denoted by ‘untag(‘f), is defined by recursion on ‘f:

• ‘f if ‘f is a leaf small function extension
• \{\text{untag('left('f)), un\text{tag('right('f))}\} if 'f is a pair tagged small function extension

• If 'f is a rule tagged small function extension 'f containing <'model, 'tag>: 'untag('f) is the rule small function extension 'untagF such that '\text{dom('untagF)} = '\text{src('model)} and, for each '\text{dom('untagF)} program 'x, '\text{untagF<int('x)> = 'untag('model('x))}.

For a tagged small function extension 'f, all the following hold:

• 'untag('f) is a leaf small function extension iff 'f is a leaf small function extension

• 'untag('f) is a pair small function extension iff 'f is a pair tagged small function extension

• 'untag('f) is a rule small function extension iff 'f is a rule tagged small function extension

• 'untag('f) is a simple small function extension iff 'f is a simple tagged small function extension

• 'untag('f) = 'Func.Sm.Ext.null iff 'f = 'Func.Sm.Ext.null

• 'untag('f) = 'Func.Sm.Ext.zero iff 'f = 'Func.Sm.Ext.zero

• 'untag('f) = 'Func.Sm.Ext.one iff 'f = 'Func.Sm.Ext.one

• 'untag('f) is a nuro iff 'f is a nuro

• 'untag('f) is a Boolean iff 'f is a Boolean

Because of the domain extension irrelevance theorem, tagging adds no information. The **tag irrelevance theorem**: For tagged small function extensions 'f and 'g, if 'untag('f) = 'untag('g), then 'f = 'g.

*Proof.*

• By induction on 'f.

• If 'f and 'g are leaf small function extensions: 'untag('f) = 'f. 'untag('g) = 'g.
• If \( f \) and \( g \) are pair tagged small function extensions: \( \text{untag}(f) = \{\text{untag}(\text{left}(f)), \text{untag}(\text{right}(f))\} \). \( \text{untag}(g) = \{\text{untag}(\text{left}(g)), \text{untag}(\text{right}(g))\} \). \( \text{untag}(\text{left}(f)) = \text{untag}(\text{left}(g)) \). \( \text{untag}(\text{right}(f)) = \text{untag}(\text{right}(g)) \). \( \text{left}(f) = \text{left}(g) \) (by inductive hypothesis). \( \text{right}(f) = \text{right}(g) \) (by inductive hypothesis).

• If \( f \) and \( g \) are rule tagged small function extensions: Let \( f \) contain \(< \text{modelF}, \text{tagF}>\). Let \( g \) contain \(< \text{modelG}, \text{tagG}>\). \( \text{untag}(f) \) is the rule small function extension \( \text{untagF} \) such that \( \text{dom}(\text{untagF}) = \text{src}(\text{modelF}) \) and, for each \( \text{dom}(\text{untagF}) \) program \( x \), \( \text{untagF}<x> = \text{untag}(\text{modelF}(x)) \). \( \text{untag}(g) \) is the rule small function extension \( \text{untagG} \) such that \( \text{dom}(\text{untagG}) = \text{src}(\text{modelG}) \) and, for each \( \text{dom}(\text{untagG}) \) program \( x \), \( \text{untagG}<x> = \text{untag}(\text{modelG}(x)) \). \( \text{untagF} = \text{untagG} \). \( \text{dom}(\text{untagF}) = \text{dom}(\text{untagG}) \). \( \text{src}(\text{modelF}) = \text{src}(\text{modelG}) \). For each \( \text{src}(\text{modelF}) \) program \( x \), \( \text{untagF}<x> = \text{untagG}<x> \). \( \text{untag}(\text{modelF}(x)) = \text{untag}(\text{modelG}(x)) \), and \( \text{modelF}(x) = \text{modelG}(x) \) (by inductive hypothesis). \( \text{modelF} = \text{modelG} \). \( \text{dom}(\text{tagF}) = \text{dom}(\text{tagG}) \). \( \text{tagF} = \text{tagG} \) (by domain extension irrelevance theorem).

For a tagged small function extension \( f \), \( f \) is a **tree** iff \( \text{untag}(f) \) is a tree.

Let \( \text{Func.Sm.Ext.Tagged.Tree} \) be the language of all tree tagged small function extensions.

For a tagged small function extension \( f \), the property of \( f \) being a tree is given by one of the following mutually exclusive cases:

• If \( f \) is a leaf small function extension: \( f \) is a tree.

• If \( f \) is a pair tagged small function extension: \( f \) is a tree iff \( \text{left}(f) \) and \( \text{right}(f) \) are trees.

• If \( f \) is a rule tagged small function extension: \( f \) is not a tree.

**Proof.**

• If \( f \) is a leaf small function extension: \( \text{untag}(f) = f \) is a tree.

• If \( f \) is a pair tagged small function extension: \( \text{untag}(f) = \{\text{untag}(\text{left}(f)), \text{untag}(\text{right}(f))\} \). Let \( \text{untagF} = \text{untag}(f) \). \( \text{untagF} \) is a tree iff \( \text{left}(\text{untagF}) \) and \( \text{right}(\text{untagF}) \) are trees.

• If \( f \) is a rule tagged small function extension: \( \text{untag}(f) \) is a rule small function extension. \( \text{untag}(f) \) is not a tree.
For a valid domain extension ‘A, and a small function extension ‘f such that ‘f is a ‘dom(‘A) program, the tagged in ‘A of ‘f (a tagged small function extension), denoted by ‘tagged(‘A, ‘f), is defined by recursion on ‘f:

• ‘f if ‘f is a leaf small function extension


• If ‘f is a rule small function extension: ‘tagged(‘A, ‘f) is the rule tagged small function extension containing <’modelTagged, ‘tag> where:

For a valid domain extension ‘A, and a small function extension ‘f such that ‘f is a ‘dom(‘A) program, ‘untag(‘tagged(‘A, ‘f)) = ‘f.

**Proof.**

• By induction on ‘f.


• If \( f \) is a rule small function extension: \( \text{\texttt{tagged}}(A, f) \) is the rule tagged small function extension containing \(<\text{\texttt{modelTagged}}, \text{\texttt{tag}}\) where:
  
  – \( \text{\texttt{src}}(\text{\texttt{modelTagged}}) = \text{\texttt{dom}}(f) \).
  
  – For each \( \text{\texttt{src}}(\text{\texttt{modelTagged}}) \) program \( x \), \( \text{\texttt{modelTagged}}(x) = \text{\texttt{tagged}}(\text{\texttt{inferDomExt}}(A, f, x), f<x>) \).
  
  – \( \text{\texttt{tag}} = \text{\texttt{inferDomExt}}(A, f) \).

\( \text{\texttt{untag}}(\text{\texttt{tagged}}(A, f)) \) the rule small function extension \( \text{\texttt{untagF}} \) such that

\( \text{\texttt{dom}}(\text{\texttt{untagF}}) = \text{\texttt{src}}(\text{\texttt{modelTagged}}) \) and, for each \( \text{\texttt{dom}}(\text{\texttt{untagF}}) \) program \( x \),

\( \text{\texttt{untagF}}<x> = \text{\texttt{untag}}(\text{\texttt{modelTagged}}(x)) \).

\( \text{\texttt{dom}}(\text{\texttt{untagF}}) = \text{\texttt{dom}}(f) \).

For each \( \text{\texttt{dom}}(f) \) program \( x \),

\( \text{\texttt{untagF}}<x> = f<x> \) (by inductive hypothesis).

\( \text{\texttt{untagF}} = f \). \( \blacksquare \)

For a valid domain extension \( A \), and a tagged small function extension \( f \) such that

\( \text{\texttt{untag}}(f) \) is a \( \text{\texttt{dom}}(A) \) program, \( \text{\texttt{tagged}}(A, \text{\texttt{untag}}(f)) = f \).

**Proof.** \( \text{\texttt{untag}}(\text{\texttt{tagged}}(A, \text{\texttt{untag}}(f))) = \text{\texttt{untag}}(f) \), \( \text{\texttt{tagged}}(A, \text{\texttt{untag}}(f)) = f \) (by tag irrelevance theorem). \( \blacksquare \)

For valid domain extensions \( A \) and \( B \), and a small function extension \( f \) such that \( f \) is a \( \text{\texttt{dom}}(A) \) program and a \( \text{\texttt{dom}}(B) \) program, \( \text{\texttt{tagged}}(A, f) = \text{\texttt{tagged}}(B, f) \).

**Proof.** \( \text{\texttt{untag}}(\text{\texttt{tagged}}(A, f)) = f = \text{\texttt{untag}}(\text{\texttt{tagged}}(B, f)) \), \( \text{\texttt{tagged}}(A, f) = \text{\texttt{tagged}}(B, f) \) (by tag irrelevance theorem). \( \blacksquare \)

For a small function extension \( f \), \( f \) is **taggable** iff there exists some tagged small function extension \( g \) such that \( \text{\texttt{untag}}(g) = f \).

For a small function extension \( f \), \( f \) is **untaggable** iff \( f \) is not taggable.

For a small function extension \( f \), if there exists some valid domain extension \( A \) such that \( f \) is a \( \text{\texttt{dom}}(A) \) program, then \( f \) is taggable.

**Proof.** \( \text{\texttt{untag}}(\text{\texttt{tagged}}(A, f)) = f \). \( \blacksquare \)

For a small function extension \( f \), if \( f \) is untaggable, then, for each valid domain extension \( A \), \( f \) is **not** a \( \text{\texttt{dom}}(A) \) program.

For valid domain extensions \( A \) and \( B \), \( A = B \) iff, for each tagged small function extension \( f \), \( \text{\texttt{untag}}(f) \) is a \( \text{\texttt{dom}}(A) \) program iff \( \text{\texttt{untag}}(f) \) is a \( \text{\texttt{dom}}(B) \) program.

**Proof.**
• Holds if \( A = B \).

• If for each tagged small function extension \( f \), \( \text{untag}(f) \) is a \( \text{dom}(A) \) program iff \( \text{untag}(f) \) is a \( \text{dom}(B) \) program:
  
  – For each small function extension \( g \):
    
    * If \( g \) is taggable: Let \( f \) be a tagged small function extension such that \( \text{untag}(f) = g \). \( g \) is a \( \text{dom}(A) \) program iff \( g \) is a \( \text{dom}(B) \) program.
    
    * If \( g \) is untaggable: \( g \) is neither a \( \text{dom}(A) \) program nor a \( \text{dom}(B) \) program.

  – \( \text{dom}(A) = \text{dom}(B) \). \( A = B \) (by domain extension irrelevance theorem).

For a \textit{valid} domain extension family \( F \), and a \( \text{dom}(F) \) program \( x \), the \textit{tagged} by \( F \) of \( x \), denoted by \( \text{tagged}(F; x) \), is \( \text{tagged}(\text{domExt}(F), x) \). Note that \( \text{dom}(\text{domExt}(F)) = \text{dom}(F) \).

For a \textit{valid} domain extension family \( F \), and a \( \text{dom}(F) \) program \( x \), \( \text{untag}(\text{tagged}(F; x)) = x \).

\textit{Proof.} \( \text{untag}(\text{tagged}(F; x)) = \text{untag}(\text{tagged}(\text{domExt}(F), x)) \).

For a \textit{valid} domain extension family \( F \), and a tagged small function extension \( x \), if \( \text{untag}(x) \) is a \( \text{dom}(F) \) program, \( \text{tagged}(F; \text{untag}(x)) = x \).

\textit{Proof.} \( \text{dom}(\text{domExt}(F)) = \text{dom}(F) \). \( \text{tagged}(F; \text{untag}(x)) = \text{tagged}(\text{domExt}(F), \text{untag}(x)) \).

For \textit{valid} domain extension families \( F \) and \( G \), and a small function extension \( x \) such that \( x \) is a \( \text{dom}(F) \) program and a \( \text{dom}(G) \) program, \( \text{tagged}(F; x) = \text{tagged}(G, x) \).

\textit{Proof.} \( \text{tagged}(F; x) = \text{tagged}(\text{domExt}(F), x) = \text{tagged}(\text{domExt}(G), x) = \text{tagged}(G, x) \).

\section{7.12 Domain, Domain Extension, Specific Result and Rank of a Tagged Small Function Extension}

For a tagged small function extension \( f \), the \textit{domain} of \( f \) (a small sub-language of
'Func.Sm.Ext), denoted by 'dom('f), is 'dom('untag('f)).

For a tagged small function extension 'f, 'dom('f) is given by one of the following mutually exclusive cases:

- 'Func.Sm.Ext.Null if 'f = 'Func.Sm.Ext.null
- 'Func.Sm.Ext.Null if 'f = 'Func.Sm.Ext.zero
- 'Func.Sm.Ext.Nuro if 'f = 'Func.Sm.Ext.one
- 'Func.Sm.Ext.Leaf if 'f is a pair tagged small function extension
- 'src('model) if 'f is a rule tagged small function extension containing <'model, 'tag>

For a tree tagged small function extension 't, 'dom('t) is a sub-language of 'Func.Sm.Ext.Leaf.

For a tagged small function extension 'f, the domain extension of 'f, denoted by 'domExt('f), is given by one of the following mutually exclusive cases:

- 'domExt('untag('f)) if 'f is a simple tagged small function extension
- 'tag if 'f is a rule tagged small function extension containing <'model, 'tag>

For a tagged small function extension 'f, 'domExt('f) is valid.

For a tagged small function extension 'f, 'dom('domExt('f)) = 'dom('f).

Proof.

- If 'f is a simple tagged small function extension: 'dom('domExt('f)) = 'dom('domExt('untag('f))) = 'dom('untag('f)) = 'dom('f).

- If 'f is a rule tagged small function extension containing <'model, 'tag>:
  'domExt('f) = 'tag. 'dom('tag) = 'src('model). 'dom('f) = 'src('model). □

For a tagged small function extension 'f, and a 'dom('f) program 'x, the tagged by 'f of 'x, denoted by 'tagged('f, 'x), is 'tagged('domExt('f), 'x). Note that 'dom('domExt('f)) = 'dom('f).

For a tagged small function extension 'f, and a 'dom('f) program 'x, 'untag('tagged('f, 'x)) = 'x.
Proof. \[ \text{untag}(\text{tagged}(\mathbf{f}, \mathbf{x})) = \text{untag}(\text{tagged}(\text{domExt}(\mathbf{f}), \mathbf{x})). \]

For tagged small function extensions \( \mathbf{f} \) and \( \mathbf{x} \), if \( \text{untag}(\mathbf{x}) \) is a \( \text{dom}(\mathbf{f}) \) program, \( \text{tagged}(\mathbf{f}, \text{untag}(\mathbf{x})) = \mathbf{x} \).

Proof. \[ \text{tagged}(\mathbf{f}, \text{untag}(\mathbf{x})) = \text{tagged}(\text{domExt}(\mathbf{f}), \text{untag}(\mathbf{x})). \]

For tagged small function extensions \( \mathbf{f} \) and \( \mathbf{g} \), and a small function extension \( \mathbf{x} \) such that \( \mathbf{x} \) is a \( \text{dom}(\mathbf{f}) \) program and a \( \text{dom}(\mathbf{g}) \) program, \( \text{tagged}(\mathbf{f}, \mathbf{x}) = \text{tagged}(\mathbf{g}, \mathbf{x}) \).

Proof. \[ \text{tagged}(\mathbf{f}, \mathbf{x}) = \text{tagged}(\text{domExt}(\mathbf{f}), \mathbf{x}) = \text{tagged}(\text{domExt}(\mathbf{g}), \mathbf{x}) = \text{tagged}(\mathbf{g}, \mathbf{x}). \]

For a tagged small function extension \( \mathbf{f} \), \( \text{Func.Sm.Ext.null} \) is a \( \text{dom}(\mathbf{f}) \) program, and \( \text{dom}(\mathbf{f}) \) is non-empty.

Proof. \( \text{Func.Sm.Ext.null} \) is a \( \text{dom}(\text{domExt}(\mathbf{f})) = \text{dom}(\mathbf{f}) \) program.

For tagged small function extensions \( \mathbf{f} \) and \( \mathbf{g} \), \( \text{domExt}(\mathbf{f}) = \text{domExt}(\mathbf{g}) \) iff \( \text{dom}(\mathbf{f}) = \text{dom}(\mathbf{g}) \).

Proof. \[ \text{dom}(\mathbf{f}) = \text{dom}(\text{domExt}(\mathbf{f})). \text{dom}(\mathbf{g}) = \text{dom}(\text{domExt}(\mathbf{g})). \]

• Holds if \( \text{domExt}(\mathbf{f}) = \text{domExt}(\mathbf{g}) \).

• If \( \text{dom}(\mathbf{f}) = \text{dom}(\mathbf{g}) \); \( \text{dom}(\text{domExt}(\mathbf{f})) = \text{dom}(\text{domExt}(\mathbf{g})). \text{domExt}(\mathbf{f}) = \text{domExt}(\mathbf{g}) \) (by domain extension irrelevance theorem).

For tagged small function extensions \( \mathbf{f} \) and \( \mathbf{g} \), \( \text{domExt}(\mathbf{f}) = \text{domExt}(\mathbf{g}) \) iff, for each tagged small function extension \( \mathbf{x} \), \( \text{untag}(\mathbf{x}) \) is a \( \text{dom}(\mathbf{f}) \) program iff \( \text{untag}(\mathbf{x}) \) is a \( \text{dom}(\mathbf{g}) \) program.

For tagged small function extensions \( \mathbf{f} \) and \( \mathbf{g} \), \( \text{dom}(\mathbf{f}) = \text{dom}(\mathbf{g}) \) iff, for each tagged small function extension \( \mathbf{x} \), \( \text{untag}(\mathbf{x}) \) is a \( \text{dom}(\mathbf{f}) \) program iff \( \text{untag}(\mathbf{x}) \) is a \( \text{dom}(\mathbf{g}) \) program.

For a tagged small function extension \( \mathbf{f} \), and a \( \text{dom}(\mathbf{f}) \) program \( \mathbf{x} \), the specific result of \( \mathbf{f} \) at \( \mathbf{x} \), denoted by \( \mathbf{f}<\mathbf{x}> \), is given by one of the following mutually exclusive cases:

• \( \mathbf{x} \) if \( \mathbf{f} \) is a leaf small function extension
• ‘Func.Sm.Ext.null if ‘f is a pair tagged small function extension and ‘x = ‘Func.Sm.Ext.null

• ‘left(‘f) if ‘f is a pair tagged small function extension and ‘x = ‘Func.Sm.Ext.zero

• ‘right(‘f) if ‘f is a pair tagged small function extension and ‘x = ‘Func.Sm.Ext.one

• ‘model(‘x) if ‘f is a rule tagged small function extension containing <‘model, ‘tag>

For a tagged small function extension ‘f, the range of ‘f (a small sub-language of ‘Func.Sm.Ext.Tagged), denoted by ‘ran(‘f), is the language of all ‘f<‘x> such that ‘x is a ‘dom(‘f) program.

For a pair tagged small function extension ‘p, ‘ran(‘p) is the language whose only programs are ‘Func.Sm.Ext.null, ‘left(‘p) and ‘right(‘p).


For a tree tagged small function extension ‘t, ‘ran(‘t) is a sub-language of ‘Func.Sm.Ext.Tagged.Tree.

For a tagged small function extension ‘f ≠ ‘Func.Sm.Ext.null, and a ‘dom(‘f) program ‘x, ‘x is structurally smaller than ‘f.

For a tagged small function extension ‘f ≠ ‘Func.Sm.Ext.null, and a ‘ran(‘f) program ‘x, ‘x is structurally smaller than ‘f.

For rule tagged small function extensions ‘f and ‘g, ‘f = ‘g iff ‘domExt(‘f) = ‘domExt(‘g) and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘g<‘x>.

Proof.

• Holds if ‘f = ‘g.

• If ‘domExt(‘f) = ‘domExt(‘g) and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘g<‘x>:

   ‘dom(‘f) = ‘dom(‘g).

   □

For rule tagged small function extensions ‘f and ‘g, ‘f = ‘g iff ‘dom(‘f) = ‘dom(‘g) and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘g<‘x>.

For tagged small function extensions ‘f and ‘g, ‘f = ‘g iff exactly one of the following holds:

• ‘f = ‘Func.Sm.Ext.null and ‘g = ‘Func.Sm.Ext.null.

• ‘f = ‘Func.Sm.Ext.zero and ‘g = ‘Func.Sm.Ext.zero.
50

• ‘f = ‘Func.Sm.Ext.one and ‘g = ‘Func.Sm.Ext.one.

• ‘f and ‘g are pair tagged small function extensions, and ‘left(‘f) = ‘left(‘g) and ‘right(‘f) = ‘right(‘g).

• ‘f and ‘g are rule tagged small function extensions, and ‘domExt(‘f) = ‘domExt(‘g), and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘g<‘x>.

For a rule tagged small function extension ‘f, ‘untag(‘f) is the rule small function extension ‘untagF such that ‘dom(‘untagF) = ‘dom(‘f) and, for each ‘dom(‘untagF) program ‘x, ‘untagF<‘x> = ‘untag(‘f<‘x>).

For a tagged small function extension ‘f, the rank of ‘f, denoted by ‘rank(‘f), is ‘rank(‘untag(‘f)).

7.13 IDENTITY TAGGED SMALL FUNCTION EXTENSIONS

For a valid domain extension ‘A, the identity tagged small function extension on ‘A, denoted by ‘Func.Sm.Ext.Tagged.identity(‘A), is the rule tagged small function extension ‘f such that ‘domExt(‘f) = ‘A and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘tagged(‘f, ‘x).


For a tagged small function extension ‘f, the domain tagged small function extension of ‘f, denoted by ‘domFuncExt(‘f), is ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f)).

For a tagged small function extension ‘f, ‘domFuncExt(‘f) is given by one of the following mutually exclusive cases:


• ‘Func.Sm.Ext.Tagged.Leaf.set if ‘f is a pair tagged small function extension

• ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f)) if ‘f is a rule tagged small function extension

For a tagged small function extension ‘f, ‘domFuncExt(‘f) is a rule tagged small function extension.

For a tagged small function extension ‘f, ‘domExt(‘domFuncExt(‘f)) = ‘domExt(‘f).

Proof. ‘domFuncExt(‘f) = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f)).


For a tagged small function extension ‘f, ‘dom(‘domFuncExt(‘f)) = ‘dom(‘f).

Proof. ‘domExt(‘domFuncExt(‘f)) = ‘domExt(‘f).

For tagged small function extensions ‘f and ‘g, ‘domFuncExt(‘f) = ‘domFuncExt(‘g) iff ‘domExt(‘f) = ‘domExt(‘g).

Proof.


• If ‘domFuncExt(‘f) = ‘domFuncExt(‘g): ‘domExt(‘domFuncExt(‘f)) = ‘domExt(‘g).

For tagged small function extensions ‘f and ‘g, ‘domFuncExt(‘f) = ‘domFuncExt(‘g) iff, for each tagged small function extension ‘x, ‘untag(‘x) is a ‘dom(‘f) program iff ‘untag(‘x) is a ‘dom(‘g) program.

For rule tagged small function extensions ‘f and ‘g, ‘f = ‘g iff ‘domFuncExt(‘f) = ‘domFuncExt(‘g) and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘g<‘x>.

For a tagged small function extension ‘f, ‘f is an identity iff, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘tagged(‘f, ‘x).

For a valid domain extension ‘A, ‘func.Sm.Ext.Tagged.identity(‘A) is an identity.

For a tagged small function extension ‘f, ‘domFuncExt(‘f) is an identity.

For rule tagged small function extensions ‘f and ‘g, if ‘f and ‘g are identities, then ‘f = ‘g iff ‘dom(‘f) = ‘dom(‘g).
Proof.

- Holds if \( f = g \).
- If \( \text{dom}(f) = \text{dom}(g) \): For each \( \text{dom}(f) \) program \( x \), \( f\langle x \rangle = \text{tagged}(f, x) = \text{tagged}(g, x) = g\langle x \rangle \).

For rule tagged small function extensions \( f \) and \( g \), if \( f \) and \( g \) are identities, then \( f = g \) iff \( \text{domExt}(f) = \text{domExt}(g) \).

For a rule tagged small function extension \( f \), if \( f \) is an identity, then \( \text{domFuncExt}(f) = f \).

Proof. \( \text{domExt}(\text{domFuncExt}(f)) = \text{domExt}(f) \). \( \text{domFuncExt}(f) \) is a rule tagged small function extension and an identity.

For a tagged small function extension \( f \), \( \text{domFuncExt}(\text{domFuncExt}(f)) = \text{domFuncExt}(f) \).

Proof. \( \text{domFuncExt}(f) \) is a rule tagged small function extension and an identity.

7.14 COERCION OF A TAGGED SMALL FUNCTION EXTENSION, AND COERCION STABILITY THEOREM

Coercion is to be used to define tagged small function extensions over all tagged small function extensions. Of course, coercion should be reasonable and useful. Coercion is also computable. For a valid domain extension \( A \), and a tagged small function extension \( f \), the general principles of the coercion to \( A \) of \( f \) are:

- If \( A \) is a constant domain extension, check whether \( \text{untag}(f) \) is a \( \text{dom}(A) \) program. If so, return \( \text{untag}(f) \). If not, return \( \text{Func.Sm.Ext.null} \).

- If \( A \) is a dependent sum domain extension, and \( f \) is a pair tagged small function extension, coerce \( \text{left}(f) \) first, then \( \text{right}(f) \).

- If \( A \) is a dependent sum domain extension, and \( f \) is not a pair tagged small function extension, return \( \text{Func.Sm.Ext.null} \).

- If \( A \) is a dependent product domain extension, and \( f \) is a rule tagged small function extension, coerce \( f \) to the desired domain and codomain by adding pre-coercion and post-coercion to \( f \).
• If 'A is a dependent product domain extension, and 'f is not a rule tagged small function extension, return 'Func.Sm.Ext.null.

For a valid domain extension 'A, and a tagged small function extension 'f, the coercion to 'A of 'f (a 'dom('A) program), denoted by 'coer('A, 'f), is defined by recursion on <'A, 'f> using < on coercion pairs (a well-founded relation to be defined shortly):

• If 'A is a constant domain extension: 'coer('A, 'f) is 'untag('f) if 'untag('f) is a 'dom('A) program; and 'Func.Sm.Ext.null otherwise.

• If 'A is a dependent sum domain extension containing 'F, and 'f is a pair tagged small function extension: 'coer('A, 'f) is the pair small function extension 'p such that 'left('p) = 'coer('domExt('F), 'left('f)) and 'right('p) = 'coer('F< 'left('p)>, 'right('f)). Note that 'dom('domExt('F)) = 'dom('F), 'left('p) is 'dom('F) program, and 'right('p) is a 'dom('F< 'left('p)>) program.

• If 'A is a dependent product domain extension containing 'F, and 'f is not a pair tagged small function extension: 'coer('A, 'f) = 'Func.Sm.Ext.null.

• If 'A is a dependent product domain extension containing 'F, and 'f is a rule tagged small function extension: 'coer('A, 'f) is the rule small function extension 'r such that 'dom('r) = 'dom('F) and, for each 'dom('r) program 'x, 'r< 'x> = 'coer('F< 'x>, 'f< 'coer('domExt('f), 'tagged('F ; 'x))>). Note that 'dom('domExt('f)) = 'dom('f), and 'r< 'x> is a 'dom('F< 'x>) program.

• If 'A is a dependent product domain extension, and 'f is not a rule tagged small function extension: 'coer('A, 'f) = 'Func.Sm.Ext.null.

A coercion pair is <'A, 'f> where 'A is a valid domain extension and 'f is a tagged small function extension. An ordinal pair is <'A, 'f> where 'A and 'f are ordinals. For a coercion pair 'p = <'A, 'f>, the ordinal pair of 'p, denoted by 'ord('p), is <'rank('A), 'rank('f)>.

The well-founded relation used to define coercion is not a well-founded square dance, but a well-founded tango. For ordinal pairs 'p = <'A, 'f> and 'q = <'B, 'g>, let 'p < 'q iff at least one of the following holds:

1. 'A < 'B and 'f ≤ 'g.

2. 'A ≤ 'B and 'f < 'g.
3. ‘A < ‘g and ‘f ≤ ‘B.

4. ‘A ≤ ‘g and ‘f < ‘B.

Cases 1 and 2 are easy steps, and cases 3 and 4 are twists.

For coercion pairs ‘p and ‘q, let ‘p < ‘q iff ‘ord(‘p) < ‘ord(‘q).


All the recursive calls in the definition of coercion are decreasing.

Proof.


For ordinal pairs ‘p = ‘< ‘A, ‘f> and ‘q = ‘< ‘B, ‘g>, let ‘p < s ‘q iff at least one of the following holds:

1. ‘A < ‘B and ‘f ≤ ‘g.

2. ‘A ≤ ‘B and ‘f < ‘g.

< s on ordinal pairs is well-founded.

Proof.

• Suppose, for contradiction, that < s on ordinal pairs is not well-founded. Then there is some model ‘p from ‘Nat such that, for each natural number ‘m, ‘p(‘m) is an ordinal pair, ‘p(‘m + 1) < s ‘p(‘m) and at least one of the following holds:

1. ‘left(‘p(‘m + 1)) < ‘left(‘p(‘m)) and ‘right(‘p(‘m + 1)) ≤ ‘right(‘p(‘m)).
2. \( \text{left}(p(m+1)) \leq \text{left}(p(m)) \) and \( \text{right}(p(m+1)) < \text{right}(p(m)) \).

• If case 1 holds for only finitely many \( m \), then case 2 holds for infinitely many \( m \), and there is an infinite descending chain in the right. Otherwise, there is an infinite descending chain in the left. Either way, \( < \) on ordinals is not well-founded, a contradiction.

< on ordinal pairs is well-founded.

Proof.

• Suppose, for contradiction, that \( < \) on ordinal pairs is not well-founded. Then there is some model \( p \) from \( \text{Nat} \) such that, for each natural number \( m \), \( p(m) \) is an ordinal pair, \( p(m+1) < p(m) \) and at least one of the following holds:

1. \( \text{left}(p(m+1)) < \text{left}(p(m)) \) and \( \text{right}(p(m+1)) \leq \text{right}(p(m)) \).
2. \( \text{left}(p(m+1)) \leq \text{left}(p(m)) \) and \( \text{right}(p(m+1)) < \text{right}(p(m)) \).
3. \( \text{left}(p(m+1)) < \text{right}(p(m)) \) and \( \text{right}(p(m+1)) \leq \text{left}(p(m)) \).
4. \( \text{left}(p(m+1)) \leq \text{right}(p(m)) \) and \( \text{right}(p(m+1)) < \text{left}(p(m)) \).

• Let \( \text{twist} \) be the model from \( \text{Nat} \) such that, for each natural number \( m \), \( \text{twist}(m) = 0 \) if case 1 or 2 holds; and 1 otherwise. Let \( \text{path} \) be the model from \( \text{Nat} \) such that \( \text{path}(0) = 0 \) and, for each natural number \( m \), \( \text{path}(m+1) = \text{not}(\text{path}(m)) \) if \( \text{twist}(m) \); and \( \text{path}(m) \) otherwise. Let \( p_0 \) be the model from \( \text{Nat} \) such that, for each natural number \( m \), \( p_0(m) = \text{flip}(p(m)) \) if \( \text{path}(m) \); and \( p(m) \) otherwise. For each natural number \( m \), \( p_0(m) \) is an ordinal pair and at least one of the following holds:

1. \( \text{left}(p_0(m+1)) < \text{left}(p_0(m)) \) and \( \text{right}(p_0(m+1)) \leq \text{right}(p_0(m)) \).
2. \( \text{left}(p_0(m+1)) \leq \text{left}(p_0(m)) \) and \( \text{right}(p_0(m+1)) < \text{right}(p_0(m)) \).

• For each natural number \( m \), \( p_0(m+1) < p_0(m) \). \( < \) on ordinal pairs is not well-founded, a contradiction.

< on coercion pairs is well-founded.

Proof. Suppose, for contradiction, that \( < \) on coercion pairs is not well-founded. Then there is some model \( p \) from \( \text{Nat} \) such that, for each natural number \( m \), \( p(m) \) is a coercion pair, \( p(m+1) < p(m) \) and \( \text{ord}(p(m+1)) < \text{ord}(p(m)) \). Let \( p_0 \) be the model
from ‘Nat such that, for each natural number ‘m, ‘p0(‘m) = ‘ord(‘p(‘m)). For each natural number ‘m, ‘p0(‘m) is an ordinal pair and ‘p0(‘m + 1) < ‘p0(‘m). < on ordinal pairs is not well-founded, a contradiction.

For a valid domain extension family ‘F, and a tagged small function extension ‘x, the coercion by ‘F of ‘x, denoted by ‘coer(‘F, ‘x), is ‘coer(‘domExt(‘F), ‘x).

For a valid domain extension family ‘F, and a tagged small function extension ‘x, ‘coer(‘F, ‘x) is a ‘dom(‘F) program.


For tagged small function extensions ‘f and ‘x, the coercion by ‘f of ‘x, denoted by ‘coer(‘f, ‘x), is ‘coer(‘domExt(‘f), ‘x).

For tagged small function extensions ‘f and ‘x, ‘coer(‘f, ‘x) is a ‘dom(‘f) program.


For a valid dependent sum domain extension ‘A containing ‘F, and a pair tagged small function extension ‘f, ‘coer(‘A, ‘f) is the pair small function extension ‘p such that ‘left(‘p) = ‘coer(‘F, ‘left(‘f)) and ‘right(‘p) = ‘coer(‘F<‘left(‘p)> ‘right(‘f)).

For a valid dependent product domain extension ‘A containing ‘F, and a rule tagged small function extension ‘f, ‘coer(‘A, ‘f) is the rule small function extension ‘r such that ‘dom(‘r) = ‘dom(‘F) and, for each ‘dom(‘r) program ‘x, ‘r<‘x> = ‘coer(‘F<‘x>, ‘f<‘coer(‘f, ‘tagged(‘F, ‘x))>).

Coercion does not make unnecessary changes. The coercion stability theorem: For a valid domain extension ‘A, and a tagged small function extension ‘f, if ‘untag(‘f) is a ‘dom(‘A) program, then ‘coer(‘A, ‘f) = ‘untag(‘f).

Proof.

• By induction on ‘<‘A, ‘f> using < on coercion pairs.

• Holds if ‘A is a constant domain extension.

‘dom(F<‘left(p)>) program and ‘coer(F<‘left(p)>, ‘right(f)) = ‘right(‘untagF) (by inductive hypothesis). ‘right(p) = ‘right(‘untagF). ‘p = ‘untagF.


• If ‘A is a dependent product domain extension containing ‘F, and ‘f is a rule tagged small function extension: ‘coer(‘A, ‘f) is the rule small function extension ‘r such that ‘dom(‘r) = ‘dom(‘F) and, for each ‘dom(‘r) program ‘x, ‘r<‘x> = ‘coer(F<‘x>, ‘f<‘coer(domExt(‘f), ‘tagged(‘F, ‘x))>). ‘untag(f) is the rule small function extension ‘untagF such that ‘dom(‘untagF) = ‘dom(‘f) and, for each ‘dom(‘untagF) program ‘x, ‘untagF<‘x> = ‘untag(f<‘x>). ‘dom(‘untagF) = ‘dom(‘F) = ‘dom(‘r). ‘dom(domExt(‘f)) = ‘dom(f) = ‘dom(‘untag(f)) = ‘dom(‘F).

  For each ‘dom(‘r) program ‘x, ‘x is a ‘dom(domExt(‘f)) program, ‘untag(tagged(‘F, ‘x)) = ‘x, and ‘coer(domExt(‘f), ‘tagged(‘F, ‘x)) = ‘x (by inductive hypothesis). For each ‘dom(‘r) program ‘x, ‘untagF<‘x> = ‘untag(f<‘x>) is a ‘dom(F<‘x>) program, and ‘coer(F<‘x>, ‘f<‘x>) = ‘untagF<‘x> (by inductive hypothesis). ‘For each ‘dom(‘r) program ‘x, ‘r<‘x> = ‘untagF<‘x>. ‘r = ‘untagF.


  For a valid domain extension ‘A, and a tagged small function extension ‘f, if ‘coer(‘A, ‘f) = ‘untag(f), then ‘untag(f) is a ‘dom(‘A) program.

  Proof. ‘coer(‘A, ‘f) is a ‘dom(‘A) program.

  For a valid domain extension family ‘F, and a tagged small function extension ‘x, if ‘untag(‘x) is a ‘dom(‘F) program, then ‘coer(‘F, ‘x) = ‘untag(‘x).


  For tagged small function extensions ‘f and ‘x, if ‘untag(‘x) is a ‘dom(‘f) program, then ‘coer(‘f, ‘x) = ‘untag(‘x).


  Proof. ‘Func.Sm.Ext.null is a ‘dom(‘A) program.

For a tagged small function extension \( f \), \( \text{coer}(\text{Dom.Ext.Nuro}, f) = f \) if \( f \) is a nuro; and \( \text{Func.Sm.Ext.null} \) otherwise.

For a tagged small function extension \( f \), \( \text{coer}(\text{Dom.Ext.Leaf}, f) = f \) if \( f \) is a leaf small function extension; and \( \text{Func.Sm.Ext.null} \) otherwise.

For a tagged small function extension \( f \), \( \text{coer}(\text{Dom.Ext.Tree}, f) = \text{untag}(f) \) if \( f \) is a tree; and \( \text{Func.Sm.Ext.null} \) otherwise.

### 7.15 RESULT OF A TAGGED SMALL FUNCTION EXTENSION

Coercion is now used to define tagged small function extensions over all tagged small function extensions, while maintaining computability. This generalized definition of result is the basis for reduction.

For tagged small function extensions \( f \) and \( x \), the **result** of \( f \) and \( x \), denoted by \( f(x) \), is \( f<\text{coer}(f, x)> \).

For a *valid dependent product* domain extension \( A \) containing \( F \), and a *rule* tagged small function extension \( f \), \( \text{coer}(A, f) \) is the rule small function extension \( r \) such that \( \text{dom}(r) = \text{dom}(F) \) and, for each \( \text{dom}(r) \) program \( x \), \( r<x> = \text{coer}( F<x>, f(\text{tagged}(F, x)) ) \).

**Proof.** For each \( \text{dom}(F) \) program \( x \), \( f(\text{tagged}(F, x)) = f<\text{coer}(f, \text{tagged}(F, x)> \). □

For tagged small function extensions \( f \) and \( x \), if \( f \) is an identity, then \( f(x) = \text{tagged}(f, \text{coer}(f, x)) \) and \( \text{coer}(f, x) = \text{untag}(f(x)) \).

**Proof.** \( f(x) = f<\text{coer}(f, x)> = \text{tagged}(f, \text{coer}(f, x)) \). □

For tagged small function extensions \( f \) and \( x \), if \( f \) is an identity, then \( \text{untag}(f(x)) \) is a \( \text{dom}(f) \) program.

For a *valid* domain extension \( A \), and a tagged small function extension \( x \), \( \text{Func.Sm.Ext.Tagged.identity}(A)(x) = \text{tagged}(A, \text{coer}(A, x)) \) and \( \text{coer}(A, x) = \text{untag}(\text{Func.Sm.Ext.Tagged.identity}(A)(x)) \).

**Proof.** \( \text{Func.Sm.Ext.Tagged.identity}(A) \) is an identity.
\( \text{Func.Sm.Ext.Tagged.identity}(A)(x) = \text{tagged}(\text{Func.Sm.Ext.Tagged.identity}(A), \text{coer}(\text{Func.Sm.Ext.Tagged.identity}(A), x)). \text{domExt}(\text{Func.Sm.Ext.Tagged.identity}(A)) = A. \) □
For a valid domain extension 'A, and a tagged small function extension 'x, 'untag('Func.Sm.Ext.Tagged.identity('A)('x)) is a 'dom('A) program.

For tagged small function extensions 'f and 'x, 'domFuncExt('f)('x) = 'tagged('f, 'coer('f, 'x)) and 'coer('f, 'x) = 'untag('domFuncExt('f)('x)).

**Proof.** 'domFuncExt('f) = 'Func.Sm.Ext.Tagged.identity('domExt('f)). 'domFuncExt('f)('x) = 'Func.Sm.Ext.Tagged.identity('domExt('f))('x) = 'tagged('domExt('f), 'coer('domExt('f), 'x)).

For tagged small function extensions 'f and 'x, 'untag('domFuncExt('f)('x)) is a 'dom('f) program.

For tagged small function extensions 'f and 'x, 'f('x) =

'f<untag('domFuncExt('f)('x))>.

**Proof.** 'f('x) = 'f<coer('f, 'x)> 'coer('f, 'x) = 'untag('domFuncExt('f)('x)).

For tagged small function extensions 'f and 'x, if 'untag('x) is a 'dom('f) program, then 'f('x) = 'f<untag('x)>.

**Proof.** 'f('x) = 'f<coer('f, 'x)> 'coer('f, 'x) = 'untag('x).

For a tagged small function extension 'f, and a 'dom('f) program 'x, 'f<tagged('f, 'x)> =

'f<'.

**Proof.** 'untag(tagged('f, 'x)) = 'x. 'untag(tagged('f, 'x)) is a 'dom('f) program.

For a tagged small function extension 'f, 'ran('f) is the language of all 'f<tagged('f, 'x)> such that 'x is a 'dom('f) program.

For a tagged small function extension 'f, 'ran('f) is the language of all 'f('x) such that 'x is a tagged small function extension.

**Proof.**

- For each 'ran('f) program 'y: There exists some 'dom('f) program 'z such that 'f<tagged('f, 'z)> = 'y.

- For each tagged small function extension 'x: 'f('x) = 'f<coer('f, 'x)> 'f('x) is a 'ran('f) program.

For tagged small function extensions 'f and 'x, if 'f is an identity, then 'untag('x) is a 'dom('f) program iff 'f('x) = 'x.

**Proof.**
• If ‘untag(x) is a ‘dom(f) program: f(x) = f<untag(x)> = tagged(f, untag(x)) = x.

• If f(x) = x: untag(f(x)) is a ‘dom(f) program.

For tagged small function extensions f and x, untag(x) is a ‘dom(f) program iff domFuncExt(f)(x) = x.

Proof. domFuncExt(f) is an identity. untag(x) is a ‘dom(domFuncExt(f)) program iff domFuncExt(f)(x) = x. dom(domFuncExt(f)) = dom(f).

For tagged small function extensions f and x, domFuncExt(f)(domFuncExt(f)(x)) = domFuncExt(f)(x).

Proof. untag(domFuncExt(f)(x)) is a ‘dom(f) program.

For rule tagged small function extensions f and g, f = g iff dom(f) = dom(g) and, for each tagged small function extension x, f(x) = g(x).

Proof.

• Holds if f = g.

• If dom(f) = dom(g) and, for each tagged small function extension x, f(x) = g(x):
  
  – For each dom(f) program y: Let x = tagged(f, y). x = tagged(g, y). untag(x) = y. f(x) = f<y>. g(x) = g<y>. f(x) = g(x). f<y> = g<y>.
  
  – f = g.

For rule tagged small function extensions f and g, f = g iff domExt(f) = domExt(g) and, for each tagged small function extension x, f(x) = g(x).

For rule tagged small function extensions f and g, f = g iff domFuncExt(f) = dom- FuncExt(g) and, for each tagged small function extension x, f(x) = g(x).

For a tagged small function extension x, Func.Sm.Ext.null(x) = Func.Sm.Ext.null.


For a tagged small function extension x, Func.Sm.Ext.zero(x) = Func.Sm.Ext.null.

For a tagged small function extension ‘x, ‘Func.Sm.Ext.one(\(x\)) = ‘x if ‘x is a nuro; and ‘Func.Sm.Ext.null otherwise.


For a pair tagged small function extension ‘f, and a tagged small function extension ‘x, ‘f(\(x\)) is given by one of the following mutually exclusive cases:

- ‘\(\text{left}(f)\) if ‘x = ‘Func.Sm.Ext.zero
- ‘\(\text{right}(f)\) if ‘x = ‘Func.Sm.Ext.one
- ‘‘Func.Sm.Ext.null if ‘x is not Boolean

Proof.

- ‘\(f(x) = f<‘\text{coer}(‘\text{Dom.Ext.Leaf}, ‘x)>\).

- ‘\(f(x)\) is given by one of the following mutually exclusive cases:
  - ‘\(f<‘\text{Func.Sm.Ext.zero}\) if ‘x = ‘Func.Sm.Ext.zero
  - ‘\(f<‘\text{Func.Sm.Ext.one}\) if ‘x = ‘Func.Sm.Ext.one
  - ‘\(f<‘\text{Func.Sm.Ext.null}\) if ‘x is not Boolean □

For pair tagged small function extensions ‘f and ‘g, ‘f = ‘g iff ‘f(‘Func.Sm.Ext.zero) = ‘g(‘Func.Sm.Ext.zero), and ‘f(‘Func.Sm.Ext.one) = ‘g(‘Func.Sm.Ext.one).

7.16 EXTENSIONALITY THEOREM

Since NummSquared does not include sets as primitive, within NummSquared, equals on rule tagged small function extensions cannot refer to equals on their domains (which are languages). One alternative would be to refer to equals on their domain extensions. But the coercion stability theorem permits a second and simpler alternative, which is embodied in the following extensionality theorem.
For rule tagged small function extensions 'f and 'g, if 'f and 'g are identities, then 'f = 'g iff, for each tagged small function extension 'x, 'f('x) = 'g('x).

Proof.

- Holds if 'f = 'g.
- If for each tagged small function extension 'x, 'f('x) = 'g('x):
  - For each tagged small function extension 'x: 'f('x) = 'x iff 'g('x) = 'x. 'untag('x) is a 'dom('f) program iff 'untag('x) is a 'dom('g) program.
  - 'domExt('f) = 'domExt('g).

For tagged small function extensions 'f and 'g, 'domFuncExt('f) = 'domFuncExt('g) iff, for each tagged small function extension 'x, 'domFuncExt('f)('x) = 'domFuncExt('g)('x).

Proof.

- Holds if 'domFuncExt('f) = 'domFuncExt('g).
- If for each tagged small function extension 'x, 'domFuncExt('f)('x) = 'domFuncExt('g)('x): 'domFuncExt('f) and 'domFuncExt('g) are rule tagged small function extensions and identities.

The extensionality theorem: For rule tagged small function extensions 'f and 'g, 'f = 'g iff for each tagged small function extension 'x, 'domFuncExt('f)('x) = 'domFuncExt('g)('x) and 'f('x) = 'g('x).

Proof.

- Holds if 'f = 'g.
- If for each tagged small function extension 'x, 'domFuncExt('f)('x) = 'domFuncExt('g)('x) and 'f('x) = 'g('x): 'domFuncExt('f) = 'domFuncExt('g).
7.17 SOME TAGGED SMALL FUNCTION EXTENSIONS


For a tagged small function extension 'x, 'Func.Sm Ext.Tagged.Nuro.set('x) = 'x if 'x is a nuro; and 'Func.Sm.Ext.null otherwise.


For a tagged small function extension 'x, 'Func.Sm.Ext.Tagged.Leaf.set('x) is 'x if 'x is a leaf small function extension; and 'Func.Sm.Ext.null otherwise.

Proof. 'Func.Sm.Ext.Tagged.Leaf.set('x) = 'tagged('Dom.Ext.Leaf, 'coer('Dom.Ext.Leaf, 'x)).

For a tagged small function extension 'x, 'Func.Sm.Ext.Tagged.Tree.set('x) = 'x if 'x is a tree; and 'Func.Sm.Ext.null otherwise.

Proof. 'Func.Sm.Ext.Tagged.Tree.set('x) = 'tagged('Dom.Ext.Tree, 'coer('Dom.Ext.Tree, 'x)). 'coer('Dom.Ext.Tree, 'x) = 'untag('x) if 'x is a tree; and 'Func.Sm.Ext.null otherwise.

For a valid domain extension family 'F, let 'Func.Sm.Ext.Tagged.sum.dep('F) = 'Func.Sm.Ext.Tagged.identity(' A) where ' A is the dependent sum domain extension containing 'F.

For a valid domain extension family 'F, and a pair tagged small function extension 'x, 'Func.Sm.Ext.Tagged.sum.dep('F)('x) is the pair tagged small function extension 'p such that 'left('p) = 'Func.Sm.Ext.Tagged.identity('domExt('F))('left('x)) and 'right('p) = 'Func.Sm.Ext.Tagged.identity('F<untag('left('p))>('right('x)). Note that 'untag('left('p)) is a 'dom('domExt('F)) = 'dom('F) program.

Proof. 'Func.Sm.Ext.Tagged.sum.dep('F) = 'Func.Sm.Ext.Tagged.identity(' A) where ' A is the dependent sum domain extension containing 'F. 'untag('Func.Sm.Ext.Tagged.sum.dep('F)('x)) = 'coer('A, 'x). 'coer('A, 'x) is the pair small function extension 'q such that 'left('q) = 'coer('F, 'left('x)) and 'right('q) = 'coer('F<left('q)>,'right('x)). 'left('p) = 'tagged('F, 'coer('F, 'left('x))). 'right('p) =
tagged('F<untag('left('p))>, 'coer('F<untag('left('p))>, 'right('x))). 'untag('p) = {'untag('left('p)), 'untag('right('p))}. Let 'untagP = 'untag('p). 'left('untagP) = 'untag('left('p)) = 'left('q). 'right('untagP) = 'coer('F<untag('left('p))>, 'right('x)) = 'right('q). 'q = 'untagP. 'Func.Sm.Ext.Tagged.sum.dep('F)('x) = 'p (by tag irrelevance theorem).

For a valid domain extension family 'F, and a non-pair tagged small function extension 'x, 'Func.Sm.Ext.Tagged.sum.dep('F)('x) = 'Func.Sm.Ext.null.

Proof. 'Func.Sm.Ext.Tagged.sum.dep('F) = 'Func.Sm.Ext.Tagged.identity('A) where 'A is the dependent sum domain extension containing 'F.

'Func.Sm.Ext.Tagged.sum.dep('F)('x) = 'tagged('A, 'coer('A, 'x)).

For a valid domain extension family 'F, let 'Func.Sm.Ext.Tagged.prod.dep('F) = 'Func.Sm.Ext.Tagged.identity('A) where 'A is the dependent product domain extension containing 'F.

For a valid domain extension family 'F, and a rule tagged small function extension 'x, 'Func.Sm.Ext.Tagged.prod.dep('F)('x) is the rule tagged small function extension 'r such that 'domExt('r) = 'domExt('F) and, for each 'dom('r) program 'y, 'r<y> = 'Func.Sm.Ext.Tagged.identity('F<y>)(x('tagged('r, 'y))). Note that 'dom('r) = 'dom('domExt('r)) = 'dom('domExt('F)) = 'dom('F).

Proof. 'Func.Sm.Ext.Tagged.prod.dep('F) = 'Func.Sm.Ext.Tagged.identity('A) where 'A is the dependent product domain extension containing 'F. 'untag('Func.Sm.Ext.Tagged.prod.dep('F)('x)) = 'coer('A, 'x). 'coer('A, 'x) is the rule small function extension 's such that 'dom('s) = 'dom('F) and, for each 'dom('s) program 'y, 's<y> = 'coer('F<y>, 'x('tagged('F, 'y))). 'dom('s) = 'dom('r). For each 'dom('r) program 'y, 'r<y> = 'tagged('F<y>, 'x('tagged('r, 'y))). 'untag('r) is the rule small function extension 'untagR such that 'dom('untagR) = 'dom('r) and, for each 'dom('untagR) = 'dom('s) program 'y, 'untagR<y> = 'untag('r<y>) = 's<y>. 's = 'untagR. 'Func.Sm.Ext.Tagged.prod.dep('F)('x) = 'r (by tag irrelevance theorem).

For a valid domain extension family 'F, and a non-rule tagged small function extension 'x, 'Func.Sm.Ext.Tagged.prod.dep('F)('x) = 'Func.Sm.Ext.null.

Proof. 'Func.Sm.Ext.Tagged.prod.dep('F) = 'Func.Sm.Ext.Tagged.identity('A) where 'A is the dependent product domain extension containing 'F.

'Func.Sm.Ext.Tagged.prod.dep('F)('x) = 'tagged('A, 'coer('A, 'x)).

Domain extensions never appear directly in NummSquared programs, but tagged small function extensions are used to create domain extensions when necessary.
For a tagged small function extension \( f \), the **domain extension family** of \( f \), denoted by \('\text{domExtFam}(f)\), is the valid domain extension family \( F \) such that \('\text{domExt}(F) = \text{domExt}(f)\) and, for each \('\text{dom}(F)\) program \( x \), \( F \langle x \rangle = \text{domExt}(f(\text{tagged}(F, x)))\).

For a tagged small function extension \( f \), \('\text{domExt}(\text{domExtFam}(f)) = \text{domExt}(f)\), \('\text{dom}(\text{domExtFam}(f)) = \text{dom}(f)\) and, for each \('\text{dom}(f)\) program \( x \), \('\text{domExtFam}(f) \langle x \rangle = \text{domExt}(f(\text{tagged}(f, x)))\).

**Proof.** Let \( F = \text{domExtFam}(f) \). \('\text{domExt}(F) = \text{domExt}(f)\), \('\text{dom}(F) = \text{dom}(\text{domExt}(f)) = \text{dom}(\text{domExt}(f)) = \text{dom}(f)\). □

For a tagged small function extension \( f \), the **dependent sum** of \( f \), denoted by \('\text{sumDep}(f)\), is \('\text{Func.Sm.Ext.Tagged.sum.dep}(\text{domExtFam}(f))\).

For a tagged small function extension \( f \), and a **pair** tagged small function extension \( x \), \('\text{sumDep}(f) \langle x \rangle\) is the pair tagged small function extension \( p \) such that \('\text{left}(p) = \text{domFuncExt}(f)(\text{left}(x))\) and \('\text{right}(p) = \text{domFuncExt}(f(\text{left}(p)))(\text{right}(x))\).

**Proof.** \('\text{sumDep}(f) \langle x \rangle = \text{Func.Sm.Ext.Tagged.sum.dep}(\text{domExtFam}(f)) \langle x \rangle\).
\('\text{Func.Sm.Ext.Tagged.sum.dep}(\text{domExtFam}(f)) \langle x \rangle\) is the pair tagged small function extension \( p \) such that \('\text{left}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{left}(x))\) and \('\text{right}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{right}(x))\).
\('\text{left}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{left}(x)) = \text{domFuncExt}(f)(\text{left}(x))\). \('\text{domExtFam}(f) \langle \text{untag}(\text{left}(p)) \rangle = \text{domExt}(f(\text{left}(p)))\). \('\text{right}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{right}(x)) = \text{domFuncExt}(f(\text{left}(p)))(\text{right}(x))\). □

For a tagged small function extension \( f \), and a **non-pair** tagged small function extension \( x \), \('\text{sumDep}(f) \langle x \rangle = \text{Func.Sm.Ext.null}\).

**Proof.** \('\text{sumDep}(f) \langle x \rangle = \text{Func.Sm.Ext.Tagged.sum.dep}(\text{domExtFam}(f)) \langle x \rangle\).
\('\text{Func.Sm.Ext.Tagged.sum.dep}(\text{domExtFam}(f)) \langle x \rangle\) is the pair tagged small function extension \( p \) such that \('\text{left}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{left}(x))\) and \('\text{right}(p) = \text{Func.Sm Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{right}(x))\).
\('\text{left}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{left}(x)) = \text{domFuncExt}(f)(\text{left}(x))\). \('\text{right}(p) = \text{Func.Sm.Ext.Tagged.identity}(\text{domExt}(\text{domExtFam}(f)))(\text{right}(x)) = \text{domFuncExt}(f)(\text{left}(p))\). □

For a tagged small function extension \( f \), the **dependent product** of \( f \), denoted by \('\text{prodDep}(f)\), is \('\text{Func.Sm.Ext.Tagged.prod.dep}(\text{domExtFam}(f))\).

For a tagged small function extension \( f \), and a **rule** tagged small function extension \( x \), \('\text{prodDep}(f) \langle x \rangle\) is the rule tagged small function extension \( r \) such that \('\text{domExt}(r) = \text{domExt}(f)\) and, for each \('\text{dom}(r)\) program \( y \), \('r \langle y \rangle = \text{domFuncExt}(f(\text{tagged}(r, y)))(\text{left}(y))\).

**Proof.** \('\text{prodDep}(f) \langle x \rangle = \text{Func.Sm.Ext.Tagged.prod.dep}(\text{domExtFam}(f)) \langle x \rangle\).
\('\text{Func.Sm.Ext.Tagged.prod.dep}(\text{domExtFam}(f)) \langle x \rangle\) is the rule tagged small function
extension `r such that \( \text{domExt}'(r) = \text{domExt}'(\text{domExtFam}'(f)) \) and, for each \( \text{dom}'(r) \) program \( y \), \( r<y> = \text{Func.Sm.Ext.Tagged.identity}(\text{domExtFam}'(f)<y>)(x(tagged'(r, 'y))). \text{domExt}'(\text{domExtFam}'(f)) = \text{domExt}'(f). \text{domExt}'(r) = \text{domExt}'(f). \) For each \( \text{dom}'(r) \) program \( y \), \( \text{domExtFam}'(f)<y> = \text{domExt}'(f(tagged'(r, 'y)))) = \text{domExt}'(f(tagged'(r, 'y))))(x(tagged'(r, 'y))) = \text{domFuncExt}'(f(tagged'(r, 'y)))(x(tagged'(r, 'y))). \) For a tagged small function extension 'f, and a non-rule tagged small function extension 'x, \( \text{prodDep}'(f)(x) = \text{Func.Sm.Ext.null}. \)

\textbf{Proof.} \( \text{prodDep}'(f)(x) = \text{Func.Sm.Ext.Tagged.prod.dep}(\text{domExtFam}'(f))(x). \)

\section*{7.18 LARGE FUNCTION EXTENSIONS AND TRUTH}

Whereas small function extensions are the core of NummSquared, large function extensions are the face of NummSquared.

A large function extension contains a model 'model from 'Func.Sm.Ext.Tagged to 'Func.Sm.Ext.Tagged.

Let 'Func.Lg.Ext be the language of all large function extensions.

For a large function extension 'f containing 'model, and a tagged small function extension 'x, the result of 'f at 'x, denoted by 'f'(x), is 'model(x).

For large function extensions 'f and 'g, 'f = 'g iff for each tagged small function extension 'x, 'f(x) = 'g(x).

For a large function extension 'f, the result of 'f, denoted by 'res'(f), is 'f'(Func.Sm.Ext.null).

For a large function extension 'f, 'f is unchanging iff, for each tagged small function extension 'x, 'f'(x) = 'res'(f).

For a large function extension 'f, 'f is unchanging iff, for each tagged small function extension 'x, and each tagged small function extension 'y, 'f'(x) = 'f'(y).

For a tagged small function extension 'x, 'x is true iff 'x = 'Func.Sm.Ext.one.

For a tagged small function extension 'f, 'f is universally true iff, for each tagged small function extension 'x, 'f'(x) is true.

For a tagged small function extension 'f, 'f is universally true iff for each 'ran'(f) program 'y, 'y is true.
For a tagged small function extension `f`, `f` is universally true iff for each `dom(f)` program `x`, `f(tagged(f, 'x)) = 'f<x>` is true.

A **proposition extension** is a large function extension. For a large function extension `f`, `f` is **true** iff, for each tagged small function extension `x`, `f(x)` is true.

Truth of a tagged small function extension is computable. Universal truth of a tagged small function extension is not computable. Truth of a large function extension is not computable.

### 7.19 SOME COMPUTATIONAL LARGE FUNCTION EXTENSIONS

For a tagged small function extension `y`, the **constant large function extension** to `y`, denoted by `Func.Lg.Ext.constant(y)`, is the large function extension containing `constant(Func.Sm.Ext.Tagged, y)`.

For tagged small function extensions `y` and `x`, `Func.Lg.Ext.constant(y)(x) = y`.

For a tagged small function extension `y`, `res(Func.Lg.Ext.constant(y)) = y`.

**Proof.** `Func.Lg.Ext.constant(y)(Func.Sm.Ext.null) = y`.

For a tagged small function extension `y`, and `Func.Lg.Ext.constant(y)` is unchanging.

**Proof.** For each tagged small function extension `x`, `Func.Lg.Ext.constant(y)(x) = y` and `Func.Lg.Ext.constant(y)(x) = res(Func.Lg.Ext.constant(y))`.

Let `Func.Lg.Ext.i` be the large function extension containing `identity(Func.Sm.Ext.Tagged)`.

For a tagged small function extension `x`, `Func.Lg.Ext.i(x) = x`.

Let `Func.Lg.Ext.null = Func.Lg.Ext.constant(Func.Sm.Ext.null)`.

`res(Func.Lg.Ext.null) = Func.Sm.Ext.null`.

Let `Func.Lg.Ext.zero = Func.Lg.Ext.constant(Func.Sm.Ext.zero)`.

`res(Func.Lg.Ext.zero) = Func.Sm.Ext.zero`.

Let `Func.Lg.Ext.one = Func.Lg.Ext.constant(Func.Sm.Ext.one)`.

`res(Func.Lg.Ext.one) = Func.Sm.Ext.one`.


Let \( '\text{Func.Lg.Ext.Tree.set} = '\text{Func.Lg.Ext.constant('Func.Sm.Ext.Tagged.Tree.set).} \)


Let \( '\text{Func.Lg.Ext.Null be the large function extension such that, for each tagged small function extension 'x, 'Func.Lg.Ext.Null('x) is 'Func.Sm.Ext.one if 'x = 'Func.Sm.Ext.null; and 'Func.Sm.Ext.zero otherwise.} \)

Let \( '\text{Func.Lg.Ext.Pair be the large function extension such that, for each tagged small function extension 'x, 'Func.Lg.Ext.Pair('x) is 'Func.Sm.Ext.one if 'x is a pair tagged small function extension; and 'Func.Sm.Ext.zero otherwise.} \)

Let \( '\text{Func.Lg.Ext.dom be the large function extension such that, for each tagged small function extension 'x, 'Func.Lg.Ext.dom('x) = 'domFuncExt('x).} \)

### 7.20 SOME COMPUTATIONAL COMBINATIONS OF LARGE FUNCTION EXTENSIONS

For large function extensions ‘outer and ‘inner, the **large composition** of ‘outer and ‘inner, denoted by \([‘outer ‘inner]\), is the large function extension such that, for each tagged small function extension ‘x, \([‘outer ‘inner](‘x) = ‘outer([‘inner(‘x)). Large composition is similar to axiom II.7 in [40].

For large function extensions ‘called and ‘arg, the **small composition** of ‘called and ‘arg, denoted by \((‘called ‘arg)), is the large function extension such that, for each tagged small function extension ‘x, \((‘called ‘arg)('x) = ‘called(('x)('arg('x)).

The definition of small composition requires some explanation. ‘called('x), a tagged small function extension, is called with argument ‘arg('x), another tagged small function extension.

For a natural number \( m \geq 2 \), and large function extensions ‘x\(_0\), ‘x\(_1\), ..., ‘x\(_{m-2}\), ‘x\(_{m-1}\), let \( (‘x\(_0\) ‘x\(_1\) ... ‘x\(_{m-2}\) ‘x\(_{m-1}\) = ((‘x\(_0\) ‘x\(_1\) ... ‘x\(_{m-2}\) ‘x\(_{m-1}\)\)\). Large composition is similar to axiom II.7 in [40].

For large function extensions ‘l and ‘r, the **pair** of ‘l and ‘r, denoted by \{"l ‘r\}, is the large function extension such that, for each tagged small function extension ‘x, \{"l ‘r\}(‘x) = \{"l(‘x), ‘r(‘x)). Pair is similar to axiom II.6 in [40].

For large function extensions ‘l and ‘r, \( '\text{res}\{{‘l ‘r}\} = '{\text{res}}('l), ‘\text{res}('r)\).
Proof. \( \{l \, \{\text{Func.Sm.Ext.null}\} \} = \{l(\{\text{Func.Sm.Ext.null}\}), r(\{\text{Func.Sm.Ext.null}\})\} \).

For large function extensions \( l \) and \( r \), if \( l \) and \( r \) are unchanging, then \( \{l \, r\} \) is unchanging.

Proof. For each tagged small function extension \( x \), \( \{l \, r\}(x) = \{l(x), r(x)\} = \{\text{res}(l), \text{res}(r)\} = \text{res}(\{l \, r\}) \).

Pairs are used to represent tuples (in a manner similar to [36, p.16]). For a natural number \( m \geq 2 \), and large function extensions \( x_0, x_1, ..., x_{m-2}, x_{m-1} \), let \( \{x_0 \, x_1 \, ... \, \, x_{m-2} \, x_{m-1}\} \).

Pairs are used to represent lists (in a manner similar to [29]). For a natural number \( m \), and large function extensions \( x_0, x_1, ..., x_{m-1} \), let \( \tilde{\text{list}}\{x_0 \, x_1 \, ... \, \, x_{m-1}\} = \{x_0 \, \{x_1 \, ... \, \, x_{m-1} \, \text{Func.Lg.Ext.zero}\}\} \).

\( \text{list} \} = \text{Func.Lg.Ext.zero} \), not \( \text{Func.Lg.Ext.null} \). The empty list is often interpreted differently than the absence of relevant information.

There are no multi-argument large function extensions, but tuples are used to simulate multiple arguments. The fact that all large function extensions are actually unary makes it much simpler to implement arity polymorphic combinations of large functions (for example, Curry and quantifications). For a natural number \( m \geq 2 \), and large function extensions \( f \) and \( x_0, x_1, ..., x_{m-1} \), let \( [f \, x_0 \, x_1 \, ... \, \, x_{m-1}] \).

For a large function extension \( \text{family} \), the dependent sum of \( \text{family} \), denoted by \( \tilde{\text{s.d}}[\text{family}] \), is the large function extension such that, for each tagged small function extension \( x \), \( \tilde{\text{s.d}}[\text{family}](x) = \text{sumDep}(\text{family}(x)) \).

For a large function extension \( \text{family} \), the dependent product of \( \text{family} \), denoted by \( \tilde{\text{p.d}}[\text{family}] \), is the large function extension such that, for each tagged small function extension \( x \), \( \tilde{\text{p.d}}[\text{family}](x) = \text{prodDep}(\text{family}(x)) \).

For large function extensions \( \text{uncurry} \) and \( \text{restrictor} \), the Curry of \( \text{uncurry} \) to \( \text{restrictor} \), denoted by \( \tilde{\text{c}}[\text{uncurry} \, \text{restrictor}] \), is the large function extension such that, for each tagged small function extension \( x \), \( \tilde{\text{c}}[\text{uncurry} \, \text{restrictor}](x) \) is the rule tagged small function extension \( r \) such that \( \text{domExt}(r) = \text{domExt}(\text{restrictor}(x)) \) and, for each \( \text{dom}(r) \) program \( y \), \( r < y > = \text{uncurry}([x, \text{tagged}(r, y)]) \).

The definition of Curry requires some explanation. For large function extensions \( \text{uncurry} \) and \( \text{restrictor} \), and a small function extension \( x \), \( \tilde{\text{c}}[\text{uncurry} \, \text{restrictor}](x) \) is a rule tagged small function extension \( r \) representing a partial call to \( \text{uncurry} \) at \( x \). However, \( r \) is restricted using the domain extension of \( \text{restrictor}(x) \). The restriction is
necessary because 'r is a tagged small function extension, not a large function extension.

For large function extensions 'uncurry and 'restrictor, and a tagged small function extension 'x, 'domExt(\(\tilde{c}['uncurry 'restrictor]('x)) = 'domExt('restrictor('x)),
'dom(\(\tilde{c}['uncurry 'restrictor]('x)) = 'dom('restrictor('x)), and 'domFuncExt(\(\tilde{c}['uncurry 'restrictor]('x)) = 'domFuncExt('restrictor('x)).

For large function extensions 'uncurry and 'restrictor, and tagged small function extensions 'x and 'y, \(\tilde{c}['uncurry 'restrictor]('x)('y) = 'uncurry(['x, 'domFuncExt('restrictor('x))(y))).

Proof. \(\tilde{c}['uncurry 'restrictor]('x) is the rule tagged small function extension 'r
such that 'domExt('r) = 'domExt('restrictor('x)) and, for each 'dom('r) program 'z, 'r<z> = 'uncurry(['x, tagged('r, 'z))). \(\tilde{c}['uncurry 'restrictor]('x)(y) = 'r(y) =
'r<untag('domFuncExt('r)(y))> = 'uncurry(['x, 'domFuncExt('r)(y))). 'domFuncExt('r) = 'domFuncExt('restrictor('x)).

For large function extensions 'ifP, 'thenP and 'elseP the if-then-else of 'ifP, 'thenP
and 'elseP, denoted by \(\tilde{ite}['ifP 'thenP 'elseP], is the large function extension such that,
for each tagged small function extension 'x, \(\tilde{ite}['ifP 'thenP 'elseP]('x) is given by one of
the following mutually exclusive cases:

• 'elseP('x) if 'ifP('x) = 'Func.Sm.Ext.zero

• 'thenP('x) if 'ifP('x) = 'Func.Sm.Ext.one

• 'Func.Sm.Ext.null if 'ifP('x) is not Boolean

Recall that, for a small function extension 'f ≠ 'Func.Sm.Ext.null, and a 'field('f) pro-
gram 'x, 'x is structurally smaller than 'f. This fact permits a simple terminating recur-
sion principle for NummSquared.

For large function extensions 'start and 'step, the recursion of 'start and 'step, de-
noted by \(\tilde{r}['start 'step], is the large function extension such that, for each tagged small
function extension 'x, \(\tilde{r}['start 'step]('x) is defined by recursion on 'untag('x):

• If 'x = 'Func.Sm.Ext.null: \(\tilde{r}['start 'step]('x) = 'start('x).

• If 'x ≠ 'Func.Sm.Ext.null: \(\tilde{r}['start 'step]('x) = 'step((rDom, 'Ran, 'x)) where:

  – 'rDom is the rule tagged small function extension such that 'domExt('rDom)
    = 'domExt('x) and, for each for each 'dom('rDom) program 'y, 'rDom<\'y>
7.21 SOME NON-COMPUTATIONAL LARGE FUNCTION EXTENSIONS AND COMBINATIONS

NummSquared includes equals, which is non-computational by the extensionality theorem. Equals therefore cannot be used in reduction, but is essential in propositions. Let \( \text{Func.Lg.Ext.eq} \) be the large function extension such that, for each tagged small function extension \( p \), \( \text{Func.Lg.Ext.eq}(p) \) is given by one of the following mutually exclusive cases:

- \( \text{Func.Sm.Ext.one} \) if \( p \) is a pair tagged small function extension, and \( \text{left}(p) = \text{right}(p) \)
- \( \text{Func.Sm.Ext.zero} \) if \( p \) is a pair tagged small function extension, and \( \text{left}(p) \neq \text{right}(p) \)
- \( \text{Func.Sm.Ext.null} \) if \( p \) is not a pair tagged small function extension

Hilbert’s epsilon operator is a form of the axiom of choice, and can be used to define both existential and universal quantification. The epsilon calculus is a logic based on the Hilbert operator. (See [4] for an overview and rules of inference.)

NummSquared includes adaptations of the Hilbert operator and the inference rules of the epsilon calculus. Hilbert cannot be used in reduction, but is essential in propositions. For a large function extension \( \text{pred} \), the Hilbert of \( \text{pred} \), denoted by \( \text{Hilbert}[\text{pred}] \),
is the large function extension such that, for each tagged small function extension 'x,
'h[pred](x) is some tagged small function extension 'y such that 'pred((x, 'y)) is true if
such a 'y exists; and 'Func.Sm.Ext.null otherwise.
CHAPTER 8

NUMMSQUARED SYNTAX

NummSquared abstract syntax is now defined, including reduction and proof. Abstract syntax is also related to semantics. NummSquared concrete syntax is also defined. NummSquared is variable-free.

NummSquared syntax is developed as follows:

• Normalized large functions are defined. Not all normalized large functions are in simplest form. In lambda calculus terminology, NummSquared does not reduce under lambdas.

• The extension of a normalized large function (a large function extension) is defined. A normalized large function is true iff its extension is true.

• Reduction is defined in a way that is sufficient for software where the output is a tree (which is typical), for macros performing syntactic manipulation of normalized large functions, and for manipulating proofs.

• Quoted and unquoted are defined for normalized large functions.

• Macro expanded is defined.

• Substitution is defined.

• The substitution theorem: substitution preserves equality.

• Comments and identifiers are defined.

• Large functions, syntactic sugar for normalized large functions, are defined.

• Definitions, definition lists, modules and abstract programs are defined.
• Contexts are defined.

• Normal forms and validity are defined.

• Some true large function extensions and inferences are given.

• Some true normalized large functions and inferences are given. Among these are induction, modus ponens, specialization and substitution.

• Proofs are defined.

• The proposition and validity of a proof are defined.

• The soundness theorem: the proposition of a valid proof is true.

• Quoted and unquoted are defined for proofs.

• NummSquared averts Russell’s paradox.

### 8.1 NORMALIZED LARGE FUNCTIONS

A **computational normalized constant** is exactly one of the following:

• the identity computational normalized constant, ‘Constant.Norm.Compu.i

• the null computational normalized constant, ‘Constant.Norm.Compu.null

• the zero computational normalized constant, ‘Constant.Norm.Compu.zero

• the one computational normalized constant, ‘Constant.Norm.Compu.one

• the null set computational normalized constant, ‘Constant.Norm.Compu.Null.set

• the nuro set computational normalized constant, ‘Constant.Norm.Compu.Nuro.set

• the leaf set computational normalized constant, ‘Constant.Norm.Compu.Leaf.set

• the tree set computational normalized constant, ‘Constant.Norm.Compu.Tree.set
• the null predicate computational normalized constant, ‘Constant.Norm.Compu.Null

• the pair predicate computational normalized constant, ‘Constant.Norm.Compu.Pair

• the domain computational normalized constant, ‘Constant.Norm.Compu.dom

The above computational normalized constants are written in the concrete syntax as follows:

~i
~null
~zero
~one
~Null.set
~Nuro.set
~Leaf.set
~Tree.set
~Null
~Pair
~dom

A non-computational normalized constant is exactly one of the following:

• the equals non-computational normalized constant, ‘Constant.Norm.Noncompu.eq

The above non-computational normalized constants are written in the concrete syntax as follows:

~=

A normalized constant is exactly one of the following:

• a computational normalized constant

• a non-computational normalized constant

Normalized large functions are defined inductively. Let ‘Func.Lg.Norm be the language of all normalized large functions.

A normalized large function is exactly one of the following:
• a normalized constant

• a normalized combination

A normalized combination is exactly one of the following:

• a computational normalized combination

• a non-computational normalized combination

A computational normalized combination is exactly one of the following:

• a large composition computational normalized combination

• a small composition computational normalized combination

• a pair computational normalized combination

• a dependent sum computational normalized combination

• a dependent product computational normalized combination

• a Curry computational normalized combination

• an if-then-else computational normalized combination

• a recursion computational normalized combination

A large composition computational normalized combination contains <outer, inner> where ‘outer and ‘inner are normalized large functions. For normalized large functions ‘outer and ‘inner, let [‘outer ‘inner] be the large composition computational normalized combination containing <‘outer, ‘inner>.

A small composition computational normalized combination contains <‘called, ‘arg> where ‘called and ‘arg are normalized large functions. For normalized large functions ‘called and ‘arg, let (‘called ‘arg) be the small composition computational normalized combination containing <‘called, ‘arg>.

A pair computational normalized combination contains <‘left, ‘right> where ‘left and ‘right are normalized large functions. For normalized large functions ‘left and ‘right, let {‘left ‘right} be the pair computational normalized combination containing <‘left, ‘right>.

A dependent sum computational normalized combination contains ‘family where ‘family is a normalized large function. For a normalized large function ‘family, let
A dependent product computational normalized combination contains ‘family where ‘family is a normalized large function. For a normalized large function ‘family, let \( \tilde{p}.d['family] \) be the dependent product computational normalized combination containing ‘family.

A Curry computational normalized combination contains \(<'uncurry, 'restrictor>\) where ‘uncurry and ‘restrictor are normalized large functions. For normalized large functions ‘uncurry and ‘restrictor, let \( \tilde{c['uncurry 'restrictor]} \) be the Curry computational normalized combination containing \(<'uncurry, 'restrictor>\).

An if-then-else computational normalized combination contains \(<'ifP, 'thenP, 'elseP>\) where ‘ifP, ‘thenP and ‘elseP are normalized large functions. For normalized large functions ‘ifP, ‘thenP and ‘elseP, let \( \tilde{ite['ifP 'thenP 'elseP]} \) be the if-then-else computational normalized combination containing \(<'ifP, 'thenP, 'elseP>\).

A recursion computational normalized combination contains \(<'start, 'step>\) where ‘start and ‘step are normalized large functions. For normalized large functions ‘start and ‘step, let \( \tilde{r['start 'step]} \) be the recursion computational normalized combination containing \(<'start, 'step>\).

A non-computational normalized combination is exactly one of the following:

- a Hilbert non-computational normalized combination

A Hilbert non-computational normalized combination contains ‘pred where ‘pred is a normalized large function. For a normalized large function ‘pred, let \( \tilde{h['pred]} \) be the Hilbert non-computational normalized combination containing ‘pred.

This concludes the inductive definition.

For a natural number ‘m \(\geq 2\), and normalized large functions ‘x0, ‘x1, ..., ‘xm-2, ‘xm-1, let \( ('x_0 'x_1 ... 'x_{m-2} 'x_{m-1}) = ((('x_0 'x_1) ... 'x_{m-2}) 'x_{m-1}) \).

For a natural number ‘m \(\geq 2\), and normalized large functions ‘x0, ‘x1, ..., ‘xm-2, ‘xm-1, let \( ('x_0 'x_1 ... 'x_{m-2} 'x_{m-1}) = (((x_0 'x_1) ... 'x_{m-2}) 'x_{m-1}) \).

For a natural number ‘m, and normalized large functions ‘x0, ‘x1, ..., ‘xm-1, let \( l[ 'x_0 'x_1 ... 'x_{m-1} = {x_0 [ 'x_1 ... [ 'x_{m-1} 'Constant.Norm.Compu.zero]]) \).

For a natural number ‘m \(\geq 2\), and normalized large functions ‘f and ‘x0, ‘x1, ..., ‘xm-1, let \( [f 'x_0 'x_1 ... 'x_{m-1}] = [f ['x_0 'x_1 ... 'x_{m-1}] \).
8.2 EXTENSION AND TRUTH OF A NORMALIZED LARGE FUNCTION

For a normalized constant \( c \), the extension of \( c \) (a large function extension), denoted by \( \text{ext}(c) \), is given by one of the following mutually exclusive cases:

- \( '\text{Func.Lg.Ext.i} \) if \( c = '\text{Constant.Norm.Compu.i} \)
- \( '\text{Func.Lg.Ext.null} \) if \( c = '\text{Constant.Norm.Compu.null} \)
- \( '\text{Func.Lg.Ext.zero} \) if \( c = '\text{Constant.Norm.Compu.zero} \)
- \( '\text{Func.Lg.Ext.one} \) if \( c = '\text{Constant.Norm.Compu.one} \)
- \( '\text{Func.Lg.Ext.Null.set} \) if \( c = '\text{Constant.Norm.Compu.Null.set} \)
- \( '\text{Func.Lg.Ext.Leaf.set} \) if \( c = '\text{Constant.Norm.Compu.Leaf.set} \)
- \( '\text{Func.Lg.Ext.Tree.set} \) if \( c = '\text{Constant.Norm.Compu.Tree.set} \)
- \( '\text{Func.Lg.Ext.Null} \) if \( c = '\text{Constant.Norm.Compu.Null} \)
- \( '\text{Func.Lg.Ext.Pair} \) if \( c = '\text{Constant.Norm.Compu.Pair} \)
- \( '\text{Func.Lg.Ext.dom} \) if \( c = '\text{Constant.Norm.Compu.dom} \)
- \( '\text{Func.Lg.Ext.eq} \) if \( c = '\text{Constant.Norm.Compu.eq} \)

For a normalized large function \( f \), the extension of \( f \) (a large function extension), denoted by \( \text{ext}(f) \), is defined by recursion on \( f \):

- as above if \( f \) is a normalized constant
- \( ['\text{ext('outer)} '\text{ext('inner))] \) if \( f = ['\text{outer} '\text{inner}] \)
- \( ('\text{ext('called)} '\text{ext('arg)) \) if \( f = ('\text{called} '\text{arg}) \)
- \( ('\text{ext('left)} '\text{ext('right)) \) if \( f = ('\text{left} '\text{right}) \)
- \( \sim s.d['\text{ext('family}}]) \) if \( f = \sim s.d['\text{family}] \)
- \( \sim p.d['\text{ext('family}}]) \) if \( f = \sim p.d['\text{family}] \)
• \(\neg c[\text{\text{\text{\text{\text{\text{ext('uncurry) 'ext('restrictor)}}}}}}] \text{ if } f = \neg c[\text{\text{\text{\text{\text{\text{uncurry 'restrictor)}}}}}}]

• \(\neg \text{ite[\text{\text{\text{\text{\text{\text{ext('ifP) 'ext('thenP) 'ext('elseP)}}}}}}] \text{ if } f = \neg \text{ite[\text{\text{\text{\text{\text{\text{ifP 'thenP 'elseP)}}}}}}]}

• \(\neg r[\text{\text{\text{\text{\text{\text{ext('start) 'ext('step)}}}}}}] \text{ if } f = \neg r[\text{\text{\text{\text{\text{\text{start 'step)}}}}}}]

• \(\neg h[\text{\text{\text{\text{\text{\text{ext('pred)}}}}}}] \text{ if } f = \neg h[\text{\text{\text{\text{\text{\text{pred)}}}}}}]

There is some large function extension \(f\) such that there exists no normalized large function \(fn\) with \(\text{ext}(fn) = f\).

\textbf{Proof.} 'Func.Lg.Ext is uncountable. 'Func.Lg.Norm is countably infinite.

For a normalized large function \(fn\), there is some normalized large function \(fn0 \neq fn\) with \(\text{ext}(fn0) = \text{ext}(fn)\).

\textbf{Proof.} Let \(fn0 = [\text{\text{\text{\text{\text{\text{Constant.Norm.Compu.i 'fn)}}}}}}\). Not all normalized large functions are in simplest form. In lambda calculus terminology, NummSquared does not reduce under lambdas. In future, NummSquared may reduce under lambdas.

For a normalized large function \(f\), and a tagged small function extension \(x\), the result of \(f\) at \(x\), denoted by \(f(x)\), is \(\text{ext}(f)(x)\).

For a normalized large function \(f\), the result of \(f\), denoted by \(\text{res}(f)\), is \(\text{res}(\text{ext}(f))\).

For a normalized large function \(f\), \(f\) is unchanging iff \(\text{ext}(f)\) is unchanging.

A \textbf{normalized proposition} is a normalized large function. For a normalized large function \(f\), \(f\) is \textbf{true} iff \(\text{ext}(f)\) is true.

\section{8.3 REDUCTION: COMPUTED OF A NORMALIZED LARGE FUNCTION}

For a normalized large function \(f\), the property of \(f\) being \textbf{deep computational} is defined by recursion on \(f\):

• If \(f\) is a computational normalized constant: \(f\) is deep computational.

• If \(f\) is a non-computational normalized constant: \(f\) is \textit{not} deep computational.

• If \(f = ['outer 'inner]\): \(f\) is deep computational iff \('outer\) and \('inner\) are deep computational.
• If \( f = (\text{called} \ 'arg) \): \( f \) is deep computational iff ‘called and ‘arg are deep computational.

• If \( f = \{\text{left} \ 'right\} \): \( f \) is deep computational iff ‘left and ‘right are deep computational.

• If \( f = \text{s.d}[\text{family}] \): ‘f is deep computational iff ‘family is deep computational.

• If \( f = \text{p.d}[\text{family}] \): ‘f is deep computational iff ‘family is deep computational.

• If \( f = \text{c}[\text{uncurry} \ 'restrictor] \): ‘f is deep computational iff ‘uncurry and ‘restrictor are deep computational.

• If \( f = \text{ite}[\text{ifP} \ 'thenP \ 'elseP] \): ‘f is deep computational iff ‘ifP, ‘thenP and ‘elseP are deep computational.

• If \( f = \text{r}[\text{start} \ 'step] \): ‘f is deep computational iff ‘start and ‘step are deep computational.

• If ‘f is a non-computational normalized combination: ‘f is not deep computational.

For a deep computational normalized large function ‘f, ‘res(‘f) is computable. However, ‘res(‘f) is a tagged small function extension (a semantic object), but a normalized large function (a syntactic object) is desired for reduction.

For a normalized large function ‘f, the property of ‘f being a tree is defined by recursion on ‘f:


• If ‘f = ‘{left ‘right}: ‘f is a tree iff ‘left and ‘right are trees.

• Otherwise, ‘f is not a tree.

For a tree normalized large function ‘f, ‘f is deep computational.

\textbf{Proof}.

• By induction on ‘f.
• Holds if \( f = \text{Constant.Norm.Compu.null}, \ f = \text{Constant.Norm.Compu.zero} \) or \( f = \text{Constant.Norm.Compu.one} \)

• If \( f = \{ \text{left} \ \text{right} \} \): the \text{left} and \text{right} are deep computational (by inductive hypothesis).

For a \text{tree} normalized large function \( f \), \( \text{res}(f) \) is given by one of the following mutually exclusive cases:

• \( \text{Func.Sm.Ext.null} \) if \( f = \text{Constant.Norm.Compu.null} \)

• \( \text{Func.Sm.Ext.zero} \) if \( f = \text{Constant.Norm.Compu.zero} \)

• \( \text{Func.Sm.Ext.one} \) if \( f = \text{Constant.Norm.Compu.one} \)

• \( \{ \text{res(left)}, \text{res(right)} \} \) if \( f = \{ \text{left} \ \text{right} \} \)

\textbf{Proof.}

• Holds if \( f = \text{Constant.Norm.Compu.null}, \ f = \text{Constant.Norm.Compu.zero} \) or \( f = \text{Constant.Norm.Compu.one} \)

• If \( f = \{ \text{left} \ \text{right} \} \): \( \text{res(\{left \ \text{right}\})} = \text{res(\text{ext(\{left \ \text{right}\})})} = \text{res(\text{ext(left) \ \text{ext(right)})})} = \{ \text{res(left)}, \text{res(right)} \} \).

For a \text{tree} normalized large function \( f \), \( \text{res}(f) \) is a \text{tree}.

\textbf{Proof.}

• By induction on \( f \).

• Holds if \( f = \text{Constant.Norm.Compu.null}, \ f = \text{Constant.Norm.Compu.zero} \) or \( f = \text{Constant.Norm.Compu.one} \)

• If \( f = \{ \text{left} \ \text{right} \} \): \( \text{res(left)} \) and \( \text{res(right)} \) are trees (by inductive hypothesis).
  \( \{ \text{res(left)}, \text{res(right)} \} \) is a tree. \( \text{res(\{left \ \text{right}\})} \) is a tree.

For a \text{tree} normalized large function \( f \), \( f \) is unchanging.

\textbf{Proof.}

• By induction on \( f \).

• If ‘f = {‘left ‘right}: ‘left and ‘right are unchanging (by inductive hypothesis).
  ‘ext(‘left) and ‘ext(‘right) are unchanging. {‘ext(‘left) ‘ext(‘right)} is unchanging.
  ‘ext(‘left ‘right)) is unchanging. {‘left ‘right) is unchanging.

For a tree tagged small function extension ‘x, the normal form of ‘x (a tree normalized large function), denoted by ‘norm(‘x), is defined by recursion on ‘x:

• ‘Constant.Norm.Compu.null if ‘x = ‘Func.Sm.Ext.null

• ‘Constant.Norm.Compu.zero if ‘x = ‘Func.Sm.Ext.zero

• ‘Constant.Norm.Compu.one if ‘x = ‘Func.Sm.Ext.one

• {‘norm(‘left(‘x)) ‘norm(‘right(‘x))} if ‘x is a pair tagged small function extension

For a tree tagged small function extension ‘x, ‘res(‘norm(‘x)) = ‘x.

Proof.

• By induction on ‘x.


  • If ‘x is a pair tagged small function extension: ‘norm(‘x) = {‘norm(‘left(‘x)) ‘norm(‘right(‘x))}. ‘res(‘norm(‘x)) = {‘res(‘norm(‘left(‘x))), ‘res(‘norm(‘right(‘x)))} = {‘left(‘x), ‘right(‘x)} (by induction hypothesis). {‘left(‘x), ‘right(‘x)} = ‘x.

For a normalized large function ‘f, the normalized result of ‘f, denoted by ‘resNorm(‘f), is ‘norm(‘res(‘f)) if ‘res(‘f) is a tree; and ‘null otherwise.

For a deep computational normalized large function ‘f, ‘resNorm(‘f) is computable.

For a tree normalized large function ‘f, ‘resNorm(‘f) = ‘f.
Proof.

• \( \text{res}(f) \) is a tree. \( \text{resNorm}(f) = \text{norm}(\text{res}(f)) \).

• By induction on \( f \).


• If \( f = \text{Constant.Norm.Compu.one} \): \( \text{res}(\text{Constant.Norm.Compu.one}) = \text{Func.Sm.Ext.one} \). \( \text{norm}(\text{Func.Sm.Ext.one}) = \text{Constant.Norm.Compu.one} \).

• If \( f = \{\left \left \right \right \} \): \( \left \left \right \right \) are trees. \( \text{res}(\text{left}) \) and \( \text{res}(\text{right}) \) are trees.
  \( \text{resNorm}(\text{left}) = \text{norm}(\text{res}(\text{left})) \) and \( \text{resNorm}(\text{right}) = \text{norm}(\text{res}(\text{right})) \).
  \( \text{res}(f) = \{\text{res}(\text{left}), \text{res}(\text{right})\}. \text{norm}(\text{res}(f)) = \{\text{norm}(\text{left}(\text{res}(f))) \text{norm}(\text{right}(\text{res}(f)))) = \{\text{left} \text{right} \} \) (by induction hypothesis).

For a normalized large function \( f \), the \textbf{computed} of \( f \), denoted by \text{computed}(f), is \( \text{resNorm}(f) \) if \( f \) is deep computational; and \text{null} otherwise.

For a normalized large function \( f \), \text{computed}(f) is computable.

For a normalized large function \( f \), \text{computed}(f) = \text{null} iff \( f \) is not deep computational, or \( \text{res}(f) \) is not a tree.

For a \textit{tree} normalized large function \( f \), \text{computed}(f) = \( f \).

\begin{proof}
  \( f \) is deep computational. \text{computed}(f) = \text{resNorm}(f).
\end{proof}

In NummSquared, the computed of a normalized large function embodies the concept of reduction.

In future, the definition of \( \text{norm}(x) \) may be extended to the case where \( x \) includes rule tagged small function extensions. To do so seems to simply require including more syntactic information in the semantics so that rule tagged small function extensions generated from computation may be transformed into one of the following normal forms:

• \( \text{Constant.Norm.Compu.Null.set} \)

• \( \text{Constant.Norm.Compu.Nuro.set} \)
• ‘Constant.Norm.Compu.Leaf.set

• ‘Constant.Norm.Compu.Tree.set

• a dependent sum computational normalized combination

• a dependent product computational normalized combination

• a Curry computational normalized combination

However, the present definition of ‘norm(x) is sufficient for software where the output is a tree (which is typical). Of course, nothing in the present definition of ‘norm(x) prevents rule tagged small function extensions from being used in the computation of the output, provided they are not present in the output itself. Also, as is demonstrated below, the present definition of ‘norm(x) is even sufficient for macros performing syntactic manipulation of normalized large functions, and for manipulating proofs.

8.4 NORMAL FORM OF A NATURAL NUMBER

For a natural number ‘m, the normal form of ‘m (a tree normalized large function), denoted by ‘norm(m), is defined by recursion on ‘m:

• ‘Constant.Norm.Compu.zero if ‘m = 0

• ‘Constant.Norm.Compu.one if ‘m = 1

• {‘norm(m - 1) ‘Constant.Norm.Compu.null} if ‘m ≥ 2

8.5 QUOTED OF A NORMALIZED LARGE FUNCTION

Because NummSquared is variable-free, quotation is very easy. The quoted of a normalized large function is a tree normalized large function containing a tag and a list of children.

For a natural number ‘tag, and a normalized large function ‘children, the tree of ‘tag and ‘children, denoted by ‘tree(‘tag, ‘children), is {‘norm(‘tag) ‘children}.

For a natural number ‘tag, and a tree normalized large function ‘children, ‘tree(‘tag, ‘children) is a tree.
For a normalized large function \( f \), the tag of \( f \), denoted by \( \text{tag}(f) \), is given by one of the following mutually exclusive cases:

- 0 if \( f = \text{Constant.Norm.Compu.i} \)
- 1 if \( f = \text{Constant.Norm.Compu.null} \)
- 2 if \( f = \text{Constant.Norm.Compu.zero} \)
- 3 if \( f = \text{Constant.Norm.Compu.one} \)
- 4 if \( f = \text{Constant.Norm.Compu.Null.set} \)
- 5 if \( f = \text{Constant.Norm.Compu.Nuro.set} \)
- 6 if \( f = \text{Constant.Norm.Compu.Leaf.set} \)
- 7 if \( f = \text{Constant.Norm.Compu.Tree.set} \)
- 8 if \( f = \text{Constant.Norm.Compu.Null} \)
- 9 if \( f = \text{Constant.Norm.Compu.Pair} \)
- 10 if \( f = \text{Constant.Norm.Compu.dom} \)
- 11 if \( f = \text{Constant.Norm.Noncompu.eq} \)
- 12 if \( f \) is a large composition computational normalized combination
- 13 if \( f \) is a small composition computational normalized combination
- 14 if \( f \) is a pair computational normalized combination
- 15 if \( f \) is a dependent sum computational normalized combination
- 16 if \( f \) is a dependent product computational normalized combination
- 17 if \( f \) is a Curry computational normalized combination
- 18 if \( f \) is an if-then-else computational normalized combination
- 19 if \( f \) is a recursion computational normalized combination
- 20 if \( f \) is a Hilbert non-computational normalized combination
For a normalized constant `c`, the quoted of `c` (a tree normalized large function), denoted by `quoted(c)`, is `tree(tag(c), "l[\]`).

For a normalized large function `f`, the quoted of `f` (a tree normalized large function), denoted by `quoted(f)`, is defined by recursion on `f`:

- as above if `f` is a normalized constant
- `tree(tag(f), "l[quoted('outer') quoted('inner')]`) if `f = ['outer 'inner]`
- `tree(tag(f), "l[quoted('called') quoted('arg')]`) if `f = ('called 'arg)`
- `tree(tag(f), "l[quoted('left') quoted('right')]`) if `f = {'left 'right}`
- `tree(tag(f), "l[quoted('family')]`) if `f = s.d[family]`
- `tree(tag(f), "l[family])`) if `f = p.d[family]`
- `tree(tag(f), "l[uncurry 'quoted('restrictor)) if `f = c[uncurry 'restrictor]`
- `tree(tag(f), "l[ifP 'thenP 'elseP])`) if `f = ifP 'thenP 'elseP]`
- `tree(tag(f), "l[start 'step])`) if `f = r[start 'step]`
- `tree(tag(f), "l[pred])`) if `f = h[']pred]

8.6 UNQUOTED OF A NORMALIZED LARGE FUNCTION

For a normalized large function `f`, the unquoted of `f`, denoted by `unquoted(f)`, is the normalized large function `g` such that `quoted(g) = f` if such exists; and `null` otherwise.

For a normalized large function `f`, `unquoted(f)` is computable.

For a normalized large function `f`, `f` is quoted iff `unquoted(f) ≠ null`.

For a normalized large function `f`, `f` is quoted iff there exists a normalized large function `g` such that `quoted(g) = f`.

For a normalized large function `f`, if `f` is quoted, then `f` is a tree.
8.7 MACRO EXPANDED

Macro expansion combines quotation, computation and unquotation to perform syntactic manipulation of normalized large functions.

For a list \( l = \langle 'x_0, 'x_1, ..., 'x_{m - 1} \rangle \) of Func.Lg.Norm, the quoted of \( l \), denoted by \( \text{quoted}(l) \), is \( \langle \text{quoted('x_0)}, \text{quoted('x_1)}, ..., \text{quoted('x_{m - 1})} \rangle \).

For a list \( l \) of Func.Lg.Norm, \( \text{quoted}(l) \) is a tree.

For a normalized large function \( f \), and a list \( l \) of Func.Lg.Norm, the macro pre-expanded of \( f \) at \( l \), denoted by \( \text{macroPreexpanded}(f, l) \), is \( [f \text{ quoted}(l)] \).

For a normalized large function \( f \), and a list \( l \) of Func.Lg.Norm, \( \text{macroPreexpanded}(f, l) \) is deep computational iff \( f \) is deep computational.

For a normalized large function \( f \), and a list \( l \) of Func.Lg.Norm, \( \text{macroExpanded}(f, l) \) is computable.

For a normalized large function \( f \), and a list \( l \) of Func.Lg.Norm, \( \text{macroExpanded}(f, l) = \text{null} \) iff \( \text{computed}(\text{macroPreexpanded}(f, l)) = \text{null} \); and \( \text{unquoted}(\text{computed}(\text{macroPreexpanded}(f, l))) \) otherwise.

For a normalized large function \( f \), and a list \( l \) of Func.Lg.Norm, \( \text{macroExpanded}(f, l) \) is computable.

8.8 SUBSTITUTION AND SUBSTITUTION THEOREM

Because NummSquared is variable-free, substitution is very easy.

For normalized large functions \( f, g, x \) and \( y \), the predicate \( f \) substitutes to \( g \) replacing \( x \) by \( y \), denoted by \( \text{subst}(f, g, x, y) \), is defined by recursion on \( f \). For normalized large functions \( f, g, x \) and \( y \), \( \text{subst}(f, g, x, y) \) is true iff at least one of the following holds:

- \( f = x \) and \( g = y \).
- \( f \) and \( g \) are normalized constants and \( f = g \).
- \( f = [\text{outerF} \ 'innerF], g = [\text{outerG} \ 'innerG], \text{subst}(\text{outerF}, \text{outerG}, x, y) \), and \( \text{subst}(\text{innerF}, \text{innerG}, x, y) \).
• The other normalized combination cases are similar and are omitted.

In substitution, replacement of an occurrence of ‘x by ‘y is optional.

The substitution theorem: For normalized large functions ‘f, ‘g, ‘x and ‘y, if ‘ext(‘x) = ‘ext(‘y) and ‘subst(‘f, ‘g, ‘x, ‘y), then ‘ext(‘f) = ‘ext(‘g).

Proof.

• By induction on ‘f.

• If ‘f = ‘x and ‘g = ‘y: ‘ext(‘f) = ‘ext(‘x). ‘ext(‘g) = ‘ext(‘y).

• Holds if ‘f and ‘g are normalized constants and ‘f = ‘g.

• If ‘f = ['outerF 'innerF], ‘g = ['outerG 'innerG], ‘subst('outerF, ‘outerG, ‘x, ‘y), and ‘subst('innerF, ‘innerG, ‘x, ‘y): ‘ext('outerF) = ‘ext('outerG) and ‘ext('innerF) = ‘ext('innerG) (by inductive hypothesis). ‘ext('f) = ['ext('outerF) 'ext('innerF)]. ‘ext('g) = ['ext('outerG) 'ext('innerG)].

• The other normalized combination cases are similar and are omitted.

8.9 COMMENTS

A comment contains a list of ‘Nat. Recall that natural numbers in the range 0-1114111 are Unicode code points. Natural numbers above this range may be interpreted in whatever way you wish.

In the concrete syntax, a comment is written between ‘ and ‘. A comment containing 0 may be omitted in the concrete syntax. In the future, more details will be provided.

8.10 IDENTIFIERS

An identifier start character is an uppercase letter character (A-Z), a lowercase letter character (a-z), or one of the following:

! & + - / <= => \ ^ |
A digit character is one of 0-9.

An identifier continue character is an identifier start character or a digit character. Let 'Chr.Ident.Cont be the language of all identifier continue characters.

A simple identifier contains <'start, 'conts> where 'start is an identifier start character and 'conts is a list of 'Chr.Ident.Cont. Let 'Ident.Simp be the language of all simple identifiers.

A simple identifier containing <'start, 'conts> where 'conts = l<'x0, 'x1, ..., 'xm-1> is written in the concrete syntax as follows:

'\texttt{start} \texttt{x0 x1...xm-1}

An identifier contains a non-empty list of 'Ident.Simp. Let 'Ident be the language of all identifiers.

An identifier containing l<'x0, 'x1, ..., 'xm-2, 'xm-1> is written in the concrete syntax as follows:

'\texttt{x0 . x1.....xm-2 . xm-1}

In NummSquared, identifiers are hierarchical names. However, an object is always referenced by its entire identifier. Therefore, careful choice of short prefixes and suffixes is encouraged.

8.11 LARGE FUNCTIONS

Large functions are just syntactic sugar for normalized large functions.

A natural number primitive contains a natural number. In the concrete syntax, a natural number primitive is written in decimal notation. In the future, more details will be provided.

A character primitive contains a natural number. Recall that natural numbers in the range 0-1114111 are Unicode code points. Natural numbers above this range may be interpreted in whatever way you wish.

In the concrete syntax, a character primitive is written between ‘ and \texttt{(not )}. In the future, more details will be provided.

A string primitive contains a list of `Nat.

In the concrete syntax, a string primitive is written between " and \texttt{(not ")} In the future, more details will be provided.
A **primitive** is exactly one of the following:
- a natural number primitive
- a character primitive
- a string primitive

A **computational non-normalized constant** is exactly one of the following:
- the left computational non-normalized constant, ‘Constant.Nonnorm.Compu.left
- the right computational non-normalized constant, ‘Constant.Nonnorm.Compu.right
- the confirmation with null computational non-normalized constant, ‘Constant.Nonnorm.Compu.conf.n
- the negation with null computational non-normalized constant, ‘Constant.Nonnorm.Compu.not.n
- the one predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.One
- the leaf predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Leaf
- the simple predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Simp
• the rule predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Rule

• the tree predicate step pair computational non-normalized constant, ‘Constant.Nonnorm.Compu.Tree.step.pair

• the tree predicate step computational non-normalized constant, ‘Constant.Nonnorm.Compu.Tree.step

• the tree predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Tree

• the result computational non-normalized constant, ‘Constant.Nonnorm.Compu.res

• the nuro set result computational non-normalized constant, ‘Constant.Nonnorm.Compu.Nuro.set.res

• the tree set result computational non-normalized constant, ‘Constant.Nonnorm.Compu.Tree.set.res

• the dependent sum result left computational non-normalized constant, ‘Constant.Nonnorm.Compu.s.d.res.left

• the dependent sum result right computational non-normalized constant, ‘Constant.Nonnorm.Compu.s.d.res.right

• the dependent sum result computational non-normalized constant, ‘Constant.Nonnorm.Compu.s.d.res

• the dependent product result rule uncurry computational non-normalized constant, ‘Constant.Nonnorm.Compu.p.d.res.rule.uncurry

• the dependent product result rule computational non-normalized constant, ‘Constant.Nonnorm.Compu.p.d.res.rule

• the dependent product result computational non-normalized constant, ‘Constant.Nonnorm.Compu.p.d.res

• the negation computational non-normalized constant, ‘Constant.Nonnorm.Compu.not
• the implication with null computational non-normalized constant, 'Constant.Nonnorm.Compu.imp.n

• the implication computational non-normalized constant, 'Constant.Nonnorm.Compu.imp

The above computational non-normalized constants are written in the concrete syntax as follows:

~left
~right
~conf.n
~not.n
~Null.to.Zero
~Zero
~One
~Nuro
~Boo
~Leaf
~Simp
~Rule
~Tree.step.pair
~Tree.step
~Tree
~res
~Nuro.set.res
~Tree.set.res
~s.d.res.left
~s.d.res.right
~s.d.res
~p.d.res.rule.uncurry
~p.d.res.rule
~p.d.res
~not
~imp.n
~imp
A **non-computational non-normalized constant** is exactly one of the following:

- the small universal quantification non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.all.sm
- the equal pairs non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.eq.pair
- the equal results at non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.eq.res.at
- the equal results non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.eq.res
- the equal domain results non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.eq.dom.res
- the equal both results non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.eq.both.res
- the equals right-hand-side non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.eq.rhs
- the not equals non-computational non-normalized constant, ‘Constant.Nonnorm.Noncompu.not.eq

The above non-computational non-normalized constants are written in the concrete syntax as follows:

```plaintext
~all.sm
~=.pair
~=.res.at
~=.res
~=.dom.res
~=.both.res
~=.rhs
~not.=
```

A **non-normalized constant** is exactly one of the following:

- a computational non-normalized constant
• a non-computational non-normalized constant

A constant is exactly one of the following:
• a normalized constant
• a non-normalized constant

Large functions are defined inductively. Let ‘Func.Lg be the language of all large functions.
A large function is exactly one of the following:
• a primitive
• a constant
• a combination
• a global name
• a local name
• a computation
• a quotation
• an unquotation
• a macro expansion

Combinations, computations, quotations, unquotations and macro expansions are written in the concrete syntax in the same way as in the informal part.
A combination is exactly one of the following:
• a computational combination
• a non-computational combination

A computational combination is exactly one of the following:
• a large composition computational combination
• a small composition computational combination
• a tuple computational combination
• a list computational combination
• a dependent sum computational combination
• a dependent product computational combination
• a Curry computational combination
• an if-then-else computational combination
• a recursion computational combination
• a restrict computational combination
• a restrict to range computational combination
• a Curry augmented uncurry computational combination
• a Curry augmented computational combination
• a Curry result computational combination
• a recursion on domain computational combination
• a recursion on range computational combination
• a recursion step computational combination
• a recursion right-hand-side computational combination

A large composition computational combination contains <outer, ‘inners> where ‘outer is a large function, and ‘inners is a non-empty list of Func.Lg. For a large function ‘outer, and a list ‘inners = l<‘x0, ‘x1, ..., ‘xm-1> of Func.Lg such that ‘m ≥ 1, let [outer ‘x0 ‘x1 ... ‘xm-1] be the large composition computational combination containing <outer, ‘inners>.

A small composition computational combination contains a list ‘calledAndArgs of Func.Lg of length ≥ 2. For a list ‘calledAndArgs = l<‘x0, ‘x1, ..., ‘xm-1> of Func.Lg such that ‘m ≥ 2, let (‘x0 ‘x1 ... ‘xm-1) be the small composition computational combination containing ‘calledAndArgs.
A **tuple computational combination** contains a list 'components of 'Func.Lg of length $\geq 2$. For a list 'components = l< 'x0, 'x1, ..., 'xm-1 > of 'Func.Lg such that 'm $\geq 2$, let ('x0 'x1 ... 'xm-1) be the tuple computational combination containing 'components.

A **list computational combination** contains a list 'elements of 'Func.Lg. For a list 'elements = l< 'x0, 'x1, ..., 'xm-1 > of 'Func.Lg, let l['x0 'x1 ... 'xm-1] be the list computational combination containing 'elements.

A **dependent sum computational combination** contains 'family where 'family is a large function. For a large function 'family, let s.d['family] be the dependent sum computational combination containing 'family.

A **dependent product computational combination** contains 'family where 'family is a large function. For a large function 'family, let p.d['family] be the dependent product computational combination containing 'family.

A **Curry computational combination** contains <'uncurry, 'restrictor> where 'uncurry and 'restrictor are large functions. For large functions 'uncurry and 'restrictor, let c['uncurry 'restrictor] be the Curry computational combination containing <'uncurry, 'restrictor>.

An **if-then-else computational combination** contains <'ifP, 'thenP, 'elseP> where 'ifP, 'thenP and 'elseP are large functions. For large functions 'ifP, 'thenP and 'elseP, let ite['ifP 'thenP 'elseP] be the if-then-else computational combination containing <'ifP, 'thenP, 'elseP>.

A **recursion computational combination** contains <'start, 'step> where 'start and 'step are large functions. For large functions 'start and 'step, let r['start 'step] be the recursion computational combination containing <'start, 'step>.

A **restrict computational combination** contains 'unrestrict where 'unrestrict is a large function. For a large function 'unrestrict, let restrict['unrestrict] be the restrict computational combination containing 'unrestrict.

A **restrict to range computational combination** contains 'unrestrict where 'unrestrict is a large function. For a large function 'unrestrict, let restrict.ran['unrestrict] be the restrict to range computational combination containing 'unrestrict.

A **Curry augmented uncurry computational combination** contains <'uncurry, 'augmentor> where 'uncurry and 'augmentor are large functions. For large functions 'uncurry and 'augmentor, let c.aug.uncurry['uncurry 'augmentor] be the Curry augmented uncurry computational combination containing <'uncurry, 'augmentor>.

A **Curry augmented computational combination** contains <'uncurry, 'restrictor, 'augmentor> where 'uncurry, 'restrictor and 'augmentor are large functions. For large
functions ‘uncurry, ‘restrictor and ‘augmentor, let \( \sim c.\text{aug}'\text{uncurry} '\text{restrictor} '\text{augmentor} \) be the Curry augmented computational combination containing <‘uncurry, ‘restrictor, ‘augmentor>.

A **Curry result computational combination** contains ‘uncurry where ‘uncurry is a large function. For a large function ‘uncurry, let \( \sim c.\text{res}'\text{uncurry} \) be the Curry result computational combination containing ‘uncurry.

A **recursion on domain computational combination** contains <‘start, ‘step> where ‘start and ‘step are large functions. For large functions ‘start and ‘step, let \( \sim r.\text{dom}'\text{start} '\text{step} \) be the recursion on domain computational combination containing <‘start, ‘step>.

A **recursion on range computational combination** contains <‘start, ‘step> where ‘start and ‘step are large functions. For large functions ‘start and ‘step, let \( \sim r.\text{ran}'\text{start} '\text{step} \) be the recursion on range computational combination containing <‘start, ‘step>.

A **recursion step computational combination** contains <‘start, ‘step> where ‘start and ‘step are large functions. For large functions ‘start and ‘step, let \( \sim r.\text{step}'\text{start} '\text{step} \) be the recursion step computational combination containing <‘start, ‘step>.

A **recursion right-hand-side computational combination** contains <‘start, ‘step> where ‘start and ‘step are large functions. For large functions ‘start and ‘step, let \( \sim r.\text{rhs}'\text{start} '\text{step} \) be the recursion right-hand-side computational combination containing <‘start, ‘step>.

A **non-computational combination** is exactly one of the following:

- a Hilbert non-computational combination
- an existential quantification non-computational combination
- a not universal quantification non-computational combination
- a universal quantification non-computational combination
- a unary universal quantification non-computational combination
- an inductive domain hypothesis non-computational combination
- an inductive range hypothesis non-computational combination
- an inductive case at non-computational combination
- an inductive case non-computational combination
A Hilbert non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘h[‘pred] be the Hilbert non-computational combination containing ‘pred.

An existential quantification non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘exist[‘pred] be the existential quantification non-computational combination containing ‘pred.

A not universal quantification non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘not.all[‘pred] be the not universal quantification non-computational combination containing ‘pred.

A universal quantification non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘all[‘pred] be the universal quantification non-computational combination containing ‘pred.

A unary universal quantification non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘all.una[‘pred] be the unary universal quantification non-computational combination containing ‘pred.

An inductive domain hypothesis non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘induc.hyp.dom[‘pred] be the inductive domain hypothesis non-computational combination containing ‘pred.

An inductive range hypothesis non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘induc.hyp.ran[‘pred] be the inductive range hypothesis non-computational combination containing ‘pred.

An inductive case at non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘induc.case.at[‘pred] be the inductive case at non-computational combination containing ‘pred.

An inductive case non-computational combination contains ‘pred where ‘pred is a large function. For a normalized large function ‘pred, let ‘induc.case[‘pred] be the inductive case non-computational combination containing ‘pred.

A global name contains an identifier. Global names are used to reference definitions. A global name containing ‘id is written in the concrete syntax as ‘id.

A local name contains an identifier. Local names are used to reference local tuple accessors. Local names are not variables. A local name containing ‘id is written in the concrete syntax as follows:
Global and local names are easily distinguished in the concrete syntax and therefore do not conflict.

A computation contains a large function ‘called. For a large function ‘called, let `C[‘called] be the computation containing ‘called.

A quotation contains a large function ‘unquoted. For a large function ‘unquoted, let `Q[‘unquoted] be the quotation containing ‘unquoted.

An unquotation contains a large function ‘quoted. For a large function ‘quoted, let `UQ[‘quoted] be the unquotation containing ‘quoted.

A macro expansion contains <‘called, ‘args> where ‘called is a large function, and ‘args is a list of ‘Func.Lg. For a large function ‘called, and a list ‘args = l<‘x0, ‘x1, ..., ‘x_m-1> of ‘Func.Lg, let #‘called[‘x0 ‘x1 ... ‘x_m-1] be the macro expansion containing <‘called, ‘args>. As the syntax for macro expansion suggests, ‘called (a large function) is used to combine the elements of ‘args (also large functions). ‘called abstracts over all large functions, but can perform only syntactic manipulation of ‘args. For macros, syntactic manipulation is often sufficient.

This concludes the inductive definition.

8.12 DEFINITIONS, DEFINITION LISTS, MODULES AND ABSTRACT PROGRAMS

An identifier list is a list of ‘Ident.

A local tuple accessor list contains an identifier list of length ≥ 2. Each identifier is the name of a local tuple accessor.

A local tuple accessor list containing l<‘id0, ‘id1, ..., ‘id_m-1> is written in the concrete syntax as follows:

{ `%id_m-1 ... `%id1 `%id0}

There is a reversal between the abstract syntax and the concrete syntax.

A local tuple accessor checker contains <‘lis, ‘onFail> where ‘lis is a local tuple accessor list, and ‘onFail is a large function.

A local tuple accessor checker containing <‘lis, ‘onFail> is written in the concrete syntax as follows:
'lis \ 'onFail

If 'onFail = ‘Constant.Norm.Compu.null, \ 'onFail may be omitted in the concrete syntax. ('onFail = ‘Constant.Norm.Compu.null is the default.)

For a local tuple accessor checker 'checker containing <'lis, 'onFail>, the list of 'checker, denoted by 'lis('checker), is 'lis.

For a local tuple accessor checker 'checker containing <'lis, 'onFail>, the on fail of 'checker, denoted by 'onFail('checker), is 'onFail.

A local tuple accessor descriptor is exactly one of the following:

- 0
- a local tuple accessor checker

The local tuple accessor descriptor 0 is omitted in the concrete syntax.

A definition contains <'comment, 'name, 'accessTupleLocDesc, 'rhs> where 'comment is a comment, 'name is an identifier, 'accessTupleLocDesc is a local tuple accessor descriptor, and 'rhs is a large function. Let 'Def be the language of all definitions.

A definition containing <'comment, 'name, 'accessTupleLocDesc, 'rhs> is written in the concrete syntax as follows:

'comment
'name 'accessTupleLocDesc = 'rhs;

For a definition 'def containing <'comment, 'name, 'accessTupleLocDesc, 'rhs>, the name of 'def, denoted by 'name('def), is 'name.

For a definition 'def containing <'comment, 'name, 'accessTupleLocDesc, 'rhs>, the right-hand-side of 'def, denoted by 'rhs('def), is 'rhs.

A definition list contains a list of 'Def.

A definition list containing l<'def0, 'def1, ..., 'defm-1 > is written in the concrete syntax as follows:

'defm-1
.
.
.
'def1
'def0
There is a *reversal* between the abstract syntax and the concrete syntax.

For definition lists `dl0` containing `'l0` and `dl1` containing `'l1`, the **concatenation** of `dl0` and `dl1`, denoted by `dl0 + dl1`, is the definition list containing `'l0 + `'l1`.

A **module** contains `<'comment, 'name, 'defLis>` where `comment` is a comment, `name` is an identifier, and `defLis` is a definition list. Let `Modu` be the language of all modules.

A module containing `<'comment, 'name, 'defLis>` is written in the concrete syntax as follows:

```plaintext
'comment
'name {
'defLis
}
```

A NummSquared module serves only as a logical grouping and a place to attach an overview comment. The name of the module has no effect on the names of the definitions in the module. All definitions in a module can be referenced from later modules, without qualifying by the module name. In future, NummSquared modules may serve additional purposes.

For a module `modu` containing `<comment, name, defLis>`, the **name** of `modu`, denoted by `name(modu)`, is `name`.

For a module `modu` containing `<comment, name, defLis>`, the **definition list** of `modu`, denoted by `defLis(modu)`, is `defLis`.

An **abstract program** contains a list of `Modu`.

An abstract program containing `<modu0, modu1, ..., modu_m-1>` is written in the concrete syntax as follows:

```plaintext
'modu_m-1
.
.
.
'modu1
'modu0
```

There is a *reversal* between the abstract syntax and the concrete syntax.

For an abstract program `prog` containing `<modu0, modu1, ..., modu_m-1>`, the **module name list** of `prog`, denoted by `moduNameLis(prog)`, is `<name(modu0), name(modu1), ..., name(modu_m-1)>`. 
For an abstract program `prog` containing `<modu_0, modu_1, ..., modu_{m-1}>`, the definition list of `prog`, denoted by `defLis(prog)`, is `defLis(modu_0) + defLis(modu_1) + ... + defLis(modu_{m-1}).

### 8.13 CONTEXTS

A normalized definition contains `<name, rhs>` where `name` is an identifier and `rhs` is a normalized large function. Let `Def.Norm` be the language of all normalized definitions.

For a normalized definition `def` containing `<name, rhs>`, the name of `def`, denoted by `name(def)`, is `name`.

For a normalized definition `def` containing `<name, rhs>`, the right-hand-side of `def`, denoted by `rhs(def)`, is `rhs`.

A global context contains a list of `Def.Norm`.

For a global context `cg` containing `l`, and an identifier `id`, let `search(cg, id)` be the search first data for a normalized definition `def` such that `name(def) = id` in `l`.

For a global context `cg` containing `l`, and an identifier `id`, let `cg(id)` be `null` if `search(cg, id) = null`; and `rhs(search(cg, id))` otherwise.

For a global context `cg` containing `l`, `cg` is valid iff, for each identifier `id`, the property of being normalized definition `def` such that `name(def) = id` is not duplicitous in `l`.

For an identifier list `l`, and an identifier `id`, let `l(id)` be the search first index for an identifier `id0` such that `id0 = id` in `l`.

For an identifier list `l`, `l` is valid iff, for each identifier `id`, the property of being an identifier `id0` such that `id0 = id` is not duplicitous in `l`.

For a local tuple accessor list `accessors` containing `l`, let `len(accessors) = len(l)`.

For a local tuple accessor list `accessors` containing `l`, and an identifier `id`, let `accessors(id) = l(id)`.

For a local tuple accessor list `accessors` containing `l`, `accessors is valid` iff `l` is valid.

A local context is exactly one of the following:

- 0

- a local tuple accessor list
For a local context ‘cl, let ‘len(‘cl) be given by one of the following mutually exclusive cases:

- 0 if ‘cl = 0
- as above if ‘cl is a local tuple accessor list

For a local context ‘cl, and an identifier ‘id, let ‘cl(‘id) be given by one of the following mutually exclusive cases:

- ‘null if ‘cl = 0
- as above if ‘cl is a local tuple accessor list

For a local context ‘cl, the property of ‘cl being valid is given by one of the following mutually exclusive cases:

- If ‘cl = 0: ‘cl is valid.
- as above if ‘cl is a local tuple accessor list

A global context and a local context are needed to define the normal form of a large function ‘f. The normal form of ‘f is either a normalized large function or ‘null (indicating that ‘f is invalid).

A normalized local tuple accessor checker contains <‘lis, ‘onFail> where ‘lis is a local tuple accessor list, and ‘onFail is a normalized large function.

For a normalized local tuple accessor checker ‘checker containing <‘lis, ‘onFail>, the list of ‘checker, denoted by ‘lis(‘checker), is ‘lis.

For a normalized local tuple accessor checker ‘checker, let ‘len(‘checker) = ‘len(‘lis(‘checker)).

For a normalized local tuple accessor checker ‘checker containing <‘lis, ‘onFail>, the on fail of ‘checker, denoted by ‘onFail(‘checker), is ‘onFail.

For a normalized local tuple accessor checker ‘checker, ‘checker is valid iff ‘lis(‘checker) is valid.

A normalized local tuple accessor descriptor is exactly one of the following:

- 0
- a normalized local tuple accessor checker
For a normalized local tuple accessor descriptor `desc`, `len(desc)` be given by one of the following mutually exclusive cases:

- 0 if `desc = 0`
- as above if `desc` is a normalized local tuple accessor checker

For a normalized local tuple accessor descriptor `desc`, the property of `desc` being valid is given by one of the following mutually exclusive cases:

- If `desc = 0`: `desc` is valid.
- as above if `desc` is a normalized local tuple accessor checker

For a normalized local tuple accessor descriptor `desc`, the local context of `desc`, denoted by `contextLoc(desc)` is given by one of the following mutually exclusive cases:

- 0 if `desc = 0`
- `lis(desc)` if `desc` is a normalized local tuple accessor checker

For a valid normalized local tuple accessor descriptor `desc`, `contextLoc(desc)` is valid.

### 8.14 NORMAL FORM OF A PRIMITIVE

For a natural number primitive `primNat` containing `m`, the normal form of `primNat`, denoted by `norm(primNat)`, is `norm(m)`.

For a character primitive `primChr` containing `m`, the normal form of `primChr`, denoted by `norm(primChr)`, is `norm(m)`.

For a string primitive `primStr` containing `<x₀, x₁, ..., xₘ₋₁>`, the normal form of `primStr`, denoted by `norm(primStr)`, is `l[norm(x₀) norm(x₁) ... norm(xₘ₋₁)]`.

### 8.15 NORMAL FORM OF A NORMALIZED CONSTANT

For a normalized constant `c`, the normal form of `c`, denoted by `norm(c)`, is `c`.

### 8.16 NORMAL FORM OF A GLOBAL NAME

For a global context `cg`, and a global name `ng` containing `id`, the normal form in `cg` of `ng`, denoted by `norm(cg, ng)`, is `cg(id)`.
8.17 PSEUDO-NUMMSQUARED

In the informal part, for ease of reading, pseudo-NummSquared (similar to Numm-Squared concrete syntax) henceforth represents normalized large functions. For example, in pseudo-NummSquared, a NummSquared identifier represents the corresponding normalized large function. Of course, pseudo-NummSquared cannot include constructs whose normal forms have not yet been defined.

Pseudo-NummSquared may include informal identifiers (for example, ‘x, ‘X, ‘X0 and ‘A.x), which are written as follows:

‘x
‘X
‘X0
‘A.x

To obtain the normalized large functions represented by pseudo-NummSquared, informal identifiers are replaced by the things they represent.

In pseudo-NummSquared, confusion between informal identifiers and Numm-Squared comments is unlikely to occur.

Informal identifiers are distinct from NummSquared identifiers.

8.18 NORMAL FORM OF A LOCAL NAME

~left =
~ite[
  ~Pair
  (~i 0)
  ~null
];

Henceforth, for each definition in pseudo-NummSquared (for example, the definition with name ~left), there is an implicit definition associating the corresponding informal identifier with the corresponding large function extension (for example, ‘Func.Lg.Ext.left = ‘ext(~left)).

For a tagged small function extension ‘x, ‘Func.Lg.Ext.left(‘x) = ‘left(~x) if ‘x is a pair tagged small function extension; and ‘Func.Sm.Ext.null otherwise.
For a tagged small function extension 'x, 'Func.Lg.Ext.right('x) = 'right('x) if 'x is a pair tagged small function extension; and 'Func.Sm.Ext.null otherwise.

For a natural number 'm, if 'm = 0:

~left('m) = ~left;

For a natural number 'm, if 'm = 'n + 1:

~left('m) = [~left('n) ~left];

For a natural number 'm, if 'm = 0:

~right('m) = ~right;

For a natural number 'm, if 'm = 'n + 1:

~right('m) = [~right ~left('n)];

A tuple locator is a pair <'side, 'm> where 'side is a Boolean and 'm is a natural number.

For a tuple locator 'tl = <'side, 'm>, let ~tuple.by.locator('tl) be ~left('m) if 'side = 0; and ~right('m) otherwise.

For natural numbers 'm and 'i such that 'i < 'm, let 'tupleIndexToLocator('m, 'i) be given by one of the following mutually exclusive cases:

• <0, 'm - 2> if 'i = 'm - 1
• <1, 'i> if 'i < 'm - 1

The tuple index 0 designates the rightmost component of the tuple.

For natural numbers 'm and 'i such that 'i < 'm, let ~tuple.by.index('m, 'i) be ~tuple.by.locator('tl) where 'tl = 'tupleIndexToLocator('m, 'i).

For a local context 'cl, and a local name 'nl containing 'id, the normal form in 'cl of 'nl, denoted by 'norm('cl, 'nl), is given by one of the following mutually exclusive cases:
• ‘null if ‘cl(id) = ‘null

• If ‘cl(id) ≠ ‘null: ~tuple.by.index(‘m ‘i) where ‘m = ‘len(cl) and ‘i = ‘cl(id)

8.19 LOCAL TUPLE ACCESSOR CHECK

When a definition includes a local tuple accessor checker ‘checker, the normalized large function being defined automatically checks that its argument is a sufficiently deep tuple. If not, ‘onFail(‘checker) is automatically called.

For a natural number ‘m, if ‘m = 0:

~Tuple(‘m) = ~Pair;

For a natural number ‘m, if ‘m = ‘n + 1:

~Tuple(‘m) = [~Tuple(‘n) ~left];

For a natural number ‘m, and normalized large functions ‘onFail and ‘f:

~Tuple.check(‘m ‘onFail ‘f) =
~ite[
    ~Tuple(‘m)
    ‘f
    ‘onFail
];

For a normalized local tuple accessor checker ‘checker, and a normalized large function ‘f, let ‘addCheck(‘checker, ‘f) be ~Tuple.check(‘m ‘onFail ‘f) where ‘m = ‘len(‘checker) - 2 and ‘onFail = ‘onFail(‘checker).

For a normalized local tuple accessor descriptor ‘desc, and a normalized large function ‘f, ‘addCheck(‘desc, ‘f) is given by one of the following mutually exclusive cases:

• ‘f if ‘desc = 0

• as above if ‘desc is a normalized local tuple accessor checker

In pseudo-NummSquared, when a definition includes a local tuple accessor checker, ‘addCheck is implicitly applied.
\section{NORMAL FORM OF A COMPUTATIONAL NON-NORMALIZED CONSTANT OR COMPUTATIONAL COMBINATION}

For a computational non-normalized constant \( f \), the \textbf{normal form} of \( f \), denoted by \( \text{norm}(f) \), is defined to be the corresponding normalized large function below.

The normal form of a computational combination \( f \) cannot be defined at this point because the normal form of \( f \) depends upon the normal forms of the components of \( f \). Instead, the corresponding combination of \textit{normalized} large functions is defined.

Corresponding combinations of normalized large functions have already been defined for the following:
\begin{itemize}
  \item a large composition computational combination
  \item a small composition computational combination
  \item a tuple computational combination
  \item a list computational combination
  \item a dependent sum computational combination
  \item a dependent product computational combination
  \item a Curry computational combination
  \item an if-then-else computational combination
  \item a recursion computational combination
\end{itemize}
\texttt{~left} and \texttt{~right} have already been defined.

\subsection{CONFIRMATION WITH NULL}

\begin{verbatim}
~conf.n = ~ite[
    ~i
    1
    0
];
\end{verbatim}
For a tagged small function extension ‘x, ‘Func.Lg.Ext.conf.n(’x) is ‘x if ‘x is a leaf small function extension; and ‘Func.Sm.Ext.null otherwise.


8.20.2 NEGATION WITH NULL

~not.n =
~ite[
  ~i
  0
  1
];

For a tagged small function extension ‘x, ‘Func.Lg.Ext.not.n(’x) is given by one of the following mutually exclusive cases:

- ‘Func.Sm.Ext.one if ‘x = ‘Func.Sm.Ext.zero
- ‘Func.Sm.Ext.zero if ‘x = ‘Func.Sm.Ext.one
- ‘Func.Sm.Ext.null if ‘x is not Boolean

8.20.3 NULL TO ZERO

~Null.to.Zero =
~ite[
  ~Null
  0
  ~i
];

8.20.4 KIND PREDICATES

\[ \sim \text{Zero} = [\sim \text{Null.to.Zero} \sim \text{not.n}] ; \]

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.Zero}(x) = \text{Func.Sm.Ext.one} \) if \( x = \text{Func.Sm.Ext.zero} \) and \( \text{Func.Sm.Ext.zero} \) otherwise.

\[ \sim \text{One} = [\sim \text{Null.to.Zero} \sim \text{conf.n}] ; \]

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.One}(x) = \text{Func.Sm.Ext.one} \) if \( x = \text{Func.Sm.Ext.one} \) and \( \text{Func.Sm.Ext.zero} \) otherwise.

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.One}(x) \) is a Boolean, and \( \text{Func.Lg.Ext.One}(x) \) is true iff \( x \) is true.

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.One}(x) = \text{Func.Lg.Ext.conf.n}(\text{Func.Sm.Ext.one}) \) if \( x = \text{Func.Sm.Ext.one} \) and \( \text{Func.Lg.Ext.conf.n}(\text{Func.Sm.Ext.zero}) \) otherwise.

\[ \sim \text{Nuro} = \]
\[ \sim \text{ite}[ \sim \text{Null} \]
\[ 1 \]
\[ \sim \text{Zero} \]
\[ ]; \]

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.Nuro}(x) = \text{Func.Sm.Ext.one} \) if \( x \) is a nuro; and \( \text{Func.Sm.Ext.zero} \) otherwise.

\[ \sim \text{Boo} = \]
\[ \sim \text{ite}[ \sim \text{Zero} \]
\[ 1 \]
\[ \sim \text{One} \]
\[ ]; \]

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.Boo}(x) = \text{Func.Sm.Ext.one} \) if \( x \) is a Boolean; and \( \text{Func.Sm.Ext.zero} \) otherwise.
~Leaf = 
~ite[
    ~Nuro
    1
    ~One
 ];

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Leaf(‘x) = ‘Func.Sm.Ext.one if ‘x is a leaf small function extension; and ‘Func.Sm.Ext.zero otherwise.

~Simp = 
~ite[
    ~Leaf
    1
    ~Pair
 ];

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Simp(‘x) = ‘Func.Sm.Ext.one if ‘x is a simple tagged small function extension; and ‘Func.Sm.Ext.zero otherwise.

~Rule = [~not.n ~Simp];

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Rule(‘x) = ‘Func.Sm.Ext.one if ‘x is a rule tagged small function extension; and ‘Func.Sm.Ext.zero otherwise.

8.20.5 TREE PREDICATE

~Tree.step.pair {%r.dom %r.ran %func} = 
~ite[
    (%r.ran 0)
    [~conf.n (%r.ran 1)]
    0
 ];

For a tagged small function extension ‘x = {%r.dom, ‘r.ran, ‘func}, ‘Func.Lg.Ext.Tree.step.pair(‘x) is given by one of the following mutually exclusive cases:
• 'Func.Lg.Ext.conf.n('r.ran('Func.Sm.Ext.one)) if 'r.ran('Func.Sm.Ext.zero) = 'Func.Sm.Ext.one

• 'Func.Sm.Ext.zero if 'r.ran('Func.Sm.Ext.zero) = 'Func.Sm.Ext.zero

• 'Func.Sm.Ext.null if 'r.ran('Func.Sm.Ext.zero) is not Boolean

~Tree.step {%r.dom %r.ran %func} =
~ite[
  ~Leaf %func
  1
~ite[
  ~Pair %func
  ~Tree.step.pair
0]);

For a tagged small function extension 'x = {'r.dom, 'r.ran, 'func},
'Func.Lg.Ext.Tree.step('x) is given by one of the following mutually exclusive cases:

• 'Func.Sm.Ext.one if 'func is a leaf small function extension

• 'Func.Lg.Ext.Tree.step.pair('x) if 'func is a pair tagged small function extension

• 'Func.Sm.Ext.zero if 'func is a rule tagged small function extension

~Tree = ~r[1 ~Tree.step];

For a tagged small function extension 'x, 'Func.Lg.Ext.Tree('x) is given by one of the following mutually exclusive cases:

• If 'x is a leaf small function extension: 'Func.Lg.Ext.Tree('x) = 'Func.Sm.Ext.one.

• If 'x is a pair tagged small function extension: 'Func.Lg.Ext.Tree('x) is given by one of the following mutually exclusive cases:
  
  – 'Func.Lg.Ext.conf.n('Func.Lg.Ext.Tree('right('x))) if
    'Func.Lg.Ext.Tree('left('x)) = 'Func.Sm.Ext.one
  
  – 'Func.Sm.Ext.zero if 'Func.Lg.Ext.Tree('left('x)) = 'Func.Sm.Ext.zero
  
  – 'Func.Sm.Ext.null if 'Func.Lg.Ext.Tree('left('x)) is not Boolean
• If \( x \) is a rule tagged small function extension: 
  \[
  \text{Func.Lg.Ext.Tree}(x) = \text{Func.Sm.Ext.zero}.
  \]

**Proof.**

• If \( x = \text{Func.Sm.Ext.null} \): 
  \[
  \text{Func.Lg.Ext.Tree}(x) = \text{Func.Sm.Ext.one}.
  \]

• If \( x \neq \text{Func.Sm.Ext.null} \): \( \text{Func.Lg.Ext.Tree}(x) = \text{Func.Lg.Ext.Tree}.step([rDom, rRan, x]) \) where:
  
  – \( rDom \) is the rule tagged small function extension such that \( \text{domExt}(rDom) = \text{domExt}(x) \) and, for each program \( y \), \( rDom < y > = \text{Func.Lg.Ext.Tree}(\text{tagged}(rDom, y)) \).
  
  – \( rRan \) is the rule tagged small function extension such that \( \text{domExt}(rRan) = \text{domExt}(x) \) and, for each program \( y \), \( rRan < y > = \text{Func.Lg.Ext.Tree}(x(\text{tagged}(rRan, y))) \).

For a tagged small function extension \( x \), \( \text{Func.Lg.Ext.Tree}(x) = \text{Func.Sm.Ext.one} \) if \( x \) is a tree; and \( \text{Func.Sm.Ext.zero} \) otherwise.

**Proof.**

• By induction on \( x \).

• Holds if \( x \) is a leaf small function extension.

• If \( x \) is a pair tagged small function extension: 
  \[
  \text{Func.Lg.Ext.Tree}(\text{left}(x)) = \text{Func.Sm.Ext.one} \text{ if } \text{left}(x) \text{ is a tree}; \text{ and } \text{Func.Sm.Ext.zero} \text{ otherwise (by inductive hypothesis)}. \text{Func.Lg.Ext.Tree}(\text{right}(x)) = \text{Func.Sm.Ext.one} \text{ if } \text{right}(x) \text{ is a tree}; \text{ and } \text{Func.Sm.Ext.zero} \text{ otherwise (by inductive hypothesis).} \text{Func.Lg.Ext.Tree}(x) \text{ is } \text{Func.Sm.Ext.one} \text{ if } \text{left}(x) \text{ and } \text{right}(x) \text{ are trees}; \text{ and } \text{Func.Sm.Ext.zero} \text{ otherwise.}
  \]

• Holds if \( x \) is a rule tagged small function extension. \( \square \)
8.20.6 RESULT

\[ \text{~res \{ %func \ %arg \} = (%func \ %arg); } \]

For a tagged small function extension 'x = {'func, 'arg}, 'Func.Lg.Ext.res('x) = 'func('arg).

8.20.7 RESTRICT

For a normalized large function 'unrestrict:

\[ \text{~restrict[ 'unrestrict] = ~c[ [ 'unrestrict \ ~right] \ ~i]; } \]

For a definition in pseudo-NummSquared that is parameterized by a normalized large function (for example, \text{~restrict[ 'unrestrict] parameterized by the normalized large function 'unrestrict}), the corresponding implicit definition associating the corresponding informal identifier with the corresponding large function extension is parameterized by a large function extension (for example, \text{~restrict[ 'unrestrict] parameterized by the large function extension 'unrestrict}).

For a large function extension 'unrestrict, and a tagged small function extension 'x, \text{~restrict[ 'unrestrict](x)} is the rule tagged small function extension 'r such that 'domExt('r) = 'domExt('x) and, for each 'dom('r) program 'y, 'r'y = 'unrestrict('tagged('r, 'y)).

Proof. \text{~restrict[ 'unrestrict](x)} is the rule tagged small function extension 'r such that 'domExt('r) = 'domExt('x) and, for each 'dom('r) program 'y, 'r'y = [unrestrict 'Func.Lg.Ext.right][('x, 'tagged('r, 'y))] = 'unrestrict('tagged('r, 'y)).

8.20.8 RESTRICT TO RANGE

For a normalized large function 'unrestrict:

\[ \text{~restrict.ran[ 'unrestrict] = ~c[ [ 'unrestrict \ ~res] \ ~i]; } \]

For a large function extension 'unrestrict, and a tagged small function extension 'x, \text{~restrict.ran[ 'unrestrict](x)} is the rule tagged small function extension 'r such that 'domExt('r) = 'domExt('x) and, for each 'dom('r) program 'y, 'r'y = 'unrestrict('x('tagged('r, 'y))) = 'unrestrict('x<y>).
Proof. \( \sim\text{restrict}.\text{ran}[\text{unrestrict}](x) \) is the rule tagged small function extension 'r such that 'domExt('r) = 'domExt('x) and, for each 'dom('r) program 'y, \( 'r<y> = [\text{unrestrict}
\text{Func.Lg.Ext.res}](\langle x, \text{tagged}('r, 'y)\rangle) = \text{unrestrict}'(\text{tagged}('r, 'y))).
\( \square \)

### 8.20.9 NURO SET RESULT

\[ \sim\text{Nuro.set.res} = \sim\text{ite} \begin{cases} \sim i \\
\sim\text{null} \\
0 
\end{cases} \]

For a tagged small function extension 'x, 'Func.Lg.Ext.Nuro.set.res('x) is 'x if 'x is a nuro; and 'Func.Sm.Ext.null otherwise.


For a tagged small function extension 'x, 'Func.Lg.Ext.Nuro.set.res('x) = 'Func.Sm.Ext.one('x).

### 8.20.10 TREE SET RESULT

\[ \sim\text{Tree.set.res} = \sim\text{ite} \begin{cases} \sim\text{Tree} \\
\sim i \\
\sim\text{null} 
\end{cases} \]

For a tagged small function extension 'x, 'Func.Lg.Ext.Tree.set.res('x) is 'x if 'x is a tree; and 'Func.Sm.Ext.null otherwise.

8.20.11 DEPENDENT SUM RESULT

\[
\sim s.d.res.left \{\%family \%pair\} = \sim ite[
\quad \sim Pair \%pair
\quad (\sim dom \%family \sim left \%pair)
\quad \sim null
];
\]

For a tagged small function extension ‘x = {family, pair}, ‘Func.Lg.Ext.s.d.res.left(‘x) is ‘domFuncExt(family)(left(‘pair)) if ‘pair is a pair tagged small function extension; and ‘Func.Sm.Ext.null otherwise.

\[
\sim s.d.res.right \{\%family \%pair\} = \sim ite[
\quad \sim Pair \%pair
\quad (\sim dom (%family \sim s.d.res.left)) \sim right \%pair)
\quad \sim null
];
\]

For a tagged small function extension ‘x = {family, pair}, ‘Func.Lg.Ext.s.d.res.right(‘x) is ‘domFuncExt(family)(Func.Lg.Ext.s.d.res.left(‘x))(right(‘pair)) if ‘pair is a pair tagged small function extension; and ‘Func.Sm.Ext.null otherwise.

\[
\sim s.d.res \{\%family \%pair\} = \sim ite[
\quad \sim Pair \%pair
\quad \{\sim s.d.res.left \sim s.d.res.right\}
\quad \sim null
];
\]

For a tagged small function extension ‘x = {family, pair}, ‘Func.Lg.Ext.s.d.res(‘x) is {‘Func.Lg.Ext.s.d.res.left(‘x), ‘Func.Lg.Ext.s.d.res.right(‘x)} if ‘pair is a pair tagged small function extension; and ‘Func.Sm.Ext.null otherwise.

For a tagged small function extension ‘x = {family, pair}, ‘Func.Lg.Ext.s.d.res(‘x) = ‘sumDep(family)(pair).
Proof.

- If 'pair is a pair tagged small function extension: 'Func.Lg.Ext.s.d.res('x) is the pair tagged small function extension 'p such that 'left('p) = 'domFuncExt('family)('left('pair)) and 'right('p) = 'dom-FuncExt('family('left('p)))('right('pair)).

- If 'pair is not a pair tagged small function extension: 'Func.Lg.Ext.s.d.res('x) = 'Func.Sm.Ext.null.

8.20.12 DEPENDENT PRODUCT RESULT

~p.d.res.rule.uncurry {%family %rule %arg} =
( [~dom (%family %arg)] (%rule %arg) );


~p.d.res.ruleพฤศ {%family %rule} =
~c[~p.d.res.rule.uncurry %family];

For a tagged small function extension ‘x = {‘family, ‘rule}, ‘Func.Lg.Ext.p.d.res.rule('x) is the rule tagged small function extension ‘r such that ‘domExt('r) = ‘domExt('family) and, for each ‘dom('r) program program ‘y, ‘r<‘y> = ‘Func.Lg.Ext.p.d.res.rule.uncurry({'family, ‘rule, ‘tagged('r, ‘y)}).

For a tagged small function extension ‘x = {‘family, ‘rule}, if ‘rule is a rule tagged small function extension, then ‘Func.Lg.Ext.p.d.res.rule('x) = ‘prodDep('family)('rule).

Proof. ‘Func.Lg.Ext.p.d.res.rule('x) is the rule tagged small function extension ‘r such that ‘domExt('r) = ‘domExt('family) and, for each ‘dom('r) program program ‘y, ‘r<‘y> = ‘domFuncExt('family('tagged('r, ‘y)))('rule('tagged('r, ‘y))).

~p.d.res %family %rule} =
~ite[
  [~,Rule %rule]
  ~p.d.res.rule
  ~null
];
For a tagged small function extension \( x = \{ \text{family}, \text{rule} \} \), \( \text{Func.Lg.Ext.p.d.res}(x) \) is \( \text{Func.Lg.Ext.p.d.res.rule}(x) \) if \( \text{rule} \) is a rule tagged small function extension; and \( \text{Func.Sm.Ext.null} \) otherwise.

For a tagged small function extension \( x = \{ \text{family}, \text{rule} \} \), \( \text{Func.Lg.Ext.p.d.res}(x) = \text{prodDep(\text{family})(\text{rule})} \).

**Proof.**

- If \( \text{rule} \) is a rule tagged small function extension: \( \text{Func.Lg.Ext.p.d.res}(x) = \text{Func.Lg.Ext.p.d.res.rule}(x) \).

- If \( \text{rule} \) is not a rule tagged small function extension: \( \text{Func.Lg.Ext.p.d.res}(x) = \text{Func.Sm.Ext.null} \).

**8.20.13 CURRY AUGMENTED**

For normalized large functions \( \text{‘uncurry and \ ‘augmentor} \):
\[
\text{~c.aug.uncurry[‘uncurry \ ‘augmentor]} \ {\%x \ %y} = \\
[\text{‘uncurry} \ [\text{‘augmentor} \ %x] \ %y];
\]

For large function extensions \( \text{‘uncurry and \ ‘augmentor} \), and a tagged small function extension \( z = \{ x, y \} \), \( \text{~c.aug.uncurry[‘uncurry \ ‘augmentor]}(z) = \text{‘uncurry}((\text{‘augmentor}(x), y)) \).

For normalized large functions \( \text{‘uncurry, \ ‘restrictor and \ ‘augmentor} \):
\[
\text{~c.aug[‘uncurry \ ‘restrictor \ ‘augmentor]} = \\
~c[ \\
\text{~c.aug.uncurry[‘uncurry \ ‘augmentor]} \\
\text{[‘restrictor \ ‘augmentor]} \\
];
\]

For large function extensions \( \text{‘uncurry, \ ‘restrictor and \ ‘augmentor} \), and a tagged small function extension \( x \), \( \text{~c.aug[‘uncurry \ ‘restrictor \ ‘augmentor]}(x) \) is the rule tagged small function extension \( r \) such that \( \text{domExt}(r) = \text{domExt(‘restrictor(‘augmentor(x)))} \) and, for each \( \text{dom}(r) \) program \( y \), \( r<y> = \text{~c.aug.uncurry[‘uncurry \ ‘augmentor]}(\{x, \ ‘tagged(r, y)\}) = \text{‘uncurry(\{‘augmentor(x), \ ‘tagged(r, y)\})}. \)
For large function extensions 'uncurry, 'restrictor and 'augmentor, and a tagged small function extension 'x, \( \tilde{c}.\text{aug}[\text{'uncurry 'restrictor 'augmentor}](\text{'x}) = \tilde{c}[\text{'uncurry 'restrictor 'augmentor}](\text{'x}) \).

**Proof.** \( \tilde{c}[\text{'uncurry 'restrictor 'augmentor}](\text{'x}) = \tilde{c}[\text{'uncurry 'restrictor}](\text{'augmentor}(\text{'x})) \) is the rule tagged small function extension 'r such that \( \text{domExt('r)} = \text{domExt('restrictor('augmentor('x)))} \) and, for each \( \text{dom('r)} \) program program 'y, \( 'r<\text{'y}> = \text{'uncurry(('augmentor('x), 'tagged('r, 'y)))} \).

### 8.20.14 CURRY RESULT

For a normalized large function 'uncurry:
\[
\sim c.\text{res}[\text{'uncurry}](\text{'x 'domFuncExt('restrictor)('y)}) =
\text{[ 'uncurry \text{'x } ([\sim\text{dom 'restrictor}] \text{'y}) ]}.
\]

For a large function extension 'uncurry, and a tagged small function extension \( 'z = \{\text{'x, 'restrictor, 'y}\} \), \( \sim c.\text{res}[\text{'uncurry}]('z) = \text{'uncurry(('[\text{'x, 'domFuncExt('restrictor)('y)})} \).

### 8.20.15 RECURSION RIGHT-HAND-SIDE

For normalized large functions 'start and 'step:
\[
\sim r.\text{ran}[\text{'start 'step}] = \sim\text{restrict. ran}[\sim r[\text{'start 'step}]];
\]

For large function extensions 'start and 'step, and a tagged small function extension 'x, \( \sim r.\text{dom}[\text{'start 'step}]('x) \) is the rule tagged small function extension 'r such that \( \text{domExt('r)} = \text{domExt('x)} \) and, for each \( \text{dom('r)} \) program program 'y, \( 'r<\text{'y}> = \sim r[\text{'start 'step}][\text{'tagged('r, 'y)})] \).

For normalized large functions 'start and 'step:
\[
\sim r.\text{ran}[\text{'start 'step}] = \sim\text{restrict. ran}[\sim r[\text{'start 'step}]];
\]

For large function extensions 'start and 'step, and a tagged small function extension 'x, \( \sim r.\text{ran}[\text{'start 'step}]('x) \) is the rule tagged small function extension 'r such that \( \text{domExt('r)} = \text{domExt('x)} \) and, for each \( \text{dom('r)} \) program program 'y, \( 'r<\text{'y}> = \sim r[\text{'start 'step}][\text{'x('tagged('r, 'y)})] \).

For normalized large functions 'start and 'step:
~r.step['start 'step] =
['step ~r.dom['start 'step] ~r.ran['start 'step] ~i];

For large function extensions 'start and 'step, and a tagged small function extension 'x, ~r.step['start 'step]('x) = 'step([~r.dom['start 'step]('x), ~r.ran['start 'step]('x), 'x]).

For normalized large functions 'start and 'step:

~r.rhs['start 'step] =
~ite[
    ~Null
    'start
    ~r.step['start 'step]
];

For large function extensions 'start and 'step, and a tagged small function extension 'x, ~r.rhs['start 'step]('x) = 'start('x) if 'x = 'Func.Sm.Ext.null; and ~r.step['start 'step]('x) otherwise.

For large function extensions 'start and 'step, and a tagged small function extension 'x, ~r.rhs['start 'step]('x) = ~r['start 'step]('x).

8.20.16 NEGATION

~not = [~not.n ~One];

For a tagged small function extension 'x, 'Func.Lg.Ext.not('x) = 'Func.Lg.Ext.not.n('Func.Lg.Ext.One('x)).

For a tagged small function extension 'x, 'Func.Lg.Ext.not('x) = 'Func.Sm.Ext.zero if 'x is true; and 'Func.Sm.Ext.one otherwise.

8.20.17 IMPLICATION WITH NULL

~imp.n {%b %c} =
~ite[
    %b
    [~,conf.n %c]
8.20.18 IMPLICATION

\[ \neg \text{imp } \{ \%b \%c \} = [\neg \text{imp.n } [\neg \text{One } \%b] [\neg \text{One } \%c]]; \]

For a tagged small function extension \( x = \{ \%b, \%c \} \), \( \text{Func.Lg.Ext.imp.n}(x) \) is given by one of the following mutually exclusive cases:

- \( \text{Func.Sm.Ex.one} \) if \( \%b = \text{Func.Sm.Ext.zero} \)
- \( \text{Func.Lg.Ext.conf.n}(\%c) \) if \( \%b = \text{Func.Sm.Ext.one} \)
- \( \text{Func.Sm.Ext.null} \) if \( \%b \) is not Boolean

8.21 NORMAL FORM OF A NON-COMPUTATIONAL NON-NORMALIZED CONSTANT OR NON-COMPUTATIONAL COMBINATION

For a non-computational non-normalized constant \( f \), the normal form of \( f \), denoted by \( \text{norm}(f) \), is defined to be the corresponding normalized large function below. The normal form of a non-computational combination \( f \) cannot be defined at this point because the normal form of \( f \) depends upon the normal forms of the components of \( f \). Instead, the corresponding combination of normalized large functions is defined.

Corresponding combinations of normalized large functions have already been defined for the following:

- a Hilbert non-computational combination
8.21.1 EXISTENTIAL QUANTIFICATION

Existential quantification is now defined using Hilbert in a manner somewhat similar to [4].

For a normalized large function `pred`:
\[
\neg \exists \text{[`pred]} = [ \neg \text{One} \ [`\text{pred} \ \neg i \ \neg h[`\text{pred}]] ];
\]

For a large function extension `pred`, and a tagged small function extension `x`, \( \neg \exists [\text{`pred}](`x) \) = `Func.Lg.Ext.One( `pred([`x, `h[`pred](`x)]) )`.

For a large function extension `pred`, and a tagged small function extension `x`, \( \neg \exists [\text{`pred}](`x) \) is `Func.Sm.Ext.one if there exists some tagged small function extension `y` such that `pred([`x, `y])` is true; and `Func.Sm.Ext.zero otherwise.

\textit{Proof.}

- If there exists some tagged small function extension `y` such that `pred([`x, `y])` is true: `h[`pred](`x) is some tagged small function extension `y` such that `pred([`x, `y])` is true. \( \neg \exists [\text{`pred}](`x) = `Func.Lg.Ext.One( `pred([`x, `y]) ) = `Func.Sm.Ext.one.

- If there does not exist some tagged small function extension `y` such that `pred([`x, `y])` is true: `pred([`x, `h[`pred](`x)]) \neq `Func.Sm.Ext.one.

8.21.2 UNIVERSAL QUANTIFICATION

Universal quantification is now defined using existential quantification in a manner similar to [9, p.34].

For a normalized large function `pred`:
\[
\neg \forall \text{[`pred]} = \neg \exists [[\neg \text{not} \ `\text{pred}] ];
\]

For a large function extension `pred`, and a tagged small function extension `x`, \( \neg \forall [\text{`pred}](`x) \) is `Func.Sm.Ext.zero if, for each tagged small function extension `y`, `pred([`x, `y])` is true; and `Func.Sm.Ext.one otherwise.

\textit{Proof.} \( \neg \forall [\text{`pred}](`x) \) is `Func.Sm.Ext.one if there exists some tagged small function extension `y` such that `[Func.Lg.Ext not `pred]([`x, `y])` is true; and `Func.Sm.Ext.zero otherwise. \( \neg \forall [\text{`pred}](`x) \) is `Func.Sm.Ext.one if there exists some tagged small
function extension 'y such that 'pred(('x, 'y)) is not true; and 'Func.Sm.Ext.zero otherwise.

For a normalized large function 'pred:
\[ \sim \text{all}[^\text{pred}] = [\sim \not \sim \not \text{all}[^\text{pred}]]; \]

For a large function extension 'pred, and a tagged small function extension 'x,
\( \sim \text{all}[^\text{pred}]('x) \) is 'Func.Sm.Ext.one if, for each tagged small function extension 'y,
'pred(('x, 'y)) is true; and 'Func.Sm.Ext.zero otherwise.

8.21.3 UNARY UNIVERSAL QUANTIFICATION

For a normalized large function 'pred:
\[ \sim \text{all.una}[^\text{pred}] = \sim \text{all}[^\{^\text{pred} \sim \text{right}\}]; \]

For a large function extension 'pred, and a tagged small function extension 'x,
\( \sim \text{all.una}[^\text{pred}]('x) \) is 'Func.Sm.Ext.one if, for each tagged small function extension 'y,
'pred('y) is true; and 'Func.Sm.Ext.zero otherwise.

For a large function extension 'pred, and a tagged small function extension 'x,
\( \sim \text{all.una}[^\text{pred}]('x) \) is 'Func.Sm.Ext.one if 'pred is true; and 'Func.Sm.Ext.zero otherwise.

For a large function extension 'pred, \( \sim \text{all.una}[^\text{pred}] \) is unchanging.

8.21.4 SMALL UNIVERSAL QUANTIFICATION

\[ \sim \text{all.sm} = \sim \text{all}[\sim \text{res}]; \]

For a tagged small function extension 'x, 'Func.Lg.Ext.all.sm('x) is 'Func.Sm.Ext.one if, for each tagged small function extension 'y, 'x('y) is true; and 'Func.Sm.Ext.zero otherwise.

For a tagged small function extension 'x, 'Func.Lg.Ext.all.sm('x) is 'Func.Sm.Ext.one if 'x is universally true; and 'Func.Sm.Ext.zero otherwise.

For a tagged small function extension 'x, 'Func.Lg.Ext.all.sm('x) is 'Func.Sm.Ext.one if, for each 'dom('x) program 'y, 'x('tagged('x, 'y)) = 'x<y> is true; and 'Func.Sm.Ext.zero otherwise.
8.21.5 EQUALS RIGHT-HAND-SIDE

```plaintext
~.pair {~.x ~.y} =
~ite[
    [~Pair ~.x]
    ~ite[
        [~Pair ~.y]
        ~ite[
            [~left ~.x] [~left ~.y]
            [~right ~.x] [~right ~.y]
            0
        ]
        ~null
    ]
    ~null
];
```

For a tagged small function extension ‘z = {'x, ‘y}, ‘Func.Lg.Ext.eq.pair('z) is given by one of the following mutually exclusive cases:

- If ‘x and ‘y are both pair tagged small function extensions: ‘Func.Lg.Ext.eq.pair('z) is ‘Func.Sm.Ext.one if ‘left('x) = ‘left('y) and ‘right('x) = ‘right('y); and ‘Func.Sm.Ext.zero otherwise.

- ‘Func.Sm.Ext.null if ‘x and ‘y are not both pair tagged small function extensions

For a tagged small function extension ‘z = {'x, ‘y}, if ‘x and ‘y are pair tagged small function extensions, then ‘Func.Lg.Ext.eq.pair('z) is ‘Func.Sm.Ext.one if ‘x = ‘y; and ‘Func.Sm.Ext.zero otherwise.

```plaintext
~.res.at {~.x ~.y ~.arg} = [~ (~.x ~.arg) (~.y ~.arg)];
```

For a tagged small function extension ‘z = {'x, ‘y, ‘arg}, ‘Func.Lg.Ext.eq.res.at('z) is ‘Func.Sm.Ext.one if ‘x('arg) = ‘y('arg); and ‘Func.Sm.Ext.zero otherwise.

```plaintext
~.res {~.x ~.y} = ~all[~.res.at];
```
For a tagged small function extension \( z = \{x, y \} \), \( \text{Func.Lg.Ext.eq.res}(z) \) is \( \text{Func.Sm.Ext.one} \) if, for each tagged small function extension \( w \), \( \text{Func.Lg.Ext.eq.res.at}(\{x, y, w\}) \) is true; and \( \text{Func.Sm.Ext.zero} \) otherwise.

For a tagged small function extension \( z = \{x, y \} \), \( \text{Func.Lg.Ext.eq.res}(z) \) is \( \text{Func.Sm.Ext.one} \) if, for each tagged small function extension \( w \), \( x(w) = y(w) \); and \( \text{Func.Sm.Ext.zero} \) otherwise.

\[
= \text{dom.res \{x %y\}} = [= \text{res}[\text{~dom x}][\text{~dom y}]];
\]

For a tagged small function extension \( z = \{x, y \} \), \( \text{Func.Lg.Ext.eq.dom.res}(z) \) is \( \text{Func.Sm.Ext.one} \) if, for each tagged small function extension \( w \), \( \text{domFuncExt}(x)(w) = \text{domFuncExt}(y)(w) \); and \( \text{Func.Sm.Ext.zero} \) otherwise.

\[
= \text{both.res \{x %y\}} =
= \text{ite}[
= = \text{dom.res}
= = \text{res}
0
];
\]

For a tagged small function extension \( z = \{x, y \} \), \( \text{Func.Lg.Ext.eq.both.res}(z) \) is \( \text{Func.Sm.Ext.one} \) if, for each tagged small function extension \( w \), \( \text{domFuncExt}(x)(w) = \text{domFuncExt}(y)(w) \) and \( x(w) = y(w) \); and \( \text{Func.Sm.Ext.zero} \) otherwise.

For a tagged small function extension \( z = \{x, y \} \), if \( x \) and \( y \) are rule tagged small function extensions, then \( \text{Func.Lg.Ext.eq.both.res}(z) \) is \( \text{Func.Sm.Ext.one} \) if \( x = y \); and \( \text{Func.Sm.Ext.zero} \) otherwise.

\[
= \text{rhs \{x %y\}} =
= \text{ite}[
[\text{~Null x}]
[\text{~Null y}]
= \text{ite}[
[\text{~Zero x}]
[\text{~Zero y}]
= \text{ite}[
[\text{~One x}]
[\text{~One y}]
]
For a tagged small function extension \( 'p \), 'Func.Lg.Ext.eq.rhs('p) is given by one of the following mutually exclusive cases:

- 'Func.Sm.Ext.one if \( 'p \) is a pair tagged small function extension, and \( \text{left('p) = right('p) \) }

- 'Func.Sm.Ext.zero if \( 'p \) is a pair tagged small function extension, and \( \text{left('p) \neq right('p) \) }

- 'Func.Sm.Ext.null if \( 'p \) is not a pair tagged small function extension

For a tagged small function extension \( 'p \), 'Func.Lg.Ext.eq.rhs('p) = 'Func.Lg.Ext.eq('p).

### 8.21.6 NOT EQUALS

\(~\text{not.}= = \{ %x \ %y \} ~\text{not} ~= \) ;

For a tagged small function extension \( 'z = \{ 'x, 'y \} \), 'Func.Lg.Ext.not.eq('x) = 'Func.Sm.Ext.zero if \( 'x = 'y \); and 'Func.Sm.Ext.one otherwise.
8.21.7 INDUCTIVE CASE

Recall that, for a small function extension \( f \neq \text{Func.Sm.Ext.null} \), and a \( \text{field}(f) \) program \( x \), \( x \) is structurally smaller than \( f \). This fact permits a simple induction principle for NummSquared, complementing the terminating recursion principle.

For a normalized large function \( \text{pred} \):
\[
\sim\text{induc.hyp.dom}[\text{pred}] = [\sim\text{all.sm} \sim\text{restrict}[\text{pred}]];
\]

For a large function extension \( \text{pred} \), and a tagged small function extension \( x \), \( \sim\text{induc.hyp.dom}[\text{pred}](x) = \text{Func.Lg.Ext.all.sm}(\sim\text{restrict}[\text{pred}](x)) \).

For a large function extension \( \text{pred} \), and a tagged small function extension \( x \), \( \sim\text{induc.hyp.dom}[\text{pred}](x) \) is \( \text{Func.Sm.Ext.one} \) if, for each \( \text{dom}(x) \) program \( y \), \( \text{pred}(\text{tagged}(x, y)) \) is true; and \( \text{Func.Sm.Ext.zero} \) otherwise.

\textbf{Proof.} \( \sim\text{induc.hyp.dom}[\text{pred}](x) \) is \( \text{Func.Sm.Ext.one} \) if, for each \( \text{dom}(\sim\text{restrict}[\text{pred}](x)) \) program \( y \), \( \text{restrict}[\text{pred}](x)<y \) is true; and \( \text{Func.Sm.Ext.zero} \) otherwise. \( \text{restrict}[\text{pred}](x) \) is the rule tagged small function extension \( r \) such that \( \text{domExt}(r) = \text{domExt}(x) \) and, for each \( \text{dom}(r) \) program \( y \), \( r<y = \text{pred}(\text{tagged}(r, y)) \).

For a normalized large function \( \text{pred} \):
\[
\sim\text{induc.hyp.ran}[\text{pred}] = [\sim\text{all.sm} \sim\text{restrict.ran}[\text{pred}]];
\]

For a large function extension \( \text{pred} \), and a tagged small function extension \( x \), \( \sim\text{induc.hyp.ran}[\text{pred}](x) = \text{Func.Lg.Ext.all.sm}(\sim\text{restrict.ran}[\text{pred}](x)) \).

For a large function extension \( \text{pred} \), and a tagged small function extension \( x \), \( \sim\text{induc.hyp.ran}[\text{pred}](x) \) is \( \text{Func.Sm.Ext.one} \) if, for each \( \text{dom}(x) \) program \( y \), \( \text{pred}(x(\text{tagged}(x, y))) = \text{pred}(x<y) \) is true; and \( \text{Func.Sm.Ext.zero} \) otherwise.

\textbf{Proof.} \( \sim\text{induc.hyp.ran}[\text{pred}](x) \) is \( \text{Func.Sm.Ext.one} \) if, for each \( \text{dom}(\sim\text{restrict.ran}[\text{pred}](x)) \) program \( y \), \( \text{restrict.ran}[\text{pred}](x)<y \) is true; and \( \text{Func.Sm.Ext.zero} \) otherwise. \( \text{restrict.ran}[\text{pred}](x) \) is the rule tagged small function extension \( r \) such that \( \text{domExt}(r) = \text{domExt}(x) \) and, for each \( \text{dom}(r) \) program \( y \), \( r<y = \text{pred}(x(\text{tagged}(r, y))) \).

\begin{verbatim}
~induc.case.at[\text{pred}] = 
[~imp
   ~induc.hyp.dom[\text{pred}]
]
\end{verbatim}
For a large function extension 'pred, and a tagged small function extension 'x, `induc.case.at['pred]('x) is 'Func.Lg.Ext.ONE('pred('x)) if `induc.hyp.dom['pred]('x) and `induc.hyp.ran['pred]('x) are true; and 'Func.Sm.Ext.ONE otherwise.

For a normalized large function 'pred:

`induc.case['pred] = `all.una[`induc.case.at['pred]];

For a large function extension 'pred, and a tagged small function extension 'x, `induc.case['pred]('x) is 'Func.Sm.Ext.ONE if `induc.case.at['pred] is true; and 'Func.Sm.Ext.ZERO otherwise.

For a large function extension 'pred, `induc.case['pred] is unchanging.

The induction principle itself is given along with other true large function extensions.

8.22 NORMAL FORM AND VALIDITY OF A LARGE FUNCTION

For a global context 'cg, a local context 'cl, and a large function 'f, the normal form in 'cg and 'cl of 'f (a normalized large function or 'null), denoted by `norm('cg, 'cl, 'f), is defined by recursion on 'f:

- `norm('f) if 'f is a primitive
- `norm('f) if 'f is a constant
- 'null if 'f = ['outer 'x0 'x1 ... 'xm-1] and at least one of `norm('cg, 'cl, 'outer), `norm('cg, 'cl, 'x0), 'norm('cg, 'cl, 'x1), ..., 'norm('cg, 'cl, 'xm-1) is 'null
- ['norm('cg, 'cl, 'outer) 'norm('cg, 'cl, 'x0) 'norm('cg, 'cl, 'x1) ... 'norm('cg, 'cl, 'xm-1)] if 'f = ['outer 'x0 'x1 ... 'xm-1] and all of 'norm('cg, 'cl, 'outer), 'norm('cg, 'cl, 'x0), 'norm('cg, 'cl, 'x1), ..., 'norm('cg, 'cl, 'xm-1) are ≠ 'null
- The other combination cases are similar and are omitted.
• `norm(cg, f)` if `f` is a global name

• `norm(cl, f)` if `f` is a local name

• `null` if `f = ¬C[called]` and `norm(cg, cl, called) = null`

• `computed(norm(cg, cl, called))` if `f = ¬C[called]` and `norm(cg, cl, called) ≠ null`

• `null` if `f = ¬Q[unquoted]` and `norm(cg, cl, unquoted) = null`

• `quoted(norm(cg, cl, unquoted))` if `f = ¬Q[unquoted]` and `norm(cg, cl, unquoted) ≠ null`

• `null` if `f = ¬UQ[quoted]` and `norm(cg, cl, quoted) = null`

• `unquoted(norm(cg, cl, quoted))` if `f = ¬UQ[quoted]` and `norm(cg, cl, quoted) ≠ null`

• `null` if `f = #called[x0 x1 ... x_{m-1}]` and at least one of `norm(cg, cl, called)`, `norm(cg, cl, x0)`, `norm(cg, cl, x1)`, ..., `norm(cg, cl, x_{m-1})` is `null`

• `macroExpanded(norm(cg, cl, called), l < norm(cg, cl, x0), norm(cg, cl, x1), ..., norm(cg, cl, x_{m-1}) >)` if `f = #called[x0 x1 ... x_{m-1}]` and all of `norm(cg, cl, called)`, `norm(cg, cl, x0)`, `norm(cg, cl, x1)`, ..., `norm(cg, cl, x_{m-1})` are `null`

For a global context `cg`, a local context `cl`, and a large function `f`, `f` is valid in `cg` and `cl` iff `norm(cg, cl, f) ≠ null`.

### 8.23 NORMAL FORM AND VALIDITY OF A DEFINITION, DEFINITION LIST OR ABSTRACT PROGRAM

For a global context `cg`, and a local tuple accessor checker `checker` containing `<lis, onFail>`, the normal form in `cg` of `checker` (a valid normalized local tuple accessor checker or `null`), denoted by `norm(cg, checker)`, is the normalized local tuple accessor checker containing `<lis, norm(cg, 0, onFail)>` if `lis` is valid and `onFail` is valid in `cg` and `0`; and `null` otherwise.
For a global context 'cg, and a local tuple accessor checker 'checker, 'checker is valid in 'cg iff \( \text{norm('cg, 'checker)} \neq 'null \).

For a global context 'cg, and a local tuple accessor descriptor 'desc, the normal form in 'cg of 'desc (a valid normalized local tuple accessor descriptor or 'null), denoted by \( \text{norm('cg, 'desc)} \), is given by one of the following mutually exclusive cases:

- 0 if 'desc = 0
- as above if 'desc is a normalized local tuple accessor checker

For a global context 'cg, and a local tuple accessor descriptor 'desc, 'desc is valid in 'cg iff \( \text{norm('cg, 'desc)} \neq 'null \).

For a global context 'cg, and a definition 'def containing <'comment, 'name, 'accessTupleLocDesc, 'rhs>, the normal form in 'cg of 'def, denoted by \( \text{norm('cg, 'def)} \), is the normalized definition containing <'name, 'addCheck('norm('cg, 'accessTupleLocDesc), 'norm('cg, 'contextLoc('norm('cg, 'accessTupleLocDesc)), 'rhs))> if 'cg('name) = 'null, 'accessTupleLocDesc is valid in 'cg, and 'rhs is valid in 'cg and 'contextLoc('norm('cg, 'accessTupleLocDesc)); and 'null otherwise.

For a global context 'cg, and a definition 'def, 'def is valid in 'cg iff \( \text{norm('cg, 'def)} \neq 'null \).

For a definition list 'dl containing 'l, the normal form of 'dl (a valid global context or 'null), denoted by \( \text{norm('dl)} \), is defined by recursion on 'l:

- the global context containing 0 if 'l = 0
- If 'l = <'def, 'r>: Let 'dlR be the definition list containing 'r. Let 'cgR = \( \text{norm('dlR)} \). 'norm('dl) is given by one of the following mutually exclusive cases:
  - 'null if 'cgR = 'null
  - If 'cgR \neq 'null: Let 'cgR contain 'cgRL. 'norm('dl) is the global context containing <'norm('cgR, 'def), 'cgRL> if 'def is valid in 'cgR; and 'null otherwise.

A definition list 'dl is valid iff \( \text{norm('dl)} \neq 'null \).

For a program 'prog, the normal form of 'prog, denoted by \( \text{norm('prog)} \), is \( \text{norm('defLis('prog))} \).

A program 'prog is valid iff 'moduNameLis('prog) is valid and \( \text{norm('prog)} \neq 'null \).
8.24 PSEUDO-NUMMSQUARED COMPLETE

At this point, normal forms have been completely defined. Therefore, pseudo-NummsSquared can include the full NummsSquared concrete syntax.

8.25 SOME TRUE LARGE FUNCTION EXTENSIONS

For large function extensions ‘f and ‘g, [Func.Lg.Ext.eq ‘f ‘g] is true iff ‘f = ‘g.

Proof. For each tagged small function extension ‘x: [Func.Lg.Ext.eq ‘f ‘g](‘x) = Func.Lg.Ext.eq((‘f(‘x), ‘g(‘x))). [Func.Lg.Ext.eq ‘f ‘g](‘x) is true iff ‘f(‘x) = ‘g(‘x).

For large function extensions ‘f and ‘x, if ‘f is unchanging, the following is true:

[Func.Lg.Ext.eq ‘f ‘x] ‘f

Proof. For each tagged small function extension ‘y: [f ‘x](‘y) = ‘f(‘x(‘y)) = ‘f(‘y).

For a large function extension ‘f such that, for each tagged small function extension ‘x, ‘f(‘x) is a rule tagged small function extension and an identity, the following is true:

[Func.Lg.Ext.eq Func.Lg.Ext.dom ‘f] ‘f

Proof. For each tagged small function extension ‘x: [Func.Lg.Ext.dom ‘f](‘x) = ‘dom-FuncExt(‘f(‘x)) = ‘f(‘x).

8.25.1 IDENTITY

Identity large composition axiom: For a normalized large function ‘x:

ax.i.co.lg[‘x] = [=] [‘x]

For a large function extension ‘x, ‘Func.Lg.Ext.ax.i.co.lg[‘x] is true.

Proof. For each tagged small function extension ‘y: [Func.Lg.Ext.i ‘x](‘y) = Func.Lg.Ext.i(‘x(‘y)) = ‘x(‘y).

Identity large composition right axiom: For a normalized large function ‘x:

ax.i.co.lg.right[‘x] = [~] [‘x]

For a large function extension ‘x, ‘Func.Lg.Ext.ax.i.co.lg.right[‘x] is true.

Proof. For each tagged small function extension ‘y: [‘x ‘Func.Lg.Ext.i](‘y) = ‘x(‘Func.Lg.Ext.i(‘y)) = ‘x(‘y).
8.25.2 NULL

**Null large composition axiom:** For a normalized large function 'x:

\[ ax.\text{null.co.lg}[^x] = [\sim \sim\text{null} \ 'x] \sim\text{null}] \]

For a large function extension 'x, 'Func.Lg.Ext.ax.null.co.lg[^x] is true.

*Proof.* 'Func.Lg.Ext.null is unchanging.

**Null null predicate axiom:**

\[ ax.\text{null.Null} = [\sim \sim\text{null} \sim\text{Null}] 1] \]

'Func.Lg.Ext.ax.null.Null is true.

*Proof.* For each tagged small function extension 'x: ['Func.Lg.Ext.Null

'Func.Lg.Ext.null]('x) = 'Func.Lg.Ext.Null('Func.Lg.Ext.null('x)) =


**Null pair predicate axiom:**

\[ ax.\text{null.Pair} = [\sim \sim\text{null} \sim\text{Pair}] 0] \]

'Func.Lg.Ext.ax.null.Pair is true.

*Proof.* For each tagged small function extension 'x: ['Func.Lg.Ext.Pair

'Func.Lg.Ext.null]('x) = 'Func.Lg.Ext.Pair('Func.Lg.Ext.null('x)) =


**Null domain axiom:**

\[ ax.\text{null.dom} = [\sim \sim\text{null} \sim\text{Null.set}] \]

'Func.Lg.Ext.ax.null.dom is true.

*Proof.* For each tagged small function extension 'x: ['Func.Lg.Ext.dom

'Func.Lg.Ext.null]('x) = 'domFuncExt('Func.Lg.Ext.null('x)) = 'dom-


**Null small composition axiom:** For a normalized large function 'x:

\[ ax.\text{null.co.sm}[^x] = [\sim \sim\text{null} \ 'x] \sim\text{null}] \]

For a large function extension 'x, 'Func.Lg.Ext.ax.null.co.sm[^x] is true.
Proof. For each tagged small function extension ‘y: ‘(Func.Lg.Ext.null ‘x)‘(y)
= ‘Func.Lg.Ext.null(‘y)‘(‘x‘(y)) = ‘Func.Sm.Ext.null(‘x‘(y)) = ‘Func.Sm.Ext.null
= ‘Func.Lg.Ext.null(‘y).

Null if-then-else axiom: For normalized large functions ‘t and ‘e:
ax.null.ite[‘t ‘e] = [~= ~ite[~null ‘t ‘e] ~null];

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.null.ite[‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.null(‘x)
‘Func.Lg.Ext.null(‘x).

8.25.3 ZERO

Zero large composition axiom: For a normalized large function ‘x:
ax.zero.co.lg[‘x] = [~= [0 ‘x] 0];

For a large function extension ‘x, ‘Func.Lg.Ext.ax.zero.co.lg[‘x] is true.

Proof. ‘Func.Lg.Ext.zero is unchanging.

Zero null predicate axiom:

ax.zero.Null = [~= [~Null 0] 0];

‘Func.Lg.Ext.ax.zero.Null is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null

Zero pair predicate axiom:

ax.zero.Pair = [~= [~Pair 0] 0];

‘Func.Lg.Ext.ax.zero.Pair is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair

Zero domain axiom:
ax.zero.dom = [\sim \sim (\text{dom} \ 0) \sim \text{Null.set}];

’Func.Lg.Ext.ax.zero.dom is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.dom
‘Func.Lg.Ext.zero]’x) = ‘\text{dom}\text{FuncExt}’(‘Func.Lg.Ext.zero’x)) = ‘\text{dom}’

\[ \]

\textbf{Zero small composition axiom:} For a normalized large function ‘x:
ax.zero.co.sm[‘x] = [\sim (0 \ ‘x) \sim \text{null}];

For a large function extension ‘x, ‘Func.Lg.Ext.ax.zero.co.sm[‘x] is true.

Proof. For each tagged small function extension ‘y: [‘Func.Lg.Ext.zero ‘x]’y)
‘Func.Lg.Ext.null’y).

\[ \]

\textbf{Zero if-then-else axiom:} For normalized large functions ‘t and ‘e:
ax.zero.ite[‘t ‘e] = [\sim \text{ite}[0 \ ‘t \ ‘e] \ ‘e];

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.zero.ite[‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.zero’x) =

\[ \]

8.25.4 \ ONE

\textbf{One large composition axiom:} For a normalized large function ‘x:
ax.one.co.lg[‘x] = [\sim [1 \ ‘x] \ 1];

For a large function extension ‘x, ‘Func.Lg.Ext.ax.one.co.lg[‘x] is true.

Proof. ‘Func.Lg.Ext.one is unchanging.

\[ \]

\textbf{One null predicate axiom:}
ax.one.Null = [\sim [\sim \text{Null} \ 1] \ 0];

‘Func.Lg.Ext.ax.one.Null is true.
Proof. For each tagged small function extension \('x\): 

One pair predicate axiom:

\[
\text{ax.one.Pair} = \begin{bmatrix} \sim \sim \text{Pair} \ 1 \ 0 \end{bmatrix};
\]

'Func.Lg.Ext.ax.one.Pair is true.

Proof. For each tagged small function extension \('x\): 

One domain axiom:

\[
\text{ax.one.dom} = \begin{bmatrix} \sim \sim \text{dom} \ 1 \ \sim \text{Nuro.set} \end{bmatrix};
\]

'Func.Lg.Ext.ax.one.dom is true.

Proof. For each tagged small function extension \('x\): 

One small composition axiom: For a normalized large function \('x\):

\[
\text{ax.one.co.sm['}x'] = \begin{bmatrix} \sim \sim 1 \ 'x \ \sim \text{Nuro.set.res '}x' \end{bmatrix};
\]

For a large function extension \('x\), 'Func.Lg.Ext.ax.one.co.sm['}x'] is true.

Proof. For each tagged small function extension \('y\): 

One if-then-else axiom: For normalized large functions \('t\) and \('e\):

\[
\text{ax.one.ite['}t 'e'] = \begin{bmatrix} \sim \sim \text{ite}1 \ 't \ 'e \ 't \end{bmatrix};
\]

For large function extensions \('t\) and \('e\), 'Func.Lg.Ext.ax.one.ite['}t 'e'] is true.

Proof. For each tagged small function extension \('x\): 
'Func.Lg.Ext.one('x) = 'Func.Sm.Ext.one. 'ite['}Func.Lg.Ext.one 't 'e']('x) = 't('x).
8.25.5 NULL SET

**Null set large composition axiom:** For a normalized large function ‘x:

\[
\text{ax.Null.set.co.lg}[x] = [\sim (\sim \text{Null.set } x \sim \text{Null.set})];
\]

For a large function extension ‘x, ‘Func.Lg.Ext.ax Null.set.co.lg[‘x] is true.


**Null set null predicate axiom:**

\[
\text{ax.Null.set.Null} = [\sim (\sim \text{Null} \sim \text{Null.set}) 0];
\]


*Proof.* For each tagged small function extension ‘x: ['Func.Lg.Ext.Null


‘Func.Lg.Ext.zero('x).

**Null set pair predicate axiom:**

\[
\text{ax.Null.set.Pair} = [\sim (\sim \text{Pair} \sim \text{Null.set}) 0];
\]


*Proof.* For each tagged small function extension ‘x: ['Func.Lg.Ext.Pair


‘Func.Lg.Ext.zero('x).

**Null set domain axiom:**

\[
\text{ax.Null.set.dom} = [\sim (\sim \text{dom} \sim \text{Null.set}) \sim \text{Null.set}];
\]

‘Func.Lg.Ext.ax.Null.set.dom is true.

*Proof.* For each tagged small function extension ‘x, ‘Func.Lg_EXT.Null.set('x) =

‘Func.Sm_EXT.Tagged.Null.set is a rule tagged small function extension and an iden-

tity.

**Null set small composition axiom:** For a normalized large function ‘x:

\[
\text{ax.Null.set.co.sm}[x] = [\sim (\sim \text{Null.set } x \sim \text{null})];
\]

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Null.set.co.sm[‘x] is true.

Null set if-then-else axiom: For normalized large functions ‘t and ‘e:

ax.Null.set.ite[‘t ‘e] =
[~= ~ite[~Null.set ‘t ‘e] ~null];

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.Null.set.ite[‘t ‘e] is true.


8.25.6 NURO SET

NuRo set large composition axiom: For a normalized large function ‘x:

ax.Nuro.set.co.lg[‘x] = [~= [~Nuro.set ‘x] ~Nuro.set];

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Nuro.set.co.lg[‘x] is true.


NuRo set null predicate axiom:


NuRo set pair predicate axiom:

ax.Nuro.set.Pair = [~= [~Pair ~Nuro.set] 0];


**Nuro set domain axiom:**

\[
\text{ax.Nuro.set.dom = \{\text{~= \{~dom ~Nuro.set\} ~Nuro.set\};}
\]

‘Func.Lg.Ext.ax.Nuro.set.dom is true.


**Nuro set small composition axiom:** For a normalized large function ‘x:

\[
\text{ax.Nuro.set.co.sm['x] =}
\]

\[
\text{[= (~Nuro.Set ‘x) [~Nuro.set.res ‘x]];}
\]

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Nuro.set.co.sm['x] is true.


**Nuro set if-then-else axiom:** For normalized large functions ‘t and ‘e:

\[
\text{ax.Nuro.set.ite['t 'e] =}
\]

\[
\text{[= ~ite[~Nuro.set ‘t ‘e] ~null]};
\]

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.Nuro.set.ite['t 'e] is true.


8.25.7 LEAF SET

**Leaf set large composition axiom:** For a normalized large function ‘x:

\[
\text{ax.Leaf.set.co.lg[‘x] = \{\text{~= \{~Leaf.set \ ‘x\} ~Leaf.set\};}
\]

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Leaf.set.co.lg[‘x] is true.
Proof. 'Func.Lg.Ext.Leaf.set is unchanging.

**Leaf set null predicate axiom:**

\[\text{ax.Leaf.set.Null} = \lnot [\text{Null} \land \text{Leaf.set}] 0;\]

'Func.Lg.Ext.ax.Leaf.set.Null is true.

Proof. For each tagged small function extension 'x:


**Leaf set pair predicate axiom:**

\[\text{ax.Leaf.set.Pair} = \lnot [\text{Pair} \land \text{Leaf.set}] 0;\]

'Func.Lg.Ext.ax.Leaf.set.Pair is true.

Proof. For each tagged small function extension 'x:


**Leaf set domain axiom:**

\[\text{ax.Leaf.set.dom} = \lnot [\text{dom} \land \text{Leaf.set}] \land \text{Leaf.set};\]

'Func.Lg.Ext.ax.Leaf.set.dom is true.

Proof. For each tagged small function extension 'x:

\[\text{'Func.Lg.Ext.Leaf.set} ('x) = \text{'Func.Sm.Ext.Tagged.Leaf.set is a rule tagged small function extension and an identity.}\]

**Leaf set small composition axiom:** For a normalized large function 'x:

\[\text{ax.Leaf.set.co.sm['x]} = \lnot [\text{Leaf.Set 'x}] \land [\text{conf.n 'x}];\]

For a large function extension 'x, 'Func.Lg.Ext.ax.Leaf.set.co.sm['x] is true.

Proof. For each tagged small function extension 'y:

\[\text{'Func.Lg.Ext.Leaf.Set 'x}('y) = \text{'Func.Lg.Ext.Leaf.Set('y)('x('y))} = \text{'Func.Sm.Ext.Tagged.Leaf.set('x('y))} = \text{'Func.Lg.Ext.conf.n('x('y)) = 'Func.Lg.Ext.conf.n 'x}('y).\]

**Leaf set if-then-else axiom:** For normalized large functions 't and 'e:
ax.Leaf.set.ite[‘t ‘e] =
[~=  ~ite[~Leaf.set  ‘t ‘e] ~null];

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.Leaf.set.ite[‘t ‘e] is true.


8.25.8 TREE SET

Tree set large composition axiom: For a normalized large function ‘x:
ax.Tree.set.co.lg[‘x] = [~=  [~Tree.set  ‘x] ~Tree.set];

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Tree.set.co.lg[‘x] is true.

Proof. ‘Func.Lg.Ext.Tree.set is unchanging.

Tree set null predicate axiom:
ax.Tree.set.Null = [~=  [~Null ~Tree.set] 0];


Proof. For each tagged small function extension ‘x:
‘Func.Lg.Ext.zero(‘x).

Tree set pair predicate axiom:
ax.Tree.set.Pair = [~=  [~Pair ~Tree.set] 0];

‘Func.Lg.Ext.ax.Tree.set.Pair is true.


Tree set domain axiom:
ax.Tree.set.dom = [~=  [~dom ~Tree.set] ~Tree.set];
'Func.Lg.Ext.ax.Tree.set.dom is true.

**Proof.** For each tagged small function extension 'x, 'Func.Lg.Ext.Tree.set('x) = 'Func.Sm.Ext.Tagged.Tree.set is a rule tagged small function extension and an identity.

**Tree set small composition axiom:** For a normalized large function 'x:

\[
ax.Tree.set.co.sm['x] = \\
[\sim (\sim Tree.set 'x) \sim Tree.set.res 'x]];
\]

For a large function extension 'x, 'Func.Lg.Ext.ax.Tree.set.co.sm['x] is true.


**Tree set if-then-else axiom:** For normalized large functions 't and 'e:

\[
ax.Tree.set.ite['t 'e] = \\
[\sim \simite[\sim Tree.set 't 'e] \simnull];
\]

For large function extensions 't and 'e, 'Func.Lg.Ext.ax.Tree.set.ite['t 'e] is true.


8.25.9 LARGE COMPOSITION

**Large composition large composition axiom:** For normalized large functions 'outer, 'inner and 'x:

\[
ax.co.lg.co.lg['outer 'inner 'x] = \\
[\sim [[[\sim 'outer 'inner] 'x] \sim 'outer [[inner 'x]]] ];
\]

For large function extensions 'outer, 'inner and 'x, 'Func.Lg.Ext.ax.co.lg.co.lg['outer 'inner 'x] is true.

**Proof.** For each tagged small function extension 'y: [[['outer 'inner] 'x]('y) = [['outer 'inner]('x('y)) = 'outer('inner('x('y))) = 'outer([inner 'x]('y)) = ['outer ['inner 'x]]('y).
8.25.10 SMALL COMPOSITION

Small composition large composition axiom: For normalized large functions 'called, 'arg and 'x:
\[
\text{ax.co.lg.co.sm['called 'arg 'x]} = \\
[ \sim = [ [ 'called 'arg 'x] [ 'arg 'x] ] ];
\]

For large function extensions 'called, 'arg and 'x, 'Func.Lg.Ext.ax.co.lg.co.sm['called 'arg 'x] is true.

Proof. For each tagged small function extension 'y:
\[
[ 'called 'arg 'x]('y) = ('called 'arg)('y) = \text{[called ('y))('arg('y)) = } [ 'called 'x]('y)([ 'arg 'x]('y)) = ([ 'called 'x][ 'arg 'x]) ('y).
\]

8.25.11 PAIR

Pair large composition axiom: For normalized large functions 'left, 'right and 'x:
\[
\text{ax.pair.co.lg['left 'right 'x]} = \\
[ \sim = [ [ 'left 'right 'x] [ 'left 'x] [ 'right 'x] ] ];
\]

For large function extensions 'l, 'r and 'x, 'Func.Lg.Ext.ax.pair.co.lg['l 'r 'x] is true.

Proof. For each tagged small function extension 'y:
\[
[ 'left 'right 'x]('y) = [ 'left 'x]('y) = [ 'left 'x]'('y), [ 'right 'x]('y) = [ 'right 'x]'('y).
\]

Pair null predicate axiom: For normalized large functions 'left and 'right:
\[
\text{ax.pair.Null['left 'right]} = \\
[ \sim = [ \text{Null } ] ];
\]

For large function extensions 'l and 'r, 'Func.Lg.Ext.ax.pair.Null['l 'r] is true.

Proof. For each tagged small function extension 'x:
\[
[ 'left 'right]('x) = [ 'left 'x]('x) = [ 'left 'x]'('x), [ 'right 'x]('x) = [ 'right 'x]'('x).
\]

Pair pair predicate axiom: For normalized large functions 'left and 'right:
\[
\text{ax.pair.Pair['left 'right]} = \\
[ \sim = [ \text{Pair } ] ];
\]

For large function extensions 'l and 'r, 'Func.Lg.Ext.ax.pair.Pair['l 'r] is true.
Proof. For each tagged small function extension \(x\): \([\text{Func.Lg.Ext.Pair} \{l \ r\}](x) = \text{Func.Lg.Ext.Pair}([l(x), r(x)]) = \text{Func.Sm.Ext.one} = \text{Func.Lg.Ext.one}(x)\). □

**Pair domain axiom**: For normalized large functions \(l\) and \(r\):

\[
\text{ax.pair.dom}[l \ r] = \left[ \sim \text{dom} \{l, r\} \sim \text{Leaf.set} \right];
\]

For large function extensions \(l\) and \(r\), \(\text{Func.Lg.Ext.ax.pair.dom}[l \ r]\) is true.

Proof. For each tagged small function extension \(x\): \([\text{Func.Lg.Ext.dom} \{l \ r\}](x) = \text{dom-FuncExt}([l(x), r(x)]) = \text{Func.Sm.Ext.Tagged.Leaf.set} = \text{Func.Lg.Ext.Leaf.set}(x)\). □

**Pair small composition axiom**: For normalized large functions \(l\), \(r\) and \(x\):

\[
\text{ax.pair.co.sm}[l \ r \ x] = \left[ \sim \{l, r\} \sim x \sim \text{ite}[x \ r \ l] \right];
\]

For large function extensions \(l\), \(r\) and \(x\), \(\text{Func.Lg.Ext.ax.pair.co.sm}[l \ r \ x]\) is true.

Proof.

- \([l \ r \ x](y) = [l \ r](y)(x(y)) = [l(x), r(x)](y)\). \([l \ r \ x](y)\) is given by one of the following mutually exclusive cases:
  - \(l(x)\) if \(x(y) = \text{Func.Sm.Ext.zero}\)
  - \(r(x)\) if \(x(y) = \text{Func.Sm.Ext.one}\)
  - \(\text{Func.Sm.Ext.null}\) if \(x(y)\) is not Boolean

- \(\sim \text{ite}[x \ r \ l](y)\) is given by one of the following mutually exclusive cases:
  - \(l(y)\) if \(x(y) = \text{Func.Sm.Ext.zero}\)
  - \(r(y)\) if \(x(y) = \text{Func.Sm.Ext.one}\)
  - \(\text{Func.Sm.Ext.null}\) if \(x(y)\) is not Boolean

- \([l \ r \ x](y) = \sim \text{ite}[x \ r \ l](y)\). □

**Pair if-then-else axiom**: For normalized large functions \(l\), \(r\), \(t\) and \(e\):

\[
\text{ax.pair.ite}[l \ r \ t \ e] = \left[ \sim \text{ite}[l, r, t \ e] \sim \text{null} \right];
\]

For large function extensions \(l\), \(r\), \(t\) and \(e\), \(\text{Func.Lg.Ext.ax.pair.ite}[l \ r \ t \ e]\) is true.
Proof. For each tagged small function extension ‘x: {'l 'r}('x) = {'l('x), 'r('x)). ˜ite[{'l 'r} 't 'e]('x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null('x).

8.25.12 DEPENDENT SUM

Dependent sum large composition axiom: For normalized large functions ‘family and ‘x:
ax.s.d.co.lg['family 'x] =
[ =~ [~s.d['family] 'x] ~s.d[[‘family ‘x]] ];

For large function extensions ‘family and ‘x, ‘Func.Lg.Ext.ax.s.d.co.lg['family ‘x] is true.

Proof. For each tagged small function extension ‘y: [˜s.d[‘family ‘x]('y) =
˜s.d[‘family]('y) = ‘sumDep(‘family('y)) = ‘sumDep(‘sumDep(‘family ‘x)) = ˜s.d[‘family ‘x]('y).

Dependent sum null predicate axiom: For a normalized large function ‘family:
ax.s.d.Null[‘family] = [~ = [~Null ~s.d[~‘family]] 0];

For a large function extension ‘family, ‘Func.Lg.Ext.ax.s.d.Null['family] is true.


Dependent sum pair predicate axiom: For a normalized large function ‘family:
ax.s.d.Pair[‘family] = [~ = [~Pair ~s.d[‘family]] 0];

For a large function extension ‘family, ‘Func.Lg.Ext.ax.s.d.Pair[‘family] is true.


Dependent sum domain axiom: For a normalized large function ‘family:
ax.s.d.dom[‘family] =
[ =~ [~dom ~s.d[~‘family]] ~s.d[‘family] ];

For a large function extension ‘family, ‘Func.Lg.Ext.ax.s.d.dom[‘family] is true.
Proof. For each tagged small function extension \( x \), \( \sim s.d[\text{family}](x) = \text{sumDep(}\text{family}(x)) \) is a rule tagged small function extension and an identity. □

**Dependent sum small composition axiom**: For normalized large functions \( \text{family} \) and \( x \):

\[
\text{ax.s.d.co.sm}[\text{family } x] = \left[ \sim = (\sim s.d[\text{family } x]) \sim s.d[\text{family } x] \right];
\]

For large function extensions \( \text{family} \) and \( x \), \( \text{Func.Lg.Ext.ax.s.d.co.sm}[\text{family } x] \) is true.

Proof. For each tagged small function extension \( y \): \( \sim s.d[\text{family } x](y) = \sim s.d[\text{family } y](x(y)) = \text{sumDep(}\text{family}(y))(x(y)) = \text{Func.Lg.Ext.s.d.res}((\text{family}(y), x(y))) = \text{Func.Lg.Ext.s.d.res} \text{family } x(y). \)

**Dependent sum if-then-else axiom**: For normalized large functions \( \text{family}, t \) and \( e \):

\[
\text{ax.s.d.ite}[\text{family } t e] = \left[ \sim = \sim \text{ite}(\sim s.d[\text{family } t e] \sim \text{null}] \right];
\]

For large function extensions \( \text{family} \), \( t \) and \( e \), \( \text{Func.Lg.Ext.ax.s.d.ite}[\text{family } t e] \) is true.

Proof. For each tagged small function extension \( x \): \( \sim s.d[\text{family } x](x) = \sim s.d[\text{family } x](t e)(x) = \text{Func.Sm.Ext.null = Func.Lg.Ext.null}(x). \)

8.25.13 DEPENDENT PRODUCT

**Dependent product large composition axiom**: For normalized large functions \( \text{family} \) and \( x \):

\[
\text{ax.p.d.co.lg}[\text{family } x] = \left[ \sim = \sim p.d[\text{family } x] \sim p.d[\text{family } x] \right];
\]

For large function extensions \( \text{family} \) and \( x \), \( \text{Func.Lg.Ext.ax.p.d.co.lg}[\text{family } x] \) is true.
\textit{Proof}. For each tagged small function extension 'y: [\texttt{p.d['family 'x]}('y) = \\
\texttt{p.d['family](x('y)) = 'prodDep('family('x('y))) = 'prodDep([\texttt{family 'x]}('y)) = \texttt{p.d['family 'x]]('y)}.

\textbf{Dependent product null predicate axiom}: For a normalized large function 'family:
\begin{align*}
\text{ax.p.d.Null['family]} &= \sim \text{Null} \sim \text{p.d['family]} 0; \\
\text{For a large function extension 'family, 'Func.Lg.Ext.ax.p.d.Null['family]} &\text{ is true.}
\end{align*}

\textit{Proof}. For each tagged small function extension 'x: ['Func.Lg.Ext.Null \texttt{p.d['family]}('x)
\texttt{=} 'Func.Lg.Ext.Null(\texttt{p.d['family]('x)) = 'Func.Lg.Ext.Null('prodDep('family('x))) =
\texttt{Func.Sm.Ext.zero = 'Func.Lg.Ext.zero('x)}.

\textbf{Dependent product pair predicate axiom}: For a normalized large function 'family:
\begin{align*}
\text{ax.p.d.Pair['family]} &= \sim \text{Pair} \sim \text{p.d['family]} 0; \\
\text{For a large function extension 'family, 'Func.Lg.Ext.ax.p.d.Pair['family]} &\text{ is true.}
\end{align*}

\textit{Proof}. For each tagged small function extension 'x: ['Func.Lg.Ext.Pair \texttt{p.d['family]}('x)
\texttt{=} 'Func.Lg.Ext.Pair(\texttt{p.d['family]('x)) = 'Func.Lg.Ext.Pair('prodDep('family('x))) =
\texttt{Func.Sm.Ext.zero = 'Func.Lg.Ext.zero('x)}.

\textbf{Dependent product domain axiom}: For a normalized large function 'family:
\begin{align*}
\text{ax.p.d.dom['family]} &= \\
\text{For a large function extension 'family, 'Func.Lg.Ext.ax.p.d.dom['family]} &\text{ is true.}
\end{align*}

\textit{Proof}. For each tagged small function extension 'x, \texttt{p.d['family]('x) = 'prodDep('family('x)) is a rule tagged small function extension and an identity.}

\textbf{Dependent product small composition axiom}: For normalized large functions
'family and 'x:
\begin{align*}
\text{ax.p.d.co.sm['family 'x]} &= \\
\text{For large function extensions 'family and 'x, 'Func.Lg.Ext.ax.p.d.co.sm['family 'x]} &\text{ is true.}
\end{align*}
Proof. For each tagged small function extension 'y: (~p.d['family'] 'x)('y) =
~p.d['family]('y)('x('y)) = 'prodDep('family('y))('x('y)) = 'Func.Lg.Ext.p.d.res('family('y),
'x('y))) = ['Func.Lg.Ext.p.d.res 'family 'x]('y).

**Dependent product if-then-else axiom:** For normalized large functions 'family, 't
and 'e:

\[ ax.p.d.ite['family 't 'e] = \]
\[ [\sim \sim p.d['family] 't 'e] ~null; \]

For large function extensions 'family, 't and 'e, 'Func.Lg.Ext.ax.p.d.ite['family 'x] is true.

Proof. For each tagged small function extension 'x: ~p.d['family]('x) =
'prodDep('family('x)). ~ite[~p.d['family] 't 'e]('x) = 'Func.Sm.Ext.null = 'Func.Lg.Ext.null('x).

8.25.14 CURRY

**Curry large composition axiom:** For normalized large functions 'uncurry, 'restrictor
and 'x:

\[ ax.c.co.lg['uncurry 'restrictor 'x] = \]
\[ [\sim c[\sim c['uncurry 'restrictor] 'x]
\sim c.aug[\sim c['uncurry 'restrictor] 'x] \]; \]

For large function extensions 'uncurry, 'restrictor and 'x, 'Func.Lg.Ext.ax.c.co.lg['uncurry 'restrictor 'x] is true.

Proof. For each tagged small function extension 'y: [\sim c['uncurry 'restrictor] 'x]('y) =
\sim c.aug['uncurry 'restrictor]('y).

**Curry null predicate axiom:** For normalized large functions 'uncurry and 'restrictor:

\[ ax.c.Null['uncurry 'restrictor] = \]
\[ [\sim Null \sim c['uncurry 'restrictor]] 0]; \]

For large function extensions 'uncurry and 'restrictor, 'Func.Lg.Ext.ax.c.Null['uncurry 'restrictor] is true.
Proof. For each tagged small function extension \( x \): \( [\text{Func.Lg.Ext.Null } \sim c[\text{uncurry 'restrictor}]](x) = \text{Func.Lg.Ext.Null}(\sim c[\text{uncurry 'restrictor}](x)) = \text{Func.Sm.Ext.zero} = \text{Func.Lg.Ext.zero}(x) \).

Curry pair predicate axiom: For normalized large functions `uncurry` and `restrictor`:

\[
\text{ax.c.Pair}['\text{uncurry 'restrictor}'] =
[~ = [~Pair \sim c[\text{uncurry 'restrictor}]] 0];
\]

For large function extensions `uncurry` and `restrictor`,
\text{Func.Lg.Ext.ax.c.Pair['uncurry 'restrictor']} is true.

Proof. For each tagged small function extension `x`: \( [\text{Func.Lg.Ext.Pair } \sim c[\text{uncurry 'restrictor}]](x) = \text{Func.Lg.Ext.Pair}(\sim c[\text{uncurry 'restrictor}](x)) = \text{Func.Sm.Ext.zero} = \text{Func.Lg.Ext.zero}(x) \).

Curry domain axiom: For normalized large functions `uncurry` and `restrictor`:

\[
\text{ax.c.dom}['\text{uncurry 'restrictor}'] =
[~ =
[~\text{dom } \sim c[\text{uncurry 'restrictor}]]
[~\text{dom } '\text{restrictor}]
];
\]

For large function extensions `uncurry` and `restrictor`,
\text{Func.Lg.Ext.ax.c.dom['uncurry 'restrictor']} is true.

Proof. For each tagged small function extension `x`: \( [\text{Func.Lg.Ext.dom } \sim c[\text{uncurry 'restrictor}]](x) = \text{domFuncExt}(\sim c[\text{uncurry 'restrictor}](x)) = \text{domFuncExt('restrictor}(x)) = [\text{Func.Lg.Ext.dom 'restrictor}](x) \).

Curry small composition axiom: For normalized large functions `uncurry`, `restrictor` and `x`:

\[
\text{ax.c.co.sm}['\text{uncurry 'restrictor 'x}'] =
[~ =
(\sim c[\text{uncurry 'restrictor} 'x])
[\sim c.res[\text{uncurry} ~i '\text{restrictor 'x}]
];
\]

For large function extensions `uncurry`, `restrictor` and `x`,
\text{Func.Lg.Ext.ax.c.co.sm['uncurry 'restrictor 'x]} is true.
\textbf{Proof.} For each tagged small function extension \( y: (\sim c[\text{uncurry restrictor}] x)(y) = \sim c[\text{uncurry restrictor}](y)(x(y)) = \text{uncurry}((y, \text{domFuncExt}(\text{restrictor}(y))(x(y)))) = \sim c.\res[\text{uncurry}](\{y, \text{restrictor}(y), x(y)\}) = [\sim c.\res[\text{uncurry}] \Func.Lg.Ext.i \text{ restrictor } x](y). \)

\textbf{Curry if-then-else axiom:} For normalized large functions \( \text{uncurry}, \text{restrictor}, t \) and \( e \):

\begin{align*}
\text{ax.c.ite}[\text{uncurry restrictor } t e] &= \\
&= [= \sim \text{ite}[\sim c[\text{uncurry restrictor}] t e] \sim \text{null}];
\end{align*}

For large function extensions \( \text{uncurry}, \text{restrictor}, t \) and \( e \), \( \Func.Lg.Ext.ax.c.ite[\text{uncurry restrictor } t e] \) is true.

\textbf{Proof.} For each tagged small function extension \( x: \sim \text{ite}[\sim c[\text{uncurry restrictor}] t e](x) = \text{Func.Sm.Ext.null} = \text{Func.Lg.Ext.null}(x). \)

\section{8.25.15 IF-THEN-ELSE}

\textbf{If-then-else large composition axiom:} For normalized large functions \( \text{ifP}, \text{thenP}, \text{elseP} \) and \( x \):

\begin{align*}
\text{ax.ite.co.lg}[\text{ifP thenP elseP } x] &= \\
&= [= \sim \text{ite}[\sim \text{ite}[\sim \text{ite}[\sim \text{ite}[\sim \text{ite}[\sim \text{ite}[\sim \text{ite}[\sim \text{ite}[\text{ifP thenP elseP } x]] x]] x]] x]] x]] x]] x]] x]] x]] x]] x]] x]] x]
\end{align*}

For large function extensions \( \text{ifP}, \text{thenP}, \text{elseP} \) and \( x \), \( \Func.Lg.Ext.ax.ite.co.lg[\text{ifP thenP elseP } x] \) is true.

\textbf{Proof.}

\begin{itemize}
\item For each tagged small function extension \( y: [\sim \text{ite}[\text{ifP thenP elseP } x](y) = \sim \text{ite}[\text{ifP thenP elseP } x](y)). [\sim \text{ite}[\text{ifP thenP elseP } x](y) \) is given by one of the following mutually exclusive cases:
\begin{itemize}
\item \( \text{elseP}(x(y)) \) if \( \text{ifP}(x(y)) = \text{Func.Sm.Ext.zero} \)
\item \( \text{thenP}(x(y)) \) if \( \text{ifP}(x(y)) = \text{Func.Sm.Ext.one} \)
\item \( \text{Func.Sm.Ext.null} \) if \( \text{ifP}(x(y)) \) is not Boolean
\end{itemize}
\end{itemize}
• \( \texttt{ite}([\texttt{ifP } x] [\texttt{thenP } x] [\texttt{elseP } x])(y) \) is given by one of the following mutually exclusive cases:
  
  - \( [\texttt{elseP } x](y) \) if \( [\texttt{ifP } x](y) = '\text{Func.Sm.Ext.zero} \)
  
  - \( [\texttt{thenP } x](y) \) if \( [\texttt{ifP } x](y) = '\text{Func.Sm.Ext.one} \)
  
  - '\text{Func.Sm.Ext.null} if \( [\texttt{ifP } x](y) \) is not Boolean

• \( \neg \texttt{ite}('\text{ifP } x '\text{thenP } x '\text{elseP } x') = \neg \texttt{ite}('\text{ifP } x x '\text{thenP } x '\text{elseP } x')(y) \).

### 8.25.16 RECURSION

**Recursion right-hand-side axiom:** For normalized large functions \( '\text{start} \) and \( '\text{step} \):

\[
ax.r.rhs['\text{start} ' '\text{step}] = \\
[\neg \sim \text{r}['\text{start} ' step] \sim \text{r}.rhs['\text{start} ' step]]; \\
\]

For large function extensions \( '\text{start} \) and \( '\text{step} \), \('\text{Func.Lg.Ext.ax.r.rhs['\text{start} ' step]} \) is true.

**Proof.** For each tagged small function extension \( 'x \): \( \sim \text{r}['\text{start} ' step]('x) = \sim \text{r}.rhs['\text{start} ' step]('x) \).

### 8.25.17 PROPOSITIONAL LOGIC

The following are similar to logical axioms 1, 2 and 3 in [36, p.5]. Propositional logic in NummSquared is classical.

**Logic weakening axiom:**

\[
ax.logic.weakening \{b c\} \ \ \text{l} = \\
[\neg \text{imp} \ %b \ [\neg \text{imp} \ %c \ %b]]; \\
\]

\('\text{Func.Lg.Ext.ax.logic.weakening} \) is true.

**Logic nested implication axiom:**

\[
ax.logic.imp.nested \{b c d\} \ \text{l} = \\
[\neg \text{imp} \\
[\neg \text{imp} \ %b \ [\neg \text{imp} \ %c \ %d]] \\
[\neg \text{imp} \ [\neg \text{imp} \ %b \ %c] \ [\neg \text{imp} \ %b \ %d]] \\
\]
Logic contrapositive axiom:

\[
\text{ax.logic.contrapos } \{\%b \%c\} \ \perp \!\! \perp =
\begin{array}{l}
\quad \text{[imp} \\
\quad \quad \text{[imp} \ [\text{not } \%b] \ [\text{not } \%c]\]}
\quad \text{[imp } \%c \ \%b]
\end{array}
\]

'Func.Lg.Ext.ax.logic.contrapos is true.

8.25.18 TRUTH

Truth introduction axiom:

\[
\text{ax.truth.intro } =
\begin{array}{l}
\quad \text{[imp} \\
\quad \quad \text{[=} \ \text{i } \perp \!\! \perp]}
\quad \text{i}
\end{array}
\]

'Func.Lg.Ext.ax.truth.intro is true.

Proof. For each tagged small function extension 'x: If 'Func.Lg.Ext.eq({'x, 'Func.Sm.Ext.one}) is true, then 'x is true.

Truth elimination axiom:

\[
\text{ax.truth.elim } =
\begin{array}{l}
\quad \text{[imp} \\
\quad \quad \text{i} \\
\quad \quad \quad \text{[=} \ \text{i } \perp \!\! \perp]}
\end{array}
\]

'Func.Lg.Ext.ax.truth.elim is true.

Proof. For each tagged small function extension 'x: If 'x is true, then 'Func.Lg.Ext.eq({'x, 'Func.Sm.Ext.one}) is true.
8.25.19  EQUALS

Equals right-hand-side axiom:

\[ \text{ax.eq.rhs} = [= \; = \; =.rhs]; \]

\text{Func.Lg.Ext.ax.eq.rhs} is true.

Proof. For each tagged small function extension 'x: \text{Func.Lg.Ext.eq('x)} = \text{Func.Lg.Ext.eq.rhs('x)}.

The following is somewhat similar to reflexivity of equality in [30, p.74].

Equals reflexive axiom: For a normalized large function 'x:

\[ \text{ax.eq.reflex['x]} = [= \; 'x \; 'x]; \]

For a large function extension 'x, \text{Func.Lg.Ext.ax.eq.reflex['x]} is true.

Proof. For each tagged small function extension 'y: \text{eq('x,'y')} = \text{eq('x,'y)}.

The following is somewhat similar to substitutivity of equality in [30, p.74]. However, in NummSquared, substitution does not actually take place here.

Equals substitutive axiom: For a normalized large function 'pred:

\[ \text{ax.eq.subst['pred]} \{ = \%x \%y \%z \} \; 1 = \]

\[ [\text{imp} \]

\[ [= \; %y \; %z] \]

\[ [\text{imp} \]

\[ ['pred \; %x \; %y] \]

\[ ['pred \; %x \; %z] \]

\[ ]; \]

For a large function extension 'pred, \text{Func.Lg.Ext.ax.eq.subst['pred]} is true.

Proof. For each tagged small function extension 'w = ('x, 'y, 'z): If \text{Func.Lg.Ext.eq(('y, 'z)} and \text{eq(('x, 'y)} are true, then 'y = 'z, and \text{eq(('x, 'z)} is true.

8.25.20  HILBERT

The following is somewhat similar to Hilbert's transfinite axiom in [4].

Hilbert transfinite axiom: For a normalized large function 'pred:

\[ \text{ax.h.transfinite['pred]} \{ = \%x \%y \} \; 1 = \]
[\text{~imp}  \\
  \text{'pred} \\
  \text{[\text{~exist}['\text{pred}] \ %x]} \\
]\);

For a large function extension 'pred, 'Func.Lg.Ext.ax.h.transfinite['pred] is true.

\textit{Proof}. For each tagged small function extension 'z = {'x, 'y}: If 'pred('z) is true, then \text{~exist}['pred]('x) is true. □

\section*{8.25.21 INDUCTION}

\textbf{Induction axiom:} For a normalized large function 'pred:

ax.induc['pred] = 
  [\text{~imp}  \\
  \text{['pred \ ~null]} \\
  [\text{~imp}  \\
  \text{~induc.case['pred]} \\
  \text{'pred}] 
];

For a large function extension 'pred, 'Func.Lg.Ext.ax.induc['pred] is true.

\textit{Proof}.

\begin{itemize}
  \item If 'pred('Func.Sm.Ext.null) is true, and \text{~induc.case['pred]('Func.Sm.Ext.null)} is true, then for each tagged small function extension 'x, 'pred('x) is true - this is now proved by induction on 'untag('x):
    \begin{itemize}
      \item Holds if 'x = 'Func.Sm.Ext.null.
      \item If 'x \neq 'Func.Sm.Ext.null: For each 'dom('x) program 'y, 'pred('tagged('x, 'y)) and 'pred('x<>'y) are true (by inductive hypothesis).
    \end{itemize}
  \item For each tagged small function extension 'x, if 'pred('Func.Sm.Ext.null) is true, and \text{~induc.case['pred](x) = ~induc.case['pred]('Func.Sm.Ext.null)} is true, then 'pred('x) is true. □
\end{itemize}
8.25.22 LEFTOVERS

Since 'Func.Lg.Ext.Null, 'Func.Lg.Ext.Pair, 'Func.Lg.Ext.dom and if-then-else are above described by cases, some of their general properties are now described.

**Null predicate otherwise axiom:**

\[
\text{ax.Null.otw} = \begin{cases} 
\text{[~imp~} \\
\begin{cases}
\text{[~not.= ~i ~null]} \\
\text{[= ~Null 0]}
\end{cases}
\end{cases}
\text{;}
\]

'Func.Lg.Ext.ax.Null.otw is true.

**Proof.** For each tagged small function extension 'x: If 'Func.Lg.Ext.not.eq({'x, 'Func.Sm.Ext.null}) is true, then 'x ≠ 'Func.Sm.Ext.null, 'Func.Lg.Ext.Null('x) = 'Func.Sm.Ext.zero, and 'Func.Lg.Ext.eq({'Func.Lg.Ext.Null('x), 'Func.Sm.Ext.zero}) is true.

**Pair predicate otherwise axiom:**

\[
\text{ax.Pair.otw} = \begin{cases} 
\text{[~imp~} \\
\begin{cases}
\text{[~not.= ~i (~left ~right)]} \\
\text{[= ~Pair 0]}
\end{cases}
\end{cases}
\text{;}
\]

'Func.Lg.Ext.ax.Pair.otw is true.

**Proof.** For each tagged small function extension 'x: If 'Func.Lg.Ext.not.eq({'x, {'Func.Lg.Ext.left('x), 'Func.Lg.Ext.right('x)}) is true, then 'x is not a pair tagged small function extension, 'Func.Lg.Ext.Pair('x) = 'Func.Sm.Ext.zero, and 'Func.Lg.Ext.eq({'Func.Lg.Ext.Pair('x), 'Func.Sm.Ext.zero}) is true.

**Domain null predicate axiom:**

\[
\text{ax.dom.Null} = [~ = [~Null ~dom] 0];
\]

'Func.Lg.Ext.ax.dom.Null is true.

Domain pair predicate axiom:

ax.dom.Pair = [~= [~Pair ~dom] 0];

‘Func.Lg.Ext.ax.dom.Pair is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair
‘Func.Lg.Ext.zero(‘x).

Domain domain axiom:

ax.dom.dom = [~= [~dom ~dom] ~dom];

‘Func.Lg.Ext.ax.dom.dom is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.dom
‘Func.Lg.Ext.dom](‘x) = domFuncExt(domFuncExt(‘x)) = domFuncExt(‘x) =
‘Func.Lg.Ext.dom(‘x).

Domain if-then-else axiom: For normalized large functions ‘t and ‘e:

ax.dom.ite[‘t ‘e] = [~= ~ite[~dom ‘t ‘e] ~null];

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.dom.ite[‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.dom(‘x)
= domFuncExt(‘x). ite[‘Func.Lg.Ext.dom ‘t ‘e](‘x) = ‘Func.Sm.Ext.null =
‘Func.Lg.Ext.null(‘x).

Domain idempotent axiom: For a normalized large function ‘x:

ax.dom.idempotent[‘x] =
[~= (~dom (~dom ‘x)) (~dom ‘x)];

For a large function extension ‘x, ‘Func.Lg.Ext.ax.dom.idempotent[‘x] is true.

Proof. For each tagged small function extension ‘y: (‘Func.Lg.Ext.dom
(‘Func.Lg.Ext.dom ‘x))(‘y) = domFuncExt(‘y)(domFuncExt(‘y)(‘x(‘y))) = dom-
FuncExt(‘y)(‘x(‘y)) = (‘Func.Lg.Ext.dom ‘x)(‘y).

If-then-else otherwise axiom: For normalized large functions ‘ifP, ‘thenP, ‘elseP:
The following is somewhat similar to the substitution rule in [4]. However, in
Specialization inference: For large function extensions ‘pred and ‘x, if the following is true:

‘pred
then the following is true:

[‘pred ‘x]

Proof.

• For each tagged small function extension ‘y: ‘pred(‘y) is true.

• For each tagged small function extension ‘y: ‘pred(x(‘y)) is true.

8.27 SOME TRUE NORMALIZED LARGE FUNCTIONS

The above true large function extensions are now translated into true normalized large functions.

For a normalized large function ‘x, ax.i.co.lg[‘x] is true.

Proof. ‘ext(ax.i.co.lg[‘x]) = ‘Func.Lg.Ext.ax.i.co.lg[‘ext(‘x)] is true.

The other axioms are similar and are omitted.

8.28 SOME INFERENCES FROM TRUE NORMALIZED LARGE FUNCTIONS

The above inferences from true large function extensions are now translated into inferences from true normalized large functions. Also, substitution inference (which is syntactic) is added.

8.28.1 MODUS PONENS

Modus ponens inference: For normalized large functions ‘b and ‘c, if the following is true:

[~imp ‘b ‘c]

and the following is true:

‘b
then the following is true:
‘c

8.28.2 SPECIALIZATION

Specialization inference: For normalized large functions ‘pred and ‘x, if the following is true:
‘pred
then the following is true:
[‘pred ‘x]

8.28.3 SUBSTITUTION

The following is somewhat similar to substitution in [9, p.69].

Substitution inference: For normalized large functions ‘pred0, ‘pred1, ‘x and ‘y such that subst(‘pred0, ‘pred1, ‘x, ‘y), if the following is true:
[~= ‘x ‘y]
and the following is true:
‘pred0
then the following is true:
‘pred1

‘ext(‘pred0) is true. ‘ext(‘pred1) is true.

8.29 PROOFS

The above true normalized large functions correspond to NummSquared axioms. The above inferences from true normalized large functions correspond to Numm-Squared inferences.

An identity large composition axiom contains a normalized large function ‘x.
An axiom is exactly one of the following:
• an identity large composition axiom
• The other axioms are similar and are omitted.

Proofs are defined inductively.
A proof is exactly one of the following:
• an axiom
• an inference

An **inference** is exactly one of the following:

• a modus ponens inference

• a specialization inference

• a substitution inference

A **modus ponens inference** contains `<b, c, major, minor>` where `b` and `c` are normalized large functions, and `major` and `minor` are proofs.

A **specialization inference** contains `<pred, x, general>` where `pred` and `x` are normalized large functions, and `general` is a proof.

A **substitution inference** contains `<pred0, pred1, x, y, equality, before>` where `pred0`, `pred1`, `x` and `y` are normalized large functions, and `equality` and `before` are proofs.

This concludes the inductive definition.

### 8.30 PROPOSITION AND VALIDITY OF A PROOF, AND SOUNDNESS THEOREM

For a proof `p`, the **proposition** of `p` (a normalized large function), denoted by `prp(p)`, is given by one of the following mutually exclusive cases:

• `ax.i.co.lg[x]` if `p` is an identity large composition axiom containing `x`
• The other axiom cases are similar and are omitted.
• `c` if `p` is a modus ponens inference containing `<b, c, major, minor>`
• `[pred x]` if `p` is a specialization inference containing `<pred, x, general>`
• `pred1` if `p` is a substitution inference containing `<pred0, pred1, x, y, equality, before>`

For a proof `p`, `p follows` iff exactly one of the following holds:
• ‘p is an axiom.

• ‘p is a modus ponens inference containing <‘b, ‘c, ‘major, ‘minor> and ‘prp(‘major) = [~ imp ‘b ‘c], and ‘prp(‘minor) = ‘b.

• ‘p is a specialization inference containing <‘pred, ‘x, ‘general>, and ‘prp(‘general) = ‘pred.

• ‘p is a substitution inference containing <‘pred0, ‘pred1, ‘x, ‘y, ‘equality, ‘before>, ‘subst(‘pred0, ‘pred1, ‘x, ‘y), ‘prp(‘equality) = [~ = ‘x ‘y], and ‘prp(‘before) = ‘pred0.

For a proof ‘p, the property of ‘p being valid is defined by recursion on ‘p:

• If ‘p is an axiom, ‘p is valid iff ‘p follows.

• If ‘p is a modus ponens inference containing <‘b, ‘c, ‘major, ‘minor>, ‘p is valid iff ‘p follows, and ‘major and ‘minor are valid.

• If ‘p is specialization inference containing <‘pred, ‘x, ‘general>, ‘p is valid iff ‘p follows, and ‘general is valid.

• If ‘p is a substitution inference containing <‘pred0, ‘pred1, ‘x, ‘y, ‘equality, ‘before>, ‘p is valid iff ‘p follows, and ‘equality and ‘before are valid.

Validity of a proof is computable.

The soundness theorem: For a proof ‘p, if ‘p is valid, then ‘prp(‘p) is true. (The soundness theorem, as are all theorems of the informal part, is relative to the languages of the informal part.)

Proof:

• By induction on ‘p.

• Holds if ‘p is an axiom.

• If ‘p is a modus ponens inference containing <‘b, ‘c, ‘major, ‘minor>: ‘prp(‘major) = [~ imp ‘b ‘c]. ‘prp(‘minor) = ‘b. ‘major and ‘minor are valid. [~ imp ‘b ‘c] and ‘b are true (by inductive hypothesis). ‘c is true.
• If \( p \) is a specialization inference containing \( \langle \text{\textit{pred}}, \text{\textit{x}}, \text{\textit{general}} \rangle \): \( \text{prp}(\text{\textit{general}}) = \text{\textit{pred}}. \) \( \text{\textit{general}} \) is valid. \( \text{\textit{pred}} \) is true (by inductive hypothesis). \[ \text{\textit{pred}} \text{\textit{x}} \] is true.

• If \( p \) is a substitution inference containing \( \langle \text{\textit{pred}}_0, \text{\textit{pred}}_1, \text{\textit{x}}, \text{\textit{y}}, \text{\textit{equality}}, \text{\textit{before}} \rangle \): \( \text{subst}(\text{\textit{pred}}_0, \text{\textit{pred}}_1, \text{\textit{x}}, \text{\textit{y}}). \) \( \text{prp}(\text{\textit{equality}}) = [\ ~= \text{\textit{x}} \text{\textit{y}}]. \) \( \text{prp}(\text{\textit{before}}) = \text{\textit{pred}}_0. \) \( \text{\textit{equality}} \) and \( \text{\textit{before}} \) are valid. \[ \ ~= \text{\textit{x}} \text{\textit{y}} \] and \( \text{\textit{pred}}_0 \) are true (by inductive hypothesis). \( \text{\textit{pred}}_1 \) is true.

8.31 QUOTED OF A PROOF

The quoted of a proof is a tree normalized large function containing a tag, a list of normalized large function children, and a list of proof children.

For a natural number \( \text{tag} \), and normalized large functions \( \text{children}_0 \) and \( \text{children}_1 \), the tree of \( \text{tag} \), \( \text{children}_0 \) and \( \text{children}_1 \), denoted by \( \text{tree}(\text{tag}, \text{children}_0, \text{children}_1) \), is \( \{ \text{norm}(\text{tag}) \text{children}_0 \text{children}_1 \} \).

For a natural number \( \text{tag} \), and \( \text{tree} \) normalized large functions \( \text{children}_0 \) and \( \text{children}_1 \), \( \text{tree}(\text{tag}, \text{children}_0, \text{children}_1) \) is a tree.

Let \( \text{axiomCount} \) be the number of axiom cases.

For a proof \( p \), the \text{tag} of \( p \), denoted by \( \text{tag}(p) \), is given by one of the following mutually exclusive cases:

• 0 if \( p \) is an identity large composition axiom

• The other axiom cases are similar and are omitted.

• \( \text{axiomCount} \) if \( p \) is a modus ponens inference

• \( \text{axiomCount} + 1 \) if \( p \) is a specialization inference

• \( \text{axiomCount} + 2 \) if \( p \) is a substitution inference

For an axiom \( a \), the \text{quoted} of \( a \) (a tree normalized large function), denoted by \( \text{quoted}(a) \), is given by one of the following mutually exclusive cases:

• \( \text{tree}(\text{tag}(a), \lnot\{\text{quoted}(x)\}, \lnot\{\}) \) if \( a \) is an identity large composition axiom containing \( x \)
• The other axiom cases are similar and are omitted.

For a proof \texttt{p}, the \textit{quoted} of \texttt{p} (a tree normalized large function), denoted by \texttt{quoted(p)}, is defined by recursion on \texttt{p}:

- as above if \texttt{p} is an axiom
- \texttt{tree(tag(p), l quoted(b) quoted(c), l quoted(major) quoted(minor))} if \texttt{p} is a modus ponens inference containing \texttt{<b, c, major, minor>}
- \texttt{tree(tag(p), l quoted(pred) quoted(x), l quoted(general))} if \texttt{p} is specialization inference containing \texttt{<pred, x, general>}
- \texttt{tree(tag(p), l quoted(pred0) quoted(pred1) quoted(x) quoted(y), l quoted(equality) quoted(before))} if \texttt{p} is a substitution inference containing \texttt{<pred0, pred1, x, y, equality, before>}

\section*{8.32 PROOF UNQUOTED OF A NORMALIZED LARGE FUNCTION}

For a normalized large function \texttt{f}, the \texttt{proof unquoted} of \texttt{f}, denoted by \texttt{unquotedProof(f)}, is the proof \texttt{p} such that \texttt{quoted(p)} = \texttt{f} if such exists; and \texttt{null} otherwise.

For a normalized large function \texttt{f}, \texttt{unquotedProof(f)} is computable.

For a normalized large function \texttt{f}, \texttt{f} is a \textit{quoted proof} iff \texttt{unquotedProof(f)} \neq \texttt{null}.

For a normalized large function \texttt{f}, \texttt{f} is a quoted proof iff there exists a proof \texttt{p} such that \texttt{quoted(p)} = \texttt{f}.

For a normalized large function \texttt{f}, if \texttt{f} is quoted proof, then \texttt{f} is a tree.

For a normalized large function \texttt{f}, \texttt{f} is a \textit{valid quoted proof} iff \texttt{f} is a quoted proof and \texttt{unquotedProof(f)} is valid.

Proofs never appear directly in NummSquared programs. Instead, quoted proofs are created and manipulated by functions (small and large). When necessary, a quoted proof may be unquoted for validity checking.

\section*{8.33 RUSSELL'S PARADOX AVERTED}

It is interesting to examine how NummSquared averts Russell's paradox.
Rus = (\sim i \sim i);

Rus.sm = \sim \text{restrict}[\text{Rus}];

Rus.paradox = [\text{Rus} \text{Rus.sm}];

Of course, ‘\text{Func.Lg.Ext.Rus}(\text{Func.Lg.Ext.Rus})’ cannot be constructed since
‘\text{Func.Lg.Ext.Rus}’ is a large function extension.

For a tagged small function extension ‘x’, ‘\text{Func.Lg.Ext.Rus}(x)’ = ‘x(x).

For a tagged small function extension ‘x’, ‘\text{Func.Lg.Ext.Rus.sm}(x)’ is the rule tagged
small function extension ‘r’ such that ‘\text{domExt}(r)’ = ‘\text{domExt}(x)’ and, for each ‘\text{dom}(r)
program ‘y’, ‘r<’y>’ = ‘\text{Func.Lg.Ext.Rus}(\text{tagged}(r, ‘y))’ = ‘\text{tagged}(r, ‘y)(\text{tagged}(r, ‘y))’ =
‘\text{tagged}(x, ‘y)(\text{tagged}(x, ‘y)).

For tagged small function extensions ‘x’ and ‘y’, ‘\text{Func.Lg.Ext.Rus.sm}(x)(y)
= ‘\text{Func.Lg.Ext.Rus.sm}(x)<‘\text{coer}(\text{Func.Lg.Ext.Rus.sm}(x), ‘y)>’ =
‘\text{Func.Lg.Ext.Rus.sm}(x)<‘\text{coer}(x, ‘y)>’ = ‘\text{tagged}(x, ‘\text{coer}(x, ‘y))(‘\text{tagged}(x, ‘\text{coer}(x, ‘y))).

For a tagged small function extension ‘x’, ‘\text{Func.Lg.Ext.Rus.paradox}(x)
= ‘\text{Func.Lg.Ext.Rus}(\text{Func.Lg.Ext.Rus.sm}(x))’ =
‘\text{Func.Lg.Ext.Rus}(x)(‘\text{Func.Lg.Ext.Rus.sm}(x))’ = ‘\text{tagged}(x, ‘\text{coer}(x,
‘\text{Func.Lg.Ext.Rus.sm}(x)))’ (‘\text{tagged}(x, ‘\text{coer}(x, ‘\text{Func.Lg.Ext.Rus.sm}(x)))).

‘\text{Func.Sm.Ext.null}(\text{Func.Sm.Ext.null})’ = ‘\text{Func.Sm.Ext.null}.

Thus the result of Russell’s paradox is ‘\text{Func.Sm.Ext.null}, and Russell’s paradox does
not cause any logical or computational problems.
CHAPTER 9

THE FORMAL PART

9.1 PREFACE TO THE FORMAL PART

Poohbist.Nummsquared.Preface

9.1.1 THE FORMAL PART

What follows is the formal part (work in progress) defining NummSquared within a Coq program.

9.1.2 A QUICK SURVEY OF COQ

A quick survey of some relevant aspects of Coq is provided here. These informal comments are purely explanatory. [8] is the complete and definitive reference on Coq. For a tutorial on Coq, see [23].

9.1.2.1 COQ TERMS, CONTEXTS, ENVIRONMENTS, TYPE-CHECKING, REDUCTION, NORMAL FORMS AND CONVERTIBILITY

Coq terms are defined in [8, section 4.1.3].

A Coq context is a list of variable declarations. A Coq environment is a list of global declarations. (See [8, section 4.2].) In NummSquared Formally, a Coq e-context is <'e, 'c> where 'e is an environment and 'c is a context.
In NummSquared Formally, for an e-context ‘ec = <‘e, ‘c>, then ‘ec is defined to be well-formed iff ‘e is well-formed, and ‘c is valid in ‘e (see [8, section 4.2] for further explanation). For an e-context ‘ec, and terms ‘t and ‘T, [8, section 4.2] defines whether or not ‘t type-checks as ‘T in ‘ec. For an e-context ‘ec, and terms ‘t and ‘T, one writes ‘t:‘T in ‘ec iff ‘t type-checks as ‘T in ‘ec.

In NummSquared Formally, for an e-context ‘ec, and a term ‘t, then ‘T is defined to be a type for ‘t in ‘ec iff ‘T is a term and ‘t:‘T in ‘ec. In NummSquared Formally, for an e-context ‘ec, and a term ‘t, then ‘t is defined to type-check in ‘ec iff there exists some type for ‘t in ‘ec. In NummSquared Formally, a Coq term-in-context is <‘ec, ‘t> where ‘ec is a well-formed e-context, and ‘t is a term that type-checks in ‘ec. In NummSquared Formally, for a term-in-context ‘tc = <‘ec, ‘t>, then ‘T is defined to be a type for ‘tc iff ‘T is a type for ‘t in ‘ec.

In NummSquared Formally, for an e-context ‘ec, and terms ‘t0 and ‘t1, then ‘t0 is defined to one-step reduce to ‘t1 in ‘ec iff ‘t0 |> ‘t1 in ‘ec (see [8, section 4.3] for further explanation).

For an e-context ‘ec, and a term ‘t0, then ‘t0 is a normal form in ‘ec iff there exists no term ‘t1 to which ‘t0 one-step reduces in ‘ec. For an e-context ‘ec, and terms ‘t0 and ‘t1, then ‘t0 and ‘t1 are convertible in ‘ec iff there exists some term ‘t2 such that ‘t0 and ‘t1 both zero-or-more-step reduce to ‘t2 in ‘ec. (See [8, section 4.3].)

9.1.2.2 COQ SORTS

A Coq sort is one of the following three Coq terms: Prop, Set and Type. (Actually, Coq internally replaces each occurrence of Type by one sort in an infinite hierarchy of sorts indexed by the natural numbers.) (See [8, section 4.1.1] for more on sorts.)

For an e-context ‘ec, and a term ‘t, then:

- ‘t is a proposition in ‘ec iff ‘t:Prop in ‘ec

- ‘t is a set in ‘ec iff ‘t:Set in ‘ec

- ‘t is a type in ‘ec iff ‘t:Type in ‘ec (for some replacement of Type)

In any well-formed e-context, Prop:Type and Set:Type (for any replacements of Type). Furthermore, for a well-formed e-context ‘ec, and a term ‘t, if ‘t is a proposition or set in ‘ec, then ‘t:Type in ‘ec (for any replacement of Type). (See [8, sections 4.2,
Thus, for a well-formed e-context \( ec \), and a term \( t \), then \( t \) is a type in \( ec \) iff \( t : s \) for some sort \( s \).

9.1.2.3 COQ PROOFS

For an e-context \( ec \), and terms \( P \) and \( p \), if \( P \) is a proposition in \( ec \), then \( p \) is a proof of \( P \) in \( ec \) iff \( p : P \) in \( ec \). Thus, in Coq, proof checking is a special case of type checking. (See [8, "Introduction", section 4.1.1].)

For an e-context \( ec \), and a term \( P \); if \( P \) is a proposition in \( ec \), then proving \( P \) means writing some term \( p \) such that \( p \) is a proof of \( P \) in \( ec \).

9.1.2.4 COQ DEPENDENT PRODUCTS, FUNCTIONS AND APPLICATIONS

For a term \( A \), a simple identifier \( x \), and a term \( B \) (which may include \( x \)), the Coq term \( \forall (x : A), B \) is the dependent product (a.k.a. dependent function space) from \( x : A \) to \( B \).

For a term \( A \), a simple identifier \( x \), and a term \( b \) (which may include \( x \)), the Coq term \( \text{fun}(x : A) \Rightarrow b \) is the function that maps \( x : A \) onto \( b \).

For terms \( f \) and \( a \), the Coq term \( (f \ a) \) is the application of \( f \) to \( a \).

(See [8, sections 4.1.3, 4.2] for more on dependent products, functions and applications.)

9.1.2.5 COQ TYPE CASTS

For terms \( t \) and \( T \), the Coq term \( t : T \) is a type cast. For an e-context \( ec \), if \( t : T \) in \( ec \), then \( (t : T) : T \) in \( ec \). (See [8, section 1.2.10].) Note that if \( T0 \) is a type for \( (t : T) \) in \( ec \), then \( T0 \) is also a type for \( t \) in \( ec \). Thus a type cast does not give a term any new types. However, a type cast is useful for checking that a desired type for a term is indeed a type for that term.
9.1.2.6 COQ MODULES, COMMANDS AND GLOBAL DECLARATIONS

A Coq file-level module is a list of Coq commands. A Coq file-level module may be hierarchically organized into Coq intra-file modules. (There are Coq commands for starting and ending a Coq intra-file module.) A Coq intra-file module is also a list of Coq commands. (See [8, sections 2.4, 2.5] for more on Coq intra-file and file-level modules.)

Among the Coq commands are global declarations. In NummSquared Formally, global declarations include global assumptions, global definitions and inductive definitions. (See [8, section 4.2]. In [8, section 1.3], "declaration" means just assumption.)

9.1.2.7 NAMING OF COQ MODULES AND GLOBAL DECLARATIONS

A Coq qualified identifier is a list of one or more simple identifiers, separated by periods (.). (See [8, section 1.2.1].)

A file-level module has, as its short name, the simple identifier ‘x corresponding to the filename (excluding the extension). However, the file-level module has, as its absolute name, the qualified identifier obtained by prefixing ‘x with a particular relative path. (See [8, section 2.5.1].) For example, you are now reading the file-level module whose absolute name is PoohbistTechnology.NummsSquared.v2006a0.Preface. The short name of PoohbistTechnology.NummsSquared.v2006a0.Preface is Preface.

An intra-file module or global declaration has, as its short name, a simple identifier ‘x. (See [8, sections 1.3, 2.4].) However, the intra-file module or global declaration has, as its absolute name, the qualified identifier obtained by prefixing ‘x with the absolute name of the containing file-level module or intra-file module. (See [8, section 2.5.2].)

For a file-level module, intra-file module or global declaration ‘g, a qualified name of ‘g is a non-empty suffix of the absolute name of ‘g. (See [8, section 2.5.2].)

9.1.3 NUMMSQUARED FORMALLY STYLE

{NummSquared Formally Style} is a particular style of using Coq, and is used throughout the body of NummSquared Formally. NummSquared Formally Style is not defined in the formal part of NummSquared Formally, but some rules are given in informal comments.
9.1.3.1 MAKE DESIRED TYPES EXPLICIT USING TYPE CASTS

For clarity, each dependent product $(\forall (x : A), B)$ is written within a type cast $(\forall (x : A), B) : s$ such that $s$ is a sort.

For clarity, each function $\text{fun}(x1 : A) \Rightarrow b$ is written within a type cast $(\text{fun } x1 \Rightarrow b) : ( (\forall (x0 : A), B) : s )$. Note that $x0 : A$ is written as part of the dependent product, and Coq can therefore infer $x1 : A$ for the function.

9.1.3.2 USE TYPE, NOT SET

$\text{Set}$ is not be used. $\text{Type}$ is used instead. ($\text{Type}$ is more flexible because Coq internally replaces each occurrence of $\text{Type}$ by one sort in an infinite hierarchy of sorts.)

9.1.3.3 MAKE REUSABLE TERMS INTO SEPARATE GLOBAL DECLARATIONS

Each dependent product or function is the content of a separate global declaration, so the dependent product or function can be reused.

Coq local definitions (see [8, section 1.2.12]) are not used, since they are less reusable than global definitions.

9.1.3.4 USE UNDERSCORE FOR HIERARCHICAL NAMING

The underscore character (_) is used as a separator to create hierarchical names within simple identifiers. (Although intra-file modules could be used to create qualified names, that scheme would require the hierarchical naming structure to correspond to the order of definitions, which is not always the case.)

The suffix $\_Ty$ indicates a type. This suffix is used only when it is needed to distinguish a type.

9.2 FUNDAMENTALS: OPERATORS: MAIN


9.2.1 OPERATORS

An operator from $A$ to $B$ is a function from $a : A \to B$.

Definition $Op_Ty := ( \forall (A : Type)(B : Type), Type ) : Type$.

Definition $Op := ( \text{fun } A B \Rightarrow ( \forall (a : A), B ) ) : Op_Ty$.

9.2.2 THE CONSTANT OPERATOR

The constant operator from $A$ to $B$ onto $b : B$ is the operator from $A$ to $B$ mapping $a : A$ onto $b$.

Definition $Op_{\text{const}}_Ty :=$

\[
( \forall (A : Type)(B : Type)(b : B), (Op A B) ) : Type.
\]

Definition $Op_{\text{const}} := ( \text{fun } A B b a \Rightarrow b ) : Op_{\text{const}}_Ty$.

9.2.3 SIMPLE OPERATORS

A simple operator on $A$ is an operator from $A$ to $A$.

Definition $Op_{\text{Simp}}_Ty := ( \forall (A : Type), Type ) : Type$.

Definition $Op_{\text{Simp}} := ( \text{fun } A \Rightarrow (Op A A) ) : Op_{\text{Simp}}_Ty$.

9.2.4 THE IDENTITY SIMPLE OPERATOR

The identity simple operator on $A$ is the simple operator on $A$ mapping $a : A$ onto $a$.

Definition $Op_{\text{Simp-identity}}_Ty := ( \forall (A : Type), (Op_{\text{Simp}} A) ) : Type$.

Definition $Op_{\text{Simp-identity}} := ( \text{fun } A a \Rightarrow a ) : Op_{\text{Simp-identity}}_Ty$.

9.2.5 BINARY OPERATORS

A binary operator from $A0, A1$ to $B$ is an operator from $A0$ to an operator from $A1$ to
\[ B. \]

Definition \( \text{Op\_Bin\_Ty} := ( \forall (A0 : \text{Type})(A1 : \text{Type})(B : \text{Type}), \text{Type}) : \text{Type}. \)

Definition \( \text{Op\_Bin} := (\text{fun A0 A1 B } \Rightarrow (\text{Op\_A0 (Op\_A1 B)})) : \text{Op\_Bin\_Ty}. \)

\[ \textbf{9.2.6 CONNECTIVE BINARY OPERATORS} \]

A connective binary operator from \( A \) to \( B \) is a binary operator from \( A, A \) to \( B \).

Definition \( \text{Op\_Bin\_Conn\_Ty} := (\forall (A : \text{Type})(B : \text{Type}), \text{Type}) : \text{Type}. \)

Definition \( \text{Op\_Bin\_Conn} := (\text{fun A B } \Rightarrow (\text{Op\_Bin A A B})) : \text{Op\_Bin\_Conn\_Ty}. \)

\[ \textbf{9.2.7 SIMPLE BINARY OPERATORS} \]

A simple binary operator on \( A \) is a connective binary operator from \( A, A \) to \( A \).

Definition \( \text{Op\_Bin\_Simp\_Ty} := (\forall (A : \text{Type}), \text{Type}) : \text{Type}. \)

Definition \( \text{Op\_Bin\_Simp} := (\text{fun A } \Rightarrow (\text{Op\_Bin\_Conn A A})) : \text{Op\_Bin\_Simp\_Ty}. \)

\[ \textbf{9.2.8 TRINARY OPERATORS} \]

A trinary operator from \( A0, A1, A2 \) to \( B \) is an operator from \( A0 \) to a binary operator from \( A1, A2 \) to \( B \).

Definition \( \text{Op\_Tri\_Ty} := (\forall (A0 : \text{Type})(A1 : \text{Type})(A2 : \text{Type})(B : \text{Type}), \text{Type}) : \text{Type}. \)

Definition \( \text{Op\_Tri} := (\text{fun A0 A1 A2 B } \Rightarrow (\text{Op\_A0 (Op\_Bin\_Conn A A)})) : \text{Op\_Tri\_Ty}. \)

\[ \textbf{9.2.9 CONNECTIVE TRINARY OPERATORS} \]

A connective trinary operator from \( A \) to \( B \) is a trinary operator from \( A, A, A \) to \( B \).

Definition \( \text{Op\_Tri\_Conn\_Ty} := (\forall (A : \text{Type})(B : \text{Type}), \text{Type}) : \text{Type}. \)

Definition \( \text{Op\_Tri\_Conn} := (\text{fun A B } \Rightarrow (\text{Op\_Tri A A A})) : \text{Op\_Tri\_Conn\_Ty}. \)
9.2.10 SIMPLE TRINARY OPERATORS

A simple trinary operator on $A$ is a connective trinary operator from $A$ to $A$.

Definition $Op_{. \text{Tri}_\text{Simp}_\text{Ty}} := (\forall (A : \text{Type}), \text{Type}) : \text{Type}$.

Definition $Op_{. \text{Tri}_\text{Simp}} := (\text{fun } A \Rightarrow (Op_{. \text{Tri}_\text{Conn} A A ) : Op_{. \text{Tri}_\text{Simp}_\text{Ty}}$.

9.2.11 QUATERNARY OPERATORS

A quaternary operator from $A_0, A_1, A_2, A_3$ to $B$ is an operator from $A_0$ to a trinary operator from $A_1, A_2, A_3$ to $B$.

Definition $Op_{. \text{Quat}_\text{Ty}} :=$

\[
(\forall (A_0 : \text{Type})(A_1 : \text{Type})(A_2 : \text{Type})(A_3 : \text{Type})(B : \text{Type}), \text{Type})
\]

: Type.

Definition $Op_{. \text{Quat}} :=$

\[
(\text{fun } A_0 A_1 A_2 A_3 B \Rightarrow (Op A_0 (Op_{. \text{Tri}_A_1 A_2 A_3 B))) : Op_{. \text{Quat}_\text{Ty}}.
\]

9.2.12 CONNECTIVE QUATERNARY OPERATORS

A connective quaternary operator from $A$ to $B$ is a quaternary operator from $A, A, A$ to $B$.

Definition $Op_{. \text{Quat}_\text{Conn}_\text{Ty}} := (\forall (A : \text{Type})(B : \text{Type}), \text{Type}) : \text{Type}$.

Definition $Op_{. \text{Quat}_\text{Conn}} :=$

\[
(\text{fun } A B \Rightarrow (Op_{. \text{Quat} A A A A B}) : Op_{. \text{Quat}_\text{Conn}_\text{Ty}}.$

9.2.13 SIMPLE QUATERNARY OPERATORS

A simple quaternary operator on $A$ is a connective quaternary operator from $A$ to $A$.

Definition $Op_{. \text{Quat}_\text{Simp}_\text{Ty}} := (\forall (A : \text{Type}), \text{Type}) : \text{Type}$.

Definition $Op_{. \text{Quat}_\text{Simp}} := (\text{fun } A \Rightarrow (Op_{. \text{Quat}_\text{Conn} A A ) : Op_{. \text{Quat}_\text{Simp}_\text{Ty}}$.

9.2.14 QUINARY OPERATORS

A quinary operator from $A_0, A_1, A_2, A_3, A_4$ to $B$ is an operator from $A_0$ to a quaternary operator from $A_1, A_2, A_3, A_4$ to $B$. 

Definition \( Op\_Quin\_Ty := \)
\[
( \forall \\
( A0 : Type) \\
( A1 : Type) \\
( A2 : Type) \\
( A3 : Type) \\
( A4 : Type) \\
( B : Type), \\
Type \\
): Type.
\]

Definition \( Op\_Quin := \)
\[
( \text{fun A0 A1 A2 A3 A4 B } \Rightarrow (Op\_A0 (Op\_Quat A1 A2 A3 A4 B)) ) \\
: Op\_Quin\_Ty.
\]

9.2.15 CONNECTIVE QUINARY OPERATORS

A connective quinary operator from \( A \) to \( B \) is a quinary operator from \( A, A, A, A, A \) to \( B \).

Definition \( Op\_Quin\_Conn\_Ty := ( \forall (A : Type)(B : Type), Type ) : Type. \)

Definition \( Op\_Quin\_Conn := \)
\[
( \text{fun A B } \Rightarrow (Op\_Quin A A A A A B) ) : Op\_Quin\_Conn\_Ty.
\]

9.2.16 SIMPLE QUINARY OPERATORS

A simple quinary operator on \( A \) is a connective quinary operator from \( A \) to \( A \).

Definition \( Op\_Quin\_Simp\_Ty := ( \forall (A : Type), Type ) : Type. \)

Definition \( Op\_Quin\_Simp := ( \text{fun A } \Rightarrow (Op\_Quin\_Conn A A A A) ) : Op\_Quin\_Simp\_Ty. \)

9.3 FUNDAMENTALS: PROPOSITIONS: MAIN


nary propositional predicates, quinary propositional predicates, some fundamental propositional predicates, the true proposition, and the false proposition.

9.3.1 DEPENDENCIES


9.3.2 PROPOSITIONAL PREDICATES

A propositional predicate on $A$ is an operator from $A$ to $Prop$.

Definition $Prp_{Pred} := ( \forall (A : Type), Type ) : Type$.

Definition $Prp_{Pred} := ( fun A \Rightarrow ( Op A Prop ) ) : Prp_{Pred}$. 

9.3.3 THE CONSTANT PROPOSITIONAL PREDICATE

The constant propositional predicate on $A$ onto $P : Prop$ is the constant operator from $A$ to $Prop$ onto $P$.

Definition $Prp_{Pred\_const} := ( \forall (A : Type)(P : Prop), (Prp_{Pred} A) ) : Type$.

Definition $Prp_{Pred\_const} := ( fun A P \Rightarrow (Op\_const A Prop P) ) : Prp_{Pred\_const}$. 

9.3.4 BINARY PROPOSITIONAL PREDICATES

A binary propositional predicate on $A0$, $A1$ is a binary operator from $A0$, $A1$ to $Prop$.

Definition $Prp_{Pred\_Bin} := ( \forall (A0 : Type)(A1 : Type), Type ) : Type$.

Definition $Prp_{Pred\_Bin} := ( fun A0 A1 \Rightarrow (Op\_Bin A0 A1 Prop) ) : Prp_{Pred\_Bin}$.

9.3.5 CONNECTIVE BINARY PROPOSITIONAL PREDICATES

A connective binary propositional predicate on $A$ is a binary propositional predicate on $A$, $A$. 
Definition \( \text{Prp\_Pred\_Bin\_Conn\_Ty} := ( \forall (A : \text{Type}), \text{Type}) : \text{Type} \).

Definition \( \text{Prp\_Pred\_Bin\_Conn} := \\
( \text{fun} A \Rightarrow (\text{Prp\_Pred\_Bin} A A) ) : \text{Prp\_Pred\_Bin\_Conn\_Ty} .

### 9.3.6 TRINARY PROPOSITIONAL PREDICATES

A trinary propositional predicate on \( A_0, A_1, A_2 \) is a trinary operator from \( A_0, A_1, A_2 \) to \( \text{Prop} \).

Definition \( \text{Prp\_Pred\_Tri\_Ty} := \\
( \forall (A_0 : \text{Type})(A_1 : \text{Type})(A_2 : \text{Type}), \text{Type}) : \text{Type} .

Definition \( \text{Prp\_Pred\_Tri} := \\
( \text{fun} A_0 A_1 A_2 \Rightarrow (\text{Op\_Tri} A_0 A_1 A_2 \text{Prop}) ) : \text{Prp\_Pred\_Tri\_Ty} .

### 9.3.7 CONNECTIVE TRINARY PROPOSITIONAL PREDICATES

A connective trinary propositional predicate on \( A \) is a trinary propositional predicate on \( A, A, A \).

Definition \( \text{Prp\_Pred\_Tri\_Conn\_Ty} := ( \forall (A : \text{Type}), \text{Type}) : \text{Type} .

Definition \( \text{Prp\_Pred\_Tri\_Conn} := \\
( \text{fun} A \Rightarrow (\text{Prp\_Pred\_Tri} A A A) ) : \text{Prp\_Pred\_Tri\_Conn\_Ty} .

### 9.3.8 QUATERNARY PROPOSITIONAL PREDICATES

A quaternary propositional predicate on \( A_0, A_1, A_2, A_3 \) is a quaternary operator from \( A_0, A_1, A_2, A_3 \) to \( \text{Prop} \).

Definition \( \text{Prp\_Pred\_Quat\_Ty} := \\
( \forall (A_0 : \text{Type})(A_1 : \text{Type})(A_2 : \text{Type})(A_3 : \text{Type}), \text{Type}) : \text{Type} .

Definition \( \text{Prp\_Pred\_Quat} := \\
( \text{fun} A_0 A_1 A_2 A_3 \Rightarrow (\text{Op\_Quat} A_0 A_1 A_2 A_3 \text{Prop}) ) : \text{Prp\_Pred\_Quat\_Ty} .

### 9.3.9 CONNECTIVE QUATERNARY PROPOSITIONAL PREDICATES

A connective quaternary propositional predicate on \( A \) is a quaternary propositional

Definition $Prp\_Pred\_Quat\_Conn\_Ty := ( \forall (A : Type), Type ) : Type$.

Definition $Prp\_Pred\_Quat\_Conn :=$

$\quad ( \text{fun } A \Rightarrow (Prp\_Pred\_Quat A A A A A)) : Prp\_Pred\_Quat\_Conn\_Ty$.

### 9.3.10 QUINARY PROPOSITIONAL PREDICATES

A quinary propositional predicate on $A0, A1, A2, A3, A4$ is a quinary operator from $A0, A1, A2, A3, A4$ to $\text{Prop}$.

Definition $Prp\_Pred\_Quin\_Ty :=$

$\quad ( \forall (A0 : Type)(A1 : Type)(A2 : Type)(A3 : Type)(A4 : Type), Type )$

$\quad : Type$.

Definition $Prp\_Pred\_Quin :=$

$\quad ( \text{fun } A0 A1 A2 A3 A4 \Rightarrow (Op\_Quin A0 A1 A2 A3 A4 \text{Prop}) )$

$\quad : Prp\_Pred\_Quin\_Ty$.

### 9.3.11 CONNECTIVE QUINARY PROPOSITIONAL PREDICATES

A connective quinary propositional predicate on $A$ is a quinary propositional predicate on $A, A, A, A, A$.

Definition $Prp\_Pred\_Quin\_Conn\_Ty := ( \forall (A : Type), Type ) : Type$.

Definition $Prp\_Pred\_Quin\_Conn :=$

$\quad ( \text{fun } A \Rightarrow (Prp\_Pred\_Quin A A A A A A) ) : Prp\_Pred\_Quin\_Conn\_Ty$.

### 9.3.12 THE TRUE PROPOSITION

There is exactly one proof of the true proposition: the true proposition proof.

$Prp\_T$ is defined in the same way as $\text{True}$ in $\text{Coq.Init.Logic}$.

Inductive $Prp\_T : \text{Prop} :=$

$\mid Prp\_T\_proof : Prp\_T$. 
9.3.13 THE FALSE PROPOSITION

There are no proofs of the false proposition.

$Prp\_F$ is defined in the same way as $False$ in $Coq.Init.Logic$.

Inductive $Prp\_F : Prop :=$ .

9.4 FUNDAMENTALS: BOOLEANS: MAIN


9.4.1 DEPENDENCIES


9.4.2 BOOLEANS

A Boolean is exactly one of the following:

- the true Boolean
- the false Boolean

$Boo$ is defined in the same way as $bool$ in $Coq.Init.Datatypes$, except that $Boo : Type$ whereas $bool : Set$.

Inductive $Boo : Type :=$

| Boo\_t : Boo
| Boo\_f : Boo.
9.4.3 BOOLEAN PREDICATES

A Boolean predicate on \( A \) is an operator from \( A \) to \( Boo \).

Definition \( Boo\_Pred\_Ty := \forall (A : Type), Type : Type. \)

Definition \( Boo\_Pred := (fun A \Rightarrow (Op\ A\ Boo)) : Boo\_Pred\_Ty. \)

9.4.4 THE CONSTANT BOOLEAN PREDICATE

The constant Boolean predicate on \( A \) onto \( b : Boo \) is the constant operator from \( A \) to \( Boo \) onto \( b \).

Definition \( Boo\_Pred\_const\_Ty := \)

\[ (\forall (A : Type)(b : Boo), (Boo\_Pred A)) : Type. \]

Definition \( Boo\_Pred\_const := \)

\[ (fun A b \Rightarrow (Op\_const A\ Boo b)) : Boo\_Pred\_const\_Ty. \]

9.4.5 BINARY BOOLEAN PREDICATES

A binary Boolean predicate on \( A0, A1 \) is a binary operator from \( A0, A1 \) to \( Boo \).

Definition \( Boo\_Pred\_Bin\_Ty := (\forall (A0 : Type)(A1 : Type), Type) : Type. \)

Definition \( Boo\_Pred\_Bin := \)

\[ (fun A0 A1 \Rightarrow (Op\_Bin A0 A1 Boo)) : Boo\_Pred\_Bin\_Ty. \]

9.4.6 CONNECTIVE BINARY BOOLEAN PREDICATES

A connective binary Boolean predicate on \( A \) is a binary Boolean predicate on \( A, A. \)

Definition \( Boo\_Pred\_Bin\_Conn\_Ty := (\forall (A : Type), Type) : Type. \)

Definition \( Boo\_Pred\_Bin\_Conn := \)

\[ (fun A \Rightarrow (Boo\_Pred\_Bin A A)) : Boo\_Pred\_Bin\_Conn\_Ty. \]

9.4.7 TRINARY BOOLEAN PREDICATES

A trinary Boolean predicate on \( A0, A1, A2 \) is a trinary operator from \( A0, A1, A2 \) to \( Boo \).

Definition \( Boo\_Pred\_Tri\_Ty := \)
(∀(A0 : Type)(A1 : Type)(A2 : Type), Type) : Type.

Definition Boo_Pred_Tri :=
  ( fun A0 A1 A2 ⇒ (Op_Tri A0 A1 A2 Boo) ) : Boo_Pred_Tri_Ty.

### 9.4.8 CONNECTIVE TRINARY BOOLEAN PREDICATES

A connective trinary Boolean predicate on A is a trinary Boolean predicate on A, A, A.

Definition Boo_Pred_Tri_Conn_Ty := (∀(A : Type), Type) : Type.

Definition Boo_Pred_Tri_Conn :=
  ( fun A ⇒ (Boo_Pred_Tri A A A) ) : Boo_Pred_Tri_Conn_Ty.

### 9.4.9 QUATERNARY BOOLEAN PREDICATES

A quaternary Boolean predicate on A0, A1, A2, A3 is a quaternary operator from A0, A1, A2, A3 to Boo.

Definition Boo_Pred_Quat_Ty :=
  (∀(A0 : Type)(A1 : Type)(A2 : Type)(A3 : Type), Type) : Type.

Definition Boo_Pred_Quat :=
  ( fun A0 A1 A2 A3 ⇒ (Op_Quat A0 A1 A2 A3 Boo) ) : Boo_Pred_Quat_Ty.

### 9.4.10 CONNECTIVE QUATERNARY BOOLEAN PREDICATES

A connective quaternary Boolean predicate on A is a quaternary Boolean predicate on A, A, A, A.

Definition Boo_Pred_Quat_Conn_Ty := (∀(A : Type), Type) : Type.

Definition Boo_Pred_Quat_Conn :=
  ( fun A ⇒ (Boo_Pred_Quat A A A A) ) : Boo_Pred_Quat_Conn_Ty.

### 9.4.11 QUINARY BOOLEAN PREDICATES

A quinary Boolean predicate on A0, A1, A2, A3, A4 is a quinary operator from A0, A1, A2, A3, A4 to Boo.
Definition $\text{Boo\_Pred\_Quin\_Ty} :=$
\[ \forall (A0 : \text{Type})(A1 : \text{Type})(A2 : \text{Type})(A3 : \text{Type})(A4 : \text{Type}), \text{Type} \]
: Type.

Definition $\text{Boo\_Pred\_Quin} :=$
\[ (\text{fun} A0 A1 A2 A3 A4 \Rightarrow (\text{Op\_Quin} A0 A1 A2 A3 A4 \text{Boo})) \]
: $\text{Boo\_Pred\_Quin\_Ty}.$

### 9.4.12 CONNECTIVE QUINARY BOOLEAN PREDICATES

A connective quinary Boolean predicate on $A$ is a quinary Boolean predicate on $A, A, A, A, A$.

Definition $\text{Boo\_Pred\_Quin\_Conn\_Ty} := (\forall (A : \text{Type}), \text{Type}) : \text{Type}.$

Definition $\text{Boo\_Pred\_Quin\_Conn} :=$
\[ (\text{fun} A \Rightarrow (\text{Boo\_Pred\_Quin} A A A A A) ) : \text{Boo\_Pred\_Quin\_Conn\_Ty}. \]

### 9.4.13 BOOLEAN TO PROPOSITION

$(\text{Boo\_to\_Prp} b)$ is the true proposition if $b$, and the false proposition otherwise.

$\text{Boo\_to\_Prp}$ is defined in the same way as $\text{Is\_true}$ in $\text{Coq.Bool.Bool}$.

Definition $\text{Boo\_to\_Prp} := (\text{fun} b \Rightarrow$
\[ \begin{align*}
  & \text{if} \ b \\
  & \text{return} \ \text{Prop} \\
  & \text{then} \ \text{Prp\_T} \\
  & \text{else} \ \text{Prp\_F} \\
\end{align*} \]
\[ ) : (\text{Prp\_Pred Boo}). \]

### 9.4.14 BOOLEAN EQUALS

$(\text{Boo\_eq} b0 b1)$ is the true Boolean if $b0$ and $b1$ are structurally equal; and the false Boolean otherwise.

Definition $\text{Boo\_eq} := (\text{fun} b0 b1 \Rightarrow$
\[ \begin{align*}
  & \text{match} \ b0, b1 \\
  & \text{return} \ Boo \\
\end{align*} \]

with
| Boo_t, Boo_t ⇒ Boo_t
| Boo_f, Boo_f ⇒ Boo_t
| _, _ ⇒ Boo_f
end
) : (Boo_Pred_Bin_Conn Boo).

9.4.15 BOOLEAN NOT

(Boo_not b) is the false Boolean if b; and the true Boolean otherwise.
Definition Boo_not := (fun b ⇒
if b
return Boo
then Boo_f
else Boo_t
) : (Boo_Pred Boo).

9.5 FUNDAMENTALS: NATURALS: MAIN


9.5.1 DEPENDENCIES


9.5.2 NATURAL NUMBERS

A natural number is exactly one of the following:
• the zero natural number
• for some natural number m, the successor natural number of m
Nat is defined in the same way as nat in Coq.Init.Datatypes, except that Nat : Type whereas nat : Set.

Inductive Nat : Type :=
  | Nat_z : Nat

9.5.3 ABBREVIATIONS FOR SOME NATURAL NUMBERS

Definition Nat_n1 := (Nat_s Nat_z).
Definition Nat_n2 := (Nat_s Nat_n1).
Definition Nat_n3 := (Nat_s Nat_n2).
Definition Nat_n4 := (Nat_s Nat_n3).
Definition Nat_n5 := (Nat_s Nat_n4).
Definition Nat_n6 := (Nat_s Nat_n5).
Definition Nat_n7 := (Nat_s Nat_n6).
Definition Nat_n8 := (Nat_s Nat_n7).
Definition Nat_n9 := (Nat_s Nat_n8).
Definition Nat_n10 := (Nat_s Nat_n9).
Definition Nat_n11 := (Nat_s Nat_n10).
Definition Nat_n12 := (Nat_s Nat_n11).
Definition Nat_n13 := (Nat_s Nat_n12).
Definition Nat_n14 := (Nat_s Nat_n13).
Definition Nat_n15 := (Nat_s Nat_n14).
Definition Nat_n16 := (Nat_s Nat_n15).
Definition Nat_n17 := (Nat_s Nat_n16).
Definition Nat_n18 := (Nat_s Nat_n17).
Definition Nat_n19 := (Nat_s Nat_n18).
Definition Nat_n20 := (Nat_s Nat_n19).
Definition Nat_n21 := (Nat_s Nat_n20).
Definition Nat_n22 := (Nat_s Nat_n21).
Definition Nat_n23 := (Nat_s Nat_n22).
Definition Nat_n24 := (Nat_s Nat_n23).
Definition Nat_n25 := (Nat_s Nat_n24).
Definition \( Nat\_n26 := (Nat\_s Nat\_n25) \).
Definition \( Nat\_n27 := (Nat\_s Nat\_n26) \).
Definition \( Nat\_n28 := (Nat\_s Nat\_n27) \).
Definition \( Nat\_n29 := (Nat\_s Nat\_n28) \).
Definition \( Nat\_n30 := (Nat\_s Nat\_n29) \).
Definition \( Nat\_n31 := (Nat\_s Nat\_n30) \).
Definition \( Nat\_n32 := (Nat\_s Nat\_n31) \).
Definition \( Nat\_n33 := (Nat\_s Nat\_n32) \).
Definition \( Nat\_n34 := (Nat\_s Nat\_n33) \).
Definition \( Nat\_n35 := (Nat\_s Nat\_n34) \).
Definition \( Nat\_n36 := (Nat\_s Nat\_n35) \).
Definition \( Nat\_n37 := (Nat\_s Nat\_n36) \).
Definition \( Nat\_n38 := (Nat\_s Nat\_n37) \).
Definition \( Nat\_n39 := (Nat\_s Nat\_n38) \).
Definition \( Nat\_n40 := (Nat\_s Nat\_n39) \).
Definition \( Nat\_n41 := (Nat\_s Nat\_n40) \).
Definition \( Nat\_n42 := (Nat\_s Nat\_n41) \).
Definition \( Nat\_n43 := (Nat\_s Nat\_n42) \).
Definition \( Nat\_n44 := (Nat\_s Nat\_n43) \).
Definition \( Nat\_n45 := (Nat\_s Nat\_n44) \).
Definition \( Nat\_n46 := (Nat\_s Nat\_n45) \).
Definition \( Nat\_n47 := (Nat\_s Nat\_n46) \).
Definition \( Nat\_n48 := (Nat\_s Nat\_n47) \).
Definition \( Nat\_n49 := (Nat\_s Nat\_n48) \).
Definition \( Nat\_n50 := (Nat\_s Nat\_n49) \).
Definition \( Nat\_n51 := (Nat\_s Nat\_n50) \).
Definition \( Nat\_n52 := (Nat\_s Nat\_n51) \).
Definition \( Nat\_n53 := (Nat\_s Nat\_n52) \).
Definition \( Nat\_n54 := (Nat\_s Nat\_n53) \).
Definition \( Nat\_n55 := (Nat\_s Nat\_n54) \).
Definition \( Nat\_n56 := (Nat\_s Nat\_n55) \).
Definition \( Nat\_n57 := (Nat\_s Nat\_n56) \).
Definition \( Nat\_n58 := (Nat\_s Nat\_n57) \).
Definition \( Nat\_n59 := (Nat\_s Nat\_n58) \).
Definition \( Nat\_n60 := (Nat\_s Nat\_n59) \).
Definition Nat\_n61 := (Nat\_s Nat\_n60).
Definition Nat\_n62 := (Nat\_s Nat\_n61).
Definition Nat\_n63 := (Nat\_s Nat\_n62).
Definition Nat\_n64 := (Nat\_s Nat\_n63).
Definition Nat\_n65 := (Nat\_s Nat\_n64).
Definition Nat\_n66 := (Nat\_s Nat\_n65).
Definition Nat\_n67 := (Nat\_s Nat\_n66).
Definition Nat\_n68 := (Nat\_s Nat\_n67).
Definition Nat\_n69 := (Nat\_s Nat\_n68).
Definition Nat\_n70 := (Nat\_s Nat\_n69).
Definition Nat\_n71 := (Nat\_s Nat\_n70).
Definition Nat\_n72 := (Nat\_s Nat\_n71).
Definition Nat\_n73 := (Nat\_s Nat\_n72).
Definition Nat\_n74 := (Nat\_s Nat\_n73).
Definition Nat\_n75 := (Nat\_s Nat\_n74).
Definition Nat\_n76 := (Nat\_s Nat\_n75).
Definition Nat\_n77 := (Nat\_s Nat\_n76).
Definition Nat\_n78 := (Nat\_s Nat\_n77).
Definition Nat\_n79 := (Nat\_s Nat\_n78).
Definition Nat\_n80 := (Nat\_s Nat\_n79).
Definition Nat\_n81 := (Nat\_s Nat\_n80).
Definition Nat\_n82 := (Nat\_s Nat\_n81).
Definition Nat\_n83 := (Nat\_s Nat\_n82).
Definition Nat\_n84 := (Nat\_s Nat\_n83).
Definition Nat\_n85 := (Nat\_s Nat\_n84).
Definition Nat\_n86 := (Nat\_s Nat\_n85).
Definition Nat\_n87 := (Nat\_s Nat\_n86).
Definition Nat\_n88 := (Nat\_s Nat\_n87).
Definition Nat\_n89 := (Nat\_s Nat\_n88).
Definition Nat\_n90 := (Nat\_s Nat\_n89).
Definition Nat\_n91 := (Nat\_s Nat\_n90).
Definition Nat\_n92 := (Nat\_s Nat\_n91).
Definition Nat\_n93 := (Nat\_s Nat\_n92).
Definition Nat\_n94 := (Nat\_s Nat\_n93).
Definition Nat\_n95 := (Nat\_s Nat\_n94).
Definition $\text{Nat}_n96' := (\text{Nat}_s \text{Nat}_n95$).
Definition $\text{Nat}_n97' := (\text{Nat}_s \text{Nat}_n96$).
Definition $\text{Nat}_n98' := (\text{Nat}_s \text{Nat}_n97$).
Definition $\text{Nat}_n99' := (\text{Nat}_s \text{Nat}_n98$).
Definition $\text{Nat}_n100' := (\text{Nat}_s \text{Nat}_n99$).
Definition $\text{Nat}_n101' := (\text{Nat}_s \text{Nat}_n100$).
Definition $\text{Nat}_n102' := (\text{Nat}_s \text{Nat}_n101$).
Definition $\text{Nat}_n103' := (\text{Nat}_s \text{Nat}_n102$).
Definition $\text{Nat}_n104' := (\text{Nat}_s \text{Nat}_n103$).
Definition $\text{Nat}_n105' := (\text{Nat}_s \text{Nat}_n104$).
Definition $\text{Nat}_n106' := (\text{Nat}_s \text{Nat}_n105$).
Definition $\text{Nat}_n107' := (\text{Nat}_s \text{Nat}_n106$).
Definition $\text{Nat}_n108' := (\text{Nat}_s \text{Nat}_n107$).
Definition $\text{Nat}_n109' := (\text{Nat}_s \text{Nat}_n108$).
Definition $\text{Nat}_n110' := (\text{Nat}_s \text{Nat}_n109$).
Definition $\text{Nat}_n111' := (\text{Nat}_s \text{Nat}_n110$).
Definition $\text{Nat}_n112' := (\text{Nat}_s \text{Nat}_n111$).
Definition $\text{Nat}_n113' := (\text{Nat}_s \text{Nat}_n112$).
Definition $\text{Nat}_n114' := (\text{Nat}_s \text{Nat}_n113$).
Definition $\text{Nat}_n115' := (\text{Nat}_s \text{Nat}_n114$).
Definition $\text{Nat}_n116' := (\text{Nat}_s \text{Nat}_n115$).
Definition $\text{Nat}_n117' := (\text{Nat}_s \text{Nat}_n116$).
Definition $\text{Nat}_n118' := (\text{Nat}_s \text{Nat}_n117$).
Definition $\text{Nat}_n119' := (\text{Nat}_s \text{Nat}_n118$).
Definition $\text{Nat}_n120' := (\text{Nat}_s \text{Nat}_n119$).
Definition $\text{Nat}_n121' := (\text{Nat}_s \text{Nat}_n120$).
Definition $\text{Nat}_n122' := (\text{Nat}_s \text{Nat}_n121$).
Definition $\text{Nat}_n123' := (\text{Nat}_s \text{Nat}_n122$).
Definition $\text{Nat}_n124' := (\text{Nat}_s \text{Nat}_n123$).
Definition $\text{Nat}_n125' := (\text{Nat}_s \text{Nat}_n124$).
Definition $\text{Nat}_n126' := (\text{Nat}_s \text{Nat}_n125$).

9.5.4 NATURAL NUMBER EQUALS

$(\text{Nat}_\text{eq} \text{m}_0 \text{m}_1)$ is the true Boolean if $\text{m}_0$ and $\text{m}_1$ are structurally equal; and the
false Boolean otherwise.

Definition \( \text{Nat}_{\text{eq}} := ( \)
\[
\text{fix } \text{fp}(m_0 : \text{Nat})(m_1 : \text{Nat})\{\text{struct } m_0\} : \text{Boo} := \\
\text{match } m_0, m_1 \\
\text{return Boo} \\
\text{with} \\
| \text{Nat}_z, \text{Nat}_z \Rightarrow \text{Boo}_t \\
| \text{Nat}_s m_0 \text{Pre}, \text{Nat}_s m_1 \text{Pre} \Rightarrow (\text{fp } m_0 \text{Pre } m_1 \text{Pre}) \\
| \_, _ \Rightarrow \text{Boo}_f \\
\text{end} \\
) : (\text{Boo}_\text{Pred}_\text{Bin}_\text{Conn }\text{Nat}).
\]

9.5.5 NATURAL NUMBER ITERATE

\((\text{Nat}_{\text{iter }} A f a m)\) applies \( f \) \( m \)-times starting with \( a \).

Definition \( \text{Nat}_{\text{iter}} \text{Ty} := 
(\forall (A : \text{Type})(f : (\text{Op}_{\text{Simp}} A)), (\text{Op}_{\text{Bin}} A \text{ Nat} A)) : \text{Type}.

Definition \( \text{Nat}_{\text{iter}} := (\text{fun } A f a \Rightarrow 
\text{fix } \text{fp}(m : \text{Nat})\{\text{struct } m\} : A := \\
\text{match } m \\
\text{return } A \\
\text{with} \\
| \text{Nat}_z \Rightarrow a \\
| \text{Nat}_s m \text{Pre} \Rightarrow (f (\text{fp } m \text{Pre})) \\
\text{end} \\
) : \text{Nat}_{\text{iter}} \text{Ty}.
\)

9.5.6 NATURAL NUMBER ADD

\((\text{Nat}_{\text{add }} m n)\) is \( m + n \).

\text{Nat}_{\text{add}} \) is somewhat similar in concept to the plus function in \([28, \text{p.234}]\).

Definition \( \text{Nat}_{\text{add}} := (\text{fun } m \Rightarrow 
\text{fix } \text{fp}(n : \text{Nat})\{\text{struct } n\} : \text{Nat} := \\
\text{match } n 
)
return Nat
with
| Nat_z ⇒ m
| Nat_s nPre ⇒ (Nat_s (fp nPre))
end
) : (Op_Bin_Simp Nat).

9.5.7  NATURAL NUMBER MULTIPLY

(Nat_mult m n) is m * n.

Nat_mult is somewhat similar in concept to the mult function in [28, p.235].

Definition Nat_mult := ( fun m ⇒
fix fp(n : Nat){struct n} : Nat :=
match n
return Nat
with
| Nat_z ⇒ Nat_z
| Nat_s nPre ⇒ (Nat_add (fp nPre) m)
end
) : (Op_Bin_Simp Nat).

9.6  FUNDAMENTALS: NATURALS: EFFICIENT: MAIN


9.6.1  DEPENDENCIES

9.6.2 EFFICIENT NATURAL NUMBERS

Parameter *Nat_Eff*: Type.

9.6.3 EFFICIENT NATURAL NUMBER EQUALS

Parameter *Nat_Eff_eq*: *(Boo_Pred_Bin_Conn Nat_Eff)*.

9.7 FUNDAMENTALS: UNITS: MAIN

*Poohbist.NummSquared.Fundamentals.Units.Main*

*Poohbist.NummSquared.Fundamentals.Units.Main* defines units, and some operators on units.

9.7.1 DEPENDENCIES


9.7.2 UNITS

There is exactly one unit: the unit element.

*Uni* is defined in the same way as *unit* in *Coq.Init.Datatypes*, except that *Uni*: Type whereas *unit*: Set.

Inductive *Uni*: Type :=
| *Uni_elem*: *Uni*.

9.7.3 UNIT EQUALS

*(Uni_eq u0 u1)* is the true Boolean if *u0* and *u1* are structurally equal; and the false Boolean otherwise. Of course, *(Uni_eq u0 u1)* is always the true Boolean.

Definition *Uni_eq*: ( fun *u0 u1* ⇒ *Boo_t* ) : *(Boo_Pred_Bin_Conn Uni)*.
9.8 FUNDAMENTALS: OPTIONALS: MAIN


9.8.1 DEPENDENCIES


9.8.2 OPTIONALS

An optional $A$ is exactly one of the following:

- the none optional $A$
- for some $a : A$, the one optional $A$ containing $a$

*Optional* is defined in the same way as *option* in *Coq.Init.Datatypes*, except that *Optional* is *Type* whereas *option* is *Set*.

Inductive *Optional* ($A : Type$) : Type :=

| *Optional_none* : (Optional $A$) |
| *Optional_one* : (Op $A$ (Optional $A$)). |

9.8.3 OPTIONAL RELATED TO

(*Optional_rel $A_0$ $A_1$ rel01 o0 o1*) is the true Boolean if $o0$ and $o1$ have the same shape, and their corresponding elements $a0 : A0$, $a1 : A1$ satisfy (rel01 a0 a1); and the false Boolean otherwise.

Definition *Optional_rel_Ty* :=

(∀

  (A0 : Type)

  (A1 : Type))
(rel01 : (Boo_Pred_Bin A0 A1)),
(Boo_Pred_Bin (Optional A0) (Optional A1))
)
: Type.

Definition Optional_rel :=
(fun A0 A1 rel01 o0 o1 ⇒
match o0, o1
return Boo
with
| Optional_none, Optional_none ⇒ Boo_t
| Optional_one a0, Optional_one a1 ⇒ (rel01 a0 a1)
| _, _ ⇒ Boo_f
end
)
: Optional_rel_Ty.

9.8.4 OPTIONAL RELATED TO, CONNECTIVE

( Optional_rel_conn A relA o0 o1 ) is (Optional_rel A A relA o0 o1).

Definition Optional_rel_conn_Ty :=
( ∀
(A : Type)
(relA : (Boo_Pred_Bin.Conn A)),
(Boo_Pred_Bin.Conn (Optional A))
)
: Type.

Definition Optional_rel_conn :=
(fun A relA o0 o1 ⇒ (Optional_rel A A relA o0 o1) )
: Optional_rel_conn_Ty.

9.8.5 OPTIONAL NON-EMPTY

( Optional_nonEmpty A o ) is the false Boolean if o is the none optional A; and the
true Boolean otherwise.

Definition Optional_nonEmpty_Ty :=
( ∀(A : Type), (Boo_Pred (Optional A)) ) : Type.

Definition Optional_nonEmpty :=
(fun A o ⇒
match o
return Boo
with
| Optional_none ⇒ Boo_f
| Optional_one a ⇒ Boo_t
end
) : Optional_nonEmpty_Ty.

9.8.6 OPTIONAL EMPTY

(Optional_empty A o) is (Boo_not (Optional_nonEmpty A o)).

Definition Optional_empty_Ty :=
( ∀(A : Type), (Boo_Pred (Optional A)) ) : Type.

Definition Optional_empty := ( fun A o ⇒
(Boo_not (Optional_nonEmpty A o))
) : Optional_empty_Ty.

9.8.7 THE OPTIONAL ONE OPERATOR

(Optional_op_one A B opA) is the operator from A to an optional B mapping a : A onto the one optional B containing (opA a).

Definition Optional_op_one_Ty :=
( ∀
    (A : Type)
    (B : Type)
    (opA : (Op A B)),
    (Op A (Optional B))
) : Type.

Definition Optional_op_one :=
( fun A B opA a ⇒ (Optional_one B (opA a)) ) : Optional_op_one_Ty.

9.8.8 OPTIONAL SELECT

(Optional_select A B selectA o) is the empty optional B if o is the empty optional A; and (selectA a) if o is the one optional A containing a.
Definition \textit{Optional\_select\_Ty} :=
\[
\forall \\
(A : \text{Type}) \\
(B : \text{Type}) \\
(selectA : (\text{Op A (Optional B)})), \\
(\text{Op (Optional A) (Optional B)}) \\
\) : Type.
\]

Definition \textit{Optional\_select} := ( \text{fun A B selectA o} \Rightarrow \\
\begin{align*}
& \text{match o} \\
& \text{return (Optional B)} \\
& \text{with} \\
& | \text{Optional\_none} \Rightarrow (\text{Optional\_none B}) \\
& | \text{Optional\_one a} \Rightarrow (\text{selectA a}) \\
& \text{end} \\
\end{align*} \\
\) : Optional\_select\_Ty.

\textbf{9.8.9 OPTIONAL SELECT, TO ELEMENT}

\((\text{Optional\_select\_toElem A B selectA o})\) is \((\text{Optional\_select A B (Optional\_Op\_one A B selectA) o})\).

Definition \textit{Optional\_select\_toElem\_Ty} :=
\[
\forall \\
(A : \text{Type}) \\
(B : \text{Type}) \\
(selectA : (\text{Op A B})), \\
(\text{Op (Optional A) (Optional B)}) \\
\) : Type.

Definition \textit{Optional\_select\_toElem} := ( \text{fun A B selectA o} \Rightarrow \\
\begin{align*}
& (\text{Optional\_select A B (Optional\_Op\_one A B selectA) o}) \\
\end{align*} \\
\) : Optional\_select\_toElem\_Ty.
9.9 FUNDAMENTALS: BOOLEANS: AND OPTIONALS


9.9.1 DEPENDENCIES


9.9.2 BOOLEAN TO OPTIONAL

(Boo_toOptional A b a) is the one optional A containing a if b; and the none optional A otherwise.
Definition Boo_toOptional_Ty :=
(\forall
 (A : Type),
 (Op_Bin Boo A (Optional A))
) : Type.
Definition Boo_toOptional := (fun A b a =>
 if b
 return (Optional A)
 then (Optional_one A a)
 else (Optional_none A)
 ) : Boo_toOptional_Ty.

9.9.3 THE BOOLEAN OPTIONAL OPERATOR

(Bool_OpOptional A predA) is the operator from A to an optional A mapping a : A onto (Boo_toOptional A (predA a) a).
Definition Boo_OpOptional_Ty :=
\[(\forall \ A : Type) \ (\text{predA} : (\text{Boo}_A \text{Pred} A)), \ (\text{OpA} \ (\text{Optional} A)) \) : Type.\]

Definition \(\text{Boo}_\text{Op}_{\text{Optional}} := \) 
\[
(\text{fun} \ A \ \text{predA} \ a \Rightarrow (\text{Boo}_{\text{to}}_{\text{Optional}} A \ (\text{predA} a) \ a)) \)
\[: \text{Boo}_\text{Op}_{\text{Optional}_\text{Ty}}.\]

9.10  FUNDAMENTALS: CHOICES: MAIN


9.10.1 DEPENDENCIES


9.10.2 CHOICES

A choice \(F, S\) is exactly one of the following:
- for \(f : F\), the first choice \(F, S\) containing \(f\)
- for \(s : S\), the second choice \(F, S\) containing \(s\)

\(\text{Choice}\) is defined in the same way as \(\text{sum}\) in \(\text{Coq.Init.Datatypes}\), except that \(\text{Choice:Type}\) whereas \(\text{sum:Set}\).

Inductive \(\text{Choice}(F : Type)(S : Type) : Type :=\)
\[
| \text{Choice.first} : (\text{Op}\ F \ (\text{Choice} F S))\]
| Choice_second : (Op S (Choice F S)).

### 9.10.3 CHOICE RELATED TO

\((Choice\_rel\ F0\ S0\ F1\ S1\ relF01\ relS01\ c0\ c1)\) is the true Boolean if \(c0\) and \(c1\) have the same shape, and their corresponding elements \(f0 : F0, f1 : F1\) satisfy \((relF01\ f0\ f1)\) or \(s0 : S0, s1 : S1\) satisfy \((relS01\ s0\ s1)\); and the false Boolean otherwise.

**Definition** \(Choice\_rel\_Ty :=\)

\[
(\forall
\begin{align*}
(F0 : Type) \\
(S0 : Type) \\
(F1 : Type) \\
(S1 : Type) \\
(relF01 : (Boo\_Pred\_Bin F0 F1)) \\
(relS01 : (Boo\_Pred\_Bin S0 S1)), \\
(Boo\_Pred\_Bin (Choice F0 S0) (Choice F1 S1))
\end{align*}
) : Type.
\]

**Definition** \(Choice\_rel := (fun F0 S0 F1 S1 relF01 relS01 c0 c1 ⇒\)

\[
\begin{align*}
&\text{match } c0, c1 \\
&\text{return Boo} \\
&\text{with} \\
&| Choice\_first f0, Choice\_first f1 ⇒ (relF01 f0 f1) \\
&| Choice\_second s0, Choice\_second s1 ⇒ (relS01 s0 s1) \\
&| _, _ ⇒ Boo\_f
\end{align*}
\]

\) : Choice\_rel\_Ty.

### 9.10.4 CHOICE RELATED TO, CONNECTIVE

\((Choice\_rel\_conn\ F S relF relS c0 c1)\) is \((Choice\_rel\ F F S S relF relS c0 c1)\).

**Definition** \(Choice\_rel\_conn\_Ty :=\)

\[
(\forall
\begin{align*}
\end{align*}
)
\[(F : Type)\]
\[(S : Type)\]
\[(relF : (Boo_Pred_Bin_Conn F))\]
\[(relS : (Boo_Pred_Bin_Conn S)),\]
\[(Boo_Pred_Bin_Conn (Choice F S))\]

): Type.

Definition Choice_rel_conn :=
\[
(\text{fun}\ F\ S\ relF\ relS\ c0\ c1 \Rightarrow (Choice\_rel\ F\ F\ S\ relF\ relS\ c0\ c1))
\]
: Choice_rel_conn_Ty.

9.10.5 CHOICE TO OPTIONAL

\((Choice\_to\_Optional\ A\ c)\) is the one optional \(A\) containing \(a\) if \(c\) is the first choice \(A\), unit containing \(a\); and the none optional \(A\) otherwise.

Definition Choice_to_Optimal_Ty :=
\[
(\forall\ (A : Type),\ (Op\ (Choice\ A\ Uni)\ (Optional\ A)))
\]
: Type.

Definition Choice_to_Optimal := (fun A c ⇒
match c
return (Optional A)
with
| Choice_first a ⇒ (Optional_one A a)
| Choice_second elem ⇒ (Optional_none A)
end
) : Choice_to_Optimal_Ty.

9.10.6 CHOICE MERGE

\((Choice\_merge\ A\ c)\) is \(a\) where \(c\) is the first or second choice \(A\), \(A\) containing \(a\).

Definition Choice_merge_Ty :=
\[
(\forall (A : Type),\ (Op\ (Choice\ A\ A)\ A)) : Type.
\]

Definition Choice_merge := (fun A c ⇒
match c
return A
with
  | Choice_first a ⇒ a
  | Choice_second a ⇒ a
end
) : Choice_merge_Ty.

9.11  FUNDAMENTALS: PAIRS: MAIN


9.11.1  DEPENDENCIES


9.11.2  PAIRS

A pair $L, R$ named $p$ contains all of the following:

- the left of $p$, which is an $L$
- the right of $p$, which is an $R$

Record Pair($L : Type$)$($R : Type$) : Type := Pairctor {
  Pair_left : $L$;
  Pair_right : $R$
}.
9.11.3 PAIR RELATED TO

(Pair_rel L0 R0 L1 R1 relL01 relR01 p0 p1) is the true Boolean if (relL01 (Pair_left L0 R0 p0) (Pair_left L1 R1 p1)) and (relR01 (Pair_right L0 R0 p0) (Pair_right L1 R1 p1)); and the false Boolean otherwise.
Definition Pair_rel_Ty :=

(\forall

  (L0 : Type)
  (R0 : Type)
  (L1 : Type)
  (R1 : Type)
  (relL01 : (Boo_Pred_Bin L0 L1))
  (relR01 : (Boo_Pred_Bin R0 R1)),
  (Boo_Pred_Bin (Pair L0 R0) (Pair L1 R1))
) : Type.

Definition Pair_rel := (fun L0 R0 L1 R1 relL01 relR01 p0 p1 ⇒

  if (relL01 (Pair_left L0 R0 p0) (Pair_left L1 R1 p1))
  then (relR01 (Pair_right L0 R0 p0) (Pair_right L1 R1 p1))
  else Boo_f
) : Pair_rel_Ty.

9.11.4 PAIR RELATED TO, CONNECTIVE

(Pair_rel_conn L R relL relR p0 p1) is (Pair_rel L R L R relL relR p0 p1).

Definition Pair_rel_conn_Ty :=

(\forall

  (L : Type)
  (R : Type)
  (relL : (Boo_Pred_Bin_Conn L))
  (relR : (Boo_Pred_Bin_Conn R)),
  (Boo_Pred_Bin_Conn (Pair L R))
) : Type.

Definition Pair_rel_conn :=

  (fun L R relL relR p0 p1 ⇒ (Pair_rel L R L R relL relR p0 p1))
9.11.5 TRIPLES

A triple $L_0, L_1, R_1$ is a $(Pair L_0 (Pair L_1 R_1))$.
Definition $Trip_Ty := (\forall (L_0 : Type)(L_1 : Type)(R_1 : Type), Type) : Type$.
Definition $Trip := (fun L_0 L_1 R_1 \Rightarrow (Pair L_0 (Pair L_1 R_1))) : Trip_Ty$.

9.11.6 TRIPLE LEFT 0

$(Trip_{left0} L_0 L_1 R_1 t)$ is the left of $t$.
Definition $Trip_{left0_Ty} :=$

$(\forall$

$(L_0 : Type)$
$(L_1 : Type)$
$(R_1 : Type)$,
$(Op (Trip L_0 L_1 R_1) L_0)$

$) : Type$.
Definition $Trip_{left0} :=$

$(fun L_0 L_1 R_1 t \Rightarrow (Pair_{left} L_0 (Pair L_1 R_1) t)) : Trip_{left0_Ty}$.

9.11.7 TRIPLE RIGHT 0

$(Trip_{right0} L_0 L_1 R_1 t)$ is the right of $t$.
Definition $Trip_{right0_Ty} :=$

$(\forall$

$(L_0 : Type)$
$(L_1 : Type)$
$(R_1 : Type)$,
$(Op (Trip L_0 L_1 R_1) (Pair L_1 R_1))$

$) : Type$.
Definition $Trip_{right0} :=$

$(fun L_0 L_1 R_1 t \Rightarrow (Pair_{right} L_0 (Pair L_1 R_1) t)) : Trip_{right0_Ty}$.
9.11.8  TRIPLE LEFT 1

(Trip_left1 L0 L1 R1 t) is the right-left of t.

Definition Trip_left1_Type :=

(\forall

  (L0 : Type)
  (L1 : Type)
  (R1 : Type),
  (Op (Trip L0 L1 R1) L1)
)
: Type.

Definition Trip_left1 :=

  (fun L0 L1 R1 t \to (Pair_left L1 R1 (Trip_right0 L0 L1 R1 t)))

  : Trip_left1_Type.

9.11.9  TRIPLE RIGHT 1

(Trip_right1 L0 L1 R1 t) is the right-right of t.

Definition Trip_right1_Type :=

(\forall

  (L0 : Type)
  (L1 : Type)
  (R1 : Type),
  (Op (Trip L0 L1 R1) R1)
)
: Type.

Definition Trip_right1 :=

  (fun L0 L1 R1 t \to (Pair_right L1 R1 (Trip_right0 L0 L1 R1 t)))

  : Trip_right1_Type.

9.11.10 QUADRUPLES

A quadruple L0, L1, L2, R2 is a (Pair L0 (Trip L1 L2 R2)).

Definition Quad_Type :=

  (\forall
\[(L0 : \text{Type})\]
\[(L1 : \text{Type})\]
\[(L2 : \text{Type})\]
\[(R2 : \text{Type}),\]
\text{Type}
\): Type.

Definition \text{Quad} := (\text{fun } L0 \ L1 \ L2 \ R2 \Rightarrow
(Pair L0 (Trip L1 L2 R2))
) : \text{Quad\_Ty}.

### 9.11.11 QUADRPUPLE LEFT 0

\((\text{Quad\_left0}\ L0 \ L1 \ L2 \ R2 \ t)\) is the left of \(q\).

Definition \text{Quad\_left0\_Ty} :=
\((\forall
(L0 : \text{Type})
(L1 : \text{Type})
(L2 : \text{Type})
(R2 : \text{Type}),
(Op (Quad L0 L1 L2 R2) L0)
) : Type.

Definition \text{Quad\_left0} :=
\((\text{fun } L0 \ L1 \ L2 \ R2 \ t \Rightarrow (\text{Pair\_left} L0 (\text{Trip L1 L2 R2}) q) )\)
: \text{Quad\_left0\_Ty}.

### 9.11.12 QUADRPUPLE RIGHT 0

\((\text{Quad\_right0}\ L0 \ L1 \ L2 \ R2 \ t)\) is the right of \(q\).

Definition \text{Quad\_right0\_Ty} :=
\((\forall
(L0 : \text{Type})
(L1 : \text{Type})
(L2 : \text{Type})
(R2 : \text{Type}),\)
\begin{align*}
(\text{Op} \ (\text{Quad} \ L0 \ L1 \ L2 \ R2) \ (\text{Trip} \ L1 \ L2 \ R2))
\end{align*}

\) : Type.\)

Definition \texttt{Quad\_right0} :=
\begin{align*}
( \text{fun} \ L0 \ L1 \ L2 \ R2 \ q \Rightarrow (\text{Pair\_right} \ L0 \ (\text{Trip} \ L1 \ L2 \ R2) \ q) )
\end{align*}

: \texttt{Quad\_right0\_Ty}.

\large\textbf{9.11.13 QUADRUPLE LEFT 1}

\( (\text{Quad\_left1} \ L0 \ L1 \ L2 \ R2 \ t) \) is the right-left of \( q \).

Definition \texttt{Quad\_left1\_Ty} :=
\begin{align*}
( \forall \ \\

(L0 : \text{Type}) \\
(L1 : \text{Type}) \\
(L2 : \text{Type}) \\
(R2 : \text{Type}), \\
(\text{Op} \ (\text{Quad} \ L0 \ L1 \ L2 \ R2) \ L1)
\end{align*}

: Type.\)

Definition \texttt{Quad\_left1} := ( \text{fun} \ L0 \ L1 \ L2 \ R2 \ q \Rightarrow \\

(\text{Trip\_left0} \ L1 \ L2 \ R2 \ (\text{Quad\_right0} \ L0 \ L1 \ L2 \ R2 \ q)) \\

) : \texttt{Quad\_left1\_Ty}.

\large\textbf{9.11.14 QUADRUPLE LEFT 2}

\( (\text{Quad\_left2} \ L0 \ L1 \ L2 \ R2 \ t) \) is the right-right-left of \( q \).

Definition \texttt{Quad\_left2\_Ty} :=
\begin{align*}
( \forall \ \\

(L0 : \text{Type}) \\
(L1 : \text{Type}) \\
(L2 : \text{Type}) \\
(R2 : \text{Type}), \\
(\text{Op} \ (\text{Quad} \ L0 \ L1 \ L2 \ R2) \ L2)
\end{align*}

: Type.\)

Definition \texttt{Quad\_left2} := ( \text{fun} \ L0 \ L1 \ L2 \ R2 \ q \Rightarrow \\

(\text{Trip\_left1} \ L1 \ L2 \ R2 \ (\text{Quad\_right0} \ L0 \ L1 \ L2 \ R2)) \\

)
9.11.15 QUADRUPLE RIGHT 2

\((\text{Quad\_right2 } L0 \ L1 \ L2 \ R2 \ t)\) is the right-right-right of \(q\).

Definition \(\text{Quad\_right2\_Ty} :=\)
\[
(\forall \quad \\
(L0 : \text{Type}) \ \\
(L1 : \text{Type}) \ \\
(L2 : \text{Type}) \ \\
(R2 : \text{Type}), \ \\
(Op (\text{Quad } L0 \ L1 \ L2 \ R2) \ R2) \ \\
) : \text{Type}.
\]

Definition \(\text{Quad\_right2} := (\text{fun } L0 \ L1 \ L2 \ R2 \ q \Rightarrow\)
\[
(\text{Trip\_right1 } L1 \ L2 \ R2 (\text{Quad\_right0 } L0 \ L1 \ L2 \ R2 \ q)) \ \\
) : \text{Quad\_right2\_Ty}.
\]

9.12 FUNDAMENTALS: LISTS: MAIN


9.12.1 DEPENDENCIES


9.12.2 LISTS

A list \(A\) is exactly one of the following:
• the nil list \( A \)

• for some head : A and rest : (Lis A), the cons list A of head and rest

\( \text{Lis} \) is defined in the same way as \text{list} in \text{Coq.Lists.List}, except that \( \text{Lis.Type} \) whereas \( \text{list}:\text{Set} \).

\text{Inductive Lis}(A : \text{Type}) : \text{Type} :=
| Lis.nil : (Lis A)
| Lis.cons : (Op_Bin A (Lis A) (Lis A)).

\section{LIST NOTATION}

\( A, a0, \ldots, a1 \) is the list A containing the elements \( a0, \ldots, a1 \). \( a0, \ldots, a1 \) must contain at least one element.

\( A, a0, \ldots, a1 \) is defined in the same way as the list notation in [8, section 11.1.11], except that \( A, a0, \ldots, a1 \) explicitly includes A.

\text{Notation} \\
"[ A, a0, \ldots, a1 ]" :=
(Lis.cons A a0 .. (Lis.cons A a1 (Lis.nil A)) ..) : Lis.scope.

\text{Open Scope Lis_scope}.

\section{LIST RELATED TO}

\( (\text{Lis}_{rel} A0 A1 rel01 l0 l1) \) is the true Boolean if \( l0 \) and \( l1 \) have the same shape, and their corresponding elements \( a0 : A0, a1 : A1 \) satisfy \( (\text{rel01} a0 a1) \); and the false Boolean otherwise.

\text{Definition} \( \text{Lis}_{rel} \text{Ty} :=
\)
\( (\forall \)
\( (A0 : \text{Type})\)
\( (A1 : \text{Type})\)
\( (\text{rel01} : (\text{Boo_Pred_Bin} A0 A1)),\)
\( (\text{Boo_Pred_Bin} (\text{Lis} A0) (\text{Lis} A1))\)
\( ) : \text{Type} \).

\text{Definition} \( \text{Lis}_{rel} := (\text{fun} A0 A1 rel01 \Rightarrow \)
\[\text{fix } \text{fp}(l0 : \text{(Lis A0)})(l1 : \text{(Lis A1)})(\text{struct } l0) : \text{Boo} :=
\]
\[\text{match } l0, l1\]
\[\text{return Boo} \]
\[\text{with}
| \text{Lis\_nil, Lis\_nil } \Rightarrow \text{Boo\_t}
| \text{Lis\_cons } l0\text{Head} l0\text{Rest}, \text{Lis\_cons } l1\text{Head} l1\text{Rest} \Rightarrow
\]
\[\text{if } (\text{rel01 } l0\text{Head} l1\text{Head})\]
\[\text{return Boo}\]
\[\text{then } (\text{fp } l0\text{Rest} l1\text{Rest})\]
\[\text{else Boo\_f}\]
\[| \_\_, \_ \Rightarrow \text{Boo\_f}\]
\[\text{end}\]
\[) : \text{Lis\_rel\_Ty}.\]

### 9.12.5 LIST RELATED TO, CONNECTIVE

\((\text{Lis\_rel\_conn } A \text{relA } l0 l1)\) is \((\text{Lis\_rel } A A \text{relA } l0 l1)\).

**Definition** \(\text{Lis\_rel\_conn\_Ty} :=
\]
\[\text{( } \forall
\]
\[\text{(A : Type)}\]
\[\text{(relA : (Boo\_Pred\_Bin\_Conn A)),}
\]
\[\text{(Boo\_Pred\_Bin\_Conn (Lis A))}
\]
\[\text{) : Type.}\]

**Definition** \(\text{Lis\_rel\_conn} :=
\]
\[\text{( } \text{fun A relA } l0 l1 \Rightarrow (\text{Lis\_rel } A A \text{relA } l0 l1) \text{) : Lis\_rel\_conn\_Ty.}\]

### 9.12.6 LIST HEAD

\((\text{Lis\_head } A l)\) is the none optional \(A\) if \(l\) is the nil list \(A\); and the one optional \(A\) containing \(l\text{Head}\) if \(l\) is the cons list \(A\) of \(l\text{Head}\) and \(l\text{Rest}\).

**Definition** \(\text{Lis\_head\_Ty} :=
\]
\[\text{( } \forall (A : \text{Type}), \text{(Op (Lis A) (Optional A))) } \text{) : Type.}\]

**Definition** \(\text{Lis\_head} := \text{( } \text{fun A } l \Rightarrow
\]
\[\text{match } l\]
return (Optional A)
with
| Lis_nil ⇒ (Optional_none A)
| Lis_cons lHead lRest ⇒ (Optional_one A lHead)
end
) : Lis_head_Type.

9.12.7 LIST REST

(Lis_rest A l) is the none optional list A if l is the nil list A; and the one optional list A containing lRest if l is the cons list A of lHead and lRest.
Definition Lis_rest_Type :=
( ∀(A : Type), (Op (Lis A) (Optional (Lis A))) ) : Type.
Definition Lis_rest := ( fun A l ⇒
  match l
  return (Optional (Lis A))
  with
  | Lis_nil ⇒ (Optional_none (Lis A))
  | Lis_cons lHead lRest ⇒ (Optional_one (Lis A) lRest)
  end
) : Lis_rest_Type.

9.12.8 LIST NON-EMPTY

(Lis_nonEmpty A l) is the false Boolean if l is the nil list A; and the true Boolean otherwise.
Definition Lis_nonEmpty_Type :=
( ∀(A : Type), (Boo_Pred (Lis A))) : Type.
Definition Lis_nonEmpty := ( fun A l ⇒
  match l
  return Boo
  with
  | Lis_nil ⇒ Boo_f
  | Lis_cons lHead lRest ⇒ Boo_t
end
) : Lis_nonEmpty_Ty.

9.12.9 LIST EMPTY

\( \text{(Lis\_empty}\ A\ \text{l})\) is \(\text{(Boo\_not}\ \text{(Lis\_nonEmpty}\ A\ \text{l})}\).

Definition \text{Lis\_empty\_Ty} :=
( \forall (A : Type), \text{(Boo\_Pred} (\text{Lis} A)) ) : Type.

Definition \text{Lis\_empty} := ( \text{fun}\ A\ l \Rightarrow
\text{(Boo\_not}\ \text{(Lis\_nonEmpty}\ A\ \text{l})}
) : \text{Lis\_empty\_Ty}.

9.12.10 LIST CONCATENATE

\( \text{(Lis\_cat}\ A\ \text{l0}\ \text{l1})\) is the list \(A\) containing the elements in \(l0\) followed by the elements in \(l1\).

Definition \text{Lis\_cat\_Ty} :=
( \forall (A : Type), \text{(Op\_Bin\_Simp} (\text{Lis} A)) ) : Type.

Definition \text{Lis\_cat} := ( \text{fun}\ A \Rightarrow
\text{fix}\ \text{fp}(l0 : (\text{Lis} A))(l1 : (\text{Lis} A))\{\text{struct}\ l0\} : (\text{Lis} A) :=\n\text{match}\ l0\n\text{return}\ (\text{Lis} A)\n\text{with}\n\text{| Lis\_nil} \Rightarrow l1\n\text{| Lis\_cons}\ \text{l0Head}\ \text{l0Rest} \Rightarrow (\text{Lis\_cons}\ A\ \text{l0Head}\ (\text{fp}\ \text{l0Rest}\ \text{l1}))\n\text{end}\n\} : \text{Lis\_cat\_Ty}.

9.12.11 LIST APPEND

\( \text{(Lis\_append}\ A\ \text{l}\ a)\) is \(\text{(Lis\_cat} A\ \text{l}\ [A, a])}\).

Definition \text{Lis\_append\_Ty} :=
( \forall (A : Type), \text{(Op\_Bin}\ (\text{Lis} A)\ A\ (\text{Lis} A)) ) : Type.

Definition \text{Lis\_append} :=
9.12.12 THE LIST SINGLETON OPERATOR

\((\text{Lis}_\text{Op}_{\text{singleton}} A B \text{op}A)\) is the operator from \(A\) to a list \(B\) mapping \(a : A\) onto \([B, (\text{op}A a)]\).

Definition \(\text{Lis}_\text{Op}_\text{singleton}_\text{Ty} \equiv\)

\[
(\forall (A : \text{Type})
(B : \text{Type})
(\text{op}A : (\text{Op} A B)),
(\text{Op} A (\text{Lis} B))
) : \text{Type}.
\]

Definition \(\text{Lis}_\text{Op}_{\text{singleton}} \equiv\)

\[
(\text{fun} A B \text{op}A a \Rightarrow [B, (\text{op}A a)]) : \text{Lis}_\text{Op}_{\text{singleton}}_\text{Ty}.
\]

9.12.13 THE LIST SINGLETON BINARY OPERATOR

\((\text{Lis}_\text{Op}_{\text{singleton}_{\text{bin}}} A0 A1 B \text{op}A)\) is the binary operator from \(A0, A1\) to a list \(B\) mapping \(a0 : A0, a1 : A1\) onto \([B, (\text{op}A a0 a1)]\).

Definition \(\text{Lis}_\text{Op}_{\text{singleton}_{\text{bin}}}_\text{Ty} \equiv\)

\[
(\forall (A0 : \text{Type})
(A1 : \text{Type})
(B : \text{Type})
(\text{op}A : (\text{Op}_{\text{Bin}} A0 A1 B)),
(\text{Op}_{\text{Bin}} A0 A1 (\text{Lis} B))
) : \text{Type}.
\]

Definition \(\text{Lis}_\text{Op}_{\text{singleton}_{\text{bin}}} \equiv\)

\[
(\text{fun} A0 A1 B \text{op}A a0 a1 \Rightarrow [B, (\text{op}A a0 a1)]) : \text{Lis}_\text{Op}_{\text{singleton}_{\text{bin}}}_\text{Ty}.
\]

9.12.14 THE LIST PREFIX OPERATOR

\((\text{Lis}_\text{Op}_{\text{prefix}} A B \text{op}A \text{prefix})\) is the operator from \(A\) to a list \(B\) mapping \(a : A\) onto
\( (\text{Lis\_cat} \ B \ \text{prefix} (\text{opA} \ a)) \).

Definition \( \text{Lis\_Op\_prefix\_Ty} := \)

\[
(\forall \\
(A : \text{Type}) \\
(B : \text{Type}) \\
(\text{opA} : (\text{Op} \ A (\text{Lis}\ B))) \\
(\text{prefix} : (\text{Lis}\ B)), \\
(\text{Op} \ A (\text{Lis}\ B))
) : \text{Type}.
\]

Definition \( \text{Lis\_Op\_prefix} := (\text{fun} \ A \ B \ \text{opA} \ \text{prefix} \ a \Rightarrow \\
(\text{Lis\_cat} \ B \ \text{prefix} (\text{opA} \ a))
) : \text{Lis\_Op\_prefix\_Ty}.

\section{9.12.15 THE LIST SUFFIX OPERATOR}

\( (\text{Lis\_Op\_suffix} \ A \ B \ \text{opA} \ \text{suffix}) \) is the operator from \( A \) to a list \( B \) mapping \( a : A \) onto \( (\text{Lis\_cat} \ B \ (\text{opA} \ a) \ \text{suffix}) \).

Definition \( \text{Lis\_Op\_suffix\_Ty} := \)

\[
(\forall \\
(A : \text{Type}) \\
(B : \text{Type}) \\
(\text{opA} : (\text{Op} \ A (\text{Lis}\ B))) \\
(\text{suffix} : (\text{Lis}\ B)), \\
(\text{Op} \ A (\text{Lis}\ B))
) : \text{Type}.
\]

Definition \( \text{Lis\_Op\_suffix} := (\text{fun} \ A \ B \ \text{opA} \ \text{suffix} \ a \Rightarrow \\
(\text{Lis\_cat} \ B \ (\text{opA} \ a) \ \text{suffix})
) : \text{Lis\_Op\_suffix\_Ty}.

\section{9.12.16 LIST GENERATE}

\( (\text{Lis\_generate} \ A \ \text{genA} \ m) \) is the list \( A \) whose elements are obtained by concatenating the following lists \( A \): \( (\text{genA} \ \text{Nat} \ z), ..., (\text{genA} \ m) \).

Definition \( \text{Lis\_generate\_Ty} := \)
\[(\forall (A : Type)\)
\((\text{genA} : (\text{Op Nat (Lis A)}))\),
\((\text{Op Nat (Lis A)})\)\) : Type.

Definition \(\text{Lis\_generate} := \text{fun A genA} \Rightarrow\)
\(\text{fix} \text{fp}(m : \text{Nat})\{\text{struct m} : (\text{Lis A}) :=\)
\(\text{match m return (Lis A)}\)
\(\text{with}\)
\(| \text{Nat}_z \Rightarrow (\text{genA Nat}_z)\)
\(| \text{Nat}_s \text{mPre} \Rightarrow (\text{Lis\_cat A (fp mPre) (genA m)})\)
\(\text{end}\)
\) : \(\text{Lis\_generate\_Ty}\).

### 9.12.17 LIST GENERATE, TO ELEMENT

\((\text{Lis\_generate\_toElem A genA m})\) is \((\text{Lis\_generate A (Lis\_Op\_singleton Nat A genA) m})\).

Definition \(\text{Lis\_generate\_toElem\_Ty} :=\)
\(\forall (A : Type)\)
\(\text{genA} : (\text{Op Nat A}),\)
\((\text{Op Nat (Lis A)})\)
\) : Type.

Definition \(\text{Lis\_generate\_toElem} := \text{fun A genA m} \Rightarrow\)
\(\text{Lis\_generate A (Lis\_Op\_singleton Nat A genA) m}\)
\) : \(\text{Lis\_generate\_toElem\_Ty}\).

### 9.12.18 NON-EMPTY LISTS

A non-empty list \(A\) named \(l\) contains all of the following:
- the head of \(l\), which is an \(A\)
- the rest of \(l\), which is a list \(A\)
9.12.19 NON-EMPTY LIST RELATED TO

(List_ne_rel A0 A1 rel01 l0 l1) is the true Boolean if (rel01 (List_ne_head A0 l0) (List_ne_head A1 l1)) and (List_rel A0 A1 rel01 (List_ne_rest A0 l0) (List_ne_rest A1 l1)); and the false Boolean otherwise.

Definition Lis_ne_rel_Ty :=

(\forall
 (A0 : Type)
 (A1 : Type)
 (rel01 : (Boo_Pred_Bin A0 A1)),
 (Boo_Pred_Bin (List_ne A0) (List_ne A1))
 ) : Type.

Definition Lis_ne_rel := (fun A0 A1 rel01 l0 l1 ⇒
 if (rel01 (List_ne_head A0 l0) (List_ne_head A1 l1))
 return Boo
 then
 (List_rel A0 A1 rel01 (List_ne_rest A0 l0) (List_ne_rest A1 l1))
 else Boo_f
 ) : Lis_ne_rel_Ty.

9.12.20 NON-EMPTY LIST RELATED TO, CONNECTIVE

(List_ne_rel_conn A relA l0 l1) is (List_ne_rel AA relA l0 l1).

Definition Lis_ne_rel_conn_Ty :=

(\forall
 (A : Type)
 (relA : (Boo_Pred_Bin_Conn A)),

Record Lis_ne(A : Type) : Type := Lis_ne_ctor {
    Lis_ne_head : A;
    Lis_ne_rest : (Lis A)
 }.

9.12.19 NON-EMPTY LIST RELATED TO

(List_ne_rel A0 A1 rel01 l0 l1) is the true Boolean if (rel01 (List_ne_head A0 l0) (List_ne_head A1 l1)) and (List_rel A0 A1 rel01 (List_ne_rest A0 l0) (List_ne_rest A1 l1)); and the false Boolean otherwise.

Definition Lis_ne_rel_Ty :=

(\forall
 (A0 : Type)
 (A1 : Type)
 (rel01 : (Boo_Pred_Bin A0 A1)),
 (Boo_Pred_Bin (List_ne A0) (List_ne A1))
 ) : Type.

Definition Lis_ne_rel := (fun A0 A1 rel01 l0 l1 ⇒
 if (rel01 (List_ne_head A0 l0) (List_ne_head A1 l1))
 return Boo
 then
 (List_rel A0 A1 rel01 (List_ne_rest A0 l0) (List_ne_rest A1 l1))
 else Boo_f
 ) : Lis_ne_rel_Ty.

9.12.20 NON-EMPTY LIST RELATED TO, CONNECTIVE

(List_ne_rel_conn A relA l0 l1) is (List_ne_rel AA relA l0 l1).

Definition Lis_ne_rel_conn_Ty :=

(\forall
 (A : Type)
 (relA : (Boo_Pred_Bin_Conn A)),

Record Lis_ne(A : Type) : Type := Lis_ne_ctor {
    Lis_ne_head : A;
    Lis_ne_rest : (Lis A)
 }.
\[(Boo\_Pred\_Bin\_Conn\ (Lis\_Ne\ A))\] 
\): Type.

Definition \(\text{Lis}\_\text{Ne}=_\text{rel}\_\text{conn} := \)
\(\ (\text{fun}\ A\ \text{rel}\ A\ l0\ l1 \Rightarrow (\text{Lis}\_\text{Ne}=_\text{rel}\ A\ A\ \text{rel}\ A\ l0\ l1)) \)
\): \(\text{Lis}\_\text{Ne}=_\text{rel}\_\text{conn}\_\text{Ty}\).

9.12.21 NON-EMPTY LIST SINGLETON

\((\text{List}\_\text{Ne}\_\text{singleton} A\ a)\) is the non-empty list \(A\) containing just \(a\).

Definition \(\text{Lis}\_\text{Ne}=_\text{singleton}\_\text{Ty} := \)
\(\ (\forall (A : \text{Type}), (\text{Op} A (\text{Lis}\_\text{Ne} A))) \) : Type.

Definition \(\text{Lis}\_\text{Ne}=_\text{singleton} := (\text{fun} A\ a \Rightarrow \)
\(\ (\text{Lis}\_\text{Ne}\_\text{ctor} A\ a\ (\text{Lis}\_\text{nil} A)) \)
\): \(\text{Lis}\_\text{Ne}=_\text{singleton}\_\text{Ty}\).

9.12.22 NON-EMPTY LIST TO LIST

\((\text{List}\_\text{Ne}\_\text{to}\_\text{Lis} A\ l)\) is the list \(A\) containing the same elements as \(l\).

Definition \(\text{Lis}\_\text{Ne}=_\text{to}\_\text{Lis}\_\text{Ty} := \)
\(\ (\forall (A : \text{Type}), (\text{Op} (\text{Lis}\_\text{Ne} A) (\text{Lis} A))) \) : Type.

Definition \(\text{Lis}\_\text{Ne}=_\text{to}\_\text{Lis} := (\text{fun} A\ l \Rightarrow \)
\(\ (\text{Lis}\_\text{cons} A (\text{Lis}\_\text{Ne}\_\text{head} A\ l) (\text{Lis}\_\text{Ne}\_\text{rest} A\ l)) \)
\): \(\text{Lis}\_\text{Ne}=_\text{to}\_\text{Lis}\_\text{Ty}\).

9.12.23 THE NON-EMPTY LIST HEAD OPERATOR

\((\text{Lis}\_\text{Ne}\_\text{Op}\_\text{head} A\ B\ opA)\) is the operator from a non-empty list \(A\) to \(B\) mapping \(l: (\text{Lis}\_\text{Ne} A)\) onto \((\text{opA} (\text{Lis}\_\text{Ne}\_\text{head} A\ l))\).

Definition \(\text{Lis}\_\text{Ne}\_\text{Op}\_\text{head}\_\text{Ty} := \)
\(\ (\forall \)
\(\ (A : \text{Type}) \)
\(\ (B : \text{Type}) \)
\(\ (\text{opA} : (\text{Op} A\ B)), \)
\[(\text{Op} (\text{Lis}_\text{Ne} A) B)\]
\(\text{: Type.}\)

Definition \(\text{Lis}_\text{Ne}_\text{Op}_\text{head} :=\)
\[(\text{fun} A B \text{opA} l \Rightarrow (\text{opA} (\text{Lis}_\text{Ne}_\text{head} A l)) ) : \text{Lis}_\text{Ne}_\text{Op}_\text{head}_\text{Ty}.\]

### 9.12.24 LIST TO NON-EMPTY LIST

\((\text{Lis}_\text{to}_\text{Lis}_\text{Ne} A l)\) is the none optional non-empty list \(A\) if \(l\) is the nil list \(A\); and the one optional non-empty list \(A\) containing the non-empty list \(A\) with the same elements as \(l\) otherwise.

Definition \(\text{Lis}_\text{to}_\text{Lis}_\text{Ne}_\text{Ty} :=\)
\[(\forall (A : \text{Type}), (\text{Op} (\text{Lis} A) (\text{Optional} (\text{Lis}_\text{Ne} A)))) : \text{Type}.\]

Definition \(\text{Lis}_\text{to}_\text{Lis}_\text{Ne} := (\text{fun} A l \Rightarrow\)
\(\text{match} l\)
\(\text{return} (\text{Optional} (\text{Lis}_\text{Ne} A))\)
\(\text{with}\)
\(\mid \text{Lis}_\text{nil} \Rightarrow (\text{Optional}_\text{none} (\text{Lis}_\text{Ne} A))\)
\(\mid \text{Lis}_\text{cons} l\text{Head} l\text{Rest} \Rightarrow\)
\(\text{(Optional}_\text{one} (\text{Lis}_\text{Ne} A) (\text{Lis}_\text{Ne}_\text{ctor} A l\text{Head} l\text{Rest}))\)
\(\text{end}\)
\(\) : \(\text{Lis}_\text{to}_\text{Lis}_\text{Ne}_\text{Ty}.)\]

### 9.12.25 2 PLUS LISTS

A 2 plus list \(A\) named \(l\) contains all of the following:

- the head of \(l\), which is an \(A\)
- the rest of \(l\), which is a non-empty list \(A\)

Record \(\text{Lis}_\text{P2}(A : \text{Type}) : \text{Type} := \text{Lis}_\text{P2}_\text{ctor} \{\)
\(\text{Lis}_\text{P2}_\text{head} : A;\)
\(\text{Lis}_\text{P2}_\text{rest} : (\text{Lis}_\text{Ne} A)\)
\(\}\).
9.13  FUNDAMENTALS: OPTIONALs: AND LISTS


9.13.1  DEPENDENCIES


9.13.2  OPTIONAL TO LIST

(Optional_to_Lis A o) is the nil list A if o is the none optional A; and [A, a] if o is the one optional A containing a.

Definition Optional_to_Lis Ty :=
    (∀(A : Type), (Op (Optional A) (Lis A))) : Type.

Definition Optional_to_Lis := ( fun A o ⇒
    match o
    return (Lis A)
    with
    | Optional_none ⇒ (Lis_nil A)
    | Optional_one a ⇒ [A, a]
    end ) : Optional_to_Lis Ty.

9.14  FUNDAMENTALS: BOOLEANs: AND LISTS

Poohbist.NummSquared.FundamentalsBOOLEANSAndLists

9.14.1 DEPENDENCIES


9.14.2 BOOLEAN TO LIST

\((\text{Boo\_to\_Lis} \ A \ b \ a)\) is \((\text{Optional\_to\_Lis} \ A \ (\text{Boo\_to\_Optional} \ A \ b \ a))\).

Definition \(\text{Boo\_to\_Lis\_Ty} :=\)

\[
\forall
\]

\((A : Type),
\]

\((\text{Op\_Bin Boo} \ A \ (\text{Lis A}))
\]

) : Type.

Definition \(\text{Boo\_to\_Lis} := (\text{fun} \ A \ b \ a \Rightarrow
\]

\((\text{Optional\_to\_Lis} \ A \ (\text{Boo\_to\_Optional} \ A \ b \ a))
\]

) : Boo\_to\_Lis\_Ty.

9.14.3 THE BOOLEAN LIST OPERATOR

\((\text{Boo\_Op\_Lis} \ A \ predA)\) is the operator from \(A\) to an list \(A\) mapping \(a : A\) onto
\((\text{Boo\_to\_Lis} \ A \ (predA \ a) \ a)\).

Definition \(\text{Boo\_Op\_Lis\_Ty} :=\)

\[
\forall
\]

\((A : Type)
\]

\((predA : (\text{Boo\_Pred} \ A)),
\]

\((\text{Op A} \ (\text{Lis A}))
\]

) : Type.

Definition \(\text{Boo\_Op\_Lis} :=
\]

\((\text{fun} \ A \ predA \ a \Rightarrow (\text{Boo\_to\_Lis} \ A \ (predA \ a) \ a)) : \text{Boo\_Op\_Lis\_Ty.}\)
9.15 FUNDAMENTALS: NATURALS: AND LISTS


9.15.1 DEPENDENCIES


9.15.2 NATURAL NUMBER LISTS

A natural number list is a list of natural numbers.
Definition Nat.Lis := (Lis Nat) : Type.

9.15.3 NATURAL NUMBER LIST EQUALS

(Nat.Lis.eq l0 l1) is the true Boolean if l0 and l1 are structurally equal; and the false Boolean otherwise.
Definition Nat.Lis_eq := (fun l0 l1 ⇒
    (Lis_rel_conn Nat Nat_eq l0 l1)
) : (Boo_Pred_Bin_Conn Nat_Lis).

9.16 FUNDAMENTALS: NATURALS: EFFICIENT: AND LISTS


9.16.1 DEPENDENCIES


9.16.2 EFFICIENT NATURAL NUMBER LISTS

An efficient natural number list is a list of efficient natural numbers.
Definition Nat.Eff.Lis := (Lis Nat.Eff) : Type.

9.16.3 EFFICIENT NATURAL NUMBER LIST EQUALS

(Nat.Eff.Lis.eq l0 l1) is the true Boolean if l0 and l1 are structurally equal (except using Nat.Eff.eq); and the false Boolean otherwise.
Definition Nat.Eff.Lis.eq := (fun l0 l1 ⇒
   (Lis.rel_conn Nat.Eff Nat.Eff.eq l0 l1)

9.17 FUNDAMENTALS: PAIRS: AND LISTS


9.17.1 DEPENDENCIES


9.17.2 PAIR OF HEAD AND REST TO NON-EMPTY LIST

(Pair.headRest.to.Lis.Ne A p) is (Lis.Nector A (Pair.left A (Lis A) p) (Pair.right A
\[(\text{Lis } A) \ p\). \]

Definition \(\text{Pair\_headRest\_to\_Lis\_Ne\_Ty} := \)
\[
( \forall (A : \text{Type}), (\text{Op} (\text{Pair } A (\text{Lis } A)) (\text{Lis\_Ne } A)) ) \]
\[
: \text{Type}. \]

Definition \(\text{Pair\_headRest\_to\_Lis\_Ne} := ( \text{fun } A p \Rightarrow \)
\[
(\text{Lis\_Ne\_ctor } A (\text{Pair\_left } A (\text{Lis } A) p) (\text{Pair\_right } A (\text{Lis } A) p)) \]
\[
) : \text{Pair\_headRest\_to\_Lis\_Ne\_Ty}. \]

9.18 FUNDAMENTALS: LISTS: SELECT

\(\text{Poohbist.NummSquared.Fundamentals.Lists.Select}\)


9.18.1 DEPENDENCIES


9.18.2 LIST SELECT

\((\text{Lis\_select } A B \text{ selectSuffix } l)\) is the list \(B\) obtained by first applying \text{selectSuffix} to each non-empty suffix of \(l\) (starting with \(l\) itself, if \(l\) is non-empty) made into a non-empty list \(A\); and then concatenating the resulting lists \(B\). If \(l\) is the nil list \(A\), then \((\text{Lis\_select } A B \text{ selectSuffix } l)\) is the nil list \(B\).

\(\text{Lis\_select}\) is somewhat similar in concept to the LISP mapcon function (see [27, chapter 12]).

Definition \(\text{Lis\_select\_Ty} := \)
\[
( \forall \)
(A : Type)  
(B : Type)  
(selectSuffix : (Op (Lis_Ne A) (Lis B))),  
(Op (Lis A) (Lis B))  
) : Type.

Definition Lis_select := (fun A B selectSuffix ⇒  
fix fp(l : (Lis A)) [struct l] : (Lis B) :=  
match l  
return (Lis B)  
with  
| Lis_nil ⇒ (Lis_nil B)  
| Lis_cons lHead lRest ⇒  
| Lis_cat (Lis_cons (Lis_Ne_ctor A lHead lRest)) (fp lRest)  
end  
) : Lis_select_Ty.

9.18.3 LIST SELECT, SIMPLE

(Lis_select_simp A selectSuffix l) is (Lis_select A A selectSuffix l).
Definition Lis_select_simp_Ty :=  
(∀  
(A : Type)  
(selectSuffix : (Op (Lis_Ne A) (Lis A))),  
(Op_Simp (Lis A))  
) : Type.

Definition Lis_select_simp := (fun A selectSuffix l ⇒  
(Lis_select A A selectSuffix l)  
) : Lis_select_simp_Ty.
9.18.4 LIST SELECT, ITERATE

\( \text{(Lis\_select\_iter } A \text{ selectSuffix } l \ m) \) is \( (\text{Nat\_iter } (\text{Lis } A)) \ (\text{Lis\_select\_simp } A \text{ selectSuffix } l \ m) \).

Definition \( \text{Lis\_select\_iter\_Ty} := \)

\[
( \forall \quad \\
(A : \text{Type}) \\
(\text{selectSuffix} : (\text{Op} \ (\text{Lis\_Ne } A) \ (\text{Lis } A))), \\
(\text{Op\_Bin } (\text{Lis } A) \text{ Nat } (\text{Lis } A)) \\
) : \text{Type}. 
\]

Definition \( \text{Lis\_select\_iter} := (\text{fun } A \text{ selectSuffix } l \ m \Rightarrow \quad \\
(\text{Nat\_iter } (\text{Lis } A)) \ (\text{Lis\_select\_simp } A \text{ selectSuffix } l \ m) \\
) : \text{Lis\_select\_iter\_Ty}. \)

9.18.5 LIST SELECT, TO ELEMENT

\( \text{(Lis\_select\_toElem } A \ B \text{ selectSuffix } l) \) is \( (\text{Lis\_select } A \ B) \ (\text{Lis\_Op\_singleton} \ (\text{Lis\_Ne } A) \ B \text{ selectSuffix } l) \).

\( \text{Lis\_select\_toElem} \) is somewhat similar in concept to the LISP maplist function (see [27, chapter 12]).

Definition \( \text{Lis\_select\_toElem\_Ty} := \)

\[
( \forall \quad \\
(A : \text{Type}) \\
(B : \text{Type}) \\
(\text{selectSuffix} : (\text{Op} \ (\text{Lis\_Ne } A) \ B)), \\
(\text{Op} \ (\text{Lis } A) \ (\text{Lis } B)) \\
) : \text{Type}. 
\]

Definition \( \text{Lis\_select\_toElem} := (\text{fun } A \ B \text{ selectSuffix } l \Rightarrow \quad \\
(\text{Lis\_select } A \ B) \ (\text{Lis\_Op\_singleton} \ (\text{Lis\_Ne } A) \ B \text{ selectSuffix } l) \\
) : \text{Lis\_select\_toElem\_Ty}. \)

9.18.6 LIST SELECT, TO ELEMENT, SIMPLE

\( \text{(Lis\_select\_toElem\_simp } A \text{ selectSuffix } l) \) is \( (\text{Lis\_select\_toElem } A \ A \text{ selectSuffix } l) \).
Definition \( \text{Lis\_select\_toElem\_simp\_Ty :=} \)
\[
(\forall (A : \text{Type}) \text{selectSuffix : (Op (Lis\_Ne A) A),}
\text{(Op\_Simp (Lis A)) : \text{Type}}).
\]

Definition \( \text{Lis\_select\_toElem\_simp :=} \)
\[
(\text{fun A selectSuffix l =}
\text{(Lis\_select\_toElem A A selectSuffix l)}
\text{)} : \text{Lis\_select\_toElem\_simp\_Ty}.
\]

### 9.18.7 LIST SELECT, TO ELEMENT, ITERATE

\( (\text{Lis\_select\_toElem\_iter A selectSuffix l m}) \) is \( \text{(Nat\_iter (Lis A) (Lis\_select\_toElem\_simp A selectSuffix) l m)} \).

Definition \( \text{Lis\_select\_toElem\_iter\_Ty :=} \)
\[
(\forall (A : \text{Type}) \text{selectSuffix : (Op (Lis\_Ne A) A),}
\text{(Op\_Bin (Lis A) Nat (Lis A)) : \text{Type}}).
\]

Definition \( \text{Lis\_select\_toElem\_iter :=} \)
\[
(\text{fun A selectSuffix l m =}
\text{(Nat\_iter (Lis A) (Lis\_select\_toElem\_simp A selectSuffix) l m)}
\text{)} : \text{Lis\_select\_toElem\_iter\_Ty}.
\]

### 9.18.8 LIST SELECT, BY ELEMENT

\( (\text{Lis\_select\_byElem A B selectA l}) \) is \( \text{(Lis\_select A B (Lis\_Ne\_Op\_head A (Lis B) selectA) l)} \).

\( (\text{Lis\_select\_byElem A B selectA l}) \) is the list \( B \) obtained by first applying \( \text{selectA} \) to each element in \( l \) (in the order in which the elements appear in \( l \)); and then concatenating the resulting lists \( B \). If \( l \) is the nil list \( A \), then \( (\text{Lis\_select\_byElem A B selectA l}) \) is the nil list \( B \).

\( \text{Lis\_select\_byElem} \) is somewhat similar in concept to the LISP mapcan function (see [27, chapter 12]).
Definition $\text{Lis\_select\_byElem\_Ty} :=$

\[
(\forall
\begin{align*}
(A : \text{Type}) \\
(B : \text{Type}) \\
(\text{selectA} : (\text{Op} A (\text{Lis} B)), \\
(\text{Op} (\text{Lis} A) (\text{Lis} B))
\end{align*}
) : \text{Type}.
\]

Definition $\text{Lis\_select\_byElem} := (\text{fun} A B \text{selectA} l \Rightarrow
\begin{align*}
(\text{Lis\_select} A B (\text{Lis\_Ne\_Op\_head} A (\text{Lis} B) \text{selectA} l)
\end{align*}
) : \text{Lis\_select\_byElem\_Ty}.$

9.18.9 LIST SELECT, BY ELEMENT, SIMPLE

$(\text{Lis\_select\_byElem\_simp} A \text{selectA} l)$ is $(\text{Lis\_select\_byElem} A A \text{selectA} l)$.

Definition $\text{Lis\_select\_byElem\_simp\_Ty} :=$

\[
(\forall
\begin{align*}
(A : \text{Type}) \\
(\text{selectA} : (\text{Op} A (\text{Lis} A)), \\
(\text{Op\_Simp} (\text{Lis} A))
\end{align*}
) : \text{Type}.
\]

Definition $\text{Lis\_select\_byElem\_simp} := (\text{fun} A \text{selectA} l \Rightarrow
\begin{align*}
(\text{Lis\_select\_byElem} A A \text{selectA} l)
\end{align*}
) : \text{Lis\_select\_byElem\_simp\_Ty}.$

9.18.10 LIST SELECT, BY ELEMENT, ITERATE

$(\text{Lis\_select\_byElem\_iter} A \text{selectA} l m)$ is $(\text{Nat\_iter} (\text{Lis} A) (\text{Lis\_select\_byElem\_simp} A \text{selectA} l m))$.

Definition $\text{Lis\_select\_byElem\_iter\_Ty} :=$

\[
(\forall
\begin{align*}
(A : \text{Type}) \\
(\text{selectA} : (\text{Op} A (\text{Lis} A)), \\
(\text{Op\_Bin} (\text{Lis} A) \text{Nat} (\text{Lis} A))
\end{align*}
) : \text{Type}.
\]
Definition \texttt{Lis\_select\_byElem\_iter} \(:= \) \texttt{( fun A \ selectA l m \Rightarrow (Nat\_iter \ (Lis A) \ (Lis\_select\_byElem\_simp A \ selectA) l m) ) : Lis\_select\_byElem\_iter\_Ty.}

9.18.11 LIST SELECT, BY ELEMENT, INTRODUCED

\((Lis\_select\_byElem\_intro A B selectA intro l)\) is \((Lis\_select\_byElem A B (Lis\_Op\_prefix A B selectA intro) l)\).

Definition \texttt{Lis\_select\_byElem\_intro\_Ty} \(:=\)

\(( \forall \\
(A : Type) \\
(B : Type) \\
(selectA : (Op A (Lis B))) \\
(intro : (Lis B)), \\
(Op (Lis A) (Lis B)) \\
) : Type.\)

Definition \texttt{Lis\_select\_byElem\_intro} \(:= \) \texttt{( fun A B selectA intro l \Rightarrow (Lis\_select\_byElem A B (Lis\_Op\_prefix A B selectA intro) l) ) : Lis\_select\_byElem\_intro\_Ty.}

9.18.12 LIST SELECT, BY ELEMENT, TERMINATED

\((Lis\_select\_byElem\_ter A B selectA ter l)\) is \((Lis\_select\_byElem A B (Lis\_Op\_suffix A B selectA ter) l)\).

Definition \texttt{Lis\_select\_byElem\_ter\_Ty} \(:=\)

\(( \forall \\
(A : Type) \\
(B : Type) \\
(selectA : (Op A (Lis B))) \\
(ter : (Lis B)), \\
(Op (Lis A) (Lis B)) \\
) : Type.\)

Definition \texttt{Lis\_select\_byElem\_ter} \(:= \) \texttt{( fun A B selectA ter l \Rightarrow (Lis\_select\_byElem A B (Lis\_Op\_suffix A B selectA ter) l) )}
9.18.13 LIST SELECT, BY ELEMENT, SEPARATED

\((\text{Lis\_select\_byElem\_sep } A \ B \ \text{selectA } \ \text{sep } l)\) is the nil list \(B\) if \(l\) is the nil list \(A\); and the list \(B\) obtained by concatenating \((\text{selectA } l\text{Head})\) and \((\text{Lis\_select\_byElem\_intro } A \ B \ \text{selectA } \ \text{sep } l\text{Rest})\) if \(l\) is the cons list \(A\) of \(l\text{Head}\) and \(l\text{Rest}\).

Definition \(\text{Lis\_select\_byElem\_sep}\_Ty :=\)

\[ (\forall (A : \text{Type}) (B : \text{Type}) (\text{selectA} : (\text{Op} A (\text{Lis} B))) (\text{sep} : (\text{Lis} B)), (\text{Op} (\text{Lis} A) (\text{Lis} B)) : \text{Type}. \]

Definition \(\text{Lis\_select\_byElem\_sep} := (\text{fun} A B \text{selectA} \text{sep} l \Rightarrow \text{match} l \text{return} (\text{Lis} B) \text{with} \mid \text{Lis\_nil} \Rightarrow (\text{Lis\_nil} B) \mid \text{Lis\_cons} l\text{Head} l\text{Rest} \Rightarrow (\text{Lis\_cat} B (\text{selectA} l\text{Head}) (\text{Lis\_select\_byElem\_intro} A B \text{selectA} \text{sep} l\text{Rest}))) \) ) : \(\text{Lis\_select\_byElem\_sep}\_Ty.\)

9.18.14 LIST SELECT, BY ELEMENT, TO ELEMENT

\((\text{Lis\_select\_byElem\_toElem} A B \ \text{selectA} l)\) is \((\text{Lis\_select\_byElem} A B (\text{Lis\_Op\_singleton} A B \ \text{selectA}) l)\).

\(\text{Lis\_select\_byElem\_toElem}\) is somewhat similar in concept to the LISP mapcar func-
tion (see [27, chapter 12]).

Definition \( \text{Lis\_select\_byElem\_toElem\_Ty} := \)
\[
( \forall \\
(A : \text{Type}) \\
(B : \text{Type}) \\
(\text{selectA} : (\text{Op} \ A \ B)), \\
(\text{Op} (\text{Lis} \ A) (\text{Lis} \ B)) \\
) : \text{Type}.
\]

Definition \( \text{Lis\_select\_byElem\_toElem} := (\text{fun} \ A \ B \ \text{selectA} \ l \Rightarrow \\
(\text{Lis\_select\_byElem} \ A \ B (\text{Lis\_Op\_singleton} \ A \ B \ \text{selectA}) \ l) \\
) : \text{Lis\_select\_byElem\_toElem\_Ty}.
\]

### 9.18.15 LIST SELECT, BY ELEMENT, TO ELEMENT, SIMPLE

\( (\text{Lis\_select\_byElem\_toElem\_simp} \ A \ \text{selectA} \ l) \) is \( (\text{Lis\_select\_byElem\_toElem} \ A \ A \ \text{selectA} \ l) \).

Definition \( \text{Lis\_select\_byElem\_toElem\_simp\_Ty} := \\
( \forall (A : \text{Type})(\text{selectA} : (\text{Op\_Simp} \ A)), (\text{Op\_Simp} (\text{Lis} \ A)) \\
) : \text{Type}.
\]

Definition \( \text{Lis\_select\_byElem\_toElem\_simp} := (\text{fun} \ \text{selectA} \ l \Rightarrow \\
(\text{Lis\_select\_byElem\_toElem\_simp} \ A \ A \ \text{selectA} \ l) \\
) : \text{Lis\_select\_byElem\_toElem\_simp\_Ty}.
\]

### 9.18.16 LIST SELECT, BY ELEMENT, TO ELEMENT, ITERATE

\( (\text{Lis\_select\_byElem\_toElem\_iter} \ A \ \text{selectA} \ l \ m) \) is \( (\text{Nat\_iter} (\text{Lis} \ A) \ (\text{Lis\_select\_byElem\_toElem\_simp} \ A \ \text{selectA} \ l \ m)) \).

Definition \( \text{Lis\_select\_byElem\_toElem\_iter\_Ty} := \\
( \forall \\
(A : \text{Type}) \\
(\text{selectA} : (\text{Op\_Simp} \ A)), \\
(\text{Op\_Bin} (\text{Lis} \ A) \ \text{Nat} (\text{Lis} \ A)) \\
) : \text{Type}.
\]

Definition \( \text{Lis\_select\_byElem\_toElem\_iter} := (\text{fun} \ \text{selectA} \ l \ m \Rightarrow \\
)
(Nat_iter (Lis A) (Lis_select_byElem_toElem_simp A selectA) l m) : Lis_select_byElem_toElem_iter_Ty.

9.18.17 LIST SELECT, BY PREFIX, RECURSIVE

(Lis_select_byPrefix_recur A B selectPrefix l earlier) is the list B obtained by first applying selectPrefix to each non-empty prefix of l (ending with l itself, if l is non-empty), prefixed with earlier, and with the tail separated; and then concatenating the resulting lists B. If l is the nil list A, then (Lis_select_byPrefix_recur A B selectPrefix l earlier) is the nil list B.

Definition Lis_select_byPrefix_recur_Ty :=

(∀

(A : Type)
(B : Type)
(selectPrefix : (Op_Bin (Lis A) A (Lis B))),
(Op_Bin_Conn (Lis A) (Lis B))

) : Type.

Definition Lis_select_byPrefix_recur := ( fun A B selectPrefix ⇒
fix fp(l : (Lis A))(earlier : (Lis A))(struct l) : (Lis B) :=
match l
return (Lis B)
with
| Lis_nil ⇒ (Lis_nil B)
| Lis_cons lHead lRest ⇒
  (Lis_cat
    B
    (selectPrefix earlier lHead)
    (fp lRest (Lis_append A earlier lHead))
  )
end
) : Lis_select_byPrefix_recur_Ty.
9.18.18 LIST SELECT, BY PREFIX

\((\text{Lis\_select\_byPrefix} A B \text{selectPrefix} l)\) is the list \(B\) obtained by first applying \(\text{selectPrefix}\) to each non-empty prefix of \(l\) (ending with \(l\) itself, if \(l\) is non-empty) with the tail separated; and then concatenating the resulting lists \(B\). If \(l\) is the nil list \(A\), then \((\text{Lis\_select\_byPrefix} A B \text{selectPrefix} l)\) is the nil list \(B\).

Definition \(\text{Lis\_select\_byPrefix\_Ty} :=\)

\[
\forall \\
(A : \text{Type})\\
(B : \text{Type})\\
(\text{selectPrefix} : (\text{Op\_Bin} (\text{Lis} A) A (\text{Lis} B))),\\
(\text{Op} (\text{Lis} A) (\text{Lis} B))
\) : Type.

Definition \(\text{Lis\_select\_byPrefix} := (\text{fun} A B \text{selectPrefix} l \Rightarrow \\
(\text{Lis\_select\_byPrefix\_recur} A B \text{selectPrefix} l (\text{Lis\_nil} A))
\) : \(\text{Lis\_select\_byPrefix\_Ty}\).

9.18.19 LIST SELECT, BY PREFIX, SIMPLE

\((\text{Lis\_select\_byPrefix\_simp} A \text{selectPrefix} l)\) is \((\text{Lis\_select\_byPrefix} A A \text{selectPrefix} l)\).

Definition \(\text{Lis\_select\_byPrefix\_simp\_Ty} :=\)

\[
\forall \\
(A : \text{Type})\\
(\text{selectPrefix} : (\text{Op\_Bin} (\text{Lis} A) A (\text{Lis} A))),\\
(\text{Op\_Simp} (\text{Lis} A))
\) : Type.

Definition \(\text{Lis\_select\_byPrefix\_simp} := (\text{fun} \text{selectPrefix} l \Rightarrow \\
(\text{Lis\_select\_byPrefix\_simp} A \text{selectPrefix} l)
\) : \(\text{Lis\_select\_byPrefix\_simp\_Ty}\).

9.18.20 LIST SELECT, BY PREFIX, ITERATE

\((\text{Lis\_select\_byPrefix\_iter} A \text{selectPrefix} l m)\) is \((\text{Nat\_iter} (\text{Lis} A) (\text{Lis\_select\_byPrefix\_simp} A \text{selectPrefix} l m))\).

Definition \(\text{Lis\_select\_byPrefix\_iter\_Ty} :=\)
\((\forall (A : Type)\)
\((\text{selectPrefix} : (\text{Op}_\text{Bin} (\text{Lis} A) A (\text{Lis} A))),\)
\((\text{Op}_\text{Bin} (\text{Lis} A) \text{Nat} (\text{Lis} A))\)
\) : Type.

Definition \(\text{Lis\_select\_byPrefix\_iter} := (\text{fun} A \text{selectPrefix} l m \Rightarrow\)
\((\text{Nat\_iter} (\text{Lis} A) (\text{Lis\_select\_byPrefix\_simp} A \text{selectPrefix}) l m)\)
\) : \(\text{Lis\_select\_byPrefix\_iter\_Ty}\).

### 9.18.21 LIST SELECT, BY PREFIX, TO ELEMENT

\((\text{Lis\_select\_byPrefix\_toElem} A B \text{selectPrefix} l)\) is \((\text{Lis\_select\_byPrefix} A B\)
\((\text{Lis\_Op\_singleton\_bin} (\text{Lis} A) A B \text{selectPrefix}) l)\).

Definition \(\text{Lis\_select\_byPrefix\_toElem\_Ty} :=
\((\forall (A : Type)\)
\((B : Type)\)
\((\text{selectPrefix} : (\text{Op}_\text{Bin} (\text{Lis} A) A B)),\)
\((\text{Op} (\text{Lis} A) (\text{Lis} B))\)
\) : Type.

Definition \(\text{Lis\_select\_byPrefix\_toElem} := (\text{fun} A B \text{selectPrefix} l \Rightarrow\)
\((\text{Lis\_select\_byPrefix} A B\)
\((\text{Lis\_Op\_singleton\_bin} (\text{Lis} A) A B \text{selectPrefix})\)
\(l\)
\)
\) : \(\text{Lis\_select\_byPrefix\_toElem\_Ty}\).

### 9.18.22 LIST SELECT, BY PREFIX, TO ELEMENT, SIMPLE

\((\text{Lis\_select\_byPrefix\_toElem\_simp} A \text{selectPrefix} l)\) is \((\text{Lis\_select\_byPrefix\_toElem} A A\)
\((\text{Lis\_select\_byPrefix\_toElem\_simp} A \text{selectPrefix} l)\).

Definition \(\text{Lis\_select\_byPrefix\_toElem\_simp\_Ty} :=\)
( \forall 
  (A : Type) 
  (selectPrefix : (Op_Bin (Lis A) A A)), 
  (Op_Simp (Lis A)) 
) : Type.

Definition Lis_select_byPrefix_toElem_simp := ( fun A selectPrefix l ⇒ 
  (Lis_select_byPrefix_toElem A A selectPrefix l) 
) : Lis_select_byPrefix_toElem_simp_Ty.

9.18.23 LIST SELECT, BY PREFIX, TO ELEMENT, ITERATE

(Lis_select_byPrefix_toElem_iter A selectPrefix l m) is (Nat_iter (Lis A) 
(Lis_select_byPrefix_toElem_simp A selectPrefix) l m).
Definition Lis_select_byPrefix_toElem_iter_Ty := 
( \forall 
  (A : Type) 
  (selectPrefix : (Op_Bin (Lis A) A A)), 
  (Op_Bin (Lis A) Nat (Lis A)) 
) : Type.

Definition Lis_select_byPrefix_toElem_iter := ( fun A selectPrefix l m ⇒ 
  (Nat_iter 
   (Lis A) 
   (Lis_select_byPrefix_toElem_simp A selectPrefix) 
   l 
   m 
  ) 
) : Lis_select_byPrefix_toElem_iter_Ty.

9.18.24 LIST SEARCH

(Lis_search A matA l) is (Lis_select_byElem_simp A (Boo_Op_Lis A matA) l). 
(Lis_search A matA l) is l, less those a : A that do not satisfy (matA a).
Definition Lis_search_Ty := 
( \forall
(A : Type)
(matA : (Boo_Pred A)),
(Op_Simp (Lis A))
) : Type.

Definition Lis_search := ( fun A matA l ⇒
    (Lis_select_byElem_simp A (Boo_Op_Lis A matA) l)
  ) : Lis_search_Ty.

9.18.25 LIST SEARCH, FIRST

(Lis_search_first A matA l) is (Lis_head A (Lis_search A matA l)).

Definition Lis_search_first_Ty :=
  ( ∀
    (A : Type)
    (matA : (Boo_Pred A)),
    (Op (Lis A) (Optional A))
  ) : Type.

Definition Lis_search_first := ( fun A matA l ⇒
    (Lis_head A (Lis_search A matA l))
  ) : Lis_search_first_Ty.

9.18.26 LIST SEARCH, IS FOUND

(Lis_search_isFound A matA l) is (Lis_nonEmpty A (Lis_search A matA l)).

(Lis_search_isFound A matA l) is the true Boolean if there is some a : A in l such that
(matA a); and the false Boolean otherwise.

Definition Lis_search_isFound_Ty :=
  ( ∀
    (A : Type)
    (matA : (Boo_Pred A)),
    (Boo_Pred (Lis A))
  ) : Type.

Definition Lis_search_isFound := ( fun A matA l ⇒
(\text{Lis\_nonEmpty} \ A \ (\text{Lis\_search} \ A \ \text{mat} \ A \ l)) : \text{Lis\_search\_isFound\_Ty}.

\textbf{9.18.27 LIST INTERSECTION, MATCH}

\((\text{Lis\_intersect\_mat} \ A0 \ A1 \ \text{rel}01 \ l01)\) is the Boolean predicate on \(A0\) mapping \(a0 : A0\) onto \((\text{Lis\_search\_isFound} \ A1 \ (\text{rel}01 \ a0) \ l01)\).

\((\text{Lis\_intersect\_mat} \ A0 \ A1 \ \text{rel}01 \ l01)\) is the Boolean predicate on \(A0\) mapping \(a0 : A0\) onto the true Boolean if there is some \(a1 : A1\) in \(l0\) such that \((\text{rel}01 \ a0 \ a1)\); and the false Boolean otherwise.

Definition \text{Lis\_intersect\_mat\_Ty} :=

\[
\forall \\
(A0 : \text{Type}) \\
(A1 : \text{Type}) \\
(\text{rel}01 : (\text{Boo\_Pred\_Bin} \ A0 \ A1)) \\
(l01 : (\text{Lis} \ A1)), \\
(\text{Boo\_Pred} \ A0)
\]

): Type.

Definition \text{Lis\_intersect\_mat} := (\text{fun} \ A0 \ A1 \ \text{rel}01 \ l01 \ a0 \Rightarrow

(\text{Lis\_search\_isFound} \ A1 \ (\text{rel}01 \ a0) \ l01)

): \text{Lis\_intersect\_mat\_Ty}.

\textbf{9.18.28 LIST INTERSECTION}

\((\text{Lis\_intersect} \ A0 \ A1 \ \text{rel}01 \ l01 \ l11)\) is \((\text{Lis\_search} \ A0 \ (\text{Lis\_intersect\_mat} \ A0 \ A1 \ \text{rel}01 \ l01) \ l0)\).

\((\text{Lis\_intersect} \ A0 \ A1 \ \text{rel}01 \ l01 \ l11)\) is \(l0\), less those \(a0 : A0\) for which there is no \(a1 : A1\) in \(l1\) such that \((\text{rel}01 \ a0 \ a1)\).

Definition \text{Lis\_intersect\_Ty} :=

\[
\forall \\
(A0 : \text{Type}) \\
(A1 : \text{Type}) \\
(\text{rel}01 : (\text{Boo\_Pred\_Bin} \ A0 \ A1)), \\
(\text{Op\_Bin} \ (\text{Lis} \ A0) \ (\text{Lis} \ A1) \ (\text{Lis} \ A0))
\]

): Type.
Definition \( \text{Lis\_intersect} := (\text{fun A0 A1 rel01 l0 l1} \Rightarrow \text{Lis\_search A0 (Lis\_intersect\_mat A0 A1 rel01 l1) l0}) \) : \( \text{Lis\_intersect\_Ty} \).

### 9.18.29 LIST INTERSECTION, CONNECTIVE

\((\text{Lis\_intersect\_conn A relA l0 l1})\) is \((\text{Lis\_intersect A A relA l0 l1})\).

Definition \( \text{Lis\_intersect\_conn\_Ty} := \)
\[ (\forall (\text{relA} : (\text{Boo\_Pred\_Bin\_Conn A})), (\text{Op\_Bin\_Simp (Lis A)})) ) : \text{Type.} \]

Definition \( \text{Lis\_intersect\_conn} := (\text{fun relA l0 l1} \Rightarrow \text{Lis\_intersect A A relA l0 l1}) \) : \( \text{Lis\_intersect\_conn\_Ty} \).

### 9.18.30 LIST INTERSECTION, FIRST

\((\text{Lis\_intersect\_first A0 A1 rel01 l0 l1})\) is \((\text{Lis\_head A0 (Lis\_intersect A0 A1 rel01 l0 l1)})\).

Definition \( \text{Lis\_intersect\_first\_Ty} := \)
\[ (\forall (\text{A0} : \text{Type}) (\text{A1} : \text{Type}) (\text{rel01} : (\text{Boo\_Pred\_Bin A0 A1})), (\text{Op\_Bin (Lis A0) (Lis A1) (Optional A0)})) ) : \text{Type.} \]

Definition \( \text{Lis\_intersect\_first} := (\text{fun A0 A1 rel01 l0 l1} \Rightarrow \text{Lis\_head A0 (Lis\_intersect A0 A1 rel01 l0 l1)}) \) : \( \text{Lis\_intersect\_first\_Ty} \).

### 9.18.31 LIST INTERSECTION, FIRST, CONNECTIVE

\((\text{Lis\_intersect\_first\_conn A relA l0 l1})\) is \((\text{Lis\_intersect\_first A A relA l0 l1})\).
Definition \textit{Lis\_intersect\_first\_conn\_Ty} :=
\[
( \forall (A : \text{Type})
   (\text{relA} : (\text{Boo\_Pred\_Bin\_Conn A}),
   (\text{Op\_Bin\_Conn} (\text{Lis A}) (\text{Optional A})))
) : \text{Type}.
\]

Definition \textit{Lis\_intersect\_first\_conn} := (\ fun A relA l0 l1 \Rightarrow
   (\text{Lis\_intersect\_first A A relA l0 l1})
) : \text{Lis\_intersect\_first\_conn\_Ty}.

\section{LIST INTERSECTION, NON-EMPTY}

\textit{(Lis\_intersect\_nonEmpty A0 A1 rel01 l0 l1)} is \textit{(Lis\_nonEmpty A0 (Lis\_intersect A0 A1 rel01 l0 l1))}.

Definition \textit{Lis\_intersect\_nonEmpty\_Ty} :=
\[
( \forall (A0 : \text{Type})
   (A1 : \text{Type})
   (\text{rel01} : (\text{Boo\_Pred\_Bin A0 A1}),
   (\text{Boo\_Pred\_Bin} (\text{Lis A0}) (\text{Lis A1})))
) : \text{Type}.
\]

Definition \textit{Lis\_intersect\_nonEmpty} := (\ fun A0 A1 rel01 l0 l1 \Rightarrow
   (\text{Lis\_nonEmpty A0 (Lis\_intersect A0 A1 rel01 l0 l1)})
) : \text{Lis\_intersect\_nonEmpty\_Ty}.

\section{LIST INTERSECTION, NON-EMPTY, CONNECTIVE}

\textit{(Lis\_intersect\_nonEmpty\_conn A \text{relA} l0 l1)} is \textit{(Lis\_intersect\_nonEmpty A A \text{relA} l0 l1)}.

Definition \textit{Lis\_intersect\_nonEmpty\_conn\_Ty} :=
\[
( \forall (A : \text{Type})
   (\text{relA} : (\text{Boo\_Pred\_Bin\_Conn A}),
   (\text{Boo\_Pred\_Bin\_Conn} (\text{Lis A}))
) : \text{Type}.
\]
Definition \( \text{Lis\_intersect\_nonEmpty\_conn} := (\text{fun } A \text{ relA l0 l1} \Rightarrow \\
(\text{Lis\_intersect\_nonEmpty A A relA l0 l1})) : \text{Lis\_intersect\_nonEmpty\_conn\_Ty}. \)

9.18.34 LIST TO BOOLEAN PREDICATE

\( (\text{Lis\_to\_Boo\_Pred A l}) \) is the Boolean predicate on \( (\text{Boo\_Pred A}) \) mapping \( \text{matA} : (\text{Boo\_Pred A}) \) onto \( (\text{Lis\_search\_isFound A matA l}) \).

Definition \( \text{Lis\_to\_Boo\_Pred\_Ty} := \\
(\forall \\
(\text{A : Type}), \\
(\text{Op (Lis A) (Boo\_Pred (Boo\_Pred A))})) \\
) : \text{Type}. \)

Definition \( \text{Lis\_to\_Boo\_Pred} := \\
(\text{fun } A \text{ l matA} \Rightarrow (\text{Lis\_search\_isFound A matA l})) : \text{Lis\_to\_Boo\_Pred\_Ty}. \)

9.19 FUNDAMENTALS: OPTIONALS: AND LISTS SELECT

\text{Poohbist.NummSquared.Fundamentals.Optionals.AndListsSelect}


9.19.1 DEPENDENCIES


9.19.2 OPTIONAL FLATTEN LIST

\( (\text{Optional\_flattenLis A l}) \) is \( (\text{Lis\_select\_byElem (Optional A) A (Optional\_to\_Lis A) l}) \).
Definition Optional_flattenLis_Ty :=
  (∀
   (A : Type),
   (Op (Lis (Optional A)) (Lis A))
  ) : Type.

Definition Optional_flattenLis := ( fun A l ⇒
  (Lis_select_byElem (Optional A) A (Optional_to_Lis A) l)
  ) : Optional_flattenLis_Ty.

9.20  FUNDAMENTALS: LISTFUNCTIONS: MAIN


9.20.1  DEPENDENCIES


9.20.2  LISTFUNCTIONS

A listfunction from A to B is an operator from A to a list B.

Definition Lisfunction_Ty := ( ∀(A : Type)(B : Type), Type ) : Type.

Definition Lisfunction := ( fun A B ⇒ (Op A (Lis B)) ) : Lisfunction_Ty.

9.20.3  LISTFUNCTION TO BOOLEAN PREDICATE

(Lisfunction_to_Boo_Pred A B l f a) is (Lis_to_Boo_Pred B (l f a)).

Definition Lisfunction_to_Boo_Pred_Ty :=
(∀
    (A : Type)
    (B : Type),
    (Op_Bin (Lisfunction A B) A (Boo_Pred (Boo_Pred B)))
) : Type.

Definition Lisfunction_to_Boo_Pred :=
    ( fun A B lf a ⇒ (Lis_to_Boo_Pred B (lf a))
    : Lisfunction_to_Boo_Pred_Ty.

9.20.4 SIMPLE LISTFUNCTIONS

A simple listfunction on A is a listfunction from A to A.

Definition Lisfunction_Simp_Ty := ( ∀(A : Type), Type ) : Type.

Definition Lisfunction_Simp :=
    ( fun A ⇒ (Lisfunction A A) ) : Lisfunction_Simp_Ty.

9.20.5 SIMPLE LISTFUNCTION TO BOOLEAN PREDICATE

(Lisfunction_Simp_to_Boo_Pred A lf a) is (Lisfunction_to_Boo_Pred A A lf a).

Definition Lisfunction_Simp_to_Boo_Pred_Ty :=
    (∀
        (A : Type),
        (Op_Bin (Lisfunction_Simp A) A (Boo_Pred (Boo_Pred A)))
    ) : Type.

Definition Lisfunction_Simp_to_Boo_Pred :=
    ( fun A lf a ⇒ (Lisfunction_to_Boo_Pred A A lf a) )
    : Lisfunction_Simp_to_Boo_Pred_Ty.

9.20.6 SIMPLE LISTFUNCTION ITERATE

(Lisfunction_Simp_iter A lf m) is the simple listfunction on A mapping a : A onto
(Lis_select_byElem_iter A lf [A, a] m).

Definition Lisfunction_Simp_iter_Ty :=
\( (\forall (A : \text{Type}), (\text{Op} \text{ Bin} (\text{Lisfunction Simp} A) \text{ Nat} (\text{Lisfunction Simp} A))) : \text{Type}. \)

Definition \( \text{Lisfunction Simp} \_\text{iter} := (\text{fun A lf m a} \Rightarrow (\text{Lis select byElem iter A lf [A, a] m})) : \text{Lisfunction Simp} \_\text{iter} \_\text{Ty}. \)

### 9.20.7 SIMPLE LISTFUNCTION ITERATE, CURRY 2

\( (\text{Lisfunction Simp iter c2 A lf a m}) \) is \( (\text{Lisfunction Simp iter A lf m a}). \)

Definition \( \text{Lisfunction Simp iter c2} \_\text{Ty} := (\forall (A : \text{Type}), (\text{Op} \text{ Tri (Lisfunction Simp} A) \text{ A Nat} (\text{Lis} A))) : \text{Type}. \)

Definition \( \text{Lisfunction Simp iter c2} := (\text{fun A lf a m} \Rightarrow (\text{Lisfunction Simp iter A lf m a})) : \text{Lisfunction Simp iter c2} \_\text{Ty}. \)

### 9.20.8 SIMPLE LISTFUNCTION ITERATE, CUMULATIVE

\( (\text{Lisfunction Simp iter cum} A \text{ lf m}) \) is the simple listfunction on \( A \) mapping \( a : A \) onto \( (\text{Lis generate} \text{ A (Lisfunction Simp iter c2} A \text{ lf a m})). \)

Definition \( \text{Lisfunction Simp iter cum} \_\text{Ty} := (\forall (A : \text{Type}), (\text{Op} \text{ Bin} (\text{Lisfunction Simp} A) \text{ Nat} (\text{Lisfunction Simp} A))) : \text{Type}. \)

Definition \( \text{Lisfunction Simp iter cum} := (\text{fun A lf m a} \Rightarrow (\text{Lis generate} \text{ A (Lisfunction Simp iter c2} A \text{ lf a m})) : \text{Lisfunction Simp iter cum} \_\text{Ty}. \)
9.21 NUMMSQUARED: SYNTAX: ABSTRACT: MAIN


9.21.1 DEPENDENCIES


9.21.2 NUMMSQUARED DIGIT CHARACTERS

A NummSquared digit character should be interpreted as the similarly named Unicode code point in the C0 Controls and Basic Latin range. See [38, "C0 Controls and Basic Latin"].

Inductive Ns_Chr_Digit : Type :=
| Ns_Chr_Digit_d0 : Ns_Chr_Digit
| Ns_Chr_Digit_d1 : Ns_Chr_Digit
| Ns_Chr_Digit_d2 : Ns_Chr_Digit
| Ns_Chr_Digit_d3 : Ns_Chr_Digit
| Ns_Chr_Digit_d4 : Ns_Chr_Digit
| Ns_Chr_Digit_d5 : Ns_Chr_Digit
| Ns_Chr_Digit_d6 : Ns_Chr_Digit
| Ns_Chr_Digit_d7 : Ns_Chr_Digit
| Ns_Chr_Digit_d8 : Ns_Chr_Digit
| Ns_Chr_Digit_d9 : Ns_Chr_Digit.

9.21.3 NUMMSQUARED DIGIT CHARACTER EQUALS

(Ns_Chr_Digit_eq cd0 cd1) is the true Boolean if cd0 and cd1 are structurally equal;
and the false Boolean otherwise.

Definition $\text{Ns\_Chr\_Digit\_eq} := \ (\ \text{fun} \ cd0 \ cd1 \Rightarrow$

$\text{return} \ \text{Boo}$

$\text{match} \ cd0, \ cd1$

$\text{with}$

$\text{| Ns\_Chr\_Digit\_d0, Ns\_Chr\_Digit\_d0} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d1, Ns\_Chr\_Digit\_d1} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d2, Ns\_Chr\_Digit\_d2} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d3, Ns\_Chr\_Digit\_d3} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d4, Ns\_Chr\_Digit\_d4} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d5, Ns\_Chr\_Digit\_d5} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d6, Ns\_Chr\_Digit\_d6} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d7, Ns\_Chr\_Digit\_d7} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d8, Ns\_Chr\_Digit\_d8} \Rightarrow \ \text{Boo}\_t$

$\text{| Ns\_Chr\_Digit\_d9, Ns\_Chr\_Digit\_d9} \Rightarrow \ \text{Boo}\_t$

$\text{| \_\_ \Rightarrow \ \text{Boo}\_f}$

$\text{end}$

$\text{)} : (\ \text{Boo}\_\text{Pred} \_\text{Bin} \_\text{Conn} \ \text{Ns\_Chr\_Digit})$.}

\section{9.21.4 NUMMSSQUARED IDENTIFIER START CHARACTERS}

A NummSquared identifier start character should be interpreted as the similarly named Unicode code point in the C0 Controls and Basic Latin range. See [38, "C0 Controls and Basic Latin"].

Inductive $\text{Ns\_Chr\_Ident\_Start} : \text{Type} :=$

$\text{| Ns\_Chr\_Ident\_Start\_exclamationMark} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_ampersand} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_asterisk} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_plusSign} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_hyphenMinus} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_slash} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_lessThanSign} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_equalsSign} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_greaterThanSign} : \text{Ns\_Chr\_Ident\_Start}$

$\text{| Ns\_Chr\_Ident\_Start\_A} : \text{Ns\_Chr\_Ident\_Start}$
<table>
<thead>
<tr>
<th>Ns_Chr_Ident_Start_B</th>
<th>Ns_Chr_Ident_Start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ns_Chr_Ident_Start_C</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_D</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_E</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_F</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_G</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_H</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_I</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_J</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_K</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_L</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_M</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_N</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_O</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_P</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_Q</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_R</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_S</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_T</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_U</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_V</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_W</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_X</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_Y</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_Z</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_circumflexAccent</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_a</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_b</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_c</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_d</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_e</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_f</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_g</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_h</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
<tr>
<td>Ns_Chr_Ident_Start_i</td>
<td>Ns_Chr_Ident_Start</td>
</tr>
</tbody>
</table>
| Ns_Chr_Ident_Start_j : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_k : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_l : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_m : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_n : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_o : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_p : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_q : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_r : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_s : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_t : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_u : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_v : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_w : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_x : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_y : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_z : Ns_Chr_Ident_Start  
| Ns_Chr_Ident_Start_verticalBar : Ns_Chr_Ident_Start.

### 9.21.5 NUMMSQUARED IDENTIFIER START CHARACTER EQUALS

(Ns_Chr_Ident_Start_eq cis0 cis1) is the true Boolean if cis0 and cis1 are structurally equal; and the false Boolean otherwise.

Definition \( Ns\_Chr\_Ident\_Start\_eq := (fun\ cis0\ cis1 \Rightarrow \ 
match\ cis0,\ cis1 
return\ Boo 
with 
| Ns_Chr_Ident_Start_exclamationMark, 
\ Ns\_Chr\_Ident\_Start\_exclamationMark \Rightarrow \ Boo\_t 
| Ns_Chr_Ident_Start_ampersand,\ Ns\_Chr\_Ident\_Start\_ampersand \Rightarrow \ Boo\_t 
| Ns_Chr_Ident_Start_asterisk,\ Ns\_Chr\_Ident\_Start\_asterisk \Rightarrow \ Boo\_t 
| Ns_Chr_Ident_Start_plusSign,\ Ns\_Chr\_Ident\_Start\_plusSign \Rightarrow \ Boo\_t 
| Ns_Chr_Ident_Start_hyphenMinus,\ Ns\_Chr\_Ident\_Start\_hyphenMinus \Rightarrow \ Boo\_t 
)
\[ \text{Ns_Chr_Ident_Start_slash, Ns_Chr_Ident_Start_slash} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_lessThanSign, Ns_Chr_Ident_Start_lessThanSign} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_equalsSign, Ns_Chr_Ident_Start_equalsSign} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_greaterThanSign, Ns_Chr_Ident_Start_greaterThanSign} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_A, Ns_Chr_Ident_Start_A} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_B, Ns_Chr_Ident_Start_B} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_C, Ns_Chr_Ident_Start_C} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_D, Ns_Chr_Ident_Start_D} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_E, Ns_Chr_Ident_Start_E} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_F, Ns_Chr_Ident_Start_F} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_G, Ns_Chr_Ident_Start_G} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_H, Ns_Chr_Ident_Start_H} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_I, Ns_Chr_Ident_Start_I} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_J, Ns_Chr_Ident_Start_J} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_K, Ns_Chr_Ident_Start_K} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_L, Ns_Chr_Ident_Start_L} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_M, Ns_Chr_Ident_Start_M} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_N, Ns_Chr_Ident_Start_N} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_O, Ns_Chr_Ident_Start_O} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_P, Ns_Chr_Ident_Start_P} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_Q, Ns_Chr_Ident_Start_Q} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_R, Ns_Chr_Ident_Start_R} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_S, Ns_Chr_Ident_Start_S} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_T, Ns_Chr_Ident_Start_T} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_U, Ns_Chr_Ident_Start_U} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_V, Ns_Chr_Ident_Start_V} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_W, Ns_Chr_Ident_Start_W} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_X, Ns_Chr_Ident_Start_X} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_Y, Ns_Chr_Ident_Start_Y} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_Z, Ns_Chr_Ident_Start_Z} \Rightarrow \text{Boo_t} \]
\[ \text{Ns_Chr_Ident_Start_circumflexAccent, Ns_Chr_Ident_Start_circumflexAccent} \Rightarrow \]
Boo_t
   | Ns_Chr_Ident_Start_a, Ns_Chr_Ident_Start_a ⇒ Boo_t
   | Ns_Chr_Ident_Start_b, Ns_Chr_Ident_Start_b ⇒ Boo_t
   | Ns_Chr_Ident_Start_c, Ns_Chr_Ident_Start_c ⇒ Boo_t
   | Ns_Chr_Ident_Start_d, Ns_Chr_Ident_Start_d ⇒ Boo_t
   | Ns_Chr_Ident_Start_e, Ns_Chr_Ident_Start_e ⇒ Boo_t
   | Ns_Chr_Ident_Start_f, Ns_Chr_Ident_Start_f ⇒ Boo_t
   | Ns_Chr_Ident_Start_g, Ns_Chr_Ident_Start_g ⇒ Boo_t
   | Ns_Chr_Ident_Start_h, Ns_Chr_Ident_Start_h ⇒ Boo_t
   | Ns_Chr_Ident_Start_i, Ns_Chr_Ident_Start_i ⇒ Boo_t
   | Ns_Chr_Ident_Start_j, Ns_Chr_Ident_Start_j ⇒ Boo_t
   | Ns_Chr_Ident_Start_k, Ns_Chr_Ident_Start_k ⇒ Boo_t
   | Ns_Chr_Ident_Start_l, Ns_Chr_Ident_Start_l ⇒ Boo_t
   | Ns_Chr_Ident_Start_m, Ns_Chr_Ident_Start_m ⇒ Boo_t
   | Ns_Chr_Ident_Start_n, Ns_Chr_Ident_Start_n ⇒ Boo_t
   | Ns_Chr_Ident_Start_o, Ns_Chr_Ident_Start_o ⇒ Boo_t
   | Ns_Chr_Ident_Start_p, Ns_Chr_Ident_Start_p ⇒ Boo_t
   | Ns_Chr_Ident_Start_q, Ns_Chr_Ident_Start_q ⇒ Boo_t
   | Ns_Chr_Ident_Start_r, Ns_Chr_Ident_Start_r ⇒ Boo_t
   | Ns_Chr_Ident_Start_s, Ns_Chr_Ident_Start_s ⇒ Boo_t
   | Ns_Chr_Ident_Start_t, Ns_Chr_Ident_Start_t ⇒ Boo_t
   | Ns_Chr_Ident_Start_u, Ns_Chr_Ident_Start_u ⇒ Boo_t
   | Ns_Chr_Ident_Start_v, Ns_Chr_Ident_Start_v ⇒ Boo_t
   | Ns_Chr_Ident_Start_w, Ns_Chr_Ident_Start_w ⇒ Boo_t
   | Ns_Chr_Ident_Start_x, Ns_Chr_Ident_Start_x ⇒ Boo_t
   | Ns_Chr_Ident_Start_y, Ns_Chr_Ident_Start_y ⇒ Boo_t
   | Ns_Chr_Ident_Start_z, Ns_Chr_Ident_Start_z ⇒ Boo_t
   | Ns_Chr_Ident_Start_verticalBar, Ns_Chr_Ident_Start_verticalBar ⇒ Boo_t
   | _ , _ ⇒ Boo_f
end
) : (Boo_Pred_Bin_Conn Ns_Chr_Ident_Start).
9.21.6 NUMMSQUARED IDENTIFIER CONTINUE CHARACTERS

A NummSquared identifier continue character is exactly one of the following:

• a NummSquared identifier start character

• a NummSquared digit character

Note that a NummSquared identifier start character and a NummSquared digit character never have the same Unicode code point.

Inductive \(Ns_{-}Chr_{-}Ident_{-}Cont\) : Type :=

| \(Ns_{-}Chr_{-}Ident_{-}Cont\).ident_start : (Op \(Ns_{-}Chr_{-}Ident\).Start \(Ns_{-}Chr_{-}Ident\).Cont)
| \(Ns_{-}Chr_{-}Ident_{-}Cont\).digit : (Op \(Ns_{-}Chr\).Digit \(Ns_{-}Chr_{-}Ident\).Cont).

9.21.7 NUMMSQUARED IDENTIFIER CONTINUE CHARACTER EQUALS

\((Ns_{-}Chr_{-}Ident_{-}Cont\).eq cic0 cic1\) is the true Boolean if \(cic0\) and \(cic1\) are structurally equal; and the false Boolean otherwise.

Definition

\[Ns_{-}Chr_{-}Ident_{-}Cont\].eq := ( fun cic0 cic1 ⇒

match cic0, cic1

return Boo

with

| \(Ns_{-}Chr_{-}Ident_{-}Cont\).ident_start cis0,

\(Ns_{-}Chr_{-}Ident_{-}Cont\).ident_start cis1 ⇒

\((Ns_{-}Chr_{-}Ident\).Start_eq cis0 cis1\)

| \(Ns_{-}Chr_{-}Ident_{-}Cont\).digit cd0, \(Ns_{-}Chr_{-}Ident_{-}Cont\).digit cd1 ⇒

\((Ns_{-}Chr\).Digit_eq cd0 cd1\)

| _, _ ⇒ Boo_f

end

) : (Boo_Pred_Bin_Conn \(Ns_{-}Chr_{-}Ident\).Cont).

9.21.8 NUMMSQUARED COMMENTS

A NummSquared comment is an efficient natural number list.
Recall that natural numbers in the range 0-1114111 are Unicode code points. Natural numbers above this range may be interpreted in whatever way you wish.

Definition \( Ns\_Comment := Nat\_Eff\_Lis : Type. \)

**9.21.9 NUMMSQUARED COMMENT EQUALS**

\((Ns\_Comment\_eq \text{ comment0 comment1})\) is \((Ns\_Eff\_Lis\_eq \text{ comment0 comment1})\).

Definition \( Ns\_Comment\_eq := ( \text{ fun comment0 comment1 } \Rightarrow \\
(Ns\_Eff\_Lis\_eq \text{ comment0 comment1}) \\
) : (Boo\_Pred\_Bin\_Conn Ns\_Comment). \)

**9.21.10 NUMMSQUARED SIMPLE IDENTIFIERS**

A NummSquared simple identifier \( ids \) contains all of the following:

- the start of \( ids \), which is a NummSquared identifier start character

- the continues of \( ids \), which is a list of NummSquared identifier continue characters

Record \( Ns\_Ident\_Simp : Type := Ns\_Ident\_Simp\_ctor \{
Ns\_Ident\_Simp\_start : Ns\_Chr\_Ident\_Start;
Ns\_Ident\_Simp\_conts : (Lis Ns\_Chr\_Ident\_Cont)
\}. \)

**9.21.11 NUMMSQUARED SIMPLE IDENTIFIER EQUALS**

\((Ns\_Ident\_Simp\_eq ids0 ids1)\) is the true Boolean if \( ids0 \) and \( ids1 \) are structurally equal; and the false Boolean otherwise.

Definition \( Ns\_Ident\_Simp\_eq := ( \text{ fun ids0 ids1 } \Rightarrow \\
if
(Ns\_Chr\_Ident\_Start\_eq)
(Ns\_Ident\_Simp\_start ids0)
(Ns\_Ident\_Simp\_start ids1) \\
) : (Boo\_Pred\_Bin\_Conn Ns\_Ident\_Simp). \)
\texttt{return Boo then (Lis\_rel\_conn Ns\_Chr\_Ident\_Cont Ns\_Chr\_Ident\_Cont\_eq (Ns\_Ident\_Simp\_conts ids0) (Ns\_Ident\_Simp\_conts ids1)) else Boo)}

\textbf{9.21.12 NUMMSQUARED IDENTIFIERS}

A NummSquared identifier is a non-empty list of NummSquared simple identifiers. Definition \texttt{Ns\_Ident := (Lis\_Ne Ns\_Ident\_Simp)}.

\textbf{9.21.13 NUMMSQUARED IDENTIFIER EQUALS}

\texttt{(Ns\_Ident\_eq id0 id1)} is the true Boolean if \texttt{id0} and \texttt{id1} are structurally equal; and the false Boolean otherwise. Definition \texttt{Ns\_Ident\_eq := (fun id0 id1 ⇒ (Lis\_Ne\_rel\_conn Ns\_Ident\_Simp Ns\_Ident\_Simp\_eq id0 id1)) : (Boo\_Pred\_Bin\_Conn Ns\_Ident)}.

\textbf{9.21.14 NUMMSQUARED SIMPLE IDENTIFIER TO NUMMSQUARED IDENTIFIER}

\texttt{(Ns\_Ident\_Simp\_to\_Ns\_Ident ids)} is the NummSquared identifier containing just \texttt{ids}. Definition \texttt{Ns\_Ident\_Simp\_to\_Ns\_Ident := (fun ids ⇒ (Lis\_Ne\_singleton Ns\_Ident\_Simp ids)) : (Op Ns\_Ident\_Simp Ns\_Ident)}.
9.21.15 NUMMSQUARED NATURAL NUMBER PRIMITIVES

A NummSquared natural number primitive is an efficient natural number.
Definition $Ns_{Prim_{Nat}} := Nat_{Eff} : Type$.

9.21.16 NUMMSQUARED NATURAL NUMBER PRIMITIVE EQUALS

$(Ns_{Prim_{Nat}} eq m0 m1)$ is $(Nat_{Eff} eq m0 m1)$.
Definition $Ns_{Prim_{Nat}} eq := (\text{fun } m0 m1 \Rightarrow
\ (Nat_{Eff} eq m0 m1)
\ ) : (Boo_Pred_Bin_Conn Ns_{Prim_{Nat}})$.

9.21.17 NUMMSQUARED CHARACTER PRIMITIVES

A NummSquared character primitive is an efficient natural number.
Recall that natural numbers in the range 0-1114111 are Unicode code points. Natural
numbers above this range may be interpreted in whatever way you wish.
Definition $Ns_{Prim_{Chr}} := Nat_{Eff} : Type$.

9.21.18 NUMMSQUARED CHARACTER PRIMITIVE EQUALS

$(Ns_{Prim_{Chr}} eq m0 m1)$ is $(Nat_{Eff} eq m0 m1)$.
Definition $Ns_{Prim_{Chr}} eq := (\text{fun } m0 m1 \Rightarrow
\ (Nat_{Eff} eq m0 m1)
\ ) : (Boo_Pred_Bin_Conn Ns_{Prim_{Chr}})$.

9.21.19 NUMMSQUARED STRING PRIMITIVES

A NummSquared string primitive is an efficient natural number list.
Recall that natural numbers in the range 0-1114111 are Unicode code points. Natural
numbers above this range may be interpreted in whatever way you wish.
Definition $Ns_{Prim_{Str}} := Nat_{Eff_{Lis}} : Type$. 
9.21.20 NUMMSQUARED STRING PRIMITIVE EQUALS

\((Ns\_Prim\_Str\_eq \ str0 \ str1)\) is \((Ns\_Eff\_Lis\_eq \ str0 \ str1)\).

Definition \(Ns\_Prim\_Str\_eq := \(\text{fun} \ str0 \ str1 \Rightarrow\\(Nat\_Eff\_Lis\_eq \ str0 \ str1)\\) : (Boo\_Pred\_Bin\_Conn \ Ns\_Prim\_Str)\).

9.21.21 NUMMSQUARED PRIMITIVES

A NummSquared primitive is exactly one of the following:

- a NummSquared natural number primitive
- a NummSquared character primitive
- a NummSquared string primitive

Inductive \(Ns\_Prim\) : Type :=
| \(Ns\_Prim\_nat\) : (Op Ns\_Prim\_Nat Ns\_Prim)    
| \(Ns\_Prim\_chr\) : (Op Ns\_Prim\_Chr Ns\_Prim)    
| \(Ns\_Prim\_str\) : (Op Ns\_Prim\_Str Ns\_Prim).

9.21.22 NUMMSQUARED PRIMITIVE EQUALS

\((Ns\_Prim\_eq \ prim0 \ prim1)\) is the true Boolean if \(prim0\) and \(prim1\) are structurally equal (except using \(Nat\_Eff\_eq\)); and the false Boolean otherwise.

Definition \(Ns\_Prim\_eq := \(\text{fun} \ prim0 \ prim1 \Rightarrow\\match prim0, prim1\\return Boo\\with\\| Ns\_Prim\_nat m0, Ns\_Prim\_nat m1 \Rightarrow (Ns\_Prim\_Nat\_eq m0 m1)\\| Ns\_Prim\_chr m0, Ns\_Prim\_chr m1 \Rightarrow (Ns\_Prim\_Chr\_eq m0 m1)\\| Ns\_Prim\_str str0, Ns\_Prim\_str str1 \Rightarrow (Ns\_Prim\_Str\_eq str0 str1)\\| _, _ \Rightarrow Boof\\end\\) : (Boo\_Pred\_Bin\_Conn Ns\_Prim)\).
9.21.23 NUMMSQUARED COMPUTATIONAL NORMALIZED CONSTANTS

A NummSquared computational normalized constant is exactly one of the following:

• the identity NummSquared computational normalized constant
• the null NummSquared computational normalized constant
• the zero NummSquared computational normalized constant
• the one NummSquared computational normalized constant
• the null set NummSquared computational normalized constant
• the nuro set NummSquared computational normalized constant
• the leaf set NummSquared computational normalized constant
• the tree set NummSquared computational normalized constant
• the domain NummSquared computational normalized constant
• the null predicate NummSquared computational normalized constant
• the pair predicate NummSquared computational normalized constant

Inductive $Ns\_Constant\_Norm\_Compu : Type :=$

| $Ns\_Constant\_Norm\_Compu\_i : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_null : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_zero : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_one : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_Null\_set : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_Nuro\_set : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_Leaf\_set : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_Tree\_set : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_dom : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_Null : Ns\_Constant\_Norm\_Compu$
| $Ns\_Constant\_Norm\_Compu\_Pair : Ns\_Constant\_Norm\_Compu$. 
9.21.24 **NUMMSQUARED NON-COMPUTATIONAL NORMALIZED CONSTANTS**

A NummSquared non-computational normalized constant is exactly one of the following:

- the equals NummSquared non-computational normalized constant

Inductive \texttt{Ns\_Constant\_Norm\_Noncompu} : \texttt{Type} :=
| \texttt{Ns\_Constant\_Norm\_Noncompu\_ns\_eq} : \texttt{Ns\_Constant\_Norm\_Noncompu}. 

9.21.25 **NUMMSQUARED NORMALIZED CONSTANTS**

A NummSquared normalized constant is exactly one of the following:

- a NummSquared computational normalized constant
- a NummSquared non-computational normalized constant

Inductive \texttt{Ns\_Constant\_Norm} : \texttt{Type} :=
| \texttt{Ns\_Constant\_Norm\_compu} : (\texttt{Op Ns\_Constant\_Norm\_Compu Ns\_Constant\_Norm})
| \texttt{Ns\_Constant\_Norm\_noncompu} :
  (\texttt{Op Ns\_Constant\_Norm\_Noncompu Ns\_Constant\_Norm}). 

9.21.26 **NUMMSQUARED COMPUTATIONAL NON-NORMALIZED CONSTANTS**

A NummSquared computational non-normalized constant is exactly one of the following:

- the left NummSquared computational non-normalized constant
- the right NummSquared computational non-normalized constant
• the confirmation with null NummSquared computational non-normalized constant
• the negation with null NummSquared computational non-normalized constant
• the null to zero NummSquared computational non-normalized constant
• the zero predicate NummSquared computational non-normalized constant
• the one predicate NummSquared computational non-normalized constant
• the nuro predicate NummSquared computational non-normalized constant
• the leaf predicate NummSquared computational non-normalized constant
• the simple predicate NummSquared computational non-normalized constant
• the rule predicate NummSquared computational non-normalized constant
• the tree predicate step pair unguarded NummSquared computational non-normalized constant
• the tree predicate step unguarded NummSquared computational non-normalized constant
• the tree predicate NummSquared computational non-normalized constant
• the non-empty domain NummSquared computational non-normalized constant
• the result NummSquared computational non-normalized constant
• the nuro set result NummSquared computational non-normalized constant
• the tree set result NummSquared computational non-normalized constant
• the dependent sum result left unguarded NummSquared computational non-normalized constant
• the dependent sum result right unguarded NummSquared computational non-normalized constant
• the dependent sum result unguarded NummSquared computational non-normalized constant
• the dependent sum result NummSquared computational non-normalized constant
• the dependent product result uncurry unguarded NummSquared computational non-normalized constant
• the dependent product result unguarded NummSquared computational non-normalized constant
• the dependent product result NummSquared computational non-normalized constant
• the negation NummSquared computational non-normalized constant
• the implication with null NummSquared computational non-normalized constant
• the implication NummSquared computational non-normalized constant

Inductive \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} : Type :=
\[\begin{array}{l}
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore left} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore right} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore conf\textunderscore n} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore not\textunderscore n} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Null\textunderscore to\textunderscore Zero} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Zero} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore One} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Nuro} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Leaf} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Simp} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Rule} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Tree\textunderscore step\textunderscore pair\textunderscore ug} : \\
\end{array}\]
\texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu}
\[\begin{array}{l}
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Tree\textunderscore step\textunderscore ug} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Tree} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore dom\textunderscore ne} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore res} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu} \\
| \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu\textunderscore Nuro\textunderscore set\textunderscore res} : \texttt{Ns\textunderscore Constant\textunderscore Nonnorm\textunderscore Compu}
\]
9.21.27 NUMMSQUARED NON-COMPUTATIONAL NON-NORMALIZED CONSTANTS

A NummSquared non-computational non-normalized constant is exactly one of the following:

- the not equals NummSquared non-computational non-normalized constant
- the small universal quantification NummSquared non-computational non-normalized constant
- the equal pairs unguarded NummSquared non-computational non-normalized constant
- the equal results at NummSquared non-computational non-normalized constant
- the equal results NummSquared non-computational non-normalized constant
- the equal domain results NummSquared non-computational non-normalized constant
- the equal both results NummSquared non-computational non-normalized constant
• the equals right-hand-side NummSquared non-computational non-normalized constant

Inductive \textit{Ns\_Constant\_Nonnorm\_Noncompu} : Type :=
\begin{itemize}
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_not\_eq} : \textit{Ns\_Constant\_Nonnorm\_Noncompu}
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_all\_sm} : \textit{Ns\_Constant\_Nonnorm\_Noncompu}
\end{itemize}
\textit{Ns\_Constant\_Nonnorm\_Noncompu}\_
\begin{itemize}
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_eq\_pair\_ug} :
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_eq\_res\_at} :
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_eq\_res} :
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_eq\_dom\_res} :
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_eq\_both\_res} :
\item \textit{Ns\_Constant\_Nonnorm\_Noncompu\_eq\_rhs} : \textit{Ns\_Constant\_Nonnorm\_Noncompu}.
\end{itemize}

\textbf{9.21.28  NUMMSQUARED NON-NORMALIZED CONSTANTS}

A NummSquared non-normalized constant is exactly one of the following:
• a NummSquared computational non-normalized constant
• a NummSquared non-computational non-normalized constant

Inductive \textit{Ns\_Constant\_Nonnorm} : Type :=
\begin{itemize}
\item \textit{Ns\_Constant\_Nonnorm\_compu} :
\begin{itemize}
\item \textit{Op Ns\_Constant\_Nonnorm\_Compu Ns\_Constant\_Nonnorm\_Noncompu}\_
\item \textit{Op Ns\_Constant\_Nonnorm\_Noncompu Ns\_Constant\_Nonnorm\_Noncompu}\_
\end{itemize}
\item \textit{Ns\_Constant\_Nonnorm\_noncompu} :
\begin{itemize}
\item \textit{Op Ns\_Constant\_Nonnorm\_Noncompu Ns\_Constant\_Nonnorm\_Noncompu}\_
\end{itemize}
\end{itemize}

\textbf{9.21.29  NUMMSQUARED CONSTANTS}

A NummSquared constant is exactly one of the following:
• a NummSquared normalized constant

• a NummSquared non-normalized constant

\textbf{Inductive} \texttt{Ns\_Constant} : \textit{Type} :=
| \texttt{Ns\_Constant\_norm} : (\texttt{Op Ns\_Constant\_Norm Ns\_Constant})
| \texttt{Ns\_Constant\_nonnorm} : (\texttt{Op Ns\_Constant\_Nonnorm Ns\_Constant}).

\textbf{9.21.30 NUMMSQUARED LARGE FUNCTIONS}

A NummSquared large composition computational combination \( c \) contains all of the following:
• the outer of \( c \), which is a NummSquared large function

• the inners of \( c \), which is a NummSquared large function non-empty list

A NummSquared small composition computational combination \( c \) contains all of the following:
• the called and arguments of \( c \), which is a NummSquared large function 2 plus list

A NummSquared tuple computational combination \( c \) contains all of the following:
• the components of \( c \), which is a NummSquared large function 2 plus list

A NummSquared list computational combination \( c \) contains all of the following:
• the elements of \( c \), which is a NummSquared large function list

A NummSquared dependent sum computational combination \( c \) contains all of the following:
• the family of \( c \), which is a NummSquared large function

A NummSquared dependent product computational combination \( c \) contains all of the following:
• the family of $c$, which is a NummSquared large function

A NummSquared Curry computational combination $c$ contains all of the following:

• the root of $c$, which is a NummSquared large function
• the restrictor of $c$, which is a NummSquared large function

A NummSquared if-then-else computational combination $c$ contains all of the following:

• the if-part of $c$, which is a NummSquared large function
• the then-part of $c$, which is a NummSquared large function
• the else-part of $c$, which is a NummSquared large function

A NummSquared recursion computational combination $c$ contains all of the following:

• the start of $c$, which is a NummSquared large function
• the step of $c$, which is a NummSquared large function

A NummSquared restrict computational combination $c$ contains all of the following:

• the root of $c$, which is a NummSquared large function

A NummSquared restrict to range computational combination $c$ contains all of the following:

• the root of $c$, which is a NummSquared large function

A NummSquared Curry augmented root computational combination $c$ contains all of the following:

• the root of $c$, which is a NummSquared large function
• the augmentor of $c$, which is a NummSquared large function
A NummSquared Curry augmented computational combination $c$ contains all of the following:

- the root of $c$, which is a NummSquared large function
- the restrictor of $c$, which is a NummSquared large function
- the augmentor of $c$, which is a NummSquared large function

A NummSquared Curry result computational combination $c$ contains all of the following:

- the root of $c$, which is a NummSquared large function

A NummSquared recursion on domain computational combination $c$ contains all of the following:

- the start of $c$, which is a NummSquared large function
- the step of $c$, which is a NummSquared large function

A NummSquared recursion on range computational combination $c$ contains all of the following:

- the start of $c$, which is a NummSquared large function
- the step of $c$, which is a NummSquared large function

A NummSquared recursion step computational combination $c$ contains all of the following:

- the start of $c$, which is a NummSquared large function
- the step of $c$, which is a NummSquared large function

A NummSquared recursion right-hand-side computational combination $c$ contains all of the following:

- the start of $c$, which is a NummSquared large function
• the step of \( c \), which is a NummSquared large function

A NummSquared computational combination is exactly one of the following:

• a NummSquared large composition computational combination
• a NummSquared small composition computational combination
• a NummSquared tuple computational combination
• a NummSquared list computational combination
• a NummSquared dependent sum computational combination
• a NummSquared dependent product computational combination
• a NummSquared Curry computational combination
• a NummSquared if-then-else computational combination
• a NummSquared recursion computational combination
• a NummSquared restrict computational combination
• a NummSquared restrict to range computational combination
• a NummSquared Curry augmented root computational combination
• a NummSquared Curry augmented computational combination
• a NummSquared Curry result computational combination
• a NummSquared recursion on domain computational combination
• a NummSquared recursion on range computational combination
• a NummSquared recursion step computational combination
• a NummSquared recursion right-hand-side computational combination

A NummSquared Hilbert non-computational combination \( c \) contains all of the following:
• the predicate of $c$, which is a NummSquared large function

A NummSquared existential quantification unguarded non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function

A NummSquared existential quantification non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function

A NummSquared not universal quantification non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function

A NummSquared universal quantification non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function

A NummSquared unary universal quantification non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function

A NummSquared inductive domain hypothesis non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function

A NummSquared inductive range hypothesis non-computational combination $c$ contains all of the following:

• the predicate of $c$, which is a NummSquared large function
A NummSquared inductive case at non-computational combination \( c \) contains all of the following:

- the predicate of \( c \), which is a NummSquared large function

A NummSquared inductive case non-computational combination \( c \) contains all of the following:

- the predicate of \( c \), which is a NummSquared large function

A NummSquared non-computational combination is exactly one of the following:

- a NummSquared Hilbert non-computational combination
- a NummSquared existential quantification unguarded non-computational combination
- a NummSquared existential quantification non-computational combination
- a NummSquared not universal quantification non-computational combination
- a NummSquared universal quantification non-computational combination
- a NummSquared unary universal quantification non-computational combination
- a NummSquared inductive domain hypothesis non-computational combination
- a NummSquared inductive range hypothesis non-computational combination
- a NummSquared inductive case at non-computational combination
- a NummSquared inductive case non-computational combination

A NummSquared combination is exactly one of the following:

- a NummSquared computational combination
- a NummSquared non-computational combination

A NummSquared computation \( computation \) contains all of the following:
• the called of computation, which is a NummSquared large function

A NummSquared quotation quotation contains all of the following:

• the unquoted of quotation, which is a NummSquared large function

A NummSquared unquotation unquotation contains all of the following:

• the quoted of unquotation, which is a NummSquared large function

A NummSquared macro expansion macroExpansion contains all of the following:

• the called of macroExpansion, which is a NummSquared large function

• the arguments of macroExpansion, which is a NummSquared large function list

A NummSquared large function is exactly one of the following:

• a NummSquared primitive

• a NummSquared constant

• a NummSquared combination

• for some NummSquared identifier id, the global name NummSquared large function of id

• for some NummSquared identifier id, the local name NummSquared large function of id

• a NummSquared computation

• a NummSquared quotation

• a NummSquared unquotation

• a NummSquared macro expansion

A NummSquared large function list is exactly one of the following:

• the nil NummSquared large function list
• for some NummSquared large function head and NummSquared large function list rest, the cons NummSquared large function list of head and rest.

A NummSquared large function non-empty list l contains all of the following:
• the head of l, which is a NummSquared large function
• the rest of l, which is a NummSquared large function list

A NummSquared large function 2 plus list l contains all of the following:
• the head of l, which is a NummSquared large function
• the rest of l, which is a NummSquared large function non-empty list

Inductive Ns_Combo_Compu_Co_Lg : Type :=
    | Ns_Combo_Compu_Co_Lgctor :
        (Op_Bin Ns_Func_Lg Ns_Func_Lg_Lis_Ne Ns_Combo_Compu_Co_Lg)
    with Ns_Combo_Compu_Co_Sm : Type :=
        | Ns_Combo_Compu_Co_Smctor : (Op Ns_Func_Lg_Lis_P2
Ns_Combo_Compu_Co_Sm)
    with Ns_Combo_Compu_Tuple : Type :=
        | Ns_Combo_Compu_Tuplector : (Op Ns_Func_Lg_Lis_P2
Ns_Combo_Compu_Tuple)
    with Ns_Combo_Compu_Lis : Type :=
        | Ns_Combo_Compu_Liscctor : (Op Ns_Func_Lg_Lis Ns_Combo_Compu_Lis)
    with Ns_Combo_Compu_S_D : Type :=
        | Ns_Combo_Compu_S_Dctor : (Op Ns_Func_Lg Ns_Combo_Compu_S_D)
    with Ns_Combo_Compu_P_D : Type :=
        | Ns_Combo_Compu_P_Dctor : (Op Ns_Func_Lg Ns_Combo_Compu_P_D)
    with Ns_Combo_Compu_C : Type :=
        | Ns_Combo_Compu_Cctor : (Op_Bin_Conn Ns_Func_Lg Ns_Combo_Compu_C)
    with Ns_Combo_Compu_Ite : Type :=
        | Ns_Combo_Compu_Itector : (Op_Tri_Conn Ns_Func_Lg Ns_Combo_Compu_Ite)
    with Ns_Combo_Compu_R : Type :=
        | Ns_Combo_Compu_Rctor : (Op_Bin_Conn Ns_Func_Lg Ns_Combo_Compu_R)
    with Ns_Combo_Compu_Restrict : Type :=
| Ns_Combo_Compu_Restrict_ctor : (Op Ns_Func_Lg Ns_Combo_Compu_Restrict) with Ns_Combo_Compu_Restrict_Ran : Type :=
| Ns_Combo_Compu_Restrict_Ran_ctor :
  (Op Ns_Func_Lg Ns_Combo_Compu_Restrict_Ran) with Ns_Combo_Compu_C_Aug Root : Type :=
| Ns_Combo_Compu_C_Aug_ctor : (Op Tri_Conn Ns_Func_Lg
Ns_Combo_Compu_C_Aug) with Ns_Combo_Compu_C_Res : Type :=
| Ns_Combo_Compu_C_Res_ctor : (Op Ns_Func_Lg Ns_Combo_Compu_C_Res) with Ns_Combo_Compu_R_Dom : Type :=
| Ns_Combo_Compu_R_Dom_ctor : (Op Bin_Conn Ns_Func_Lg
Ns_Combo_Compu_R_Dom) with Ns_Combo_Compu_R_Ran : Type :=
| Ns_Combo_Compu_R_Ran_ctor : (Op Bin_Conn Ns_Func_Lg
Ns_Combo_Compu_R_Ran) with Ns_Combo_Compu_R_Step : Type :=
| Ns_Combo_Compu_R_Step_ctor :
  (Op Bin_Conn Ns_Func_Lg Ns_Combo_Compu_R_Step) with Ns_Combo_Compu_R_Rhs : Type :=
| Ns_Combo_Compu_R_Rhs_ctor : (Op Bin_Conn Ns_Func_Lg
Ns_Combo_Compu_R_Rhs) with Ns_Combo_Compu : Type :=
| Ns_Combo_Compu_co_Lg : (Op Ns_Combo_Compu_CO_Lg Ns_Combo_Compu)
| Ns_Combo_Compu_co_sm : (Op Ns_Combo_Compu_CO_Sm Ns_Combo_Compu)
| Ns_Combo_Compu_tuple : (Op Ns_Combo_Compu_Tuple Ns_Combo_Compu)
| Ns_Combo_Compu_lis : (Op Ns_Combo_Compu_Lis Ns_Combo_Compu)
| Ns_Combo_Compu_s_d : (Op Ns_Combo_Compu_S_D Ns_Combo_Compu)
| Ns_Combo_Compu_p_d : (Op Ns_Combo_Compu_P_D Ns_Combo_Compu)
| Ns_Combo_Compu_c : (Op Ns_Combo_Compu_C Ns_Combo_Compu)
| Ns_Combo_Compu_ite : (Op Ns_Combo_Compu_Ite Ns_Combo_Compu)
| Ns_Combo_Compu_r : (Op Ns_Combo_Compu_R Ns_Combo_Compu)
| Ns_Combo_Compu_restrict : (Op Ns_Combo_Compu_Restrict Ns_Combo_Compu)
| Ns_Combo_Compu_restrict_ran : (Op Ns_Combo_Compu_Restrict_Ran Ns_Combo_Compu) |
| Ns_Combo_Compu_c_aug_root : (Op Ns_Combo_Compu_C_Aug_Root Ns_Combo_Compu) |
| Ns_Combo_Compu_c_aug : (Op Ns_Combo_Compu_C_Aug Ns_Combo_Compu) |
| Ns_Combo_Compu_c_res : (Op Ns_Combo_Compu_C_Res Ns_Combo_Compu) |
| Ns_Combo_Compu_r_dom : (Op Ns_Combo_Compu_R_Dom Ns_Combo_Compu) |
| Ns_Combo_Compu_r_ran : (Op Ns_Combo_Compu_R_Ran Ns_Combo_Compu) |
| Ns_Combo_Compu_r_step : (Op Ns_Combo_Compu_R_Step Ns_Combo_Compu) |
| Ns_Combo_Compu_r_rhs : (Op Ns_Combo_Compu_R_Rhs Ns_Combo_Compu) |
| with Ns_Combo_Noncompu_H : Type := |
| Ns_Combo_Noncompu_H_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_H) |
| with Ns_Combo_Noncompu_Exist_Ug : Type := |
| Ns_Combo_Noncompu_Exist_Ug_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_Exist_Ug) |
| with Ns_Combo_Noncompu_Exist : Type := |
| Ns_Combo_Noncompu_Exist_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_Exist) |
| with Ns_Combo_Noncompu_Not_All : Type := |
| Ns_Combo_Noncompu_Not_All_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_Not_All) |
| with Ns_Combo_Noncompu_All : Type := |
| Ns_Combo_Noncompu_All_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_All) |
| with Ns_Combo_Noncompu_All_UNA : Type := |
| Ns_Combo_Noncompu_All_UNA_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_All_UNA) |
| with Ns_Combo_Noncompu_Induc_Hyp_Dom : Type := |
| Ns_Combo_Noncompu_Induc_Hyp_Dom_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_Induc_Hyp_Dom) |
| with Ns_Combo_Noncompu_Induc_Hyp_Ran : Type := |
| Ns_Combo_Noncompu_Induc_Hyp_Ran_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_Induc_Hyp_Ran) |
| with Ns_Combo_Noncompu_Induc_Case_At : Type := |
| Ns_Combo_Noncompu_Induc_Case_At_ctor : (Op Ns_Func_Lg Ns_Combo_Noncompu_Induc_Case_At) |
| with Ns_Combo_Noncompu_Induc_Case : Type := |
| Ns_Combo_Noncompu_Induc_Case_ctor : 
  (Op Ns_Func_Lg Ns_Combo_Noncompu_Induc_Case) 
with Ns_Combo_Noncompu : Type := 
| Ns_Combo_Noncompu_h : (Op Ns_Combo_Noncompu_H Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_exist_ug : 
  (Op Ns_Combo_Noncompu_Exist_Ug Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_exist : (Op Ns_Combo_Noncompu_Exist Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_not_all : 
  (Op Ns_Combo_Noncompu_Not_All Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_all : (Op Ns_Combo_Noncompu_All Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_all_una : 
  (Op Ns_Combo_Noncompu_All_Una Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_induc_hyp_dom : 
  (Op Ns_Combo_Noncompu_Induc_Hyp_Dom Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_induc_hyp_ran : 
  (Op Ns_Combo_Noncompu_Induc_Hyp_Ran Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_induc_case_at : 
  (Op Ns_Combo_Noncompu_Induc_Case_At Ns_Combo_Noncompu) 
| Ns_Combo_Noncompu_induc_case : 
  (Op Ns_Combo_Noncompu_Induc_Case Ns_Combo_Noncompu) 
with Ns_Combo : Type := 
| Ns_Combo_compu : (Op Ns_Combo_Compu Ns_Combo) 
| Ns_Combo_noncompu : (Op Ns_Combo_Noncompu Ns_Combo) 
with Ns_Computation : Type := 
| Ns_Computation_ctor : (Op Ns_Func_Lg Ns_Computation) 
with Ns_Quotation : Type := 
| Ns_Quotation_ctor : (Op Ns_Func_Lg Ns_Quotation) 
with Ns_Unquotation : Type := 
| Ns_Unquotation_ctor : (Op Ns_Func_Lg Ns_Unquotation) 
with Ns_Macro_Expansion : Type := 
| Ns_Macro_Expansion_ctor : 
  (Op_Bin Ns_Func_Lg Ns_Func_Lg_Lis Ns_Macro_Expansion) 
with Ns_Func_Lg : Type :=
\begin{verbatim}
| Ns_Func_Lg_prim : (Op Ns_Prim Ns_Func_Lg)
| Ns_Func_Lg_constant : (Op Ns_Constant Ns_Func_Lg)
| Ns_Func_Lg_combo : (Op Ns_Combo Ns_Func_Lg)
| Ns_Func_Lg_name_glob : (Op Ns_Ident Ns_Func_Lg)
| Ns_Func_Lg_name_loc : (Op Ns_Ident Ns_Func_Lg)
| Ns_Func_Lg_computation : (Op Ns_Computation Ns_Func_Lg)
| Ns_Func_Lg_quotation : (Op Ns_Quotation Ns_Func_Lg)
| Ns_Func_Lg_unquotation : (Op Ns_Unquotation Ns_Func_Lg)
| Ns_Func_Lg_macro_expansion : (Op Ns_Macro_Expansion Ns_Func_Lg)
with Ns_Func_Lg_Lis : Type :=
| Ns_Func_Lg_Lis_nil : Ns_Func_Lg_Lis
| Ns_Func_Lg_Lis_cons : (Op_Bin Ns_Func_Lg Ns_Func_Lg_Lis Ns_Func_Lg_Lis)
with Ns_Func_Lg_Lis_Ne : Type :=
| Ns_Func_Lg_Lis_Ne_ctor :
    (Op_Bin Ns_Func_Lg Ns_Func_Lg_Lis Ns_Func_Lg_Lis_Ne)
with Ns_Func_Lg_Lis_P2 : Type :=
| Ns_Func_Lg_Lis_P2_ctor :
    (Op_Bin Ns_Func_Lg Ns_Func_Lg_Lis_Ne Ns_Func_Lg_Lis_P2).
\end{verbatim}

\section{NUMMSQUARED LOCAL TUPLE ACCESSOR LISTS}

A NummSquared local tuple accessor list is a 2 plus list of NummSquared identifiers.

The order is reversed relative to the concrete syntax.

Definition \texttt{Ns\_Access\_Tuple\_Loc\_Lis} := (\texttt{Lis\_P2 Ns\_Ident}).

\section{NUMMSQUARED LOCAL CONTEXTS}

A NummSquared local context is an optional NummSquared local tuple accessor list.

Definition \texttt{Ns\_Context\_Loc} := (Optional Ns\_Access\_Tuple\_Loc\_Lis).
9.21.33 NUMMSQUARED DEFINITIONS

Record Ns_Def : Type := Ns_Def_ctor {
    Ns_Def_comment : Ns_Comment;
    Ns_Def_name : Ns_Ident;
    Ns_Def_context_loc : Ns_Context_Loc;
    Ns_Def_rhs : Ns Func_Lg
}.  

9.21.34 NUMMSQUARED GLOBAL CONTEXTS

The order is reversed relative to the concrete syntax.
Definition Ns_Context_Glob := (Lis Ns_Def).

9.21.35 NUMMSQUARED MODULES

Record Ns_Modu : Type := Ns_Modu_ctor {
    Ns_Modu_comment : Ns_Comment;
    Ns_Modu_name : Ns_Ident;
    Ns_Modu_context_glob : Ns_Context_Glob
}.  

9.21.36 NUMMSQUARED ABSTRACT PROGRAMS

The order is reversed relative to the concrete syntax.
Definition Ns_Program_Abs := (Lis Ns_Modu).
CHAPTER 10

CONCLUSION

NummSquared is a formal language, and a new well-founded functional foundation for logic, mathematics and computer science. Functions are the only fundamental concept in NummSquared. NummSquared includes reduction and ensures that it always terminates. NummSquared minimizes constraints on the logician, mathematician or programmer. Because of coercion, there are no types, and functions are defined and called without proof, yet reduction terminates. NummSquared supports proof as desired but not required, is variable-free, supports reflection, and has an interpreter called NsGo (work in progress) so the language can be practically used. NummSquared has a classical logic, and attempts to follow set theory as much as possible.

NummSquared coercion is (loosely) a generalization to higher order functions of coercion (type conversion) found in many programming languages. For coercion and computational reasons, the domain of a rule small function extension is represented by a domain extension. A domain extension contains the same information as a type in type theory, but with a different purpose.

Among the important theorems about NummSquared are:

- domain extension irrelevance: domain extensions contain no more information than their domains
- tag irrelevance: because of the domain extension irrelevance theorem, tagging adds no information
- coercion stability: coercion does not make unnecessary changes
- extensionality: characterizes equals on rule tagged small function extensions
- substitution: substitution preserves equality
• soundness: the proposition of a valid proof is true

Among the important definitions about NummSquared are small function extensions, domain extensions, tagged small function extensions, coercion (defined by well-founded tango), generalized result, large function extensions, truth of a tagged small function extension or large function extension, Curry, recursion, equals, Hilbert, normalized large functions, extension and truth of a normalized large function, reduction (terminating by definition), quoted and unquoted for normalized large functions, macro expanded, substitution, large functions, normal forms and validity, proofs, proposition and validity of a proof, and quoted and unquoted for proofs.


INDEX

abstract program, 101
axiom, 158
Boolean, 17, 24
character primitive, 89
coection, 53, 56
coection pair, 53
coection stability theorem, 56
combination, 94
combination domain extension, 30
comment, 88
computation, 99
computational combination, 94
computational non-normalized constant, 90
computational normalized combination, 76
computational normalized constant, 74
computed, 83
concatenation, 19, 101
constant, 94
constant domain extension, 30
constant large function extension, 67
constant model, 18
Curry, 69
Curry augmented computational combination, 96
Curry augmented uncurry computational combination, 96
Curry computational combination, 96
Curry computational normalized combination, 77
Curry domain axiom, 148
Curry if-then-else axiom, 149
Curry large composition axiom, 147
Curry null predicate axiom, 147
Curry pair predicate axiom, 148
Curry result computational combination, 97
Curry small composition axiom, 148
depth computational, 79
definition, 100
definition list, 100–102
dependent product, 65, 69
dependent product computational combination, 96
dependent product computational normalized combination, 77
Dependent product domain axiom, 146
dependent product domain extension, 30
Dependent product if-then-else axiom, 147
Dependent product large composition axiom, 145
Dependent product null predicate axiom, 146
Dependent product pair predicate axiom, 146
Dependent product small composition axiom, 146
dependent sum, 65, 69
inferred domain extension, 38
informal, 13
informal part, 13, 14
interpretation, 18
intersection, 18
language, 13
large composition, 68
large composition computational combination, 95
large composition computational normalized combination, 76
Large composition large composition axiom, 141
large function, 94
large function extension, 66
Leaf set domain axiom, 139
Leaf set if-then-else axiom, 139
Leaf set large composition axiom, 138
Leaf set null predicate axiom, 139
Leaf set pair predicate axiom, 139
Leaf set small composition axiom, 139
leaf small function extension, 23
left, 18, 24, 41
length, 19
like this, 16
list, 19, 100, 103
list computational combination, 96
local context, 102, 104
local name, 98
local tuple accessor checker, 99
local tuple accessor descriptor, 100
local tuple accessor list, 99
Logic contrapositive axiom, 151
Logic nested implication axiom, 150
Logic weakening axiom, 150
logical reflection, 11
lowercase letter character, 88
macro expanded, 87
macro expansion, 99
macro pre-expanded, 87
model, 18
module, 101
module name list, 101
Modus ponens inference, 156, 157
modus ponens inference, 159
name, 100–102
natural number, 16
natural number primitive, 89
negation, 17
non-computational combination, 97
non-computational non-normalized constant, 93
non-computational normalized combination, 77
non-computational normalized constant, 75
non-normalized constant, 93
normal form, 82, 84, 104, 106, 108, 121, 128–130
normalized combination, 76
normalized constant, 75
normalized definition, 102
normalized large function, 75
normalized local tuple accessor checker, 103
normalized local tuple accessor descriptor, 103
normalized proposition, 79
normalized result, 82
not universal quantification non-computational combination, 98
NsGo, 5
Null domain axiom, 132
Null if-then-else axiom, 133
Null large composition axiom, 132
Null null predicate axiom, 132
Null pair predicate axiom, 132
Null predicate otherwise axiom, 154
null rule small function extension, 32
Null set domain axiom, 136
Null set if-then-else axiom, 137
Null set large composition axiom, 136
Null set null predicate axiom, 136
Null set pair predicate axiom, 136
small composition computational normalized combination, 76
Small composition large composition axiom, 142
small function extension, 23
soundness theorem, 160
source, 18
Specialization inference, 157, 158
specialization inference, 159
specific result, 25, 31, 48
string primitive, 89
sub-language, 17
substitutes, 87
Substitution inference, 158
substitution inference, 159
substitution theorem, 88
tag, 85, 161
tag irrelevance theorem, 42
taggable, 45
tagged, 44, 46, 47
tagged small function extension, 40
tail, 19
to, 18
tree, 24, 43, 80
Tree set domain axiom, 140
Tree set if-then-else axiom, 141
Tree set large composition axiom, 140
Tree set null predicate axiom, 140
Tree set pair predicate axiom, 140
Tree set small composition axiom, 141
true, 66, 67, 79
Truth elimination axiom, 151
Truth introduction axiom, 151
tuple computational combination, 96
tuple locator, 106
unary universal quantification non-computational combination, 98
unchanging, 66, 79
Unicode code point, 16
union, 18
universal quantification non-computational combination, 98
universally true, 66
unquotation, 99
unquoted, 86
untaggable, 45
untagged, 41
uppercase letter character, 88
valid, 32, 102–104, 129, 130, 160
valid quoted proof, 162
well-founded, 4, 20
Zero domain axiom, 133
Zero if-then-else axiom, 134
Zero large composition axiom, 133
Zero null predicate axiom, 133
Zero pair predicate axiom, 133
Zero small composition axiom, 134
zero small function extension, 22