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

The signature page goes here.

DALHOUSIE UNIVERSITY

Date: October 10, 2006

AUTHOR: Samuel Howse

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

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: PhD CONVOCATION: May YEAR: 2007

Signature of Author

For the inspirational Dr. L. S. River, and Nummists everywhere.
Visit `http://nummist.com/`.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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. (not (x x))), consider (‘R ‘R), which reduces to (not (‘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 reponse 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 membership. (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.

von Neumann	combinatory logic
axiom II.1	I combinator
axiom II.2	K combinator
axiom II.6	S combinator

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 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.

2.7 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 **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 ‘L0 and ‘L1, ‘L0 = ‘L1 iff, for each thing ‘x, ‘x is an ‘L0 program iff ‘x is an ‘L1 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 $\vdash$ T = F $\rightarrow$ 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 ≠ 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 ≥ 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 ‘left(‘t) is an ‘m - 1 tuple.

For a natural number ‘m ≥ 2, and things ‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$, let <‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$> be the ‘m tuple <<<‘x_0, ‘x_1>, ..., ‘$x_{‘m-2}$>, ‘$x_{‘m-1}$>.

For a pair ‘p = <‘l, ‘r>, let ‘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 ‘head(‘l), is ‘h; and the **rest** of ‘l, denoted by ‘rest(‘l), is ‘r.

For a list ‘l, the **length** of ‘l, denoted by ‘len(‘l), is defined by recursion on ‘l:

- 0 if ‘l = 0
- ‘len(‘r) + 1 if ‘l = <‘h, ‘r>

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

For a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, an **element** of ‘l is one of ‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$.

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, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$>, the **tail** of ‘l, denoted by ‘tail(‘l), is ‘$x_{‘m-1}$; and the **pretail** of ‘l, denoted by ‘pretail(‘l), is l<‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$>.

For lists ‘l0 = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$> and ‘l1 = l<‘y_0, ‘y_1, ..., ‘$y_{‘n-1}$>, the **concatenation** of ‘l0 and ‘l1, denoted by ‘l0 + ‘l1, is l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$, ‘y_0, ‘y_1, ..., ‘$y_{‘n-1}$>.

For a property ‘P, and a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, the **search** for ‘P in ‘l is the list of those <0, ‘x_0>, <1, ‘x_1>, ..., <‘m-1, ‘$x_{‘m-1}$> whose right satisfies ‘P (in order).

For a property ‘P, and a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, the **search first** for ‘P in ‘l is the head of the search for ‘P in ‘l if the search for ‘P in ‘l is non-empty; and ‘null otherwise.

For a property ‘P, and a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, the **search first index** for ‘P in ‘l is ‘null if the search first for ‘P in ‘l is ‘null; and the left of the search first for ‘P in ‘l otherwise.

For a property ‘P, and a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, the **search first data** for ‘P in ‘l is ‘null if the search first for ‘P in ‘l is ‘null; and the right of the search first for ‘P in ‘l otherwise.

For a property ‘P, and a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, the **search length** for ‘P in ‘l is the length of the search for ‘P in ‘l.

For a property ‘P, and a list ‘l = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, ‘P is **duplicitous** in ‘l iff the search length for ‘P in ‘l is > 1.

6.8 WELL-FOUNDED RELATIONS

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

6.9 SMALL LANGUAGES

For a language ‘L, ‘L is **small** iff there exists some ZFC set ‘s (see [36, p.84,132-133]) and some model ‘m from ‘s such that ‘L is a sub-language of ‘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. NummSquared 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.null and ‘y = ‘Func.Sm.Ext.null.
- ‘x = ‘Func.Sm.Ext.zero and ‘y = ‘Func.Sm.Ext.zero.
- ‘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 ‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$, let {‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$} be the ‘m tuple {{{‘x_0, ‘x_1}, ..., ‘$x_{‘m-2}$}, ‘$x_{‘m-1}$}.

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<‘x>, 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<‘x> such that ‘x is a ‘dom(‘f) program.

‘ran(‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.Null.

‘ran(‘Func.Sm.Ext.zero) = ‘Func.Sm.Ext.Null.

‘ran(‘Func.Sm.Ext.one) = ‘Func.Sm.Ext.Nuro.

For a leaf small function extension ‘x, ‘ran(‘x) = ‘dom(‘x).

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

For a rule small function extension ‘r containing ‘model, ‘ran(‘r) = ‘des(‘model).

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

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

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

For a small function extension ‘f, the **field** of ‘f (a small sub-language of ‘Func.Sm.Ext), denoted by ‘field(‘f), is ‘union(‘dom(‘f), ‘ran(‘f)).

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

For 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.
- ‘f = ‘Func.Sm.Ext.one and ‘g = ‘Func.Sm.Ext.one.
- ‘f and ‘g are pair small function extensions, and ‘left(‘f) = ‘left(‘g) and ‘right(‘f) = ‘right(‘g).
- ‘f and ‘g are rule small function extensions, and ‘dom(‘f) = ‘dom(‘g), and, for each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘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]).

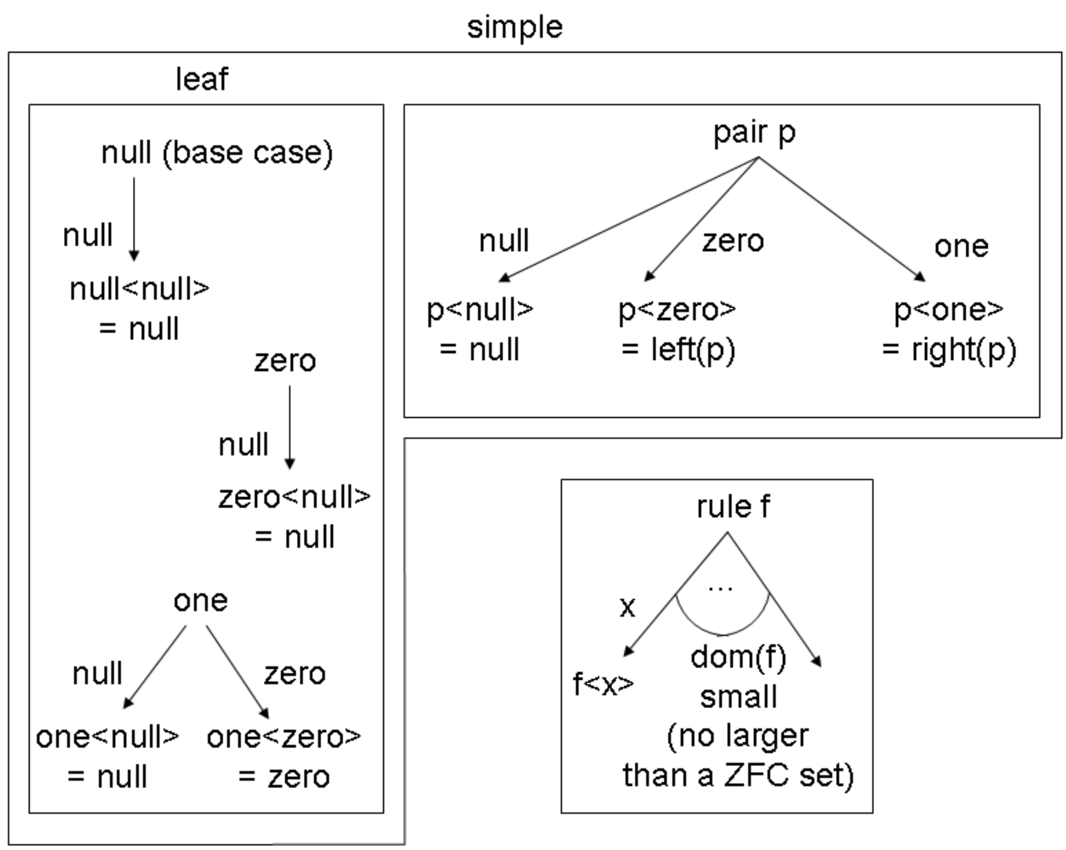

Figure 7.1: Small function extensions

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 ‘dom(‘f) = ‘dom(‘g): ‘domFuncExt(‘f) = ‘Func.Sm.Ext.identity(‘dom(‘f)). ‘domFuncExt(‘g) = ‘Func.Sm.Ext.identity(‘dom(‘g)).

- 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 ‘dom(‘f) = ‘dom(‘g).

Proof.

- Holds if ‘f = ‘g.
- If ‘dom(‘f) = ‘dom(‘g): For each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘x = ‘g<‘x>. □

For a *rule* small function extension ‘f, if ‘f is an identity, then ‘f = ‘domFuncExt(‘f).

Proof. ‘dom(‘f) = ‘dom(‘domFuncExt(‘f)). ‘domFuncExt(‘f) is a rule small function extension and an identity. □

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

Proof. ‘domFuncExt(‘f) is a rule small function extension and an identity. □

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 ‘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>.
- ‘domExt(‘F) = ‘domExt(‘G).

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:

- ‘Func.Sm.Ext.Null if ‘A = ‘Dom.Ext.Null
- ‘Func.Sm.Ext.Nuro if ‘A = ‘Dom.Ext.Nuro
- ‘Func.Sm.Ext.Leaf if ‘A = ‘Dom.Ext.Leaf
- ‘Func.Sm.Ext.Tree if ‘A = ‘Dom.Ext.Tree
- 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<‘x> = ‘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<‘x> = ‘Func.Sm.Ext.null is a ‘dom(‘F<‘x>) 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<‘x> 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). □

Let ‘Func.Sm.Ext.Pair.null be {‘Func.Sm.Ext.null, ‘Func.Sm.Ext.null}.

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(‘F<‘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).

For a *dependent sum* 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 *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 a *dependent sum* domain extension ‘A containing ‘F, and a ‘dom(‘F) program ‘l, ‘rank(‘F<‘l>) ≤ ‘rank(‘A).

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 ‘dom(‘FA) program iff there exists some small function extension ‘r such that {‘l, ‘r} is a ‘dom(‘A) program. ‘l is a ‘dom(‘FB) program iff there exists some small function extension ‘r such that {‘l, ‘r} is a ‘dom(‘B) program. ‘l is a ‘dom(‘FA) program iff ‘l is a ‘dom(‘FB) program.

- ‘dom(‘FA) = ‘dom(‘FB).

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

- For each ‘dom(‘FA) program ‘l, ‘dom(‘FA<‘l>) = ‘dom(‘FB<‘l>). □

For a *dependent product* domain extension ‘A containing ‘F, and a small function extension ‘x, ‘x is a ‘dom(‘F) 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.

Proof.

- If there exists some *rule* small function extension ‘f such that ‘f is a ‘dom(‘A) program and ‘x is a ‘dom(‘f) program: ‘dom(‘f) = ‘dom(‘F). ‘x is a ‘dom(‘F) program.

- If ‘x is a ‘dom(‘F) program: Let ‘f = ‘Func.Sm.Ext.Rule.null(‘dom(‘F)). ‘f is a rule small function extension and a ‘dom(‘A) program. ‘dom(‘f) = ‘dom(‘F). ‘x is a ‘dom(‘f) program. □

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

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

Proof.

- If there exists some *rule* small function extension ‘f such that ‘f is a ‘dom(‘A) program and ‘f<‘x> = ‘y: ‘y is a ‘dom(‘F<‘x>) program.

- If ‘y is a ‘dom(‘F<‘x>) program: Let ‘f be the rule small function extension such that ‘dom(‘f) = ‘dom(‘F) and, for each ‘dom(‘f) program ‘z, ‘f<‘z> = ‘y if ‘z = ‘x; and ‘Func.Sm.Ext.null otherwise. ‘f<‘x> = ‘y. ‘f is a ‘dom(‘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) 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 = ‘Func.Sm.Ext.zero

- ‘Dom.Ext.Nuro if ‘f = ‘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(‘f) 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, and ‘f = ‘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, ‘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 = Func.Sm.Ext.zero: ‘f<‘x> = ‘left(‘f) is a ‘dom(‘F) program. ‘dom(‘inferDomExt(‘A, ‘f, ‘x)) = ‘dom(‘domExt(‘F)) = ‘dom(‘F).

- If ‘A is a dependent sum domain extension containing ‘F, ‘f is a pair small function extension, and ‘x = Func.Sm.Ext.one: ‘f<‘x> = ‘right(‘f) is a ‘dom(‘F<‘left(‘f)>) program. ‘dom(‘inferDomExt(‘A, ‘f, ‘x)) = ‘dom(‘F<‘left(‘f)>).

- If ‘A is a dependent product domain extension, and ‘f = ‘Func.Sm.Ext.null: ‘f<‘x> = ‘Func.Sm.Ext.null. ‘dom(‘inferDomExt(‘A, ‘f, ‘x)) = ‘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 ‘dom(‘F<‘x>) program. ‘dom(‘inferDomExt(‘A, ‘f, ‘x)) = ‘dom(‘F<‘x>). □

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 ‘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 <‘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 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 ‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$, let {‘x_0, ‘x_1, ..., ‘$x_{‘m-2}$, ‘$x_{‘m-1}$} be the ‘m tuple {{{‘x_0, ‘x_1}, ..., ‘$x_{‘m-2}$}, ‘$x_{‘m-1}$}.

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

- {‘untag(‘left(‘f)), ‘untag(‘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 ‘dom(‘untagF) = ‘src(‘model) and, for each ‘dom(‘untagF) program ‘x, ‘untagF<‘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: ‘untag(‘f) = {‘untag(‘left(‘f)), ‘untag(‘right(‘f))}. ‘untag(‘g) = {‘untag(‘left(‘g)), ‘untag(‘right(‘g))}. ‘untag(‘left(‘f)) = ‘untag(‘left(‘g)). ‘untag(‘right(‘f)) = ‘untag(‘right(‘g)). ‘left(‘f) = ‘left(‘g) (by inductive hypothesis). ‘right(‘f) = ‘right(‘g) (by inductive hypothesis).

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

For a tagged small function extension ‘f, ‘f is a **tree** iff ‘untag(‘f) is a tree.

Let ‘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 ‘left(‘f) and ‘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: ‘untag(‘f) = ‘f is a tree.

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

- If ‘f is a rule tagged small function extension: ‘untag(‘f) is a rule small function extension. ‘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
- { ‘tagged(‘inferDomExt(‘A, ‘f, ‘Func.Sm.Ext.zero), ‘left(‘f)), ‘tagged(‘inferDomExt(‘A, ‘f, ‘Func.Sm.Ext.one), ‘right(‘f)) } if ‘f is a pair small function extension. Note that ‘left(‘f) is a ‘dom(‘inferDomExt(‘A, ‘f, ‘Func.Sm.Ext.zero)) program, and ‘right(‘f) is a ‘dom(‘inferDomExt(‘A, ‘f, ‘Func.Sm.Ext.one)) program.
- If ‘f is a rule small function extension: ‘tagged(‘A, ‘f) is the rule tagged small function extension containing <‘modelTagged, ‘tag> where:
  - ‘src(‘modelTagged) = ‘dom(‘f).
  - For each ‘src(‘modelTagged) program ‘x, ‘modelTagged(‘x) = ‘tagged(‘inferDomExt(‘A, ‘f, ‘x), ‘f<‘x>). Note that ‘f<‘x> is a ‘dom(‘inferDomExt(‘A, ‘f, ‘x)) program.
  - ‘tag = ‘inferDomExt(‘A, ‘f). Note that ‘dom(‘inferDomExt(‘A, ‘f)) = ‘dom(‘f) and ‘dom(‘tag) = ‘src(‘modelTagged).

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 leaf small function extension: ‘tagged(‘A, ‘f) = ‘f. ‘untag(‘f) = ‘f.
- If ‘f is a pair small function extension: ‘tagged(‘A, ‘f) = { ‘tagged(‘inferDomExt(‘A, ‘f, ‘Func.Sm.Ext.zero), ‘left(‘f)), ‘tagged(‘inferDomExt(‘A, ‘f, ‘Func.Sm.Ext.one), ‘right(‘f)) }. Let ‘taggedF = ‘tagged(‘A, ‘f). ‘untag(‘taggedF) = {‘untag(‘left(‘taggedF)), ‘untag(‘right(‘taggedF))}. Let ‘untagF = ‘untag(‘taggedF). ‘left(‘untagF) = ‘untag(‘left(‘taggedF)) = ‘left(‘f) (by inductive hypothesis). ‘right(‘untagF) = ‘untag(‘right(‘taggedF)) = ‘right(‘f) (by inductive hypothesis). ‘untagF = ‘f.

- If ‘f is a rule small function extension: ‘tagged(‘A, ‘f) is the rule tagged small function extension containing <‘modelTagged, ‘tag> where:
  - ‘src(‘modelTagged) = ‘dom(‘f).
  - For each ‘src(‘modelTagged) program ‘x, ‘modelTagged(‘x) = ‘tagged(‘inferDomExt(‘A, ‘f, ‘x), ‘f<‘x>).
  - ‘tag = ‘inferDomExt(‘A, ‘f).

  ‘untag(‘tagged(‘A, ‘f)) the rule small function extension ‘untagF such that ‘dom(‘untagF) = ‘src(‘modelTagged) and, for each ‘dom(‘untagF) program ‘x, ‘untagF<‘x> = ‘untag(‘modelTagged(‘x)). ‘dom(‘untagF) = ‘dom(‘f). For each ‘dom(‘f) program ‘x, ‘untagF<‘x> = ‘untag(‘tagged(‘inferDomExt(‘A, ‘f, ‘x), ‘f<‘x>)), and ‘untagF<‘x> = ‘f<‘x> (by inductive hypothesis). ‘untagF = ‘f. □

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

Proof. ‘untag(‘tagged(‘A, ‘untag(‘f))) = ‘untag(‘f). ‘tagged(‘A, ‘untag(‘f)) = ‘f (by tag irrelevance theorem). □

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

Proof. ‘untag(‘tagged(‘A, ‘f)) = ‘f = ‘untag(‘tagged(‘B, ‘f)). ‘tagged(‘A, ‘f) = ‘tagged(‘B, ‘f) (by tag irrelevance theorem). □

For a small function extension ‘f, ‘f is **taggable** iff there exists some tagged small function extension ‘g such that ‘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 ‘dom(‘A) program, then ‘f is taggable.

Proof. ‘untag(‘tagged(‘A, ‘f)) = ‘f. □

For a small function extension ‘f, if ‘f is untaggable, then, for each *valid* domain extension ‘A, ‘f is *not* a ‘dom(‘A) program.

For *valid* domain extensions ‘A and ‘B, ‘A = ‘B iff, for each tagged small function extension ‘f, ‘untag(‘f) is a ‘dom(‘A) program iff ‘untag(‘f) is a ‘dom(‘B) program.

Proof.

- Holds if ‘A = ‘B.
- If for each tagged small function extension ‘f, ‘untag(‘f) is a ‘dom(‘A) program iff ‘untag(‘f) is a ‘dom(‘B) program:
  - For each small function extension ‘g:
    - If ‘g is taggable: Let ‘f be a tagged small function extension such that ‘untag(‘f) = ‘g. ‘g is a ‘dom(‘A) program iff ‘g is a ‘dom(‘B) program.
    - If ‘g is untaggable: ‘g is neither a ‘dom(‘A) program nor a ‘dom(‘B) program.
  - ‘dom(‘A) = ‘dom(‘B). ‘A = ‘B (by domain extension irrelevance theorem). □

For a *valid* domain extension family ‘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 *valid* domain extension family ‘F, and a ‘dom(‘F) program ‘x, ‘untag(‘tagged(‘F, ‘x)) = ‘x.

Proof. ‘untag(‘tagged(‘F, ‘x)) = ‘untag(‘tagged(‘domExt(‘F), ‘x)). □

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

Proof. ‘dom(‘domExt(‘F)) = ‘dom(‘F). ‘tagged(‘F, ‘untag(‘x)) = ‘tagged(‘domExt(‘F), ‘untag(‘x)). □

For *valid* domain extension families ‘F and ‘G, and a small function extension ‘x such that ‘x is a ‘dom(‘F) program and a ‘dom(‘G) program, ‘tagged(‘F, ‘x) = ‘tagged(‘G, ‘x).

Proof. ‘tagged(‘F, ‘x) = ‘tagged(‘domExt(‘F), ‘x) = ‘tagged(‘domExt(‘G), ‘x) = ‘tagged(‘G, ‘x). □

7.12 DOMAIN, DOMAIN EXTENSION, SPECIFIC RESULT AND RANK OF A TAGGED SMALL FUNCTION EXTENSION

For a tagged small function extension ‘f, the **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. ‘untag(‘tagged(‘f, ‘x)) = ‘untag(‘tagged(‘domExt(‘f), ‘x)). □

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

Proof. ‘tagged(‘f, ‘untag(‘x)) = ‘tagged(‘domExt(‘f), ‘untag(‘x)). □

For tagged small function extensions ‘f and ‘g, and a small function extension ‘x such that ‘x is a ‘dom(‘f) program and a ‘dom(‘g) program, ‘tagged(‘f, ‘x) = ‘tagged(‘g, ‘x).

Proof. ‘tagged(‘f, ‘x) = ‘tagged(‘domExt(‘f), ‘x) = ‘tagged(‘domExt(‘g), ‘x) = ‘tagged(‘g, ‘x). □

For a tagged small function extension ‘f, ‘Func.Sm.Ext.null is a ‘dom(‘f) program, and ‘dom(‘f) is non-empty.

Proof. ‘Func.Sm.Ext.null is a ‘dom(‘domExt(‘f)) = ‘dom(‘f) program. □

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

Proof.

- ‘dom(‘f) = ‘dom(‘domExt(‘f)). ‘dom(‘g) = ‘dom(‘domExt(‘g)).
- 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 tagged small function extensions ‘f and ‘g, ‘domExt(‘f) = ‘domExt(‘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 tagged small function extensions ‘f and ‘g, ‘dom(‘f) = ‘dom(‘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 a tagged small function extension ‘f, and a ‘dom(‘f) program ‘x, the **specific result** of ‘f at ‘x, denoted by ‘f<‘x>, 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 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 rule tagged small function extension ‘r containing <‘model, ‘tag>, ‘ran(‘r) = ‘des(‘model).

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.

- ‘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).

Let ‘Func.Sm.Ext.Tagged.Null.set = ‘Func.Sm.Ext.Tagged.identity(‘Dom.Ext.Null).

Let ‘Func.Sm.Ext.Tagged.Nuro.set = ‘Func.Sm.Ext.Tagged.identity(‘Dom.Ext.Nuro).

Let ‘Func.Sm.Ext.Tagged.Leaf.set = ‘Func.Sm.Ext.Tagged.identity(‘Dom.Ext.Leaf).

Let ‘Func.Sm.Ext.Tagged.Tree.set = ‘Func.Sm.Ext.Tagged.identity(‘Dom.Ext.Tree).

For a *valid* domain extension ‘A, ‘domExt(‘Func.Sm.Ext.Tagged.identity(‘A)) = ‘A, and ‘dom(‘Func.Sm.Ext.Tagged.identity(‘A)) = ‘dom(‘A).

Proof. Let ‘f = ‘Func.Sm.Ext.Tagged.identity(‘A). ‘domExt(‘f) = ‘A. ‘dom(‘f) = ‘dom(‘domExt(‘f)) = ‘dom(‘A). □

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.Null.set if ‘f = ‘Func.Sm.Ext.null or ‘f = ‘Func.Sm.Ext.zero

- ‘Func.Sm.Ext.Tagged.Nuro.set if ‘f = ‘Func.Sm.Ext.one
- ‘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)).
‘domExt(‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f))) = ‘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 ‘domExt(‘f) = ‘domExt(‘g): ‘domFuncExt(‘f) = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f)). ‘domFuncExt(‘g) = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘g)).
- If ‘domFuncExt(‘f) = ‘domFuncExt(‘g): ‘domExt(‘domFuncExt(‘f)) = ‘domExt(‘f). ‘domExt(‘domFuncExt(‘g)) = ‘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 ‘dom(‘f) = ‘dom(‘g): For each ‘dom(‘f) program ‘x, ‘f<‘x> = ‘tagged(‘f, ‘x) = ‘tagged(‘g, ‘x) = ‘g<‘x>. □

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

For a *rule* tagged small function extension ‘f, if ‘f is an identity, then ‘domFuncExt(‘f) = ‘f.

Proof. ‘domExt(‘domFuncExt(‘f)) = ‘domExt(‘f). ‘domFuncExt(‘f) is a rule tagged small function extension and an identity. □

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

Proof. ‘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 ‘untag(‘f) is a ‘dom(‘A) program. If so, return ‘untag(‘f). If not, return ‘Func.Sm.Ext.null.
- If ‘A is a dependent sum domain extension, and ‘f is a pair tagged small function extension, coerce ‘left(‘f) first, then ‘right(‘f).
- If ‘A is a dependent sum domain extension, and ‘f is *not* a pair tagged small function extension, return ‘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 sum domain extension, 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).

For coercion pairs ‘p = <‘A, ‘f> and ‘q = <‘B, ‘g>, ‘p < ‘q iff <‘rank(‘A), ‘rank(‘f)> < <‘rank(‘B), ‘rank(‘g)>.

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

Proof.

- <‘domExt(‘F), ‘left(‘f)> < <‘A, ‘f>: ‘rank(‘domExt(‘F)) ≤ ‘rank(‘A). ‘rank(‘left(‘f)) < ‘rank(‘f). By case 2.

- <‘F<‘left(‘p)>, ‘right(‘f)> < <‘A, ‘f>: ‘rank(‘F<‘left(‘p)>) ≤ ‘rank(‘A). ‘rank(‘right(‘f)) < ‘rank(‘f). By case 2.

- <‘domExt(‘f), ‘tagged(‘F, ‘x)> < <‘A, ‘f>: ‘rank(‘domExt(‘f)) = ‘rank(‘dom(‘f)) ≤ ‘rank(‘f). ‘rank(‘tagged(‘F, ‘x)) = ‘rank(‘untag(‘tagged(‘F, ‘x))) = ‘rank(‘x) < ‘rank(‘r) < ‘rank(‘A). By case 4.

- < ‘F<‘x>, ‘f<‘coer(‘domExt(‘f), ‘tagged(‘F, ‘x))> > < <‘A, ‘f>: ‘rank(‘F<‘x>) ≤ ‘rank(‘A). ‘rank(‘f<‘coer(‘domExt(‘f), ‘tagged(‘F, ‘x))>) < ‘rank(‘f). By case 2. □

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. ‘left(‘p(‘m + 1)) ≤ ‘left(‘p(‘m)) and ‘right(‘p(‘m + 1)) < ‘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 ‘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. ‘left(‘p(‘m + 1)) < ‘left(‘p(‘m)) and ‘right(‘p(‘m + 1)) ≤ ‘right(‘p(‘m)).
  2. ‘left(‘p(‘m + 1)) ≤ ‘left(‘p(‘m)) and ‘right(‘p(‘m + 1)) < ‘right(‘p(‘m)).
  3. ‘left(‘p(‘m + 1)) < ‘right(‘p(‘m)) and ‘right(‘p(‘m + 1)) ≤ ‘left(‘p(‘m)).
  4. ‘left(‘p(‘m + 1)) ≤ ‘right(‘p(‘m)) and ‘right(‘p(‘m + 1)) < ‘left(‘p(‘m)).

- Let ‘twist be the model from ‘Nat such that, for each natural number ‘m, ‘twist(‘m) = 0 if case 1 or 2 holds; and 1 otherwise. Let ‘path be the model from ‘Nat such that ‘path(0) = 0 and, for each natural number ‘m, ‘path(m + 1) = ‘not(‘path(‘m)) if ‘twist(‘m); and ‘path(‘m) otherwise. Let ‘p0 be the model from ‘Nat such that, for each natural number ‘m, ‘p0(‘m) = ‘flip(‘p(‘m)) if ‘path(‘m); and ‘p(‘m) otherwise. For each natural number ‘m, ‘p0(‘m) is an ordinal pair and at least one of the following holds:

  1. ‘left(‘p0(‘m + 1)) < ‘left(‘p0(‘m)) and ‘right(‘p0(‘m + 1)) ≤ ‘right(‘p0(‘m)).
  2. ‘left(‘p0(‘m + 1)) ≤ ‘left(‘p0(‘m)) and ‘right(‘p0(‘m + 1)) < ‘right(‘p0(‘m)).

- For each natural number ‘m, ‘p0(‘m + 1) <s ‘p0(‘m). <s 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 ‘Nat such that, for each natural number ‘m, ‘p(‘m) is a coercion pair, ‘p(‘m + 1) < ‘p(‘m) and ‘ord(‘p(‘m + 1)) < ‘ord(‘p(‘m)). Let ‘p0 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.

Proof. ‘coer(‘F, ‘x) = ‘coer(‘domExt(‘F), ‘x). ‘dom(‘domExt(‘F)) = ‘dom(‘F). □

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.

Proof. ‘coer(‘f, ‘x) = ‘coer(‘domExt(‘f), ‘x). ‘dom(‘domExt(‘f)) = ‘dom(‘f). □

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.
- 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)). ‘untag(‘f) = {‘untag(‘left(‘f)), ‘untag(‘right(‘f))}. Let ‘untagF = ‘untag(‘f). ‘left(‘untagF) = ‘untag(‘left(‘f)) is a ‘dom(‘F) = ‘dom(‘domExt(‘F)) program and ‘coer(‘domExt(‘F), ‘left(‘f)) = ‘left(‘untagF) (by inductive hypothesis). ‘left(‘p) = ‘left(‘untagF). ‘right(‘untagF) = ‘untag(‘right(‘f)) is a ‘dom(‘F<‘left(‘untagF)>) =

‘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 sum domain extension, and ‘f is *not* a pair tagged small function extension: ‘coer(‘A, ‘f) = ‘Func.Sm.Ext.null. ‘untag(‘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))>). ‘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.

- 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. ‘untag(‘f) = ‘Func.Sm.Ext.null. □

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).

Proof. ‘dom(‘domExt(‘F)) = ‘dom(‘F). ‘coer(‘F, ‘x) = ‘coer(‘domExt(‘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. ‘dom(‘domExt(‘f)) = ‘dom(‘f). ‘coer(‘f, ‘x) = ‘coer(‘domExt(‘f), ‘x) = ‘untag(‘x). □

For a *valid* domain extension ‘A, ‘coer(‘A, ‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.null.

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

For a tagged small function extension ‘f, ‘coer(‘Dom.Ext.Null, ‘f) = ‘Func.Sm.Ext.null.

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

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

For a tagged small function extension ‘f, ‘coer(‘Dom.Ext.Tree, ‘f) = ‘untag(‘f) if ‘f is a tree; and ‘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<‘coer(‘f, ‘x)>.

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(‘tagged(‘F, ‘x))).

Proof. For each ‘dom(‘F) program ‘x, ‘f(‘tagged(‘F, ‘x)) = ‘f<‘coer(‘f, ‘tagged(‘F, ‘x))>. □

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

Proof. ‘f(‘x) = ‘f<‘coer(‘f, ‘x)> = ‘tagged(‘f, ‘coer(‘f, ‘x)). □

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

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

Proof. ‘Func.Sm.Ext.Tagged.identity(‘A) is an identity.
‘Func.Sm.Ext.Tagged.identity(‘A)(‘x) = ‘tagged(‘Func.Sm.Ext.Tagged.identity(‘A), ‘coer(‘Func.Sm.Ext.Tagged.identity(‘A), ‘x)). ‘domExt(‘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<‘x>.

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) = ‘domFuncExt(‘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.

Proof. ‘Func.Sm.Ext.null(‘x) = ‘Func.Sm.Ext.null<‘coer(‘Dom.Ext.Null, ‘x)> = ‘Func.Sm.Ext.null<‘Func.Sm.Ext.null> = ‘Func.Sm.Ext.null. □

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

Proof. ‘Func.Sm.Ext.zero(‘x) = ‘Func.Sm.Ext.zero<‘coer(‘Dom.Ext.Null, ‘x)> = ‘Func.Sm.Ext.zero<‘Func.Sm.Ext.null> = ‘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.

Proof. ‘Func.Sm.Ext.one(‘x) = ‘Func.Sm.Ext.one<‘coer(‘Dom.Ext.Nuro, ‘x)>. ‘Func.Sm.Ext.one(‘x) is ‘Func.Sm.Ext.one<‘x> if ‘x is a nuro; and ‘Func.Sm.Ext.one<‘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:

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

Proof.

- ‘f(‘x) = ‘f<‘coer(‘Dom.Ext.Leaf, ‘x)>.
- ‘f(‘x) is given by one of the following mutually exclusive cases:
  - ‘f<‘Func.Sm.Ext.zero> if ‘x = ‘Func.Sm.Ext.zero
  - ‘f<‘Func.Sm.Ext.one> if ‘x = ‘Func.Sm.Ext.one
  - ‘f<‘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.Null.set(‘x) = ‘Func.Sm.Ext.null.

Proof. ‘Func.Sm.Ext.Tagged.Null.set(‘x) = ‘tagged(‘Dom.Ext.Null, ‘coer(‘Dom.Ext.Null, ‘x)). □

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.

Proof. ‘Func.Sm.Ext.Tagged.Nuro.set(‘x) = ‘tagged(‘Dom.Ext.Nuro, ‘coer(‘Dom.Ext.Nuro, ‘x)). □

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>, ‘coer(‘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 ‘domExtFam(‘f), is the valid domain extension family ‘F such that ‘domExt(‘F) = ‘domExt(‘f) and, for each ‘dom(‘F) program ‘x, ‘F<‘x> = ‘domExt(‘f(‘tagged(‘F, ‘x))).

For a tagged small function extension ‘f, ‘domExt(‘domExtFam(‘f)) = ‘domExt(‘f), ‘dom(‘domExtFam(‘f)) = ‘dom(‘f) and, for each ‘dom(‘f) program ‘x, ‘domExtFam(‘f)<‘x> = ‘domExt(‘f(‘tagged(‘f, ‘x))) = ‘domExt(‘f<‘x>).

Proof. Let ‘F = ‘domExtFam(‘f). ‘domExt(‘F) = ‘domExt(‘f). ‘dom(‘F) = ‘dom(‘domExt(‘F)) = ‘dom(‘domExt(‘f)) = ‘dom(‘f). □

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

For a tagged small function extension ‘f, and a *pair* tagged small function extension ‘x, ‘sumDep(‘f)(‘x) is the pair tagged small function extension ‘p such that ‘left(‘p) = ‘domFuncExt(‘f)(‘left(‘x)) and ‘right(‘p) = ‘domFuncExt(‘f(‘left(‘p)))(‘right(‘x)).

Proof. ‘sumDep(‘f)(‘x) = ‘Func.Sm.Ext.Tagged.sum.dep(‘domExtFam(‘f))(‘x). ‘Func.Sm.Ext.Tagged.sum.dep(‘domExtFam(‘f))(‘x) is the pair tagged small function extension ‘p such that ‘left(‘p) = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘domExtFam(‘f)))(‘left(‘x)) and ‘right(‘p) = ‘Func.Sm.Ext.Tagged.identity (‘domExtFam(‘f)<‘untag(‘left(‘p))>)(‘right(‘x)). ‘left(‘p) = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f))(‘left(‘x)) = ‘domFuncExt(‘f)(‘left(‘x)). ‘domExtFam(‘f)<‘untag(‘left(‘p))> = ‘domExt(‘f(‘left(‘p))). ‘right(‘p) = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f(‘left(‘p))))(‘right(‘x)) = ‘domFuncExt(‘f(‘left(‘p)))(‘right(‘x)). □

For a tagged small function extension ‘f, and a *non-pair* tagged small function extension ‘x, ‘sumDep(‘f)(‘x) = ‘Func.Sm.Ext.null.

Proof. ‘sumDep(‘f)(‘x) = ‘Func.Sm.Ext.Tagged.sum.dep(‘domExtFam(‘f))(‘x). □

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

For a tagged small function extension ‘f, and a *rule* tagged small function extension ‘x, ‘prodDep(‘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> = ‘domFuncExt(‘f(‘tagged(‘r, ‘y)))(‘x(‘tagged(‘r, ‘y))).

Proof. ‘prodDep(‘f)(‘x) = ‘Func.Sm.Ext.Tagged.prod.dep(‘domExtFam(‘f))(‘x). ‘Func.Sm.Ext.Tagged.prod.dep(‘domExtFam(‘f))(‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘domExtFam(‘f)) and, for each ‘dom(‘r) program ‘y, ‘r<‘y> = ‘Func.Sm.Ext.Tagged.identity(‘domExtFam(‘f)<‘y>)(‘x(‘tagged(‘r, ‘y))). ‘domExt(‘domExtFam(‘f)) = ‘domExt(‘f). ‘domExt(‘r) = ‘domExt(‘f). For each ‘dom(‘r) program ‘y, ‘domExtFam(‘f)<‘y> = ‘domExt(‘f(‘tagged(‘f, ‘y))) = ‘domExt(‘f(‘tagged(‘r, ‘y))), ‘r<‘y> = ‘Func.Sm.Ext.Tagged.identity(‘domExt(‘f(‘tagged(‘r, ‘y))))(‘x(‘tagged(‘r, ‘y))) = ‘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, ‘prodDep(‘f)(‘x) = ‘Func.Sm.Ext.null.

Proof. ‘prodDep(‘f)(‘x) = ‘Func.Sm.Ext.Tagged.prod.dep(‘domExtFam(‘f))(‘x). □

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 ‘Func.Lg.Ext.Null.set = ‘Func.Lg.Ext.constant(‘Func.Sm.Ext.Tagged.Null.set).

‘res(‘Func.Lg.Ext.Null.set) = ‘Func.Sm.Ext.Tagged.Null.set.

Let ‘Func.Lg.Ext.Nuro.set = ‘Func.Lg.Ext.constant(‘Func.Sm.Ext.Tagged.Nuro.set).

‘res(‘Func.Lg.Ext.Nuro.set) = ‘Func.Sm.Ext.Tagged.Nuro.set.

Let ‘Func.Lg.Ext.Leaf.set = ‘Func.Lg.Ext.constant(‘Func.Sm.Ext.Tagged.Leaf.set).

‘res(‘Func.Lg.Ext.Leaf.set) = ‘Func.Sm.Ext.Tagged.Leaf.set.

Let ‘Func.Lg.Ext.Tree.set = ‘Func.Lg.Ext.constant(‘Func.Sm.Ext.Tagged.Tree.set).

‘res(‘Func.Lg.Ext.Tree.set) = ‘Func.Sm.Ext.Tagged.Tree.set.

‘Func.Lg.Ext.null, ‘Func.Lg.Ext.zero, ‘Func.Lg.Ext.one, ‘Func.Lg.Ext.Null.set, ‘Func.Lg.Ext.Nuro.set, ‘Func.Lg.Ext.Leaf.set and ‘Func.Lg.Ext.Tree.set are unchanging.

Let ‘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 ‘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 ‘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 ≥ 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) ... ‘$x‘_{m-2}$) ‘$x‘_{m-1}$).

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, ‘res({‘l ‘r}) = {‘res(‘l), ‘res(‘r)}.

Proof. {‘l ‘r}(‘Func.Sm.Ext.null) = {‘l(‘Func.Sm.Ext.null), ‘r(‘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)} = {‘res(‘l), ‘res(‘r)} = ‘res({‘l ‘r}). □

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

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, ..., ‘$x_{\text{‘m-1}}$, let ˜l{‘x_0 ‘x_1 ... ‘$x_{\text{‘m-1}}$} = {‘x_0 {‘x_1 ... {‘$x_{\text{‘m-1}}$ ‘Func.Lg.Ext.zero}}}.

˜l{} = ‘Func.Lg.Ext.zero, not ‘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 ≥ 2, and large function extensions ‘f and ‘x_0, ‘x_1, ..., ‘$x_{\text{‘m-1}}$, let [‘f ‘x_0 ‘x_1 ... ‘$x_{\text{‘m-1}}$] = [‘f {‘x_0 ‘x_1 ... ‘$x_{\text{‘m-1}}$}].

For a large function extension ‘family, the **dependent sum** of ‘family, denoted by ˜s.d[‘family], is the large function extension such that, for each tagged small function extension ‘x, ˜s.d[‘family](‘x) = ‘sumDep(‘family(‘x)).

For a large function extension ‘family, the **dependent product** of ‘family, denoted by ˜p.d[‘family], is the large function extension such that, for each tagged small function extension ‘x, ˜p.d[‘family](‘x) = ‘prodDep(‘family(‘x)).

For large function extensions ‘uncurry and ‘restrictor, the **Curry** of ‘uncurry to ‘restrictor, denoted by ˜c[‘uncurry ‘restrictor], is the large function extension such that, for each tagged small function extension ‘x, ˜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 ‘y, ‘r<‘y> = ‘uncurry({‘x, ‘tagged(‘r, ‘y)}).

The definition of Curry requires some explanation. For large function extensions ‘uncurry and ‘restrictor, and a small function extension ‘x, ˜c[‘uncurry ‘restrictor](‘x) is a rule tagged small function extension ‘r representing a partial call to ‘uncurry at ‘x. However, ‘r is restricted using the domain extension of ‘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(˜c[‘uncurry ‘restrictor](‘x)) = ‘domExt(‘restrictor(‘x)), ‘dom(˜c[‘uncurry ‘restrictor](‘x)) = ‘dom(‘restrictor(‘x)), and ‘domFuncExt(˜c[‘uncurry ‘restrictor](‘x)) = ‘domFuncExt(‘restrictor(‘x)).

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

Proof. ˜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)}). ˜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 ˜ite[‘ifP ‘thenP ‘elseP], is the large function extension such that, for each tagged small function extension ‘x, ˜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) program ‘x, ‘x is structurally smaller than ‘f. This fact permits a simple terminating recursion principle for NummSquared.

For large function extensions ‘start and ‘step, the **recursion** of ‘start and ‘step, denoted by ˜r[‘start ‘step], is the large function extension such that, for each tagged small function extension ‘x, ˜r[‘start ‘step](‘x) is defined by recursion on ‘untag(‘x):

- If ‘x = ‘Func.Sm.Ext.null: ˜r[‘start ‘step](‘x) = ‘start(‘x).
- If ‘x ≠ ‘Func.Sm.Ext.null: ˜r[‘start ‘step](‘x) = ‘step({‘rDom, ‘rRan, ‘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>

= ˜r[‘start ‘step](‘tagged(‘rDom, ‘y)). Note that ‘dom(‘rDom) = ‘dom(‘x) = ‘dom(‘untag(‘x)) and, for each ‘dom(‘rDom) program ‘y, ‘untag(‘tagged(‘rDom, ‘y)) = ‘y, and ‘y is structurally smaller than ‘untag(‘x).

– ‘rRan is the rule tagged small function extension such that ‘domExt(‘rRan) = ‘domExt(‘x) and, for each ‘dom(‘rRan) program ‘y, ‘rRan<‘y> = ˜r[‘start ‘step](‘x(‘tagged(‘rRan, ‘y))). Note that ‘dom(‘rRan) = ‘dom(‘x) and, for each ‘dom(‘rRan) program ‘y, ‘x(‘tagged(‘rRan, ‘y)) = ‘x(‘tagged(‘x, ‘y)) = ‘x<‘y>, and ‘untag(‘x<‘y>) is structurally smaller than ‘untag(‘x).

The above recursion principle requires some explanation. In the ‘x ≠ ‘Func.Sm.Ext.null case, ‘rDom and ‘rRan are the restrictions of ˜r[‘start ‘step] to ‘dom(‘x) and ‘ran(‘x), respectively.

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 ‘Func.Lg.Ext.eq be the large function extension such that, for each tagged small function extension ‘p, ‘Func.Lg.Ext.eq(‘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 ‘left(‘p) = ‘right(‘p)
- ‘Func.Sm.Ext.zero if ‘p is a pair tagged small function extension, and ‘left(‘p) ≠ ‘right(‘p)
- ‘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 ‘pred, the **Hilbert** of ‘pred, denoted by ˜h[‘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

˜s.d[‘family] be the dependent sum computational normalized combination containing ‘family.

A **dependent product computational normalized combination** contains ‘family where ‘family is a normalized large function. For a normalized large function ‘family, let ˜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 ˜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 ˜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 ˜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 ˜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 ‘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‘$_{m-1}$).

For a natural number ‘m $\geq$ 2, and normalized large functions ‘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‘$_{m-1}$}.

For a natural number ‘m, and normalized large functions ‘x$_0$, ‘x$_1$, ..., ‘x‘$_{m-1}$, 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 ‘x$_0$, ‘x$_1$, ..., ‘x‘$_{m-1}$, 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 ‘ext(‘c), is given by one of the following mutually exclusive cases:

- ‘Func.Lg.Ext.i if ‘c = ‘Constant.Norm.Compu.i
- ‘Func.Lg.Ext.null if ‘c = ‘Constant.Norm.Compu.null
- ‘Func.Lg.Ext.zero if ‘c = ‘Constant.Norm.Compu.zero
- ‘Func.Lg.Ext.one if ‘c = ‘Constant.Norm.Compu.one
- ‘Func.Lg.Ext.Null.set if ‘c = ‘Constant.Norm.Compu.Null.set
- ‘Func.Lg.Ext.Nuro.set if ‘c = ‘Constant.Norm.Compu.Nuro.set
- ‘Func.Lg.Ext.Leaf.set if ‘c = ‘Constant.Norm.Compu.Leaf.set
- ‘Func.Lg.Ext.Tree.set if ‘c = ‘Constant.Norm.Compu.Tree.set
- ‘Func.Lg.Ext.Null if ‘c = ‘Constant.Norm.Compu.Null
- ‘Func.Lg.Ext.Pair if ‘c = ‘Constant.Norm.Compu.Pair
- ‘Func.Lg.Ext.dom if ‘c = ‘Constant.Norm.Compu.dom
- ‘Func.Lg.Ext.eq if ‘c = ‘Constant.Norm.Noncompu.eq

For a normalized large function ‘f, the **extension** of ‘f (a large function extension), denoted by ‘ext(‘f), is defined by recursion on ‘f:

- as above if ‘f is a normalized constant
- [‘ext(‘outer) ‘ext(‘inner)], if ‘f = [‘outer ‘inner]
- (‘ext(‘called) ‘ext(‘arg)) if ‘f = (‘called ‘arg)
- {‘ext(‘left) ‘ext(‘right)} if ‘f = {‘left ‘right}
- ˜s.d[‘ext(‘family)] if ‘f = ˜s.d[‘family]
- ˜p.d[‘ext(‘family)] if ‘f = ˜p.d[‘family]

- ˜c[‘ext(‘uncurry) ‘ext(‘restrictor)] if ‘f = ˜c[‘uncurry ‘restrictor]
- ˜ite[‘ext(‘ifP) ‘ext(‘thenP) ‘ext(‘elseP)] if ‘f = ˜ite[‘ifP ‘thenP ‘elseP]
- ˜r[‘ext(‘start) ‘ext(‘step)] if ‘f = ˜r[‘start ‘step]
- ˜h[‘ext(‘pred)] if ‘f = ˜h[‘pred]

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

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 ≠ ‘fn with ‘ext(‘fn0) = ‘ext(‘fn).

Proof. Let ‘fn0 = [‘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 ‘ext(‘f)(‘x).

For a normalized large function ‘f, the **result** of ‘f, denoted by ‘res(‘f), is ‘res(‘ext(‘f)).

For a normalized large function ‘f, ‘f is **unchanging** iff ‘ext(‘f) is unchanging.

A **normalized proposition** is a normalized large function. For a normalized large function ‘f, ‘f is **true** iff ‘ext(‘f) is true.

8.3 REDUCTION: COMPUTED OF A NORMALIZED LARGE FUNCTION

For a normalized large function ‘f, the property of ‘f being **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 *not* deep computational.
- If ‘f = [‘outer ‘inner]: ‘f is deep computational iff ‘outer and ‘inner are deep computational.

- If ‘f = (‘called ‘arg): ‘f is deep computational iff ‘called and ‘arg are deep computational.
- If ‘f = {‘left ‘right}: ‘f is deep computational iff ‘left and ‘right are deep computational.
- If ‘f = ~s.d[‘family]: ‘f is deep computational iff ‘family is deep computational.
- If ‘f = ~p.d[‘family]: ‘f is deep computational iff ‘family is deep computational.
- If ‘f = ~c[‘uncurry ‘restrictor]: ‘f is deep computational iff ‘uncurry and ‘restrictor are deep computational.
- If ‘f = ~ite[‘ifP ‘thenP ‘elseP]: ‘f is deep computational iff ‘ifP, ‘thenP and ‘elseP are deep computational.
- If ‘f = ~r[‘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 = ‘Constant.Norm.Compu.null, ‘f = ‘Constant.Norm.Compu.zero or ‘f = ‘Constant.Norm.Compu.one: ‘f is a tree.
- 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.

Proof.

- By induction on ‘f.

- Holds if ‘f = ‘Constant.Norm.Compu.null, ‘f = ‘Constant.Norm.Compu.zero or ‘f = ‘Constant.Norm.Compu.one
- If ‘f = {‘left ‘right}: ‘left and ‘right are deep computational (by inductive hypothesis). □

For a *tree* normalized large function ‘f, ‘res(‘f) is given by one of the following mutually exclusive cases:

- ‘Func.Sm.Ext.null if ‘f = ‘Constant.Norm.Compu.null
- ‘Func.Sm.Ext.zero if ‘f = ‘Constant.Norm.Compu.zero
- ‘Func.Sm.Ext.one if ‘f = ‘Constant.Norm.Compu.one
- {‘res(‘left), ‘res(‘right)} if ‘f = {‘left ‘right}

Proof.

- Holds if ‘f = ‘Constant.Norm.Compu.null, ‘f = ‘Constant.Norm.Compu.zero or ‘f = ‘Constant.Norm.Compu.one
- If ‘f = {‘left ‘right}: ‘res({‘left ‘right}) = ‘res(‘ext({‘left ‘right})) = ‘res({‘ext(‘left) ‘ext(‘right)}) = {‘res(‘ext(‘left)), ‘res(‘ext(‘right))} = {‘res(‘left), ‘res(‘right)}. □

For a *tree* normalized large function ‘f, ‘res(‘f) is is a tree.

Proof.

- By induction on ‘f.
- Holds if ‘f = ‘Constant.Norm.Compu.null, ‘f = ‘Constant.Norm.Compu.zero or ‘f = ‘Constant.Norm.Compu.one
- If ‘f = {‘left ‘right}: ‘res(‘left) and ‘res(‘right) are trees (by inductive hypothesis). {‘res(‘left), ‘res(‘right)} is a tree. ‘res({‘left ‘right}) is a tree. □

For a *tree* normalized large function ‘f, ‘f is unchanging.

Proof.

- By induction on ‘f.

- Holds if ‘f = ‘Constant.Norm.Compu.null, ‘f = ‘Constant.Norm.Compu.zero or ‘f = ‘Constant.Norm.Compu.one
- 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 = ‘Func.Sm.Ext.null: ‘norm(‘Func.Sm.Ext.null) = ‘Constant.Norm.Compu.null. ‘res(‘Constant.Norm.Compu.null) = ‘Func.Sm.Ext.null.
- If ‘x = ‘Func.Sm.Ext.zero: ‘norm(‘Func.Sm.Ext.zero) = ‘Constant.Norm.Compu.zero. ‘res(‘Constant.Norm.Compu.zero) = ‘Func.Sm.Ext.zero.
- If ‘x = ‘Func.Sm.Ext.one: ‘norm(‘Func.Sm.Ext.one) = ‘Constant.Norm.Compu.one. ‘res(‘Constant.Norm.Compu.one) = ‘Func.Sm.Ext.one.
- 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.

- ‘res(‘f) is a tree. ‘resNorm(‘f) = ‘norm(‘res(‘f)).
- By induction on ‘f.
- If ‘f = ‘Constant.Norm.Compu.null: ‘res(‘Constant.Norm.Compu.null) = ‘Func.Sm.Ext.null. ‘norm(‘Func.Sm.Ext.null) = ‘Constant.Norm.Compu.null.
- If ‘f = ‘Constant.Norm.Compu.zero: ‘res(‘Constant.Norm.Compu.zero) = ‘Func.Sm.Ext.zero. ‘norm(‘Func.Sm.Ext.zero) = ‘Constant.Norm.Compu.zero.
- If ‘f = ‘Constant.Norm.Compu.one: ‘res(‘Constant.Norm.Compu.one) = ‘Func.Sm.Ext.one. ‘norm(‘Func.Sm.Ext.one) = ‘Constant.Norm.Compu.one.
- If ‘f = {‘left ‘right}: ‘left and ‘right are trees. ‘res(‘left) and ‘res(‘right) are trees. ‘resNorm(‘left) = ‘norm(‘res(‘left)) and ‘resNorm(‘right) = ‘norm(‘res(‘right)). ‘res(‘f) = {‘res(‘left), ‘res(‘right)}. ‘norm(‘res(‘f)) = {‘norm(‘left(‘res(‘f))) ‘norm(‘right(‘res(‘f)))} = {‘norm(‘res(‘left)) ‘norm(‘res(‘right))} = {‘left ‘right} (by induction hypothesis). □

For a normalized large function ‘f, the **computed** of ‘f, denoted by ‘computed(‘f), is ‘resNorm(‘f) if ‘f is deep computational; and ‘null otherwise.

For a normalized large function ‘f, ‘computed(‘f) is computable.

For a normalized large function ‘f, ‘computed(‘f) = ‘null iff ‘f is *not* deep computational, or ‘res(‘f) is *not* a tree.

For a *tree* normalized large function ‘f, ‘computed(‘f) = ‘f.

Proof. ‘f is deep computational. ‘computed(‘f) = ‘resNorm(‘f). □

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

In future, the definition of ‘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:

- ‘Constant.Norm.Compu.Null.set
- ‘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 ‘tag(‘f), is given by one of the following mutually exclusive cases:

- 0 if ‘f = ‘Constant.Norm.Compu.i
- 1 if ‘f = ‘Constant.Norm.Compu.null
- 2 if ‘f = ‘Constant.Norm.Compu.zero
- 3 if ‘f = ‘Constant.Norm.Compu.one
- 4 if ‘f = ‘Constant.Norm.Compu.Null.set
- 5 if ‘f = ‘Constant.Norm.Compu.Nuro.set
- 6 if ‘f = ‘Constant.Norm.Compu.Leaf.set
- 7 if ‘f = ‘Constant.Norm.Compu.Tree.set
- 8 if ‘f = ‘Constant.Norm.Compu.Null
- 9 if ‘f = ‘Constant.Norm.Compu.Pair
- 10 if ‘f = ‘Constant.Norm.Compu.dom
- 11 if ‘f = ‘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{‘quoted(‘family)}) if ‘f = ~p.d[‘family]
- ‘tree(‘tag(‘f), ~l{‘quoted(‘uncurry) ‘quoted(‘restrictor)}) if ‘f = ~c[‘uncurry ‘restrictor]
- ‘tree(‘tag(‘f), ~l{‘quoted(‘ifP) ‘quoted(‘thenP) ‘quoted(‘elseP)}) if ‘f = ~ite[‘ifP ‘thenP ‘elseP]
- ‘tree(‘tag(‘f), ~l{‘quoted(‘start) ‘quoted(‘step)}) if ‘f = ~r[‘start ‘step]
- ‘tree(‘tag(‘f), ~l{‘quoted(‘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 = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$> of ‘Func.Lg.Norm, the **quoted** of ‘l, denoted by ‘quoted(‘l), is ~l{‘quoted(‘x_0) ‘quoted(‘x_1) ... ‘quoted(‘$x_{‘m-1}$)}.

For a list ‘l of ‘Func.Lg.Norm, ‘quoted(‘l) is a tree.

For a normalized large function ‘f, and a list ‘l of ‘Func.Lg.Norm, the **macro preexpanded** of ‘f at ‘l, denoted by ‘macroPreexpanded(‘f, ‘l), is [‘f ‘quoted(‘l)].

For a normalized large function ‘f, and a list ‘l of ‘Func.Lg.Norm, ‘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, the **macro expanded** of ‘f at ‘l, denoted by ‘macroExpanded(‘f, ‘l), is ‘null if ‘computed(‘macroPreexpanded(‘f, ‘l)) = ‘null; and ‘unquoted(‘computed(‘macroPreexpanded(‘f, ‘l))) otherwise.

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

For a normalized large function ‘f, and a list ‘l of ‘Func.Lg.Norm, ‘macroExpanded(‘f, ‘l) = ‘null iff ‘computed(‘macroPreexpanded(‘f, ‘l)) = ‘null, or ‘unquoted(‘computed(‘macroPreexpanded(‘f, ‘l))) = ‘null.

For a normalized large function ‘f, and a list ‘l of ‘Func.Lg.Norm, ‘macroExpanded(‘f, ‘l) = ‘null iff ‘f is *not* deep computational, ‘res(‘macroPreexpanded(‘f, ‘l)) is *not* a tree, or ‘computed(‘macroPreexpanded(‘f, ‘l)) is *not* quoted.

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 ‘subst(‘f, ‘g, ‘x, ‘y), is defined by recursion on ‘f. For normalized large functions ‘f, ‘g, ‘x and ‘y, ‘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 = [‘outerF ‘innerF], ‘g = [‘outerG ‘innerG], ‘subst(‘outerF, ‘outerG, ‘x, ‘y), and ‘subst(‘innerF, ‘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<‘x_0, ‘x_1, ..., ‘$x_{‘m\text{-}1}$> is written in the concrete syntax as follows:

```
`start`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<‘x_0, ‘x_1, ..., ‘$x_{‘m\text{-}2}$, ‘$x_{‘m\text{-}1}$> is written in the concrete syntax as follows:

```
`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 `` ` `` (*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 `` ` `` (*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 null to zero computational non-normalized constant, ‘Constant.Nonnorm.Compu.Null.to.Zero
- the zero predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Zero
- the one predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.One
- the nuro predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Nuro
- the Boolean predicate computational non-normalized constant, ‘Constant.Nonnorm.Compu.Boo
- 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:

```
~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<‘x_0, ‘x_1, ..., ‘x‘$_{m-1}$> of ‘Func.Lg such that ‘m ≥ 1, let [‘outer ‘x_0 ‘x_1 ... ‘x‘$_{m-1}$] 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<‘x_0, ‘x_1, ..., ‘x‘$_{m-1}$> of ‘Func.Lg such that ‘m ≥ 2, let (‘x_0 ‘x_1 ... ‘x‘$_{m-1}$) be the small composition computational combination containing ‘calledAndArgs.

A **tuple computational combination** contains a list ‘components of ‘Func.Lg of length ≥ 2. For a list ‘components = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$> of ‘Func.Lg such that ‘m ≥ 2, let {‘x_0 ‘x_1 ... ‘$x_{‘m-1}$} be the tuple computational combination containing ‘components.

A **list computational combination** contains a list ‘elements of ‘Func.Lg. For a list ‘elements = l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$> of ‘Func.Lg, let ˜l{‘x_0 ‘x_1 ... ‘$x_{‘m-1}$} 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 ~c.aug[‘uncurry ‘restrictor ‘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 ~c.res[‘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 ~r.dom[‘start ‘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 ~r.ran[‘start ‘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 ~r.step[‘start ‘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 ~r.rhs[‘start ‘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:

```
%‘id
```

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<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$> of ‘Func.Lg, let #‘called[‘x_0 ‘x_1 ... ‘$x_{‘m-1}$] 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<‘id_0, ‘id_1, ..., ‘$id_{‘m-1}$> is written in the concrete syntax as follows:

```
{%‘idm-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<‘def$_0$, ‘def$_1$, ..., ‘def‘$_{m-1}$> 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:

```
`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 l<‘modu$_0$, ‘modu$_1$, ..., ‘modu$_{\text{‘m-1}}$> is written in the concrete syntax as follows:

```
`modum-1
.
.
.
`modu1
`modu0
```

There is a *reversal* between the abstract syntax and the concrete syntax.

For an abstract program ‘prog containing l<‘modu$_0$, ‘modu$_1$, ..., ‘modu$_{\text{‘m-1}}$>, the **module name list** of ‘prog, denoted by ‘moduNameLis(‘prog), is l<‘name(‘modu$_0$), ‘name(‘modu$_1$), ..., ‘name(‘modu$_{\text{‘m-1}}$)>.

For an abstract program ‘prog containing l<‘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 l<‘x_0, ‘x_1, ..., ‘$x_{‘m-1}$>, the **normal form** of ‘primStr, denoted by ‘norm(‘primStr), is ~l{‘norm(‘x_0) ‘norm(‘x_1) ... ‘norm(‘$x_{‘m-1}$)}.

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 NummSquared 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 NummSquared 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.

```
~right =
~ite[
    ~Pair
    (~i  1)
    ~null
];
```

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.

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

For a computational non-normalized constant ‘f, the **normal form** of ‘f, denoted by ‘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 *normalized* large functions is defined.

Corresponding combinations of normalized large functions have already been defined for 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

`~left` and `~right` have already been defined.

8.20.1 CONFIRMATION WITH NULL

```
~conf.n =
~ite[
    ~i
    1
    0
];
```

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.

For a tagged small function extension ‘x, ‘Func.Lg.Ext.conf.n(‘x) = ‘Func.Sm.Ext.Tagged.Leaf.set(‘x).

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
];
```

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Null.to.Zero(‘x) = ‘Func.Sm.Ext.zero if ‘x = ‘Func.Sm.Ext.null; and ‘x otherwise.

8.20.4 KIND PREDICATES

```
~Zero = [~Null.to.Zero  ~not.n];
```

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Zero(‘x) = ‘Func.Sm.Ext.one if ‘x = ‘Func.Sm.Ext.zero; and ‘Func.Sm.Ext.zero otherwise.

```
~One = [~Null.to.Zero  ~conf.n];
```

For a tagged small function extension ‘x, ‘Func.Lg.Ext.One(‘x) = ‘Func.Sm.Ext.one if ‘x = ‘Func.Sm.Ext.one; and ‘Func.Sm.Ext.zero otherwise.

For a tagged small function extension ‘x, ‘Func.Lg.Ext.One(‘x) is a Boolean, and ‘Func.Lg.Ext.One(‘x) is true iff ‘x is true.

For a tagged small function extension ‘x, ‘Func.Lg.Ext.One(‘x) = ‘Func.Lg.Ext.conf.n(‘Func.Sm.Ext.one) if ‘x = ‘Func.Sm.Ext.one; and ‘Func.Lg.Ext.conf.n(‘Func.Sm.Ext.zero) otherwise.

```
~Nuro =
~ite[
    ~Null
    1
    ~Zero
];
```

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

```
~Boo =
~ite[
    ~Zero
    1
    ~One
];
```

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Boo(‘x) = ‘Func.Sm.Ext.one if ‘x is a Boolean; and ‘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: ‘Func.Lg.Ext.Tree(‘x) = ‘Func.Sm.Ext.zero.

Proof.

- If ‘x = ‘Func.Sm.Ext.null: ‘Func.Lg.Ext.Tree(‘x) = ‘Func.Sm.Ext.one.
- If ‘x ≠ ‘Func.Sm.Ext.null: ‘Func.Lg.Ext.Tree(‘x) = ‘Func.Lg.Ext.Tree.step({‘rDom, ‘rRan, ‘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> = ‘Func.Lg.Ext.Tree(‘tagged(‘rDom, ‘y)).
  - ‘rRan is the rule tagged small function extension such that ‘domExt(‘rRan) = ‘domExt(‘x) and, for each ‘dom(‘rRan) program ‘y, ‘rRan<‘y> = ‘Func.Lg.Ext.Tree(‘x(‘tagged(‘rRan, ‘y))). □

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Tree(‘x) = ‘Func.Sm.Ext.one if ‘x is a tree; and ‘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: ‘Func.Lg.Ext.Tree(‘left(‘x)) = ‘Func.Sm.Ext.one if ‘left(‘x) is a tree; and ‘Func.Sm.Ext.zero otherwise (by inductive hypothesis). ‘Func.Lg.Ext.Tree(‘right(‘x)) = ‘Func.Sm.Ext.one if ‘right(‘x) is a tree; and ‘Func.Sm.Ext.zero otherwise (by inductive hypothesis). ‘Func.Lg.Ext.Tree(‘x) is ‘Func.Sm.Ext.one if ‘left(‘x) and ‘right(‘x) are trees; and ‘Func.Sm.Ext.zero otherwise.
- Holds if ‘x is a rule tagged small function extension. □

8.20.6 RESULT

```
~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:

```
~restrict['unrestrict] = ~c[['unrestrict  ~right]  ~i];
```

For a definition in pseudo-NummSquared that is parameterized by a normalized large function (for example, `~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, ˜restrict[‘unrestrict] parameterized by the large function extension ‘unrestrict).

For a large function extension ‘unrestrict, and a tagged small function extension ‘x, ˜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. ˜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:

```
~restrict.ran['unrestrict] = ~c[['unrestrict  ~res]  ~i];
```

For a large function extension ‘unrestrict, and a tagged small function extension ‘x, ˜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. ~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 ‘Func.Lg.Ext.res]({‘x, ‘tagged(‘r, ‘y)}) = ‘unrestrict(‘x(‘tagged(‘r, ‘y))). □

8.20.9 NURO SET RESULT

```
~Nuro.set.res =
~ite[
    ~i
    ~null
    0
];
```

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.Tagged.Nuro.set(‘x).

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

```
~Tree.set.res =
~ite[
    ~Tree
    ~i
    ~null
];
```

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.

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Tree.set.res(‘x) = ‘Func.Sm.Ext.Tagged.Tree.set(‘x).

8.20.11 DEPENDENT SUM RESULT

```
~s.d.res.left {%family %pair} =
~ite[
    [~Pair  %pair]
    ([~dom  %family]  [~left  %pair])
    ~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.

```
~s.d.res.right {%family %pair} =
~ite[
    [~Pair  %pair]
    ( [~dom  (%family  ~s.d.res.left)]  [~right  %pair] )
    ~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.

```
~s.d.res {%family %pair} =
~ite[
    [~Pair  %pair]
    {~s.d.res.left  ~s.d.res.right}
    ~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) = ‘domFuncExt(‘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) );
```

For a tagged small function extension ‘x = {‘family, ‘rule, ‘arg}, ‘Func.Lg.Ext.p.d.res.rule.uncurry(‘x) = ‘domFuncExt(‘family(‘arg))(‘rule(‘arg)).

```
~p.d.res.rule {%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 = {‘family, ‘rule}, ‘Func.Lg.Ext.p.d.res(‘x) is ‘Func.Lg.Ext.p.d.res.rule(‘x) if ‘rule is a rule tagged small function extension; and ‘Func.Sm.Ext.null otherwise.

For a tagged small function extension ‘x = {‘family, ‘rule}, ‘Func.Lg.Ext.p.d.res(‘x) = ‘prodDep(‘family)(‘rule).

Proof.

- If ‘rule is a rule tagged small function extension: ‘Func.Lg.Ext.p.d.res(‘x) = ‘Func.Lg.Ext.p.d.res.rule(‘x).

- If ‘rule is *not* a rule tagged small function extension: ‘Func.Lg.Ext.p.d.res(‘x) = ‘Func.Sm.Ext.null. □

8.20.13 CURRY AUGMENTED

For normalized large functions ‘uncurry and ‘augmentor:

```
~c.aug.uncurry[`uncurry `augmentor] {%x %y} =
[`uncurry [`augmentor %x] %y];
```

For large function extensions ‘uncurry and ‘augmentor, and a tagged small function extension ‘z = {‘x, ‘y}, ˜c.aug.uncurry[‘uncurry ‘augmentor](‘z) = ‘uncurry({‘augmentor(‘x), ‘y}).

For normalized large functions ‘uncurry, ‘restrictor and ‘augmentor:

```
~c.aug[`uncurry `restrictor `augmentor] =
~c[
    ~c.aug.uncurry[`uncurry `augmentor]
    [`restrictor `augmentor]
];
```

For large function extensions ‘uncurry, ‘restrictor and ‘augmentor, and a tagged small function extension ‘x, ˜c.aug[‘uncurry ‘restrictor ‘augmentor](‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘restrictor(‘augmentor(‘x))) and, for each ‘dom(‘r) program ‘y, ‘r<‘y> = ˜c.aug.uncurry[‘uncurry ‘augmentor]({‘x, ‘tagged(‘r, ‘y)}) = ‘uncurry({‘augmentor(‘x), ‘tagged(‘r, ‘y)}).

For large function extensions ‘uncurry, ‘restrictor and ‘augmentor, and a tagged small function extension ‘x, ˜c.aug[‘uncurry ‘restrictor ‘augmentor](‘x) = [˜c[‘uncurry ‘restrictor] ‘augmentor](‘x).

Proof. [˜c[‘uncurry ‘restrictor] ‘augmentor](‘x) = ˜c[‘uncurry ‘restrictor](‘augmentor(‘x)) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘restrictor(‘augmentor(‘x))) and, for each ‘dom(‘r) program program ‘y, ‘r<‘y> = ‘uncurry({‘augmentor(‘x), ‘tagged(‘r, ‘y)}). □

8.20.14 CURRY RESULT

For a normalized large function ‘uncurry:

```
~c.res[`uncurry] {%x %restrictor %y} =
[ `uncurry  %x  ([~dom  %restrictor]  %y) ];
```

For a large function extension ‘uncurry, and a tagged small function extension ‘z = {‘x, ‘restrictor, ‘y}, ˜c.res[‘uncurry](‘z) = ‘uncurry({‘x, ‘domFuncExt(‘restrictor)(‘y)}).

8.20.15 RECURSION RIGHT-HAND-SIDE

For normalized large functions ‘start and ‘step:

```
~r.dom[`start  `step] = ~restrict[~r[`start  `step]];
```

For large function extensions ‘start and ‘step, and a tagged small function extension ‘x, ˜r.dom[‘start ‘step](‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘x) and, for each ‘dom(‘r) program program ‘y, ‘r<‘y> = ˜r[‘start ‘step](‘tagged(‘r, ‘y)).

For normalized large functions ‘start and ‘step:

```
~r.ran[`start  `step] = ~restrict.ran[~r[`start  `step]];
```

For large function extensions ‘start and ‘step, and a tagged small function extension ‘x, ˜r.ran[‘start ‘step](‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘x) and, for each ‘dom(‘r) program ‘y, ‘r<‘y> = ˜r[‘start ‘step](‘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]
```

```
    1
];
```

For a tagged small function extension ‘x = {‘b, ‘c}, ‘Func.Lg.Ext.imp.n(‘x) is given by one of the following mutually exclusive cases:

- ‘Func.Sm.Ex.one if ‘b = ‘Func.Sm.Ext.zero
- ‘Func.Lg.Ext.conf.n(‘c) if ‘b = ‘Func.Sm.Ext.one
- ‘Func.Sm.Ext.null if ‘b is not Boolean

8.20.18 IMPLICATION

```
~imp {%b %c} = [~imp.n  [~One  %b]  [~One  %c]];
```

For a tagged small function extension ‘x = {‘b, ‘c}, ‘Func.Lg.Ext.imp(‘x) = ‘Func.Lg.Ext.imp.n({‘Func.Lg.Ext.One(‘b), ‘Func.Lg.Ext.One(‘c)}).

For a tagged small function extension ‘x = {‘b, ‘c}, ‘Func.Lg.Ext.imp(‘x) is ‘Func.Lg.Ext.One(‘c) if ‘b is true; and ‘Func.Sm.Ext.one otherwise.

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 ‘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:

```
~exist[`pred] = [ ~One  [`pred  ~i  ~h[`pred]] ];
```

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜exist[‘pred](‘x) = ‘Func.Lg.Ext.One(‘pred({‘x, ˜h[‘pred](‘x)})).

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜exist[‘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.

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. ˜exist[‘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)}) ≠ ‘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:

```
~not.all[`pred] = ~exist[[~not  `pred]];
```

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜not.all[‘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.

Proof. ˜not.all[‘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. ˜not.all[‘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:

```
~all[`pred] = [~not  ~not.all[`pred]];
```

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜all[‘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:

```
~all.una[`pred] = ~all[[`pred  ~right]];
```

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜all.una[‘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, ˜all.una[‘pred](‘x) is ‘Func.Sm.Ext.one if ‘pred is true; and ‘Func.Sm.Ext.zero otherwise.

For a large function extension ‘pred, ˜all.una[‘pred] is unchanging.

8.21.4 SMALL UNIVERSAL QUANTIFICATION

```
~all.sm = ~all[~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

```
~=.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.

```
~=.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.

```
~=.res {%x %y} = ~all[~=.res.at];
```

For a tagged small function extension ‘z = {‘x, ‘y}, ‘Func.Lg.Ext.eq.res(‘z) is ‘Func.Sm.Ext.one if, for each tagged small function extension ‘w, ‘Func.Lg.Ext.eq.res.at({‘x, ‘y, ‘w}) is true; and ‘Func.Sm.Ext.zero otherwise.

For a tagged small function extension ‘z = {‘x, ‘y}, ‘Func.Lg.Ext.eq.res(‘z) is ‘Func.Sm.Ext.one if, for each tagged small function extension ‘w, ‘x(‘w) = ‘y(‘w); and ‘Func.Sm.Ext.zero otherwise.

```
~=.dom.res {%x %y} = [~=.res  [~dom  %x]  [~dom  %y]];
```

For a tagged small function extension ‘z = {‘x, ‘y}, ‘Func.Lg.Ext.eq.dom.res(‘z) is ‘Func.Sm.Ext.one if, for each tagged small function extension ‘w, ‘domFuncExt(‘x)(‘w) = ‘domFuncExt(‘y)(‘w); and ‘Func.Sm.Ext.zero otherwise.

```
~=.both.res {%x %y} =
~ite[
    ~=.dom.res
    ~=.res
    0
];
```

For a tagged small function extension ‘z = {‘x, ‘y}, ‘Func.Lg.Ext.eq.both.res(‘z) is ‘Func.Sm.Ext.one if, for each tagged small function extension ‘w, ‘domFuncExt(‘x)(‘w) = ‘domFuncExt(‘y)(‘w) and ‘x(‘w) = ‘y(‘w); and ‘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 ‘Func.Lg.Ext.eq.both.res(‘z) is ‘Func.Sm.Ext.one if ‘x = ‘y; and ‘Func.Sm.Ext.zero otherwise.

```
~=.rhs {%x %y} =
~ite[
    [~Null  %x]
    [~Null  %y]
~ite[
    [~Zero  %x]
    [~Zero  %y]
~ite[
    [~One  %x]
    [~One  %y]
```

```
~ite[
    [~Pair  %x]
    ~ite[
        [~Pair  %y]
        ~=.pair
        0
    ]
~ite[
    [~Rule  %y]
    ~=.both.res
    0
]
]]]];
```

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 ‘left(‘p) = ‘right(‘p)
- ‘Func.Sm.Ext.zero if ‘p is a pair tagged small function extension, and ‘left(‘p) ≠ ‘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

```
~not.= = {%x %y} [~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 ≠ ‘Func.Sm.Ext.null, and a ‘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 ‘pred:

```
~induc.hyp.dom[‘pred] = [~all.sm  ~restrict[‘pred]];
```

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜induc.hyp.dom[‘pred](‘x) = ‘Func.Lg.Ext.all.sm(˜restrict[‘pred](‘x)).

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜induc.hyp.dom[‘pred](‘x) is ‘Func.Sm.Ext.one if, for each ‘dom(‘x) program ‘y, ‘pred(‘tagged(‘x, ‘y)) is true; and ‘Func.Sm.Ext.zero otherwise.

Proof. ˜induc.hyp.dom[‘pred](‘x) is ‘Func.Sm.Ext.one if, for each ‘dom(˜restrict[‘pred](‘x)) program ‘y, ˜restrict[‘pred](‘x)<‘y> is true; and ‘Func.Sm.Ext.zero otherwise. ˜restrict[‘pred](‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘x) and, for each ‘dom(‘r) program ‘y, ‘r<‘y> = ‘pred(‘tagged(‘r, ‘y)). □

For a normalized large function ‘pred:

```
~induc.hyp.ran[‘pred] = [~all.sm  ~restrict.ran[‘pred]];
```

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜induc.hyp.ran[‘pred](‘x) = ‘Func.Lg.Ext.all.sm(˜restrict.ran[‘pred](‘x)).

For a large function extension ‘pred, and a tagged small function extension ‘x, ˜induc.hyp.ran[‘pred](‘x) is ‘Func.Sm.Ext.one if, for each ‘dom(‘x) program ‘y, ‘pred(‘x(‘tagged(‘x, ‘y))) = ‘pred(‘x<‘y>) is true; and ‘Func.Sm.Ext.zero otherwise.

Proof. ˜induc.hyp.ran[‘pred](‘x) is ‘Func.Sm.Ext.one if, for each ‘dom(˜restrict.ran[‘pred](‘x)) program ‘y, ˜restrict.ran[‘pred](‘x)<‘y> is true; and ‘Func.Sm.Ext.zero otherwise. ˜restrict.ran[‘pred](‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘x) and, for each ‘dom(‘r) program ‘y, ‘r<‘y> = ‘pred(‘x(‘tagged(‘r, ‘y))). □

For a normalized large function ‘pred:

```
~induc.case.at[‘pred] =
[~imp
    ~induc.hyp.dom[‘pred]
```

```
[~imp
    ~induc.hyp.ran['pred]
    'pred
]];
```

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 ‘x_0 ‘x_1 ... ‘$x_{‘m-1}$] and at least one of ‘norm(‘cg, ‘cl, ‘outer), ‘norm(‘cg, ‘cl, ‘x_0), ‘norm(‘cg, ‘cl, ‘x_1), ..., ‘norm(‘cg, ‘cl, ‘$x_{‘m-1}$) is ‘null
- [‘norm(‘cg, ‘cl, ‘outer) ‘norm(‘cg, ‘cl, ‘x_0) ‘norm(‘cg, ‘cl, ‘x_1) ... ‘norm(‘cg, ‘cl, ‘$x_{‘m-1}$)] if ‘f = [‘outer ‘x_0 ‘x_1 ... ‘$x_{‘m-1}$] and all of ‘norm(‘cg, ‘cl, ‘outer), ‘norm(‘cg, ‘cl, ‘x_0), ‘norm(‘cg, ‘cl, ‘x_1), ..., ‘norm(‘cg, ‘cl, ‘$x_{‘m-1}$) 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[‘x_0 ‘x_1 ... ‘x‘$_{m-1}$] and at least one of ‘norm(‘cg, ‘cl, ‘called), ‘norm(‘cg, ‘cl, ‘x_0), ‘norm(‘cg, ‘cl, ‘x_1), ..., ‘norm(‘cg, ‘cl, ‘x‘$_{m-1}$) is ‘null
- ‘macroExpanded(‘norm(‘cg, ‘cl, ‘called), l<‘norm(‘cg, ‘cl, ‘x_0), ‘norm(‘cg, ‘cl, ‘x_1), ..., ‘norm(‘cg, ‘cl, ‘x‘$_{m-1}$)>) if ‘f = #‘called[‘x_0 ‘x_1 ... ‘x‘$_{m-1}$] and all of ‘norm(‘cg, ‘cl, ‘called), ‘norm(‘cg, ‘cl, ‘x_0), ‘norm(‘cg, ‘cl, ‘x_1), ..., ‘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 ‘norm(‘cg, ‘checker) ≠ ‘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 ‘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 ‘norm(‘cg, ‘desc) ≠ ‘null.

For a global context ‘cg, and a definition ‘def containing <‘comment, ‘name, ‘accessTupleLocDesc, ‘rhs>, the **normal form** in ‘cg of ‘def, denoted by ‘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 ‘norm(‘cg, ‘def) ≠ ‘null.

For a definition list ‘dl containing ‘l, the **normal form** of ‘dl (a valid global context or ‘null), denoted by ‘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 = ‘norm(‘dlR). ‘norm(‘dl) is given by one of the following mutually exclusive cases:
  - ‘null if ‘cgR = ‘null
  - If ‘cgR ≠ 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 ‘norm(‘dl) ≠ ‘null.

For a program ‘prog, the **normal form** of ‘prog, denoted by ‘norm(‘prog), is ‘norm(‘defLis(‘prog)).

A program ‘prog is **valid** iff ‘moduNameLis(‘prog) is valid and ‘norm(‘prog) ≠ ‘null.

8.24 PSEUDO-NUMMSQUARED COMPLETE

At this point, normal forms have been completely defined. Therefore, pseudo-NummSquared can include the full NummSquared 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] = [~=  [~i  '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  ~i]  '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.null.co.lg[‘x] = [~=  [~null  ‘x]  ~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.null.Null = [~=  [~Null  ~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)) = ‘Func.Lg.Ext.Null(‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.one = ‘Func.Lg.Ext.one(‘x). □

Null pair predicate axiom:

```
ax.null.Pair = [~=  [~Pair  ~null]  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)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Null domain axiom:

```
ax.null.dom = [~=  [~dom  ~null]  ~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)) = ‘domFuncExt(‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.Tagged.Null.set = ‘Func.Lg.Ext.Null.set(‘x). □

Null small composition axiom: For a normalized large function ‘x:

```
ax.null.co.sm[‘x] = [~=  (~null  ‘x)  ~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.Sm.Ext.null. ˜ite[‘Func.Lg.Ext.null ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘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 ‘Func.Lg.Ext.zero](‘x) = ‘Func.Lg.Ext.Null(‘Func.Lg.Ext.zero(‘x)) = ‘Func.Lg.Ext.Null(‘Func.Sm.Ext.zero) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

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 ‘Func.Lg.Ext.zero](‘x) = ‘Func.Lg.Ext.Pair(‘Func.Lg.Ext.zero(‘x)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.zero) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Zero domain axiom:

```
ax.zero.dom = [~=  [~dom  0]  ~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) = ‘domFuncExt(‘Func.Lg.Ext.zero(‘x)) = ‘domFuncExt(‘Func.Sm.Ext.zero) = ‘Func.Sm.Ext.Tagged.Null.set = ‘Func.Lg.Ext.Null.set(‘x). □

Zero small composition axiom: For a normalized large function ‘x:

```
ax.zero.co.sm[‘x] = [~=  (0  ‘x)  ~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.zero(‘y)(‘x(‘y)) = ‘Func.Sm.Ext.zero(‘x(‘y)) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘y). □

Zero if-then-else axiom: For normalized large functions ‘t and ‘e:

```
ax.zero.ite[‘t ‘e] = [~=  ~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) = ‘Func.Sm.Ext.zero. ˜ite[‘Func.Lg.Ext.zero ‘t ‘e](‘x) = ‘e(‘x). □

8.25.4 ONE

One large composition axiom: For a normalized large function ‘x:

```
ax.one.co.lg[‘x] = [~=  [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. □

One null predicate axiom:

```
ax.one.Null = [~=  [~Null  1]  0];
```

‘Func.Lg.Ext.ax.one.Null is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ‘Func.Lg.Ext.one](‘x) = ‘Func.Lg.Ext.Null(‘Func.Lg.Ext.one(‘x)) = ‘Func.Lg.Ext.Null(‘Func.Sm.Ext.one) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

One pair predicate axiom:

```
ax.one.Pair = [~=  [~Pair  1]  0];
```

‘Func.Lg.Ext.ax.one.Pair is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ‘Func.Lg.Ext.one](‘x) = ‘Func.Lg.Ext.Pair(‘Func.Lg.Ext.one(‘x)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.one) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

One domain axiom:

```
ax.one.dom = [~=  [~dom  1]  ~Nuro.set];
```

‘Func.Lg.Ext.ax.one.dom is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.dom ‘Func.Lg.Ext.one](‘x) = ‘domFuncExt(‘Func.Lg.Ext.one(‘x)) = ‘domFuncExt(‘Func.Sm.Ext.one) = ‘Func.Sm.Ext.Tagged.Nuro.set = ‘Func.Lg.Ext.Nuro.set(‘x). □

One small composition axiom: For a normalized large function ‘x:

```
ax.one.co.sm[`x] = [~=  (1  `x)  [~Nuro.set.res  `x]];
```

For a large function extension ‘x, ‘Func.Lg.Ext.ax.one.co.sm[‘x] is true.

Proof. For each tagged small function extension ‘y: (‘Func.Lg.Ext.one ‘x)(‘y) = ‘Func.Lg.Ext.one(‘y)(‘x(‘y)) = ‘Func.Sm.Ext.one(‘x(‘y)) = ‘Func.Lg.Ext.Nuro.set.res(‘x(‘y)) = [‘Func.Lg.Ext.Nuro.set.res ‘x](‘y). □

One if-then-else axiom: For normalized large functions ‘t and ‘e:

```
ax.one.ite[`t  `e] = [~=  ~ite[1  `t  `e]  `t];
```

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:

```
ax.Null.set.co.lg[‘x] = [~= [~Null.set ‘x] ~Null.set];
```

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Null.set.co.lg[‘x] is true.

Proof. ‘Func.Lg.Ext.Null.set is unchanging. □

Null set null predicate axiom:

```
ax.Null.set.Null = [~= [~Null ~Null.set] 0];
```

‘Func.Lg.Ext.ax.Null.set.Null is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ‘Func.Lg.Ext.Null.set](‘x) = ‘Func.Lg.Ext.Null(‘Func.Lg.Ext.Null.set(‘x)) = ‘Func.Lg.Ext.Null(‘Func.Sm.Ext.Tagged.Null.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Null set pair predicate axiom:

```
ax.Null.set.Pair = [~= [~Pair ~Null.set] 0];
```

‘Func.Lg.Ext.ax.Null.set.Pair is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ‘Func.Lg.Ext.Null.set](‘x) = ‘Func.Lg.Ext.Pair(‘Func.Lg.Ext.Null.set(‘x)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.Tagged.Null.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Null set domain axiom:

```
ax.Null.set.dom = [~= [~dom ~Null.set] ~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 identity. □

Null set small composition axiom: For a normalized large function ‘x:

```
ax.Null.set.co.sm[‘x] = [~= (~Null.set ‘x) ~null];
```

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Null.set.co.sm[‘x] is true.

Proof. For each tagged small function extension ‘y: (‘Func.Lg.Ext.Null.set ‘x)(‘y) = ‘Func.Lg.Ext.Null.set(‘y)(‘x(‘y)) = ‘Func.Sm.Ext.Tagged.Null.set(‘x(‘y)) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘y). □

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.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.Null.set(‘x) = ‘Func.Sm.Ext.Tagged.Null.set. ˜ite[‘Func.Lg.Ext.Null.set ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘x). □

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.

Proof. ‘Func.Lg.Ext.Nuro.set is unchanging. □

Nuro set null predicate axiom:

```
ax.Nuro.set.Null = [~=  [~Null  ~Nuro.set]  0];
```

‘Func.Lg.Ext.ax.Nuro.set.Null is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ‘Func.Lg.Ext.Nuro.set](‘x) = ‘Func.Lg.Ext.Null(‘Func.Lg.Ext.Nuro.set(‘x)) = ‘Func.Lg.Ext.Null(‘Func.Sm.Ext.Tagged.Nuro.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Nuro set pair predicate axiom:

```
ax.Nuro.set.Pair = [~=  [~Pair  ~Nuro.set]  0];
```

‘Func.Lg.Ext.ax.Nuro.set.Pair is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ‘Func.Lg.Ext.Nuro.set](‘x) = ‘Func.Lg.Ext.Pair(‘Func.Lg.Ext.Nuro.set(‘x)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.Tagged.Nuro.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Nuro set domain axiom:

```
ax.Nuro.set.dom = [~=  [~dom  ~Nuro.set]  ~Nuro.set];
```

‘Func.Lg.Ext.ax.Nuro.set.dom is true.

Proof. For each tagged small function extension ‘x, ‘Func.Lg.Ext.Nuro.set(‘x) = ‘Func.Sm.Ext.Tagged.Nuro.set is a rule tagged small function extension and an identity. □

Nuro set small composition axiom: For a normalized large function ‘x:

```
ax.Nuro.set.co.sm[‘x] =
[~=  (~Nuro.Set  ‘x)  [~Nuro.set.res  ‘x]];
```

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Nuro.set.co.sm[‘x] is true.

Proof. For each tagged small function extension ‘y: (‘Func.Lg.Ext.Nuro.Set ‘x)(‘y) = ‘Func.Lg.Ext.Nuro.Set(‘y)(‘x(‘y)) = ‘Func.Sm.Ext.Tagged.Nuro.set(‘x(‘y)) = ‘Func.Lg.Ext.Nuro.set.res(‘x(‘y)) = [‘Func.Lg.Ext.Nuro.set.res ‘x](‘y). □

Nuro set if-then-else axiom: For normalized large functions ‘t and ‘e:

```
ax.Nuro.set.ite[‘t ‘e] =
[~=  ~ite[~Nuro.set  ‘t  ‘e]  ~null];
```

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.Nuro.set.ite[‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.Nuro.set(‘x) = ‘Func.Sm.Ext.Tagged.Nuro.set. ~ite[‘Func.Lg.Ext.Nuro.set ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘x). □

8.25.7 LEAF SET

Leaf set large composition axiom: For a normalized large function ‘x:

```
ax.Leaf.set.co.lg[‘x] = [~=  [~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:

```
ax.Leaf.set.Null = [~=  [~Null  ~Leaf.set]  0];
```

‘Func.Lg.Ext.ax.Leaf.set.Null is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ‘Func.Lg.Ext.Leaf.set](‘x) = ‘Func.Lg.Ext.Null(‘Func.Lg.Ext.Leaf.set(‘x)) = ‘Func.Lg.Ext.Null(‘Func.Sm.Ext.Tagged.Leaf.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Leaf set pair predicate axiom:

```
ax.Leaf.set.Pair = [~=  [~Pair  ~Leaf.set]  0];
```

‘Func.Lg.Ext.ax.Leaf.set.Pair is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ‘Func.Lg.Ext.Leaf.set](‘x) = ‘Func.Lg.Ext.Pair(‘Func.Lg.Ext.Leaf.set(‘x)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.Tagged.Leaf.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Leaf set domain axiom:

```
ax.Leaf.set.dom = [~=  [~dom  ~Leaf.set]  ~Leaf.set];
```

‘Func.Lg.Ext.ax.Leaf.set.dom is true.

Proof. For each tagged small function extension ‘x, ‘Func.Lg.Ext.Leaf.set(‘x) = ‘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:

```
ax.Leaf.set.co.sm[‘x] =
[~=  (~Leaf.Set  ‘x)  [~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: (‘Func.Lg.Ext.Leaf.Set ‘x)(‘y) = ‘Func.Lg.Ext.Leaf.Set(‘y)(‘x(‘y)) = ‘Func.Sm.Ext.Tagged.Leaf.set(‘x(‘y)) = ‘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.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.Leaf.set(‘x) = ‘Func.Sm.Ext.Tagged.Leaf.set. ˜ite[‘Func.Lg.Ext.Leaf.set ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘x). □

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];
```

‘Func.Lg.Ext.ax.Tree.set.Null is true.

Proof. For each tagged small function extension ‘x:

[‘Func.Lg.Ext.Null ‘Func.Lg.Ext.Tree.set](‘x)
= ‘Func.Lg.Ext.Null(‘Func.Lg.Ext.Tree.set(‘x)) =
‘Func.Lg.Ext.Null(‘Func.Sm.Ext.Tagged.Tree.set) = ‘Func.Sm.Ext.zero =
‘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.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ‘Func.Lg.Ext.Tree.set](‘x) = ‘Func.Lg.Ext.Pair(‘Func.Lg.Ext.Tree.set(‘x)) = ‘Func.Lg.Ext.Pair(‘Func.Sm.Ext.Tagged.Tree.set) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

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] =
[~=  (~Tree.set  ‘x)  [~Tree.set.res  ‘x]];
```

For a large function extension ‘x, ‘Func.Lg.Ext.ax.Tree.set.co.sm[‘x] is true.

Proof. For each tagged small function extension ‘y: (‘Func.Lg.Ext.Tree.set ‘x)(‘y) = ‘Func.Lg.Ext.Tree.set(‘y)(‘x(‘y)) = ‘Func.Sm.Ext.Tagged.Tree.set(‘x(‘y)) = ‘Func.Lg.Ext.Tree.set.res(‘x(‘y)) = [‘Func.Lg.Ext.Tree.set.res ‘x](‘y). □

Tree set if-then-else axiom: For normalized large functions ‘t and ‘e:

```
ax.Tree.set.ite[‘t ‘e] =
[~=  ~ite[~Tree.set  ‘t  ‘e]  ~null];
```

For large function extensions ‘t and ‘e, ‘Func.Lg.Ext.ax.Tree.set.ite[‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.Tree.set(‘x) = ‘Func.Sm.Ext.Tagged.Tree.set. ~ite[‘Func.Lg.Ext.Tree.set ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘x). □

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] =
[ ~=  [[‘outer  ‘inner]  ‘x]  [‘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:

```
ax.co.lg.co.sm[‘called ‘arg ‘x] =
[ ~=  [(‘called  ‘arg)  ‘x]  ([‘called  ‘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)(‘x(‘y)) = ‘called(‘x(‘y))(‘arg(‘x(‘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:

```
ax.pair.co.lg[‘left ‘right ‘x] =
[ ~=  [{‘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: [{‘l ‘r} ‘x](‘y) = {‘l ‘r}(‘x(‘y)) = {‘l(‘x(‘y)), ‘r(‘x(‘y))} = {[‘l ‘x](‘y), [‘r ‘x](‘y)} = {[‘l ‘x] [‘r ‘x]}(‘y). □

Pair null predicate axiom: For normalized large functions ‘left and ‘right:

```
ax.pair.Null[‘left ‘right] =
[~=  [~Null  {‘left  ‘right}]  0];
```

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: [‘Func.Lg.Ext.Null {‘l ‘r}](‘x) = ‘Func.Lg.Ext.Null({‘l(‘x), ‘r(‘x)}) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Pair pair predicate axiom: For normalized large functions ‘left and ‘right:

```
ax.pair.Pair[‘left ‘right] =
[~=  [~Pair  {‘left  ‘right}]  1];
```

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: [‘Func.Lg.Ext.Pair {‘l ‘r}](‘x) = ‘Func.Lg.Ext.Pair({‘l(‘x), ‘r(‘x)}) = ‘Func.Sm.Ext.one = ‘Func.Lg.Ext.one(‘x). □

Pair domain axiom: For normalized large functions ‘left and ‘right:

```
ax.pair.dom['left 'right] =
[~= [~dom {'left 'right}] ~Leaf.set];
```

For large function extensions ‘l and ‘r, ‘Func.Lg.Ext.ax.pair.dom[‘l ‘r] is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.dom {‘l ‘r}](‘x) = ‘domFuncExt({‘l(‘x), ‘r(‘x)}) = ‘Func.Sm.Ext.Tagged.Leaf.set = ‘Func.Lg.Ext.Leaf.set(‘x). □

Pair small composition axiom: For normalized large functions ‘left, ‘right and ‘x:

```
ax.pair.co.sm['left 'right 'x] =
[ ~= ({'left 'right} 'x) ~ite['x 'right 'left] ];
```

For large function extensions ‘l, ‘r and ‘x, ‘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)}(‘x(‘y)). ({‘l ‘r} ‘x)(‘y) is given by one of the following mutually exclusive cases:
  - ‘l(‘x) if ‘x(‘y) = ‘Func.Sm.Ext.zero
  - ‘r(‘x) if ‘x(‘y) = ‘Func.Sm.Ext.one
  - ‘Func.Sm.Ext.null if ‘x(‘y) is not Boolean
- ~ite[‘x ‘r ‘l](‘y) is given by one of the following mutually exclusive cases:
  - ‘l(‘y) if ‘x(‘y) = ‘Func.Sm.Ext.zero
  - ‘r(‘y) if ‘x(‘y) = ‘Func.Sm.Ext.one
  - ‘Func.Sm.Ext.null if ‘x(‘y) is not Boolean
- ({‘l ‘r} ‘x)(‘y) = ~ite[‘x ‘r ‘l](‘y). □

Pair if-then-else axiom: For normalized large functions ‘left, ‘right, ‘t and ‘e:

```
ax.pair.ite['left 'right 't 'e] =
[~= ~ite[{'left 'right} 't 'e] ~null];
```

For large function extensions ‘l, ‘r, ‘t and ‘e, ‘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](‘x(‘y)) = ‘sumDep(‘family(‘x(‘y))) = ‘sumDep([‘family ‘x](‘y)) = ˜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.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ˜s.d[‘family]](‘x) = ‘Func.Lg.Ext.Null(˜s.d[‘family](‘x)) = ‘Func.Lg.Ext.Null(‘sumDep(‘family(‘x))) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

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.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ˜s.d[‘family]](‘x) = ‘Func.Lg.Ext.Pair(˜s.d[‘family](‘x)) = ‘Func.Lg.Ext.Pair(‘sumDep(‘family(‘x))) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

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, ˜s.d[‘family](‘x) = ‘sumDep(‘family(‘x)) is a rule tagged small function extension and an identity. □

Dependent sum small composition axiom: For normalized large functions ‘family and ‘x:

```
ax.s.d.co.sm[`family `x] =
[ ~=  (~s.d[`family]  `x)  [~s.d.res  `family  `x] ];
```

For large function extensions ‘family and ‘x, ‘Func.Lg.Ext.ax.s.d.co.sm[‘family ‘x] is true.

Proof. For each tagged small function extension ‘y: (˜s.d[‘family] ‘x)(‘y) = ˜s.d[‘family](‘y)(‘x(‘y)) = ‘sumDep(‘family(‘y))(‘x(‘y)) = ‘Func.Lg.Ext.s.d.res({‘family(‘y), ‘x(‘y)}) = [‘Func.Lg.Ext.s.d.res ‘family ‘x](‘y). □

Dependent sum if-then-else axiom: For normalized large functions ‘family, ‘t and ‘e:

```
ax.s.d.ite[`family `t `e] =
[~=  ~ite[~s.d[`family]  `t  `e]  ~null];
```

For large function extensions ‘family, ‘t and ‘e, ‘Func.Lg.Ext.ax.s.d.ite[‘family ‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ˜s.d[‘family](‘x) = ‘sumDep(‘family(‘x)). ˜ite[˜s.d[‘family] ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘x). □

8.25.13 DEPENDENT PRODUCT

Dependent product large composition axiom: For normalized large functions ‘family and ‘x:

```
ax.p.d.co.lg[`family `x] =
[ ~=  [~p.d[`family]  `x]  ~p.d[[`family  `x]] ];
```

For large function extensions ‘family and ‘x, ‘Func.Lg.Ext.ax.p.d.co.lg[‘family ‘x] is true.

Proof. For each tagged small function extension ‘y: [˜p.d[‘family] ‘x](‘y) = ˜p.d[‘family](‘x(‘y)) = ‘prodDep(‘family(‘x(‘y))) = ‘prodDep([‘family ‘x](‘y)) = ˜p.d[[‘family ‘x]](‘y). □

Dependent product null predicate axiom: For a normalized large function ‘family:

```
ax.p.d.Null[`family] = [~=  [~Null  ~p.d[`family]]  0];
```

For a large function extension ‘family, ‘Func.Lg.Ext.ax.p.d.Null[‘family] is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ˜p.d[‘family]](‘x) = ‘Func.Lg.Ext.Null(˜p.d[‘family](‘x)) = ‘Func.Lg.Ext.Null(‘prodDep(‘family(‘x))) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Dependent product pair predicate axiom: For a normalized large function ‘family:

```
ax.p.d.Pair[`family] = [~=  [~Pair  ~p.d[`family]]  0];
```

For a large function extension ‘family, ‘Func.Lg.Ext.ax.p.d.Pair[‘family] is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ˜p.d[‘family]](‘x) = ‘Func.Lg.Ext.Pair(˜p.d[‘family](‘x)) = ‘Func.Lg.Ext.Pair(‘prodDep(‘family(‘x))) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Dependent product domain axiom: For a normalized large function ‘family:

```
ax.p.d.dom[`family] =
[ ~=  [~dom  ~p.d[`family]]  ~p.d[`family] ];
```

For a large function extension ‘family, ‘Func.Lg.Ext.ax.p.d.dom[‘family] is true.

Proof. For each tagged small function extension ‘x, ˜p.d[‘family](‘x) = ‘prodDep(‘family(‘x)) is a rule tagged small function extension and an identity. □

Dependent product small composition axiom: For normalized large functions ‘family and ‘x:

```
ax.p.d.co.sm[`family `x] =
[ ~=  (~p.d[`family]  `x)  [~p.d.res  `family  `x] ];
```

For large function extensions ‘family and ‘x, ‘Func.Lg.Ext.ax.p.d.co.sm[‘family ‘x] is true.

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] =
[~=  ~ite[~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] =
[~=
    [~c['uncurry  'restrictor]  'x]
    ~c.aug['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: [˜c[‘uncurry ‘restrictor] ‘x](‘y) = ˜c.aug[‘uncurry ‘restrictor ‘x](‘y). □

Curry null predicate axiom: For normalized large functions ‘uncurry and ‘restrictor:

```
ax.c.Null['uncurry 'restrictor] =
[~=  [~Null  ~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: [‘Func.Lg.Ext.Null ˜c[‘uncurry ‘restrictor]](‘x) = ‘Func.Lg.Ext.Null(˜c[‘uncurry ‘restrictor](‘x)) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Curry pair predicate axiom: For normalized large functions ‘uncurry and ‘restrictor:

```
ax.c.Pair[`uncurry `restrictor] =
[~=  [~Pair  ~c[`uncurry  `restrictor]]  0];
```

For large function extensions ‘uncurry and ‘restrictor, ‘Func.Lg.Ext.ax.c.Pair[‘uncurry ‘restrictor] is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Pair ˜c[‘uncurry ‘restrictor]](‘x) = ‘Func.Lg.Ext.Pair(˜c[‘uncurry ‘restrictor](‘x)) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

Curry domain axiom: For normalized large functions ‘uncurry and ‘restrictor:

```
ax.c.dom[`uncurry `restrictor] =
[~=
    [~dom  ~c[`uncurry  `restrictor]]
    [~dom  `restrictor]
];
```

For large function extensions ‘uncurry and ‘restrictor, ‘Func.Lg.Ext.ax.c.dom[‘uncurry ‘restrictor] is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.dom ˜c[‘uncurry ‘restrictor]](‘x) = ‘domFuncExt(˜c[‘uncurry ‘restrictor](‘x)) = ‘domFuncExt(‘restrictor(‘x)) = [‘Func.Lg.Ext.dom ‘restrictor](‘x). □

Curry small composition axiom: For normalized large functions ‘uncurry, ‘restrictor and ‘x:

```
ax.c.co.sm[`uncurry `restrictor `x] =
[~=
    (~c[`uncurry  `restrictor]  `x)
    [~c.res[`uncurry]  ~i  `restrictor  `x]
];
```

For large function extensions ‘uncurry, ‘restrictor and ‘x, ‘Func.Lg.Ext.ax.c.co.sm[‘uncurry ‘restrictor ‘x] is true.

Proof. For each tagged small function extension ‘y: (˜c[‘uncurry ‘restrictor] ‘x)(‘y) = ˜c[‘uncurry ‘restrictor](‘y)(‘x(‘y)) = ‘uncurry({‘y, ‘domFuncExt(‘restrictor(‘y))(‘x(‘y))}) = ˜c.res[‘uncurry]({‘y, ‘restrictor(‘y), ‘x(‘y)}) = [˜c.res[‘uncurry] ‘Func.Lg.Ext.i ‘restrictor ‘x](‘y). □

Curry if-then-else axiom: For normalized large functions ‘uncurry, ‘restrictor, ‘t and ‘e:

```
ax.c.ite[`uncurry `restrictor `t `e] =
[~=  ~ite[~c[`uncurry  `restrictor]  `t  `e]  ~null];
```

For large function extensions ‘uncurry, ‘restrictor, ‘t and ‘e, ‘Func.Lg.Ext.ax.c.ite[‘uncurry ‘restrictor ‘t ‘e] is true.

Proof. For each tagged small function extension ‘x: ˜ite[˜c[‘uncurry ‘restrictor] ‘t ‘e](‘x) = ‘Func.Sm.Ext.null = ‘Func.Lg.Ext.null(‘x). □

8.25.15 IF-THEN-ELSE

If-then-else large composition axiom: For normalized large functions ‘ifP, ‘thenP, ‘elseP and ‘x:

```
ax.ite.co.lg[`ifP `thenP `elseP `x] =
[~=
    [~ite[`ifP  `thenP  `elseP]  `x]
    ~ite[[`ifP  `x]  [`thenP  `x]  [`elseP  `x]]
];
```

For large function extensions ‘ifP, ‘thenP, ‘elseP and ‘x, ‘Func.Lg.Ext.ax.ite.co.lg[‘ifP ‘thenP ‘elseP ‘x] is true.

Proof.

- For each tagged small function extension ‘y: [˜ite[‘ifP ‘thenP ‘elseP] ‘x](‘y) = ˜ite[‘ifP ‘thenP ‘elseP](‘x(‘y)). [˜ite[‘ifP ‘thenP ‘elseP] ‘x](‘y) is given by one of the following mutually exclusive cases:
  - ‘elseP(‘x(‘y)) if ‘ifP(‘x(‘y)) = ‘Func.Sm.Ext.zero
  - ‘thenP(‘x(‘y)) if ‘ifP(‘x(‘y)) = ‘Func.Sm.Ext.one
  - ‘Func.Sm.Ext.null if ‘ifP(‘x(‘y)) is not Boolean

- ˜ite[[‘ifP ‘x] [‘thenP ‘x] [‘elseP ‘x]](‘y) is given by one of the following mutually exclusive cases:
  - [‘elseP ‘x](‘y) if [‘ifP ‘x](‘y) = ‘Func.Sm.Ext.zero
  - [‘thenP ‘x](‘y) if [‘ifP ‘x](‘y) = ‘Func.Sm.Ext.one
  - ‘Func.Sm.Ext.null if [‘ifP ‘x](‘y) is not Boolean
- [˜ite[‘ifP ‘thenP ‘elseP] ‘x](‘y) = ˜ite[[‘ifP ‘x] [‘thenP ‘x] [‘elseP ‘x]](‘y). □

8.25.16 RECURSION

Recursion right-hand-side axiom: For normalized large functions ‘start and ‘step:

```
ax.r.rhs[`start `step] =
[~= ~r[`start `step] ~r.rhs[`start `step]];
```

For large function extensions ‘start and ‘step, ‘Func.Lg.Ext.ax.r.rhs[‘start ‘step] is true.

Proof. For each tagged small function extension ‘x: ˜r[‘start ‘step](‘x) = ˜r.rhs[‘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} \ 1 =
[~imp %b [~imp %c %b]];
```

‘Func.Lg.Ext.ax.logic.weakening is true.

Logic nested implication axiom:

```
ax.logic.imp.nested {%b %c %d} \ 1 =
[~imp
   [~imp %b [~imp %c %d]]
   [~imp [~imp %b %c] [~imp %b %d]]
```

```
];
```

‘Func.Lg.Ext.ax.logic.imp.nested is true.

Logic contrapositive axiom:

```
ax.logic.contrapos {%b %c} \ 1 =
[~imp
    [~imp  [~not  %b]  [~not  %c]]
    [~imp  %c  %b]
];
```

‘Func.Lg.Ext.ax.logic.contrapos is true.

8.25.18 TRUTH

Truth introduction axiom:

```
ax.truth.intro =
[~imp
    [~=  ~i  1]
    ~i
];
```

‘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:

```
ax.truth.elim =
[~imp
    ~i
    [~=  ~i  1]
];
```

‘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:

```
ax.eq.rhs = [~=  ~=  ~=.rhs];
```

‘Func.Lg.Ext.ax.eq.rhs is true.

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.eq(‘x) = ‘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:

```
ax.eq.reflex[‘x] = [~=  ‘x  ‘x];
```

For a large function extension ‘x, ‘Func.Lg.Ext.ax.eq.reflex[‘x] is true.

Proof. For each tagged small function extension ‘y: ‘x(‘y) = ‘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:

```
ax.eq.subst[‘pred] {%x %y %z} \ 1 =
[~imp
    [~=  %y  %z]
[~imp
    [‘pred  %x  %y]
    [‘pred  %x  %z]
]];
```

For a large function extension ‘pred, ‘Func.Lg.Ext.ax.eq.subst[‘pred] is true.

Proof. For each tagged small function extension ‘w = {‘x, ‘y, ‘z}: If ‘Func.Lg.Ext.eq({‘y, ‘z}) and ‘pred({‘x, ‘y}) are true, then ‘y = ‘z, and ‘pred({‘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:

```
ax.h.transfinite[‘pred] {%x %y} \ 1 =
```

```
[~imp
    `pred
    [~exist[`pred]  %x]
];
```

For a large function extension ‘pred, ‘Func.Lg.Ext.ax.h.transfinite[‘pred] is true.

Proof. For each tagged small function extension ‘z = {‘x, ‘y}: If ‘pred(‘z) is true, then ˜exist[‘pred](‘x) is true. □

8.25.21 INDUCTION

Induction axiom: For a normalized large function ‘pred:

```
ax.induc[`pred] =
[~imp
    [`pred  ~null]
[~imp
    ~induc.case[`pred]
    `pred
]];
```

For a large function extension ‘pred, ‘Func.Lg.Ext.ax.induc[‘pred] is true.

Proof.

- If ‘pred(‘Func.Sm.Ext.null) is true, and ˜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):
  - Holds if ‘x = ‘Func.Sm.Ext.null.
  - If ‘x ≠ ‘Func.Sm.Ext.null: For each ‘dom(‘x) program ‘y, ‘pred(‘tagged(‘x, ‘y)) and ‘pred(‘x<‘y>) are true (by inductive hypothesis).
- For each tagged small function extension ‘x, if ‘pred(‘Func.Sm.Ext.null) is true, and ˜induc.case[‘pred](‘x) = ˜induc.case[‘pred](‘Func.Sm.Ext.null) is true, then ‘pred(‘x) is true. □

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:

```
ax.Null.otw =
[~imp
    [~not.=  ~i  ~null]
    [~=  ~Null  0]
];
```

‘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:

```
ax.Pair.otw =
[~imp
    [~not.=  ~i  {~left  ~right}]
    [~=  ~Pair  0]
];
```

‘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:

```
ax.dom.Null = [~=  [~Null  ~dom]  0];
```

‘Func.Lg.Ext.ax.dom.Null is true.

Proof. For each tagged small function extension ‘x: [‘Func.Lg.Ext.Null ‘Func.Lg.Ext.dom](‘x) = ‘Func.Lg.Ext.Null(‘domFuncExt(‘x)) = ‘Func.Sm.Ext.zero = ‘Func.Lg.Ext.zero(‘x). □

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.dom](‘x) = ‘Func.Lg.Ext.Pair(‘domFuncExt(‘x)) = ‘Func.Sm.Ext.zero = ‘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))) = ‘domFuncExt(‘y)(‘x(‘y)) = (‘Func.Lg.Ext.dom ‘x)(‘y). □

If-then-else otherwise axiom: For normalized large functions ‘ifP, ‘thenP, ‘elseP:

```
ax.ite.otw[‘ifP ‘thenP ‘elseP] =
[~imp
    [~not.=  ‘ifP  0]
[~imp
    [~not.=  ‘ifP  1]
    [~=  ~ite[‘ifP  ‘thenP  ‘elseP]  ~null]
]];
```

‘Func.Lg.Ext.ax.ite.otw is true.

Proof. For each tagged small function extension ‘x: If ‘Func.Lg.Ext.not.eq({‘ifP(‘x), ‘Func.Sm.Ext.zero}) and ‘Func.Lg.Ext.not.eq({‘ifP(‘x), ‘Func.Sm.Ext.one}) are true, then ‘ifP(‘x) is not Boolean, ˜ite[‘ifP ‘thenP ‘elseP](‘x) = ‘Func.Sm.Ext.null, and ‘Func.Lg.Ext.eq({˜ite[‘ifP ‘thenP ‘elseP](‘x), ‘Func.Sm.Ext.null}) is true. □

8.26 SOME INFERENCES FROM TRUE LARGE FUNCTION EXTENSIONS

8.26.1 MODUS PONENS

The following is similar to rule of inference 1 in [36, p.6].

Modus ponens inference: For large function extensions ‘b and ‘c, if the following is true:

[‘Func.Lg.Ext.imp ‘b ‘c]

and the following is true:

‘b

then the following is true:

‘c

Proof. For each tagged small function extension ‘x: ‘Func.Lg.Ext.imp({‘b(‘x), ‘c(‘x)}) is true. ‘b(‘x) is true. ‘c(‘x) is true. □

8.26.2 SPECIALIZATION

The following is somewhat similar to the substitution rule in [4]. However, in

NummSquared, substitution does not actually take place here.

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
```

Proof. ‘ext(‘x) = ‘ext(‘y). ‘ext(‘pred0) = ‘ext(‘pred1) (by substitution theorem). ‘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 NummSquared 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 <‘pred, ‘x, ‘general>: ‘prp(‘general) = `‘pred`. ‘general is valid. `‘pred` is true (by inductive hypothesis). `[ ‘pred  ‘x]` is true.

- If ‘p is a substitution inference containing <‘pred0, ‘pred1, ‘x, ‘y, ‘equality, ‘before>: ‘subst(‘pred0, ‘pred1, ‘x, ‘y). ‘prp(‘equality) = `[~=  ‘x  ‘y]`. ‘prp(‘before) = `‘pred0`. ‘equality and ‘before are valid. `[~=  ‘x  ‘y]` and `‘pred0` are true (by inductive hypothesis). `‘pred1` 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 ‘tag, and normalized large functions ‘children0 and ‘children1, the tree of ‘tag, ‘children0 and ‘children1, denoted by ‘tree(‘tag, ‘children0, ‘children1), is {‘norm(‘tag) ‘children0 ‘children1}.

For a natural number ‘tag, and *tree* normalized large functions ‘children0 and ‘children1, ‘tree(‘tag, ‘children0, ‘children1) is a tree.

Let ‘axiomCount be the number of axiom cases.

For a proof ‘p, the **tag** of ‘p, denoted by ‘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.

- ‘axiomCount if ‘p is a modus ponens inference

- ‘axiomCount + 1 if ‘p is a specialization inference

- ‘axiomCount + 2 if ‘p is a substitution inference

For an axiom ‘a, the **quoted** of ‘a (a tree normalized large function), denoted by ‘quoted(‘a), is given by one of the following mutually exclusive cases:

- ‘tree(‘tag(‘a), ~l{‘quoted(‘x)}, ~l{}) if ‘a is an identity large composition axiom containing ‘x

- The other axiom cases are similar and are omitted.

For a proof ‘p, the **quoted** of ‘p (a tree normalized large function), denoted by ‘quoted(‘p), is defined by recursion on ‘p:

- as above if ‘p is an axiom
- ‘tree(‘tag(‘p), ~l{‘quoted(‘b) ‘quoted(‘c)}, ~l{‘quoted(‘major) ‘quoted(‘minor)}) if ‘p is a modus ponens inference containing <‘b, ‘c, ‘major, ‘minor>
- ‘tree(‘tag(‘p), ~l{‘quoted(‘pred) ‘quoted(‘x)}, ~l{‘quoted(‘general)}) if ‘p is specialization inference containing <‘pred, ‘x, ‘general>
- ‘tree(‘tag(‘p), ~l{‘quoted(‘pred0) ‘quoted(‘pred1) ‘quoted(‘x) ‘quoted(‘y)}, ~l{‘quoted(‘equality) ‘quoted(‘before)}) if ‘p is a substitution inference containing <‘pred0, ‘pred1, ‘x, ‘y, ‘equality, ‘before>

8.32 PROOF UNQUOTED OF A NORMALIZED LARGE FUNCTION

For a normalized large function ‘f, the **proof unquoted** of ‘f, denoted by ‘unquotedProof(‘f), is the proof ‘p such that ‘quoted(‘p) = ‘f if such exists; and ‘null otherwise.

For a normalized large function ‘f, ‘unquotedProof(‘f) is computable.

For a normalized large function ‘f, ‘f is a **quoted proof** iff ‘unquotedProof(‘f) ≠ ‘null.

For a normalized large function ‘f, ‘f is a quoted proof iff there exists a proof ‘p such that ‘quoted(‘p) = ‘f.

For a normalized large function ‘f, if ‘f is quoted proof, then ‘f is a tree.

For a normalized large function ‘f, ‘f is a **valid quoted proof** iff ‘f is a quoted proof and ‘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.

8.33 RUSSELL’S PARADOX AVERTED

It is interesting to examine how NummSquared averts Russell’s paradox.

```
Rus = (~i  ~i);

Rus.sm = ~restrict[Rus];

Rus.paradox = [Rus  Rus.sm];
```

Of course, ‘Func.Lg.Ext.Rus(‘Func.Lg.Ext.Rus) cannot be constructed since ‘Func.Lg.Ext.Rus is a large function extension.

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Rus(‘x) = ‘x(‘x).

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Rus.sm(‘x) is the rule tagged small function extension ‘r such that ‘domExt(‘r) = ‘domExt(‘x) and, for each ‘dom(‘r) program ‘y, ‘r<‘y> = ‘Func.Lg.Ext.Rus(‘tagged(‘r, ‘y)) = ‘tagged(‘r, ‘y)(‘tagged(‘r, ‘y)) = ‘tagged(‘x, ‘y)(‘tagged(‘x, ‘y)).

For tagged small function extensions ‘x and ‘y, ‘Func.Lg.Ext.Rus.sm(‘x)(‘y) = ‘Func.Lg.Ext.Rus.sm(‘x)<‘coer(‘Func.Lg.Ext.Rus.sm(‘x), ‘y)> = ‘Func.Lg.Ext.Rus.sm(‘x)<‘coer(‘x, ‘y)> = ‘tagged(‘x, ‘coer(‘x, ‘y))(‘tagged(‘x, ‘coer(‘x, ‘y))).

For a tagged small function extension ‘x, ‘Func.Lg.Ext.Rus.paradox(‘x) = ‘Func.Lg.Ext.Rus(‘Func.Lg.Ext.Rus.sm(‘x)) = ‘Func.Lg.Ext.Rus.sm(‘x)(‘Func.Lg.Ext.Rus.sm(‘x)) = ‘tagged(‘x, ‘coer(‘x, ‘Func.Lg.Ext.Rus.sm(‘x))) (‘tagged(‘x, ‘coer(‘x, ‘Func.Lg.Ext.Rus.sm(‘x)))).

‘res(‘Func.Lg.Ext.Rus.paradox) = ‘Func.Lg.Ext.Rus.paradox(‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.null(‘Func.Sm.Ext.null) = ‘Func.Sm.Ext.null.

Thus the result of Russell’s paradox is ‘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 occurence 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, 4.3].) 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 ∀(‘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 *fun*(‘x : ‘A) ⇒ ‘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.NummSquared.v2006a0.Preface*. The short name of *PoohbistTechnology.NummSquared.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 ∀(‘x : ‘A), ‘B is written within a type cast (∀(‘x : ‘A), ‘B) : ‘s such that ‘s is a sort.

For clarity, each function *fun*(‘x1 : ‘A) ⇒ ‘b is written within a type cast (*fun* ‘x1 ⇒ ‘b) : ((∀(‘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*

Set is not be used. *Type* is used instead. (*Type* is more flexible because Coq internally replaces each occurence of *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

Poohbist.NummSquared.Fundamentals.Operators.Main

Poohbist.NummSquared.Fundamentals.Operators.Main defines operators, binary operators, trinary operators, quaternary operators, quinary operators, and some fundamental operators.

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 $Op := (\ fun\ A\ B \Rightarrow (\ \forall (a : A),\ B\ )\ ) : Op\_Ty.$

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 $Op\_const\_Ty :=$
$(\ \forall (A : Type)(B : Type)(b : B),\ (Op\ A\ B)\ ) : Type.$

Definition $Op\_const := (\ fun\ A\ B\ b\ a \Rightarrow b\ ) : Op\_const\_Ty.$

9.2.3 SIMPLE OPERATORS

A simple operator on A is an operator from A to A.

Definition $Op\_Simp\_Ty := (\ \forall (A : Type),\ Type\ ) : Type.$

Definition $Op\_Simp := (\ fun\ A \Rightarrow (Op\ A\ A)\ ) : Op\_Simp\_Ty.$

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 $Op\_Simp\_identity\_Ty := (\ \forall (A : Type),\ (Op\_Simp\ A)\ ) : Type.$

Definition $Op\_Simp\_identity := (\ fun\ A\ a \Rightarrow a\ ) : Op\_Simp\_identity\_Ty.$

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 Op_Bin_Ty := ($\forall(A0 : Type)(A1 : Type)(B : Type), Type$) : $Type$.

Definition Op_Bin := ($fun\ A0\ A1\ B \Rightarrow (Op\ A0\ (Op\ A1\ B))$) : Op_Bin_Ty.

9.2.6 CONNECTIVE BINARY OPERATORS

A connective binary operator from A to B is a binary operator from A, A to B.

Definition $Op\_Bin\_Conn\_Ty$:= ($\forall(A : Type)(B : Type), Type$) : $Type$.

Definition Op_Bin_Conn := ($fun\ A\ B \Rightarrow (Op\_Bin\ A\ A\ B)$) : $Op\_Bin\_Conn\_Ty$.

9.2.7 SIMPLE BINARY OPERATORS

A simple binary operator on A is a connective binary operator from A to A.

Definition $Op\_Bin\_Simp\_Ty$:= ($\forall(A : Type), Type$) : $Type$.

Definition Op_Bin_Simp := ($fun\ A \Rightarrow (Op\_Bin\_Conn\ A\ A)$) : $Op\_Bin\_Simp\_Ty$.

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 Op_Tri_Ty :=

($\forall(A0 : Type)(A1 : Type)(A2 : Type)(B : Type), Type$) : $Type$.

Definition Op_Tri :=

($fun\ A0\ A1\ A2\ B \Rightarrow (Op\ A0\ (Op\_Bin\ A1\ A2\ B))$) : Op_Tri_Ty.

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 $Op\_Tri\_Conn\_Ty$:= ($\forall(A : Type)(B : Type), Type$) : $Type$.

Definition Op_Tri_Conn := ($fun\ A\ B \Rightarrow (Op\_Tri\ A\ A\ A\ B)$) : $Op\_Tri\_Conn\_Ty$.

9.2.10 SIMPLE TRINARY OPERATORS

A simple trinary operator on *A* is a connective trinary operator from *A* to *A*.

Definition $Op\_Tri\_Simp\_Ty := (\,\forall (A : Type),\, Type\,) : Type.$

Definition $Op\_Tri\_Simp := (\,fun\ A \Rightarrow (Op\_Tri\_Conn\ A\ A)\,) : Op\_Tri\_Simp\_Ty.$

9.2.11 QUATERNARY OPERATORS

A quaternary operator from *A0*, *A1*, *A2*, *A3* to *B* is an operator from *A0* to a trinary operator from *A1*, *A2*, *A3* to *B*.

Definition $Op\_Quat\_Ty :=$

$$(\,\forall (A0 : Type)(A1 : Type)(A2 : Type)(A3 : Type)(B : Type),\, Type\,)$$
$$: Type.$$

Definition $Op\_Quat :=$

$$(\,fun\ A0\ A1\ A2\ A3\ B \Rightarrow (Op\ A0\ (Op\_Tri\ A1\ A2\ A3\ B))\,) : Op\_Quat\_Ty.$$

9.2.12 CONNECTIVE QUATERNARY OPERATORS

A connective quaternary operator from *A* to *B* is a quaternary operator from *A*, *A*, *A*, *A* to *B*.

Definition $Op\_Quat\_Conn\_Ty := (\,\forall (A : Type)(B : Type),\, Type\,) : Type.$

Definition $Op\_Quat\_Conn :=$

$$(\,fun\ A\ B \Rightarrow (Op\_Quat\ A\ A\ A\ A\ B)\,) : Op\_Quat\_Conn\_Ty.$$

9.2.13 SIMPLE QUATERNARY OPERATORS

A simple quaternary operator on *A* is a connective quaternary operator from *A* to *A*.

Definition $Op\_Quat\_Simp\_Ty := (\,\forall (A : Type),\, Type\,) : Type.$

Definition $Op\_Quat\_Simp := (\,fun\ A \Rightarrow (Op\_Quat\_Conn\ A\ A)\,) : Op\_Quat\_Simp\_Ty.$

9.2.14 QUINARY OPERATORS

A quinary operator from *A0*, *A1*, *A2*, *A3*, *A4* to *B* is an operator from *A0* to a quaternary operator from *A1*, *A2*, *A3*, *A4* to *B*.

Definition Op_Quin_Ty :=
($\forall$
($A0$: $Type$)
($A1$: $Type$)
($A2$: $Type$)
($A3$: $Type$)
($A4$: $Type$)
(B : $Type$),
$Type$
) : $Type$.

Definition Op_Quin :=
($fun\ A0\ A1\ A2\ A3\ A4\ B \Rightarrow (Op\ A0\ (Op\_Quat\ A1\ A2\ A3\ A4\ B))$)
: 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 :=
($fun\ A\ B \Rightarrow (Op\_Quin\ A\ A\ A\ A\ A\ B)$) : $Op\_Quin\_Conn\_Ty$.

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 := ($fun\ A \Rightarrow (Op\_Quin\_Conn\ A\ A)$) : $Op\_Quin\_Simp\_Ty$.

9.3 FUNDAMENTALS: PROPOSITIONS: MAIN

Poohbist.NummSquared.Fundamentals.Propositions.Main

Poohbist.NummSquared.Fundamentals.Propositions.Main defines propositional predicates, binary propositional predicates, trinary propositional predicates, quaternary propositional predicates, quinary propositional predicates, some fundamental propositional predicates, the true proposition, and the false proposition.

9.3.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main*.

9.3.2 PROPOSITIONAL PREDICATES

A propositional predicate on *A* is an operator from *A* to *Prop*.

Definition *Prp_Pred_Ty* := (∀(*A* : *Type*), *Type*) : *Type*.

Definition *Prp_Pred* := (*fun A* ⇒ (*Op A Prop*)) : *Prp_Pred_Ty*.

9.3.3 THE CONSTANT PROPOSITIONAL PREDICATE

The constant propositonal predicate on *A* onto *P* : *Prop* is the constant operator from *A* to *Prop* onto *P*.

Definition *Prp_Pred_const_Ty* :=
(∀(*A* : *Type*)(*P* : *Prop*), (*Prp_Pred A*)) : *Type*.

Definition *Prp_Pred_const* :=
(*fun A P* ⇒ (*Op_const A Prop P*)) : *Prp_Pred_const_Ty*.

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_Ty* := (∀(*A0* : *Type*)(*A1* : *Type*), *Type*) : *Type*.

Definition *Prp_Pred_Bin* :=
(*fun A0 A1* ⇒ (*Op_Bin A0 A1 Prop*)) : *Prp_Pred_Bin_Ty*.

9.3.5 CONNECTIVE BINARY PROPOSITIONAL PREDICATES

A connective binary propositional predicate on *A* is a binary propositional predicate on *A*, *A*.

Definition $Prp\_Pred\_Bin\_Conn\_Ty := (\,\forall (A : Type),\ Type\,) : Type.$

Definition $Prp\_Pred\_Bin\_Conn :=$

$$(\,fun\ A \Rightarrow (Prp\_Pred\_Bin\ A\ A)\,) : Prp\_Pred\_Bin\_Conn\_Ty.$$

9.3.6 TRINARY PROPOSITIONAL PREDICATES

A trinary propositional predicate on *A0, A1, A2* is a trinary operator from *A0, A1, A2* to *Prop*.

Definition $Prp\_Pred\_Tri\_Ty :=$

$$(\,\forall (A0 : Type)(A1 : Type)(A2 : Type),\ Type\,) : Type.$$

Definition $Prp\_Pred\_Tri :=$

$$(\,fun\ A0\ A1\ A2 \Rightarrow (Op\_Tri\ A0\ A1\ A2\ Prop)\,) : Prp\_Pred\_Tri\_Ty.$$

9.3.7 CONNECTIVE TRINARY PROPOSITIONAL PREDICATES

A connective trinary propositional predicate on *A* is a trinary propositional predicate on *A, A, A*.

Definition $Prp\_Pred\_Tri\_Conn\_Ty := (\,\forall (A : Type),\ Type\,) : Type.$

Definition $Prp\_Pred\_Tri\_Conn :=$

$$(\,fun\ A \Rightarrow (Prp\_Pred\_Tri\ A\ A\ A)\,) : Prp\_Pred\_Tri\_Conn\_Ty.$$

9.3.8 QUATERNARY PROPOSITIONAL PREDICATES

A quaternary propositional predicate on *A0, A1, A2, A3* is a quaternary operator from *A0, A1, A2, A3* to *Prop*.

Definition $Prp\_Pred\_Quat\_Ty :=$

$$(\,\forall (A0 : Type)(A1 : Type)(A2 : Type)(A3 : Type),\ Type\,) : Type.$$

Definition $Prp\_Pred\_Quat :=$

$$(\,fun\ A0\ A1\ A2\ A3 \Rightarrow (Op\_Quat\ A0\ A1\ A2\ A3\ Prop)\,) : Prp\_Pred\_Quat\_Ty.$$

9.3.9 CONNECTIVE QUATERNARY PROPOSITIONAL PREDICATES

A connective quaternary propositional predicate on *A* is a quaternary propositional predicate on *A*, *A*, *A*, *A*.

Definition $Prp\_Pred\_Quat\_Conn\_Ty := (\,\forall (A : Type),\ Type\,) : Type.$

Definition $Prp\_Pred\_Quat\_Conn :=$

$(\,fun\ A \Rightarrow (Prp\_Pred\_Quat\ A\ A\ A\ A)\,) : Prp\_Pred\_Quat\_Conn\_Ty.$

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 *Prop*.

Definition $Prp\_Pred\_Quin\_Ty :=$

$(\,\forall (A0 : Type)(A1 : Type)(A2 : Type)(A3 : Type)(A4 : Type),\ Type\,)$

$: Type.$

Definition $Prp\_Pred\_Quin :=$

$(\,fun\ A0\ A1\ A2\ A3\ A4 \Rightarrow (Op\_Quin\ A0\ A1\ A2\ A3\ A4\ Prop)\,)$

$: Prp\_Pred\_Quin\_Ty.$

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 $Prp\_Pred\_Quin\_Conn :=$

$(\,fun\ A \Rightarrow (Prp\_Pred\_Quin\ A\ A\ A\ A\ A)\,) : Prp\_Pred\_Quin\_Conn\_Ty.$

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 *True* in *Coq.Init.Logic*.

Inductive $Prp\_T : Prop :=$

$\mid Prp\_T\_proof : Prp\_T.$

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

Poohbist.NummSquared.Fundamentals.Booleans.Main

Poohbist.NummSquared.Fundamentals.Booleans.Main defines Booleans, Boolean predicates, binary Boolean predicates, trinary Boolean predicates, quaternary Boolean predicates, quinary Boolean predicates, some fundamental Boolean predicates, and an operator from Booleans to propositions.

9.4.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Propositions.Main*.

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 $Boo\_Pred\_Tri\_Ty$:=

($\forall(A0 : Type)(A1 : Type)(A2 : Type), Type$) : *Type*.

Definition *Boo_Pred_Tri* :=

(*fun A0 A1 A2* $\Rightarrow$ (*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* := ($\forall(A : Type), Type$) : *Type*.

Definition *Boo_Pred_Tri_Conn* :=

(*fun A* $\Rightarrow$ (*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* :=

($\forall(A0 : Type)(A1 : Type)(A2 : Type)(A3 : Type), Type$) : *Type*.

Definition *Boo_Pred_Quat* :=

(*fun A0 A1 A2 A3* $\Rightarrow$ (*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* := ($\forall(A : Type), Type$) : *Type*.

Definition *Boo_Pred_Quat_Conn* :=

(*fun A* $\Rightarrow$ (*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 *Boo_Pred_Quin_Ty* :=
    ($\forall$(*A0* : *Type*)(*A1* : *Type*)(*A2* : *Type*)(*A3* : *Type*)(*A4* : *Type*), *Type*)
    : *Type*.

Definition *Boo_Pred_Quin* :=
    (*fun A0 A1 A2 A3 A4* $\Rightarrow$ (*Op_Quin A0 A1 A2 A3 A4 Boo*))
    : *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 *Boo_Pred_Quin_Conn_Ty* := ($\forall$(*A* : *Type*), *Type*) : *Type*.

Definition *Boo_Pred_Quin_Conn* :=
    (*fun A* $\Rightarrow$ (*Boo_Pred_Quin A A A A A*)) : *Boo_Pred_Quin_Conn_Ty*.

9.4.13 BOOLEAN TO PROPOSITION

(*Boo_to_Prp b*) is the true proposition if *b*; and the false proposition otherwise.

Boo_to_Prp is defined in the same way as *Is_true* in *Coq.Bool.Bool*.

Definition *Boo_to_Prp* := (*fun b* $\Rightarrow$
    if b
    return Prop
    then Prp_T
    else Prp_F
) : (*Prp_Pred Boo*).

9.4.14 BOOLEAN EQUALS

(*Boo_eq b0 b1*) is the true Boolean if *b0* and *b1* are structurally equal; and the false Boolean otherwise.

Definition *Boo_eq* := (*fun b0 b1* $\Rightarrow$
    match b0, b1
    return Boo

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

Poohbist.NummSquared.Fundamentals.Naturals.Main

Poohbist.NummSquared.Fundamentals.Naturals.Main defines natural numbers, and some operators on natural numbers.

9.5.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*

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*
| *Nat_s* : (*Op_Simp 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 $Nat\_n96 := (Nat\_s\ Nat\_n95)$.

Definition $Nat\_n97 := (Nat\_s\ Nat\_n96)$.

Definition $Nat\_n98 := (Nat\_s\ Nat\_n97)$.

Definition $Nat\_n99 := (Nat\_s\ Nat\_n98)$.

Definition $Nat\_n100 := (Nat\_s\ Nat\_n99)$.

Definition $Nat\_n101 := (Nat\_s\ Nat\_n100)$.

Definition $Nat\_n102 := (Nat\_s\ Nat\_n101)$.

Definition $Nat\_n103 := (Nat\_s\ Nat\_n102)$.

Definition $Nat\_n104 := (Nat\_s\ Nat\_n103)$.

Definition $Nat\_n105 := (Nat\_s\ Nat\_n104)$.

Definition $Nat\_n106 := (Nat\_s\ Nat\_n105)$.

Definition $Nat\_n107 := (Nat\_s\ Nat\_n106)$.

Definition $Nat\_n108 := (Nat\_s\ Nat\_n107)$.

Definition $Nat\_n109 := (Nat\_s\ Nat\_n108)$.

Definition $Nat\_n110 := (Nat\_s\ Nat\_n109)$.

Definition $Nat\_n111 := (Nat\_s\ Nat\_n110)$.

Definition $Nat\_n112 := (Nat\_s\ Nat\_n111)$.

Definition $Nat\_n113 := (Nat\_s\ Nat\_n112)$.

Definition $Nat\_n114 := (Nat\_s\ Nat\_n113)$.

Definition $Nat\_n115 := (Nat\_s\ Nat\_n114)$.

Definition $Nat\_n116 := (Nat\_s\ Nat\_n115)$.

Definition $Nat\_n117 := (Nat\_s\ Nat\_n116)$.

Definition $Nat\_n118 := (Nat\_s\ Nat\_n117)$.

Definition $Nat\_n119 := (Nat\_s\ Nat\_n118)$.

Definition $Nat\_n120 := (Nat\_s\ Nat\_n119)$.

Definition $Nat\_n121 := (Nat\_s\ Nat\_n120)$.

Definition $Nat\_n122 := (Nat\_s\ Nat\_n121)$.

Definition $Nat\_n123 := (Nat\_s\ Nat\_n122)$.

Definition $Nat\_n124 := (Nat\_s\ Nat\_n123)$.

Definition $Nat\_n125 := (Nat\_s\ Nat\_n124)$.

Definition $Nat\_n126 := (Nat\_s\ Nat\_n125)$.

9.5.4 NATURAL NUMBER EQUALS

($Nat\_eq\ m0\ m1$) is the true Boolean if $m0$ and $m1$ are structurally equal; and the false Boolean otherwise.

Definition Nat_eq := (

$fix\ fp(m0 : Nat)(m1 : Nat)\{struct\ m0\} : Boo :=$

$match\ m0, m1$

$return\ Boo$

$with$

$|\ Nat\_z, Nat\_z \Rightarrow Boo\_t$

$|\ Nat\_s\ m0Pre, Nat\_s\ m1Pre \Rightarrow (fp\ m0Pre\ m1Pre)$

$|\ \_, \_ \Rightarrow Boo\_f$

end

$) : (Boo\_Pred\_Bin\_Conn\ Nat).$

9.5.5 NATURAL NUMBER ITERATE

$(Nat\_iter\ A\ f\ a\ m)$ applies f m-times starting with a.

Definition Nat_iter_Ty :=

$(\ \forall(A : Type)(f : (Op\_Simp\ A)), (Op\_Bin\ A\ Nat\ A)\ ) : Type.$

Definition Nat_iter := ($fun\ A\ f\ a \Rightarrow$

$fix\ fp(m : Nat)\{struct\ m\} : A :=$

$match\ m$

$return\ A$

$with$

$|\ Nat\_z \Rightarrow a$

$|\ Nat\_s\ mPre \Rightarrow (f\ (fp\ mPre))$

end

$) : Nat\_iter\_Ty.$

9.5.6 NATURAL NUMBER ADD

$(Nat\_add\ m\ n)$ is $m + n$.

Nat_add is somewhat similar in concept to the plus function in [28, p.234].

Definition Nat_add := ($fun\ m \Rightarrow$

$fix\ fp(n : Nat)\{struct\ n\} : Nat :=$

$match\ n$

$return\ Nat$
$with$
$|\ Nat\_z \Rightarrow m$
$|\ Nat\_s\ nPre \Rightarrow (Nat\_s\ (fp\ nPre))$
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 $Nat\_mult := (\ fun\ m \Rightarrow$
$fix\ fp(n : Nat)\{struct\ n\} : Nat :=$
$match\ n$
$return\ Nat$
$with$
$|\ Nat\_z \Rightarrow Nat\_z$
$|\ Nat\_s\ nPre \Rightarrow (Nat\_add\ (fp\ nPre)\ m)$
end
$) : (Op\_Bin\_Simp\ Nat).$

9.6 FUNDAMENTALS: NATURALS: EFFICIENT: MAIN

Poohbist.NummSquared.Fundamentals.Naturals.Efficient.Main

Poohbist.NummSquared.Fundamentals.Naturals.Efficient.Main defines efficient natural numbers, and some operators on efficient natural numbers.

9.6.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*

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

Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main*.

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

Poohbist.NummSquared.Fundamentals.Optionals.Main

Poohbist.NummSquared.Fundamentals.Optionals.Main defines optionals, and some operators on optionals.

9.8.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*

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*:*Type* whereas *option*:*Set*.
Inductive *Optional*(*A* : *Type*) : *Type* :=
| *Optional_none* : (*Optional A*)
| *Optional_one* : (*Op A* (*Optional A*)).

9.8.3 OPTIONAL RELATED TO

(*Optional_rel A0 A1 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 *Optional_select_Ty* :=
(∀
(*A* : *Type*)
(*B* : *Type*)
(*selectA* : (*Op A* (*Optional B*))),
(*Op* (*Optional A*) (*Optional B*))
) : *Type*.

Definition *Optional_select* := (*fun A B selectA o* ⇒
match o
return (*Optional B*)
with
| *Optional_none* ⇒ (*Optional_none B*)
| *Optional_one a* ⇒ (*selectA a*)
end
) : *Optional_select_Ty*.

9.8.9 OPTIONAL SELECT, TO ELEMENT

(*Optional_select_toElem A B selectA o*) is (*Optional_select A B* (*Optional_Op_one A B selectA*) *o*).

Definition *Optional_select_toElem_Ty* :=
(∀
(*A* : *Type*)
(*B* : *Type*)
(*selectA* : (*Op A B*)),
(*Op* (*Optional A*) (*Optional B*))
) : *Type*.

Definition *Optional_select_toElem* := (*fun A B selectA o* ⇒
(*Optional_select A B* (*Optional_Op_one A B selectA*) *o*)
) : *Optional_select_toElem_Ty*.

9.9 FUNDAMENTALS: BOOLEANS: AND OPTIONALS

Poohbist.NummSquared.Fundamentals.Booleans.AndOptionals

Poohbist.NummSquared.Fundamentals.Booleans.AndOptionals defines some operators relating Booleans and optionals.

9.9.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Units.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main.*

9.9.2 BOOLEAN TO OPTIONAL

(*Boo_to_Optional A b a*) is the one optional *A* containing *a* if *b*; and the none optional *A* otherwise.

Definition *Boo_to_Optional_Ty* :=
(∀
(*A* : *Type*),
(*Op_Bin Boo A* (*Optional A*))
) : *Type*.

Definition *Boo_to_Optional* := (*fun A b a* ⇒
if b
return (*Optional A*)
then (*Optional_one A a*)
else (*Optional_none A*)
) : *Boo_to_Optional_Ty*.

9.9.3 THE BOOLEAN OPTIONAL OPERATOR

(*Bool_Op_Optional A predA*) is the operator from *A* to an optional *A* mapping *a* : *A* onto (*Boo_to_Optional A* (*predA a*) *a*).

Definition *Boo_Op_Optional_Ty* :=

$(\forall$

$(A : Type)$

$(predA : (Boo\_Pred\ A)),$

$(Op\ A\ (Optional\ A))$

$) : Type.$

Definition $Boo\_Op\_Optional :=$

$(\ fun\ A\ predA\ a \Rightarrow (Boo\_to\_Optional\ A\ (predA\ a)\ a)\ )$

$: Boo\_Op\_Optional\_Ty.$

9.10 FUNDAMENTALS: CHOICES: MAIN

Poohbist.NummSquared.Fundamentals.Choices.Main

Poohbist.NummSquared.Fundamentals.Choices.Main defines choices, and some operators on choices.

9.10.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Units.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main.*

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

Choice is defined in the same way as *sum* in *Coq.Init.Datatypes*, except that *Choice*:*Type* whereas *sum*:*Set*.

Inductive $Choice (F : Type) (S : Type) : Type :=$

$|\ Choice\_first : (Op\ F\ (Choice\ F\ S))$

| *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* :=

    (∀
        (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))
) : Type.

Definition *Choice_rel* := (*fun F0 S0 F1 S1 relF01 relS01 c0 c1* ⇒

    match c0, c1
    return Boo
    with
    | Choice_first f0, Choice_first f1 ⇒ (relF01 f0 f1)
    | Choice_second s0, Choice_second s1 ⇒ (relS01 s0 s1)
    | _, _ ⇒ Boo_f
    end
) : Choice_rel_Ty.

9.10.4 CHOICE RELATED TO, CONNECTIVE

(*Choice_rel_conn F S relF relS c0 c1*) is (*Choice_rel F S F S relF relS c0 c1*).

Definition *Choice_rel_conn_Ty* :=

    (∀

(*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* :=
(*fun F S relF relS c0 c1* ⇒ (*Choice_rel F S 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_Optional_Ty* :=
(∀
(*A* : *Type*),
(*Op* (*Choice A Uni*) (*Optional A*))
) : *Type*.

Definition *Choice_to_Optional* := (*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_Optional_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* :=
(∀(*A* : *Type*), (*Op* (*Choice A A*) *A*)) : *Type*.

Definition *Choice_merge* := (*fun A c* ⇒

match c
return A
with
| *Choice_first a* $\Rightarrow$ *a*
| *Choice_second a* $\Rightarrow$ *a*
end
) : *Choice_merge_Ty*.

9.11 FUNDAMENTALS: PAIRS: MAIN

Poohbist.NummSquared.Fundamentals.Pairs.Main

Poohbist.NummSquared.Fundamentals.Pairs.Main defines pairs, triples, quadruples, and some operators on pairs, triples and quadruples.

9.11.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main*.

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* := *Pair_ctor* {
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* :=
  (∀
    (*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*))
  return Boo
  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* :=
  (∀
    (*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*))

: *Pair_rel_conn_Ty*.

9.11.5 TRIPLES

A triple *L0*, *L1*, *R1* is a (*Pair L0* (*Pair L1 R1*)).

Definition *Trip_Ty* := (∀(*L0* : *Type*)(*L1* : *Type*)(*R1* : *Type*), *Type*) : *Type*.

Definition *Trip* := (*fun L0 L1 R1* ⇒ (*Pair L0* (*Pair L1 R1*))) : *Trip_Ty*.

9.11.6 TRIPLE LEFT 0

(*Trip_left0 L0 L1 R1 t*) is the left of *t*.

Definition *Trip_left0_Ty* :=

(∀

(*L0* : *Type*)

(*L1* : *Type*)

(*R1* : *Type*),

(*Op* (*Trip L0 L1 R1*) *L0*)

) : *Type*.

Definition *Trip_left0* :=

(*fun L0 L1 R1 t* ⇒ (*Pair_left L0* (*Pair L1 R1*) *t*)) : *Trip_left0_Ty*.

9.11.7 TRIPLE RIGHT 0

(*Trip_right0 L0 L1 R1 t*) is the right of *t*.

Definition *Trip_right0_Ty* :=

(∀

(*L0* : *Type*)

(*L1* : *Type*)

(*R1* : *Type*),

(*Op* (*Trip L0 L1 R1*) (*Pair L1 R1*))

) : *Type*.

Definition *Trip_right0* :=

(*fun L0 L1 R1 t* ⇒ (*Pair_right L0* (*Pair L1 R1*) *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_Ty* :=
(∀
(*L0* : *Type*)
(*L1* : *Type*)
(*R1* : *Type*),
(*Op* (*Trip L0 L1 R1*) *L1*)
) : *Type*.

Definition *Trip_left1* :=
(*fun L0 L1 R1 t* ⇒ (*Pair_left L1 R1* (*Trip_right0 L0 L1 R1 t*)))
: *Trip_left1_Ty*.

9.11.9 TRIPLE RIGHT 1

(*Trip_right1 L0 L1 R1 t*) is the right-right of *t*.

Definition *Trip_right1_Ty* :=
(∀
(*L0* : *Type*)
(*L1* : *Type*)
(*R1* : *Type*),
(*Op* (*Trip L0 L1 R1*) *R1*)
) : *Type*.

Definition *Trip_right1* :=
(*fun L0 L1 R1 t* ⇒ (*Pair_right L1 R1* (*Trip_right0 L0 L1 R1 t*)))
: *Trip_right1_Ty*.

9.11.10 QUADRUPLES

A quadruple *L0*, *L1*, *L2*, *R2* is a (*Pair L0* (*Trip L1 L2 R2*)).

Definition *Quad_Ty* :=
(∀

$(L0 : Type)$

$(L1 : Type)$

$(L2 : Type)$

$(R2 : Type),$

$Type$

$) : Type.$

Definition $Quad := (\ fun\ L0\ L1\ L2\ R2 \Rightarrow$

$(Pair\ L0\ (Trip\ L1\ L2\ R2))$

$) : Quad\_Ty.$

9.11.11 QUADRUPLE LEFT 0

$(Quad\_left0\ L0\ L1\ L2\ R2\ t)$ is the left of q.

Definition $Quad\_left0\_Ty :=$

$(\forall$

$(L0 : Type)$

$(L1 : Type)$

$(L2 : Type)$

$(R2 : Type),$

$(Op\ (Quad\ L0\ L1\ L2\ R2)\ L0)$

$) : Type.$

Definition $Quad\_left0 :=$

$(\ fun\ L0\ L1\ L2\ R2\ q \Rightarrow (Pair\_left\ L0\ (Trip\ L1\ L2\ R2)\ q)\ )$

$: Quad\_left0\_Ty.$

9.11.12 QUADRUPLE RIGHT 0

$(Quad\_right0\ L0\ L1\ L2\ R2\ t)$ is the right of q.

Definition $Quad\_right0\_Ty :=$

$(\forall$

$(L0 : Type)$

$(L1 : Type)$

$(L2 : Type)$

$(R2 : Type),$

(*Op* (*Quad L0 L1 L2 R2*) (*Trip L1 L2 R2*))

) : *Type*.

Definition *Quad_right0* :=

(*fun L0 L1 L2 R2 q* ⇒ (*Pair_right L0* (*Trip L1 L2 R2*) *q*))

: *Quad_right0_Ty*.

9.11.13 QUADRUPLE LEFT 1

(*Quad_left1 L0 L1 L2 R2 t*) is the right-left of *q*.

Definition *Quad_left1_Ty* :=

(∀

(*L0* : *Type*)

(*L1* : *Type*)

(*L2* : *Type*)

(*R2* : *Type*),

(*Op* (*Quad L0 L1 L2 R2*) *L1*)

) : *Type*.

Definition *Quad_left1* := (*fun L0 L1 L2 R2 q* ⇒

(*Trip_left0 L1 L2 R2* (*Quad_right0 L0 L1 L2 R2 q*))

) : *Quad_left1_Ty*.

9.11.14 QUADRUPLE LEFT 2

(*Quad_left2 L0 L1 L2 R2 t*) is the right-right-left of *q*.

Definition *Quad_left2_Ty* :=

(∀

(*L0* : *Type*)

(*L1* : *Type*)

(*L2* : *Type*)

(*R2* : *Type*),

(*Op* (*Quad L0 L1 L2 R2*) *L2*)

) : *Type*.

Definition *Quad_left2* := (*fun L0 L1 L2 R2 q* ⇒

(*Trip_left1 L1 L2 R2* (*Quad_right0 L0 L1 L2 R2 q*))

) : *Quad_left2_Ty*.

9.11.15 QUADRUPLE RIGHT 2

(*Quad_right2 L0 L1 L2 R2 t*) is the right-right-right of *q*.

Definition *Quad_right2_Ty* :=
(∀
(*L0* : *Type*)
(*L1* : *Type*)
(*L2* : *Type*)
(*R2* : *Type*),
(*Op* (*Quad L0 L1 L2 R2*) *R2*)
) : *Type*.

Definition *Quad_right2* := (*fun L0 L1 L2 R2 q* ⇒
(*Trip_right1 L1 L2 R2* (*Quad_right0 L0 L1 L2 R2 q*))
) : *Quad_right2_Ty*.

9.12 FUNDAMENTALS: LISTS: MAIN

Poohbist.NummSquared.Fundamentals.Lists.Main

Poohbist.NummSquared.Fundamentals.Lists.Main defines lists, non-empty lists, and some operators on lists and non-empty lists.

9.12.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main*.

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*

Lis is defined in the same way as *list* in *Coq.Lists.List*, except that *Lis*:*Type* whereas *list*:*Set*.

Inductive *Lis*(*A* : *Type*) : *Type* :=
| *Lis_nil* : (*Lis A*)
| *Lis_cons* : (*Op_Bin A* (*Lis A*) (*Lis A*)).

9.12.3 LIST NOTATION

A , *a0* , .. , *a1* is the list *A* containing the elements *a0*, ..., *a1*. *a0*, ..., *a1* must contain at least one element.

A , *a0* , .. , *a1* is defined in the same way as the list notation in [8, section 11.1.11], except that *A* , *a0* , .. , *a1* explicitly includes *A*.

Notation "[A , a0 , .. , a1]" :=
(*Lis_cons A a0* .. (*Lis_cons A a1* (*Lis_nil A*)) ..) : *Lis_scope*.
Open Scope Lis_scope.

9.12.4 LIST RELATED TO

(*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 (*rel01 a0 a1*); and the false Boolean otherwise.

Definition *Lis_rel_Ty* :=
(∀
(*A0* : *Type*)
(*A1* : *Type*)
(*rel01* : (*Boo_Pred_Bin A0 A1*)),
(*Boo_Pred_Bin* (*Lis A0*) (*Lis A1*))
) : *Type*.

Definition *Lis_rel* := (*fun A0 A1 rel01* ⇒

fix fp(*l0* : (*Lis A0*))(*l1* : (*Lis A1*)){*struct l0*} : *Boo* :=
match l0, l1
return Boo
with
| *Lis_nil, Lis_nil* ⇒ *Boo_t*
| *Lis_cons l0Head l0Rest, Lis_cons l1Head l1Rest* ⇒
if (*rel01 l0Head l1Head*)
return Boo
then (*fp l0Rest l1Rest*)
else Boo_f
| _, _ ⇒ *Boo_f*
end
) : *Lis_rel_Ty*.

9.12.5 LIST RELATED TO, CONNECTIVE

(*Lis_rel_conn A relA l0 l1*) is (*Lis_rel A A relA l0 l1*).
Definition *Lis_rel_conn_Ty* :=
(∀
(*A* : *Type*)
(*relA* : (*Boo_Pred_Bin_Conn A*)),
(*Boo_Pred_Bin_Conn* (*Lis A*))
) : *Type*.

Definition *Lis_rel_conn* :=
(*fun A relA l0 l1* ⇒ (*Lis_rel A A relA l0 l1*)) : *Lis_rel_conn_Ty*.

9.12.6 LIST HEAD

(*Lis_head A l*) is the none optional *A* if *l* is the nil list *A*; and the one optional *A* containing *lHead* if *l* is the cons list *A* of *lHead* and *lRest*.
Definition *Lis_head_Ty* :=
(∀(*A* : *Type*), (*Op* (*Lis A*) (*Optional A*))) : *Type*.

Definition *Lis_head* := (*fun A l* ⇒
match l

return (*Optional A*)
with
| *Lis_nil* ⇒ (*Optional_none A*)
| *Lis_cons lHead lRest* ⇒ (*Optional_one A lHead*)
end
) : *Lis_head_Ty*.

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_Ty* :=
(∀(*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_Ty*.

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_Ty* :=
(∀(*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

(*Lis_empty A l*) is (*Boo_not* (*Lis_nonEmpty A l*)).

Definition *Lis_empty_Ty* :=

(∀(*A* : *Type*), (*Boo_Pred* (*Lis A*))) : *Type*.

Definition *Lis_empty* := (*fun A l* ⇒

(*Boo_not* (*Lis_nonEmpty A l*))

) : *Lis_empty_Ty*.

9.12.10 LIST CONCATENATE

(*Lis_cat A l0 l1*) is the list *A* containing the elements in *l0* followed by the elements in *l1*.

Definition *Lis_cat_Ty* :=

(∀(*A* : *Type*), (*Op_Bin_Simp* (*Lis A*))) : *Type*.

Definition *Lis_cat* := (*fun A* ⇒

fix fp(*l0* : (*Lis A*))(*l1* : (*Lis A*)){*struct l0*} : (*Lis A*) :=

match l0

return (*Lis A*)

with

| *Lis_nil* ⇒ *l1*

| *Lis_cons l0Head l0Rest* ⇒ (*Lis_cons A l0Head* (*fp l0Rest l1*))

end

) : *Lis_cat_Ty*.

9.12.11 LIST APPEND

(*Lis_append A l a*) is (*Lis_cat A l* [*A*, *a*]).

Definition *Lis_append_Ty* :=

(∀(*A* : *Type*), (*Op_Bin* (*Lis A*) *A* (*Lis A*))) : *Type*.

Definition *Lis_append* :=

(*fun A l a* ⇒ (*Lis_cat A l* [A, a])) : *Lis_append_Ty*.

9.12.12 THE LIST SINGLETON OPERATOR

(*Lis_Op_singleton A B opA*) is the operator from *A* to a list *B* mapping *a* : *A* onto [*B*, (*opA a*)].

Definition *Lis_Op_singleton_Ty* :=

(∀

(*A* : *Type*)

(*B* : *Type*)

(*opA* : (*Op A B*)),

(*Op A* (*Lis B*))

) : *Type*.

Definition *Lis_Op_singleton* :=

(*fun A B opA a* ⇒ [B, (opA a)]) : *Lis_Op_singleton_Ty*.

9.12.13 THE LIST SINGLETON BINARY OPERATOR

(*Lis_Op_singleton_bin A0 A1 B opA*) is the binary operator from *A0*, *A1* to a list *B* mapping *a0* : *A0*, *a1* : *A1* onto [*B*, (*opA a0 a1*)].

Definition *Lis_Op_singleton_bin_Ty* :=

(∀

(*A0* : *Type*)

(*A1* : *Type*)

(*B* : *Type*)

(*opA* : (*Op_Bin A0 A1 B*)),

(*Op_Bin A0 A1* (*Lis B*))

) : *Type*.

Definition *Lis_Op_singleton_bin* :=

(*fun A0 A1 B opA a0 a1* ⇒ [B, (opA a0 a1)]) : *Lis_Op_singleton_bin_Ty*.

9.12.14 THE LIST PREFIX OPERATOR

(*Lis_Op_prefix A B opA prefix*) is the operator from *A* to a list *B* mapping *a* : *A* onto

$(\mathit{Lis\_cat}\ B\ \mathit{prefix}\ (\mathit{opA}\ a))$.

Definition $\mathit{Lis\_Op\_prefix\_Ty} :=$

$($ $\forall$

$(A : \mathit{Type})$

$(B : \mathit{Type})$

$(\mathit{opA} : (\mathit{Op}\ A\ (\mathit{Lis}\ B)))$

$(\mathit{prefix} : (\mathit{Lis}\ B))$,

$(\mathit{Op}\ A\ (\mathit{Lis}\ B))$

$) : \mathit{Type}$.

Definition $\mathit{Lis\_Op\_prefix} := (\ \mathit{fun}\ A\ B\ \mathit{opA}\ \mathit{prefix}\ a \Rightarrow$

$(\mathit{Lis\_cat}\ B\ \mathit{prefix}\ (\mathit{opA}\ a))$

$) : \mathit{Lis\_Op\_prefix\_Ty}$.

9.12.15 THE LIST SUFFIX OPERATOR

$(\mathit{Lis\_Op\_suffix}\ A\ B\ \mathit{opA}\ \mathit{suffix})$ is the operator from A to a list B mapping $a : A$ onto $(\mathit{Lis\_cat}\ B\ (\mathit{opA}\ a)\ \mathit{suffix})$.

Definition $\mathit{Lis\_Op\_suffix\_Ty} :=$

$($ $\forall$

$(A : \mathit{Type})$

$(B : \mathit{Type})$

$(\mathit{opA} : (\mathit{Op}\ A\ (\mathit{Lis}\ B)))$

$(\mathit{suffix} : (\mathit{Lis}\ B))$,

$(\mathit{Op}\ A\ (\mathit{Lis}\ B))$

$) : \mathit{Type}$.

Definition $\mathit{Lis\_Op\_suffix} := (\ \mathit{fun}\ A\ B\ \mathit{opA}\ \mathit{suffix}\ a \Rightarrow$

$(\mathit{Lis\_cat}\ B\ (\mathit{opA}\ a)\ \mathit{suffix})$

$) : \mathit{Lis\_Op\_suffix\_Ty}$.

9.12.16 LIST GENERATE

$(\mathit{Lis\_generate}\ A\ \mathit{genA}\ m)$ is the list A whose elements are obtained by conatenating the following lists A: $(\mathit{genA}\ \mathit{Nat\_z})$, ..., $(\mathit{genA}\ m)$.

Definition $\mathit{Lis\_generate\_Ty} :=$

($\forall$

$(A : Type)$

$(genA : (Op\ Nat\ (Lis\ A)))$,

$(Op\ Nat\ (Lis\ A))$

) : $Type$.

Definition $Lis\_generate$:= ($fun\ A\ genA \Rightarrow$

$fix\ fp(m : Nat)\{struct\ m\} : (Lis\ A) :=$

$match\ m$

$return\ (Lis\ A)$

$with$

$|\ Nat\_z \Rightarrow (genA\ Nat\_z)$

$|\ Nat\_s\ mPre \Rightarrow (Lis\_cat\ A\ (fp\ mPre)\ (genA\ m))$

end

) : $Lis\_generate\_Ty$.

9.12.17 LIST GENERATE, TO ELEMENT

$(Lis\_generate\_toElem\ A\ genA\ m)$ is $(Lis\_generate\ A\ (Lis\_Op\_singleton\ Nat\ A\ genA)\ m)$.

Definition $Lis\_generate\_toElem\_Ty$:=

($\forall$

$(A : Type)$

$(genA : (Op\ Nat\ A))$,

$(Op\ Nat\ (Lis\ A))$

) : $Type$.

Definition $Lis\_generate\_toElem$:= ($fun\ A\ genA\ m \Rightarrow$

$(Lis\_generate\ A\ (Lis\_Op\_singleton\ Nat\ A\ genA)\ m)$

) : $Lis\_generate\_toElem\_Ty$.

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

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* (*Lis_Ne_head A0 l0*) (*Lis_Ne_head A1 l1*)) and (*Lis_rel A0 A1 rel01* (*Lis_Ne_rest A0 l0*) (*Lis_Ne_rest A1 l1*)); and the false Boolean otherwise.

```
Definition Lis_Ne_rel_Ty :=
    ( ∀
        (A0 : Type)
        (A1 : Type)
        (rel01 : (Boo_Pred_Bin A0 A1)),
        (Boo_Pred_Bin (Lis_Ne A0) (Lis_Ne A1))
    ) : Type.
Definition Lis_Ne_rel := ( fun A0 A1 rel01 l0 l1 ⇒
    if (rel01 (Lis_Ne_head A0 l0) (Lis_Ne_head A1 l1))
    return Boo
    then
        (Lis_rel A0 A1 rel01 (Lis_Ne_rest A0 l0) (Lis_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 (*Lis_Ne_rel A A relA l0 l1*).

```
Definition Lis_Ne_rel_conn_Ty :=
    ( ∀
        (A : Type)
        (relA : (Boo_Pred_Bin_Conn A)),
```

(*Boo_Pred_Bin_Conn* (*Lis_Ne A*))

) : *Type*.

Definition *Lis_Ne_rel_conn* :=

(*fun A relA l0 l1* ⇒ (*Lis_Ne_rel A A relA l0 l1*))

: *Lis_Ne_rel_conn_Ty*.

9.12.21 NON-EMPTY LIST SINGLETON

(*List_Ne_singleton A a*) is the non-empty list *A* containing just *a*.

Definition *Lis_Ne_singleton_Ty* :=

(∀(*A* : *Type*), (*Op A* (*Lis_Ne A*))) : *Type*.

Definition *Lis_Ne_singleton* := (*fun A a* ⇒

(*Lis_Ne_ctor A a* (*Lis_nil A*))

) : *Lis_Ne_singleton_Ty*.

9.12.22 NON-EMPTY LIST TO LIST

(*List_Ne_to_Lis A l*) is the list *A* containing the same elements as *l*.

Definition *Lis_Ne_to_Lis_Ty* :=

(∀(*A* : *Type*), (*Op* (*Lis_Ne A*) (*Lis A*))) : *Type*.

Definition *Lis_Ne_to_Lis* := (*fun A l* ⇒

(*Lis_cons A* (*Lis_Ne_head A l*) (*Lis_Ne_rest A l*))

) : *Lis_Ne_to_Lis_Ty*.

9.12.23 THE NON-EMPTY LIST HEAD OPERATOR

(*Lis_Ne_Op_head A B opA*) is the operator from a non-empty list *A* to *B* mapping *l* : (*Lis_Ne A*) onto (*opA* (*Lis_Ne_head A l*)).

Definition *Lis_Ne_Op_head_Ty* :=

(∀

(*A* : *Type*)

(*B* : *Type*)

(*opA* : (*Op A B*)),

$(Op\ (Lis\_Ne\ A)\ B)$

$)$: $Type$.

Definition $Lis\_Ne\_Op\_head$:=

$(\ fun\ A\ B\ opA\ l \Rightarrow (opA\ (Lis\_Ne\_head\ A\ l))\ )$: $Lis\_Ne\_Op\_head\_Ty$.

9.12.24 LIST TO NON-EMPTY LIST

$(Lis\_to\_Lis\_Ne\ A\ l)$ 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 $Lis\_to\_Lis\_Ne\_Ty$:=

$(\ \forall (A : Type), (Op\ (Lis\ A)\ (Optional\ (Lis\_Ne\ A)))\ )$: $Type$.

Definition $Lis\_to\_Lis\_Ne$:= $(\ fun\ A\ l \Rightarrow$

$match\ l$

$return\ (Optional\ (Lis\_Ne\ A))$

$with$

$|\ Lis\_nil \Rightarrow (Optional\_none\ (Lis\_Ne\ A))$

$|\ Lis\_cons\ lHead\ lRest \Rightarrow$

$(Optional\_one\ (Lis\_Ne\ A)\ (Lis\_Ne\_ctor\ A\ lHead\ lRest))$

end

$)$: $Lis\_to\_Lis\_Ne\_Ty$.

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 $Lis\_P2 (A : Type) : Type$:= Lis_P2_ctor {

$Lis\_P2\_head : A$;

$Lis\_P2\_rest : (Lis\_Ne\ A)$

}.

9.13 FUNDAMENTALS: OPTIONALS: AND LISTS

Poohbist.NummSquared.Fundamentals.Optionals.AndLists

Poohbist.NummSquared.Fundamentals.Optionals.AndLists defines some operators relating optionals and lists.

9.13.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main*.

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.Fundamentals.Booleans.AndLists

Poohbist.NummSquared.Fundamentals.Booleans.AndLists defines some operators relating Booleans and lists.

9.14.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.AndOptionals.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.AndLists.*

9.14.2 BOOLEAN TO LIST

(*Boo_to_Lis A b a*) is (*Optional_to_Lis A* (*Boo_to_Optional A b a*)).

Definition *Boo_to_Lis_Ty* :=
 (∀
  (*A* : *Type*),
  (*Op_Bin Boo A* (*Lis A*))
) : *Type*.

Definition *Boo_to_Lis* := (*fun A b a* ⇒
 (*Optional_to_Lis A* (*Boo_to_Optional A b a*))
) : *Boo_to_Lis_Ty*.

9.14.3 THE BOOLEAN LIST OPERATOR

(*Boo_Op_Lis A predA*) is the operator from *A* to an list *A* mapping *a* : *A* onto (*Boo_to_Lis A* (*predA a*) *a*).

Definition *Boo_Op_Lis_Ty* :=
 (∀
  (*A* : *Type*)
  (*predA* : (*Boo_Pred A*)),
  (*Op A* (*Lis A*))
) : *Type*.

Definition *Boo_Op_Lis* :=
 (*fun A predA a* ⇒ (*Boo_to_Lis A* (*predA a*) *a*)) : *Boo_Op_Lis_Ty*.

9.15 FUNDAMENTALS: NATURALS: AND LISTS

Poohbist.NummSquared.Fundamentals.Naturals.AndLists

Poohbist.NummSquared.Fundamentals.Naturals.AndLists defines natural number lists, and some operators on natural number lists.

9.15.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main*.

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

Poohbist.NummSquared.Fundamentals.Naturals.Efficient.AndLists

Poohbist.NummSquared.Fundamentals.Naturals.Efficient.AndLists defines efficient natural number lists, and some operators on efficient natural number lists.

9.16.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Efficient.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main.*

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*)
) : (*Boo_Pred_Bin_Conn Nat_Eff_Lis*).

9.17 FUNDAMENTALS: PAIRS: AND LISTS

Poohbist.NummSquared.Fundamentals.Pairs.AndLists

Poohbist.NummSquared.Fundamentals.Pairs.AndLists defines some operators relating pairs and lists.

9.17.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Pairs.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main.*

9.17.2 PAIR OF HEAD AND REST TO NON-EMPTY LIST

(*Pair_headRest_to_Lis_Ne A p*) is (*Lis_Ne_ctor A* (*Pair_left A* (*Lis A*) *p*) (*Pair_right A*

(*Lis A*) *p*)).
Definition *Pair_headRest_to_Lis_Ne_Ty* :=
(∀ (*A* : *Type*), (*Op* (*Pair A* (*Lis A*)) (*Lis_Ne A*)))
: *Type*.

Definition *Pair_headRest_to_Lis_Ne* := (*fun A p* ⇒
(*Lis_Ne_ctor A* (*Pair_left A* (*Lis A*) *p*) (*Pair_right A* (*Lis A*) *p*))
) : *Pair_headRest_to_Lis_Ne_Ty*.

9.18 FUNDAMENTALS: LISTS: SELECT

Poohbist.NummSquared.Fundamentals.Lists.Select

Poohbist.NummSquared.Fundamentals.Lists.Select defines some selection operators on lists.

9.18.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.AndLists.*

9.18.2 LIST SELECT

(*Lis_select A B selectSuffix l*) is the list *B* obtained by first applying *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 (*Lis_select A B selectSuffix l*) is the nil list *B*.

Lis_select is somewhat similar in concept to the LISP mapcon function (see [27, chapter 12]).
Definition *Lis_select_Ty* :=
(∀

$(A : Type)$
$(B : Type)$
$(selectSuffix : (Op\ (Lis\_Ne\ A)\ (Lis\ B))),$
$(Op\ (Lis\ A)\ (Lis\ B))$
$) : Type.$

Definition $Lis\_select := (\ fun\ A\ B\ selectSuffix \Rightarrow$
$fix\ fp(l : (Lis\ A))\{struct\ l\} : (Lis\ B) :=$
$match\ l$
$return\ (Lis\ B)$
$with$
$|\ Lis\_nil \Rightarrow (Lis\_nil\ B)$
$|\ Lis\_cons\ lHead\ lRest \Rightarrow$
$(Lis\_cat$
B
$(selectSuffix\ (Lis\_Ne\_ctor\ A\ lHead\ lRest))$
$(fp\ lRest)$
$)$
end
$) : Lis\_select\_Ty.$

9.18.3 LIST SELECT, SIMPLE

$(Lis\_select\_simp\ A\ selectSuffix\ l)$ is $(Lis\_select\ A\ A\ selectSuffix\ l)$.

Definition $Lis\_select\_simp\_Ty :=$
$(\ \forall$
$(A : Type)$
$(selectSuffix : (Op\ (Lis\_Ne\ A)\ (Lis\ A))),$
$(Op\_Simp\ (Lis\ A))$
$) : Type.$

Definition $Lis\_select\_simp := (\ fun\ A\ selectSuffix\ l \Rightarrow$
$(Lis\_select\ A\ A\ selectSuffix\ l)$
$) : Lis\_select\_simp\_Ty.$

9.18.4 LIST SELECT, ITERATE

(*Lis_select_iter A selectSuffix l m*) is (*Nat_iter* (*Lis A*) (*Lis_select_simp A selectSuffix*) *l m*).

Definition *Lis_select_iter_Ty* :=
    (∀
        (*A* : *Type*)
        (*selectSuffix* : (*Op* (*Lis_Ne A*) (*Lis A*))),
        (*Op_Bin* (*Lis A*) *Nat* (*Lis A*))
) : *Type*.

Definition *Lis_select_iter* := (*fun A selectSuffix l m* ⇒
    (*Nat_iter* (*Lis A*) (*Lis_select_simp A selectSuffix*) *l m*)
) : *Lis_select_iter_Ty*.

9.18.5 LIST SELECT, TO ELEMENT

(*Lis_select_toElem A B selectSuffix l*) is (*Lis_select A B* (*Lis_Op_singleton* (*Lis_Ne A*) *B selectSuffix*) *l*).

Lis_select_toElem is somewhat similar in concept to the LISP maplist function (see [27, chapter 12]).

Definition *Lis_select_toElem_Ty* :=
    (∀
        (*A* : *Type*)
        (*B* : *Type*)
        (*selectSuffix* : (*Op* (*Lis_Ne A*) *B*)),
        (*Op* (*Lis A*) (*Lis B*))
) : *Type*.

Definition *Lis_select_toElem* := (*fun A B selectSuffix l* ⇒
    (*Lis_select A B* (*Lis_Op_singleton* (*Lis_Ne A*) *B selectSuffix*) *l*)
) : *Lis_select_toElem_Ty*.

9.18.6 LIST SELECT, TO ELEMENT, SIMPLE

(*Lis_select_toElem_simp A selectSuffix l*) is (*Lis_select_toElem A A selectSuffix l*).

Definition *Lis_select_toElem_simp_Ty* :=

(∀

(*A* : *Type*)

(*selectSuffix* : (*Op* (*Lis_Ne A*) *A*)),

(*Op_Simp* (*Lis A*))

) : *Type*.

Definition *Lis_select_toElem_simp* := (*fun A selectSuffix l* ⇒

(*Lis_select_toElem A A selectSuffix l*)

) : *Lis_select_toElem_simp_Ty*.

9.18.7 LIST SELECT, TO ELEMENT, ITERATE

(*Lis_select_toElem_iter A selectSuffix l m*) is (*Nat_iter* (*Lis A*) (*Lis_select_toElem_simp A selectSuffix*) *l m*).

Definition *Lis_select_toElem_iter_Ty* :=

(∀

(*A* : *Type*)

(*selectSuffix* : (*Op* (*Lis_Ne A*) *A*)),

(*Op_Bin* (*Lis A*) *Nat* (*Lis A*))

) : *Type*.

Definition *Lis_select_toElem_iter* := (*fun A selectSuffix l m* ⇒

(*Nat_iter* (*Lis A*) (*Lis_select_toElem_simp A selectSuffix*) *l m*)

) : *Lis_select_toElem_iter_Ty*.

9.18.8 LIST SELECT, BY ELEMENT

(*Lis_select_byElem A B selectA l*) is (*Lis_select A B* (*Lis_Ne_Op_head A* (*Lis B*) *selectA*) *l*).

(*Lis_select_byElem A B selectA l*) is the list *B* obtained by first applying *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 (*Lis_select_byElem A B selectA l*) is the nil list *B*.

Lis_select_byElem is somewhat similar in concept to the LISP mapcan function (see [27, chapter 12]).

Definition *Lis_select_byElem_Ty* :=
(∀
(*A* : *Type*)
(*B* : *Type*)
(*selectA* : (*Op A* (*Lis B*))),
(*Op* (*Lis A*) (*Lis B*))
) : *Type*.

Definition *Lis_select_byElem* := (*fun A B selectA l* ⇒
(*Lis_select A B* (*Lis_Ne_Op_head A* (*Lis B*) *selectA*) *l*)
) : *Lis_select_byElem_Ty*.

9.18.9 LIST SELECT, BY ELEMENT, SIMPLE

(*Lis_select_byElem_simp A selectA l*) is (*Lis_select_byElem A A selectA l*).
Definition *Lis_select_byElem_simp_Ty* :=
(∀
(*A* : *Type*)
(*selectA* : (*Op A* (*Lis A*))),
(*Op_Simp* (*Lis A*))
) : *Type*.

Definition *Lis_select_byElem_simp* := (*fun A selectA l* ⇒
(*Lis_select_byElem A A selectA l*)
) : *Lis_select_byElem_simp_Ty*.

9.18.10 LIST SELECT, BY ELEMENT, ITERATE

(*Lis_select_byElem_iter A selectA l m*) is (*Nat_iter* (*Lis A*) (*Lis_select_byElem_simp A selectA*) *l m*).
Definition *Lis_select_byElem_iter_Ty* :=
(∀
(*A* : *Type*)
(*selectA* : (*Op A* (*Lis A*))),
(*Op_Bin* (*Lis A*) *Nat* (*Lis A*))
) : *Type*.

Definition $Lis\_select\_byElem\_iter :=$ ($fun\ A\ selectA\ l\ m \Rightarrow$
    (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 $Lis\_select\_byElem\_intro\_Ty :=$
    (∀
        (A : $Type$)
        (B : $Type$)
        ($selectA$: ($Op\ A$ ($Lis\ B$)))
        ($intro$: ($Lis\ B$)),
        (Op ($Lis\ A$) ($Lis\ B$))
) : $Type$.

Definition $Lis\_select\_byElem\_intro :=$ ($fun\ A\ B\ selectA\ intro\ l \Rightarrow$
    ($Lis\_select\_byElem\ A\ B$ ($Lis\_Op\_prefix\ A\ B\ selectA\ intro$) 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 $Lis\_select\_byElem\_ter\_Ty :=$
    (∀
        (A : $Type$)
        (B : $Type$)
        ($selectA$: ($Op\ A$ ($Lis\ B$)))
        (ter : ($Lis\ B$)),
        (Op ($Lis\ A$) ($Lis\ B$))
) : $Type$.

Definition $Lis\_select\_byElem\_ter :=$ ($fun\ A\ B\ selectA\ ter\ l \Rightarrow$
    ($Lis\_select\_byElem\ A\ B$ ($Lis\_Op\_suffix\ A\ B\ selectA\ ter$) l)

) : *Lis_select_byElem_ter_Ty*.

9.18.13 LIST SELECT, BY ELEMENT, SEPARATED

(*Lis_select_byElem_sep A B selectA sep l*) is the nil list *B* if *l* is the nil list *A*; and the list *B* obtained by concatenating (*selectA lHead*) and (*Lis_select_byElem_intro A B selectA sep lRest*) if *l* is the cons list *A* of *lHead* and *lRest*.

Definition *Lis_select_byElem_sep_Ty* :=

```
( ∀
        (A : Type)
        (B : Type)
        (selectA : (Op A (Lis B)))
        (sep : (Lis B)),
        (Op (Lis A) (Lis B))
) : Type.
```

Definition *Lis_select_byElem_sep* := (*fun A B selectA sep l* ⇒

```
match l
return (Lis B)
with
| Lis_nil ⇒ (Lis_nil B)
| Lis_cons lHead lRest ⇒
            (Lis_cat
                B
                (selectA lHead)
                (Lis_select_byElem_intro A B selectA sep lRest)
            )
end
) : Lis_select_byElem_sep_Ty.
```

9.18.14 LIST SELECT, BY ELEMENT, TO ELEMENT

(*Lis_select_byElem_toElem A B selectA l*) is (*Lis_select_byElem A B* (*Lis_Op_singleton A B selectA*) *l*).

Lis_select_byElem_toElem is somewhat similar in concept to the LISP mapcar function (see [27, chapter 12]).

Definition $\mathit{Lis\_select\_byElem\_toElem\_Ty} :=$

(∀

(*A* : *Type*)

(*B* : *Type*)

(*selectA* : (*Op A B*)),

(*Op* (*Lis A*) (*Lis B*))

) : *Type*.

Definition $\mathit{Lis\_select\_byElem\_toElem} :=$ (*fun A B selectA l* ⇒

($\mathit{Lis\_select\_byElem}$ *A B* ($\mathit{Lis\_Op\_singleton}$ *A B selectA*) *l*)

) : $\mathit{Lis\_select\_byElem\_toElem\_Ty}$.

9.18.15 LIST SELECT, BY ELEMENT, TO ELEMENT, SIMPLE

($\mathit{Lis\_select\_byElem\_toElem\_simp}$ *A selectA l*) is ($\mathit{Lis\_select\_byElem\_toElem}$ *A A selectA l*).

Definition $\mathit{Lis\_select\_byElem\_toElem\_simp\_Ty} :=$

(∀(*A* : *Type*)(*selectA* : ($\mathit{Op\_Simp}$ *A*)), ($\mathit{Op\_Simp}$ (*Lis A*))

) : *Type*.

Definition $\mathit{Lis\_select\_byElem\_toElem\_simp} :=$ (*fun A selectA l* ⇒

($\mathit{Lis\_select\_byElem\_toElem}$ *A A selectA l*)

) : $\mathit{Lis\_select\_byElem\_toElem\_simp\_Ty}$.

9.18.16 LIST SELECT, BY ELEMENT, TO ELEMENT, ITERATE

($\mathit{Lis\_select\_byElem\_toElem\_iter}$ *A selectA l m*) is ($\mathit{Nat\_iter}$ (*Lis A*) ($\mathit{Lis\_select\_byElem\_toElem\_simp}$ *A selectA*) *l m*).

Definition $\mathit{Lis\_select\_byElem\_toElem\_iter\_Ty} :=$

(∀

(*A* : *Type*)

(*selectA* : ($\mathit{Op\_Simp}$ *A*)),

($\mathit{Op\_Bin}$ (*Lis A*) *Nat* (*Lis A*))

) : *Type*.

Definition $\mathit{Lis\_select\_byElem\_toElem\_iter} :=$ (*fun A selectA l m* ⇒

(*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

(*Lis_select_byPrefix A B selectPrefix l*) 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) with the tail separated; and then concatenating the resulting lists *B*. If *l* is the nil list *A*, then (*Lis_select_byPrefix A B selectPrefix l*) is the nil list *B*.

Definition *Lis_select_byPrefix_Ty* :=
(∀
(*A* : *Type*)
(*B* : *Type*)
(*selectPrefix* : (*Op_Bin* (*Lis A*) *A* (*Lis B*))),
(*Op* (*Lis A*) (*Lis B*))
) : *Type*.

Definition *Lis_select_byPrefix* := (*fun A B selectPrefix l* ⇒
(*Lis_select_byPrefix_recur A B selectPrefix l* (*Lis_nil A*))
) : *Lis_select_byPrefix_Ty*.

9.18.19 LIST SELECT, BY PREFIX, SIMPLE

(*Lis_select_byPrefix_simp A selectPrefix l*) is (*Lis_select_byPrefix A A selectPrefix l*).

Definition *Lis_select_byPrefix_simp_Ty* :=
(∀
(*A* : *Type*)
(*selectPrefix* : (*Op_Bin* (*Lis A*) *A* (*Lis A*))),
(*Op_Simp* (*Lis A*))
) : *Type*.

Definition *Lis_select_byPrefix_simp* := (*fun A selectPrefix l* ⇒
(*Lis_select_byPrefix A A selectPrefix l*)
) : *Lis_select_byPrefix_simp_Ty*.

9.18.20 LIST SELECT, BY PREFIX, ITERATE

(*Lis_select_byPrefix_iter A selectPrefix l m*) is (*Nat_iter* (*Lis A*) (*Lis_select_byPrefix_simp A selectPrefix*) *l m*).

Definition *Lis_select_byPrefix_iter_Ty* :=

($\forall$

(*A* : *Type*)

(*selectPrefix* : (*Op_Bin* (*Lis A*) *A* (*Lis A*))),

(*Op_Bin* (*Lis A*) *Nat* (*Lis A*))

) : *Type*.

Definition *Lis_select_byPrefix_iter* := (*fun A selectPrefix l m* ⇒

(*Nat_iter* (*Lis A*) (*Lis_select_byPrefix_simp A selectPrefix*) *l m*)

) : *Lis_select_byPrefix_iter_Ty*.

9.18.21 LIST SELECT, BY PREFIX, TO ELEMENT

(*Lis_select_byPrefix_toElem A B selectPrefix l*) is (*Lis_select_byPrefix A B* (*Lis_Op_singleton_bin* (*Lis A*) *A B selectPrefix*) *l*).

Definition *Lis_select_byPrefix_toElem_Ty* :=

($\forall$

(*A* : *Type*)

(*B* : *Type*)

(*selectPrefix* : (*Op_Bin* (*Lis A*) *A B*)),

(*Op* (*Lis A*) (*Lis B*))

) : *Type*.

Definition *Lis_select_byPrefix_toElem* := (*fun A B selectPrefix l* ⇒

(*Lis_select_byPrefix*

A

B

(*Lis_Op_singleton_bin* (*Lis A*) *A B selectPrefix*)

l

)

) : *Lis_select_byPrefix_toElem_Ty*.

9.18.22 LIST SELECT, BY PREFIX, TO ELEMENT, SIMPLE

(*Lis_select_byPrefix_toElem_simp A selectPrefix l*) is (*Lis_select_byPrefix_toElem A A selectPrefix l*).

Definition *Lis_select_byPrefix_toElem_simp_Ty* :=

(∀

(*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* :=

(∀

(*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* :=

(∀

$(A : Type)$
$(matA : (Boo\_Pred\ A)),$
$(Op\_Simp\ (Lis\ A))$
$) : Type.$

Definition $Lis\_search := (\ fun\ A\ matA\ l \Rightarrow$
$(Lis\_select\_byElem\_simp\ A\ (Boo\_Op\_Lis\ A\ matA)\ l)$
$) : Lis\_search\_Ty.$

9.18.25 LIST SEARCH, FIRST

$(Lis\_search\_first\ A\ matA\ l)$ is $(Lis\_head\ A\ (Lis\_search\ A\ matA\ l))$.

Definition $Lis\_search\_first\_Ty :=$
$(\forall$
$(A : Type)$
$(matA : (Boo\_Pred\ A)),$
$(Op\ (Lis\ A)\ (Optional\ A))$
$) : Type.$

Definition $Lis\_search\_first := (\ fun\ A\ matA\ l \Rightarrow$
$(Lis\_head\ A\ (Lis\_search\ A\ matA\ l))$
$) : Lis\_search\_first\_Ty.$

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 $Lis\_search\_isFound\_Ty :=$
$(\forall$
$(A : Type)$
$(matA : (Boo\_Pred\ A)),$
$(Boo\_Pred\ (Lis\ A))$
$) : Type.$

Definition $Lis\_search\_isFound := (\ fun\ A\ matA\ l \Rightarrow$

(*Lis_nonEmpty A* (*Lis_search A matA l*))
) : *Lis_search_isFound_Ty*.

9.18.27 LIST INTERSECTION, MATCH

(*Lis_intersect_mat A0 A1 rel01 l1*) is the Boolean predicate on *A0* mapping *a0* : *A0* onto (*Lis_search_isFound A1* (*rel01 a0*) *l1*).

(*Lis_intersect_mat A0 A1 rel01 l1*) is the Boolean predicate on *A0* mapping *a0* : *A0* onto the true Boolean if there is some *a1* : *A1* in *l1* such that (*rel01 a0 a1*); and the false Boolean otherwise.

Definition *Lis_intersect_mat_Ty* :=
(∀
(*A0* : *Type*)
(*A1* : *Type*)
(*rel01* : (*Boo_Pred_Bin A0 A1*))
(*l1* : (*Lis A1*)),
(*Boo_Pred A0*)
) : *Type*.

Definition *Lis_intersect_mat* := (*fun A0 A1 rel01 l1 a0* ⇒
(*Lis_search_isFound A1* (*rel01 a0*) *l1*)
) : *Lis_intersect_mat_Ty*.

9.18.28 LIST INTERSECTION

(*Lis_intersect A0 A1 rel01 l0 l1*) is (*Lis_search A0* (*Lis_intersect_mat A0 A1 rel01 l1*) *l0*).

(*Lis_intersect A0 A1 rel01 l0 l1*) is *l0*, less those *a0* : *A0* for which there is no *a1* : *A1* in *l1* such that (*rel01 a0 a1*).

Definition *Lis_intersect_Ty* :=
(∀
(*A0* : *Type*)
(*A1* : *Type*)
(*rel01* : (*Boo_Pred_Bin A0 A1*)),
(*Op_Bin* (*Lis A0*) (*Lis A1*) (*Lis A0*))
) : *Type*.

Definition *Lis_intersect* := (*fun A0 A1 rel01 l0 l1* ⇒
(*Lis_search A0* (*Lis_intersect_mat A0 A1 rel01 l1*) *l0*)
) : *Lis_intersect_Ty*.

9.18.29 LIST INTERSECTION, CONNECTIVE

(*Lis_intersect_conn A relA l0 l1*) is (*Lis_intersect A A relA l0 l1*).

Definition *Lis_intersect_conn_Ty* :=
(∀
(*A* : *Type*)
(*relA* : (*Boo_Pred_Bin_Conn A*)),
(*Op_Bin_Simp* (*Lis A*))
) : *Type*.

Definition *Lis_intersect_conn* := (*fun A relA l0 l1* ⇒
(*Lis_intersect A A relA l0 l1*)
) : *Lis_intersect_conn_Ty*.

9.18.30 LIST INTERSECTION, FIRST

(*Lis_intersect_first A0 A1 rel01 l0 l1*) is (*Lis_head A0* (*Lis_intersect A0 A1 rel01 l0 l1*)).

Definition *Lis_intersect_first_Ty* :=
(∀
(*A0* : *Type*)
(*A1* : *Type*)
(*rel01* : (*Boo_Pred_Bin A0 A1*)),
(*Op_Bin* (*Lis A0*) (*Lis A1*) (*Optional A0*))
) : *Type*.

Definition *Lis_intersect_first* := (*fun A0 A1 rel01 l0 l1* ⇒
(*Lis_head A0* (*Lis_intersect A0 A1 rel01 l0 l1*))
) : *Lis_intersect_first_Ty*.

9.18.31 LIST INTERSECTION, FIRST, CONNECTIVE

(*Lis_intersect_first_conn A relA l0 l1*) is (*Lis_intersect_first A A relA l0 l1*).

Definition *Lis_intersect_first_conn_Ty* :=

(∀

(*A* : *Type*)

(*relA* : (*Boo_Pred_Bin_Conn A*)),

(*Op_Bin_Conn* (*Lis A*) (*Optional A*))

) : *Type*.

Definition *Lis_intersect_first_conn* := (*fun A relA l0 l1* ⇒

(*Lis_intersect_first A A relA l0 l1*)

) : *Lis_intersect_first_conn_Ty*.

9.18.32 LIST INTERSECTION, NON-EMPTY

(*Lis_intersect_nonEmpty A0 A1 rel01 l0 l1*) is (*Lis_nonEmpty A0* (*Lis_intersect A0 A1 rel01 l0 l1*)).

Definition *Lis_intersect_nonEmpty_Ty* :=

(∀

(*A0* : *Type*)

(*A1* : *Type*)

(*rel01* : (*Boo_Pred_Bin A0 A1*)),

(*Boo_Pred_Bin* (*Lis A0*) (*Lis A1*))

) : *Type*.

Definition *Lis_intersect_nonEmpty* := (*fun A0 A1 rel01 l0 l1* ⇒

(*Lis_nonEmpty A0* (*Lis_intersect A0 A1 rel01 l0 l1*))

) : *Lis_intersect_nonEmpty_Ty*.

9.18.33 LIST INTERSECTION, NON-EMPTY, CONNECTIVE

(*Lis_intersect_nonEmpty_conn A relA l0 l1*) is (*Lis_intersect_nonEmpty A A relA l0 l1*).

Definition *Lis_intersect_nonEmpty_conn_Ty* :=

(∀

(*A* : *Type*)

(*relA* : (*Boo_Pred_Bin_Conn A*)),

(*Boo_Pred_Bin_Conn* (*Lis A*))

) : *Type*.

Definition $Lis\_intersect\_nonEmpty\_conn$:= ($fun\ A\ relA\ l0\ l1 \Rightarrow$
(Lis_intersect_nonEmpty A A relA l0 l1)
) : $Lis\_intersect\_nonEmpty\_conn\_Ty$.

9.18.34 LIST TO BOOLEAN PREDICATE

($Lis\_to\_Boo\_Pred\ A\ l$) is the Boolean predicate on ($Boo\_Pred\ A$) mapping $matA$: ($Boo\_Pred\ A$) onto ($Lis\_search\_isFound\ A\ matA\ l$).

Definition $Lis\_to\_Boo\_Pred\_Ty$:=
(∀
(A : $Type$),
(Op ($Lis\ A$) (Boo_Pred ($Boo\_Pred\ A$)))
) : $Type$.

Definition $Lis\_to\_Boo\_Pred$:=
($fun\ A\ l\ matA \Rightarrow$ ($Lis\_search\_isFound\ A\ matA\ l$)) : $Lis\_to\_Boo\_Pred\_Ty$.

9.19 FUNDAMENTALS: OPTIONALS: AND LISTS SELECT

Poohbist.NummSquared.Fundamentals.Optionals.AndListsSelect

Poohbist.NummSquared.Fundamentals.Optionals.AndListsSelect defines some operators relating optionals and list selection operators.

9.19.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main*.
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.AndLists*.
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Select*.

9.19.2 OPTIONAL FLATTEN LIST

($Optional\_flattenLis\ A\ l$) is ($Lis\_select\_byElem$ ($Optional\ A$) A ($Optional\_to\_Lis\ A$) l).

Definition $\mathit{Optional\_flattenLis\_Ty} :=$
$(\forall$
$(A : \mathit{Type}),$
$(\mathit{Op}\ (\mathit{Lis}\ (\mathit{Optional}\ A))\ (\mathit{Lis}\ A))$
$) : \mathit{Type}.$

Definition $\mathit{Optional\_flattenLis} := (\ \mathit{fun}\ A\ l \Rightarrow$
$(\mathit{Lis\_select\_byElem}\ (\mathit{Optional}\ A)\ A\ (\mathit{Optional\_to\_Lis}\ A)\ l)$
$) : \mathit{Optional\_flattenLis\_Ty}.$

9.20 FUNDAMENTALS: LISTFUNCTIONS: MAIN

Poohbist.NummSquared.Fundamentals.Listfunctions.Main

Poohbist.NummSquared.Fundamentals.Listfunctions.Main defines listfunctions, simple listfunctions, and some operators on listfunctions and simple listfunctions.

9.20.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Select.*

9.20.2 LISTFUNCTIONS

A listfunction from A to B is an operator from A to a list B.

Definition $\mathit{Lisfunction\_Ty} := (\ \forall (A : \mathit{Type})(B : \mathit{Type}),\ \mathit{Type}\ ) : \mathit{Type}.$

Definition $\mathit{Lisfunction} := (\ \mathit{fun}\ A\ B \Rightarrow (\mathit{Op}\ A\ (\mathit{Lis}\ B))\ ) : \mathit{Lisfunction\_Ty}.$

9.20.3 LISTFUNCTION TO BOOLEAN PREDICATE

$(\mathit{Lisfunction\_to\_Boo\_Pred}\ A\ B\ \mathit{lf}\ a)$ is $(\mathit{Lis\_to\_Boo\_Pred}\ B\ (\mathit{lf}\ a))$.

Definition $\mathit{Lisfunction\_to\_Boo\_Pred\_Ty} :=$

($\forall$

$(A : Type)$

$(B : Type)$,

$(Op\_Bin\ (Lisfunction\ A\ B)\ A\ (Boo\_Pred\ (Boo\_Pred\ B)))$

) : *Type*.

Definition *Lisfunction_to_Boo_Pred* :=

(*fun A B lf a* $\Rightarrow$ (*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* := ($\forall (A : Type)$, *Type*) : *Type*.

Definition *Lisfunction_Simp* :=

(*fun A* $\Rightarrow$ (*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* :=

($\forall$

$(A : Type)$,

$(Op\_Bin\ (Lisfunction\_Simp\ A)\ A\ (Boo\_Pred\ (Boo\_Pred\ A)))$

) : *Type*.

Definition *Lisfunction_Simp_to_Boo_Pred* :=

(*fun A lf a* $\Rightarrow$ (*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* :=

(∀

(*A* : *Type*),

(*Op_Bin* (*Lisfunction_Simp A*) *Nat* (*Lisfunction_Simp A*))

) : *Type*.

Definition *Lisfunction_Simp_iter* := (*fun A lf m a* ⇒

(*Lis_select_byElem_iter A lf* [A, a] *m*)

) : *Lisfunction_Simp_iter_Ty*.

9.20.7 SIMPLE LISTFUNCTION ITERATE, CURRY 2

(*Lisfunction_Simp_iter_c2 A lf a m*) is (*Lisfunction_Simp_iter A lf m a*).

Definition *Lisfunction_Simp_iter_c2_Ty* :=

(∀

(*A* : *Type*),

(*Op_Tri* (*Lisfunction_Simp A*) *A Nat* (*Lis A*))

) : *Type*.

Definition *Lisfunction_Simp_iter_c2* :=

(*fun A lf a m* ⇒ (*Lisfunction_Simp_iter A lf m a*))

: *Lisfunction_Simp_iter_c2_Ty*.

9.20.8 SIMPLE LISTFUNCTION ITERATE, CUMULATIVE

(*Lisfunction_Simp_iter_cum A lf m*) is the simple listfunction on *A* mapping *a* : *A* onto (*Lis_generate A* (*Lisfunction_Simp_iter_c2 A lf a*) *m*).

Definition *Lisfunction_Simp_iter_cum_Ty* :=

(∀

(*A* : *Type*),

(*Op_Bin* (*Lisfunction_Simp A*) *Nat* (*Lisfunction_Simp A*))

) : *Type*.

Definition *Lisfunction_Simp_iter_cum* := (*fun A lf m a* ⇒

(*Lis_generate A* (*Lisfunction_Simp_iter_c2 A lf a*) *m*)

) : *Lisfunction_Simp_iter_cum_Ty*.

9.21 NUMMSQUARED: SYNTAX: ABSTRACT: MAIN

Poohbist.NummSquared.NummSquared.Syntax.Abstract.Main defines the NummSquared abstract syntax types, and some operators on these types.

9.21.1 DEPENDENCIES

Require Import *Poohbist.NummSquared.Fundamentals.Operators.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Booleans.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Efficient.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Naturals.Efficient.AndLists.*
Require Import *Poohbist.NummSquared.Fundamentals.Optionals.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Main.*
Require Import *Poohbist.NummSquared.Fundamentals.Lists.Select.*

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 *Ns_Chr_Digit_eq* := (*fun cd0 cd1* ⇒
match cd0, cd1
return Boo
with
| *Ns_Chr_Digit_d0, Ns_Chr_Digit_d0* ⇒ *Boo_t*
| *Ns_Chr_Digit_d1, Ns_Chr_Digit_d1* ⇒ *Boo_t*
| *Ns_Chr_Digit_d2, Ns_Chr_Digit_d2* ⇒ *Boo_t*
| *Ns_Chr_Digit_d3, Ns_Chr_Digit_d3* ⇒ *Boo_t*
| *Ns_Chr_Digit_d4, Ns_Chr_Digit_d4* ⇒ *Boo_t*
| *Ns_Chr_Digit_d5, Ns_Chr_Digit_d5* ⇒ *Boo_t*
| *Ns_Chr_Digit_d6, Ns_Chr_Digit_d6* ⇒ *Boo_t*
| *Ns_Chr_Digit_d7, Ns_Chr_Digit_d7* ⇒ *Boo_t*
| *Ns_Chr_Digit_d8, Ns_Chr_Digit_d8* ⇒ *Boo_t*
| *Ns_Chr_Digit_d9, Ns_Chr_Digit_d9* ⇒ *Boo_t*
| _, _ ⇒ *Boo_f*
end
) : (*Boo_Pred_Bin_Conn Ns_Chr_Digit*).

9.21.4 NUMMSQUARED 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 *Ns_Chr_Ident_Start* : *Type* :=
| *Ns_Chr_Ident_Start_exclamationMark* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_ampersand* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_asterisk* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_plusSign* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_hyphenMinus* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_slash* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_lessThanSign* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_equalsSign* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_greaterThanSign* : *Ns_Chr_Ident_Start*
| *Ns_Chr_Ident_Start_A* : *Ns_Chr_Ident_Start*

| $Ns\_Chr\_Ident\_Start\_B : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_C : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_D : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_E : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_F : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_G : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_H : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_I : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_J : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_K : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_L : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_M : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_N : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_O : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_P : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_Q : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_R : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_S : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_T : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_U : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_V : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_W : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_X : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_Y : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_Z : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_circumflexAccent : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_a : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_b : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_c : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_d : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_e : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_f : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_g : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_h : Ns\_Chr\_Ident\_Start$
| $Ns\_Chr\_Ident\_Start\_i : Ns\_Chr\_Ident\_Start$

| *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* ⇒
match cis0, cis1
return Boo
with
| *Ns_Chr_Ident_Start_exclamationMark,*
Ns_Chr_Ident_Start_exclamationMark ⇒
Boo_t
| *Ns_Chr_Ident_Start_ampersand, Ns_Chr_Ident_Start_ampersand* ⇒ *Boo_t*
| *Ns_Chr_Ident_Start_asterisk, Ns_Chr_Ident_Start_asterisk* ⇒ *Boo_t*
| *Ns_Chr_Ident_Start_plusSign, Ns_Chr_Ident_Start_plusSign* ⇒ *Boo_t*
| *Ns_Chr_Ident_Start_hyphenMinus, Ns_Chr_Ident_Start_hyphenMinus* ⇒
Boo_t

$| \; Ns\_Chr\_Ident\_Start\_slash, Ns\_Chr\_Ident\_Start\_slash \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_lessThanSign, Ns\_Chr\_Ident\_Start\_lessThanSign \Rightarrow$
Boo_t

$| \; Ns\_Chr\_Ident\_Start\_equalsSign, Ns\_Chr\_Ident\_Start\_equalsSign \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_greaterThanSign,$
$Ns\_Chr\_Ident\_Start\_greaterThanSign \Rightarrow$
Boo_t

$| \; Ns\_Chr\_Ident\_Start\_A, Ns\_Chr\_Ident\_Start\_A \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_B, Ns\_Chr\_Ident\_Start\_B \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_C, Ns\_Chr\_Ident\_Start\_C \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_D, Ns\_Chr\_Ident\_Start\_D \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_E, Ns\_Chr\_Ident\_Start\_E \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_F, Ns\_Chr\_Ident\_Start\_F \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_G, Ns\_Chr\_Ident\_Start\_G \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_H, Ns\_Chr\_Ident\_Start\_H \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_I, Ns\_Chr\_Ident\_Start\_I \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_J, Ns\_Chr\_Ident\_Start\_J \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_K, Ns\_Chr\_Ident\_Start\_K \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_L, Ns\_Chr\_Ident\_Start\_L \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_M, Ns\_Chr\_Ident\_Start\_M \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_N, Ns\_Chr\_Ident\_Start\_N \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_O, Ns\_Chr\_Ident\_Start\_O \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_P, Ns\_Chr\_Ident\_Start\_P \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_Q, Ns\_Chr\_Ident\_Start\_Q \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_R, Ns\_Chr\_Ident\_Start\_R \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_S, Ns\_Chr\_Ident\_Start\_S \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_T, Ns\_Chr\_Ident\_Start\_T \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_U, Ns\_Chr\_Ident\_Start\_U \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_V, Ns\_Chr\_Ident\_Start\_V \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_W, Ns\_Chr\_Ident\_Start\_W \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_X, Ns\_Chr\_Ident\_Start\_X \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_Y, Ns\_Chr\_Ident\_Start\_Y \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_Z, Ns\_Chr\_Ident\_Start\_Z \Rightarrow Boo\_t$

$| \; Ns\_Chr\_Ident\_Start\_circumflexAccent,$
$Ns\_Chr\_Ident\_Start\_circumflexAccent \Rightarrow$

```
            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 comment0 comment1*) is (*Ns_Eff_Lis_eq comment0 comment1*).

Definition *Ns_Comment_eq* := (*fun comment0 comment1* ⇒
        (*Nat_Eff_Lis_eq 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* := (*fun ids0 ids1* ⇒
    if
        (*Ns_Chr_Ident_Start_eq*
            (*Ns_Ident_Simp_start ids0*)
            (*Ns_Ident_Simp_start ids1*)

```
            )
        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_f
        ) : (Boo_Pred_Bin_Conn Ns_Ident_Simp).
```

9.21.12 NUMMSQUARED IDENTIFIERS

A NummSquared identifier is a non-empty list of NummSquared simple identifiers.

```
Definition Ns_Ident := (Lis_Ne Ns_Ident_Simp).
```

9.21.13 NUMMSQUARED IDENTIFIER EQUALS

(*Ns_Ident_eq id0 id1*) is the true Boolean if *id0* and *id1* are structurally equal; and the false Boolean otherwise.

```
Definition 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).
```

9.21.14 NUMMSQUARED SIMPLE IDENTIFIER TO NUMMSQUARED IDENTIFIER

(*Ns_Ident_Simp_to_Ns_Ident ids*) is the NummSquared identifier containing just ids.

```
Definition 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* := (*fun m0 m1* ⇒
(*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* := (*fun m0 m1* ⇒
(*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* := (*fun str0 str1* ⇒
(*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* := (*fun prim0 prim1* ⇒
match prim0, prim1
return Boo
with
| *Ns_Prim_nat m0, Ns_Prim_nat m1* ⇒ (*Ns_Prim_Nat_eq m0 m1*)
| *Ns_Prim_chr m0, Ns_Prim_chr m1* ⇒ (*Ns_Prim_Chr_eq m0 m1*)
| *Ns_Prim_str str0, Ns_Prim_str str1* ⇒ (*Ns_Prim_Str_eq str0 str1*)
| _, _ ⇒ *Boo_f*
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 $\mathit{Ns\_Constant\_Norm\_Noncompu} : \mathit{Type} :=$

$\quad | \; \mathit{Ns\_Constant\_Norm\_Noncompu\_ns\_eq} : \mathit{Ns\_Constant\_Norm\_Noncompu}.$

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 $\mathit{Ns\_Constant\_Norm} : \mathit{Type} :=$

$\quad | \; \mathit{Ns\_Constant\_Norm\_compu} : (\mathit{Op}\ \mathit{Ns\_Constant\_Norm\_Compu}\ \mathit{Ns\_Constant\_Norm})$

$\quad | \; \mathit{Ns\_Constant\_Norm\_noncompu} :$

$\qquad\qquad (\mathit{Op}\ \mathit{Ns\_Constant\_Norm\_Noncompu}\ \mathit{Ns\_Constant\_Norm}).$

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 *Ns_Constant_Nonnorm_Compu* : *Type* :=
| *Ns_Constant_Nonnorm_Compu_left* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_right* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_conf_n* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_not_n* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Null_to_Zero* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Zero* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_One* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Nuro* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Leaf* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Simp* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Rule* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Tree_step_pair_ug* :
Ns_Constant_Nonnorm_Compu
| *Ns_Constant_Nonnorm_Compu_Tree_step_ug* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Tree* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_dom_ne* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_res* : *Ns_Constant_Nonnorm_Compu*
| *Ns_Constant_Nonnorm_Compu_Nuro_set_res* : *Ns_Constant_Nonnorm_Compu*

$\mid$ *Ns_Constant_Nonnorm_Compu_Tree_set_res* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_s_d_res_left_ug* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_s_d_res_right_ug* :
Ns_Constant_Nonnorm_Compu
$\mid$ *Ns_Constant_Nonnorm_Compu_s_d_res_ug* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_s_d_res* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_p_d_res_uncurry_ug* :
Ns_Constant_Nonnorm_Compu
$\mid$ *Ns_Constant_Nonnorm_Compu_p_d_res_ug* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_p_d_res* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_not* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_imp_n* : *Ns_Constant_Nonnorm_Compu*
$\mid$ *Ns_Constant_Nonnorm_Compu_imp* : *Ns_Constant_Nonnorm_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 *Ns_Constant_Nonnorm_Noncompu* : *Type* :=
| *Ns_Constant_Nonnorm_Noncompu_not_eq* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_all_sm* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_eq_pair_ug* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_eq_res_at* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_eq_res* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_eq_dom_res* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_eq_both_res* : *Ns_Constant_Nonnorm_Noncompu*
| *Ns_Constant_Nonnorm_Noncompu_eq_rhs* : *Ns_Constant_Nonnorm_Noncompu.*

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 *Ns_Constant_Nonnorm* : *Type* :=
| *Ns_Constant_Nonnorm_compu* :
(*Op Ns_Constant_Nonnorm_Compu Ns_Constant_Nonnorm*)
| *Ns_Constant_Nonnorm_noncompu* :
(*Op Ns_Constant_Nonnorm_Noncompu Ns_Constant_Nonnorm*).

9.21.29 NUMMSQUARED CONSTANTS

A NummSquared constant is exactly one of the following:

- a NummSquared normalized constant

- a NummSquared non-normalized constant

Inductive $Ns\_Constant$: $Type$:=
| $Ns\_Constant\_norm$: (Op $Ns\_Constant\_Norm$ $Ns\_Constant$)
| $Ns\_Constant\_nonnorm$: (Op $Ns\_Constant\_Nonnorm$ $Ns\_Constant$).

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 *compuation*, 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_Lg_ctor* :
(*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_Sm_ctor* : (*Op Ns_Func_Lg_Lis_P2*
Ns_Combo_Compu_Co_Sm)
with Ns_Combo_Compu_Tuple : *Type* :=
| *Ns_Combo_Compu_Tuple_ctor* : (*Op Ns_Func_Lg_Lis_P2*
Ns_Combo_Compu_Tuple)
with Ns_Combo_Compu_Lis : *Type* :=
| *Ns_Combo_Compu_Lis_ctor* : (*Op Ns_Func_Lg_Lis Ns_Combo_Compu_Lis*)
with Ns_Combo_Compu_S_D : *Type* :=
| *Ns_Combo_Compu_S_D_ctor* : (*Op Ns_Func_Lg Ns_Combo_Compu_S_D*)
with Ns_Combo_Compu_P_D : *Type* :=
| *Ns_Combo_Compu_P_D_ctor* : (*Op Ns_Func_Lg Ns_Combo_Compu_P_D*)
with Ns_Combo_Compu_C : *Type* :=
| *Ns_Combo_Compu_C_ctor* : (*Op_Bin_Conn Ns_Func_Lg Ns_Combo_Compu_C*)
with Ns_Combo_Compu_Ite : *Type* :=
| *Ns_Combo_Compu_Ite_ctor* : (*Op_Tri_Conn Ns_Func_Lg Ns_Combo_Compu_Ite*)
with Ns_Combo_Compu_R : *Type* :=
| *Ns_Combo_Compu_R_ctor* : (*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_Root_ctor* :
(*Op_Bin_Conn Ns_Func_Lg Ns_Combo_Compu_C_Aug_Root*)
with Ns_Combo_Compu_C_Aug : *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* :=

| *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*).

9.21.31 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 *Ns_Access_Tuple_Loc_Lis* := (*Lis_P2 Ns_Ident*).

9.21.32 NUMMSQUARED LOCAL CONTEXTS

A NummSquared local context is an optional NummSquared local tuple accessor list.
Definition *Ns_Context_Loc* := (*Optional Ns_Access_Tuple_Loc_Lis*).

9.21.33 NUMMSQUARED DEFINITIONS

Record $\mathit{Ns\_Def} : \mathit{Type} := \mathit{Ns\_Def\_ctor}$ {
$\mathit{Ns\_Def\_comment} : \mathit{Ns\_Comment}$;
$\mathit{Ns\_Def\_name} : \mathit{Ns\_Ident}$;
$\mathit{Ns\_Def\_context\_loc} : \mathit{Ns\_Context\_Loc}$;
$\mathit{Ns\_Def\_rhs} : \mathit{Ns\_Func\_Lg}$
}.

9.21.34 NUMMSQUARED GLOBAL CONTEXTS

The order is reversed relative to the concrete syntax.
Definition $\mathit{Ns\_Context\_Glob} := (\mathit{Lis}\ \mathit{Ns\_Def})$.

9.21.35 NUMMSQUARED MODULES

Record $\mathit{Ns\_Modu} : \mathit{Type} := \mathit{Ns\_Modu\_ctor}$ {
$\mathit{Ns\_Modu\_comment} : \mathit{Ns\_Comment}$;
$\mathit{Ns\_Modu\_name} : \mathit{Ns\_Ident}$;
$\mathit{Ns\_Modu\_context\_glob} : \mathit{Ns\_Context\_Glob}$
}.

9.21.36 NUMMSQUARED ABSTRACT PROGRAMS

The order is reversed relative to the concrete syntax.
Definition $\mathit{Ns\_Program\_Abs} := (\mathit{Lis}\ \mathit{Ns\_Modu})$.

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.

BIBLIOGRAPHY

[1] Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006. URL `http://www.cs.nott.ac.uk/~txa/`.

[2] James H. Andrews. A weakly-typed higher order logic with general lambda terms and Y combinator. In *Proceedings, Works In Progress Track, 15th International Conference on Theorem Proving in Higher Order Logics*, number CP-2002-211736. NASA Conference Publication, August 2002. URL `http://www.csd.uwo.ca/faculty/andrews/papers/index.html`.

[3] Sergei N. Artemov. On explicit reflection in theorem proving and formal verification. In *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1999. URL `http://web.cs.gc.cuny.edu/~sartemov/`.

[4] Jeremy Avigad and Richard Zach. The epsilon calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Summer 2002. URL `http://plato.stanford.edu/archives/sum2002/entries/epsilon-calculus/`.

[5] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, August 1978.

[6] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992. URL `http://www.cs.ru.nl/~henk/papers.html`.

[7] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. Technical report, State University of New York at Stony Brook. URL `ftp://ftp.cs.sunysb.edu/pub/TechReports/kifer/hilog.pdf`.

[8] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. URL `http://coq.inria.fr/`.

[9] H. D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Springer-Verlag, 1984.

[10] William M. Farmer. Stmm: A set theory for mechanized mathematics. *Journal of Automated Reasoning*, 2001. URL `http://imps.mcmaster.ca/doc/stmm.pdf`.

[11] Paul C. Gilmore. *Logicism Renewed: Logical Foundations for Mathematics and Computer Science.* Association for Symbolic Logic & AK Peters, 2005.

[12] Paul C. Gilmore. Soundness & cut-elimination for NaDSyL. Technical Report TR-97-1, University of British Columbia, February 1997. URL `http://www.cs.ubc.ca/cgi-bin/tr/1997/TR-97-01`.

[13] Georges Gonthier. *A computer-checked proof of the Four Colour Theorem.* Microsoft Research, 2004. URL `http://research.microsoft.com/~gonthier/4colproof.pdf`.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley Professional, 3^{rd} edition, 2005. URL `http://java.sun.com/docs/books/jls/`.

[15] Klaus Grue. Lambda-calculus as a foundation for mathematics. WWW site, August 1997. URL `http://www.diku.dk/~grue/`.

[16] Klaus Grue. Map theory with classical maps. WWW site, June 2001. URL `http://www.diku.dk/~grue/`.

[17] William S. Hatcher. *Foundations of Mathematics.* W. B. Saunders Company, 1968.

[18] Hugo Herbelin, Florent Kirchner, Benjamin Monate, and Julien Narboux. *Coq Version 8.0 for the Clueless.* LogiCal Project, April 2005. URL `http://coq.inria.fr/`. Online FAQ.

[19] C. A. R. Hoare and D. C. S. Allison. Incomputability. *Computing Surveys*, 4(3), September 1972.

[20] Douglas J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *Proceedings of the Ninth International Conference on Automated Deduction*, number 310 in LNCS. Springer-Verlag, 1988. URL `http://www.nuprl.org/documents/Howe/ComputationalMetatheory.html`.

[21] Douglas J. Howe. A classical set-theoretic model of polymorphic extensional type theory. URL `http://citeseer.ist.psu.edu/howe97classical.html`. 1997.

[22] Samuel Howse. *NummSquared 2006a0 Done Formally*. Poohbist Technology, October 2006. URL `http://nummist.com/poohbist/`. October 18, 2006 pre-release.

[23] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq proof assistant: a tutorial*. LogiCal Project, 2004. URL `http://coq.inria.fr/`.

[24] INRIA. An overview of the Caml language and tools. WWW site, January 2005. URL `http://caml.inria.fr/about/overview.en.html`.

[25] A. D. Irvine. Russell's paradox. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Summer 2004. URL `http://plato.stanford.edu/archives/sum2004/entries/russell-paradox/`.

[26] Roger Bishop Jones. Pure functions. WWW site, December 1998. URL `http://www.rbjones.com/rbjpub/logic/inter013.htm`.

[27] David B. Lamkins. *Successful Lisp: How to Understand and Use Common Lisp*. bookfix.com, 2004. URL `http://psg.com/~dlamkins/Site/sl.html`.

[28] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 2^{nd} edition, 1998.

[29] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, April 1960. URL `http://www-formal.stanford.edu/jmc/recursive.html`.

[30] Elliott Mendelson. *Introduction to Mathematical Logic*. Wadsworth & Brooks/Cole Advanced Books & Software, 3^{rd} edition, 1987.

[31] *C# Language Specification*. Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/vcsharp/programming/language/`. Version 1.2.

[32] *The F# Manual*. Microsoft Corporation, 2006. URL `http://research.microsoft.com/fsharp/manual/default.aspx`.

[33] Praxis. Introduction to SPARK. WWW site, February 2006. URL `http://www.praxis-his.com/sparkada/intro.asp`.

[34] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness: An experiment in MIZAR. *Journal of Automated Reasoning*, 1999.

[35] Jonathan P. Seldin. The logic of Church and Curry. In Dov Gabbay and John Woods, editors, *Handbook of the History of Logic*, volume 5. Elsevier. URL `http://www.cs.uleth.ca/~seldin/publications.shtml`. To appear.

[36] Gaisi Takeuti and Wilson M. Zaring. *Introduction to Axiomatic Set Theory*. Springer-Verlag, 2^{nd} edition, 1982.

[37] John Tromp. Binary lambda calculus and combinatory logic. WWW site, March 2006. URL `http://homepages.cwi.nl/~tromp/cl/LC.pdf`.

[38] *The Unicode Character Code Charts*. The Unicode Consortium, October 2005. URL `http://www.unicode.org/charts/`. Version 4.1.

[39] *The Unicode Standard, Version 4.1.0*. The Unicode Consortium, March 2005. URL `http://www.unicode.org/versions/Unicode4.1.0/`. defined by: The Unicode Standard, Version 4.0 (Boston, MA, Addison-Wesley, 2003. ISBN 0-321-18578-1), as amended by Unicode 4.0.1 (`http://www.unicode.org/versions/Unicode4.0.1/`) and by Unicode 4.1.0.

[40] John von Neumann. An axiomatization of set theory. In Jean van Heijenoort, editor, *From Frege to Gödel*. Harvard University Press, 1967. Paper originally published 1925.

[41] Edward N. Zalta. Frege's logic, theorem, and foundations for arithmetic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Summer 2006. URL `http://plato.stanford.edu/entries/frege-logic/`.

INDEX