NummSquared: a New Foundation for Formal Methods

Samuel Howse
Poohbist Technology
607 Francklyn Street
Halifax, Nova Scotia B3H 3B6
Canada
Phone: 1-902-422-0845
Email: samuelhowse@poohbist.com
Web page: http://poohbist.com/

December 15, 2006

Abstract

To spread the use of formal methods, a language must appeal to programmers, mathematicians and logicians. Set theory is the standard mathematical foundation, but often ignores computational aspects. Type theory has good support for mixing specification and implementation, but often imposes type constraints in excess of those found in typical programming languages. Furthermore, standard mathematics is untyped. Languages based on the untyped lambda calculus often permit non-terminating programs and require reasoning in non-classical logics. Languages without higher order functions often lack polymorphism. There is a wide gap between conventional programming languages and logic. This paper proposes NummSquared, a new formal language based only on untyped higher order functions, which allows only terminating programs, has a classical logic, is related to well-founded set theory, and supports reflection. NummSquared supports rapid prototyping without proofs to reduce cost, and supports adding proofs later.

1 Overview and comparison

A feature that is elegant in a programming language may be inappropriate for formal methods, particularly when the programming language must also serve as a logic. The untyped lambda calculus, a useful model for many programming languages, is elegant because it is based only on untyped functions, and any function may be passed as an argument to any other function. For example, let f be the function such that $f(x) = x(x)$. But this flexibility is a source of trouble. For example, $f(f)$ reduces to itself - a non-terminating program. Worse, the untyped lambda calculus augmented by negation is inconsistent. Let R be the function such that $R(x) = \mathsf{not}(x(x))$. Then $R(R) = \mathsf{not}(R(R))$, so $R(R)$ is neither true nor false. This Russell's paradox, and other paradoxes in the untyped lambda calculus, are discussed by Seldin [12].

Both set theory and type theory offer solutions to the paradoxes based on well-foundedness (roughly, objects are built from existing objects, thus avoiding the circularity of the untyped lambda calculus). Even so, for reasons given in the abstract, languages based on sets or types have some disadvantages. Another avenue may be of interest: *well-founded* languages based on untyped higher order functions. This avenue is much less explored. Von Neumann

[14] and Jones [10] propose such languages, but ignore computational aspects. Grue's map theory [6] has a well-founded subset, but also has non-terminating programs and a logic that is partially non-classical. This paper proposes NummSquared, a new formal language. NummSquared pursues this avenue by being entirely well-founded, and including features for classical logic, mathematics and computation. NummSquared's well-foundedness implies less flexibility than the untyped lambda calculus, which allows non-terminating programs and paradoxes to be averted.

All NummSquared programs terminate and avoid paradoxes without the need for the programmer to supply proofs. Thus proofs may be omitted initially, then added as needed to prove that implementation satisfies specification. A benefit is to save the cost of doing proofs during the rapid prototyping phase.

NummSquared is variable-free, as is Backus's FP [3]. But, for readability, NummSquared concrete syntax is somewhat reminiscent of lambda calculi. NummSquared supports reflection: NummSquared is its own macro language, and NummSquared programs can manipulate NummSquared proofs. Variable-freeness eases reflection.

NummSquared has no side-effects or global state, which simplifies mathematical reasoning, and improves security.

This paper highlights some interesting NummSquared features. For more details on NummSquared, see [9]. The NummSquared interpreter, NsGo, is a work in progress. NsGo, an F#/C# .NET assembly, is mostly automatically extracted from a program of the Coq proof assistant. (See [4] and [11].) Using a high quality proof assistant such as Coq increases the reliability of NsGo.

1.1 Coercion

The challenge in using untyped higher order functions is how to restrict the argument to prevent non-termination and paradoxes. In set theory, a function is represented by a set of ordered pairs, the domain of a function is a set, and the domain does not include all sets. However, when the domain is a function space (the higher order case), membership in the domain is not computable.

NummSquared uses an indirect approach: Since membership in the domain is not computable, rejecting an argument outside the domain is not possible. Therefore, NummSquared coerces the argument to a member of the domain. Coercion in NummSquared is somewhat related to type conversion, a feature of many programming languages, but NummSquared coercion is generalized to higher order functions. Howe [8, section 2.2] restricts untyped lambda terms to set domains, which is somewhat related to NummSquared coercion, but Howe includes non-terminating terms. In contrast to NummSquared coercion, coercion in Observational Type Theory [1, section 2.2] is not automatic and requires proof.

2 NummSquared semantics

The semantics of NummSquared are described briefly. Many more definitions and theorems, and also proofs, may be found in [9].

2.1 Small function extensions

The central objects in NummSquared are the small functions. Small function extensions (semantic objects) are divided into rule small function extensions (represented by rules), and simple small function extensions (represented by simpler means). Simple small function extensions are further divided into leaf small function extensions (which are atomic), and pair small function extensions (ordered pairs of small function extensions).

A leaf small function extension is null, zero or one. null is analogous to the null pointer in many programming languages, and represents the absence of relevant information. null does *not* represent undefined nor non-termination.

zero and one represent false and true, respectively. It is convenient for zero to contain null, and for one to contain null and zero. (Containment means structural containment. For example, a record contains its fields.)

A pair small function extension p contains left(p) and right(p), which are small function extensions. It is convenient for a pair small function extension p to additionally contain null, zero and one.

A rule small function extension f contains the domain of f (a *small* set of small function extensions), denoted by dom(f), and, for each dom(f) element x, the specific result of f at x (a small function extension), denoted by f<x>. A small set is a set no larger than a ZFC set. (ZFC is Zermelo-Fraenkel set theory with the axiom of choice - see [13, p.84,132-133].) The requirement that dom(f) be small is essential. Without that requirement, dom(f) could be the set of all small function extensions, entailing the troubles of the untyped lambda calculus.

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.

A small function extension f is a Boolean iff f = zero or f = one. Let Null be the set of null. Let Nuro be the set of null and zero. Let Leaf be the set of all leaf small function extensions, i.e. null, zero and one.

A small function extension f is a tree iff f contains, directly or indirectly, only simple small function extensions. A tree small function extension contains no rule small function extensions, and is therefore a particularly simple way to represent data. Let Tree be the set of all tree small function extensions.

Rule small function extensions obviously deserve to be called functions, and simple small function extensions can be viewed as functions too, due to the following definitions.

Let dom(null) = Null, dom(zero) = Null, and dom(one) = Nuro. For a pair small function extension p, let dom(p) = Leaf.

For a leaf small function extension l, and a dom(l) element x, let l<x> = x. For a pair small function extension p, and a dom(p) element x, let p<x> be as follows:

- null if x = null
- left(p) if x = zero
- right(p) if x = one

For a small function extension f, the range of f, denoted by ran(f), is the set of all f<x> such that x is a dom(f) element. For a small function extension f, the field of f, denoted by field(f), is the union of dom(f) and ran(f).

For a small function extension f ≠ null, and a field(f) element x, x is structurally smaller than f. Thus f ≠ null is built after the elements of its field. In this way, NummSquared is well-founded. Jones's Pure Functions [10] are well-founded in a similar way. The well-foundedness of NummSquared enables recursion and induction on small function extensions.

null is built from nothing at all. However, null is an element of its own field. When doing recursion and induction, it is therefore necessary to separately handle null as a base case.

For a small function extension f, and a dom(f) element x, the specific result of f at x, denoted by f<x>, has been defined. The challenge is to generalize the definition of result to include *all* small function extensions f and x satisfying a certain constraint which does not involve dom(f). This challenge is addressed by using NummSquared coercion to coerce x into a dom(f) element.

2.2 Domain extensions

To coerce x into a dom(f) element, it is not sufficient that dom(f) is a small set of small function extensions - a constraint on dom(f) is required. The required constraint is type information, as in type theory. However, the purpose of the type information differs: the type information is used for runtime coercion, *not* for type checking (compile-time

or runtime). Therefore, the NummSquared programmer is not constrained by type constraints designed to make type checking computable. The type information about dom(f) is a NummSquared domain extension.

Domain extensions are divided into constant domain extensions (analogous to primitive data types) and combination domain extensions (analogous to defined data types). Combination domain extensions are further divided into dependent sum domain extensions, and dependent product domain extensions (analogous to the corresponding types in type theory - see Coq [4, sections 3.1.4, 4.1.3, 4.2]).

A constant domain extension is Dom.Null, Dom.Nuro, Dom.Leaf or Dom.Tree.

A dependent sum domain extension A contains domExtFam(A) (a domain extension family). A dependent product domain extension A contains domExtFam(A) (a domain extension family).

A domain extension family F contains the following:

- the domain of F (a *small* set of small function extensions), denoted by dom(F)
- for each dom(F) element x, the specific result of F at x (a domain extension), denoted by F<x>
- the domain extension of F (a domain extension), denoted by domExt(F)

A domain extension represents a domain. Let dom(Dom.Null) = Null, dom(Dom.Nuro) = Nuro, dom(Dom.Leaf) = Leaf, and dom(Dom.Tree) = Tree.

For a dependent sum domain extension A, let dom(A) be the set of null and all pair small function extensions p such that left(p) is a dom(domExtFam(A)) element, and right(p) is a dom(domExtFam(A)<left(p)>) element.

For a dependent product domain extension A, let dom(A) be the set of null and all rule small function extensions f such that dom(f) = dom(domExtFam(A)) and, for each dom(f) element x, f<x> is a dom(domExtFam(A)<x>) element.

A domain extension f or domain extension family F is valid iff, for each domain extension family G contained, directly or indirectly, in f or F, and for G = F, dom(domExt(G)) = dom(G).

The domain extension irrelevance theorem proves that a domain extension contains no more information than the domain it represents: for *valid* domain extensions A and B, if dom(A) = dom(B), then A = B. However, domain extensions still serve as a constraint on domains, and are computationally useful for coercion.

2.3 Tagged small function extensions

A rule tagged small function extension is a rule small function extension constrained by a domain extension, as defined below. Tagging is somewhat analogous to adding runtime type information.

Tagged small function extensions are divided into simple tagged small function extensions, and rule tagged small function extensions. Simple tagged small function extensions are further divided into leaf small function extensions, and pair tagged small function extensions.

A pair tagged small function extension p contains left(p) and right(p), which are tagged small function extensions. It is convenient for a pair tagged small function extension p to additionally contain null, zero and one.

A rule tagged small function extension f contains the following:

- the domain of f (a *small* set of small function extensions), denoted by dom(f)
- for each dom(f) element x, the specific result of f at x (a tagged small function extension), denoted by f<x>
- the domain extension of f (a *valid* domain extension), denoted by domExt(f), such that dom(domExt(f)) = dom(f)

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 tagged small function extension f, let untag(f) be the small function extension obtained from f by removing all domain extensions contained, directly or indirectly, in f.

The tag irrelevance theorem: for tagged small function extensions f and g, if untag(f) = untag(g), then f = g.

A tagged small function extension f is a tree iff untag(f) is a tree.

For a *valid* domain extension A, and a dom(A) element f, let tagged(A, f) be the tagged small function extension such that untag(tagged(A, f)) = f. The existence of such a tagged small function extension is proved in [9]. The fact that f is a dom(A) element enables certain type information be inferred about f, which is sufficient to define the tagged small function extension.

For a *valid* domain extension family F, and a dom(F) element x, let tagged(F, x) = tagged(domExt(F), x).

For a simple tagged small function extension f, let dom(f) = dom(untag(f)).

Let domExt(null) = Dom.Null, domExt(zero) = Dom.Null, and domExt(one) = Dom.Nuro. For a pair small function extension p, let domExt(p) = Dom.Leaf.

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

For a tagged small function extension f, and a dom(f) element x, let tagged(f, x) = tagged(domExt(f), x).

For a pair tagged small function extension p, and a dom(p) element x, let p<x> be as follows:

- null if x = null
- left(p) if x = zero
- right(p) if x = one

2.4 Coercion

Tagged small function extensions are sufficiently constrained to enable coercion. For a *valid* domain extension A, and a tagged small function extension f, the coercion to A of f (a dom(A) element), denoted by coer(A, f), is as follows:

- If A is a constant domain extension: coer(A, f) is untag(f) if untag(f) is a dom(A) program; and null otherwise.
- If A is a dependent sum domain extension, 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(domExtFam(A)), left(f)) and right(p) = coer(domExtFam(A)<left(p)>, right(f)). Note that left(f) is coerced first, then right(f) is coerced.
- If A is a dependent sum domain extension, and f is *not* a pair tagged small function extension: coer(A, f) = null.
- If A is a dependent product domain extension, and f is a rule tagged small function extension: coer(A, f) is the rule small function extension r such that dom(r) = dom(domExtFam(A)) and, for each dom(r) program x, r<x> = coer(domExtFam(A)<x>, f<coer(domExt(f), tagged(domExtFam(A), x))>). Note the pre-coercion before the call to f, and the post-coercion after the call to f. Pre-coercion and post-coercion adjust the domain and codomain of f, respectively.
- If A is a dependent product domain extension, and f is *not* a rule tagged small function extension: coer(A, f) = null.

The well-founded relation used to define coercion recursively is demonstrated in [9].

For a *valid* domain extension family F, and a tagged small function extension x, let coer(F, x) = coer(domExt(F), x).

For tagged small function extensions f and x, let coer(f, x) = coer(domExt(f), x).

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

The coercion stability theorem proves that coercion does not make unnecessary changes: 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).

2.5 Result

Coercion is now used to address the challenge of generalizing the definition of result. For tagged small function extensions f and x, the result of f at x, denoted by f(x), is f<coer(f, x)>.

2.6 Identity tagged small function extensions

For a *valid* domain extension A, the identity tagged small function extension on A, denoted by i(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 Null.set = i(Dom.Null), Nuro.set = i(Dom.Nuro), Leaf.set = i(Dom.Leaf), and Tree.set = i(Dom.Tree).

For a tagged small function extension f, the domain tagged small function extension of f, denoted by domFuncExt(f), is i(domExt(f)).

For a *valid* domain extension family F, let sumDep(F) = i(A) where A is the dependent sum domain extension containing F. For a *valid* domain extension family F, let prodDep(F) = i(A) where A is the dependent product domain extension containing F.

In NummSquared, domain extensions may be created from tagged small function extensions. 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, the dependent sum of f, denoted by sumDep(f), is sumDep(domExtFam(f)). For a tagged small function extension f, the dependent product of f, denoted by prodDep(f), is prodDep(domExtFam(f)).

2.7 Large function extensions and truth

Tagged small function extensions permit abstraction over those *small* sets of small function extensions that can be represented by domain extensions. However, for greater generality, it is often necessary to abstract over *all* tagged small function extensions - large function extensions provide this generality. Of course, a large function extension is not a small function extension - if it were, the troubles of the untyped lambda calculus would recur. Somewhat similar to NummSquared large functions, von Neumann [14] has functions that cannot be used as an argument or result.

A large function extension f contains, for each tagged small function extension x, the result of f at x (a tagged small function extension), denoted by f(x).

The result of a tagged small function extension f, denoted by res(f), is f(null). The choice of null is arbitrary - some argument must be supplied in order to start computation.

A tagged small function extension x is true iff x = one. A large function extension f is true iff, for each tagged small function extension x, f(x) is true. Thus a large function extension represents a universally quantified proposition extension.

NummSquared uses only a few built-in large functions and ways of combining large functions. Small functions do not appear in NummSquared programs, but enter indirectly because of their role as arguments and results of large functions.

For a tagged small function extension y, let Lg.constant(y) be the large function extension such that, for each tagged small function extension x, Lg.constant(y)(x) = y.

Let Lg.i be the large function extension such that, for each tagged small function extension x, Lg.i(x) = x.

Let Lg.null = Lg.constant(null).
Let Lg.zero = Lg.constant(zero).
Let Lg.one = Lg.constant(one).
Let Lg.Null.set = Lg.constant(Null.set).
Let Lg.Nuro.set = Lg.constant(Nuro.set).
Let Lg.Leaf.set = Lg.constant(Leaf.set).
Let Lg.Tree.set = Lg.constant(Tree.set).

Let Lg.Null be the large function extension such that, for each tagged small function extension x, Lg.Null(x) is one if x = null; and zero otherwise.

Let Lg.Pair be the large function extension such that, for each tagged small function extension x, Lg.Pair(x) is one if x is a pair tagged small function extension; and zero otherwise.

Let Lg.dom be the large function extension such that, for each tagged small function extension x, Lg.dom(x) = domFuncExt(x).

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)).

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)). Note that the outermost call is a call to a tagged small function extension.

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)}.

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)).

A large function extension may be Curried. The Curried function returns a partial call to the original function. However, because the partial call is a tagged small function extension, not a large function extension, its domain must be restricted. 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)}).

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 as follows:

- elseP(x) if ifP(x) = zero
- thenP(x) if ifP(x) = one
- null if ifP(x) is not Boolean

The well-foundedness of NummSquared enables a terminating recursion principle. 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 as follows:

- If x = null: ~r[start step](x) = start(x).
- If x ≠ 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 rDom is the restriction of ~r[start step] to the domain of 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 rRan is the restriction of ~r[start step] to the range of x.

NummSquared equals is not computable, but is necessary for logic. Let Lg.eq be the large function extension such that, for each tagged small function extension p, Lg.eq(p) is as follows:

- one if p is a pair tagged small function extension, and left(p) = right(p)
- zero if p is a pair tagged small function extension, and left(p) $\neq$ right(p)
- null if p is *not* a pair tagged small function extension

Hilbert's epsilon operator is a version of the axiom of choice, and existential and universal quantification can be defined from it - see Avigad [2]. In NummSquared, Hilbert (an adaptation of Hilbert's epsilon operator) is not computable, but is necessary for logic. 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 null otherwise.

3 NummSquared syntax

A few aspects of NummSquared syntax are described. The complete syntax may be found in [9].

3.1 Normalized large functions

Normalized large functions are the syntactic counterpart to large function extensions. Normalized large functions are divided into normalized constants and normalized combinations. Normalized constants are further divided into computational normalized constants and non-computational normalized constants. Normalized combinations are further divided into computational normalized combinations and non-computational normalized combinations.

The computational normalized constants, and their concrete syntax representations, are as follows:

- Constant.Norm.Compu.i - `~i`
- Constant.Norm.Compu.null - `~null`
- Constant.Norm.Compu.zero - `~zero`
- Constant.Norm.Compu.one - `~one`
- Constant.Norm.Compu.Null.set - `~Null.set`
- Constant.Norm.Compu.Nuro.set - `~Nuro.set`
- Constant.Norm.Compu.Leaf.set - `~Leaf.set`
- Constant.Norm.Compu.Tree.set - `~Tree.set`
- Constant.Norm.Compu.Null - `~Null`

- Constant.Norm.Compu.Pair - ~Pair
- Constant.Norm.Compu.dom - ~dom

The non-computational normalized constants, and their concrete syntax representations, are as follows:

- Constant.Norm.Noncompu.eq - ~=

The computational normalized combinations are as follows: (all components are normalized large functions)

- a large composition computational normalized combination f contains outer(f) and inner(f), and is denoted by [outer(f) inner(f)]
- a small composition computational normalized combination f contains called(f) and arg(f), and is denoted by (called(f) arg(f))
- a pair computational normalized combination f contains left(f) and right(f), and is denoted by {left(f) right(f)}
- a dependent sum computational normalized combination f contains family(f), and is denoted by ~s.d[family(f)]
- a dependent product computational normalized combination f contains family(f), and is denoted by ~p.d[family(f)]
- a Curry computational normalized combination f contains uncurry(f) and restrictor(f), and is denoted by ~c[uncurry(f) restrictor(f)]
- an if-then-else computational normalized combination f contains ifP(f), thenP(f) and elseP(f), and is denoted by ~ite[ifP(f) thenP(f) elseP(f)]
- a recursion computational normalized combination f contains start(f) and step(f), and is denoted by ~r[start(f) step(f)]

The non-computational normalized combinations are as follows: (all components are normalized large functions)

- a Hilbert non-computational normalized combination f contains pred(f), and is denoted by ~h[pred(f)]

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

Pairs are used to construct tuples. For a natural number $m \geq 2$, and normalized large functions $x_0, x_1, ..., x_{m-2}, x_{m-1}$, let $\{x_0\ x_1\ ...\ x_{m-2}\ x_{m-1}\} = \{\{\{x_0\ x_1\}\ ...\ x_{m-2}\}\ x_{m-1}\}$.

Pairs are used to construct lists. 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$\}\}\}$.

Tuples are used to simulate multi-argument normalized large functions. For a natural number $m \geq 2$, and normalized large functions f and $x_0, x_1, ..., x_{m-1}$, let $[f\ x_0\ x_1\ ...\ x_{m-1}] = [f\ \{x_0\ x_1\ ...\ x_{m-1}\}]$.

For a normalized large function f, the extension of f, denoted by ext(f), is as follows:

- Lg.i if f = Constant.Norm.Compu.i
- the other normalized constant cases are similar and are omitted
- [ext(outer(f)) ext(inner(f))] if f is a large composition computational normalized combination

- the other normalized combination cases are similar and are omitted

Normalized large functions need not be in simplest form. For example, [Constant.Norm.Compu.i Constant.Norm.Compu.i] ≠ Constant.Norm.Compu.i, even though both normalized large functions have the extension Lg.i.

The result of a normalized large function f, denoted by res(f), is res(ext(f)).

A normalized large function f is true iff ext(f) is true.

3.2 Reduction

The concept of reduction in NummSquared corresponds to the computed of a normalized large function, which is now defined. Termination of reduction is ensured by defining reduction directly as a computable total function.

A normalized large function f is deep computational iff f contains, directly or indirectly, only computational normalized constants and computational normalized combinations.

For a tree tagged small function extension x, the normal form of x, denoted by norm(x), is as follows:

- Constant.Norm.Compu.null if x = null
- Constant.Norm.Compu.zero if x = zero
- Constant.Norm.Compu.one if x = one
- {norm(left(x)) norm(right(x))} if x is a pair tagged small function extension

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 ∅ otherwise. For a normalized large function f, the computed of f, denoted by computed(f), is resNorm(f) if f is deep computational; and ∅ otherwise. Note that computed is a computable total function. Computation generates an error if the normalized large function is not deep computational or if its result is not a tree. Typically, the output of software is a tree, even though rules may be used to compute the output. However, in future, the tree restriction might be removed by defining the normal form of a rule tagged small function extension.

3.3 Reflection

Clearly the NummSquared programmer will want to define new ways to combine normalized large functions. Rather than introducing super-large functions, NummSquared uses reflection to abstract over all normalized large functions. A language supports reflection iff programs of the language can manipulate (to some extent) programs of the language.

NummSquared quotation converts a normalized large function to a tree representation, and unquotation is the inverse process. Unquotation is effectively the interpreter for normalized large functions, and therefore cannot be called within normalized large functions. Removing this restriction would entail the troubles of the untyped lambda calculus - see Hoare [7]. Therefore, NummSquared reflection allows normalized large functions to perform only syntactic manipulation of normalized large functions, i.e. operations that do not call the interpreter for normalized large functions. Syntactic manipulation is sufficient for common usage of macro languages. Thus NummSquared is its own macro language.

Quotation and unquotation in NummSquared have some conceptual similarities to Gilmore's implicit quotation [5], although quotation is explicit in NummSquared. The fact that NummSquared is variable-free simplifies quotation and unquotation.

For a natural number m, the normal form of m, denoted by norm(m), is as follows:

- Constant.Norm.Compu.zero if m = 0
- Constant.Norm.Compu.one if m = 1
- {norm(m - 1) Constant.Norm.Compu.null} if m ≥ 2

Quotation transforms a normalized large function into a tree containing a tag and a child list. For a natural number tag, and a normalized large function children, let tree(tag, children) be {norm(tag) children}.

For a normalized large function f, let tag(f) be as follows:

- 0 if f = Constant.Norm.Compu.i
- the other normalized constant cases are similar and are omitted
- 12 if f is a large composition computational normalized combination
- the other normalized combination cases are similar and are omitted

For a normalized large function f, the quoted of f (a normalized large function), denoted by quoted(f), is as follows:

- tree(tag(f), ~l{}) if f is a normalized constant
- tree(tag(f), ~l{quoted(outer(f)) quoted(inner(f))}) if f is a large composition computational normalized combination
- the other normalized combination cases are similar and are omitted

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 ∅ otherwise. A normalized large function f is quoted iff unquoted(f) ≠ ∅.

Macro expansion is a shortcut for quotation, computation and unquotation - these three combined are useful for syntactic manipulation of normalized large functions. For a list l = l<x_0, x_1, ..., x_{m-1}> of normalized large functions, quoted(l), is ~l{quoted(x_0) quoted(x_1) ... quoted(x_{m-1})}. For a normalized large function f, and a list l of normalized large functions, the macro pre-expanded of f at l, denoted by macroPreexpanded(f, l), is [f quoted(l)]. For a normalized large function f, and a list l of normalized large functions, the macro expanded of f at l, denoted by macroExpanded(f, l), is ∅ if computed(macroPreexpanded(f, l)) = ∅; and unquoted(computed(macroPreexpanded(f, l))) otherwise.

3.4 Large functions

Theoretically, NummSquared programs could be written as normalized large functions - but they would be very difficult to read. Large functions offer a practical and simple syntax that is normalized to obtain normalized large functions. Details are in [9].

3.5 Proofs

NummSquared proofs are constructed from many axioms, and a few modes of inference. A soundness theorem relating proof and truth has been proved. Proofs use classical logic. Proofs are quoted and unquoted in a similar way to normalized large functions, allowing normalized large functions to manipulate NummSquared proofs. Details are in [9].

4 Example

A small example is given demonstrating the flexibility of NummSquared compared to type theory. Coq 8.0pl3 [4] is used as a representative of type theory. Consider the following Coq program:

```
Inductive Null : Type :=
|   Null_null : Null.

Inductive Nuro : Type :=
|   Nuro_null : Nuro
|   Nuro_zero : Nuro.

Definition f_res_Ty := fun x y : Nuro =>
match x
with
|   Nuro_null => Null
|   Nuro_zero =>
        match y
        with
        |   Nuro_null => Null
        |   Nuro_zero => Nuro
        end
end.

Definition f := fun x y : Nuro =>
match x
return (f_res_Ty  x  y)
with
|   Nuro_null => Null_null
|   Nuro_zero =>
        match y
        return (f_res_Ty  Nuro_zero  y)
        with
        |   Nuro_null => Null_null
        |   Nuro_zero => Nuro_zero
        end
end.

Definition g := fun
(h : forall x y : Nuro, (f_res_Ty  y  x))
(x : Nuro) =>
(h  x  x).
```

Consider adding the following to the end of the Coq program:

```
Definition z := (g  f  Nuro_zero).
```

Coq gives the following error:

```
Error: The term "f" has type
"forall x y : Nuro, f_res_Ty x y"
while it is expected to have type
"forall x y : Nuro, f_res_Ty y x"
```

The reason for the error is that Coq requires a proof that `f_res_Ty x y` equals `f_res_Ty y x` in order for the call to `g` to type check. Of course, such a proof should ultimately be provided. However, the programmer may want to do the proof later, particularly during the rapid prototyping phase. Even if the programmer assumes the proof as a Coq axiom, the definition of `z` is still made more complex.

As shown below, the definition of `z` without proof is legal in NummSquared, and the proof may be done later (or not at all, although that is not recommended). Despite not requiring proof, NummSquared ensures termination and averts paradoxes. In future, NummSquared may have a way of automatically tracking proofs that should be done later.

Consider the following NummSquared program. `{%x %y}` is a local tuple accessor list, which is a replacement for argument variables since NummSquared is variable-free.

```
f.res.set {%x %y} =
~ite[
    [~Null  %x]
    ~Null.set
    ~ite[
        [~Null  %y]
        ~Null.set
        ~Nuro.set
    ]
];

f {%x %y} =
~ite[
    [~Null  %x]
    ~null
    ~ite[
        [~Null  %y]
        ~null
        ~zero
    ]
];

g {%h %x} = (%h  %x  %x);
```

`f` is a large function and therefore cannot be an argument to `g`, so `f` must be Curried to obtain a small function.

`~right` is defined as follows in [9]:

```
~right =
~ite[
    ~Pair
    (~i  1)
```

```
    ~null
];
```

`~restrict` is defined as follows in [9]: For a normalized large function `unrestrict:

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

For a natural number $m \geq 1$, a normalized large function uncurry, and normalized large functions x_0, x_1, ..., x_{m-1}, let c[uncurry x_0 x_1 ... x_{m-1}] = [~restrict[~c[...~c[uncurry x_{m-1}] ... x_1]] x_0].

```
f.c = c[f  ~Nuro.set  ~Nuro.set];

z0 = [g  f.c  ~zero];
```

Alternately, `g` can be Curried as well.

For a natural number $m \geq 1$, a normalized large function uncurry, and normalized large functions x_0, x_1, ..., x_{m-1}, let p.d[uncurry x_0 x_1 ... x_{m-1}] = ~p.d[[~restrict[~p.d[~c[...~p.d[~c[uncurry x_{m-1}]] ... x_1]]] x_0]].

```
f.rev.res.set {%x %y} =
[f.res.set  %y  %x];

f.rev.set =
p.d[f.rev.res.set  ~Nuro.set  ~Nuro.set];

g.c = c[g  f.rev.set  ~Nuro.set];

z1 = (g.c  f.c  ~zero);
```

5 Conclusion

NummSquared semantics and syntax have been overviewed. Advantages of NummSquared include:

- An untyped system that minimizes constraints on the programmer, while still ensuring termination and averting paradoxes, without the need for the programmer to supply proofs
- Just one fundamental concept: higher-order functions
- No side-effects or global state
- Reflection which allows NummSquared to be its own macro language, and allows NummSquared programs to manipulate NummSquared proofs
- A simple variable-free syntax with some familiar elements from lambda calculi
- Well-foundedness and use of classical logic, which are both widely used in mathematics and logic

Future work on NummSquared will involve completing NsGo (the NummSquared interpreter), developing NummSquared libraries, and using NummSquared for software projects.

References

[1] Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006. URL http://www.cs.nott.ac.uk/~txa/.

[2] 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/.

[3] 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.

[4] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr/.

[5] Paul C. Gilmore. *Logicism Renewed: Logical Foundations for Mathematics and Computer Science*. Association for Symbolic Logic & AK Peters, 2005.

[6] Klaus Grue. Map theory with classical maps. WWW site, June 2001. URL http://www.diku.dk/~grue/.

[7] C. A. R. Hoare and D. C. S. Allison. Incomputability. *Computing Surveys*, 4(3), September 1972.

[8] Douglas J. Howe. A classical set-theoretic model of polymorphic extensional type theory. URL http://citeseer.ist.psu.edu/howe97classical.html. 1997.

[9] Samuel Howse. *NummSquared 2006a0 Done Formally*. Poohbist Technology, October 2006. URL http://nummist.com/poohbist/. October 18, 2006 pre-release.

[10] Roger Bishop Jones. Pure functions. WWW site, December 1998. URL http://www.rbjones.com/rbjpub/logic/inter013.htm.

[11] *The F# Manual*. Microsoft Corporation, 2006. URL http://research.microsoft.com/fsharp/manual/default.aspx.

[12] 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.

[13] Gaisi Takeuti and Wilson M. Zaring. *Introduction to Axiomatic Set Theory*. Springer-Verlag, 2^{nd} edition, 1982.

[14] 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.