A smoλ theory of structural and nominal typing

Emmanouil Krasanakis - maniospas@hotmail.com

Version 1 - June 16, 2025

Abstract

Structural and nominal typing are two significantly different systems for organizing data types. The former simplifies program writing by reusing the same data layout across different types. It is also closer to performant assembly instructions that do not account for high-level semantics. On the other hand, nominal typing introduces safety guarantees by enforcing relationships between data stored in different segments of the layout. This paper introduces a theoretical comparison and combination of the two systems, leading to the implementation of the smoλ language. In particular, analysis conducted in set theory to capture the full capabilities of structural typing shows that nominal types are severely limited in terms of expressiveness. A structural type system is proposed to both match primitive types (e.g., integers, doubles) and make nominal ones behave as pseudo-primitives with safety guarantees. The system works by treating type names as primitive types and structurally unpacking them.

1 Introduction

When I first started on this monograph, my goal kept changing between writing a proper paper to creating formal verification of the smoλ language. In the end, a third option appealed to me: a short discourse on structural vs nominal typing. This echoes past attempts at unifying the two systems [1, 2, 3], but there is something new here: working with set theory to treat data structure as the most important. A retroactive justification is that the this is inherently more expressive, and therefore it makes sense to keep the benefits of primitive tuple representations by eschewing lambda calculus or linear logic.[1]

Before going further, differences between the two compared systems can be summarized in structural memory layouts being intrinsic to how low-level programming languages treat memory vs referencing types based on only on their names -hence being *nominal*- to achieve high-level safety.[2] For example, a nominal string type can ensure that the *size* segment always corresponds to the amount of allocated memory referenced in the *pointer* segment, without pairing

[1] Category theory may be a valid generalization but it does not support set cardinality.

[2] This is partially orthogonal to the well-explored distinction of dynamic vs static type checking, though there are many common axes.

arbitrary pointers to invalid sizes. Conversely, the following smoλ language program demonstrates structural typing, where the two numeric inputs to *norm* are treated as the layout of *Point2D*. Notice usage of the language's currying notation where $arg : fun(other)$ translates to $fun(arg, other)$.

```
@include std.builtins
@include std.math

smo Point2D(f64 x, f64 y) -> x,y  // basically a type
smo norm(Point2D p)
    sqx = p.x:pow(2.0)
    sqy = p.y:pow(2.0)
    -> pow(sqx+sqy, 0.5)

service main()
    m = norm(3.0, 4.0)
    print(m)  // prints 5.0
    --        // end without return
```

Listing 1: Structural typing example

Notation

In analysis below, $\times$ is the cartesian product between sets and $\oplus$ the ordered apposition of tuples, e.g., $(t_1, t_2) \oplus (t_3, t_4) = (t_1, t_2, t_3, t_4)$. A^* is the Kleene star notation. Capital letters denote sets, calligraphy is added for sets of sets, and lower letters represent functions.

2 Nominal as a less expressive structural typing

Consider a discrete setoid[3] L whose members will later play the role of nominal type names. Also consider a set containing unary sets of names $\mathcal{L} = \{\{l\} : l \in L\}$ and a superset $\mathcal{L} \subseteq \mathcal{A}_0$ that contains that and potentially more sets that play the role of primitive types. The superset is not necessarily a setoid. For instance, $L = \{1, 2, \dots\}$ or an uncountable set of sets of strings, whereas $\mathcal{A}_0$ could also contain the sets of integers, reals, or their discrete (e.g., 64-bit) counterparts.

Theoretically, the setoid of names could have finite or infinite cardinality. In practice, $|L|$ will not only be finite but also upper bounded by some software development cost (measured in time, effort, etc.); costs scale monotonically by at least a fixed overhead compared to the amount of implemented code. So, by considering that at least one code segment needs to be added for each name when implementing corresponding nominal types later, there is an upper bound to how many labels are defined. Still, it is important to get a sense of expressive power as the cardinality of the type system's distinct expressions.

We will work with the smoλ concept called runtypes, which are essentially functions whose output tuples act as types. In language examples, they are

[3] A discrete setoid is a set equipped with an equivalence relation $\sim$ between its members and it holds that $|\{l' \in l : l' \sim l\}| = 1$ for all $l \in L$. Treat $\sim$ identically to $=$.

defined with the *smo* keyword and their outputs are treated as either tuples with named elements or single values if there is only one element. Runtypes are inherently structural, but also admit nominal variations that are associated with names, as well as a type system that preserves safety (more later).

Definition 1. *A runtype is a function over tuples of primitive types:*

$$t : I \to A \text{ where } I, A \in \mathcal{A}_0^*$$

For brevity, $t(\cdot)$ will represent the set of all potential values $t(I) \subseteq A$ given that I is the input space of the runtype. Also, let $\mathcal{T}$ be the set of all runtypes, and annotate the set of non-name primitive identities as:

$$\mathcal{T}_0 = \{t : A \to A, t(x) = x : \mathcal{A}_0 - \mathcal{L}\}$$

The next definition establishes that a modeling comprising several nominal runtypes is the act of matching them to names. Then, the two accompanying theorems show that it is impossible to represent all structural runtypes with nominal variations. This holds true under the premise that structural typing has access to the names, for example during compile time checks that will be later eliminated (as smoλ does), by reflection that exposes compile-time information to the runtime or, even more ubiquitously, or by languages supporting runtime polymorphism.

When one selects or implements a specific collection of nominal runtypes to serve as modeling, they effectively establish a restriction on expressive power that enforces certain semantics. This is similar to constructing a domain-specific language (DSL) for safe operations, where safety corresponds to tying a specific runtype output to the process of generating it - that in turn could correspond to practices like safe memory management. New nominal types may also be obtained by combining previous ones, but we are ultimately restricted by our (in)ability to express more new names from the label set L.

More generally, the last theorem reveals that the collection of all runtypes will always be at least one aleph -or, in terms of logic theory, an abstraction order- greater than nominal ones. In other words, they demonstrate significantly reduced expressive power, limited by the very need to name them individually. Conversely, much more complex structural types can be quickly obtained by quickly combining simpler ones.

Definition 2. *A set $\mathcal{T}_{nom}$ will be called a nominal modeling if there exists bijection $nom : \mathcal{T}_{nom} \to nom(\mathcal{T}_{nom}) \subseteq L$, and the set's members, which will be called nominal runtypes, take the form*

$$t : \{nom(f)\} \times I \to \{nom(f)\} \times A \text{ where } I, A \in \mathcal{A}_0^*$$

Theorem 1. *Any set $\mathcal{T}_b \subseteq \mathcal{T}$ with $|\mathcal{T}_b| \leq |L|$ generates a modeling of nominal runtypes for some $nom_b : \mathcal{T}_b \to L$:*

$$\mathcal{T}_{nom} = \{t \in \mathcal{T} : t(nom(t) \oplus i) = (nom(t) \oplus t_b(i)), nom(t) = nom_b(t_b)\}$$

Proof. It suffices to show that a bijection $nom_b : \mathcal{T}_b \to nom_b(\mathcal{T}_b) \subseteq L$ exists, which holds by definition for finite sets. It also exists for infinite sets given the axiom of choice. □

Theorem 2. *For infinite names,* $2^{|\mathcal{T}_{nom}|} \leq |\mathcal{T}|$. *For finite non-zero names,* $\mathcal{T}$ *is at least countable.*

Proof. For infinite names we get $|\mathcal{T}| \geq |\mathcal{A}_0^*|^2 \geq |\mathcal{A}_0^*| \geq 2^{|A_0|} \geq 2^{|L|} \geq 2^{|\mathcal{T}_{nom}|}$. For finite names, as long as at least one exists, $\mathcal{A}_0^*$ is countable. □

Example

As an example of how nominal types can be expressed as part of a structural type system, consider the following smoλ code. We first define two runtypes that construct a point from Cartesian and circular coordinates and corresponding *norm* functions. I will not go into type inference details here but, to apply the proper version of normalization, the compiler should be able to distinguish between which point type is analyzed. To make the distinction possible, the *nom* keyword is provided.

This is similar to the *nom* function in that it automatically creates a corresponding name and substitutes itself with a type of a set containing only that name. Under the hood the name is represented as a unique integer, and the language allows setting it only as the first keyword. To make the calling convention similar to other runtypes, which can be used to zero-instantiate nameless arguments, *nom* can also be used as a first argument. In this which case, it automatically matches the name of the runtype being called.

Given that *nom*'s set/type is different in *Point2D* and *Point2R*, there is now a clear distinction between the two despite having the same trailing structure (two doubles). From a language design standpoint, smoλ matches its syntax to this section's theory in favor of implementation transparency. It also goes a step further in optimizing away any uneeded operations, so *nom*-based type checking occurs at compile time but remains a zero-cost abstraction without running time impact. Lastly, the example also contains a *Field* definition that is unused but demonstrates how nominal types can also depend on other nominal ones; the name is still obtained from the same number sequence.

```
1  @include std.builtins
2  @include std.math
3
4  smo Point2D(nom, f64 x, f64 y) -> @new  // tuple of all arguments
5  smo Point2R(nom, f64 r, f64 theta) -> @new
6  smo Field(nom, Point2D start, Point2D end) -> @new
7  smo norm(Point2D p)
8      sqx = p.y:pow(2.0)
9      sqy = p.x:pow(2.0)
10     -> pow(sqx+sqy, 0.5)
11 smo norm(Point2R p) -> p.r
12
13 service main()
14     pd = nom:Point2D(3.0, 4.0)
```

```
15    pr = nom:Point2R(1.0, 180.0)
16    print(pd:norm)  // prints 5.0
17    print(pr:norm)  // prints 1.0
```

Listing 2: Nominal semantics in structural declarations

3 Safety of nominal typing

We thus verified the limitations of the naming scheme. But we also need to show its advantages. Mainly, can we use it to guarantee safety in interpreting safety-critical data in the same way every time? The next definition is a requirement for logical safety in that sense. That is, the output of the same nominal runtypes cannot be reinterpreted, guarding the original logical or programming safety they imposed. A type declaration system that admits that unique interpretation is also shown. The system consists of runtype inputs incrementally being constructed from different runtypes. These results are trivial given that *nom* is essentially a way for naming our runtypes; the novelty is in terms of design that interweaves nominal and structural typing in parts of definitions.

Definition 3. *A set of runtypes will be called safe under nominal modeling* $\mathcal{T}_{nom}$ *only if any runtype's* t *output can be* ***uniquely*** *decomposed into*

$$t(\cdot) = a_1(\cdot) \times a_2(\cdot) \times \dots \text{ where } a_1, a_2, \dots \in \mathcal{T}_0 \cup \mathcal{T}_{nom}$$

Proof. Trivial given that nominal runtypes require the corresponding name □

Theorem 3. *The following recursively defined set of runtypes is safe*

$$\mathcal{T}_{closure} = \mathcal{T}_0 \cup \mathcal{T}_{nom} \cup \{t : I \to A : I, A \in \mathcal{T}^*_{closure}\}$$

Finally, let us consider the cost of safety. There is no cost in expressive power as long as we provide nominal runtypes for all names! Note that the equality is in terms of infinity cardinalities, so there may be some runtypes that are not expressible, but we can theoretically set up a bijection between $\mathcal{T}_{closure}$ to $\mathcal{T}$ to retrieve the missing ones; that could be hard to find but it exists.

Theorem 4. *If* $|\mathcal{L}| = |\mathcal{T}_{nom}|$ *then* $|\mathcal{T}_{closure}| = |\mathcal{T}|$

Proof. It holds that $\mathcal{T}_{closure} \subseteq T$ therefore $|\mathcal{T}_{closure}| \leq |\mathcal{T}|$. But it also holds that $|\mathcal{T}_{closure}| \geq |((\mathcal{T}_0 \cup \mathcal{T}_{nom})^2)^*| = |((\mathcal{T}_0 \cup \mathcal{L})^2)^*| = |(\mathcal{A}_0^2)^*| = |\mathcal{T}|$. □

Example

As an example of safety, look at part of smoλ's vector implementation in the standard library. This uses *@body* and *@finally* to embed C++ code for code running and resource deallocation respectively, as well as the *fail* command to gracefully exit the currently running service - services are runtypes with an extra error handling field and automatic resource deallocation. The important

aspect of this implementation is that all operations process the nominal vector runtype that ties the correct size to allocated memory but still exhibit structural characteristic for other fields. For demonstration purposes, the implementation is accompanied by a structural runtype call. In general, *vec* imposes critical safety considerations, where multiple features like this can be combined in exponentially many safe usage scenarios.

```
@include std.builtins

smo vec(nom, ptr contents, u64 size) -> @new
smo vec(u64 size)
    @body{ptr contents = new f64[size]();}
    @finally contents {if(contents)delete[] (f64*)contents;
    contents=0;}
    -> nom:vec(contents, size)
smo len(vec v) -> v.size
smo at(vec v, u64 pos) // overloads []
    if pos>=v.size -> fail("Vec out of bounds")
    @body{f64 value = ((f64*)v__contents)[pos];}
    -> value
smo set(vec v, u64 pos, f64 value)
    if pos>=v.size -> fail("Vec out of bounds")
    @body{((f64*)v__contents)[pos] = value;}
    -> v
smo dot(vec x1, vec x2)
    if x1.size!=x2.size -> fail("Incompatible vec sizes")
    &sum = 0.0 // mutable
    &i = 0
    while i<x1.size
        sum = sum+x1[i]*x2[i]
        i = i+1
        -- // end block
    -> sum

smo one_hot(u64 idx) -> idx, 1.0
service main()
    x = vec(1000)
    x:set(1:one_hot)
    x:set(5, 2.0)
    print(x:dot(x))  // prints 5.0
```

Listing 3: Nominal type safety

4 Takeaways

Before closing this short manuscript, let us reexamine what we learned. First, we have a unifying framework to treat nominal typing as a case of structural typing with less expressive power; it is only a matter of uniquely "naming" a subset of producible structural types, that is, creating a bijection between that subset and name candidates. As a side-effect of the analysis we also got a theoretical model that can analyze uncountable names, if that matters in the future.

Second, we saw what a type system that uses both nominal and structural types while maintaining nominal constraints looks like; basically, it consists of a simple composition of flattened type arguments, with nominal labels interweaved to match respective positions. This could be optimized away by compilers or, equivalently, nominal types can be treated as new primitives. Finally, such a type system does not lose expressing power compared to an unconstrained counterpart that uses an equal number of labels.

Going forward, I believe that adopting similar combinations of structural and nominal typing can help produce more semantically rich programming languages that retain safety guarantees and speedy execution. For example, if an exponential number of nominal types is required to reproduce the expressiveness of fewer number structural ones (e.g., because they need to be explicitly combined), it may be better to structural representation of type segments without critical safety concerns.

Acknowledgements

ChatGPT4o helped a lot with formatting and proofreading to point out areas of improvement.

Code

Find smoλ's repository at `https://github.com/maniospas/smol` .

References

[1] T. Kühne, "Unifying nominal and structural typing," *Software & Systems Modeling*, vol. 18, pp. 1683–1697, 2019.

[2] D. Malayeri and J. Aldrich, "Integrating nominal and structural subtyping," in *European Conference on Object-Oriented Programming*. Springer, 2008, pp. 260–284.

[3] F. Muehlboeck and R. Tate, "Transitioning from structural to nominal code with efficient gradual typing," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.