NovelML: A Novel SML Semantics

tbrick (the t stands for thea ♡), jgallicc (the jga stands for jgay)

$LOCKDOWN + (3*365.25 + 16)*24*60*60*1000

1 Abstract

Previous work has shown the potential of taking Python and transforming it into an ML-like language. In this novel work, we consider the categorical dual[1], where we introduce a novel semantics for ML, modeled on the performant and industry proven pythonic approach.

2 Introduction

In prior SIGBOVIK proceedings, Ng *et al vel non est alius* 2022, proposed taking "the opposite approach to the popular literature" and in doing so, "develop[ed] a revolutionary new system which makes the Python language Slightly More Likeable." In our novel paper, we take the opposite of the opposite approach to popular literature. This is novel research because we are intuitionists, and reject double negation elimination. Thus this research is new and novel and therefore worth publishing.

In this novel paper, we ask the following novel question: Why should I waste my time statically determining types when our program already knows the type at runtime. We replace SML's outdated 1997 typechecker with fancy and novel dynamic checks – which are critically acclaimed in industry.

Given that this approach is critically acclaimed in industry, our novel approach had industry applications before publication, thus making it worth studying (in addition to being novel).

3 A Novel Definition for NovelML

3.1 Syntax

The Definition™ of Standard ML™(1997™) provided an (at the time) novel syntax. Given the perfection of this original syntax for SML (specifically the `abstype` declaration and the wonderful `infix` and `infixr` declarations).

[1]in whatever Grothendieck universe this terminology makes sense in

	expressions	statements
statics	$O(1)$	$O(1)$
dynamics	$O(1_{\pm 2^n})$	$O(1)$

Figure 1: Asymptotic complexities of the statics/dynamics of expressions/statements (measured in milliseconds)

In order to support the novel features of our new language, we must extend the existing Basis Library[2]. After much literature review (which is what paper's that should be published do), we determined that following CMU Alumnus and creator of Java, James Gosling's approach would work best. So we introduce the keyword `novel.machine.learning.lang.System.Runtime.Type.f''''` which implements casting and hopefully balances the familiarity of Java (very useful and informative namespaces) with common ML-isms (mysterious variable names and lots of `'`s). We were going to call it `Unsafe.cast`, but SML/NJ already took that one. Then we were going to call it `Object.magic`, but Jane Street has already done enough damage to OCaml that we felt using it would be inappropriate[3] .

3.2 Novel Statics

Our novel typing rules are straightforward for expressions, and complicated for everything else. First the simple stuff:

```
---------------------- [expr]
    |- e : \tau
```

This novel semantics improves on Ng *et al vel non est alius* 2022, by dropping the context, which is no longer needed for typechecking expressions.

Typechecking declarations is more complicated:

```
-------------- [statement]
   |- s valid
```

Oh, okay, nevermind then. See? NovelML is elegant.

3.3 Novel Dynamics

NoML has simple expression statics and simple declaration statics, but in contrast has complicated expression dynamics and simple declaration dynamics (see Figure1).

To define NoML's dynamics, we need only add one novel exception to those presented in **The Definition**™™. This exception, `No` (short for "not well-typed"), indicates a runtime type error. Here are the simple declaration dynamics:

[2]we would cite a source here but we couldn't find any documentation for SML's basis library.

[3]See this based[4]tweet

[4]based on what?

1. `datatype`, `type`, `abstype` declarations are meaningless in NoML, and thus raise `No`[5]
2. `structure`, `signature`, `functor` declarations are confusing in NoML, and thus raise `No`[6]
3. `infix` is a parser flag (the syntax of SML is context sensitive). Thus it does not raise `No`, but is a no-op (novel op) in the dynamics.
4. `val` and `fun` are standard variable binders. See section "Novel Variables" for details.

The complicated expression dynamics are basically the same as SML but if there's a type error at runtime, raise `No`. See the examples section for more details.

Due to difficulties in implementing NoML, we also had to add a maximum recursion depth with default value 1000. This depth can be changed by binding to `setrecursionlimit`:

```
val setrecursionlimit = setrecursionlimit + 1/2
```

4 Novel Variables

A common confusion when learning Standard ML is variable bindings, in particular the notion that variables *shadow* other variables. In this novel language, we follow legendary former president George W. Bush Jr.'s philosophy of "No child left behind", adapted to the novel catchphrase: "No variable left in the shadows."

Variables (sometimes synonymously called assignables) are re-assigned when referred to multiple times.[7] This novel philosophy leads us to some very intuitive behavior, like in the following program, which actually mimics the familiar behaviour of Python.

```
val x = 5
val () =
  case 3
    of x => ()
val 3 = x
```

NovelML's novel dynamics ensure this executes with no (visible) exceptions, unlike in SML (which raises a non-descript "match bind" on this program!)

[5]Type definitions are a big issue students have when learning SML. Not in NoML!

[6]Modules are a big issue students have when learning SML. Not in NoML!

[7]One may wonder whether this renders NovelML not Turing complete. The authors, of course, do not care about Turing completeness, but for the sake of less-enlightened readers, we include a proof of Turing completeness:

Every NovelML program has a finite number of variables. However, each variable can hold an arbitrary (unbounded) natural number. Thus, NovelML programs can be used to simulate register machines with an arbitrary, finite number of registers. This system is known to be Turing complete. The other direction is, of course, a novel exercise for the reader. □

4.1 `op=`

SML uses `=` in two places, val declarations and equality expressions. NovelML unifies these concepts in the most logical way, with the `op=` operator also performing binding/assignment. Since it is still an expression, we adopt C's novel behavior and return the novel value of the variable that was just bound. The following program demonstrates a novel usecase of this behaviour.

```
val x =       5
val true      =
  if x = ~3 < 0
    then   true
  else    false
(* x is now ~3 *)
```

NovelML's novel dynamics ensure this executes with no (visible) exceptions, unlike in SML (we were unable to coax SML/NJ to run this program to completion, and are thus unsure of its runtime behavior.)

4.2 `ref`

This begs the novel question: what is the difference between this and `?.X1 ref`[8]? To answer this we must appeal to a little known section of **The™ Definition™**, where is clarifies that `ref` means "refuse to mutate"[9]. While industry is skeptical of the benefits of variables that refuse to mutate, we believe there may one day be a novel use case, thus include them.

In NoML, attempting to bind to a value that has the runtime type of `ref` will raise the type error `No`. The `!`[10] function simply evaluates to the novel value of the variable that refuses to mutuate (if the type is correct). But, if the variable is an integer, then it is simply the factorial function. The `:=`[11] function returns `0`[12] if the value on the right hand side is the same as the "refuse to mutate", otherwise it raises `No`.

In the following novel program, we demonstrate the power of "no variable left in the shadows" as well as the utility of the de-refuse-to-mutate/factorial function:

```
val ? = fn () => !8
val 40320 = ?() (* 8 factorial! *)

fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n - 1) + fib (n - 2)
```

[8]Oh no! The value restriction!

[9]Unfortunately the page was lost during the printing process.

[10]BANG!!!!!!!!!!!11!1 ...or de-refuse-to-mutate I guess...

[11]omg! a walrus

[12]the success error code

```
val ! = fib
val 21 = ?() (* 8 fibonacci! *)
```

5 Examples

1. The following raises the `No` exception:

   ```
   datatype 'a except = Ok of 'a | Error of string
   ```

2. The following prints `we made it!` before raising the `No` exception:

   ```
   fun polymorphism (x : int except) =
     (* Luckily, x can be of any type. *)
     let val result = x + 5 in result end

   val 5 = polymorphism 0
   val () = print "we made it!"
   val 5 = polymorphism (Ok 0)
   ```

3. The following runs to completion without visible errors:

   ```
   val x = 1
   fun addX (n : int) : int = n + x
   val x = 2

   val 5 = addX 3
   ```

4. The following raises the `No` exception, since a string is not a constructor for `exn`:

   ```
   raise "ur a naughty programmer uwu"
   ```

5. The following example prints 0:

   ```
   val () = print (Int.toString (
     (let val "hi" = 5 in true end) handle No => 0
   ))
   ```

6. The following example prints 0:

   ```
   val () = print (Int.toString (
     () (* Unit can be implicitly cast to 0 *)
   ))
   ```

7. The following example prints 0:

   ```
   val () = print (Int.toString (
     if "true" then 0 else "1 / 0"
   ))
   ```

8. The following example prints 0:

   ```
   val () = print "0"
   ```

9. The following example is a function that computes the sum of a tree:

   ```
   fun treeSum Empty rights acc =
         case rights of
           []      => acc
         | r :: rs => treeSum r rs acc
     | treeSum (Node (l,x,r)) rights acc =
         let
           val rights = r :: rights
           val acc    = x + acc
         in
           treeSum l rights acc
         end
   ```

   Alternatively, we could implement it as follows:

   ```
   fun treeSum tree =
     let
       val acc = 0
       val rights = []
       (* It's good practice to copy variables
        * into obscure names so you don't
        * interfere with the global namespace.
        *)
       val t = tree

       fun treeSum' () =
         case t of
           Empty => (
             case rights of
               [] => acc
             | r :: rs => treeSum' ()
           )
         | Node (l, x, r) =>
             let
               val rights = r :: rights
               val acc = x + acc
               val t = l
             in
               treeSum' ()
             end
     in
       treeSum' ()
     end
   ```

6 Conclusion

In this novel work, we explored the remarkable properties that arise from combining the familiarity of SML's syntax with the expressiveness of Python's runtime behavior. We discovered fun mottos along the way:

- "if it parses, it runs"
- "no variable left in the shadows"
- "`No`"

Guided by these mottos, we shall bring forth a novel society of programmer-citizens, primed with the necessary tools to raise a new world order.

Although this work is entirely novel, there are still scraps of future work we left out to give other authors a shot at being published in SIGBOVIK 2024. We list some suggestions here:

- Use NovelML to write a novel (or to do anything else (you can be our first user!))
- Find ways to reduce the complexity of NovelML's dynamic semantics (likely trivial and not novel and not worth publishing)
- Determine the complexity of the decision problem, "does the NovelML program p raise the `No` exception, assuming p has 0 occurrences of the `No` identifier? " (we suspect it to be NP-hard at least)
- Bake a batch of cookies (see References section)

7 References

1. https://tex.stackexchange.com/questions/571314/make-a-color-box-with-the-trans-pride-flag
2. Fiber Bundles and Intersectional Feminism
3. SML Help which we (the authors) helped write, that's how you know its trustworthy and novel.
4. Typestate-Oriented Programming
5. The Gospel of Wealth
6. Lord Eshteross' Maple Ginger Cookies with Turmeric