Converting the Suggested Upper Merged Ontology to Typed First-order Form

Adam Pease[1]

[1]Infosys
adam.pease@infosys.com

February 20, 2019

1 Introduction

Imagine a voice-enabled household robot that can pick up objects and transfer them from room to room. The owner might say "Pick up my red cart and put it in the garage." but the robot might hear either "Pick up my red car and put it in the garage." or the original sentence. A reasoning system might disambiguate the utterance or correct it by reasoning that a typical car weights 1.5 tons and that's 3000 pounds and the carrying capacity of the robot is 100 pounds and the alternate text must be what was said. A system must construct an answer to a question that has never been asked before ("Can the robot carry a car?") perform some simple computation involving unit conversions, to understand that 3000lbs >100lbs and possibly explain its answer if required ("Why didn't you move my car like I asked you to?")

Theorem proving[1] (or Automated Reasoning) is the process by which we can answer a question posed to a theory in a mathematic logic. It differs from search and information retrieval (such as that done with Google or Apache Lucene) in that small facts and rules are combined to synthesize an answer to a question that may never have been asked before, as opposed to matching a query to the most similar previous query and answer. Theorem proving also provides an explanation (a proof) of how the answer was derived. In this work, we use the Suggested Upper Merged Ontology (SUMO) (Niles and Pease, 2001; Pease, 2011) as the theory, or collection of statements in mathematical logic. There are many kinds of mathematical logic. One of the most popular for automated reasoning, providing a good compromise between the expressiveness and efficiency of the language, is called First Order Logic (FOL) (or predicate calculus).

[1]`https://plato.stanford.edu/entries/reasoning-automated/`

General purpose first-order theorem provers historically haven't done proofs with arithmetic, which is related to problems with Russel's Paradox (Baldwin and Lessmann, 2018). The solution to the paradox, which was discovered by Ernst Zermelo, involved separating numbers and other things into different types. The solution in automated reasoning has now been largely addressed with a new language called Typed First-order Form (TFF) (Sutcliffe et al., 2012) and implemented in several of the best modern provers, including Vampire (Kovács and Voronkov, 2013) from the University of Manchester. Writing a translator from SUMO's native formalization into TFF should open up many new opportunities for doing practical automated reasoning involving numbers and arithmetic.

To have useful and non-trivial reasoning about, for example, a robot's capabilities, or to do question answering, we need not only a language capable of arithmetic calculation (as well as first order logical reasoning) but also a non-trivial body of axioms that has the information about the real world needed to form answers to such questions. For that reason, we need to use the SUMO, which is a comprehensive and diverse set of logical statements about the world. At approximately 20,000 concepts and 80,000 logical statements, it is large enough to answer interesting questions about a wide range of topics.

In earlier work, we described (Pease and Schulz, 2014) how to translate SUMO to the strictly first order language of TPTP (Sutcliffe, 2007). SUMO has an extensive type structure and all relations have type restrictions on their arguments. Translation to TPTP involved implementing a sorted (typed) logic axiomatically in TPTP by altering all implications in SUMO to contain type restrictions on any variables that appear. Many other translation steps were needed that are described in our earlier paper and part of the translation to TFF but since they are not specific to the TFF translation they are not repeated here in detail. Note also that all the strictly higher-order content in SUMO is lost in translation to first-order, whether TPTP or TFF. Briefly however, the translation steps include:

- expanding "row variables" which allow for stating axioms without commitment to the number of arguments a relation has, similar to Lisp's @REST construct
- instantiating "predicate variables" with all possible values. This is needed for any axiom that has a variable in place of a relation.
- expanding the arity of all variable arity relations as set of relations with different names depending upon their fixed number of arguments
- renaming any relations given as arguments to other relations

The interested reader is refered to the earlier paper for more details and examples.

2 Typed First-order Form

Like TPTP, TFF forms are valid Prolog syntax (although obviously not the same semantics!) TFF has five disjoint *sorts*: integers, real numbers, rational numbers, booleans and everything else. These are respectively called $int, $real, $rat, $o and $i in TFF syntax. Each variable that is used in a logical statement must be declared to be one of these sorts, or by default it will be assumed to be type $i.

TFF has built in to the language the basic arithmetic functions and arithmetic comparison operators. Each function and operator is *polymorphic* - it is actually a set of three different operators that can handle integers, rationals and reals. Equality is also defined for $o and $i.

TFF's creators have planned to include the ability to define subtypes (subsorts) but this is not yet defined in specification or implemented in any prover. An additional issue is that since all types are disjoint, and SUMO allows multiple inheritance, there is a mismatch between the two type systems. So we have to continue to implement much of SUMO's sort system axiomatically in TFF as in TPTP, but have a special treatment of integers, rationals and reals that does use the TFF type system, so we can use its arithmetic and comparison operators.

3 Approach

SUMO's mathematical operators take a generic `Quantity` type, which then has `PhysicalQuantity` (a number with units of measure) as well as `RealNumber`(s) and `Integer`(s) as subclasses (among others see figure 1). So we have rules like figure 2

```
Quantity
  PhysicalDimension
  Number
    RealNumber
      RationalNumber
        Integer
          EvenInteger
          OddInteger
          PrimeNumber
          NonnegativeInteger
            PositiveInteger
          NegativeInteger
      IrrationalNumber
      NonnegativeRealNumber
        PositiveRealNumber
          PositiveInteger
        NegativeRealNumber
          NegativeInteger
      BinaryNumber
    ImaginaryNumber
    ComplexNumber
  PhysicalQuantity
    ConstantQuantity
    TimeMeasure
      TimeDuration
      TimePosition
        TimeInterval
```

Figure 1: SUMO's Hierarchy of Quantity Types (somewhat simplified)

```
(=>
  (and
    (instance ?PM ParticulateMatter)
    (part ?Particle ?PM)
    (approximateDiameter ?Particle
      (MeasureFn ?Size Micrometer))
    (greaterThan 10 ?Size)
    (greaterThan ?Size 2.5))
  (instance ?PM CoarseParticulateMatter)))
```

Figure 2: A rule with an arithmetic comparison

where `greaterThan` is comparing integers or reals, as well as rules like figure 3

```
(=>
  (and
    (instance ?AREA DesertClimateZone)
    (subclass ?MO Month)
    (averageTemperatureForPeriod ?AREA ?MO ?TEMP)
    (greaterThan ?TEMP
      (MeasureFn 18 CelsiusDegree)))
  (instance ?AREA SubtropicalDesertClimateZone))
```

Figure 3: A rule comparing numbers with units

where `greaterThan` is comparing quantities (numbers with units) like "18 degrees".

3.1 Type Promotion

Since TFF doesn't have a type hierarchy, we can't have relations with numbers and quantities (numbers with units) both allowed for their arguments. So the approach is to separate relations that use numbers from those that don't.

Take every relation that has a domain specification of `Quantity` or a subclass of `Quantity`. If the domain is a subclass of `Integer`, then promote it to `Integer` and add a constraint function to the antecedent. If it's a `RationalNumber` leave the type as is. If it's any subclass of `NonnegativeRealNumber` that's also not a subclass of `Integer`, promote it to `RealNumber`. All axioms containing SUMO relations that have corresponding TFF relations will require copying

with a native TFF relation if the SUMO relation is a `Quantity` but not a `RationalNumber`, `RealNumber` or `Integer`.

3.2 Type Adjustment Algorithm

Step-by-step the algorithm is

1. Collect the types of all variables
2. Constrain all types further if an equality or inequality (an instance of SUMO's `RelationExtendedToQuantities`) has one argument that is more specific than the stated argument type for the relation, unless it's already a `RealNumber` (in which case even if the other argument is an `Integer`, don't constrain it).
3. "promote" types that are specializations of `Integer` or `RealNumber` as described above
4. if a number is an `Integer` (without a decimal point) and is not a parameter or in a comparison statement with a defined `Integer` type, add ".0" to it so it's a real and record its type as a `RealNumber`
5. Rename the `RelationExtendedToQuantities` with a `__Real` or `__Integer` suffix if its arguments have been constrained to those types (or their subclasses)
6. If there are any variable types for the axiom at this stage that are below `Quantity` but not below `Number` in the SUMO hierarchy, create two version of the axiom - one with all the original names of the predicates and one with every SUMO predicate without a TFF equivalent having a `__Real` suffix and those that do have a TFF equivalent converted to that equivalent
7. Translate to TFF syntax, including translating all relations that have a corresponding native TFF relation or arithmetic operator, and have a `__Real` or `__Integer` suffix

3.3 Examples of Relevant Axioms

Figure 2 would then become figure 4 and figure 3 becomes figure 5

```
! [V_SIZE : $real, V_PARTICLE : $i, V_PM : $i] :
       (s__instance(V_PM, s__ParticulateMatter) &
        s__part(V_PARTICLE,V_PM) &
        s__approximateDiameter(V_PARTICLE,
            s__MeasureFn(V_SIZE,s__Micrometer)) &
        $greater(10.0,V_SIZE) &
        $greater(V_SIZE,2.5) =>
      s__instance(V_PM,s__CoarseParticulateMatter))
```

Figure 4: Axiom in TFF with native comparison operator

```
! [V_AREA : $i, V_MO : $i, V_TEMP : $i ]
        s__instance(V_AREA, DesertClimateZone) &
        s__subclass(V_MO, Month) &
        s__averageTemperatureForPeriod(V_AREA, V_MO, V_TEMP) &
        s__greaterThan(V_TEMP,
            s__MeasureFn(18,s__CelsiusDegree))) =>
    s__instance(V_AREA, s__SubtropicalDesertClimateZone))
```

Figure 5: Axiom in TFF with SUMO comparison operator

We already have conversion axioms like figure 6

```
(=>
  (equal ?AMOUNT
    (MeasureFn ?X Joule))
  (equal ?AMOUNT
    (MeasureFn
      (MultiplicationFn 0.0002778 ?X) Watt)))
```

Figure 6: Unit conversion axiom

So we need a new axiom to compare quantities for each equality and inequality, e.g. figure 7

```
(=>
  (and
    (equal ?Q1 (MeasureFn ?I1 ?U))
    (equal ?Q2 (MeasureFn ?I2 ?U))
    (greaterThan ?I1 ?I2))
  (greaterThan ?Q1 ?Q2))
```

Figure 7: Unit comparison axiom

but we can do this more succinctly with predicate variables as in figure 8

```
(=>
  (and
    (instance ?REL RelationExtendedToQuantities)
    (equal ?Q1 (MeasureFn ?I1 ?U))
    (equal ?Q2 (MeasureFn ?I2 ?U))
    (?REL ?I1 ?I2))
  (?REL ?Q1 ?Q2))
```

Figure 8: Unit comparison axiom for all operators

which will create the preceding axiom plus all other relevant ones (figure 9).

```
(instance AdditionFn RelationExtendedToQuantities)
(instance DivisionFn RelationExtendedToQuantities)
(instance ExponentiationFn RelationExtendedToQuantities)
(instance MaxFn RelationExtendedToQuantities)
(instance MinFn RelationExtendedToQuantities)
(instance MultiplicationFn RelationExtendedToQuantities)
(instance ReciprocalFn RelationExtendedToQuantities)
(instance RemainderFn RelationExtendedToQuantities)
(instance RoundFn RelationExtendedToQuantities)
(instance SubtractionFn RelationExtendedToQuantities)
(instance equal RelationExtendedToQuantities)
(instance greaterThan RelationExtendedToQuantities)
(instance greaterThanOrEqualTo RelationExtendedToQuantities)
(instance lessThan RelationExtendedToQuantities)
(instance lessThanOrEqualTo RelationExtendedToQuantities)
```

Figure 9: All relations that take numbers and quantities

The axiom in figure 8 plus the relation statements from figure 9 will result in the axiom in figure 7 (and many others) being generated by the predicate variable instantiation process in Sigma. Then when SUMO's `greaterThan` appears with strictly integer or real arguments the subsequent TFF conversion step in Sigma will convert it to $greater.

When caling up to processing all of SUMO to TFF, rather than a subset, we need to treat numeric type polymorphism as a "macro" - every statement that uses TFF's built-in math operators.

3.4 Detailed Example of the Algorithm

Let's take the axiom in figure 10 and the argument type specification for the functions it contains as the start of the step-by-step translation

```
(=>
  (equal
    (RemainderFn ?NUMBER1 ?NUMBER2)
    ?NUMBER)
  (equal
    (SignumFn ?NUMBER2)
    (SignumFn ?NUMBER)))

(domain RemainderFn 1 Quantity)
(domain RemainderFn 2 Quantity)
(range RemainderFn Quantity)

(domain SignumFn 1 RealNumber)
(range SignumFn Integer)
```

Figure 10: Axiom containing functions and their type signatures

For Step 1, the current Sigma translation will know to constrain each type to its most specific value in the axiom, which in this case means that `?NUMBER` and `?NUMBER2`, since they are both arguments to `SignumFn`, must be `RealNumber`, since `SignumFn` takes one argument that must be a `RealNumber - (domain SignumFn 1 RealNumber)`. We'll have a list of `[?NUMBER=RealNumber, ?NUMBER1=Quantity, ?NUMBER2=RealNumber]`

For Step 2 we do have the `RelationExtendedToQuantities` `RemainderFn` as well as `equal` where we must constrain its argument to be the same lowest common type, which among `[?NUMBER1=Quantity, ?NUMBER2=RealNumber]` is `RealNumber`. So the list of variables and their types becomes `[?NUMBER=RealNumber, ?NUMBER1=RealNumber, ?NUMBER2=RealNumber]`.
Step 3 doesn't apply since the variable list contains no specializations of `Integer` or `RealNumber`. Step 4 doesn't apply since we have no literal numbers. Since we've modified the arguments of `RemainderFn` we need to rename it in step 5 as `RemainderFn__Real`. Then we have to look at `equal`. Note that we have to keep track of a modified return type for every function that returns some numerical type. In this case, although `RemainderFn` returns a `Quantity`, its arguments have already constrained it to return a `RealNumber`. So we have to check that the arguments to `equal` are the same and constrain them if needed. In this case, `?NUMBER` is already a `RealNumber` and doesn't need to be constrained further.

Step 6 doesn't apply since there are no variables of type `Number`.

Now we have figure 11

```
(=>
  (equal
    (RemainderFn_Real ?NUMBER1 ?NUMBER2)
    ?NUMBER)
  (equal
    (SignumFn_Real ?NUMBER2)
    (SignumFn_Real ?NUMBER)))
```

Figure 11: Expansion of function names

The last step is to translate into TFF syntax and convert relations and functions to their native TFF names, when applicable. This results in figure 12.

```
! [V_NUMBER : $real, V_NUMBER1 : $real,
   V_NUMBER2 : $real] :
       ($equal(s__RemainderFn_Real(V_NUMBER1,V_NUMBER2),
               V_NUMBER) =>
       ($equal(s__SignumFn_Real(V_NUMBER2),
               s__SignumFn_Real(V_NUMBER))))
```

Figure 12: Axiom in TFF with native comparison operator

Let's take an example axiom that maps to some TFF built in functions or relations. In the case of `greaterThan`, we may have SUMO axioms in which it is used with units, and cases where it is used without. The cases without units will map to TFF's built in $greater relation, as in figure 13

```
(=>
  (and
    (instance ?X ?Y)
    (subclass ?Y PureSubstance)
    (boilingPoint ?Y
      (MeasureFn ?BOIL KelvinDegree))
    (meltingPoint ?Y
      (MeasureFn ?MELT KelvinDegree))
    (measure ?X
      (MeasureFn ?TEMP KelvinDegree))
    (greaterThan ?TEMP ?MELT)
    (lessThan ?TEMP ?BOIL))
  (attribute ?X Liquid))
```

Figure 13: Another rule with numeric comparisons

Step 1 results in the following where a '+' signifies a class rather than an instance - `[?X=Quantity, ?Y=PureSubstance+, ?BOIL=RealNumber, ?TEMP=RealNumber, ?MELT=RealNumber]`. For Step 2 we have `greaterThan` and `lessThan` but all are `RealNumber` so no change is needed. Step 3 doesn't apply. For Step 4 there are no numbers so that doesn't apply. Step 5 we have `greaterThan` and `lessThan`, both have `RealNumber` as arguments and so will get renamed to `greaterThan_Real` and `lessThan_Real` respectively, as shown in Figure 14. Step 6 doesn't apply. For Step 7 we rename `greaterThan_Real` to $greater and `lessThan_Real` to $less and translate to TFF syntax, resulting in Figure 15.

```
(=>
  (and
    (instance ?X ?Y)
    (subclass ?Y PureSubstance)
    (boilingPoint ?Y
      (MeasureFn ?BOIL KelvinDegree))
      (meltingPoint ?Y
        (MeasureFn ?MELT KelvinDegree))
      (measure ?X
        (MeasureFn ?TEMP KelvinDegree))
      (greaterThan_Real ?TEMP ?MELT)
    (lessThan_Real ?TEMP ?BOIL))
  (attribute ?X Liquid))
```

Figure 14: Rule with numeric comparisons after renaming with Real

```
! [V_X : $i, V_Y : $i, V_BOIL : $real,
   V_TEMP : $real, V_MELT : $real] :
      (s__instance(V_X,V_Y) &
       s__subclass(V_Y,s__PureSubstance) &
       s__boilingPoint(V_Y,
                       s__MeasureFn(V_BOIL,s__KelvinDegree)) &
       s__meltingPoint(V_Y,
                       s__MeasureFn(V_MELT,s__KelvinDegree)) &
       s__measure(V_X,
                  s__MeasureFn(V_TEMP,s__KelvinDegree)) &
       $greater(V_TEMP,V_MELT) &
       $less(V_TEMP,V_BOIL)) =>
      s__attribute(V_X,s__Liquid)
```

Figure 15: Rule in TFF with numeric comparisons after renaming operators

Another case is figure 16

```
(<=>
  (and
    (instance ?LD LiquidDrop)
    (approximateDiameter ?LD
      (MeasureFn ?Size Micrometer))
    (lessThan 500 ?Size))
  (instance ?LD Droplet))
```

Figure 16: Axiom with literal numbers

For Step 1 we have `[?LD=Droplet,?Size=RealNumber]`. For step 2, since `?Size` is already a `RealNumber` it doesn't need to be changed to be a `RealNumber` like '500.0' given that they are both arguments to `lessThan`. Step 3 doesn't apply. For Step 4 we promote '500' to be '500.0'. In Step 5 we change `lessThan` to `lessThan_Real` to conform to its argument types. Step 6 doesn't apply. For step 7 we make `lessThan_Real` into $less and do the translation to TFF syntax that results in figure 17.

```
! [V_Size : $real, V_LD : $i] :
  ((s__instance(V_LD,s__LiquidDrop) &
    s__approximateDiameter(V_LD,
                           s__MeasureFn(V_Size,s__Micrometer)) &
    $less(500.0,V_Size)) =>
  s__instance(V_LD,s__Droplet)))

! [V_Size : $real, V_LD : $i] :
  (s__instance(V_LD,s__Droplet) =>
   (s__instance(V_LD,s__LiquidDrop) &
    s__approximateDiameter(V_LD,
                           s__MeasureFn(V_Size,s__Micrometer)) &
    $less(500.0,V_Size)) out
```

Figure 17: TFF Axiom with literal numbers

```
(=>
    (instance ?X NegativeInteger)
    (greaterThan 0 ?X))
```

Figure 18: Axiom with NegativeInteger

For Figure 18 the transformation steps are, in order

1. `[?X=NegativeInteger]`
2. Doesn't apply
3. Promote `NegativeInteger` to `Integer`, so we have `[?X=Integer]`
4. Doesn't apply
5. We make `greaterThan` into `greaterThan_Integer`
6. Doesn't apply
7. TFF syntax translation results in figure 19

```
! [V_X : $int] :
    (s__instance(V_X,s__NegativeInteger) =>
     $greater(0,V_X))
```

Figure 19: TFF Axiom with NegativeInteger

But this doesn't work since `instance` takes $i types and not $int which is the type of `V_X`.

4 Modification to the Translation Algorithm

In effect we need to suppress all axioms involving types below `Integer` and `RealNumber`, but only if they are the definitions of those terms! Otherwise, we need to have the definitions become restrictions on integers and reals. So we build a table of definitional constraints for all types below `Integer` and `RealNumber`. This will consist of figure 20 and 21.

```
(=>
    (instance ?NUMBER EvenInteger)
    (equal
        (RemainderFn ?NUMBER 2) 0))

(=>
    (instance ?NUMBER OddInteger)
    (equal
        (RemainderFn ?NUMBER 2) 1))

(=>
    (instance ?PRIME PrimeNumber)
    (forall (?NUMBER)
        (=>
            (equal
                (RemainderFn ?PRIME ?NUMBER) 0)
            (or
                (equal ?NUMBER 1)
                (equal ?NUMBER ?PRIME)))))

(=>
    (instance ?X NonnegativeInteger)
    (greaterThan ?X -1))

(=>
    (instance ?X NegativeInteger)
    (greaterThan 0 ?X))

(=>
    (instance ?X PositiveInteger)
    (greaterThan ?X 0))

(<=>
    (instance ?NUMBER PositiveRealNumber)
    (and
        (greaterThan ?NUMBER 0)
        (instance ?NUMBER RealNumber)))

(=>
    (instance ?NUMBER PositiveRealNumber)
    (equal
        (SignumFn ?NUMBER) 1))
```

Figure 20: Number definitions

```
(<=>
    (instance ?NUMBER NegativeRealNumber)
    (and
        (lessThan ?NUMBER 0)
        (instance ?NUMBER RealNumber)))

(=>
    (instance ?NUMBER NegativeRealNumber)
    (equal
        (SignumFn ?NUMBER) -1))

(<=>
    (instance ?NUMBER NonnegativeRealNumber)
    (and
        (greaterThanOrEqualTo ?NUMBER 0)
        (instance ?NUMBER RealNumber)))
(=>
    (instance ?NUMBER NonnegativeRealNumber)
    (or
        (equal
            (SignumFn ?NUMBER) 1)
        (equal
            (SignumFn ?NUMBER) 0)))
```

Figure 21: Number definitions (continued)

That leaves some types of numbers that have no axioms other than definitions and which are not used as argument types: `IrrationalNumber`, `BinaryNumber`, `ImaginaryNumber` and `ComplexNumber`. We will exclude these from translation, at least for now. Note also that there are two subtypes that inherit from `RealNumber` and `Integer`. They are `PositiveInteger`, which inherits from `PositiveRealNumber` and `NegativeInteger`, which inherits from `NegativeRealNumber`. We will come back to this issue later.

We reformulate some of the axioms to be consistent, removing the bi-implications and the type assertions when they occur on both sides. This gives us a few revised axioms as shown in figure 22.

```
(=>
    (instance ?NUMBER PositiveRealNumber)
    (greaterThan ?NUMBER 0))

(=>
    (instance ?NUMBER NegativeRealNumber)
    (lessThan ?NUMBER 0))

(=>
    (instance ?NUMBER NonnegativeRealNumber)
    (greaterThanOrEqualTo ?NUMBER 0))
```

Figure 22: Revised number definitions

We preprocess all axioms looking for antecedents that specify an instance of a `Number` then cache the consequent in a map indexed by the type. We also extract the variable name that constrains the type in those antecedents. We wind up with the list of antecedents and variables, respectively, shown in figure 23

```
{PositiveInteger =
   (greaterThan ?X 0),
 OddInteger =
   (equal (RemainderFn ?NUMBER 2) 1),
 NegativeRealNumber =
   (equal (SignumFn ?NUMBER) -1),
 EvenInteger =
   (equal (RemainderFn ?NUMBER 2) 0),
 NonnegativeInteger =
   (greaterThan ?X -1),
 NegativeInteger =
   (greaterThan 0 ?X),

 PrimeNumber =
   (forall (?NUMBER)
     (=>
       (equal (RemainderFn ?PRIME ?NUMBER) 0)
       (or
         (equal ?NUMBER 1)
         (equal ?NUMBER ?PRIME)))),

 RationalNumber =
   (exists (?INT1 ?INT2)
     (and
       (instance ?INT1 Integer)
       (instance ?INT2 Integer)
       (equal ?NUMBER (DivisionFn ?INT1 ?INT2)))),

 NonnegativeRealNumber =
   (or
     (equal (SignumFn ?NUMBER) 1)
     (equal (SignumFn ?NUMBER) 0))}

{PositiveInteger =       X,
 OddInteger =            NUMBER,
 NegativeRealNumber =    NUMBER,
 EvenInteger =           NUMBER,
 NonnegativeInteger =    X,
 NegativeInteger =       X,
 PrimeNumber =           PRIME,
 RationalNumber =        NUMBER,
 NonnegativeRealNumber = NUMBER}
```

Figure 23: All number type conditions

For example, we would have `[PositiveInteger=(greaterThan ?NUMBER 0)]`. The value of the index will be added as an antecedent to any axiom that has that type for a variable. For example, see figure 24 where the first axiom becomes the second, and the resulting TFF version is the third.

```
(=>
    (equal ?W
        (WeekFn ?N ?Y))
    (during ?W ?Y))

(=>
  (greaterThan ?N 0)
  (=>
    (equal ?W
        (WeekFn ?N ?Y))
    (during ?W ?Y)))

! [V_N : $int, V_W : $i, V_Y : $i] :
  ($greater(V_N,0) =>
    ((V_W = s__WeekFn(V_N,V_Y)) =>
       s__during(V_W,V_Y)))
```

Figure 24: Adding a number type condition

We have a different substitution process for appearances of number subtypes in rule consequents. If there is an instance statement involving a number subtype in the consequent, then replace it with the defining condition for that sybtype *and all its parents* from the list in figure 23. For example, see figure 25, in which a variable is concluded to be a `PositiveRealNumber` and must therefore be replaced with its defining condition

```
(=>
    (measure ?QUAKE
        (MeasureFn ?VALUE RichterMagnitude))
    (instance ?VALUE PositiveRealNumber))

(=>
    (measure ?QUAKE
        (MeasureFn ?VALUE RichterMagnitude))
    (greaterThan ?VALUE 0))
```

```
! [V_QUAKE : $i, V_VALUE : $real] :
  (s__measure(V_QUAKE,
              s__MeasureFn(V_VALUE,s__RichterMagnitude)) =>
  $greater(V_VALUE,0)
```

Figure 25: Adding a number type in a consequent

5 Type Propagation from Functions

Our translation algorithm is unfortunately still incomplete, as described so far. Type changes can also propagate throughout a formula because of functions. Take Figure 26 that has an axiom with several nested functions, which in turn give the variables the types below it. But the return types of the functions should also constrain the types of the variables.

```
(<=>
  (equal
    (RemainderFn ?NUMBER1 ?NUMBER2) ?NUMBER)
  (equal
    (AdditionFn
      (MultiplicationFn
        (FloorFn
          (DivisionFn ?NUMBER1 ?NUMBER2))
          ?NUMBER2)
        ?NUMBER)
      ?NUMBER1))

{?NUMBER1=[Quantity], ?NUMBER2=[Quantity], ?NUMBER=[Quantity]}
```

Figure 26: Nested Functions

The return types of the functions are shown in Figure 27. While only `FloorFn` has types more restricted than `Quantity`, these will actually propagate through to constrain the entire set of variables.

```
(domain RemainderFn 1 Quantity)
(domain RemainderFn 2 Quantity)
(range RemainderFn Quantity)

(domain AdditionFn 1 Quantity)
(domain AdditionFn 2 Quantity)

(domain MultiplicationFn 1 Quantity)
(domain MultiplicationFn 2 Quantity)

(domain FloorFn 1 RealNumber)
(range FloorFn Integer)

(domain DivisionFn 1 Quantity)
(domain DivisionFn 2 Quantity)
```

Figure 27: Function Argument and Return Types

`FloorFn` makes the return type of `DivisionFn` a `RealNumber`, which then makes `?NUMBER1` and `?NUMBER2` `RealNumber`s but since `?NUMBER2` is an `Integer`

?NUMBER1 must also be an Integer and the return type of DivisionFn becomes Integer, meaning we need to rename as DivisionFn_Integer and then FloorFn must become FloorFn_Integer. Since MultiplicationFn_Integer returns an Integer, that constrains ?NUMBER also to be an Integer. RemainderFn then also becomes RemainderFn_Integer. The result is the variable set and TFF translation shown in Figure 28.

```
{?NUMBER1=[Integer], ?NUMBER2=[Integer], ?NUMBER=[Integer]}

! [V__NUMBER1 : $int,V__NUMBER2 : $int,V__NUMBER : $int] :
  ((s__RemainderFn__Integer(V__NUMBER1, V__NUMBER2) =
   V__NUMBER =>
     $sum(
       $product(
         s__FloorFn__Integer(
           $quotient_e(V__NUMBER1, V__NUMBER2)),
         V__NUMBER2),
       V__NUMBER) =
       V__NUMBER1) &
  ($sum(
    $product(
      s__FloorFn__Integer(
        $quotient_e(V__NUMBER1, V__NUMBER2)) ,
      V__NUMBER2),
    V__NUMBER) =
    V__NUMBER1 =>
      s__RemainderFn__Integer(V__NUMBER1, V__NUMBER2) =
        V__NUMBER))
```

Figure 28: Variable Type Propagation Result

We then also need to copy all the axioms on the original term over to the new term with the modified suffix.

Every relation that has an argument type between Quantity and Integer will need and additional Integer version. Those with arguments between Quantity and RealNumber will need Integer and RationalNumber versions. Those with arguments of Quantity, PhysicalDimension or Number will need all three - Integer, RealNumber and RationalNumber as well as their original version.

5.1 Sort Conflict

Inequalities in TFF must be used with a single sort. However, in SUMO we have the case of figure 29

```
(=>
  (equal
    (CeilingFn ?NUMBER)
    ?INT)
  (not
    (exists (?OTHERINT)
      (and
        (instance ?OTHERINT Integer)
        (greaterThanOrEqualTo ?OTHERINT ?NUMBER)
        (lessThan ?OTHERINT ?INT)))))
```

Figure 29: Type Conflict

`CeilingFn` takes a `RealNumber` and returns the smallest `Integer` greater than or equal to the argument. The `greaterThanOrEqualTo` must compare the `RealNumber` and `Integer` but that is not allowed in TFF.

We would like to be able to state the following in figure 30

```
(! [V__NUMBER : $int,V__INT : $int] :
  (s__CeilingFn__0In1ReFn(V__NUMBER) = V__INT =>
  ~ ? [V__OTHERINT:$int] :
    ($greatereq(V__OTHERINT ,V__NUMBER) &
    $less(V__OTHERINT ,V__INT)))).
```

Figure 30: Type ConflictTFF

but that generates an error due to the arguments to `$greatereq` being of different types. A similar problem exists in SUMO's axiom for `FloorFn`.

6 Inference Tests

TBD, discuss inference tests and sample proofs of numeric queries, starting with the robot example.

7 Software

TBD, Discuss the major classes in the TFF conversion implementation and API.

8 Conclusion

TBD

References

Baldwin, J. T. and Lessmann, O. (2018). What is Russell's paradox? `https://www.scientificamerican.com/article/what-is-russells-paradox/`.

Kovács, L. and Voronkov, A. (2013). First-order theorem proving and vampire. In Proceedings of the 25th International Conference on Computer Aided Verification, volume 8044 of CAV 2013, pages 1–35, New York, NY, USA. Springer-Verlag New York, Inc.

Niles, I. and Pease, A. (2001). Toward a Standard Upper Ontology. In Welty, C. and Smith, B., editors, Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001), pages 2–9.

Pease, A. (2011). Ontology: A Practical Guide. Articulate Software Press, Angwin, CA.

Pease, A. and Schulz, S. (2014). Knowledge Engineering for Large Ontologies with Sigma KEE 3.0. In The International Joint Conference on Automated Reasoning.

Sutcliffe, G. (2007). TPTP, TSTP, CASC, etc. In Proceedings of the Second International Conference on Computer Science: Theory and Applications, CSR'07, pages 6–22, Berlin, Heidelberg. Springer-Verlag.

Sutcliffe, G., Schulz, S., Claessen, K., and Baumgartner, P. (2012). The TPTP Typed First-order Form with Arithmetic. In International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2012), pages 406–419.