Set Theoretic Semantics for Common Sense (Draft – do not distribute)

Translating from SUMO to HOTG

Chad E. Brown, Adam Pease, Josef Urban

May 22, 2023

Abstract

This document is in a draft form, do not take it too seriously. A real abstract may be written later.

Contents

Chapter 1

Introduction

The Suggested Upper Merged Ontology (SUMO) [12,13] is a comprehensive ontology of around 20,000 concepts and 80,000 hand-authored logical statements in a higher-order logic, that has an associated integrated development environment called Sigma [15][1] that interfaces to leading theorem provers such as Eprover [17] and Vampire [10]. In previous work on translating SUMO to THF [1] a syntactic translation to THF was created but did not resolve many aspects of the intended higher order semantics of SUMO. It is our objective in our current efforts to lay the groundwork for a new translation to TH0, based on expressing SUMO in set theory. We believe this will attach to SUMO a stronger set-theoretical interpretation that will allow deciding more queries and provide better intuition for avoiding contradictory formalizations. Once this is done, our plan is to train ENIGMA-style [3–6] query answering and contradiction-finding [18] AITP systems on such SUMO problems and develop autoformalization [7–9, 20, 21] methods targeting common-sense reasoning based on SUMO.

In earlier work, we described [15] how to translate SUMO to the strictly first order language of TPTP-FOF [16] and TF0 [14, 19]. SUMO has an extensive type structure and all relations have type restrictions on their arguments. Translation to TPTP FOF involved implementing a sorted (typed) logic axiomatically in TPTP by altering all implications in SUMO to contain type restrictions on any variables that appear.

[1] `https://www.ontologyportal.org`

Chapter 2

Fragments of SUMO

We give grammars for fragments of the SUMO language in the domains of our translations of SUMO. There are some aspects of SUMO that do not fall into either of these fragments – namely formulas with probabilistic operators.

2.1 SUMO-K: Fragment of SUMO with Kappas

We start by defining SUMO-K terms, spines[1] and formulas. These extend the first-order fragment of SUMO with κ-classes. Formally, we have ordinary variables (x), row variables (ρ) and constants (c). We mutually define the sets of SUMO-K terms t, SUMO-K spines[2] s and SUMO-K formulas ψ as follows:

$$t ::= x|c|(x\ s)|(c\ s)|(\kappa x.\psi)$$

$$s ::= t\ s|\cdot|\rho$$

$$\begin{array}{rcl}
\psi & ::= & \bot|\top|(\neg\psi) \\
& | & (\psi \to \psi)|(\psi \wedge \psi)|(\psi \vee \psi)|(\psi \leftrightarrow \psi) \\
& | & (\forall x.\psi)|(\exists x.\psi)|(t = t) \\
& | & (\texttt{instance}\ t\ t) \\
& | & (\texttt{subclass}\ t\ t) \\
& | & (c\ s)
\end{array}$$

The definition is mutually recursive since the term $\kappa x.\psi$ depends on the formula ψ. Of course, κ, $\forall$ and $\exists$ are binders.

2.2 SUMO-KM: Fragment of SUMO with Kappas and Modalities

We next give an extended grammer for SUMO-KM terms, spines and formulas. These extend the SUMO-K language with modalities.

[1]spine = list of terms
[2]spine = list of terms

Formally, we have ordinary variables (x), row variables (ρ) and constants (c). We mutually define the sets of SUMO-KM terms t, SUMO-KM spines s and SUMO-KM formulas ψ as follows:

$$t ::= x|c|(x\ s)|(c\ s)|(\kappa x.\psi)$$

$$s ::= t\ s|\cdot|\rho$$

$$\begin{array}{rcl}
\psi & ::= & \bot|\top|(\neg\psi) \\
 & | & (\psi \to \psi)|(\psi \wedge \psi)|(\psi \vee \psi)|(\psi \leftrightarrow \psi) \\
 & | & (\forall x.\psi)|(\exists x.\psi)|(t = t) \\
 & | & (\texttt{instance}\ t\ t) \\
 & | & (\texttt{subclass}\ t\ t) \\
 & | & (\texttt{modalAttribute}\ \psi\ \texttt{Obligation}) \\
 & | & (\texttt{modalAttribute}\ \psi\ \texttt{Permission}) \\
 & | & (\texttt{modalAttribute}\ \psi\ \texttt{Necessity}) \\
 & | & (\texttt{modalAttribute}\ \psi\ \texttt{Possibility}) \\
 & | & (\texttt{knows}\ t\ \psi) \\
 & | & (\texttt{believes}\ t\ \psi) \\
 & | & (\texttt{desires}\ t\ \psi) \\
 & | & (\texttt{holdsDuring}\ t\ \psi) \\
 & | & (\texttt{confersObligation}\ \psi\ s) \\
 & | & (\texttt{holdsObligation}\ \psi\ s) \\
 & | & (\texttt{holdsRight}\ \psi\ s) \\
 & | & (\texttt{considers}\ t\ \psi) \\
 & | & (c\ s)
\end{array}$$

The definition is again mutually recursive due to κ binders.

We will generally say SUMO terms and formulas for term and formulas in SUMO-KM (or more strictly in SUMO-K), being more specific when it is necessary.

2.3 Implicit Type Guards

Properly parsing SUMO terms and formulas requires mechanisms for infering implicit type guards for variables (interpreted conjunctively for κ and $\exists$ and via implication for $\forall$). Free variables in SUMO assertions are implicitly universally quantified and are restricted by inferred type guards, as described in [15]. In previous translations targeting first-order logic, relation and function variables are instantiated during the translation (treating the general statement quantifying over relations and functions as a macro to be expanded). Since the current translation will leave these as variables, we must also deal with type guards that are not known until the relation or function is instantiated.

2.4 Can we interpret SUMO formulas as booleans?

Inspired by Kripke semantics for modalities [11] we will translate SUMO formulas to sets of worlds. The reader may wonder if we could simply translate formulas to higher-order propositions (terms of type o). Terms of type o are often called "booleans" since under classical and extensional assumptions, there are semanticlaly precisely two terms of type o. In particular, we have $\forall P : o.P = \top \vee P = \bot$. We consider a simple example to demonstrate the difficulty with interpeting SUMO formulas as booleans.

Consider the following three SUMO assertions:

```
(forall (?P) (=> (modalAttribute ?P Necessity)
                 (modalAttribute ?P Possibility)))
(modalAttribute True Necessity)
(not (modalAttribute False Possibility))
```

Let us write

- $\Box P$ for (`modalAttribute` P `Necessity`)
- and $\Diamond P$ for (`modalAttribute` P `Possibility`).

The three assertions above are $\Box P \rightarrow \Diamond P$, $\Box\top$ and $\neg\Diamond\bot$, all of which are common (though not universal) assumptions for $\Box$ and $\Diamond$. An over-simplified representation in higher-order logic would simply have $\Box : o \rightarrow o$, $\Diamond : o \rightarrow o$ and translate the three assertions above as $\forall P : o.\Box\ P \rightarrow \Diamond\ P$, $\Box\ \top$ and $\neg\ \Diamond\ \bot$.

Suppose we know Gordon is necessarily a plumber and want to know if Gordon is possibly a plumber. As a SUMO test query this appears as follows:

```
(modalAttribute (attribute Gordon Plumber) Necessity)
(query (modalAttribute (attribute Gordon Plumber) Possibility))
```

In the over-simplified representation, we would obtain a new assertion $\Box\ G$ (where $G : o$ abbreviates however (`attribute Gordon Plumber`) is represented) and conjecture $\Diamond\ G$. Indeed with such a representation an automated theorem prover can easily prove the conjecture. This "success" may mislead the reader into believing the automated theorem prover has instantiated P in

$$\forall P : o.\Box\ P \rightarrow \Diamond\ P$$

with G to make the inference. In practice many (possibly all) higher-order automated theorem provers instantiate P with both $\top$ and $\bot$ rule out G being either true or false (yielding a contradiction). While this may still be seen as a "success" it is worth considering what happens if we test the converse.

Assume Gordon is possibly a plumber and ask if Gordon is necessarily a plumber. In SUMO the test query appears as follows:

```
(modalAttribute (attribute Gordon Plumber) Possibility)
(query (modalAttribute (attribute Gordon Plumber) Necessity))
```

Intuitively, this query should fail. The over-simplified representation of the query yields new assumption is $\Diamond G$ and the new conjecture is $\Box G$. If G is true, then we know $\Box G$ by $\Box\top$. If G is false, then we obtain a contradiction from $\Diamond G$ and $\neg\Diamond\bot$. Thus an automated theorem prover can also prove that if Gordon is possibly a plumber, then Gordon is necessarily a plumber.

The takeaway from these examples is rather obvious: interpreting SUMO formulas (once modalities are included) as booleans will simply never work.

Chapter 3

Translation of SUMO-K to Set Theory

We start by giving a translation of SUMO-K, relieving us the need to deal with modalities. Our translation maps terms t to sets. The particular set theory we use is higher-order Tarski-Grothendieck as described in [2].[1]

We will often present the images of translated SUMO items using Megalodon syntax. Megalodon is an interactive theorem prover for higher-order set theory and is the successor to the Egal system also described in [2]. The details of this set theory are not important here. We only note that we have $\in$, $\subseteq$ (which will be used to interpret SUMO's `instance` and `subclass`) and that we have the ability to λ-abstract variables to form terms at higher types. The main types of interest are ι (the base type of sets), o (the type of propositions), $\iota \to \iota$ (the type of functions from sets to sets) and $\iota \to o$ (the type of predicates over sets). When we say SUMO terms t are translated to sets, we mean they are translated to terms of type ι in the higher-order set theory.

Spines s are essentially lists of sets (of varying length). We implement lists as terms of type $\iota \to \iota$ as indicated here:

```
Let nil : set -> set := fun _ => 0.
Let cons : set -> (set -> set) -> set -> set
  := fun a l i => nat_primrec a (fun m _ => l m) i.
```

Informally, a spine like $t_0 \cdots t_{n-1}$ is a function taking i to t_i for each $i \in \{0, \ldots, n-1\}$. Note that nil maps everything to 0 (the empty set) while $\mathsf{cons}\ a\ l$ maps 0 to a and $m+1$ to $l\ m$. We can also perform surgery on a list by replacing element n by a as follows:

```
Let replseq1 : (set -> set) -> set -> set -> set -> set
   := fun l n a i => if i = n then a else l i.
```

[1] Tarski-Grothendieck is a set theory in which there are universes modeling ZFC set theory. These set theoretic universes should not be confused with the universe of discourse $\mathsf{Univ1}$ introduced below.

Here replseq1 l n a is the list that agrees with l except (possibly) the element in position n which is replaced by a. (If n was longer than length of the list l, then intermediate positions will effectively be filled in with 0.) Each SUMO spine s will be translated to a term of type $\iota \to \iota$ so that a spine like $t_0 \cdots t_{n-1}$ is a function taking i to t_i for each $i \in \{0, \ldots, n-1\}$.

Let Univ1 be a set, intended to be a universe of discourse in which most (but not all) targets of interpretation for t will live.

The translation of a SUMO formula ψ can be thought of either as a set (which should be one of the sets 0 or 1) or as a proposition. We also sometimes coerce between type ι and o by considering the sets 0 and 1 to be sets corresponding to false and true. The functions bp : $\iota \to o$ and pb : $o \to \iota$ are used for the coercions.

```
Let bp:set -> prop := fun X => 0 :e X.
Let pb:prop -> set := fun p => if p then 1 else 0.
```

3.1 Preliminary Examples

Before describing the translation in more detail, we give a few simple examples to various aspects of the translation and motivate our choices.

3.1.1 Variable Arity Relations and Functions

Consider the SUMO relation partition, declared as follows:

```
(instance partition Predicate)
(instance partition VariableArityRelation)
(domain partition 1 Class)
(domain partition 2 Class)
```

The last three items indicate that partition has variable arity with at least 2 arguments, both of which are intended to be classes. If there are more than 2 arguments, the remaining arguments are also intended to be classes. In general the extra optional arguments of a variable arity relation or function are intended to have the same domain as the last required argument. We will translate partition to a set that encodes not only when the relation should hold, but also its domain information, its minimum arity and whether or not it is variable arity. To be more precise, we allow for the possibility that the domain information depends on the Kripke world, so that partition also encodes a world-indexed family of domain information. The information above maps to the following. (Note that SUMO indexes the first argument by 1, while in the set theory the first argument is indexed by 0.)

```
Variable s_PARTITION:set.
Hypothesis s_PARTITION__domseq_0: domseq s_PARTITION 0
                                    = s_CLASS.
Hypothesis s_PARTITION__domseq_1: domseq s_PARTITION 1
                                    = s_CLASS.
```

```
Hypothesis s_PARTITION__arity: arity s_PARTITION = 2.
Hypothesis s_PARTITION__vararity: vararity s_PARTITION.
Hypothesis s_PARTITION__domseq_2: domseq s_PARTITION 2
                                    = s_CLASS.
```

The image s_PARTITION is a set encoding a variety of information. In particular, applying domseq to s_PARTITION, a natural number $i \leq 2$ and a world $w \in$ Worlds, we obtain s_CLASS (the interpretation of Class). Applying arity to s_PARTITION gives its minimum arity of 2. Applying vararity to s_PARTITION gives a proposition which holds since partition is variable arity. These three selector functions are abstract, where all we know are their types: domseq : $\iota \to \iota \to \iota$, arity : $\iota \to \iota$ and vararity : $\iota \to o$. In addition there is the most important selector function: eval : $\iota \to (\iota \to \iota) \to \iota$. We use eval to determine the result of applying s_PARTITION to the translation of a spine.

Consider the following SUMO assertion:

```
(partition Entity Physical Abstract)
```

As a formula, the assertion maps to the following proposition

```
(bp (eval s_PARTITION
          (cons s_ENTITY
                (cons s_PHYSICAL
                      (cons s_ABSTRACT nil)))))
```

As an assertion, we universally quantify over worlds to obtain the following closed formula:

```
(bp (eval s_PARTITION
          (cons s_ENTITY
                (cons s_PHYSICAL
                      (cons s_ABSTRACT nil)))))
```

The following SUMO assertion uses partition with five arguments:

```
(partition Organism Animal Plant Fungus Microorganism)
```

The following is the translated set theoretical counterpart:

```
(bp (eval s_PARTITION
          (cons s_ORGANISM
            (cons s_ANIMAL
              (cons s_PLANT
                (cons s_FUNGUS
                  (cons s_MICROORGANISM nil))))))).
```

Now we consider another SUMO assertion about partitions:

```
(=>
  (and
    (exhaustiveDecomposition @ROW)
    (disjointDecomposition @ROW))
  (partition @ROW))
```

This assertion has a free spine variable (called a "row variable" in SUMO). The implicit type guards on the variable depend on the domain information declared for exhaustiveDecomposition, disjointDecomposition and partition. We simply note that the domain information for exhaustiveDecomposition and disjointDecomposition match those for partition given above. The translation of the assertion looks as follows:

```
forall r_ROW:set -> set,
     dom_of (vararity s_EXHAUSTIVEDECOMPOSITION)
            (arity s_EXHAUSTIVEDECOMPOSITION)
            (domseq s_EXHAUSTIVEDECOMPOSITION)
            r_ROW
  -> dom_of (vararity s_DISJOINTDECOMPOSITION)
            (arity s_DISJOINTDECOMPOSITION)
            (domseq s_DISJOINTDECOMPOSITION)
            r_ROW
  -> dom_of (vararity s_PARTITION)
            (arity s_PARTITION)
            (domseq s_PARTITION)
            r_ROW
  -> ((bp (eval s_EXHAUSTIVEDECOMPOSITION r_ROW))
   /\ (bp (eval s_DISJOINTDECOMPOSITION r_ROW))
  -> (bp (eval s_PARTITION r_ROW))).
```

The first three antecedents involving dom_of are the type guards made explicit by the translation. The three are semantically the same since the domain information for exhaustiveDecomposition, disjointDecomposition and partition are equal. We consider the guard for partition in detail. The proposition

```
dom_of (vararity s_PARTITION)
       (arity s_PARTITION)
       (domseq s_PARTITION)
       r_ROW
```

is intended to ensure that `r_ROW` satisfies the appropriate hypotheses for `r_ROW` to be a list of arguments for `s_PARTITION` (at a world). Using the information above, we can rewrite the proposition to be

$$\mathsf{dom\_of}\ \top\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}.$$

In order to further analyze the statement we must consider the definition of dom_of:

```
Definition dom_of:prop -> set -> (set -> set) -> (set -> set) -> prop :=
  fun varar ar dseq u =>
        varar /\ dom_of_varar ar dseq u
     \/ ~varar /\ dom_of_fixedar ar dseq u.
```

The meaning of dom_of is different based on whether or not the first argument is true (i.e., whether or not the relation or function in question has variable arity). In this case the first argument is $\top$ (i.e., s_PARTITION has variable arity), so

$$\mathsf{dom\_of}\ \top\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}$$

is equivalent to

$$\mathsf{dom\_of\_varar}\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}.$$

We next inspect the definition of dom_of_varar:

```
Definition dom_of_varar:set -> (set -> set) -> (set -> set) -> prop :=
 fun ar dseq u => exists n :e omega,
     ar c= n
  /\ (forall i :e ar, u i :e dseq i)
  /\ (forall i :e n, ar c= i -> u i :e dseq ar).
```

For dom_of_varar 2 (domseq s_PARTITION) r_ROW to hold, there must be some $n \geq 2$ (i.e., $2 \subseteq n$ as sets) such that

1. r_ROW $i \in$ domseq s_PARTITION i for all $i \in 2$ and
2. r_ROW $i \in$ domseq s_PARTITION 2 for all $i \in n \setminus 2$.

Hence we can simply say that dom_of_varar 2 (domseq s_PARTITION) r_ROW holds if r_ROW is a list of at least 2 elements where every element is in s_CLASS.

Consider yet another SUMO assertion about partitions:

```
(=> (partition ?SUPER ?SUB1 ?SUB2)
    (partition ?SUPER ?SUB2 ?SUB1))
```

In this case partition is used with exactly 3 arguments, all of which should be SUMO classes.

The translation of this assertion looks as follows:

```
forall v_SUPER, forall v_SUB1, forall v_SUB2,
    v_SUPER :e domseqm s_PARTITION 0
 -> v_SUB1 :e domseqm s_PARTITION 1
 -> v_SUB2 :e domseqm s_PARTITION 2
 -> v_SUB2 :e domseqm s_PARTITION 1
 -> v_SUB1 :e domseqm s_PARTITION 2
 -> ((bp (eval s_PARTITION
               (cons v_SUPER
                     (cons v_SUB1
                           (cons v_SUB2 nil)))))
  -> (bp (eval s_PARTITION
               (cons v_SUPER
                     (cons v_SUB2
                           (cons v_SUB1 nil))))))
```

The first five antecedents are the type guards for the three free variables in the SUMO assertion. The first says that v_SUPER must be an appropriate "zeroth" (first) argument to s_PARTITION. The remaining say that v_SUB1 and v_SUB2 must be appropriate "first" and "second" (second and third) arguments to s_PARTITION.

The attentive reader will note that in these type guards the translation uses domseqm instead of the domseq function we have seen earlier. Let us inspect the definition of domseqm:

```
Definition domseqm:set -> set -> set :=
 fun u i =>
 if vararity u then
   domseq u (if i :e arity u then i else arity u)
 else
   domseq u i.
```

As with dom_of, the definition of domseqm depends on whether or not the relation or function in question has variable arity. In our example all the type guards have the form

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

where $i \in 2$.

Let us focus on the meaning of domseqm s_PARTITION i. Since vararity s_PARTITION holds,

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

is equal to

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ (\mathtt{if}\ i \in \mathsf{arity}\ \mathsf{s\_PARTITION}\ \mathtt{then}\ i\ \mathtt{else}\ \mathsf{arity}\ \mathsf{s\_PARTITION}).$$

Since arity s_PARTITION is 2,

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

is equal to

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ (\mathtt{if}\ i \in 2\ \mathtt{then}\ i\ \mathtt{else}\ 2).$$

Thus if $i \in 2$, then

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

equals

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i$$

which equals s_CLASS. In the remaining case where $i = 2$, we have

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

equals

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ 2$$

which also equals s_CLASS. Thus the type guards are simply saying the three variables are members of s_CLASS w.

One might wonder if there is an easier alternative where we simply lookup the domain information specifically for the relation being used (e.g., Class for partition) and directly use this information in the translation. Here is a SUMO assertion that shows this is not possible:

```
(=>
   (and
      (subrelation ?REL1 ?REL2)
      (instance ?REL1 Predicate)
      (instance ?REL2 Predicate)
      (?REL1 @ROW))
   (?REL2 @ROW))
```

When translating this assertion we do not know the specific relations ?REL1 and ?REL2 since they are variables. Using the techniques above we can still obtain a translated version with type guards for these yet-unspecified relations:

```
forall v_REL1, forall v_REL2, forall r_ROW:set -> set,
      v_REL1 :e domseqm s_SUBRELATION 0
   -> v_REL2 :e domseqm s_SUBRELATION 1
   -> v_REL1 :e s_ENTITY
   -> v_REL2 :e s_ENTITY
   -> dom_of (vararity v_REL1) (arity v_REL1) (domseq v_REL1) r_ROW
   -> dom_of (vararity v_REL2) (arity v_REL2) (domseq v_REL2) r_ROW
   -> ((bp (eval s_SUBRELATION (cons v_REL1 (cons v_REL2 nil))))
    /\ (v_REL1 :e s_PREDICATE)
    /\ (v_REL2 :e s_PREDICATE)
    /\ (bp (eval v_REL1 r_ROW))
    -> (bp (eval v_REL2 r_ROW))).
```

Our focus on partition above may mislead the reader into thinking the variable arity case is the most common one. This is not true. The fact that there are variable arity functions and relations require us to take steps to handle them (e.g., defining dom_of, dom_of_varar and domseqm). However, most relations and functions have fixed arity. We consider the SUMO relation destination as an example which is declared in SUMO as follows:

```
(instance destination CaseRole)
(instance destination PartialValuedRelation)
(domain destination 1 Process)
(domain destination 2 Entity)
(subrelation destination involvedInEvent)
```

In this case since destination is *not* declared to have variable arity, and domain information is given for precisely two arguments, we declare it in the translated version as having fixed arity 2:

```
Variable s_DESTINATION:set.
Hypothesis s_DESTINATION__domseq_0: domseq s_DESTINATION 0
                                  = s_PROCESS.
Hypothesis s_DESTINATION__domseq_1: domseq s_DESTINATION 1
                                  = s_ENTITY.
Hypothesis s_DESTINATION__arity: arity s_DESTINATION = 2.
Hypothesis s_DESTINATION__not_vararity: ~ vararity s_DESTINATION.
```

We also translate the first two instance assertions:

```
Hypothesis p712: s_DESTINATION :e s_CASEROLE.
Hypothesis p713: s_DESTINATION :e s_PARTIALVALUEDRELATION.
```

The examples above motivate how type guards are handled in the presence of variable arity functions and relations. There are two other interesting aspects of SUMO that require careful consideration when designing the translation: κ binders and modalities. We consider examples to motivate our choices for these two constructs.

3.1.2 Kappa Binders

A κ-binder (called KappaFn in SUMO) creates a class by giving a bound variable and a formula indicating the condition. An example of κ in SUMO is given by the following assertion:

```
(=>
  (atomicNumber ?TYPE ?NUMBER)
  (=>
    (and
      (instance ?SUBSTANCE ?TYPE)
      (part ?ATOM ?SUBSTANCE)
      (instance ?ATOM Atom))
    (equal ?NUMBER
      (CardinalityFn
        (KappaFn ?PROTON
          (and
            (part ?PROTON ?ATOM)
            (instance ?PROTON Proton)))))))
```

SUMO's use of κ is similar to Zermelo's separation principle in set theory, though without a bounding set for the variable to range over. In SUMO-K without modalites we translate SUMO terms using a κ to sets using separation and using the fixed set Univ1 as the bounding set as described in [?].

```
forall v_TYPE, forall v_NUMBER, forall v_SUBSTANCE, forall v_ATOM,
    v_TYPE :e domseqm s_ATOMICNUMBER 0
 -> v_NUMBER :e domseqm s_ATOMICNUMBER 1
 -> v_SUBSTANCE :e s_ENTITY
 -> v_TYPE :e s_CLASS
 -> v_ATOM :e domseqm s_PART 0
 -> v_SUBSTANCE :e domseqm s_PART 1
 -> v_ATOM :e s_ENTITY
 -> v_NUMBER :e s_INTEGER
 -> v_ATOM :e domseqm s_PART 1
 -> ((bp (eval s_ATOMICNUMBER (cons v_TYPE (cons v_NUMBER nil))))
  -> ((v_SUBSTANCE :e v_TYPE)
   /\ (bp (eval s_PART (cons v_ATOM (cons v_SUBSTANCE nil))))
   /\ (v_ATOM :e s_ATOM)
```

```
-> (v_NUMBER
  = (eval s_CARDINALITYFN
      (cons {v_PROTON :e Univ1 | v_PROTON :e domseqm s_PART 0
                 /\ v_PROTON :e s_ENTITY
                 /\ (bp (eval s_PART (cons v_PROTON (cons v_ATOM nil))))
                 /\ (v_PROTON :e s_PROTON)} nil))))).
```

3.2 Background for the Translation

Most of the background for the translation has already been presented as needed in the examples above. We describe the complete background here, other than the set theoretic concepts already formalized in Megalodon.

We first declare prefix notation $-$ for the unary minus function and $+$ for the binary addition function on surreal numbers (and so in particular on integers). We will sometimes need these due to our representation of lists as functions from natural numbers to sets. We will freely use 0, 1, 2, etc., for the usual finite ordinals, where n equals $\{0, \ldots, n-1\}$.

```
Prefix - 358 := minus_SNo.
Infix + 360 right := add_SNo.
```

As described above, we define nil and cons for lists (as functions from natural numbers to sets) to represent spines. We also define the function replseq1 for replacing an element of a list.

```
Let nil : set -> set := fun _ => 0.
Let cons : set -> (set -> set) -> set -> set
  := fun a l i => nat_primrec a (fun m _ => l m) i.
Let replseq1 : (set -> set) -> set -> set -> set -> set
  := fun l n a i => if i = n then a else l i.
```

A generic variable Univ1 acts as our universe of discourse.

```
Variable Univ1:set.
```

The set interpreting a SUMO term can be thought of as a tuple consisting of five pieces of information: the value (eval), whether or not it is variable arity (vararity), the minimum arity (arity), the domain information for the arguments (domseq) and the intended range (ran).

```
Variable eval:set -> (set -> set) -> set.
Variable vararity:set -> prop.
Variable arity:set -> set.
Variable domseq:set -> set -> set.
Variable ran:set -> set.
```

Lists are encoded as having type $\iota \to \iota$ as described above, so it is best to think of eval and domseq as functions that take one argument. After applying one argument eval x (of type $(\iota \to \iota) \to \iota$) is a function waiting to take a list of arguments and return a set. Likewise, domseq x returns a list of type $\iota \to \iota$, giving the list of intended domains of the list of arguments.

We define an auxiliary function popseq to pop an integer number of entries from the beginning of a list.

```
Definition popseq:set -> (set -> set) -> set -> set
  := fun n l i => l (n + i).
```

We then define domseqm, dom_of_fixedar, dom_of_varar and dom_of to handle the variable arity and fixed arity cases separately as described above.

```
Definition domseqm:set -> set -> set
 := fun u i =>
     if vararity u then
        domseq u (if i :e arity u then i else arity u)
     else
        domseq u i.

Definition dom_of_fixedar:set -> (set -> set) -> (set -> set) -> prop
 := fun ar dseq u =>
       (forall i :e ar, u i :e dseq i).

Definition dom_of_varar:set -> (set -> set) -> (set -> set) -> prop :=
  fun ar dseq u => exists n :e omega,
       ar c= n
    /\ (forall i :e ar, u i :e dseq i)
    /\ (forall i :e n, ar c= i -> u i :e dseq ar).

Definition dom_of:prop -> set -> (set -> set) -> (set -> set) -> prop :=
  fun varar ar dseq u =>
       varar /\ dom_of_varar ar dseq u
   \/ ~varar /\ dom_of_fixedar ar dseq u.
```

We assume the arity function returns a natural number (a member of the set ω) and that for appropriate arguments domseq will yeild either a subset of Univ1 or a subset of the power set of Univ1 (since these should be where interpretations of SUMO classes live). We make a further similar assumption for the last class given in the variable arity case (which should be the intended domain for all of the optional arguments).

```
Hypothesis arity_omega : forall v, arity v :e omega.
Hypothesis arity_domseq : forall v, forall i :e arity v,
    domseq v i :e Power Univ1 :\/: Power (Power Univ1).
Hypothesis vararity_domseq : forall v, vararity v ->
    domseq v (arity v) :e Power Univ1 :\/: Power (Power Univ1).
```

We finally assume that when evaluated appropriately to a list of arguments in the intended domain we will obtain values in the intended range.

```
Hypothesis dom_ran : forall v, forall u:set -> set,
    dom_of (vararity v) (arity v) (domseq v) u
 -> eval v (fun i => u i) :e ran v.
```

Our last background definitions will be bp ("bool to prop") and pb ("prop to bool").

```
Let bp:set -> prop := fun X => 0 :e X.
Let pb:prop -> set := fun p => if p then 1 else 0.
```

The function bp takes a set X to the proposition $0 \in X$. If X is 0, this is false since $0 \notin 0$. If X is 1, this is false since $0 \in 1$. The function pb takes the proposition false to 0 and the proposition true to 1.

3.3 The Translation

We now describe the translation itself. A first pass through the SUMO files given records the typing information from domain, range, domainsubclass, rangesubclass and subrelation assertions. A finite number of secondary passes determines which names will have avariable arity (either due to a direct assertion or due to being inferred to be in a variable arity class).

The final pass translates the assertions, and this is our focus here. Each SUMO assertion is a SUMO proposition φ which may have free variables in it. Thus if we translate the SUMO proposition φ into the set theoretic proposition φ', then the translated assertion will be

$$\forall x_1 \cdots x_n . G_1 \to \cdots G_m \to \varphi'$$

where $x_1, \ldots, x_n$ are the free variables in φ and $G_1, \ldots, G_m$ are the type guards for these free variables. Note that some of these free variables may be for spine variables (i.e., row variables) and may have type $\iota \to \iota$. Such variables may also have type guards.

SUMO variables x translate to themselves where after translation x is a variable of type ι (ranging over sets). For SUMO constants c we choose a name c' and declare this as having type ι. When a variable or constant is applied to a spine we translate the spines and use eval.

- $(x\ s)$ translates to $(\mathsf{eval}\ x\ s')$ where s' is the result of translating the SUMO spine s.
- $(c\ s)$ translates to $(\mathsf{eval}\ c'\ s')$ where s' is the result of translating the SUMO spine s and c' is the chosen set as a counterpart to the SUMO constant c.

The only remaining case are κ binders.

- We translate $(\kappa x.\psi)$ to

  $$\{x \in \mathsf{Univ1} \mid G_1 \wedge \ldots G_m \wedge \psi'\}$$

  where $G_1, \ldots, G_m$ are generated type guards for x and ψ' is the result of translating the SUMO proposition ψ to a set theoretic proposition. Note that x ranges over Univ1.

The translations of spines is relatively straightforward.

- The SUMO spine $(t\ s)$ is translated to $(\mathsf{cons}\ t'\ s')$ where t' is the translation of t and s' is the translation of s.
- A spine variable ρ is translated to itself.[2]
- The empty spine is translated to nil.

We consider each case of a SUMO proposition (within our fragment). The usual logical operators are translated as the corresponding operators:

- $\bot$ and $\top$ translate simply to $\bot$ and $\top$.
- $(\neg\ \psi)$ translates to $\neg\psi'$ where ψ is a SUMO proposition which translates to the set theoretic proposition ψ'.
- $(\psi\ \ \rightarrow\ \ \xi)$ translates to $\psi' \rightarrow \xi'$ where ψ and ξ are SUMO propositions translate to the set theoretic propositions ψ' and ξ'.
- $(\psi\ \ \leftrightarrow\ \ \xi)$ translates to $\psi' \leftrightarrow \xi'$ where ψ and ξ are SUMO propositions translate to the set theoretic propositions ψ' and ξ'.
- Theoretically, $\psi \wedge \xi$ translates to $\psi' \wedge \xi'$. Practically speaking in SUMO conjunction is n-ary so it is more accurate to state that $(\mathsf{and}\ \psi_1\ \cdots \psi_n)$ translates to $\psi_1' \wedge \cdots \wedge \psi_n'$ where $\psi_1, \ldots, \psi_n$ are SUMO propositions translate to the set theoretic propositions $\psi_1', \ldots, \psi_n'$.
- Again, theoretically $\psi \vee \xi$ translates to $\psi' \vee \xi'$. Practically, $(\mathsf{or}\ \psi_1\ \cdots \psi_n)$ translates to $\psi_1' \vee \cdots \vee \psi_n'$ where $\psi_1, \ldots, \psi_n$ are SUMO propositions translate to the set theoretic propositions $\psi_1', \ldots, \psi_n'$.
- Theoretically, $\forall x.\psi$ translates to $\forall x.G_1 \rightarrow \cdots \rightarrow G_m \rightarrow \psi'$ where ψ' is the result of translating ψ and $G_1, \ldots, G_m$ are the generated type guards for x. Practically speaking, SUMO allows several variables to be universally quantified at once, so it is more accurate to say $(\mathsf{forall}\ (x_1 \ldots x_n)\ \psi)$ translates to $\forall x_1 \ldots x_n.G_1 \rightarrow \cdots \rightarrow G_m \rightarrow \psi'$ where $x_1, \ldots, x_n$ are variables, $G_1, \ldots, G_m$ are the generated type guards for these variables and ψ' is the set theoretic proposition obtained by translating the SUMO proposition ψ.
- Again, theoretically $\exists x.\psi$ translates to $\exists x.G_1 \wedge \cdots \wedge G_m \wedge \psi'$ and practically $(\mathsf{exists}\ (x_1 \ldots x_n)\ \psi)$ translates to $\exists x_1 \ldots x_n.G_1 \wedge \cdots \wedge G_m \wedge \psi'$ where $x_1, \ldots, x_n$ are variables, $G_1, \ldots, G_m$ are the generated type guards for these variables and ψ' is the set theoretic proposition obtained by translating the SUMO proposition ψ.

[2] We describe the translation this way for simplicity of presentation. In reality a spine variable is sometimes followed by one argument which we think of as appending one more elemnt to the list. The function replseq1 is used in such a case.

- $(t_1 = t_2)$ translates to $t_1' = t_2'$ where t_1 and t_2 are SUMO terms which translate to sets t_1' and t_2'.

We use set membership and inclusion to interpret `instance` and `subclass`.

- (`instance` t_1 t_2) translates to $t_1' \in t_2'$ where t_1 and t_2 are SUMO terms which translate to sets t_1' and t_2'.
- (`subclass` t_1 t_2) translates to $t_1' \subseteq t_2'$ where t_1 and t_2 are SUMO terms which translate to sets t_1' and t_2'.

Chapter 4

Translation of SUMO-KM to Set Theory

Our translation maps terms t to sets. The particular set theory we use is higher-order Tarski-Grothendieck as described in [2].

We will often present the images of translated SUMO-KM items using Megalodon syntax. Megalodon is an interactive theorem prover for higher-order set theory and is the successor to the Egal system also described in [2]. The details of this set theory are not important here. We only note that we have $\in$, $\subseteq$ (which will be used to interpret SUMO-KM's `instance` and `subclass`) and that we have the ability to λ-abstract variables to form terms at higher types. The main types of interest are ι (the base type of sets), o (the type of propositions), $\iota \to \iota$ (the type of functions from sets to sets) and $\iota \to o$ (the type of predicates over sets). When we say SUMO-KM terms t are translated to sets, we mean they are translated to terms of type ι in the higher-order set theory.

Spines s are essentially lists of sets (of varying length). We implement lists as terms of type $\iota \to \iota$ as indicated here:

```
Let nil : set -> set := fun _ => 0.
Let cons : set -> (set -> set) -> set -> set
  := fun a l i => nat_primrec a (fun m _ => l m) i.
```

Informally, a spine like $t_0 \cdots t_{n-1}$ is a function taking i to t_i for each $i \in \{0, \ldots, n-1\}$. Note that nil maps everything to 0 (the empty set) while $\mathsf{cons}\ a\ l$ maps 0 to a and $m+1$ to $l\ m$. We can also perform surgery on a list by replacing element n by a as follows:

```
Let replseq1 : (set -> set) -> set -> set -> set -> set
   := fun l n a i => if i = n then a else l i.
```

Here $\mathsf{replseq1}\ l\ n\ a$ is the list that agrees with l except (possibly) the element in position n which is replaced by a. (If n was longer than length of the list l, then intermediate positions will effectively be filled in with 0.) Each SUMO-KM spine s will be translated to a term of type $\iota \to \iota$ so that a spine like $t_0 \cdots t_{n-1}$ is a function taking i to t_i for each $i \in \{0, \ldots, n-1\}$.

We use Kripke semantics to translate modalities [11]. Since there are different kinds of modalities, we will have a family of accessiblities relations over worlds. Let $\mathsf{Univ1}$ be a set, intended to be a universe of discourse in which most (but not all) targets of interpretation for t will live. Let World map sets to sets. We define Worlds to be the set $\Pi_{x \in \mathsf{Univ1}} \mathsf{World}\ x$. The idea is simple: for each $x \in \mathsf{Univ1}$ there is a set $\mathsf{World}\ x$ of Kripke-style worlds. Worlds is the cartesian product of this family of sets of worlds. Likewise for each x we will have a binary relation $\mathsf{AccRelnSeq}\ x$ on $\mathsf{World}\ x$. We can lift these to be give binary relations on the cartesian product Worlds by defining $\mathsf{AccReln}\ x\ u\ v$ to hold if $u, v \in \mathsf{Worlds}$ (i.e., u and v are functions mapping $x \in \mathsf{Univ1}$ into $\mathsf{World}\ x$), $\mathsf{AccRelnSeq}\ x\ (u\ x)\ (v\ x)$ and u and v agree except possibly on x.

The translation of a SUMO-KM formula ψ can be thought of as a collection of worlds. Likewise the translation of a SUMO-KM term t can be thought of as a function mapping worlds to values. Analogous remarks hold for spines. Since we will never need to directly consider more than one world at a time we will always consider the "current" world to be refered to by a common variable w of type ι. Thus the translation of a SUMO-KM term t is a set (term of type ι) which may have the free variable w, the translation of a SUMO-KM spine s has type $\iota \to \iota$ and may have the free variable w, and finally the translation of a SUMO-KM formula ψ is a proposition (term of type o) which may have the free variable w. We can always λ-abstract the common w and consider the images to be types $\iota \to \iota$ (for terms), $\iota \to \iota \to \iota$ (for spines) and $\iota \to o$ (for formulas). We also sometimes coerce between type ι and o by considering the sets 0 and 1 to be sets corresponding to false and true. The functions $\mathsf{bp} : \iota \to o$ and $\mathsf{pb} : o \to \iota$ are used for the coercions.

```
Let bp:set -> prop := fun X => 0 :e X.
Let pb:prop -> set := fun p => if p then 1 else 0.
```

4.1 Preliminary Examples

Before describing the translation in more detail, we give a few simple examples to various aspects of the translation and motivate our choices.

4.1.1 Variable Arity Relations and Functions

Consider the SUMO-KM relation $\mathsf{partition}$, declared as follows:

```
(instance partition Predicate)
(instance partition VariableArityRelation)
(domain partition 1 Class)
(domain partition 2 Class)
```

The last three items indicate that $\mathsf{partition}$ has variable arity with at least 2 arguments, both of which are intended to be classes. If there are more than 2 arguments, the remaining arguments are also intended to be classes. In general the extra optional arguments of a variable arity relation or function are intended to have the same domain as the last required argument. We will translate partition to a set that encodes not only when the relation should hold, but also its domain information, its minimum arity and whether or not it is variable arity. To be more precise, we allow for the possibility that the domain information depends on the Kripke world, so that partition also encodes a world-indexed family of domain information. The information above maps to the following. (Note that SUMO-KM indexes the first argument by 1, while in the set theory the first argument is indexed by 0.)

```
Variable s_PARTITION:set.
Hypothesis s_PARTITION__domseq_0: domseq s_PARTITION 0
                                   = (fun w :e Worlds => (s_CLASS w)).
Hypothesis s_PARTITION__domseq_1: domseq s_PARTITION 1
                                   = (fun w :e Worlds => (s_CLASS w)).
Hypothesis s_PARTITION__arity: arity s_PARTITION = 2.
Hypothesis s_PARTITION__vararity: vararity s_PARTITION.
Hypothesis s_PARTITION__domseq_2: domseq s_PARTITION 2
                                   = (fun w :e Worlds => (s_CLASS w)).
```

The image s_PARTITION is a set encoding a variety of information. In particular, applying domseq to s_PARTITION, a natural number $i \leq 2$ and a world $w \in \mathsf{Worlds}$, we obtain s_CLASS w (the interpretation of Class at the world w). Applying arity to s_PARTITION gives its minimum arity of 2. Applying vararity to s_PARTITION gives a proposition which holds since partition is variable arity. These three selector functions are abstract, where all we know are their types: domseq : $\iota \to \iota \to \iota$, arity : $\iota \to \iota$ and vararity : $\iota \to o$. In addition there is the most important selector function: eval : $\iota \to (\iota \to \iota) \to \iota$. We use eval to determine the result of applying s_PARTITION to the translation of a spine.

Consider the following SUMO-KM assertion:

```
(partition Entity Physical Abstract)
```

As a formula, the assertion maps to the following proposition

```
(bp (eval (s_PARTITION w)
          (cons (s_ENTITY w)
                (cons (s_PHYSICAL w)
                      (cons (s_ABSTRACT w) nil)))))
```

As an assertion, we universally quantify over worlds to obtain the following closed formula:

```
forall w :e Worlds,
  (bp (eval (s_PARTITION w)
            (cons (s_ENTITY w)
                  (cons (s_PHYSICAL w)
                        (cons (s_ABSTRACT w) nil)))))
```

The following SUMO-KM assertion uses partition with five arguments:

```
(partition Organism Animal Plant Fungus Microorganism)
```

The following is the translated set theoretical counterpart:

```
forall w :e Worlds,
  (bp (eval (s_PARTITION w)
            (cons (s_ORGANISM w)
              (cons (s_ANIMAL w)
                (cons (s_PLANT w)
                  (cons (s_FUNGUS w)
                    (cons (s_MICROORGANISM w) nil))))))).
```

Now we consider another SUMO-KM assertion about partitions:

```
(=>
  (and
    (exhaustiveDecomposition @ROW)
    (disjointDecomposition @ROW))
  (partition @ROW))
```

This assertion has a free spine variable (called a "row variable" in SUMO-KM). The implicit type guards on the variable depend on the domain information declared for exhaustiveDecomposition, disjointDecomposition and partition. We simply note that the domain information for exhaustiveDecomposition and disjointDecomposition match those for partition given above. The translation of the assertion looks as follows:

```
forall w :e Worlds,
 forall r_ROW:set -> set,
     dom_of (vararity s_EXHAUSTIVEDECOMPOSITION)
            (arity s_EXHAUSTIVEDECOMPOSITION)
            (domseq s_EXHAUSTIVEDECOMPOSITION)
            r_ROW
  -> dom_of (vararity s_DISJOINTDECOMPOSITION)
            (arity s_DISJOINTDECOMPOSITION)
            (domseq s_DISJOINTDECOMPOSITION)
            r_ROW
  -> dom_of (vararity s_PARTITION)
            (arity s_PARTITION)
            (domseq s_PARTITION)
            r_ROW
  -> ((bp (eval (s_EXHAUSTIVEDECOMPOSITION w) r_ROW))
   /\ (bp (eval (s_DISJOINTDECOMPOSITION w) r_ROW))
  -> (bp (eval (s_PARTITION w) r_ROW))).
```

The first three antecedents involving dom_of are the type guards made explicit by the translation. The three are semantically the same since the domain information for exhaustiveDecomposition, disjointDecomposition and partition are equal. We consider the guard for partition in detail. The proposition

```
dom_of (vararity s_PARTITION)
       (arity s_PARTITION)
       (domseq s_PARTITION)
       r_ROW
```

is intended to ensure that `r_ROW` satisfies the appropriate hypotheses for `r_ROW` to be a list of arguments for `s_PARTITION` (at a world). Using the information above, we can rewrite the proposition to be

$$\mathsf{dom\_of}\ \top\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}.$$

In order to further analyze the statement we must consider the definition of $\mathsf{dom\_of}$:

```
Definition dom_of:prop -> set -> (set -> set) -> (set -> set) -> prop :=
  fun varar ar dseq u =>
         varar /\ dom_of_varar ar dseq u
     \/ ~varar /\ dom_of_fixedar ar dseq u.
```

The meaning of $\mathsf{dom\_of}$ is different based on whether or not the first argument is true (i.e., whether or not the relation or function in question has variable arity). In this case the first argument is $\top$ (i.e., $\mathsf{s\_PARTITION}$ has variable arity), so

$$\mathsf{dom\_of}\ \top\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}$$

is equivalent to

$$\mathsf{dom\_of\_varar}\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}.$$

We next inspect the definition of $\mathsf{dom\_of\_varar}$:

```
Definition dom_of_varar:set -> (set -> set) -> (set -> set) -> prop :=
 fun ar dseq u => exists n :e omega,
     ar c= n
  /\ (forall i :e ar, forall w :e Worlds, u i w :e dseq i w)
  /\ (forall i :e n, forall w :e Worlds, ar c= i -> u i w :e dseq ar w).
```

For $\mathsf{dom\_of\_varar}\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}$ to hold, there must be some $n \geq 2$ (i.e., $2 \subseteq n$ as sets) such that

1. $\mathsf{r\_ROW}\ i\ w \in \mathsf{domseq}\ \mathsf{s\_PARTITION}\ i\ w$ for all $i \in 2$ and $w \in \mathsf{Worlds}$ and
2. $\mathsf{r\_ROW}\ i\ w \in \mathsf{domseq}\ \mathsf{s\_PARTITION}\ 2\ w$ for all $i \in n \setminus 2$ and $w \in \mathsf{Worlds}$.

The reader may suspect a typing problem, since domseq has type $\iota \to \iota \to \iota$ but is applied to 3 arguments in $\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i\ w$. However, the proper reading of

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i\ w$$

is as

$$\mathsf{ap}(\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i)\ w$$

where ap is the set theoretic application operator (and has type $\iota \to \iota \to \iota$). Note that $\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i$ equals $\lambda w \in \mathsf{Worlds}.\mathsf{s\_CLASS}\ w$ for each $i \in \{0, 1, 2\}$ (by the hypotheses created when $\mathsf{s\_PARTITION}$ was declared). Here, the lambda notation $\lambda w \in \mathsf{Worlds}.\mathsf{s\_CLASS}\ w$ is indicating a function represented as a set and can more properly be written as $\mathsf{lam}\ \mathsf{Worlds}\ (\lambda w.\mathsf{s\_CLASS}\ w)$ where lam has type $\iota \to (\iota \to \iota) \to \iota$. There is a theorem called beta in the set theory that guarantees $\mathsf{ap}\ (\mathsf{lam}\ X\ f)\ x = f\ x$ if $x \in X$. We can thus compute

$$\begin{aligned} &\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i\ w \\ =\ &\mathsf{ap}(\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i)\ w \\ =\ &\mathsf{ap}(\mathsf{lam}\ \mathsf{Worlds}\ (\lambda w.\mathsf{s\_CLASS}\ w))\ w \\ =\ &\mathsf{s\_CLASS}\ w \end{aligned}$$

for every $w \in \mathsf{Worlds}$ and $i \in \{0, 1, 2\}$. Hence we can simply say that $\mathsf{dom\_of\_varar}\ 2\ (\mathsf{domseq}\ \mathsf{s\_PARTITION})\ \mathsf{r\_ROW}$ holds if $\mathsf{r\_ROW}$ is a list of at least 2 elements where every element when applied to a world w is in $\mathsf{s\_CLASS}\ w$.

Consider yet another SUMO-KM assertion about partitions:

```
(=> (partition ?SUPER ?SUB1 ?SUB2)
    (partition ?SUPER ?SUB2 ?SUB1))
```

In this case partition is used with exactly 3 arguments, all of which should be SUMO-KM classes.

The translation of this assertion looks as follows:

```
forall w :e Worlds,
  forall v_SUPER, forall v_SUB1, forall v_SUB2,
    v_SUPER w :e domseqm s_PARTITION 0 w
 -> v_SUB1 w :e domseqm s_PARTITION 1 w
 -> v_SUB2 w :e domseqm s_PARTITION 2 w
 -> v_SUB2 w :e domseqm s_PARTITION 1 w
 -> v_SUB1 w :e domseqm s_PARTITION 2 w
 -> ((bp (eval (s_PARTITION w)
               (cons (v_SUPER w)
                     (cons (v_SUB1 w)
                           (cons (v_SUB2 w) nil)))))
  -> (bp (eval (s_PARTITION w)
               (cons (v_SUPER w)
                     (cons (v_SUB2 w)
                           (cons (v_SUB1 w) nil))))))
```

The first five antecedents are the type guards for the three free variables in the SUMO-KM assertion. The first says that v_SUPER must be an appropriate "zeroth" (first) argument to s_PARTITION. The remaining say that v_SUB1 and v_SUB2 must be appropriate "first" and "second" (second and third) arguments to s_PARTITION.

The attentive reader will note that in these type guards the translation uses domseqm instead of the domseq function we have seen earlier. Let us inspect the definition of domseqm:

```
Definition domseqm:set -> set -> set :=
 fun u i =>
 if vararity u then
   domseq u (if i :e arity u then i else arity u)
 else
   domseq u i.
```

As with $\mathsf{dom\_of}$, the definition of $\mathsf{domseqm}$ depends on whether or not the relation or function in question has variable arity. In our example all the type guards have the form

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i\ w,$$

i.e.,

$$\mathsf{ap}\ (\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i)\ w,$$

where $i \in 2$.

Let us focus on the meaning of $\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$ before applying it to the world w. Since $\mathsf{vararity}\ \mathsf{s\_PARTITION}$ holds,

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

is equal to

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ (\mathtt{if}\ i \in \mathsf{arity}\ \mathsf{s\_PARTITION}\ \mathtt{then}\ i\ \mathtt{else}\ \mathsf{arity}\ \mathsf{s\_PARTITION}).$$

Since $\mathsf{arity}\ \mathsf{s\_PARTITION}$ is 2,

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

is equal to

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ (\mathtt{if}\ i \in 2\ \mathtt{then}\ i\ \mathtt{else}\ 2).$$

Thus if $i \in 2$, then

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

equals

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ i$$

which equals $(\lambda w \in \mathsf{Worlds}.\mathsf{s\_CLASS}\ w)$. In the remaining case where $i = 2$, we have

$$\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i$$

equals

$$\mathsf{domseq}\ \mathsf{s\_PARTITION}\ 2$$

which also equals $(\lambda w \in \mathsf{Worlds}.\mathsf{s\_CLASS}\ w)$. For $i \leq 2$ and $w \in \mathsf{Worlds}$ we now know

$$\begin{aligned} &\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i\ w \\ =\ &\mathsf{ap}(\mathsf{domseqm}\ \mathsf{s\_PARTITION}\ i)\ w \\ =\ &\mathsf{ap}(\lambda w \in \mathsf{Worlds}.\mathsf{s\_CLASS}\ w)\ w \\ =\ &\mathsf{s\_CLASS}\ w. \end{aligned}$$

Thus the type guards are simply saying the three variables, when applied to the world variable w, are members of s_CLASS w.

One might wonder if there is an easier alternative where we simply lookup the domain information specifically for the relation being used (e.g., Class for partition) and directly use this information in the translation. Here is a SUMO-KM assertion that shows this is not possible:

```
(=>
   (and
      (subrelation ?REL1 ?REL2)
      (instance ?REL1 Predicate)
      (instance ?REL2 Predicate)
      (?REL1 @ROW))
   (?REL2 @ROW))
```

When translating this assertion we do not know the specific relations ?REL1 and ?REL2 since they are variables. Using the techniques above we can still obtain a translated version with type guards for these yet-unspecified relations:

```
forall w :e Worlds,
 forall v_REL1, forall v_REL2, forall r_ROW:set -> set,
      v_REL1 w :e domseqm s_SUBRELATION 0 w
   -> v_REL2 w :e domseqm s_SUBRELATION 1 w
   -> v_REL1 w :e (s_ENTITY w)
   -> v_REL2 w :e (s_ENTITY w)
   -> dom_of (vararity v_REL1) (arity v_REL1) (domseq v_REL1) r_ROW
   -> dom_of (vararity v_REL2) (arity v_REL2) (domseq v_REL2) r_ROW
   -> ((bp (eval (s_SUBRELATION w) (cons (v_REL1 w) (cons (v_REL2 w) nil))))
    /\ ((v_REL1 w) :e (s_PREDICATE w))
    /\ ((v_REL2 w) :e (s_PREDICATE w))
    /\ (bp (eval (v_REL1 w) r_ROW))
    -> (bp (eval (v_REL2 w) r_ROW))).
```

Our focus on partition above may mislead the reader into thinking the variable arity case is the most common one. This is not true. The fact that there are variable arity functions and relations require us to take steps to handle them (e.g., defining dom_of, dom_of_varar and domseqm). However, most relations and functions have fixed arity. We consider the SUMO-KM relation destination as an example which is declared in SUMO-KM as follows:

```
(instance destination CaseRole)
(instance destination PartialValuedRelation)
(domain destination 1 Process)
(domain destination 2 Entity)
(subrelation destination involvedInEvent)
```

In this case since destination is *not* declared to have variable arity, and domain information is given for precisely two arguments, we declare it in the translated version as having fixed arity 2:

```
Variable s_DESTINATION:set.
Hypothesis s_DESTINATION__domseq_0: domseq s_DESTINATION 0
                                  = (fun w :e Worlds => (s_PROCESS w)).
Hypothesis s_DESTINATION__domseq_1: domseq s_DESTINATION 1
                                  = (fun w :e Worlds => (s_ENTITY w)).
Hypothesis s_DESTINATION__arity: arity s_DESTINATION = 2.
Hypothesis s_DESTINATION__not_vararity: ~ vararity s_DESTINATION.
```

We also translate the first two instance assertions:

```
Hypothesis p712: forall w :e Worlds,
                     ((s_DESTINATION w) :e (s_CASEROLE w)).
Hypothesis p713: forall w :e Worlds,
                     ((s_DESTINATION w) :e (s_PARTIALVALUEDRELATION w)).
```

The examples above motivate how type guards are handled in the presence of variable arity functions and relations. There are two other interesting aspects of SUMO-KM that require careful consideration when designing the translation: κ binders and modalities. We consider examples to motivate our choices for these two constructs.

4.1.2 Kappa Binders with Modalities

A κ-binder (called KappaFn in SUMO-KM) creates a class by giving a bound variable and a formula indicating the condition. An example of κ in SUMO-KM is given by the following assertion:

```
(=>
  (atomicNumber ?TYPE ?NUMBER)
  (=>
    (and
      (instance ?SUBSTANCE ?TYPE)
      (part ?ATOM ?SUBSTANCE)
      (instance ?ATOM Atom))
    (equal ?NUMBER
      (CardinalityFn
        (KappaFn ?PROTON
          (and
            (part ?PROTON ?ATOM)
            (instance ?PROTON Proton)))))))
```

SUMO-KM's use of κ is similar to Zermelo's separation principle in set theory, though without a bounding set for the variable to range over. Before modalities were handled, we translated SUMO-KM terms using a κ to sets using separation and using the fixed set Univ1 as the bounding set [?]. With the addition of a world argument w, we handle κ terms via a combination of Fraenkel's replacement and Zermelo's separation principles. We let the bound variable V range over the set $\mathsf{Univ1}^{\mathsf{Worlds}}$ of functions from Worlds to Univ1 and restrict to those that satisfy the translated condition. We then collect all the $V\ w$ sets where w

is the current world of the translation. The translation of the example above looks as follows:

```
forall w :e Worlds,
 forall v_TYPE, forall v_NUMBER, forall v_SUBSTANCE, forall v_ATOM,
     v_TYPE w :e domseqm s_ATOMICNUMBER 0 w
  -> v_NUMBER w :e domseqm s_ATOMICNUMBER 1 w
  -> v_SUBSTANCE w :e (s_ENTITY w)
  -> v_TYPE w :e (s_CLASS w)
  -> v_ATOM w :e domseqm s_PART 0 w
  -> v_SUBSTANCE w :e domseqm s_PART 1 w
  -> v_ATOM w :e (s_ENTITY w)
  -> v_NUMBER w :e (s_INTEGER w)
  -> v_ATOM w :e domseqm s_PART 1 w
  -> ((bp (eval (s_ATOMICNUMBER w) (cons (v_TYPE w) (cons (v_NUMBER w) nil))))
  -> (((v_SUBSTANCE w) :e (v_TYPE w))
   /\ (bp (eval (s_PART w) (cons (v_ATOM w) (cons (v_SUBSTANCE w) nil))))
   /\ ((v_ATOM w) :e (s_ATOM w))
   -> ((v_NUMBER w)
     = (eval (s_CARDINALITYFN w)
        (cons {v_PROTON w | v_PROTON :e Univ1 :^: Worlds,
                            v_PROTON w :e domseqm s_PART 0 w
                         /\ v_PROTON w :e (s_ENTITY w)
                         /\ (bp (eval (s_PART w)
                                  (cons (v_PROTON w)
                                   (cons (v_ATOM w) nil))))
                         /\ ((v_PROTON w) :e (s_PROTON w))}
             nil))))).
```

4.1.3 Modalities

SUMO-KM includes several modalities. In general we classify a SUMO-KM operator as a modality if it is not one of the basic logical operators (e.g., not) and has a formula as an argument. There is also an explicit operator called modalAttribute whose first argument is a formula and second argument is a specific modality. In this case we refer to the second argument as the modal operator. Some of these modalities fit into known modal paradigms, e.g., Necessity and Possibility can be seen as corresponding box and diamond operators. Likewise, the deontic modalities Obligation and Permission can be seen as corresponding box and diamond operators. The modal operators knows, believes and desires can be viewed as different box operators. To be more precise, each of knows, believes and desires takes a cognitive agent as its first argument and formula as its second argument. For each cognitive agent A we view knows A believes A and desires A as separate box operators, in effect giving an infinite family of such operators. Other modal operators do not naturally fit into the box and diamond paradigm: holdsDuring, confers, confersObligation, holdsObligation, holdsRight and considers. We still translate these, but in an abstract way that generalizes box and diamond operators.

One SUMO-KM assertion using modalities states that if a formula is necessary then it is possible:

```
(=>
   (modalAttribute ?FORMULA Necessity)
   (modalAttribute ?FORMULA Possibility))
```

In modal terms, this states that a certain kind of box implies a certain kind of diamond. The translated formula looks as follows:

```
forall w :e Worlds,
 forall v_FORMULA,
   ((ModalBox 1 {w :e Worlds|(bp (v_FORMULA w))} w)
 -> (ModalDia 1 {w :e Worlds|(bp (v_FORMULA w))} w)).
```

Here 1 simply indicates an index for the necessity/possibility notion of world and accessibility. The definitions of ModalBox and ModalDia follow the usual idea for Kripke semantics:

```
Definition ModalBox : set -> set -> set -> prop
  := fun i p w => forall v :e Worlds, AccReln i w v -> v :e p.
Definition ModalDia : set -> set -> set -> prop
  := fun i p w => exists v :e Worlds, AccReln i w v /\ v :e p.
```

Recall that AccReln i w v means w and v agree everywhere except possibly on i and that AccRelnSeq i $(w\ i)$ $(v\ i)$ holds. In this case i is 1. Hence

$$\mathsf{ModalBox}\ 1\ \{w \in \mathsf{Worlds} | (\mathsf{bp}\ (\mathsf{v\_FORMULA} w))\} w$$

holds iff

$$v \in \{w \in \mathsf{Worlds} | (\mathsf{bp}\ (\mathsf{v\_FORMULA} w))\}$$

holds for all $v \in$ Worlds that agree with w everywhere except possibly on 1 and AccRelnSeq 1 $(w\ 1)$ $(v\ 1)$ holds. Note that $w\ 1 \in$ World 1 if $w \in$ Worlds, so that World 1 is the set of Kripke worlds for the necessity-possibility modality. Likewise AccRelnSeq 1 is the corresponding accessibility relation on World 1. Let us temporarily use W for World 1 and $\leq$ for AccRelnSeq 1. It can be easily seen that

$$\mathsf{ModalBox}\ 1\ \{w \in \mathsf{Worlds} | (\mathsf{bp}\ (\mathsf{v\_FORMULA} w))\} w$$

holds iff

$$(\mathsf{bp}\ (\mathsf{v\_FORMULA}(\lambda i \in \mathsf{Univ1.if}\ i = 1\ \mathsf{then}\ v_1\ \mathsf{else}\ w\ i)))$$

holds for every $v_1 \in W$ such that $w\ 1 \leq v_1$. Likewise

$$\mathsf{ModalDia}\ 1\ \{w \in \mathsf{Worlds} | (\mathsf{bp}\ (\mathsf{v\_FORMULA} w))\} w$$

holds iff

$$(\mathsf{bp}\ (\mathsf{v\_FORMULA}(\lambda i \in \mathsf{Univ1.if}\ i = 1\ \mathsf{then}\ v_1\ \mathsf{else}\ w\ i)))$$

for some $v_1 \in W$ such that $w\ 1 \leq v_1$. This provides the connection between our (very) multimodal logic and the usual Kripke interpretation of a box and diamond.

The interpretation of permission and obligation are the same as possibility and necessity, except using the index 0 instead of 1.

The following is a SUMO-KM assertion using knows and believes:

```
(=>
  (knows ?AGENT ?FORMULA)
  (believes ?AGENT ?FORMULA))
```

These are translated using ModalBox with the resulting proposition looking as follows:

```
forall w :e Worlds,
 forall v_FORMULA, forall v_AGENT,
     v_AGENT w :e domseqm s_KNOWS 0 w
  -> v_AGENT w :e domseqm s_BELIEVES 0 w
  -> ((ModalBox (eval s_KNOWS (cons (v_AGENT w) nil))
                {w :e Worlds|(bp (v_FORMULA w))} w)
   -> (ModalBox (eval s_BELIEVES (cons (v_AGENT w) nil))
                {w :e Worlds|(bp (v_FORMULA w))} w)).
```

This states that if a certain box operator holds for a formula (the box operator corresponding to index eval s_KNOWS (cons (v_AGENT w) nil)), then another box operator holds for the formula (the box operator corresponding to index eval s_BELIEVES (cons (v_AGENT w) nil)).

We finally consider one modality that cannot be naturally thought of as a box or diamond operator:

```
(=>
  (holdsDuring ?TIME (leader ?X ?Y))
  (holdsDuring ?TIME (attribute ?Y Living)))
```

Instead of using ModalBox or ModalDia we use an abstract set_to_modal operator of type $\iota \to \iota \to (\iota \to \iota) \to \iota \to o$ about which we assume no properties. The SUMO-KM assertion above translates into the following set theoretic proposition:

```
forall w :e Worlds,
 forall v_X, forall v_Y, forall v_TIME,
     v_X w :e domseqm s_LEADER 0 w
  -> v_Y w :e domseqm s_LEADER 1 w
  -> v_TIME w :e domseqm s_HOLDSDURING 0 w
  -> v_Y w :e domseqm s_ATTRIBUTE 0 w
  -> ((set_to_modal s_HOLDSDURING
          {w :e Worlds|(bp (eval (s_LEADER w)
                                 (cons (v_X w)
                                  (cons (v_Y w) nil))))}
          (cons (v_TIME w) nil)
          w)
```

```
-> (set_to_modal s_HOLDSDURING
       {w :e Worlds|(bp (eval (s_ATTRIBUTE w)
                              (cons (v_Y w)
                               (cons (s_LIVING w) nil))))}
       (cons (v_TIME w) nil)
       w)).
```

The first argument to set_to_modal is the modality (s_HOLDSDURING in this case). The second argument is the set of worlds where the translated formula holds. The third argument is the spine of the modality (in this case a list with the single element v_TIME. The final argument is the current world w. It is easy to see that a particular choice of set_to$_{\mathsf{m}}$odal would yield ModalBox and another choice would yield ModalDia, demonstrating that set_to_modal is more general. Of course, the generality of set_to_modal also means we can deduce fewer conclusions from an assertion using set_to_modal.

4.2 Background for the Translation

Most of the background for the translation has already been presented as needed in the examples above. We describe the complete background here, other than the set theoretic concepts already formalized in Megalodon.

We first declare prefix notation $-$ for the unary minus function and $+$ for the binary addition function on surreal numbers (and so in particular on integers). We will sometimes need these due to our representation of lists as functions from natural numbers to sets. We will freely use 0, 1, 2, etc., for the usual finite ordinals, where n equals $\{0, \ldots, n-1\}$.

```
Prefix - 358 := minus_SNo.
Infix + 360 right := add_SNo.
```

As described above, we define nil and cons for lists (as functions from natural numbers to sets) to represent spines. We also define the function replseq1 for replacing an element of a list.

```
Let nil : set -> set := fun _ => 0.
Let cons : set -> (set -> set) -> set -> set
  := fun a l i => nat_primrec a (fun m _ => l m) i.
Let replseq1 : (set -> set) -> set -> set -> set -> set
  := fun l n a i => if i = n then a else l i.
```

For interpreting modal operators we declare an abstract family of sets of worlds World and a set Univ1 acting as our universe of discourse (and as an index for World).

```
Variable World:set -> set.
Variable Univ1:set.
```

We define Worlds as the cartesian product $\Pi_{x \in \mathsf{Univ1}} \mathsf{World}\ x$.

```
Let Worlds : set := Pi Univ1 World.
```

We declare an abstract operation to handle modal operators that do not fit neatly into the box and diamond paradigm.

```
Variable set_to_modal: set -> set -> (set -> set) -> set -> prop.
```

For each $x \in$ Univ1 we declare an abstract accessibility relation on World x. We then define AccReln lifting the abstract family of accessibility relations from World x to Worlds.

```
Variable AccRelnSeq:set -> set -> set -> prop.
Definition AccReln:set -> set -> set -> prop
 := fun i u v => u :e Worlds /\ v :e Worlds
              /\ AccRelnSeq i (u i) (v i)
              /\ forall j :e Univ1, j <> i -> u j = v j.
```

We next define the modal box and diamond operators, indexed by Univ1.

```
Definition ModalBox : set -> set -> set -> prop
  := fun i p w => forall v :e Worlds, AccReln i w v -> v :e p.
Definition ModalDia : set -> set -> set -> prop
  := fun i p w => exists v :e Worlds, AccReln i w v /\ v :e p.
```

The set interpreting a SUMO-KM term can be thought of as a tuple consisting of five pieces of information: the value (eval), whether or not it is variable arity (vararity), the minimum arity (arity), the domain information for the arguments (domseq) and the intended range (ran).

```
Variable eval:set -> (set -> set) -> set.
Variable vararity:set -> prop.
Variable arity:set -> set.
Variable domseq:set -> set -> set.
Variable ran:set -> set.
```

Lists are encoded as having type $\iota \to \iota$ as described above, so it is best to think of eval and domseq as functions that take one argument. After applying one argument eval x (of type $(\iota \to \iota) \to \iota$) is a function waiting to take a list of arguments and return a set. Likewise, domseq x returns a list of type $\iota \to \iota$, giving the list of intended domains of the list of arguments.

We define an auxiliary function popseq to pop an integer number of entries from the beginning of a list.

```
Definition popseq:set -> (set -> set) -> set -> set
  := fun n l i => l (n + i).
```

We then define domseqm, dom_of_fixedar, dom_of_varar and dom_of to handle the variable arity and fixed arity cases separately as described above.

```
Definition domseqm:set -> set -> set
 := fun u i =>
     if vararity u then
        domseq u (if i :e arity u then i else arity u)
     else
```

```
        domseq u i.

Definition dom_of_fixedar:set -> (set -> set) -> (set -> set) -> prop
 := fun ar dseq u =>
       (forall i :e ar, forall w :e Worlds, u i w :e dseq i w).

Definition dom_of_varar:set -> (set -> set) -> (set -> set) -> prop :=
  fun ar dseq u => exists n :e omega,
       ar c= n
    /\ (forall i :e ar, forall w :e Worlds, u i w :e dseq i w)
    /\ (forall i :e n, forall w :e Worlds, ar c= i -> u i w :e dseq ar w).

Definition dom_of:prop -> set -> (set -> set) -> (set -> set) -> prop :=
  fun varar ar dseq u =>
       varar /\ dom_of_varar ar dseq u
   \/ ~varar /\ dom_of_fixedar ar dseq u.
```

We assume the arity function returns a natural number (a member of the set ω) and that for appropriate arguments domseq will yeild either a subset of Univ1 or a subset of the power set of Univ1 (since these should be where interpretations of SUMO-KM classes live). We make a further similar assumption for the last class given in the variable arity case (which should be the intended domain for all of the optional arguments).

```
Hypothesis arity_omega : forall v, arity v :e omega.
Hypothesis arity_domseq : forall v, forall i :e arity v,
  forall w :e Worlds,
    domseq v i w :e Power Univ1 :\/: Power (Power Univ1).
Hypothesis vararity_domseq : forall v, vararity v ->
  forall w :e Worlds,
    domseq v (arity v) w :e Power Univ1 :\/: Power (Power Univ1).
```

We finally assume that when evaluated appropriately to a list of arguments in the intended domain we will obtain values in the intended range.

```
Hypothesis dom_ran : forall v, forall u:set -> set,
    dom_of (vararity v) (arity v) (domseq v) u
 -> forall w :e Worlds,
      eval (v w) (fun i => u i w) :e ran v w.
```

Our last background definitions will be bp ("bool to prop") and pb ("prop to bool").

```
Let bp:set -> prop := fun X => 0 :e X.
Let pb:prop -> set := fun p => if p then 1 else 0.
```

The function bp takes a set X to the proposition $0 \in X$. If X is 0, this is false since $0 \notin 0$. If X is 1, this is false since $0 \in 1$. The function pb takes the proposition false to 0 and the proposition true to 1.

4.3 The Translation

We now describe the translation itself. A first pass through the SUMO-KM files given records the typing information from domain, range, $\mathsf{domainsubclass}$, $\mathsf{rangesubclass}$ and $\mathsf{subrelation}$ assertions. A finite number of secondary passes determines which names will have avariable arity (either due to a direct assertion or due to being inferred to be in a variable arity class).

The final pass translates the assertions, and this is our focus here. Each SUMO-KM assertion is a SUMO-KM proposition φ which may have free variables in it. We further obtain a free variable w ranging over worlds introduced by the translation. Thus if we translate the SUMO-KM proposition φ into the set theoretic proposition φ', then the translated assertion will be

$$\forall w \in \mathsf{Worlds}.\forall x_1 \cdots x_n.G_1 \to \cdots G_m \to \varphi'$$

where $x_1, \ldots, x_n$ are the free variables in φ and $G_1, \ldots, G_m$ are the type guards for these free variables. Note that some of these free variables may be for spine variables (i.e., row variables) and may have type $\iota \to \iota$. Such variables may also have type guards.

SUMO-KM variables x translate to themselves where after translation x is a variable of type ι (ranging over sets). For SUMO-KM constants c we choose a name c' and declare this as having type ι. When a variable or constant is applied to a spine we translate the spines and use eval.

- $(x\ s)$ translates to $(\mathsf{eval}\ x\ s')$ where s' is the result of translating the SUMO-KM spine s.
- $(c\ s)$ translates to $(\mathsf{eval}\ c'\ s')$ where s' is the result of translating the SUMO-KM spine s and c' is the chosen set as a counterpart to the SUMO-KM constant c.

The only remaining case are κ binders.

- We translate $(\kappa x.\psi)$ to

  $$\{x\ w | x \in \mathsf{Univ1}^{\mathsf{Worlds}}, G_1 \wedge \ldots G_m \wedge \psi'\}$$

  where $G_1, \ldots, G_m$ are generated type guards for x and ψ' is the result of translating the SUMO-KM proposition ψ to a set theoretic proposition. Note that x ranges over functions from Worlds to $\mathsf{Univ1}$.

The translations of spines is relatively straightforward.

- The SUMO-KM spine $(t\ s)$ is translated to $(\mathsf{cons}\ t'\ s')$ where t' is the translation of t and s' is the translation of s.
- A spine variable ρ is translated to itself.[1]

[1]We describe the translation this way for simplicity of presentation. In reality a spine variable is sometimes followed by one argument which we think of as appending one more elemnt to the list. The function $\mathsf{replseq1}$ is used in such a case.

- The empty spine is translated to nil.

We consider each case of a SUMO-KM proposition (within our fragment). The usual logical operators are translated as the corresponding operators:

- $\bot$ and $\top$ translate simply to $\bot$ and $\top$.
- $(\neg\ \psi)$ translates to $\neg\psi'$ where ψ is a SUMO-KM proposition which translates to the set theoretic proposition ψ'.
- $(\psi\ \rightarrow\ \xi)$ translates to $\psi' \rightarrow \xi'$ where ψ and ξ are SUMO-KM propositions translate to the set theoretic propositions ψ' and ξ'.
- $(\psi\ \leftrightarrow\ \xi)$ translates to $\psi' \leftrightarrow \xi'$ where ψ and ξ are SUMO-KM propositions translate to the set theoretic propositions ψ' and ξ'.
- Theoretically, $\psi \wedge \xi$ translates to $\psi' \wedge \xi'$. Practically speaking in SUMO-KM conjunction is n-ary so it is more accurate to state that $(\mathsf{and}\ \psi_1\ \cdots \psi_n)$ translates to $\psi'_1 \wedge \cdots \wedge \psi'_n$ where $\psi_1, \ldots, \psi_n$ are SUMO-KM propositions translate to the set theoretic propositions $\psi'_1, \ldots, \psi'_n$.
- Again, theoretically $\psi \vee \xi$ translates to $\psi' \vee \xi'$. Practically, $(\mathsf{or}\ \psi_1\ \cdots \psi_n)$ translates to $\psi'_1 \vee \cdots \vee \psi'_n$ where $\psi_1, \ldots, \psi_n$ are SUMO-KM propositions translate to the set theoretic propositions $\psi'_1, \ldots, \psi'_n$.
- Theoretically, $\forall x.\psi$ translates to $\forall x.G_1 \rightarrow \cdots \rightarrow G_m \rightarrow \psi'$ where ψ' is the result of translating ψ and $G_1, \ldots, G_m$ are the generated type guards for x. Practically speaking, SUMO-KM allows several variables to be universally quantified at once, so it is more accurate to say $(\mathsf{forall}\ (x_1 \ldots x_n)\ \psi)$ translates to $\forall x_1 \ldots x_n.G_1 \rightarrow \cdots \rightarrow G_m \rightarrow \psi'$ where $x_1, \ldots, x_n$ are variables, $G_1, \ldots, G_m$ are the generated type guards for these variables and ψ' is the set theoretic proposition obtained by translating the SUMO-KM proposition ψ.
- Again, theoretically $\exists x.\psi$ translates to $\exists x.G_1 \wedge \cdots \wedge G_m \wedge \psi'$ and practically $(\mathsf{exists}\ (x_1 \ldots x_n)\ \psi)$ translates to $\exists x_1 \ldots x_n.G_1 \wedge \cdots \wedge G_m \wedge \psi'$ where $x_1, \ldots, x_n$ are variables, $G_1, \ldots, G_m$ are the generated type guards for these variables and ψ' is the set theoretic proposition obtained by translating the SUMO-KM proposition ψ.
- $(t_1 = t_2)$ translates to $t'_1 = t'_2$ where t_1 and t_2 are SUMO-KM terms which translate to sets t'_1 and t'_2.

We use set membership and inclusion to interpret `instance` and `subclass`.

- $(\texttt{instance}\ t_1\ t_2)$ translates to $t'_1 \in t'_2$ where t_1 and t_2 are SUMO-KM terms which translate to sets t'_1 and t'_2.
- $(\texttt{subclass}\ t_1\ t_2)$ translates to $t'_1 \subseteq t'_2$ where t_1 and t_2 are SUMO-KM terms which translate to sets t'_1 and t'_2.

The final special cases are for modalities.

- Suppose we are translating ($\mathsf{modalAttribute}\ \psi\ M$) where ψ is a SUMO-KM proposition and M is Obligation, Permission, Necessity or Possibility. We first translate ψ to ψ' and return

  $$O\ i\ \{w \in \mathsf{Worlds}|\psi'\}\ w.$$

  Here O is ModalBox if M is `Obligation` or `Necessity` and O is ModalDia if M is `Permission` or `Possibility`. Also, i is 0 if M is `Obligation` or `Permission` and i is 1 if M is `Necessity` or `Possibility`. Note that ψ' likely has w (the world variable) free, so that $\{w \in mathsfWorlds|\psi'\}$ is the set of all worlds where ψ' holds. This set is passed as an argument to the modal operator O. The last argument to the modal operator is the current world w. As a consequence the free w occurrences of ψ' are bound in

  $$O\ i\ \{w \in \mathsf{Worlds}|\psi'\}\ w$$

  and the only free occurrence of w is the last argument of O.

- Suppose we are translating ($M\ t\ \psi$) where M is either `knows`, `believes` or `desires`, t is a SUMO-KM term and ψ is a SUMO-KM proposition. We translate t as a spine (of length 1) to yield t' (of type $\iota \to \iota$) and ψ to a proposition ψ'. The reason for translating t as a spine is intended to handle cases where a modality has more than one non-proposition argument, although this does not apply in the three cases of `knows`, `believes` and `desires`. We choose an abstract set M' (s_KNOWS, s_BELIEVES or s_DESIRES) depending on M. We return

  $$\mathsf{ModalBox}\ (\mathsf{eval}\ M'\ t')\ \{w \in \mathsf{Worlds}|\psi'\}\ w.$$

- For the remaining modal cases we return

  $$\mathsf{set\_to\_modal}\ M'\ \{w \in \mathsf{Worlds}|\psi'\}\ t'\ w$$

  where ψ' is the result of translating the (unique) proposition argument, M' is a set chosen to represent the modality and t' is the result of translating the spine after deleting the proposition argument. For example, (`holdsDuring` $t\ \psi$) translates to

  $$\mathsf{set\_to\_modal}\ \mathsf{s\_HOLDSDURING}\ \{w \in \mathsf{Worlds}|\psi'\}\ t'\ w.$$

If no special case applies for a SUMO-KM proposition, then it is simply translated into a set (as a SUMO-KM term) and then coerced into being a proposition using bp. That is, if we are translating ($c\ s$), then we choose an abstract set c' to represent c as a set, translate the SUMO-KM spine s to be the list s' (of type $\iota \to \iota$) and return $\mathsf{eval}\ c'\ s'$.

Chapter 5

Test Queries and Theorem Proving

A SUMO "test query" is a collection of local assertions followed by an item of the form (`query`ψ) where ψ is a SUMO proposition. When using the SUMO-K translation such queries translate to set theoretical proposition

$$\exists x_1 \ldots x_n . G_1 \wedge \cdots \wedge G_m \wedge \psi'$$

where ψ' is the result of translation ψ, $x_1, \ldots, x_n$ are the free variables in ψ and $G_1, \ldots, G_m$ are the type guards generated for these free variables. When using the SUMO-KM translation such queries ranslate to set theoretical proposition

$$\forall w \in \mathsf{Worlds}. \exists x_1 \ldots x_n . G_1 \wedge \cdots \wedge G_m \wedge \psi'$$

where ψ' is the result of translation ψ, $x_1, \ldots, x_n$ are the free variables in ψ and $G_1, \ldots, G_m$ are the type guards generated for these free variables.

After translating the SUMO assertions and local assertions of the test query, the translated query becomes a conjecture of the set theory. If provable, it can be proven interactively (e.g., in Megalodon) or outsourced via a TH0 translation to a higher-order automated theorem prover.

We give a few examples.

5.1 Basic First Order Examples

We start by considering some examples that make no use of kappa or modalities. These only require first-order reasoning.

The test query `TQG1` declares a constant `Org1-1` to be an Organization. and then queries whether there is a member of the organization. In SUMO this appears as follows:

```
(instance Org1-1 Organization)

(query (member ?MEMBER Org1-1))
```

The SUMO-K translated version in Megalodon first includes the background and the result of translating SUMO's Merge file.[1] The additional information given by the query is translated to the following:

```
Variable s_ORG1_x2D1:set.
Hypothesis s_ORG1_x2D1__arity: arity s_ORG1_x2D1 = 0.
Hypothesis s_ORG1_x2D1__not_vararity: ~ vararity s_ORG1_x2D1.
Hypothesis p5326: (s_ORG1_x2D1 :e s_ORGANIZATION).
```

The query itself is translated to the following theorem declaration which is given without proof (using `admit` where the proof should be).

```
Theorem p5327: exists v_MEMBER,
      v_MEMBER :e domseqm s_MEMBER 0
   /\ (bp (eval s_MEMBER (cons v_MEMBER (cons s_ORG1_x2D1 nil)))).
admit.
Qed.
```

For people interested in the TH0 syntax many HO ATPs use, the conjecture looks as follows:

```
thf(conj_TQG1_9201,conjecture,
 (? [X1271:$i] :
    (((c_In @ X1271) @ ((((((domseqm @ c_Univ1) @ eval)
           @ vararity) @ arity) @ domseq) @ s_5FMEMBER) @ c_Empty))
   & ((^ [X1272:$i] : ((c_In @ c_Empty) @ X1272))
       @ ((eval @ s_5FMEMBER) @
          (((^ [X1272:$i] : (^ [X1273:($i > $i)] : (^ [X1274:$i] :
           (((nat_5Fprimrec @ X1272) @ (^ [X1275:$i] : (^ [X1276:$i] : (X1273 @ X1275))))
            @ X1274)))) @ X1271)
           @ (((^ [X1272:$i] : (^ [X1273:($i > $i)] : (^ [X1274:$i] :
              (((nat_5Fprimrec @ X1272) @ (^ [X1275:$i] : (^ [X1276:$i] : (X1273 @ X1275))))
               @ X1274)))) @ s_5FORG1_5Fx2D1) @ (^ [X1272:$i] : c_Empty)))))))).
```

We leave it to the reader to look at the details of the correspondence between the Megalodon statement and the TH0 statement with the following hints: the outer TH0 `?` corresponds to the outer `exists`, the TH0 `&` corresponds to the Megalodon conjunction `/\\` and `nat_5Fprimrec` is an operation for using primitive recursion to form lists for spines.

The SUMO-KM translated version of the same test query looks very similar with the addition of worlds. In Megalodon the translated query first includes the background and the result of translating SUMO's Merge file. The additional information given by the query is translated to the following:

```
Variable s_ORG1_x2D1:set.
Hypothesis s_ORG1_x2D1__arity: arity s_ORG1_x2D1 = 0.
Hypothesis s_ORG1_x2D1__not_vararity: ~ vararity s_ORG1_x2D1.
Hypothesis p5317: forall w :e Worlds,
                     ((s_ORG1_x2D1 w) :e (s_ORGANIZATION w)).
```

[1] We are using a copy of `Merge.kif` from 2022.

The query itself is translated to the following theorem declaration which is given without proof (using `admit` where the proof should be).

```
Theorem p5318: forall w :e Worlds, exists v_MEMBER,
    v_MEMBER w :e domseqm s_MEMBER 0 w
 /\ (bp (eval (s_MEMBER w)
              (cons (v_MEMBER w) (cons (s_ORG1_x2D1 w) nil)))).
admit.
Qed.
```

Translated to TH0 it looks like this:

```
thf(conj_TQG1_9378,conjecture,(! [X1282:$i] :
  (((c_In @ X1282) @ ((c_Pi @ c_Univ1) @ c_World))
=> (? [X1283:$i] :
     (((c_In @ ((ap @ X1283) @ X1282))
             @ ((ap @ (((((((((domseqm @ c_World) @ c_Univ1) @ set_5Fto_5Fmodal)
                         @ c_AccRelnSeq) @ eval) @ vararity) @ arity)
                        @ domseq) @ s_5FMEMBER) @ c_Empty))
                 @ X1282))
    & ((^ [X1284:$i] : ((c_In @ c_Empty) @ X1284))
     @ ((eval @ ((ap @ s_5FMEMBER) @ X1282))
              @ (((^ [X1284:$i] : (^ [X1285:($i > $i)] : (^ [X1286:$i] :
                   (((nat_5Fprimrec @ X1284)
                     @ (^ [X1287:$i] : (^ [X1288:$i] : (X1285 @ X1287))))
                    @ X1286))))
                @ ((ap @ X1283) @ X1282))
               @ (((^ [X1284:$i] : (^ [X1285:($i > $i)] : (^ [X1286:$i] :
                    (((nat_5Fprimrec @ X1284)
                      @ (^ [X1287:$i] : (^ [X1288:$i] : (X1285 @ X1287))))
                     @ X1286))))
                 @ ((ap @ s_5FORG1_5Fx2D1) @ X1282))
                @ (^ [X1284:$i] : c_Empty))))))))).
```

A proof would likely use the following SUMO assertion from Merge.kif:

```
(=>
  (instance ?COLL Collection)
  (exists (?OBJ)
    (member ?OBJ ?COLL)))
```

The translated form of this SUMO assertion is as follows:

```
Hypothesis p355: forall w :e Worlds, forall v_COLL,
    v_COLL w :e (s_ENTITY w)
 -> v_COLL w :e domseqm s_MEMBER 1 w
 -> (((v_COLL w) :e (s_COLLECTION w))
  -> (exists v_OBJ, v_OBJ w :e domseqm s_MEMBER 0 w
                 /\ (bp (eval (s_MEMBER w)
                              (cons (v_OBJ w)
                                    (cons (v_COLL w) nil)))))).
```

Next we consider a test query (TQG10) using partitions.

```
(=>
  (and
    (instance ?A Animal)
    (not
      (exists (?PART)
        (and
          (instance ?PART SpinalColumn)
          (part ?PART ?A)))))
  (not
    (instance ?A Vertebrate)))

(not
  (exists (?SPINE)
    (and
      (instance ?SPINE SpinalColumn)
      (part ?SPINE BananaSlug10-1))))

(instance BananaSlug10-1 Animal)

(and
  (instance BodyPart10-1 BodyPart)
  (component BodyPart10-1 BananaSlug10-1))

(query (instance BananaSlug10-1 Invertebrate))
```

The query after translation looks as follows:

```
Variable s_SPINALCOLUMN:set.
Hypothesis s_SPINALCOLUMN__arity: arity s_SPINALCOLUMN = 0.
Hypothesis s_SPINALCOLUMN__not_vararity: ~ vararity s_SPINALCOLUMN.
Hypothesis p5317: forall w :e Worlds, forall v_A, v_A w :e (s_ENTITY w)
 -> v_A w :e domseqm s_PART 1 w
 -> (((v_A w) :e (s_ANIMAL w))
  /\ (~ (exists v_PART, v_PART w :e (s_ENTITY w)
           /\ v_PART w :e domseqm s_PART 0 w
           /\ ((v_PART w) :e (s_SPINALCOLUMN w))
           /\ (bp (eval (s_PART w)
                        (cons (v_PART w) (cons (v_A w) nil))))))
  -> (~ ((v_A w) :e (s_VERTEBRATE w)))).
Variable s_BANANASLUG10_x2D1:set.
Hypothesis s_BANANASLUG10_x2D1__arity: arity s_BANANASLUG10_x2D1 = 0.
Hypothesis s_BANANASLUG10_x2D1__not_vararity: ~ vararity s_BANANASLUG10_x2D1.
Hypothesis p5318: forall w :e Worlds,
 (~ (exists v_SPINE, v_SPINE w :e (s_ENTITY w)
      /\ v_SPINE w :e domseqm s_PART 0 w
      /\ ((v_SPINE w) :e (s_SPINALCOLUMN w))
      /\ (bp (eval (s_PART w)
                   (cons (v_SPINE w)
                         (cons (s_BANANASLUG10_x2D1 w) nil)))))).
```

```
Hypothesis p5319: forall w :e Worlds, ((s_BANANASLUG10_x2D1 w) :e (s_ANIMAL w)).
Variable s_BODYPART10_x2D1:set.
Hypothesis s_BODYPART10_x2D1__arity: arity s_BODYPART10_x2D1 = 0.
Hypothesis s_BODYPART10_x2D1__not_vararity: ~ vararity s_BODYPART10_x2D1.
Hypothesis p5320: forall w :e Worlds,
    ((s_BODYPART10_x2D1 w) :e (s_BODYPART w))
 /\ (bp (eval (s_COMPONENT w)
              (cons (s_BODYPART10_x2D1 w) (cons (s_BANANASLUG10_x2D1 w) nil)))).
Theorem p5321: forall w :e Worlds,
  ((s_BANANASLUG10_x2D1 w) :e (s_INVERTEBRATE w)).
admit.
Qed.
```

A key SUMO assertion from `Merge.kif` used to prove this is the following:

```
(partition Animal Vertebrate Invertebrate)
```

This assertion after translation looks as follows:

```
Hypothesis p4064: forall w :e Worlds,
 (bp (eval (s_PARTITION w)
           (cons (s_ANIMAL w)
                 (cons (s_VERTEBRATE w)
                       (cons (s_INVERTEBRATE w) nil))))).
```

Finally we consider a test query (`TQG10c`) making use of a partition into more than two subclasses. This will also be the first test query making use of a row variable. We add two local assertions about lists and the behavior of the SUMO function `ListFn`. These could in principle be included in `Merge.kif` at some future date.

```
(forall (?I) (not (inList ?I (ListFn))))

(forall (?I ?H) (=> (inList ?I (ListFn ?H @ROW))
                    (or (equal ?I ?H) (inList ?I (ListFn @ROW)))))

(query (forall (?O)
        (=> (instance ?O Organism)
         (=> (not (instance ?O Animal))
          (=> (not (instance ?O Microorganism))
           (or (instance ?O Plant) (instance ?O Fungus)))))))
```

The query after translation looks as follows:

```
Hypothesis p5316: forall w :e Worlds,
 (forall v_I,
     v_I w :e domseqm s_INLIST 0 w
  -> (~ (bp (eval (s_INLIST w) (cons (v_I w) (cons (eval (s_LISTFN w) nil) nil)))))).
Hypothesis p5317: forall w :e Worlds, forall r_ROW:set -> set,
    dom_of (vararity s_LISTFN)
           (arity s_LISTFN + - 1)
           (popseq 1 (domseq s_LISTFN))
```

```
           r_ROW
 -> dom_of (vararity s_LISTFN)
           (arity s_LISTFN)
           (domseq s_LISTFN)
           r_ROW
 -> (forall v_I, v_I w :e domseqm s_INLIST 0 w
      -> (forall v_H, v_H w :e domseqm s_LISTFN 0 w
           -> ((bp (eval (s_INLIST w)
                        (cons (v_I w)
                              (cons (eval (s_LISTFN w)
                                          (cons (v_H w) r_ROW)) nil))))
            -> ((v_I w) = (v_H w))
             \/ (bp (eval (s_INLIST w)
                         (cons (v_I w)
                               (cons (eval (s_LISTFN w) r_ROW) nil))))))).
Theorem p5318: forall w :e Worlds,
 (forall v_O, v_O w :e (s_ENTITY w)
   -> (((v_O w) :e (s_ORGANISM w))
    -> ((~ ((v_O w) :e (s_ANIMAL w)))
     -> ((~ ((v_O w) :e (s_MICROORGANISM w)))
      -> ((v_O w) :e (s_PLANT w)) \/ ((v_O w) :e (s_FUNGUS w)))))).
admit.
Qed.
```

Two key SUMO assertions from Merge.kif used to prove this are the following:

```
(=>
   (exhaustiveDecomposition @ROW)
   (=>
      (inList ?ELEMENT (ListFn @ROW))
      (instance ?ELEMENT Class)))

...

(partition Organism Animal Plant Fungus Microorganism)
```

These assertions after translation looks as follows:

```
Hypothesis p137: forall w :e Worlds, forall v_ELEMENT, forall r_ROW:set -> set,
    v_ELEMENT w :e domseqm s_INLIST 0 w
 -> v_ELEMENT w :e (s_ENTITY w)
 -> dom_of (vararity s_EXHAUSTIVEDECOMPOSITION)
           (arity s_EXHAUSTIVEDECOMPOSITION)
           (domseq s_EXHAUSTIVEDECOMPOSITION)
           r_ROW
 -> dom_of (vararity s_LISTFN)
           (arity s_LISTFN)
           (domseq s_LISTFN)
           r_ROW
 -> ((bp (eval (s_EXHAUSTIVEDECOMPOSITION w) r_ROW))
```

```
  -> ((bp (eval (s_INLIST w)
                (cons (v_ELEMENT w)
                      (cons (eval (s_LISTFN w) r_ROW) nil))))
   -> ((v_ELEMENT w) :e (s_CLASS w)))).
```

...

```
Hypothesis p4025: forall w :e Worlds, (bp (eval (s_PARTITION w) (cons (s_ORGANISM w) (cons (s_ANIMAL w
```

5.2 Kappa Examples

An example of a test query involving κ is the following (`kappa1`):

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (=> (part ?E ?V)
      (instance ?E (KappaFn ?x (and (part ?x ?V) (instance ?x Electron)))))))))
```

The query asks if we know, given an atom V and an electron E where E is part of V, do we know E is an instance of the κ-formed class of all electrons that are part of V. After translation the query looks as follows:

```
Theorem p5315: forall w :e Worlds,
 (forall v_V, v_V w :e (s_ENTITY w) -> v_V w :e domseqm s_PART 1 w
 -> (((v_V w) :e (s_ATOM w))
 -> (forall v_E, v_E w :e (s_ENTITY w) -> v_E w :e domseqm s_PART 0 w
   -> (((v_E w) :e (s_ELECTRON w))
   -> ((bp (eval (s_PART w) (cons (v_E w) (cons (v_V w) nil))))
     -> ((v_E w) :e {v_X w | v_X :e Univ1 :^: Worlds,
                             v_X w :e domseqm s_PART 0 w
                          /\ v_X w :e (s_ENTITY w)
                          /\ (bp (eval (s_PART w)
                                       (cons (v_X w) (cons (v_V w) nil))))
                          /\ ((v_X w) :e (s_ELECTRON w))}))))))).
admit.
Qed.
```

A similar query (essentially the converse) is `kappa2`:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (=> (instance ?E (KappaFn ?x (and (part ?x ?V) (instance ?x Electron))))
      (part ?E ?V)))))))
```

After translation the query looks as follows:

```
Theorem p5315: forall w :e Worlds,
 (forall v_V, v_V w :e (s_ENTITY w) -> v_V w :e domseqm s_PART 1 w
   -> (((v_V w) :e (s_ATOM w))
   -> (forall v_E, v_E w :e (s_ENTITY w) -> v_E w :e domseqm s_PART 0 w
     -> (((v_E w) :e (s_ELECTRON w))
```

```
      -> (((v_E w) :e {v_X w | v_X :e Univ1 :^: Worlds,
                               v_X w :e domseqm s_PART 0 w
                            /\ v_X w :e (s_ENTITY w)
                            /\ (bp (eval (s_PART w)
                                          (cons (v_X w)
                                                (cons (v_V w) nil))))
                            /\ ((v_X w) :e (s_ELECTRON w))})
       -> (bp (eval (s_PART w) (cons (v_E w) (cons (v_V w) nil))))))))).
admit.
Qed.
```

5.3 Modal Examples

A simple example involving modalities is the following (`modal2_2`):

```
(query
  (forall (?P)
  (=> (not (modalAttribute (modalAttribute ?P Possibility) Permission))
      (modalAttribute (not (modalAttribute ?P Necessity)) Obligation))))
```

In words, if P does not have permission to be possible, then is P obliged to not be necessary? After translation, the query looks as follows:

```
Theorem p5315: forall w :e Worlds,
 (forall v_P,
   ((~ (ModalDia 0 {w :e Worlds|(ModalDia 1 {w :e Worlds|(bp (v_P w))} w)} w))
 -> (ModalBox 0 {w :e Worlds|(~ (ModalBox 1 {w :e Worlds|(bp (v_P w))} w))} w))).
admit.
Qed.
```

The following SUMO assertion from `Merge.kif` is necessary for the proof:

```
(=>
   (modalAttribute ?FORMULA Necessity)
   (modalAttribute ?FORMULA Possibility))
```

The translated form of this assertion looks as follows:

```
Hypothesis p4957: forall w :e Worlds, forall v_FORMULA,
   ((ModalBox 1 {w :e Worlds|(bp (v_FORMULA w))} w)
 -> (ModalDia 1 {w :e Worlds|(bp (v_FORMULA w))} w)).
```

Let us also consider a test query using the `believes` modality (`modal3`).

```
(instance John Human)
(instance Sue Human)

(believes John (acquaintance John Sue))

(query (exists (?X) (believes John (acquaintance John ?X))))
```

The translated version of this query looks as follows:

```
Variable s_JOHN:set.
Hypothesis s_JOHN__arity: arity s_JOHN = 0.
Hypothesis s_JOHN__not_vararity: ~ vararity s_JOHN.
Hypothesis p5315: forall w :e Worlds, ((s_JOHN w) :e (s_HUMAN w)).
Variable s_SUE:set.
Hypothesis s_SUE__arity: arity s_SUE = 0.
Hypothesis s_SUE__not_vararity: ~ vararity s_SUE.
Hypothesis p5316: forall w :e Worlds, ((s_SUE w) :e (s_HUMAN w)).
Hypothesis p5317: forall w :e Worlds,
 (ModalBox (eval s_BELIEVES (cons (s_JOHN w) nil))
           {w :e Worlds|(bp (eval (s_ACQUAINTANCE w)
                            (cons (s_JOHN w) (cons (s_SUE w) nil))))}
           w).
Theorem p5318: forall w :e Worlds,
 (exists v_X, v_X w :e domseqm s_ACQUAINTANCE 1 w
   /\ (ModalBox (eval s_BELIEVES (cons (s_JOHN w) nil))
                {w :e Worlds|(bp (eval (s_ACQUAINTANCE w)
                                 (cons (s_JOHN w) (cons (v_X w) nil))))}
                w)).
admit.
Qed.
```

In this case we briefly consider an interactive proof of this query in Megalodon. We begin by proving a claim:

```
claim L1: forall x, x :e s_HUMAN w -> x :e domseqm s_ACQUAINTANCE 1 w.
```

This can be proven using the typing information about s_ACQUANTANCE. For a reader who wants more detail, the subproof of L1 looks as follows:

```
{ let x. assume H1.
  prove x :e (if vararity s_ACQUAINTANCE then
                  domseq s_ACQUAINTANCE
                         (if 1 :e arity s_ACQUAINTANCE then
                            1
                          else
                            arity s_ACQUAINTANCE)
              else domseq s_ACQUAINTANCE 1)
               w.
  rewrite If_i_0 (vararity s_ACQUAINTANCE)
                 (domseq s_ACQUAINTANCE
                         (if 1 :e arity s_ACQUAINTANCE then
                            1
                          else
                            arity s_ACQUAINTANCE))
                 (domseq s_ACQUAINTANCE 1)
                 s_ACQUAINTANCE__not_vararity.
  prove x :e domseq s_ACQUAINTANCE 1 w.
  rewrite s_ACQUAINTANCE__domseq_1.
```

```
  prove x :e (fun w :e Worlds => s_HUMAN w) w.
  rewrite beta Worlds (fun w => s_HUMAN w) w Hw.
  exact H1.
}
```

After we have this claim, we can prove the main existential goal using s_SUE as the witness and easily verifying this witness satisfies the requirements.

```
witness s_SUE.
apply andI.
- prove s_SUE w :e domseqm s_ACQUAINTANCE 1 w.
  apply L1.
  exact p5316 w Hw.
- exact p5317 w Hw.
Qed.
```

Chapter 6

Automation

Once a test query has been translated from SUMO-KM to Megalodon, Megalodon can produce a TH0 file to be read by higher-order automated theorem provers. We will call such generated problems "main goal" problems. In addition, if a (partial or complete) proof has been written in Megalodon, then Megalodon can produce TH0 files corresponding to each subgoal that occurs during the checking of the proof. We will call problems generated by this method "subgoal" problems.

We have currently not done generally tested higher-order ATPs on such problems. However, we have run Lash 1.13[1] with certain flag settings on some of the problems. We briefly report prelimary results here.

We only have 34 "main goal" problems at the moment. Lash is able to 6 of these with 60 second timeout (four modes given 15 seconds each). One example it is able to prove is `modal2_2`.

We have 936 "subgoals" problems at the moment. 26 of these subgoal problems arise from the proof of `modal3` described above. With the same portfolio as above, Lash can prove 412 of the 936 within 60 seconds.

Eval with state-of-the-art HO ATPs

We have run Zipperposition, Vampire and E (HO) on the sumo2set problems. Vampire with its higher-order schedule can prove 276 (1 from main goals) out of the 576 problems in 60s, 304 problerms (2 from main goals) in 300s, and 307 problems (2 from main goals) in 1200s. Zipperposition using its CASC portfolio mode proves 309 of the problems in 600s. Four of those are the main problems. The two systems together prove 357 problems (4 main ones).

[1] `http:grid01.ciirc.cvut.cz/~chad/lash-1.13.tgz`

Bibliography

[1] Christoph Benzmüller and Adam Pease. Higher-Order Aspects and Context in SUMO. In Ivan José Varzinczak Jos Lehmann and Alan Bundy, editors, *Special issue on Reasoning with context in the Semantic Web*, volume 12-13. Science, Services and Agents on the World Wide Web, 2012.

[2] Chad E. Brown and Karol Pąk. A tale of two set theories. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, volume 11617 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2019.

[3] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019.

[4] Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 448–463. Springer, 2020.

[5] Jan Jakubuv and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, volume 10383 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2017.

[6] Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019,*

September 9-12, 2019, Portland, OR, USA, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[7] Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Automating formalization by statistical and semantic parsing of mathematics. In *ITP*, volume 10499 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2017.

[8] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Learning to parse on aligned corpora (rough diamond). In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 227–233. Springer, 2015.

[9] Cezary Kaliszyk, Josef Urban, Jiří Vyskočil, and Herman Geuvers. Developing corpus-based translation methods between informal and formal mathematics: Project description. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *LNCS*, pages 435–439. Springer, 2014.

[10] Laura Kovács and Andrei Voronkov. 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, 2013. Springer-Verlag New York, Inc.

[11] Saul A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9:67–96, 1963.

[12] Ian Niles and Adam Pease. Toward a Standard Upper Ontology. In Chris Welty and Barry Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, pages 2–9, 2001.

[13] Adam Pease. *Ontology: A Practical Guide*. Articulate Software Press, Angwin, CA, 2011.

[14] Adam Pease. Arithmetic and inference in a large theory. In *AI in Theorem Proving*, 2019.

[15] Adam Pease and Stephan Schulz. Knowledge Engineering for Large Ontologies with Sigma KEE 3.0. In *The International Joint Conference on Automated Reasoning*, 2014.

[16] Adam Pease, Geoff Sutcliffe, Nick Siegel, and Steven Trac. Large Theory Reasoning with SUMO at CASC. *AI Communications, Special issue on Practical Aspects of Automated Reasoning*, 23(2-3):137–144, 2010.

[17] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002.

[18] Stephan Schulz, Geoff Sutcliffe, Josef Urban, and Adam Pease. Detecting inconsistencies in large first-order knowledge bases. In *Proceedings of CADE 26*, pages 310–325. Springer, 2017.

[19] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-order Form with Arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2012)*, pages 406–419, 2012.

[20] Qingxiang Wang, Chad E. Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *CPP*, pages 85–98. ACM, 2020.

[21] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In *CICM*, volume 11006 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2018.

Chapter 7

Appendix

Examples We briefly consider two first-order example queries and three queries involving κ. Queries differ from assertions in that their free variables are implicitly existentially quantified, with implicit type guards added conjunctively. We prove the translated query in the Megalodon interactive prover (the successor to the Egal system [2]).

Oure first example is given by the SUMO query:

```
(instance Org1-1 Organization)
(query (member ?MEMBER Org1-1))
```

This translates into Megalodon as follows:

```
Variable s_ORG1_x2D1:set.
Hypothesis p5315: (s_ORG1_x2D1 :e s_ORGANIZATION).
Theorem p5316: exists v_MEMBER, v_MEMBER :e s_PHYSICAL
    /\ (bp (s_MEMBER (cons v_MEMBER (cons s_ORG1_x2D1 nil)))).
```

The (interactively constructed) proof makes use of (translated) SUMO assertions that all collections have a physical object member and that organizations are collections.

Our second example is given by the SUMO query:

```
(=>
  (and
    (instance ?A Animal)
    (not
      (exists (?PART)
        (and
          (instance ?PART SpinalColumn)
          (part ?PART ?A)))))
  (not
    (instance ?A Vertebrate)))

(not
```

```
  (exists (?SPINE)
    (and
      (instance ?SPINE SpinalColumn)
      (part ?SPINE BananaSlug10-1))))

(instance BananaSlug10-1 Animal)

(and
  (instance BodyPart10-1 BodyPart)
  (component BodyPart10-1 BananaSlug10-1))

(query (instance BananaSlug10-1 Invertebrate))
```

This translates into the following Megalodon formalization:

```
Variable s_SPINALCOLUMN:set.
Hypothesis p5320: forall v_A, v_A :e s_ENTITY -> v_A :e s_OBJECT ->
 ((v_A :e s_ANIMAL)
   /\ (~ (exists v_PART, v_PART :e s_ENTITY /\ v_PART :e s_OBJECT
            /\ (v_PART :e s_SPINALCOLUMN) /\ (bp (s_PART (cons v_PART (cons v_A nil))))))
   -> (~ (v_A :e s_VERTEBRATE))).
Variable s_BANANASLUG10_x2D1:set.
Hypothesis p5321: (~ (exists v_SPINE, v_SPINE :e s_ENTITY /\ v_SPINE :e s_OBJECT
     /\ (v_SPINE :e s_SPINALCOLUMN) /\ (bp (s_PART (cons v_SPINE (cons s_BANANASLUG10_x2D1 nil)))))).
Hypothesis p5322: (s_BANANASLUG10_x2D1 :e s_ANIMAL).
Variable s_BODYPART10_x2D1:set.
Hypothesis p5323: (s_BODYPART10_x2D1 :e s_BODYPART)
  /\ (bp (s_COMPONENT (cons s_BODYPART10_x2D1 (cons s_BANANASLUG10_x2D1 nil)))).
Theorem p5324: (s_BANANASLUG10_x2D1 :e s_INVERTEBRATE).
```

The proof uses the translation of the SUMO assertion that the classes of vertebrates and invertebrates form a partition of the class of animals.

Our κ examples are all variants of the same idea, and all are easily provable. The first κ example query is given in SUMO as follows:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (=> (part ?E ?V)
      (instance ?E (KappaFn ?x (and (part ?x ?V) (instance ?x Electron)))))))))
```

This translates to the following Megalodon formalization:

```
Theorem p5326: (forall v_V, v_V :e s_ENTITY -> v_V :e s_OBJECT -> ((v_V :e s_ATOM)
 -> (forall v_E, v_E :e s_ENTITY -> v_E :e s_OBJECT -> ((v_E :e s_ELECTRON)
 -> ((bp (s_PART (cons v_E (cons v_V nil))))
 -> (v_E :e {v_X :e Univ1 | v_X :e s_OBJECT /\ v_X :e s_ENTITY
      /\ (bp (s_PART (cons v_X (cons v_V nil)))) /\ (v_X :e s_ELECTRON)})))))).
```

The second κ example query is given in SUMO as follows:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
```

```
  (=> (instance ?E (KappaFn ?x (and (part ?x ?V) (instance ?x Electron))))
      (part ?E ?V))))))
```

This translates to the following Megalodon formalization:

```
Theorem p5327: (forall v_V, v_V :e s_ENTITY -> v_V :e s_OBJECT -> ((v_V :e s_ATOM) ->
 (forall v_E, v_E :e s_ENTITY -> v_E :e s_OBJECT -> ((v_E :e s_ELECTRON)
 -> ((v_E :e {v_X :e Univ1 | v_X :e s_OBJECT /\ v_X :e s_ENTITY
        /\ (bp (s_PART (cons v_X (cons v_V nil)))) /\ (v_X :e s_ELECTRON)})
 -> (bp (s_PART (cons v_E (cons v_V nil))))))))).
```

The final κ example does not use κ in the statement, though κ is vital to proving the translated theorem. In SUMO the example is given as follows:

```
(query (forall (?V) (=> (instance ?V Atom)
        (forall (?E) (=> (instance ?E Electron)
  (exists (?C)
    (and (instance ?C Class)
      (<=> (part ?E ?V)
           (instance ?E ?C)))))))))
```

This translates to the following Megalodon formalization:

```
Theorem p5328: (forall v_V, v_V :e s_ENTITY -> v_V :e s_OBJECT -> ((v_V :e s_ATOM) ->
 (forall v_E, v_E :e s_ENTITY -> v_E :e s_OBJECT -> ((v_E :e s_ELECTRON)
 -> (exists v_C, v_C :e s_ENTITY /\ v_C :e s_CLASS /\ (v_C :e s_CLASS)
      /\ ((bp (s_PART (cons v_E (cons v_V nil)))) <-> (v_E :e v_C))))))).
```

Converting SUMO to First Order All the strictly higher-order content in SUMO was previously lost in translation to first-order, whether TPTP or TF0. 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

SUMO has no native implementation in a theorem prover, and has no formal semantics beyond that of standard first order logic, so the process of translating SUMO into a language with a fully specified semantics, such as TPTP_FOF, TF0 or THF gives SUMO its semantics.

Type Mechanisms All relations (including functions) in SUMO have a type signature. As a consequence, we don't need an explicit syntax for types/sorts of variables, and can deduce them automatically. We can have classes as well as instances as arguments. The `domain` and `range` relations are meta-predicates that direct the Sigma translators to state that arguments to a given relation (or the return type of a function, respectively) are instances of a given type. The `domainSubclass`, and `rangeSubclass` relations state that arguments to a given relation (or the return type of a function, respectively) are a given class or one of its subclasses. For example

```
(domain DensityFn 1 MassMeasure)
(domain DensityFn 2 VolumeMeasure)
(instance DensityFn BinaryFunction)
(range DensityFn FunctionQuantity)
```

`DensityFn` is a `BinaryFunction` that takes an instance of a `MassMeasure` and a `VolumeMeasure`, respectively, as its first and second arguments. In

```
(domainSubclass typicalPart 1 Object)
(domainSubclass typicalPart 2 Object)
(instance typicalPart BinaryPredicate)
```

the first and second arguments to the `typicalPart` relation are of the class `Object` or one of its subclasses.

THF Translator Below by *SUMO objects (SOs)* we mean arbitrary SUMO classes and instances.

Our translator maps all SUMO objects to sets in HO TG set theory. The subclass relation is translated as inclusion and the instance relation as membership. SOs that are potentially large such as abstract, mathematical and related SOs thus become sets that may live in higher TG universes.

SOs can be applied to other SOs and variables, creating terms and formulas. Such SOs will be sets that encode relations and functions. Their application to other SOs is the corresponding application of the set theoretical functional and relational sets to other sets. To handle variable arities and row variables, arguments are always appended together into lists.

SUMO quantifiers and logical connectives are mapped directly to their FOL counterparts. Applications that are at predicate positions in formulas are casted by a special *bp* predicate into propositions.

To illustrate a significant higher order construct in SUMO, consider the following problem that uses an axiom with `KappaFn`, which defines a class on the fly, without the need to reify it.

```
(<=>
    (totalFacilityTypeInArea ?AREA ?TYPE ?COUNT)
    (cardinality
        (KappaFn ?ITEM
            (and
```

```
                (instance ?ITEM ?TYPE)
                (located ?ITEM ?AREA))) ?COUNT))

(instance DejvickaStation TrainStation)
(located DejvickaStation PragueCzechRepublic)
(instance HradCanskaStation TrainStation)
(located HradCanskaStation PragueCzechRepublic)

Q: (totalFacilityTypeInArea ?AREA ?TYPE ?COUNT)
A: [?AREA=PragueCzechRepublic,?TYPE=TrainStation,?COUNT=2]
```

The first axiom states that for the ternary relation of `totalFacilityTypeInArea`, which related an area, a class of `Object` and a count of those objects within that area, it is equivalent to the cardinality of the instances of the class that are defined to be instances of the same type, and present within a particular `?AREA`.

We should be able to ask what relations are deducable for `totalFacilityTypeInArea` and get the answer that, for this knowledge base, there are two instances of `TrainStation` that are known to be in the `CzechRepublic`.

```
(SLEEPING c= PSYCHOLOGICALPROCESS).
(ASLEEP :e CONSCIOUSNESSATTRIBUTE).
((bp (ATTRIBUTE (cons v_AGENT (cons ASLEEP nil))))
\/ (bp (ATTRIBUTE (cons v_AGENT (cons AWAKE nil))))
-> (bp (ATTRIBUTE (cons v_AGENT (cons LIVING nil))))).
```

are the translations of:

```
(subclass Sleeping PsychologicalProcess)
(instance Asleep ConsciousnessAttribute)
(=>
    (or
        (attribute ?AGENT Asleep)
        (attribute ?AGENT Awake))
    (attribute ?AGENT Living))
```