SetLog: A Language for Knowledge Representation

Peter Cheeseman[1], Nick Candau[2], Doug Cutrell[3]
9/18/2025

Abstract: SetLog (SL) has been developed as a logic-based language for AI that is both expressive and efficient. SL uses sets and numbers as primitive constructs, allowing quantitative as well as qualitative reasoning. SL represents all information as logical axioms organized into knowledge bases that can represent arbitrarily complex systems, including AI itself. Efficient inference is achieved by a compiler that automatically combines logical specifications in SL with procedural knowledge into executable code. The SL language uses a novel typing system that allows complex logical constraints, including constraints between arguments. This typing system allows function overloading with multiple inheritance. Treating function names as first order entities allows SL to represent and reason in higher-order logic. It can naturally represent information which is structured, probabilistic, temporal, and/or incomplete. While SL is based on higher order predicate calculus (PC), it extends this logic in ways that go well beyond standard logic. Many other aspects of knowledge representation in SL are discussed in this paper, showing its power for knowledge representation and reasoning for AI.

[1] TuringEval LLC, cheeseman.peter@gmail.com
[2] TuringEval LLC, nick.candau@turingeval.com
[3] TurinigEval LLC, doug@turingeval.com

Contents

1 Introduction

Any artificial general intelligence (AGI) system must be capable of representing and utilizing knowledge. This paper introduces SetLog (SL), a knowledge representation language specifically designed for AGI. SL integrates logic and probability to create a transparent and inspectable foundation for reasoning. In contrast to SL, large language models (LLMs) encode knowledge in dense vector embeddings that are nearly impossible to interpret. SL employs a symbolic representation, enabling direct examination, verification, and manipulation of knowledge. The distributed and opaque character of LLM embeddings obscures their underlying semantic content, thereby impeding the construction of explicit inference chains. Unlike symbolic representations, which permit the derivation of transparent reasoning steps subject to mechanical verification, embeddings lack a formally interpretable structure. As a result, their use precludes rigorous validation of inferential processes and undermines the reproducibility of logical justification. As a result, LLMs are unable to represent internal knowledge in a form that supports self-improvement through introspection. Furthermore, LLMs can occasionally return incorrect or fabricated information – a phenomenon known as "hallucination" – which undermines the reliability of their outputs (Vectara, 2025).

SL is a typed, higher-order logic extended with sets and numbers as primitive constructs, and with built-in functions such as equality and addition. It is more expressive than classical predicate calculus (PC), supporting reasoning about probability, vagueness, and time, domains that PC struggles to handle effectively.

It is a commonly held belief in logic-based AI that increased expressivity comes at the cost of computational efficiency. SL avoids this tradeoff through a compiler that enables efficient inference while allowing SL to be fully expressive. The compiler transforms logical definitions (i.e., declarative knowledge) into executable procedures, producing code whose output is equivalent to the corresponding logical inference. By clearly separating logic from control knowledge, SL avoids the pitfalls of earlier logic programming languages like Prolog and Bach, which entangled representation and execution (Kowalski, 1990; Lloyd, 2007).

SL is a function-oriented language in which functions are first-class entities, and predicates are just Boolean-valued functions. Functions use a typing system that allows complex constraints on their arguments, as well as overloading with multiple inheritance. Structured data can be represented in a logically coherent manner. SL is designed to represent *any* system, including the ability to represent and reason about itself.

To simplify inference, SL eliminates explicit quantifiers: universal quantification is replaced by universal variables, and existential quantification is handled via two distinct mechanisms: unique existence is captured using Skolem functions, while general existence is represented with sets.

In sum, SL represents a significant advancement over traditional predicate calculus in AI. Its enhanced expressivity, integration of probability, and computational efficiency make it

a powerful tool for AGI. By incorporating numbers and sets as primitives, SL enables direct reasoning about quantitative information, which is often cumbersome or inexpressible in standard logic. SL supports probabilistic reasoning in a logically sound framework, unlike predicate calculus, which cannot represent degrees of belief. SL offers a unified, general solution to this problem.

This paper reflects the current state of SetLog as of publication. As development is ongoing, future enhancements are expected.

2 SetLog System Architecture

Figure 1 below provides a schematic overview of a knowledge representation and reasoning (KRR) system based on SetLog (SL). As depicted, SL exists in two distinct forms: input SL and internal SL. Input SL is designed for user readability, supporting a flexible syntax that includes mixed infix and prefix notation, synonyms for common function names, and function precedence rules to reduce the need for parentheses. In contrast, internal SL is optimized for computational use, employing prefix notation to reduce the need for parentheses. Internal SL uses the unique name convention, ensuring each function and entity has a single, unambiguous identifier. For readability, all SL examples in this paper are presented in input SL, though the *formal* syntax described in this paper is for internal SL (See Section 11).

A translator converts input SL into internal SL, identifying any syntax errors during the process. Once translated, SL expressions or axioms are reduced to a designated normal form and validated against the existing knowledge in the target knowledge base (KB). If errors are found – either during translation or during the subsequent knowledge validation step (referred to as the "SL Checker" in Figure 1) – an appropriate error message is returned. To enhance readability, a reverse translator can convert internal SL back into input SL. Since the translation from input to internal SL omits non-essential details, such as specific synonym choices and argument ordering, the reverse translation yields a logically equivalent expression that may differ from the original input in form. This dual representation, made possible by bidirectional translation, supports both human readability and computational tractability while preserving logical equivalence. Because of this bidirectionality, the distinction between input SL and internal SL is ultimately cosmetic: both represent information at the same “logic-level,” regardless of syntactic variation. This logic-level is distinct from the procedural-level, as indicated schematically by the “executable code” box in Figure 1.

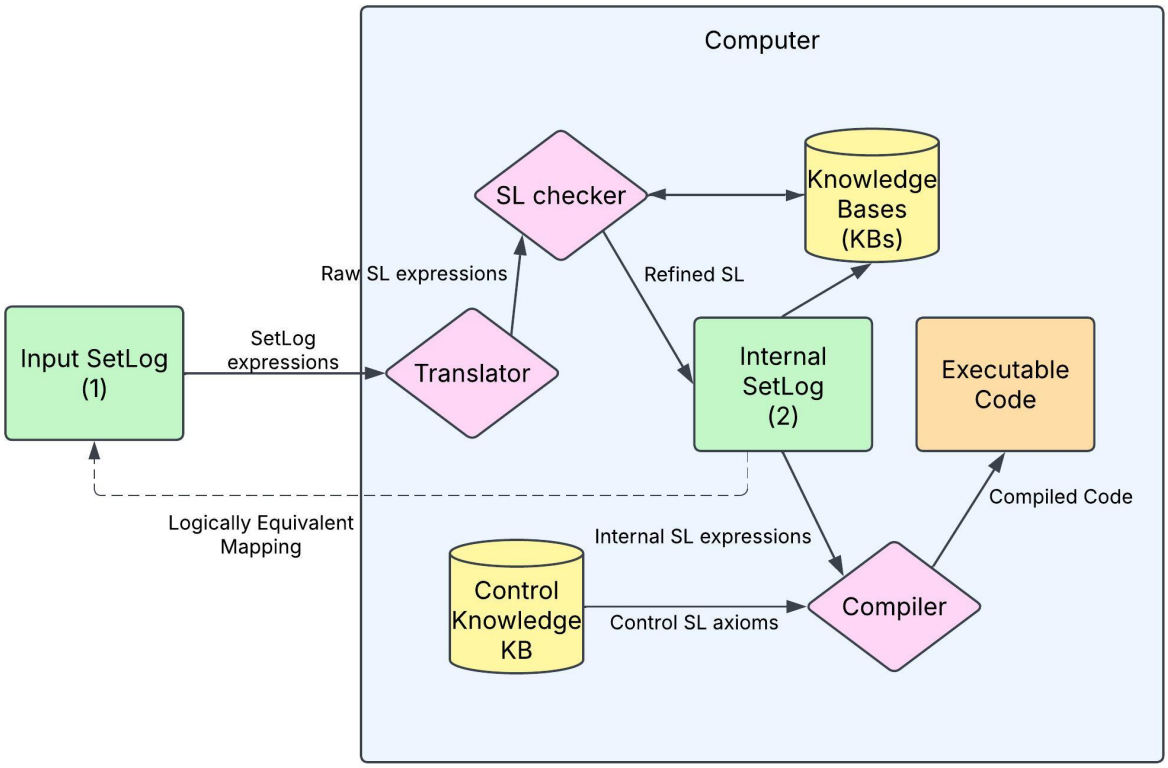

Figure 1
The two forms of SetLog (green boxes): User Input SetLog is translated into Internal SetLog, validated against Knowledge Bases, then compiled into executable logical code.

As shown in Figure 1, the KRR system incorporates a compiler that transforms SL logical expressions into executable code by combining them with specific control knowledge. Executing this compiled code with given inputs produces the same results as would be obtained through logical inference at the logic-level – that is, by using a theorem prover to apply inference rules to answer a query. By replacing general-purpose theorem proving with specialized compiled code, the system achieves computational efficiency beyond what current theorem provers can offer. The separation of declarative knowledge in SL from its executable form allows logical operations – such as simplification, equivalence checking, and normalization – to occur entirely at the logic-level, independently of computational mechanisms. In contrast, Prolog integrates logical and procedural elements within its Horn clause framework. In Prolog, changing the order of clauses can dramatically affect execution, even though such re-orderings are logically equivalent. This intermingling of logic and control in Prolog and similar logic programming languages hampers the ability to reason about logic independently of computation – a limitation that SL avoids.

3 Informal Introduction to SetLog

SetLog (SL) is a typed, higher-order, function-based logic that treats sets and numbers as primitive entities and includes equality, arithmetic, and membership as primitive functions. Unlike traditional logics, SL expressions do not use explicit quantifiers, instead they embed quantified information in the syntax. In SL, information in a knowledge base (KB) is represented as a conjunction of Boolean expressions, termed axioms. Boolean expressions in a KB are not distinguished as being either axioms or theorems but instead are all described as "axioms". SetLog is a prefix language, although common infix operators are allowed in input SL to improve readability.

As an example, consider the following simple relational axiom:

English: "The father of Mary is Bill," or "Bill is Mary's father."
Predicate Calculus: Father(Mary, Bill) – a relation between two individuals.
SetLog: `FatherP[Mary, Bill]`

In SL, the suffix "`P`" in a function name (e.g., `FatherP`) denotes a predicate, a convention that enhances readability for human users without altering the underlying logic. However, this suffix is not universally applied to common predicates – for instance, `And` and `AndP` refer to the same logical function. For clarity, SL expressions in this paper are all presented in `Courier` font.

SL supports bidirectional conversion between relations and functions, as demonstrated in the next example:

English: "The father of Mary is Bill," or "Bill is Mary's father."
Predicate Calculus: Father(Mary, Bill)
SetLog: `FatherOf[Mary] = Bill;`

The above examples illustrate two equivalent SL representations of the same fact: `FatherP[Mary, Bill]` and `FatherOf[Mary] = Bill`. This equivalence holds here due to the implicit assumption that each person has exactly one father. Conversely, if `Bill` is the input and `Mary` the output, the same relationship can be expressed as `ChildOf[Bill] = Mary`. However, if Bill has multiple children, this functional representation becomes more complex, as detailed in Section 10.3. This is an example of a broader pattern where relations can be turned into function calls by picking specific argument(s) to be the output of the expression, in this case Bill, with the rest being arguments to the function call.

Input SL allows closed sets to be defined explicitly using curly braces to enclose the member list. This input syntax is translated into a logical form using the primitive membership predicate `MembP`. For example, the input SL axiom

```
Players$ = {RedPlayer, BluePlayer};
```

is logically equivalent to:

```
MembP[x, Players$] <==> Or[x = RedPlayer, x = BluePlayer];
```

This exemplifies an *extensional definition*, explicitly enumerating all set members. Alternatively, sets can be defined *intensionally*, which is necessary for unbounded sets:

```
MembP[n, EvenNumbers$] <==> (Mod[n, 2] = 0);
```

In the above example, n is a member of the set of even numbers (`EvenNumbers$)` if and only if the modulus with base 2 applied to n equals 0. `Mod` is a primitive arithmetic function (it is not defined in terms of other functions).

English: Everybody loves somebody.
PC: ∀x∃y Loves(x, y)
SetLog: (1) `PersonP[x] ==> (Size(LovedBy$[x]) > 0);`
(2) `MembP[y, LovedBy$[x] <==> Loves[x, y];`

The predicate calculus (PC) formulation, ∀x ∃y Loves(x, y), asserts that for every individual *x*, there exists at least one individual *y* such that *x* loves *y*.

In the SetLog representation, *x* is constrained to be a person using the predicate `PersonP[x]`. Since SetLog does not include explicit quantifiers, variables like *x* and *y* are implicitly treated as universally quantified. Existential quantification is expressed indirectly: here, by constructing a parameterized set (a set-valued function): `LovedBy$[x]` that contains all *y* such that `Loves[x, y]` is true, and then asserting that this set has at least one member using the `Size(...) > 0` axiom. The definition of the set is specified separately as an axiom using the `Memb` relation in axiom (2), which defines set membership in terms of the underlying predicate.

Set-valued functions such as `LovedBy$[x]` are a central feature of SetLog. They can also be referred to as parameterized sets, because their contents vary as a function of the parameter (in this case, *x*), enabling context-dependent representations of relationships while preserving logical modularity and readability.

4 SetLog Overview

4.1 Language Representation

In SetLog (SL), all knowledge is stored as logical assertions organized into Knowledge Bases (KBs). These assertions can be expressed in two distinct syntactic forms, as illustrated in Figure 1. The first, *internal SL*, is a representation designed for computer use that exclusively uses prefix notation and enforces unique function and entity names. The second, *input SL*, is designed for human readability, offering flexibility through features such as synonyms, default values, mixed infix and prefix notation with defined operator precedence, parentheses for disambiguation, and support for comments and macro expansions from shorthand forms.

Input SL prioritizes flexibility and extensibility. At the current stage of SL development, its sole requirement is unambiguous translation into internal SL. A bidirectional translator enables conversion between these formats, ensuring that logical content remains identical across both representations, with differences confined to alternative syntax (see Figure 1). Consequently, discussions of logic-level operation treat the two forms interchangeably. For readability, this paper uses input SL, while the formal SL syntax specification pertains exclusively to internal SL (see Section 11).

4.1.1 Naming conventions

SL identifiers are strings over letters, digits, and _, with an optional terminal suffix $ or $$; no other punctuation or whitespace is permitted, and $ may appear only as a terminal suffix. Variables begin with a lowercase letter. Constants and function symbols begin with an uppercase letter. Predicates are Boolean-valued functions and end with a capital P. A terminal $ designates a set-valued symbol (including set-valued functions), and $$ designates multiset-valued symbol; absence of a suffix denotes a non-collection value, and predicates never take $ or $$. For example:

```
Colors$ = {Red, Green, Yellow};
```

defines a set named `Colors$`, and

```
CarModels$[make]
```

denotes a set-valued function returning a set of car models corresponding to the variable `make`.

In internal SL, names are unique, ensuring each entity has a single identifier. Input SL relaxes this constraint, allowing synonyms for the same entity. During translation to internal SL, these synonyms are resolved to a single, unique identifier.

4.1.2 Comments

In input SL, the # symbol denotes a comment, indicating that all subsequent text on the same line is ignored by the translator. For example:

```
# Oceans of the world
Oceans$ = {Atlantic, Pacific, Indian, Arctic, Southern};
```

4.2 Functions

SL is a function-based language where functions themselves are entities, enabling quantification over them, such as “all binary functions”. A function call consists of a function and a specific set of arguments (e.g. `FatherP[Mary, Bill]`). A function is constrained by type-axioms (see Section 7) that define permissible argument types and the type of a successful result of a function call. In input SL, certain primitive functions may use infix notation with symbolic operators, such as +, /, =, etc, but these are converted to prefix form in internal SL (see Section 6.6).

4.3 Axioms and KBs

In SL, knowledge is represented as a collection of axioms. An axiom is a Boolean well-formed formula (w.f.f.) that is assumed to hold within a specific knowledge base (KB) (see Section 12). A false statement is represented by equating the corresponding predicate to False. The equality ensures that the overall axiom is True. When constructing a KB, axioms are written in input files and separated by semicolons. In contrast to many other logics, SL axioms can represent both logical and probabilistic information. For example, the ground axiom `(SizeOf[smiths$] = 4)` expresses the information that the Smith family has 4 members – a logical assertion. If the size of this family is only known to be approximately 4, this information can be expressed in SL with the meta probability axiom:

```
Prob[SizeOf[smiths$],Evidence, Assumptions] = Beta[a, b];
```

where `a` and `b` are parameters that can be expressed as functions of the mean and variance of the Beta distribution (see Section 14). This probability axiom is meta because it is an axiom about another axiom (SizeOf).

Here are other examples of axioms in SL:

```
## axiom defining a set of constants:
Colors$ = {Red, Blue, Green};

## axiom defining a predicate function:
ColorP[x] <==> MembP[x, Colors$];

## example type-axiom:
And[StateP[x], StateP[y]] <==> BoolP[BorderingP[x, y]];

## axiom defining an integration rule:
Integrate[Sin[x], x] = -Cos[x];
```

where `x` and `y` are universal variables (see Section 4.7 and Section 11.3).

4.4 Entities

Entities in SL include constants, events, sets, and functions. Having functions as entities in SL means that they can be quantified over, making SL a higher-order logic.
A *structured entity* is a sequence with a paired recognition predicate which places type constraints on its fields. The first entry is always a *label*. (For clarity, in this paper these labels will be italicized.) For example:

`<`*`Point2D`*`, 1, -2>` or `<`*`Mass`*`, 1.2, Kg>`

The successive entries may themselves be structured entities. For example:

`<`*`LineSegment`*`, <`*`Point2D`*`, 0, 0>, <`*`Point2D`*`, 1.5, -2> >`

describes a two–dimensional oriented line segment from the origin to the point {1.5, –2}. It is a structured entity that contains two nested structured entities. The associated predicates would be named `LineSegmentP` and `Point2DP`. Further details are provided in Section 5.3.

4.5 Sets and Multisets in SetLog

Sets in SL are defined by their membership function. For instance:

Example 4.5.1

```
MembP[color, PrimaryColors$] <==>
     Or[(color = Red), (color = Yellow), (color = Blue)];
```

Here, `color` is a universal variable, and `PrimaryColors$` is a closed set. Input SL provides a shorthand for closed extensional sets:

```
PrimaryColors$ = {Red, Yellow, Blue};
```

This is expanded by the translator into the membership–based form above.

Multisets are represented similarly. In input SL, a closed multiset name uses the `$$` suffix and is enclosed in square brackets with curly braces, for example:

```
PrimaryColors$$ = {[Red, Yellow, Red, Blue]};
```

The `$$` convention distinguishes multisets from sets, and `{[...]}` is an input short–hand syntax for a multiset. Additional details are provided in Section 8.

4.6 Sequences in SetLog

In internal SL, sequences are represented as functions mapping indices to corresponding values. In input SL, sequences can be defined using angle brackets:

```
LedZeppelinIV = <"Black Dog", "Rock and Roll", "The Battle of
Evermore", "Stairway to Heaven">;
```

The corresponding translation into internal SL defines a sequence function named `LedZeppelinIV`, which takes a single natural-number argument:

```
LedZeppelinIV[1] = "Black Dog";
LedZeppelinIV[2] = "Rock and Roll";
LedZeppelinIV[3] = "The Battle of Evermore";
LedZeppelinIV[4] = "Stairway to Heaven";
## Type-axiom
And[IntegerP[indx],indx >= 1,indx <= 4] ==>
     SequenceP[LedZeppelinIV[indx]];
```

A sequence function takes a natural number as its argument (its index). It can have fixed size (i.e. a specified index range) or an indefinite size. A sequence of size one is also allowed (See Section 9). An example of treating a sequence as an entity is:

```
LengthOf[LedZeppelinIV] = 4
```

This syntax is valid in both internal and input SL, and passes the sequence as the argument to the `LengthOf` function.

4.7 Universal Variables

Predicate calculus allows free variables, but SL has no use for them. This is because all knowledge in SL is in the form of KBs, and KBs are fully interpreted. Since the interpretation process assigns values to free variables, fully interpreted KBs do not contain free variables.

In SetLog, universal variables are denoted by uncapitalized alphanumeric strings. SL syntax also allows universal variables to be subscripted, using notation such as: `angles[2],cars[5]`. They correspond to universally quantified variables in first-order logic (see Section 10.2). A universal variable does not refer to any specific value; instead, when they appear in a general axiom that axiom must be true for all possible substitutions of the universal variable. Universal variables in general axioms are scoped to each such axiom, so the same universal variable name can be used in multiple axioms without interference.

Consider the example from Section 4.5:

```
MembP[color, PrimaryColors$] <==> Or[(color = Red),
     (color = Yellow), (color = Blue)];
```

Here, `color` is a universal variable. This axiom asserts that for any value substituted for `color` is a member of `PrimaryColors$` if and only if it is equal to `Red`, `Yellow`, or `Blue`. The formula is thus false for any other substitution, and true only for those three specific values.

4.8 Quantification

A major difference between SL and predicate calculus is in how quantification is represented. Universal variables in SL are implicitly universally quantified – i.e. unlike predicate calculus, the ∀ symbol does not appear in SL. Existential quantification in SL is more complex because SL uses two types of existential quantification. SL separates existence into general or non-unique existence, and unique existence, corresponding to the use of "a" or "the" in English. In the case of unique existence, SL introduces a (Skolem) function, as shown in the following example:

English: Everybody has one and only one father.
FOL: ∀x(Person(x) ==> ∃!yFather(x,y)) #∃! Here denotes unique existence
FOL (without ∃!): ∀x(Person(x) ==> (∃yFather(x,y) ∧ ∀z(Father(x,z) ==> (y = z)))
SL: `PersonP[x] ==> MalePersonP[FatherOf[x]];`

The function `FatherOf[x]` used here returns the unique person who is the father of `x`.

For nonunique existence SL requires two axioms to specify. For example:

English: John has at least one brother.
FOL: ∃x (Brother(x, John))
SL:
```
MembP[x, BrothersOf$[John]] <==>
        And[MaleP[x], (ParentsOf$[x] = ParentsOf$[John]];
```

```
SizeOf[BrothersOf$[John]] > 0;
```

The first axiom defines the set-valued function: `BrothersOf$[John]`, and the second axiom states that there is at least one member of this set. This two-axiom representation captures the concept of general (non-unique) existence (see Section 10 for more details).

5 Entities

Entities in SetLog (SL) are *symbols that evaluate to themselves*. They include constants, sets, numbers, and functions (referring to the function itself, not its application in a function call). Within a Knowledge Base (KB), entities can be categorized into types, such as places, colors, dates, people, or events. Logically, the presence of a symbolic entity in a KB implies its existence, and the set of all such entities constitutes the KB's *domain of discourse*.

5.1 Entity Types

In SL, an entity can have multiple types. For example, "Bill" could be of type: male, alive and a person. For example, if it is known that Bill is a president of the USA, then it is sufficient to state this fact, and rely on general knowledge to infer that he must also be a person. Generally, in a set-subset hierarchy, the membership of an entity should be given for the lowest level of this hierarchy, if known, for maximum informativeness. An entity must have some "type", otherwise nothing is known about it, and its appearance in a KB doesn't change anything.

SL easily supports the notion of subtyping via axioms such as:

```
CelicaP[x] ==> ToyotaP[x];
```

Or alternatively:

```
Toyota$ ⊃ Celica$  ## Celicas are a subset of Toyotas
```

Whole hierarchies can be efficiently represented, for example:

```
IntegerP[x] ==> RealP[x] ==> ComplexP[x] ==> NumberP[x];
```

There are many more types which could be added to this number hierarchy to form a directed acyclic graph, e.g. even numbers, positive reals, etc.

The type hierarchy can implement *multiple inheritance*. For example, if `PosRealP` is a type describing any positive real number and `PositiveIntegerP` describes positive integers only, then `PositiveIntegerP` is a subtype of both `IntegerP` and `PosRealP`. However, neither of these are subtypes of the other.

```
PosIntegerP[x] ==> IntegerP[x];
PosIngegerP[x] ==> PosRealP[x];
```

Alternatively, these can be combined:

```
PosIntegerP[x] ==> And[IntegerP[x], PosRealP[x]];
```

Syntactically, entities within a KB can be classified into three kinds: elementary entities, structured entities, and virtual entities, as described below.

5.2 Elementary Entities

Elementary entities in internal SL are denoted by a *single* symbol or name. This enforces the unique name assumption, which posits a one-to-one correspondence between symbols and entities. Elementary entities can include: named individuals, Boolean constants (True, False), sets, numbers, functions, and sequences.

The unique name convention raises a complicated issue about entity identity. One criterion for identity, described by Leibniz, is that two entities are the same if and only if they have the same properties and relationships – i.e. they are indistinguishable. While this works well in static domains, it can fail in dynamic domains. This is because new information may make two previously indistinguishable entities distinguishable, and thus they should have different names.

5.3 Structured Entities

A structured entity is a sequence whose first entry is a label. It is an entity because sequences are functions, and all functions in SL are entities. For example, in the sequence `<`*`Point2D`*`, 3, 1>`, *`Point2D`* is the label, and the remaining arguments distinguish the particular entity. (For clarity, structured entity labels will be italicized in this paper.) This "structure" is defined by a Boolean-valued function `Point2DP`, which returns true only if the first entry in the sequence corresponds to the tag, and if the subsequent entries conform to type constraints for that structured entity. For example, in the previous example, the entries after the first are required to be integers. This `Point2DP` predicate represents the "type" of the structured entity.

Recognition predicates implement type constraints for a structured entity's fields. For example, 2D points restricted to integer coordinates can be defined by:

```
## Point2DP predicate definition
Point2DP[p] <==> And[SeqP[p], SizeOf[p]=3, p[1]=Point2D,
     IntegerP[p[2]],IntegerP[p[3]];

## Type-axiom for Point2DP
BoolP[Point2DP[p]];
```

In summary, the above defines a structured entity, such as `<`*`Point2D`*`, 1, −1>`, and a predicate `Point2DP` to identify such entities. The type-axiom for `Point2DP` implies that it is a total function, defined for all entities.

Functions operating on structured entities can be defined, such as:

```
## VectorPointAdd function
VectorPointAdd[<Point2D, i, j>, <Point2D, l, m>] =
     <Point2D, i+l, j+m >;
```

```
## Type-axiom for VectorPointAdd
And[Point2DP[a],Point2DP[b]] ==> Point2DP[VectorPointAdd[a,b]];
```

Structured entities support hierarchical nesting of arbitrary depth, as their arguments can themselves be structured entities. For example, a line segment defined by two points. This enables representations like:

```
LineSeg1 = <LineSegment, <Point2D, 1, 2>, <Point2D, −4, 0> >;
```

The `Point2D` structured entity not only identifies a unique point but also embeds its coordinate information in its representation. The choice between sequence or elementary entity representations is a matter of convenience, as inter–conversion between them is usually possible, as the following example shows: `<Point2D, 1, 2>` could become `Point12`, with added axioms:

```
XCoordinateOf[Point12] = 1;
YCoordinateOf[Point12] = 2;
```

And the type definition for the entity `Point12`:

```
Point2DP[Point12] = True;
```

The `VectorPointAdd` function for 2D points would then be redefined as:

```
(VectorPointAdd[p, q] = r) <==> And[
     XCoordinateOf[r] = XCoordinateOf[p] + XCoordinateOf[q],
     YCoordinateOf[r] = YCoordinateOf[p] + YCoordinateOf[q]
]
```

retaining the same type-axiom:

```
And[Point2DP[a],Point2DP[b]] ==> Point2DP[VectorPointAdd[a,b]].
```

It is possible to allow arbitrary arity in structured entities, for example, here is the definition for a generic arity integer *`NPoint`* structured entity (using the `NPointP` predicate).

```
## <Npoint, n, seq> -- where seq is a sequence of length n
## NPointP predicate definition:
NPointP[p] <==> And[SeqP[p],LengthOf[p]=3,p[1]=NPoint,
     PosIntegerP[p[2]],SeqP[p[3]],LengthOf[p[3]]=n,
     IntegerP[p[3][i]],(1 ≤ i ≤ n)];
## Type-axiom for NPointP
BoolP[NPointP[p]];
```

The definition axiom recognizes *`NPoint`* entities as having a first argument 'n' which gives the length of the sequence, held in the second argument. `IntegerP[Part[p[3],i]` indexes twice, getting the sequence argument as the second argument of `p` and then indexing into the sequence by the universal variable `i`, constrained to `i=1` to `n`.

Since structured entities can have other structured entities as components to arbitrary depth, this allows representation of structures such as trees, graphs, etc.

5.4 Virtual Entities

In first-order logic (FOL), every entity is represented explicitly by a unique symbol, which works well for small entity sets. However, in scenarios with large domains–such as representing airplane location in a 3D voxel grid–explicitly naming every voxel, most of which are unoccupied, is space inefficient. Virtual entities address this by defining a domain implicitly, and only naming specific entities explicitly as needed. For example:

```
(OccupiedP[ <Vox,i,j,k> ] = False) <==>
     Or[Not[And[i=200,j=512,k=1034]],
        Not[And[i=730,j=417,k=321]]]
```

This axiom implies that voxels <*Vox*,200,512,1034> and <*Vox*,730,417,321> are occupied, while all others in the range are not, without requiring individual named entities for each voxel. The implicit voxels are an example of virtual entities.

Virtual entities can be made explicit by adding axioms like:

```
OccupiedP[ <Vox,200,512,1034> ] = True;
```

Here the structured entity: <*Vox*,200,512,1034> is made explicit.

Counting all voxels in a domain, virtual or explicit, can be performed without generating explicit representations, meaning entity existence is independent of their virtual status. Only structured entities can be virtual, as they rely on replacing specific arguments with variables constrained to defined ranges. Entities, virtual or otherwise, are logically members of the domain of discourse, even if they are not represented explicitly.

6 Functions

Functions are the foundational element of SetLog (SL). A simple form for a function definition is:

```
FunctionName[arg] = Value;
```

Function calls use square bracket syntax and, by convention, the function name is capitalized. For example, the function call: `StateCapitalP[Texas, Austin]` evaluates to `True`. In SL, predicates are simply functions that return a Boolean value. This contrasts with first-order logic (FOL), where predicates (returning `True` or `False`) are syntactically and logically different from functions that yield entities as their value, e.g., `MotherOf[Bill] = Mary`. In SL, `True` and `False` are treated as entities in every domain of discourse. In internal SL, predicate axioms have the form: `Pred1[arg1] = True` or `Pred1[arg2] = False`. Input SL, on the other hand, allows the alternative: `Pred1[arg1]` (implying `True`) or `Not[Pred1[arg2]]`(implying `False`). For arithmetic,

internal SL uses `Add[{2, 3, 4}]`, for example, while input SL permits infix notation: `2 + 3 + 4`.

In SL, function call axioms are intrinsically logical because they are always equal to their value. When they are evaluated (a procedural operation), they then become computing constructs with a temporal order of evaluation. In lambda calculus and logic programming, functions are assumed to be procedural constructs with a definite order of evaluation, and in this case, the corresponding type theory is also procedural. When a SL function is compiled, the resulting code is procedural, and specific input–output behavior results from such code execution.

6.1 Functions at the Logic and Procedural Levels

For internal SL axioms, the only syntactic construct is a function call – even predicates are regarded as just another type of function (with a Boolean value). The representation of a function at the logic level has no inherent directionality or temporal evaluation ordering, but instead just expresses a relationship between the arguments of a function and its value given these arguments. When a function call is evaluated, there is a clear distinction between inputs (the arguments) and the output (the value), with the output only being generated once the evaluation is complete. In this case, a function is being given a procedural interpretation. In SL at the logic level, functions do not have a directionality, but when a corresponding compiled function is executed (the procedural level), there is a clear directionality. This means that whether a function is considered to be a logical or procedural construct depends on which level the function is being considered (the logic level or the procedural level).

6.2 Interconversion Between Functions and Relations

All functions can be defined in terms of a corresponding relation. For example, from the relation: `ChildMotherP[child, mother]` two different functions can be defined: `MothersOf$[child]`, and `ChildrenOf$[mother]`, by choosing one argument as the input and the other as the output. The value of a function defined this way is always a set. In the case of the `ChildrenOf$[Mary]` the resultant set is the set of all persons for which the relation: `ChildMotherP[child, Mary]` is true. Depending on the identity of `Mary`, the function: `ChildrenOf$[Mary]` represents a set of people, possibly the null set. In the case of the `MothersOf$` function, the genealogy KB contains an additional axiom that states that all persons have one and only one mother, so the resultant set is always a singleton set. In this special case the additional knowledge from the KB allows a new function to be defined:

```
MotherOf[child] = Get1[MothersOf$[child]]; ##a person, not a set
```

In this example, `Get1` is a primitive function that takes a singleton set as its argument and returns the sole member of the set.

The process of defining new functions from relations can be applied to relations with more than two arguments. For example, the relation `BetweenP[x, y, z]` can be used to

define six functions depending on which arguments are selected for inputs with the remaining arguments defining a set-valued output. For example, the function `InBetween$[x, z]` defines the set of all entities that are between x and z. Another function that could be defined from the `BetweenP` relation is `PairsBeyond$[x]`. This function takes an entity as input (the start entity) and returns the set of all ordered pairs that occur after the start entity. Although many functions can be defined from an n–ary relation, only those that are useful are typically defined. In SL it is possible to define functions directly without defining them from the corresponding relation. In such cases, it is possible to define the corresponding *relation* from the defined *function*, if it is useful to do so. In other words, whether relational or functional syntax is used is a matter of convenience, since it is always possible to inter–convert between these forms.

6.3 Function Definitions

Functions are typically defined by one or more axioms, accompanied by a corresponding type-axiom (see Section 7). An example of a function definition with a single axiom is:

```
MembP[x, ParentsOf$[child]] <==>
     Or[(x = FatherOf[child]), (x = MotherOf[child]];
```

Here, `ParentsOf$` is a set-valued function (indicated by the `$` suffix), that is defined using other functions: `FatherOf` and `MotherOf`.

Some functions require multiple axioms to define them. Consider `RowNumber`, which assigns a row number to a tic–tac–toe square (represented by integers 1–9):

```
(RowNumber[squareID] = 1) <==> And[squareID > 0,squareID < 4];
(RowNumber[squareID] = 2) <==> And[squareID > 3,squareID < 7];
(RowNumber[squareID] = 3) <==> And[squareID > 6,squareID < 10];
```

A type-axiom for RowNumber constrains `squareID` to be an integer in the range 1–9.

6.3.1 Recursively defined functions

Recursively defined functions are allowed in SL, as in the following two–axiom definition of a factorial:

```
(Factorial[n] = n * Factorial[n-1]) <==> (n > 1);
Factorial[1] = 1;
## Type-axiom for Factorial:
NatNumP[n] ==> NatNumP[Factorial[n]];
```

The type-axiom specifies that the argument must be a natural number, and the return value will be a natural number.

6.3.2 Constraint defined Functions.

Functions without explicit defining axioms can be specified via constraints. For example, in map coloring, the constraint that bordering countries must have distinct colors in SL is:

```
BordersP[x, y] ==> (ColorOf[x] != ColorOf[y]);
```

This axiom constrains the values of the `ColorOf` function without fully specifying them. The set of all function calls without explicit function definitions in a KB, form the constraint set for the corresponding undefined function. A constraint set of mutually constrained functions form a constraint satisfaction problem (CSP). An inconsistent CSP indicates an inconsistent KB, a unique solution fully determines each the value of the constraint defined function(s), and multiple solutions leave the constraint defined function(s) underdetermined. See Section 17.2.1 for a detailed example.

6.4 Sequence Functions and Sequence Arguments

Sequences in SL are functions that have a single number argument (its index) and a specified value at that position. In input SL, they can be written using an angle bracket notation:

```
PrimesLessThan10 = <2, 3, 5, 7>;
```

For open (unbounded) sequences, the functional notation can be used. For example, the Fibonacci sequence can be defined by the axioms:

```
Fib[1] = 1; Fib[2] = 1; (n > 2) ==> Fib[n] = Fib[n-1]+Fib[n-2];
```

Unlike sets, sequences preserve order and allow duplicate values in different positions. They may be fixed or indefinite length, like the Fibonacci sequence. A fixed–length sequence argument appears in:

```
Implies[Func1P, Func2P];
```

where `Func1P` and `Func2P` are predicates. In input SL, this can also be written as:

```
Implies[<Func1P, Func2P >];
```

or by using the infix operator:

```
Func1P ==> Func2P.
```

In summary, sequences are not a syntactically distinct type of entity, but in SL are just a special type of function with a natural number argument.

In standard logic, all functions and predicates are assumed to have a fixed length sequence of arguments with a definite order. This means that the concept of “order” is built directly into the syntax, although the concept of order is not defined in these systems. In SL, by using natural numbers as the argument for sequence functions, the ordering relationship of the elements of the sequence is derived directly from the ordering of the natural numbers.

6.5 Function Arguments

All SL function calls take a single argument, which may be an entity, set, multiset, or sequence, as dictated by the function’s type-axiom. This “single argument” design is not restrictive, as sets and sequences can contain multiple elements, including an indefinite

number of elements. These elements, in turn, may be sets or sequences, and so on, leading to an arbitrary depth for function arguments.

Functions with sequence arguments are written in internal SL using the syntax as shown in the following example:

```
StateCapitalP[SequenceCap];
SequenceCap[1] = Texas;
SequenceCap[2] = Austin;
```

Input SL offers alternative equivalent syntactic forms:

```
StateCapitalP[<Texas, Austin>];
```

or:

```
## Conventional function notation
StateCapitalP[Texas, Austin];
```

A sequence may have length one, but if a function consistently uses single–element sequences, redefining its argument type as an entity may be more practical.

SL's ability to use sets and sequences as arguments allows functions to effectively handle an arbitrary number of inputs (the elements of a set or sequence), unlike FOL's fixed–arity functions and predicates. For instance, `And` in internal SL takes a single set argument of arbitrary size, enabling flexible application, similar to how the And predicate is used in programming.

6.6 Function Calling

SetLog (SL) employs standard function call syntax. For example:

```
Integrate[Sin[x], x] = -Cos[x];
```

Here, the input SL infix "=" operator is used, and `x` is a universal variable. Type-axioms for `Cos`, `Sin`, and `Integrate` constrain the domain of values assignable to `x` (see Section 7). Given this axiom, evaluating an expression like `Integrate[Sin[theta], theta]` matches to the above axiom, yielding the result: `-Cos[theta]`.

In input SL, binary infix operators (e.g., =, +) are converted to prefix notation in internal SL with arbitrary number of arguments. Here are some common binary infix operators:

```
=              equality
!=             inequality
>              greater than
<              less than
>=             greater than or equal to
<=             less than or equal to
<==>           equivalence
==>            implication
<==            reverse implication (note different than <=)
```

```
+               plus
-               minus
*               multiply
/               divide
^               exponentiate
```

6.7 Primitive Functions

Logically, primitive functions are those not defined in terms of other functions; procedurally, they correspond to functions directly executable by the computer. Examples include arithmetic operations (Add, Mod), logical operators (And, Or, =), and membership tests (MembP). Some important primitive functions are:

`Equivalent[expr1, expr2]` ## infix form: `expr1 <==> expr2`)
`expr1` and `expr2` are logical expressions.

`Implies[expr1, expr2]` ## infix form: `expr1 ==> expr2`)
`expr1` and `expr2` are logical expressions.

`Equals[expr1, expr2]` ## infix form: `expr1 = expr2`)
`expr1` and `expr2` are not type restricted.

`Plus[Set[a, b, ...]]` ## infix form: `a + b +` ...)
`a`, `b`, ... are numeric expressions.

`Times[Set[a, b, ...]]` ## infix from: `a * b *` ...)
`a`, `b`, ... are numeric expressions.

`And[Set[a, b, ...]]` ## input representation: `And[a,b,...]`)
`a`, `b`, ... are logical expressions.

`Or[Set[a, b, ...]]` ## input representation: `Or[a,b,...]`)
`a`, `b`, ... are logical expressions.

`MembP[x, s$]`
`x` is not type restricted, and `s$` is a set.

`SetMap$[f, s$]`
`f` is a function name, and `s$` is a set-typed expression.

Note that in this example, a function is being passed as an argument. This is only possible because SL is a higher order logic.

6.8 Predicates

Predicates are functions returning Boolean values, whose name conventionally ends in P. For example, in input SL:

```
WeekendDayP[day] <==> Or[(day = Saturday),(day = Sunday)];
```

Here, `day` is a universal variable that ranges over the days of the week. Note that the parentheses and spaces are optional in input SL, given the operator precedence order.

6.9 Functions as Entities

Sometimes functions need to be passed as arguments. For example, the `Sort` function takes as an argument a binary Boolean predicate, such as `LessThan`, and a set to be sorted, and returns a sorted sequence. For example:

Evaluate: `Sort[LessThan, {4, 12, -1, 0}]`
Result: `<-1, 0, 4, 12>`

Considering functions as entities allows quantification over them, implying that SL is a higher-order logic.

As another example, the primitive function `SetMap` takes a function name, and applies that function to all members of a Set, returning a set of the results. For example:

```
ClassroomParents$[students$] =
          UnionOf$[SetMap$[ParentsOf$, students$]];
```

Here `ParentsOf$` is a function that takes a single student as its argument and returns a set of parents (a pair). Calling `SetMap$` returns a set of sets, then `UnionOf$` returns a set of the distinct elements of all of these sets. The result is that `ClassroomParents$` is a function that returns a set of all the parents for all the children in the class.

Since functions are entities in SL, they can have properties (e.g. `Numeric`, `Unary`) or relationships between them. For example, `GreaterP[f1, f2] <==> f1[x] > f2[x]` defines a binary relationship between two functions `f1` and `f2`. In addition, new functions can be defined in terms of other functions, for example `h = f + g` or `h = f[g]` (where `f`, `g`, and `h` are functions). More complex examples of functional algebra include convolutions and Fourier transforms. Other properties of functions can include associativity, commutativity, transitivity, and monotonicity.

6.10 Partial Evaluation, Anonymous Functions, and Anonymous Sets

Input SL contains parsimonious language features that are expanded in internal SL. These expansions allow full logical manipulations, while the input SL features assist in writing easy to understand expressions. Among these useful expansions are partial evaluation, anonymous functions, and anonymous sets.

A function can be constructed by partial evaluation of another function. For example, consider the three–argument function: `CarSales[make, model, year]`. The

quantities sold for a given make and model over a range of years could then be specified as:

```
RecentCelicaSales$ = SetMap$[CarSales[Toyota, Celica, @],
     {2020, 2021, 2022, 2023, 2024, 2025}];
```

Here, `CarSales[Toyota, Celica, @]` is an anonymous function of one argument (@) that is the first argument of `SetMap$` above. This single argument anonymous function is formed by fixing the first two arguments of `CarSales[Toyota, Celica, year]`.

An alternative representation would be to define a new function:

```
SalesOfCelica[year] = CarSales[Toyota, Celica, year];
RecentCelicaSales$ = SetMap$[SalesOfCelica,
          {2020, 2021, 2022, 2023, 2024, 2025}];
```

where `year` is a universal variable.

This new function could also be defined using the anonymous function syntax:

```
SalesOfCelica = CarSales[Toyota, Celica, @];
```

In either case, a type-axiom would be required:

```
IntegerP[year] ==> IntegerP[SalesOfCelica[year]];
```

Here is an example representing the composition of two functions:

```
ReciprocalSin = Divide[1, Sin[@]];
```

Here composition is simply nested function calling, as in most computer languages.

The standard way to represent this, without using anonymous function syntax, would be:

```
ReciprocalSin[x] = Divide[1, Sin[x]];
```

Regardless of the way it is defined, the function is called the same way:

Eval: `ReciprocalSin[Pi/6]`
Result: `2`

Anonymous functions with more than one argument are also supported:

```
SalesOfToyota = CarSales[Toyota, @1, @2];

## Type-axiom:
And[CarsP[model], IntegerP[year]] ==>
          IntegerP[SalesOfToyota[model, year]];
```

Here, `SalesOfToyota` is being bound to a function itself, not to the result of calling a function.

6.11 Functions Returning Functions

Functions are first class entities in SL, and so they can be returned as the value of another function. A function returning another function is common in mathematics (eg. h = f∘g). In programming, this is sometimes referred to as a "function factory".

Because SL is a logical language, defining a function that returns a function requires special consideration of the logical meaning of function definition. Schematically, a function definition has one or more axioms of the form:

```
(Func[args] = value) <==> (LOGICAL_EXPRESSION)
```

where the `LOGICAL_EXPRESSION` involves relations on the `args`.

To create a function returning a function, the `LOGICAL_EXPRESSION` can itself be a function definition expression.

As an example, consider the primitive `Power` function:

Evaluate: `Power[2, 3]`
Result: `8`

The Power function can be used to create the `Square`, `Cube`,..., functions. First, consider that a `Square` function defined by:

```
(Square[x] = y) <==> (y = Power[x, 2]);
```

More generally, a `PowerFactory` function can then be defined like this:

```
(PowerFactory[exp] = powFunc) <==>
     ( powFunc[x] = Power[x, exp] );
```

The `PowerFactory` function can then be used like this:

```
SquareFunc = PowerFactory[2];
CubeFunc = PowerFactory[3];
```
Evaluate: `SquareFunc[5]`
Result: `25`

Evaluate: `CubeFunc[4]`
Result: `64`

7 Type-Axioms

The input and output types of a function call are constrained in SL by axioms called *type-axioms*. Every function has at least one type-axiom. A type-axiom is a logical statement that constrains a function definition by specifying constraints on its arguments (typically the argument types), and on the function's return type. Informally, type-axioms specify when it is meaningful to call the function and what gets returned. For example:

```
PersonP[p] <==> PersonP[FatherOf[p]];
```

In this example, `FatherOf` returns a person if and only if its argument is a person.

Multiple type-axioms for the same function may be needed, as in the following example:

```
RealP[x] ==> RealP[Sin[x]];
ComplexP[x] <==> ComplexP[Sin[x]].
```

Note that bidirectionality does not hold for the `RealP` argument type-axiom above, since there are non-real complex numbers whose `Sin` is a real number. That is, the fact that `Sin` produces a real number does not imply that its argument is a real number.

Here, there are two type-axioms for `Sin` that provide useful information. One states that in general, any complex number is an acceptable argument (yielding a complex result). The other states that when the argument is a real number, the result will also be a real number. Multiple type-axioms for the same function support polymorphism (see Section 7.5).

Subtyping axioms (See Section 7.6), such as: `RealP[x] ==> ComplexP[x]` support logical inference associated with these axioms. For example, this implies that

```
RealP[x] ==> ComplexP[Sin[x]]
```

SL type-axioms act like declarations in programming languages, enabling type checking. Type checking is used to verify that the parameters to function calls satisfy the type-axioms for those functions. Type checking for embedded functions involves matching a return type of the embedded function to an argument type of the outer function. For example:

```
Sin[Cos[Pi]]
```

Here `Pi` is of type `RealP`, which matches the argument type for `Cos`. `RealP`, the inferred return type of the `Cos` function for a real argument, matches the argument constraint of the `RealP` argument type-axiom for the `Sin` function. This allows the type-inference:

```
RealP[Sin[Cos[Pi]] = True.
```

Type-axioms imply types for variables. This in turn defines the ranges for universal quantification.

Type checking can occur both at the logical level and at compile time. However, sometimes the information needed for type checks is only available at runtime. In such cases, the compiler pushes the type checking onto the run-time system.

Since SL allows multiple type-axioms for the same function, type checking then involves checking each type-axiom for a given function for a possible match. For example, a polymorphic `SizeOf` function might apply to sets, graphs, plots of land, etc. Separate type-axioms would be provided for each argument type.

When a type-axiom's argument constraints are not met, the return type of the function is undefined. In such cases, the function call returns unevaluated or triggers a runtime error, depending on the implementation.

7.1 Type-axiom Schema

Most type-axioms use a simple schema as shown below:

```
ARGUMENT_PRED <==> RETURN_PRED[FUNCTION[ARGS,…]];
```

Example:

```
And[PersonP[x], PersonP[y]] <==> BoolP[ChildMotherP[x, y]];
## Note the use of <==>
```

Or

```
ARGUMENT_PRED ==> RETURN_PRED[FUNCTION[ARGS,…]];
```

Example:

```
RealP[x] ==> RealP[Sin[x]];
## Note the use of ==>
```

The ARGUMENT_PRED is a predicate, or logical expression, constraining the ARGS, typically using predicates applied each argument separately. However, SL allows complex logic, relationships between the arguments, such as requiring that all the integer argument be relatively prime. These collective argument constraints are not possible in most typed logics.

The RETURN_PRED is a predicate which constrains the return type.

When a type-axiom uses "equivalence" (<==>) as the top-level operator, it states that knowledge of the return type implies the argument constraints, as well as knowledge that if the argument constraints are true they imply the return type. This is generally the case when there is only one type-axiom for a given function. When a function has several type-axioms, as with function overloading/polymorphism, the return value does not generally determine the argument constraints, and so only the "implication" (==>)form of type-axiom is used.

The RETURN_PRED is usually a simple predicate constraining the return type, but SL allows a further schema for type-axioms in which the RETURN_PRED predicate takes arguments. This allows the return constraint to depend on the function arguments.

```
ARGUMENT_PRED <==> RETURN_PRED[FUNCTION[ARGS1,…], ARGS2,…];
```

Example:

```
And[PriceP[p1], PriceP[p2], p1[3] = p2[3]]
     <==> PriceValuePred[PriceAdd[p1, p2], p1[3]];
```

The additional arguments effectively allow the RETURN_PRED predicate to have access to the arguments supplied to the function, which means the return type can depend on the arguments. This allows the use of named predicates with multiple arguments, and

provides another way to allow complex logic without the requirement of anonymous inner predicates. As an example, consider the structured entity:

```
<Price,quantity,currency>
```

This is defined with `quantity` as a number, and `currency` as an entity drawn from a set of possible currencies. Addition of two *Price* entities can be defined with a function `PriceAdd`, with a type-axiom that requires that the currencies of the summands must match, and which additionally requires that the currency of the sum must match as well. A `RETURN_PRED` predicate can be defined as:

```
PriceValuePred[x, curr] <==> And[PriceP[x], x[3] = curr]];
```

Here, `PriceP` is a structured entity recognition predicate that ensures `x` is of form `<Price,quantity,currency>` as described above. `PriceValuePred` ensures that

the currency supplied as `curr` matches the currency value in the return value, `x`. The full type-axiom for `PriceAdd`, using `PriceValuePred`, is:

```
And[PriceP[p1], PriceP[p2], p1[3] = p2[3]]
     <==> PriceValuePred[PriceAdd[p1, p2], p1[3]];
```

This type-axiom allows the following expression to type-check:

```
PriceAdd[PriceAdd[<Price,2,USD>,<Price,5,USD>],<Price,3,USD>]
```

7.2 Unconstrained Arguments

Some type-axioms only constrain the return type, for example:

```
BoolP[EqualsP[{x, y}]];
```

or equivalently,

```
True <==> BoolP[EqualsP[{x, y}]];
```

This specifies that `EqualsP` returns a `Boolean` value with no restrictions on its arguments `x` and `y`. Semantically, it implies `EqualsP` can test any two arguments for equality, yielding `True` or `False`, regardless of the type of the arguments.

7.3 Argument Constraints

Consider:

```
PersonP[x] <==> MaleP[FatherOf[x]];
```

This specifies that `FatherOf` function is only defined if the argument satisfies `PersonP`, and in such cases, it returns a value satisfying `MaleP`. Separate axioms (not shown) would define `FatherOf`.

The next example shows two arguments with corresponding type constraints:

```
And[PersonP[x], PersonP[y]] <==> BoolP[ChildMotherP[x, y]];
```

This constrains `ChildMotherP` to two arguments, each satisfying `PersonP`, with a `Boolean` return type.

The following is a more complex type-axiom containing a conjunction of constraints on the argument. It specifies that the function `ArithmeticMean` takes a set as an argument, and that all members of that set must be numbers.

```
And[SetP[numbers$], (MembP[x, numbers$] ==> NumberP[x])] <==>
     NumberP[ArithmeticMean[numbers$]];
```

Let's take this apart. The return type of `ArithmeticMean` is `NumberP`, simple enough. But the argument constraints are more complex. The argument constraint is a conjunction of two Boolean expressions. The first, `SetP[numbers$]`, constrains the argument `numbers$` to be a `Set`. The second expression

```
(MembP[x, numbers$] ==> NumberP[x])
```

states that all members of `numbers$` must be a number. This is an example of a typed set.

The previous example involved a simple conjunction of constraints, but type constraints for arguments can be arbitrarily complex logical functions, including relations between arguments, as seen in the next type-axiom example:

```
And[MusicianP[m], BandP[b], WasMemberP[m, b]] <==>
     YearP[MusicianBandStartYear[m, b]];
```

The function: `MusicianBandStartYear` specifies the year a musician started playing with a specific band, and it takes musician (m) and a band (b) as arguments. The type-axiom specifies that the musician must have been a member of the band at some point. It would then be a syntax error to call the function with a band with which the musician has never played. Logically, in that case the function would NOT be guaranteed to return a `YearP` type, and would instead return unevaluated or signal an error (see Section 17). Simple type theory (STT), with its use of signatures to represent type information (Hindley, 1997), is unable to represent this kind of constraint between arguments.

7.4 Return Value Constraints

The previous sections discussed constraints on the arguments of a function. The return value of a function is also constrained, typically by a single type predicate, as used in all of the previous examples. Although syntactically simple, this already supports disjunctive return value types by defining a named predicate for this purpose. For example, a version of the square root function (`SpecialSquareRoot`) that takes only real inputs, and returns either an integer or a real number might have a type-axiom with a named predicate as follows:

```
## Type-axiom for SpecialSquareRoot:
RealP[x] <==> IntOrRealP[SpecialSquareRoot[x]];
```

The above makes use of the named predicate `IntOrRealP`, which can be given as:

```
## Type-axiom for IntOrRealP:
NumericP[x] <==> BoolP[IntOrRealP[x]];

## Definition of IntOrRealP:
IntOrRealP[x] = Or[IntegerP[x], RealP[x]];
```

Return value constraints can go beyond simple disjunction by using an anonymous inner predicate (see Section 6.10). This enables arbitrarily complex logical constraints on the return type, just as with the argument constraints.

For example, suppose a return type is a set of Integers. This could be written by first defining a named predicate, `IntegerSetP`:

```
IntegerSetP[x] <==> (And[SetP[x],(MembP[y,x] ==> IntegerP[y])]);

## Type-axiom for PrimeFactors, using IntegerSetP
IntegerP[x] ==> IntegerSetP[PrimeFactors[x]]];
```

But an anonymous inner predicate could also be used instead to avoid the need to define the return type in a separate axiom. To do this, simply turn the RHS of the `IntegerSetP` definition into an anonymous function, as follows:

```
IntegerP[x] ==>
   (And[SetP[@],(MembP[y,@] ==> IntegerP[y])])[PrimeFactors[x]];
```

In this example, the function `PrimeFactors` takes an argument constrained to satisfy `IntegerP`. The return type is constrained to satisfy an anonymous predicate (underlined above), defined to be a set whose members satisfy `IntegerP`. That is, it returns a set of integers.

7.5 Polymorphism

Input SL supports function polymorphism in which the same function can have different definitions and return-value types for different argument types or numbers of arguments. This is commonly referred to as "ad hoc" polymorphism (Jun et al. 2002). This is achieved with multiple type-axioms for the same function name. For example, consider a function `Owner` that returns the owner of various entities (which might be vehicles, real estate, domestic animals, etc). The implementation of this function might be very different for different classes of entities.

The primary advantage of polymorphism is that it avoids creating a large number of specialized functions.

For example, suppose these type-axioms are given:

```
VehicleP[arg] ==> PersonP[Owner[arg]];
PetP[arg] ==> PersonP[Owner[arg]];
```

The type-axiom uses forward implication, not equivalence, since the function returns a `PersonP` entity in both cases. Multiple function definitions can then be supported:

```
(Owner[arg] = AutoRegistryLookup[arg]) <==> VehicleP[arg];
(Owner[arg] = PetRegistryLookup[arg]) <==> PetP[arg];
```

When a function call with `Owner` is made, the type of the argument determines the definition which is logically implied.

SL supports full function overloading, where the same function can be defined with completely unrelated argument types, and with unrelated return types. For example, logical negation given by the `Not` function is defined in the usual way for Boolean arguments, returning a Boolean. But `Not` can also be defined for integers, mapping any non–zero integer to zero, and mapping zero to one. In this case, the two type-axioms are:

```
BoolP[x] ==> BoolP[Not[x]];
IntegerP[x] ==> IntegerP[Not[x]];
```

Note that in this case, the return types as well as the arguments are both different and unrelated (i.e. neither is a subtype of the other).

Operator overloading is a design choice that SL makes possible but does not require.

7.6 Subtyping in Type-Axioms

The SL type-axioms are a logical constraint framework. This combines well with the notion of subtypes, at the logical level. For example, consider the following subtyping axiom which defines a subtyping relationship:

```
## Subtyping axiom
IntegerP[x] ==> RealP[x];
```

and the following type-axiom for the primitive function `LessThanP`:

```
## Type-axiom for LessThanP
And[RealP[x], RealP[y]] ==> BoolP[LessThanP[x, y]];
```

These two axioms together imply that `LessThanP` can take mixed arguments of type `IntegerP` or `RealP`. This flexibility chains into calling functions. Consider the function `WithinRangeP` with the following type-axiom and definition:

```
## Type-axiom
And[RealP[x], RealP[lower], RealP[upper]] ==>
          BoolP[WithinRangeP[x, lower, upper]];

## Definition of WithinRangeP
WithinRangeP[x, lower, upper] <==>
     And[LessThanP[lower,x], LessThanP[x,upper]];
```

With the above definition and type-axioms, function calls with mixed types are logically supported:

```
WithinRangeP[2, 0.5, 8]
```

In this example, `WithinRangeP` is being passed integer arguments, even though its type-axiom specifies a `RealP` for its arguments. This is possible because of the earlier subtyping axiom stating that integers are types of reals.

At the procedural level, the inference agent will analyze the arguments and use the appropriate specialized implementation of `LessThanP`.

8 Sets and Multisets

8.1 Sets in SetLog: Overview

A defining feature of SetLog (SL) is its treatment of sets as primitive entities, as reflected in its name. In contrast, first-order logic (FOL) does not permit sets to be treated as entities, as it restricts quantification to a domain of discourse comprising only elementary objects. While some FOL extensions incorporate set-builder notation, such additions are extra-logical and not part of core FOL. Higher-order logic (HOL), however, supports quantification over sets. As a higher-order logic, SL allows sets to be treated as first-class entities and so they are part of SL's extended domain of discourse.

The primitive function `NullP[set$]` returns `True` if `set$` is the null set and `False` otherwise. The null set can be entered in input SL using the shorthand notation: {}.

8.2 Set Definitions

In SL, every set is defined by its membership function. For example, the set `PrimaryColors$` is defined as follows:

Example 8.2.1

```
MembP[x, PrimaryColors$] <==>
     Or[x=Red, x-Yellow, x=Blue];
```

In this example, the variable x is implicitly universally quantified. By convention, set names in SL end with a $ suffix (e.g., `PrimaryColors$`) – a practice for human readability that does not affect the logic. `PrimaryColors$` is a finite, closed set. The <==> operator ensures that the axiom provides both necessary and sufficient conditions for membership, enabling deductions about is and is not a member of `PrimaryColors$`.

In input SL, sets may be defined extensionally:

```
PrimaryColors$ = {Red, Yellow, Blue};
```

The translator converts such definitions into internal intensive format (Example 8.2.1).

An alternative representation of the same information is:

Example 8.2.2

```
PrimaryColorsP[x] <==> Or[x=Red, x=Yellow, x=Blue];
MembP[x, PrimaryColors$] <==> PrimaryColorsP[x];
```

Compared to Example 8.2.1, Example 8.2.2 introduces a predicate, `PrimaryColorsP`, which is then used in the membership definition. The resulting set `PrimaryColors$` is logically identical in both examples.

Example 8.2.2 illustrates a general form for the relationship between a set and its corresponding predicate SL, summarized as:

```
MembP[x, Set$] <==> P[x];
```

This general form embodies the *set–predicate duality*. Sets and predicates are not equivalent – a set lacks a truth value, while a predicate lacks a size – but they can be interconverted. This duality has several implications for SL:

1. **All sets are intensively defined in SL.** Unlike standard logic, where sets are defined extensionally (i.e. defined by listing members), SL treats all sets as intensional (defined by a membership function). SL allows for distinguishing an extensional definition (with disjunctions of domain entities) from intensional definitions (i.e. defined by general predicates). However, sets can also combine both aspects, such as "any citizen of the USA can become president unless they are have already served two terms", which defines the set of people who could become presidents by starting with the set of all USA citizens then excepting previous presidents. The generality of set definitions in SL implies that the intensive/extensive distinction is not fundamental.
2. **Sets inherit the type constraints of their predicates.** A predicate `P[x]` returns `True` or `False` only for arguments satisfying its type-axiom; otherwise, it is undefined. Consequently, the dual set's membership function (`MembP[x, Set$]`) may also return undefined. This contrasts with standard higher-order logic (HOL), where membership functions yield only True or False. For instance, if `PrimeP[x]` requires x to be an integer, querying `MembP[Paris, Primes$]` returns undefined because `Paris` fails the integer type constraint.
3. **Sets and predicates are dual.** This duality ensures seamless conversion between a set and its defining predicate.

Another consequence of set–predicate duality is that a set's membership may vary depending on the KB's domain of discourse. For example, in SL the set defined by "people born after 1950" will differ for different genealogical KBs. However, these sets are all generated by the same criterion, so SL regards them as the same, even though they may have different members. This appears to contradict the usual rule that sets are the same iff they have the same members. However, this rule is only applicable if applied within the

same KB – there is no reason that sets in different KBs with different domains of discourse should be the same.

Determining set equality within a KB can be challenging. Syntactically distinct entities may be logically equivalent, and verifying this can be computationally expensive, complicating tests for set equality or duplicate members.

8.3 Open and Closed Sets

The previous section focused on closed sets. SetLog (SL) also supports partial set membership through appropriate logical operators. For example:

Example 8.3.1

```
MembP[x, Colors$] <==
     Or[x=Red, x=Yellow, x=Blue];
```

This differs from Example 8.2.1 only in that it uses the <== operator instead of <==>. Example 8.3.1 defines an *open set*, asserting that `Red`, `Yellow`, and `Blue` are members of `Colors$`, while allowing for other possible members. Consequently, a query for Example 8.3.1, such as: `MembP[Green, Colors$]` returns "unknown" rather than False, as for Example 8.1. Thus, Example 8.3.1 provides sufficient but not necessary conditions for membership in `Colors$`.

Another form of partial knowledge arises when only necessary conditions are known:

Example 8.3.2 `MembP[x, PrimaryColors$] ==>`

```
     Or[x=Red, x=Yellow, x=Blue];
```

Here, the ==> operator replaces <==> from Example 8.2.1. This axiom states that if x is a member of `PrimaryColors$`, it must be one of `Red`, `Yellow`, or `Blue`; however, non–members may or may not be among these colors. Examples 8.3.1 and 8.3.2 illustrate distinct types of open sets, defined by sufficient or necessary conditions, respectively. To our knowledge, prior KRR systems and formal logics have been limited to closed sets, lacking mechanisms for such partial set definitions.

For convenience, input SL provides shorthand syntax for closed sets:

Example 8.3.3 `PrimaryColors$ = {Red, Yellow, blue}`

The input to internal SL translator converts this shorthand into the membership function shown in Example 8.2.1

The above examples use the `Or` function, but ideally, the `XOr` (exclusive or) function would be more precise, as only one disjunct should hold true. However, when SL's unique name convention is enforced, only one equality in an `Or` expression can be true, making `Or` equivalent to `XOr`. If the unique name convention is relaxed, `XOr` should be used to ensure correctness.

Set definitions, like those above, can be added to any Knowledge Base (KB) as axioms without conflicting with existing axioms, provided the set's name is unique. A set definition

axiom (i.e., a membership function) is true by definition and cannot be falsified, effectively creating a new set entity.

The `PrimaryColors$` set is a fixed, constant set. SL also supports set-valued functions, such as `ChildrenOf$[x]`, which return a set (possibly the null set) for any valid `x`.

8.4 Set Size

The `SizeOf` function, a primitive in SL, returns the number of distinct elements in a set. For explicitly defined sets, `SizeOf` can be computed by counting the distinct elements. For sets defined by membership functions, determining size may require complex inference. Even when exact sizes are unknown, SL supports expressing constraints, such as discrete probability distributions over possible sizes or ranges of size, e.g., (`SizeOf[MayaCities$] < 200`), without requiring a precise count.

8.5 Set Existence

Since SL allows existential quantification over sets, it raises the question: what does it mean for a set to "exist" in SL? Two forms of set existence can be distinguished: *defined existence* ("Is the set defined?") and *membered existence* ("Does the set have at least one member?"). The latter is typically what is meant by "existence" in natural language. For membered existence, a set exists if as the name implies it is non–empty, and returns False if it equals the null set. For example, the query "Do any countries with no sea borders exist?" is a membered-existence question, answered True (e.g., Switzerland). Membered existence can be represented in SL with two axioms:

Example 8.5.1

```
MembP[x, NoSeaBorderCountries$] <==> And[CountryP[x],
          Not[HasSeaBorderP[x]]];
SizeOf[NoSeaBorderCountries$] > 0;
```

The first axiom defines the set, establishing symbolic existence, while the second asserts membered-existence.

Defined existence is a meta property of a Knowledge Base (KB), making queries about whether a set is defined or not a meta question. For example, consider the set of rectangular triangles:

Example 8.5.2

```
Memb[x, RecTriangles$] <==> And[RectangleP[x], TriangleP[x]].
```

We know from the definition of rectangles and triangles that such a set could never have members. Nevertheless, the axiom above is a well-formed formula in SL (See Section 11). While this example could not have members, and so is equal to the null set, other defined sets could. By far the most common type of sets in SL are membered sets, such as:

Example 8.5.3

```
MembP[x, BordersUSA$] <==> Or[(x = Canada), (x = Mexico)];
```

In logic literature, little distinction is made between *potentially* defined sets (all possible sets in a KB's domain) and *actually* defined sets (those explicitly defined). The number of potentially defined sets equals the power set of the domain's elements – a vast number – while only a small fraction of these possibilities are typically defined in a KB. Thus, the question "How many sets exist in a KB?" depends on whether it refers to actually or potentially defined sets.

8.6 Set Levels

To avoid Russell's paradox, SL adopts a set type-hierarchy, assigning each set a level and requiring that its members be exactly one level lower (Russell & Moore, 2015). Ground entities are level-0, sets of ground entities are level-1, sets of level-1 sets are level-2, and so on. For example, the `UnionOf$` function takes a set of sets as input and returns a set containing all distinct elements from the input sets, with the output set at the same level as the input sets' components. The null set, `{}`, is an exception, having no level and thus able to belong to any set, including itself at any level. To prevent paradoxes, SL includes the axiom `{{}} = {}`, effectively ensuring a single null set.

SL employs a *strict hierarchy*, where a set's members are exactly one level below it, unlike the *cumulative hierarchy* of Zermelo–Fraenkel set theory (used in standard formal logic), which allows members at any lower level. The strict hierarchy simplifies operations like set complement, as the strict hierarchy ensures all elements of a complement are at the same level.

SL provides numerous primitive set operations, including `Union$`, `Intersection$`, `Complement$`, and `SubSet$`. Additional useful functions include:

- `SetMap$[fn, set$]` – Returns a set formed by applying function `fn` to each element of `set$`.
- `Get1[set$]` – Returns the sole element of a singleton set, with a type-axiom ensuring `set$` has exactly one element.
- `Choose1[set$]` – Nondeterministically returns a randomly chosen element from set.
- `MaxElement[fn2, set$]` – Uses a binary Boolean comparison function `fn2` to return an element at least as large as any other in `set$`. If multiple elements share the same maximum value, the result may be nondeterministic.

8.7 Set Complement

In SetLog (SL), each set is associated with a specific level, which constrains the types of elements it may contain. This hierarchical structure ensures well-formedness in set construction and interpretation.

The complement of a set S1$ with respect to a superset U$ is defined as the set of all elements in U$ that are not members of S1$. This can be expressed using the membership predicate as follows:

```
MembP[x, Complement$[set$, superSet$]] <==>
    And[MembP[x, superSet$], Not[MembP[x, set$]]]
```

In this expression, x is one level below `set$`, which is itself one level below `superSet$`. The `Complement$` function is a set-valued function that takes a set and a superset as arguments and returns the complement of the set within the superset.

While this definition relies on the membership predicate, it is equally valid to define complement in terms of the `Subset, UnionOf,` and `IntersectOf` functions.

Given sets `S1$`, `S2$`, and `U$`, all at the same level, `S2$` is the complement of `S1$` with respect to `U$` if and only if:

1. S1$ ⊆ U$,
2. S2$ ⊆ U$,
3. S1$ U S2$ = U$
4. S1$ ∩ S2$ = {}

This can be represented in SL as:

```
Complement$[S1$, U$] = S2$ <==>
    And[
        Subset[S1$, U$],
        Subset[S2$, U$],
        (UnionOf[S1$, S2$] = U$),
        (IntersectOf[S1$, S2$] = {})
    ]
```

This formulation ensures that the complement is both exhaustive (covering all of `U$`) and mutually exclusive with `S1$`.

8.8 Multisets

SetLog (SL) supports multisets (also known as bags or heaps), which are sets that permit multiple occurrences of elements. Like sets, multisets are unordered, meaning the arrangement of elements is logically irrelevant. Multisets can be created using the Join operator, analogous to the `Union$` operator for sets, which combines two or more sets or multisets into a single multiset, preserving all elements, including duplicates. Converting a multiset to a set involves removing duplicate elements, while adding elements to a set may yield a multiset if duplicates are introduced. An example of a multiset are the eigenvalues of a matrix, as some eigenvalues may appear multiple times.

In input SL, multisets are denoted with square brackets and curly braces, e.g., `{[a, b, a, c]}`, distinguishing them from sets, which use `{a, b, c}`. Two multisets are equal if and only if they contain the same elements with identical multiplicities.

9 Sequences

In SetLog (SL), sequences are a specialized type of function that map natural numbers to values, representing elements at specific positions. For example, the sequence of recent U.S. presidents – Clinton, Bush, Obama, Trump, Biden, Trump – is represented in internal SL as:

Example 9.1

```
RecentUSPresidents[1] = Clinton;
RecentUSPresidents[2] = Bush;
RecentUSPresidents[3] = Obama;
RecentUSPresidents[4] = Trump;
RecentUSPresidents[5] = Biden;
RecentUSPresidents[6] = Trump;
```

Here, the function-call `RecentUSPresidents[2]` returns `Bush`.

In input SL, this can be represented in a simplified syntactic form as:

```
RecentUSPresidents = <Clinton,Bush,Obama,Trump,Biden,Trump>;
```

Unlike sets, sequences preserve element order, achieved by using successive positive integers as indices. Additionally, sequences allow duplicate elements at different positions, as seen with Trump appearing twice in the `RecentUSPresidents` sequence.

The previous example is an example of a fixed length sequence. SL allows sequences of arbitrary length to also be represented. For example, the sum of the elements of a sequence function is represented in mathematics using the notation:

$$\sum_{k=1}^{n} \mathrm{fn}(\mathrm{k})$$

Sigma (∑) is the name of a numeric function that sums numeric function values applied to a sequence of integers, and can be defined by the following axioms:

Example 9.2

```
## General recursive form
(k < n) ==> (Sigma[Fn,k,n] = Add[Fn[k],Sigma[Fn,k+1,n]]);

## Base case
Sigma[Fn, n, n] = Fn[n];
```

Type-axiom for `Sigma` (the serial addition function):

```
And[NumFunctionP[Fn], n > 0, k > 0, n >= k, IntP[n], IntP[k]]
     ==> NumP[Sigma[Fn, k, n]]
```

The Fibonacci sequence is another example of an arbitrary length sequence defined recursively:

Example 9.3

```
Fib[1] = 1;
Fib[2] = 1;
Fib[3] = 2;
(n > 2) ==> (Fib[n] = Fib[n-2] + Fib[n-1]);
```

Given the above logical definitions for `Sigma` and `Fib`, the following functional call:

```
Sigma[Fib, 1, 10]
```

would evaluate to the result 143.

Specialized functions can be defined to pick out components of sequences, such as:

```
Subrange[<a, b, c, d>, 2, 3] = <b, c>;
```

In the above function, the second argument gives an initial index and the third argument gives a final index for a subrange.

10 Quantification in SetLog

Quantification plays a central role in knowledge representation and reasoning (KRR), allowing systems to express and infer properties involving quantities. Classical Predicate Calculus (PC) provides two standard quantifiers–universal (∀) and existential (∃)–which enable statements about "all" or "some" members of a domain. A unique existence quantifier (∃!) can also be defined. While these quantifiers are theoretically sufficient for expressing quantitative information, their practical application can often be cumbersome and unintuitive (Barwise & Etchemendy, 2002).

10.1 Quantification in SetLog

A primary motivation behind the development of SetLog (SL) is to offer a more intuitive and practical formalism for representing and reasoning about quantities. SL enables this by treating sets and numbers as primitive types, allowing direct and concise formulations of quantitative statements. The contrast between SL and PC is illustrated in the following example:

Example 10.1
English: There are more people in Room A than in Room B.
Predicate Calculus:
$\neg\exists f\,[\forall y\,(InB(y) \rightarrow \exists! x\,(InA(x) \wedge f(y, x))) \wedge$
$\forall y_1\,\forall y_2\,\forall x\,((InB(y_1) \wedge InB(y_2) \wedge f(y_1, x) \wedge f(y_2, x)) \rightarrow y_1 = y_2) \wedge$
$\forall x\,(InA(x) \rightarrow \exists y\,(InB(y) \wedge f(y, x)))]$

The predicate calculus expression above states that there is no injective and surjective function from B to A – i.e., that the cardinality of B is strictly less than that of A. While

logically correct, this formulation is complex and difficult to interpret, particularly for those unfamiliar with functional encodings of cardinality comparisons.

SetLog:

```
MembP[x, InA$] <==> InRoom[x, A];
MembP[x, InB$] <==> InRoom[x, B];
SizeOf[InA$] > SizeOf[InB$];
```

Here, the sets `InA$` and `InB$` are defined by their membership function, and a simple inequality directly compares their sizes. This representation preserves the semantics of the original English sentence while dramatically reducing syntactic and conceptual overhead. The clarity and conciseness of the SL version exemplify how incorporating sets as first-class objects can improve the usability and readability of logical KBs.

A further complexity with quantifiers is in determining the scope of the quantifier. SL eliminates explicit quantifiers, avoiding the need to explicitly determine their scope. An analogous approach is used in Predicate Calculus, with Skolem functions to eliminate existential quantifiers, as described in (Russell & Norvig, 2016). Skolemization eliminates existential quantifiers by replacing them with Skolem functions, but in doing so, it changes the logical form of the sentence in a way that preserves satisfiability but not logical equivalence (Akama & Nantajeewarawat, 2011) In contrast, SL's quantifier elimination preserves full logical equivalence. The following subsections explain how SL achieves this.

10.2 Universal Quantification

In SL, explicit universal quantifiers are omitted, so the corresponding variables are implicitly quantified universally. These universal variables are denoted by letter strings with a leading lowercase letter (e.g., `x`, `name`, `bagNumber`) or a variable name with a parameter using square brackets (e.g., `student[3]`). For example:

```
RelativelyPrimeP[x, y] <==> (GCD[x, y] = 1);
```

This axiom is true for all integers `x` and `y`. In SL, universal variables are scoped to individual axioms, meaning the variable `x` in one axiom is distinct from `x` in another, preventing unintended interactions across axioms. The constraint that `x` and `y` must be integers follows from the type-axiom for the `RelativelyPrimeP` predicate. In general, in SL, universally quantified variables are restricted to a limited domain of discourse (relative quantification) dictated by the corresponding type-axiom in the function in which they occur. Total functions, such as "=" or `IntegerP`, have the KB's entire domain of discourse as their domain.

10.3 Existential Quantification

In SetLog (SL), named entities within a Knowledge Base (KB) are assumed to exist. The unique name convention in internal SL ensures that each entity has exactly one name. If `FatherOf[Bill] = John` is known, either `'FatherOf[Bill]'` or `'John'` could theoretically represent the same entity. However, internal SL's unique name convention mandates using the elementary form `'John'`, but retains the axiom `FatherOf[Bill] =`

`John` in the KB. This axiom allows replacing any reference to `FatherOf[Bill]` with `John`, and the unique name convention requires this substitution.

A more complex case arises when a unique entity is known to exist but its identity is unknown. For example, the assertion that every person has exactly one father can be represented in SL as:

```
PersonP[x] <==> MalePersonP[FatherOf[x]];
FatherOf[x] != x;
```

The first axiom, a type-axiom, asserts the existence of a male person for `FatherOf[x]` without identifying him. The second axiom constrains the father's identity by excluding self-reference. If sufficient constraints exist, inference may reveal the entity's identity. For any valid `x`, `FatherOf[x]` denotes a unique entity, whether identified or not.

In first-order logic (FOL), unique existence is often denoted by ∃! (exactly one exists), and can be defined using standard quantifiers. For example:
∃!x President[x] <==> ∃x [President[x] ∧ ∀y(President[y] ==> (y = x))]
This asserts a unique entity x for which President[x] holds (i.e., the current president).

In SL, unique existence is simply conveyed by a named entity, such as: `John`, or a function call with a definite value, such as: `FatherOf[Mary]`.

FOL has only one existential quantifier: "∃", such as in: ∃x PlanetP[x], indicating at least one planet exists. This quantifier does not distinguish the case where unique existence is known unless the quantifier ∃! defined above is used instead. SL, however, represents such general existence with two axioms:

```
MembP[x, Planets$] <==> PlanetP[x];
SizeOf[Planets$] > 0;
```

The first defines the set `Planets$`, and the second ensures it is non–empty. SL allows more precise set size specifications, such as `SizeOf[Planets$] = 9` for an exact count, or constraints like `169 < SizeOf[PharosOfEgypt$] < 201` for a range. Even finer existential granularity is possible by assigning probabilities numbers to each possible value.

In summary, SL distinguishes *two* forms of existential quantification: *unique existence* and *general existence*, aligning with the English determiners "the" and "a." Unique existence is represented by a named entity. General existence is captured by defining a set and specifying it has at least one member, while allowing more precise counts or ranges if that information is available. In both cases, "existence" means that there is an entity in the KB's domain of discourse, rather than making an ontological statement about some correspondence to something in the outside world.

In summary, by leveraging sets and numbers as primitive types, alongside standard numeric functions, SL facilitates intuitive and flexible representation of quantitative information.

10.4 Nested Quantifiers

Complications can occur in quantifier elimination with nested quantifiers, as shown in the following examples:

English: "Everybody loves exactly one other person"
PC: ∀x ∃!y (x ≠ y ==> Loves[x, y])
SL: `LovedBy[x]`

Here LovedBy is a function that returns the unique person who x loves.

English: "Everybody loves at least one other person"
PC: ∀x ∃y (x ≠ y ==> Loves[x, y])
SL:
```
And[MembP[y, LovedBy$[x]] <==> LovesP[x,  y],
     PersonP[x] ==> (SizeOf[LovedBy$[x]] ≥ 1)]
```

In this case, `LovedBy$` is a set-valued function that returns the set of all persons loved by `x`.

English: "There is someone whom everyone loves"
PC: ∃!x ∀y Person[y] ==> Loves[y, x]
SL: `PersonP[y] ==> Loves[y, A]`

Here `A` is a unique person (known or unknown).

English: "There is at least one person whom everyone loves"
PC: ∃x ∀y Person[y] ==> Loves[y, x]
SL:
```
And[PersonP[y], MembP[x, LovedByAll$]] ==> LovesP[y,  x],
```
and `SizeOf[LovedByAll$] ≥ 1]`

These examples show that with nested universal and existential quantifiers, the order matters. Quantifier elimination in SL for arbitrarily nest quantifiers can be achieved by applying the quantifier elimination rules outlined above recursively. The examples above do not cover the case of negated quantifiers. In such cases, the quantifier negation can be eliminated by using the substitution rules:

¬[∀x P[x]] → ∃x ¬P[x], and ¬ [∃x Q[x]] → ∀x ¬Q[x].

11 SetLog Syntax

This section outlines the syntax of *internal* SetLog (SL). *Input* SL incorporates mixed prefix and infix notation, parentheses for disambiguation, synonyms, and other features, making it more complex to specify than internal SL.

Formal Syntax for Internal SetLog in Bachus–Naur form is:

```
<term> ::= <constant>|<variable>|<funcName> "[" <term> "]"
<funcName> ::= <capital-letter> <id-rest> <opt-dollar>
<constant> ::= <capital-letter> <id-rest> <opt-dollar>|<number>
<variable> ::= <lower-letter>  <id-rest> <opt-dollar>
```

In English this form says:

- a constant is a either a sequence of alphanumeric-plus-underscore characters, with an initial capitalized letter, and with an optional terminating '`$`' character, or it is a number such as `2`, `-4.73`, etc.
- A variable is exactly the same as a non-numeric constant, except the initial letter is uncapitalized
- Underscores and digits are not allowed as the first character for numbers, constants and variables.

Other components of the BNF rules are:

```
<id-rest> ::= "" | <id-char> <id-rest>
<id-char> ::= <letter> | <digit> | "_"
<opt-dollar> ::= "" | "$"
<letter> ::= <capital-letter> | <lower-letter>
<capital-letter> ::= "A" | "B" … | "Z"
<lower-letter> ::= "a" | "b" … | "z"
<number> ::= <opt-minus><digit><num-seq><opt-dot-seq>
<opt-minus> ::= "" | "-"
<num-seq> ::= ""|<digit><num-seq>
<opt-dot-seq> ::= ""|"."<digit><num-seq>
<digit> ::= "0" | "1" … | "9"
```

Note that this syntax is recursive, since `term` is the argument to a function call, producing another term. Also, this syntax specifies that there is only one argument to a function, but this is not a problem since a term can be a sequence or a set.

A *well-formed formula (wff)* is any term constructed according to the above syntactic definition.

11.1 Entities

Entities – also known as things, objects, constants, or individuals – are distinguished by the fact that they evaluate to themselves. In SL, entities are written as sequences of alphanumeric characters according to the BNF rules above.

All entity names are capitalized. Boolean-valued functions, referred to as predicates, are usually distinguished by ending with a capital `P`. A single `$` suffix indicates a set entity or a set-valued function, while the absence of this suffix denotes a non-set entity. A double `$$` suffix is used to represent multisets.

Entities are categorized into subtypes as follows:

11.1.1 Elementary entities

Elementary entities are represented by capitalized character sequences or numeric notation, such as `Bill`, `Red`, `True`, `-3.1`, etc.

11.1.2 Higher-order entities

Higher-order entities include non-ground entities like sets, multisets, and functions, where the function name itself is treated as an entity (e.g., the function `Sort` takes a function name as an argument). Reification turns symbolic expressions, sets and functions, into higher order entities.

11.2 Functions

A function call in SL uses the syntax:

```
FunctionName[argument]
```

The argument may be an entity, set, multiset, sequence–function, function, or another function call. SL restricts functions to a single argument, but this is not limiting, as the argument can be a set or sequence containing multiple elements. In input SL, sequence arguments can omit angle bracket notation (<…>), as sequences are assumed by default. For example, `Implies[<P, Q>]` and `Implies[P, Q]` are equivalent in input SL.

Another example of a function call is: `Sort[seq, fn]`, where `seq` is a sequence and `fn` is a function name. The type-axiom for `Sort` requires `fn` to be a binary predicate that judges whether any two distinct elements from `seq` are correctly ordered. This illustrates SL as higher-order logic, since the function entity `fn` is being passed as an argument.

There are two special subtypes of functions in SL: sequence functions and predicate functions.

11.2.1 Sequence Functions

Sequence functions map natural number indices to values, for example, for the sequence function `Countries` we have:

```
Countries[1] = Afghanistan;
Countries[2] = Albania;
 …,
Countries[142] = Zimbabwe.
```

Anonymous sequence functions can be represented in input SL using the `MakeSeq` function. For example, the input SL

```
<Afghanistan, Albania,…,Zambia, Zimbabwe>
```

becomes

```
MakeSeq[<Afghanistan, Albania,…,Zambia, Zimbabwe>]
```

in internal SL. `MakeSeq` is a primitive function which simply returns its sequence argument. It is used to convert any sequence in input SL using the angle notation into an internal SL sequence function. This is true even when the sequence is used only locally inside an expression, such as:

```
AppendSeq[<3,5,7>, 9]
```

becomes:

```
AppendSeq[MakeSeq[<3,5,7>], 9].
```

The sequence generated by `MakeSeq` in this case is discarded once `AppendSeq` has evaluated.

Sequences may have a fixed length (like the countries example above) or be unbounded (e.g., the Fibonacci sequence). These equate to range restrictions on the internal sequence function. Key properties of sequences include the property that elements are ordered and the possibility of duplicate entries in the sequence, properties which follow automatically from the definition as a function on the indices.

11.2.2 Predicate Functions

Predicate functions return Boolean values – i.e. T or F. Predicate functions include the familiar functions: `>`, `=`, and `Not`. Logical functions are a subset of predicate functions, that accept only Boolean arguments, e.g., `And`, `Implies`, or `Not`, and return a Boolean value. Unlike PC, SL does not distinguish predicates from functions; predicates are simply functions that happen to be Boolean-valued. In SL, the Boolean values T or F are just specific ground entities in the domain of discourse of the KB in which they occur. Their interpretation as `True` or False only occurs at the meta-level, so an axiom such as `MaleP[Lily] = F` is interpreted by the inference agent as true, just as the axiom `Sq[3] = 9` is interpreted to be true. This is how Tarski's truth hierarchy is implemented – i.e. truth is a meta property of an axiom system, not a property that can be represented in the base axiom system itself. This distinction between T (in base axiom system) and True (a property of an axiom in the meta-system) prevents an infinite regress of the form: `((MaleP[Nick] = True) = True) = True` … A similar approach is used in Boolean algebra.

11.3 Variables in SetLog

Variables in SetLog (SL) are universal. This means they are implicitly universally quantified, so that any axiom containing such variables holds true if and only if it is true for all possible values of the variables within their domain. For example:

```
ParentP[x, y] <==> Or[MotherOf[x]=y, FatherOf[x]=y]
```

Here, `x` and `y` are variables, constrained by type-axioms for `MotherOf` and `FatherOf` to be of type `Person`; thus, their domain is the set of all people. This illustrates that SL's "universal" quantification is always relative to a specific domain, as indicated by the corresponding type-axiom. This means that `ParentP[x, y]` is undefined if `x` or `y` is not a person.

Since SL is a higher-order logic, it supports quantification over higher-order entities like sets and functions. For instance, the concept of a monotonic function is defined as:

```
MonotonicP[fn] <==> (x < y ==> fn[x] < fn[y])
```

Here, `fn` is a variable of type function, while `x` and `y` are variables of type number inferred from the type constraints of the `<` (`LessThan`) function. Similarly, for the binary predicate "`SubSetP`":

```
SubSetP[set1, set2] <==> (MembP[x, set1] ==> MembP[x, set2])
```

Here, `set1` and `set2` are variables of type `SetP`, and `x` is a variable with no type constraints at all. That is, the axiom is true for any pair of sets `set1` and `set2`, and given any particular choice of `set1` and `set2` it is true for all values of `x`.

Variables are denoted by uncapitalized names (e.g. `angle`, `modelCar`, `x`, etc.) or indexed forms (e.g., `students[3]`, `vec[2]`). They are scoped locally to the axiom or expression in which they appear, ensuring that an `x` in one axiom is distinct from an `x` in another.

12 Knowledge Bases

We assume that all knowledge in an AI is collected together into knowledge bases (KBs), and the axioms in these KBs are all in the SL language. In this section we elaborate on this assumption.

12.1 Structure of Knowledge Bases

A Knowledge Base (KB) in SL is a conjunction of SL axioms, both certain and uncertain, that model a domain. This conjunctive structure allows for logical transformations (e.g., De Morgan's laws), logical simplification, and consistency checking on the KB.

During inference, a KB is **assumed to be true**, implying the truth of all its axioms. SL doesn't distinguish between initial axioms and inferred propositions (theorems); both are simply referred to as axioms. This is because different subsets of a KB's axioms can serve as a basis for deriving all others, meaning a proposition labeled a theorem in one subset might be an axiom in another. Thus, the distinction between axioms and theorems is not fundamental, justifying SL's uniform calling both theorems and axioms, axioms.

All SL axioms adhere to the format `Fn[arg]=val`, ensuring they are Boolean. The assumption that all KB axioms are true prevents infinite regress – e.g., (`(Fn[arg]=val) = True`), etc. This assumption that a KB is True for inference purposes enables hypothetical and counterfactual reasoning, even when the KB is known at some meta-level to be False. While input SL allows predicate axioms like `MaleP[x]` or `MaleP[x] = True`, internal SL exclusively uses the (`Predicate[arg] = T` or `F`) form, treating predicates as functions returning Boolean values. That is, unlike PC, SL does not syntactically distinguish predicates from other functions, but instead treats both as functions.

We assume that for all KBs, the domain of discourse is finite. This assumption avoids problems associated with trying to include infinite quantities in knowledge representation. Also, restricting all sets to be of finite size means that the Size function is, in principle, always well defined. We do not regard the restriction to finite domains as limiting the expressiveness of SL, because the world is finite, and any agent's memory is also finite.

Even with infinite domains, like the integers, finitist methods can approximate the infinite by treating the infinite case as the limit of the finite case. Ever since Cantor introduced the idea of a completed infinity[4], a minority of mathematicians, starting with Gauss[5], have favored the limiting approach instead. These two views can yield different outcomes. For instance, Cantor held that the set of even numbers and all natural numbers share the same cardinality ($\aleph_0$). In contrast, the ratio of the number of even numbers to all numbers up to N is (½)(N+2)/(N+1), which approaches ½ as $N \to \infty$, unlike Cantor's ratio of 1. Which view better reflects reality is left to the reader.

12.2 Meta-KBs

In a multi-level knowledge base (KB) architecture, the ground-level axioms contained in one KB can be represented as corresponding entities in an associated meta-level KB. The meta-KB can record additional information about each axiom, such as its provenance, and usage history. Information concerning the properties of these meta-axioms is, in turn, maintained in a corresponding meta-meta-KB, and so forth, yielding a hierarchy of reflective KBs. Inference Agents (IAgts) operate within this hierarchy by drawing information from the appropriate level of KB, without crossing boundaries between ground and meta-levels. The mapping of specific ground-level axioms to corresponding entities in the meta-KB constitutes an extra-logical reification operation. At the meta-level, the syntactic structure of the ground-axioms is described by corresponding meta-level axioms. Consequently, reasoning about an axiom's structure (for example, for pattern matching) can be carried out at the meta-KB without the need to quote the original ground-axiom.

12.3 Theory of KBs

It is possible to construct a theory of KBs, where KBs are treated as entities, and the KB-theory represents properties of KBs and interrelationships between KBs. These relationships include compatibility (mutual consistency), disjointedness (no shared entities or relationships), hierarchy (one KB being a subset of another), abstraction (one KB is an abstraction of another), and other forms of KB relationships. This ability to represent and reason about KBs is important for AI self-knowledge introspection and reasoning.

---

[4] David Hilbert: "No one shall expel us from the paradise which Cantor has created for us."

[5] Carl Friedrich Gauss: "I protest against the use of an infinite magnitude as something completed, which is never permissible in mathematics. Infinity is merely a way of speaking." His view was that infinity should be understood as a limit concept, rather than some kind of number.

12.4 General and Specific KBs

SL axioms are categorized as **general axioms** or **domain-specific axioms**. General axioms contain only universal variables, e.g.

```
(MaternalGrandMotherOf[x] = y) <==> (MotherOf[MotherOf[x]] = y),
```

making them universally applicable. A KB consisting solely of general axioms is a "general KB," analogous to a formal system in predicate calculus. Such general KBs are incomplete because the universal variables in their axioms cannot be evaluated since there is no domain of discourse. However, it is possible to infer new general axioms from other general axioms in a general KB, such as inferring that a grandchild is always younger than their grandparents. Such derivations do not depend on any domain-specific axioms, and so are valid in any specific KB that included the general axioms in addition to ground axioms. Clearly, proving a proposition generally is better than proving the same proposition separately in every domain-specific KB, so using general KBs is preferred where applicable.

Conversely, **Domain-specific axioms**, include at least one ground entity (e.g., `MotherOf[Mary] = Anne`), and such axioms anchor the KB to a particular domain. KBs typically combine general and domain-specific axioms, allowing general axioms to be evaluated, because then universal variables have a domain of discourse to range over.

We speculate that logicians typically refer to general axiom systems when they discuss uninterpreted syntactic axiom systems, such as axiomatizations of geometry, graph theory, sets, etc. In the case of geometry, lines and points are defined in the abstract, but no specific lines and points are described. Likewise, in graph theory, nodes, edges and graphs are defined, but no specific graphs are mentioned. This abstraction allows general proof of properties of geometric shapes and definitions of such concepts as “connected” in graph theory without reference to specifics shapes or graphs. However, staying at the general level makes it impossible to infer properties of specific shapes and graphs, thus limiting the utility of general axiom systems.

12.5 Abstraction Mappings Between KBs

One advantage of organizing all knowledge in SL into KBs is it enables abstraction mappings between KBs. With abstraction, representation and reasoning can be performed at the appropriate abstracted level, without being confused by irrelevant detail. As an example, consider a genealogy KB mapping onto a Graph Theory KB (GT–KB). In this mapping, specific people map to specific nodes, and child-father and child-mother pairs map to labelled directed edges. The resulting directed graph can then be analyzed entirely within the GT–KB, to determine, for example, if two nodes are connected by a directed path. In doing the path inference, no reference is made to the original genealogy KB, as all necessary information for doing graph theoretic inference was mapped into the GT–KB.

This mapping we refer to as an “abstraction mapping” because only the relevant information has been mapped. Algebra typically uses a similar abstraction mapping, where, for example, the English description: “the price of a bag of flour and a liter of milk equals $10” is mapped into the algebraic equality: x + y = 10, where x = price of a bag of flour and y = the price of a liter of milk. Once in this abstracted form, ordinary algebraic inference can proceed without reference to the domain from which the algebraic expressions were derived. Once an algebraic solution is found, the abstraction mapping can be inverted to return domain specific values.

Logic allows multiple interpretations for the same formal system. This flexibility is mirrored by inverse abstraction mappings between KBs. For example, specific directed graphs in a GT–KB could model relationships in a social-network KB, a computer network KB, a genealogy KB, etc. Similarly, the group theory KB can have many interpretations. This means that the inverse abstraction mapping (interpretation) is typically one-to-many.

12.6 Universal KB

The Universal KB includes basic set functions such as union, intersection and size, as well as numeric functions like addition, multiplication, and comparison operators (e.g., greater than). Because sets and numbers are treated as primitive entities in the universal KB’s domain of discourse, they also become primitive entities in any knowledge base that is created by merging the universal KB with other domain-specific KBs. The universal KB also contains higher-order functions that allow the syntactic forms of axioms or other symbolic entities in a KB to be accessed and manipulated. These higher-order functions include: `Size`, `Length`, and `Arity`.

All the information in the universal KB is true by definition. That is the truth of axioms in the universal KB are not dependent on real world sensing – i.e. they are not contingent. In addition to the universal KB, an AI must build up a general knowledge KB that is the repository of all world knowledge. This includes such knowledge areas as chemistry, geography, history, engineering, etc. The general knowledge KB can be organized into domain specific sub-KBs, such as: chemistry → organic chemistry → aromatic organic chemistry etc. This component KB organization allows relevant KBs to be loaded into working memory without overloading memory. If information that is not in working memory is required, this information can be retrieved from the general knowledge KB, but at the cost of retrieval latency.

12.7 Generation of KBs

A major challenge in building KBs lies in the scale of knowledge that needs to be represented. Manual entry of SL axioms by human users is not feasible at scale, particularly for complex or dynamically changing knowledge domains. Therefore, automated methods for KB generation and maintenance are essential. The emergence of advanced Large Language Models (LLMs) has introduced the possibility of fully automating the conversion of information in text, figures, images etc. from one form to another. For example several groups have demonstrated turning English sentences into knowledge

graphs (Carta et al., 2023). Initial experiments in using LLMs to translate English sentences to SL have been promising.

12.8 Working KBs

When preparing a KB for inference in a particular subject area, the universal KB must be loaded, in addition to one or more subject-specific KBs. These KBs are merged into a single, combined working KB. For the resulting KB to function properly, the component KBs must be mutually compatible, meaning the combined working KB must remain logically consistent.

It's important to note that a working KB created for representing a specific domain may not be complete. That is, there may be queries that cannot be answered using only the information in the working KB, but which could be answered with information from other KBs. For example, in a genealogy KB, one might try to evaluate the query: "Find all people born after World War II." To answer this question, the KB must know the end date of WWII. If that information is not present in the genealogy KB, the Inference Agent (IAgt) must be able to retrieve it from the system's broader KBs. Therefore, inference in SL must be capable of reaching beyond the immediate working KB when required.

During inference, the IAgt will typically generate many deduced axioms that are added to the working KB for use in further inference steps. When the overall inference has terminated, the KB manager agent may transfer the most useful of these derived axioms to the domain KB in order to avoid having to re-derive them.

12.9 KB Dynamics

Axiom systems are typically assumed to be static. However, in an AI system, KBs will typically grow or be modified as more information becomes available. The question we address here is how such KB dynamics is managed. In the case where new axioms are to be added, the first thing to be checked is whether the new information is consistent with the previous information. If any inconsistency is detected, the inconsistency must be resolved, either by not adding the new axiom or deleting previous axiom(s). Because full consistency checking is usually computationally infeasible, only limited consistency checking is possible in practice, leading to potentially inconsistent KBs. A consequence of possible inconsistency is that the IAgt should be able to detect inconsistencies during inference and schedule their resolution.

One of the desirable properties of logical systems is that they are monotonic, which means that any inferred axioms/theorems remain true if new, consistent axioms are added to a KB. If new axioms are not redundant (i.e. not deducible from the current KB), the effect of adding new axioms is to enable more theorems/axioms to be inferred that were not inferable from the unaugmented KB. Likewise, if an axiom is deleted from a KB, it does not change the truth-value of previously inferred theorems, unless the previously inferred theorems use the deleted axioms as an essential premise(s). Efficiently deciding which theorems are dependent on which axioms can be computationally expensive, and various "truth-maintenance" methods have been developed to deal with this issue (McAllester,

1990, Doyle, 1979). Because of the complexity of truth maintenance, various non-monotonic approaches to dynamic knowledge have been developed, but we do not use this approach in SL because they are not well founded and computationally infeasible.

The monotonic property of KBs is violated for some axioms that are inferred from a KB in a dynamic environment. For example, the inference of the number of persons in a KB is no longer valid if new people are added to a KB. This means that any inferred axioms are not automatically true in a modified KB, requiring their rederivation or use of a truth maintenance method to ensure validity.

One way of constructing more comprehensive KBs is by merging separate KBs into one. For this merging operation to succeed, all the axioms in the merged system must be mutually consistent. One of the practical difficulties in such mergers is that the names of entities and functions in the original KBs may differ even when they are functionally equivalent. Such alternative naming violates the unique name convention in the merged KB. In principle, this type of name collision could be removed by consistent renaming. A more complicated issue arises when two or more structured entities are referring to the same entity. For example, two syntactically different paper citations from different sources may actually refer to the same paper. In the genealogy context, it may not be clear whether Sir William Smith is the same as William C. Smith mentioned in a different document.

12.10 Missing Information in Knowledge Bases

When dealing with real-world data, missing information is endemic. In a genealogy KB, for example, if one goes far enough back in time, unknown individuals become common. These individuals must have existed because everybody has a father and mother, even if there is no individual known for them.

In SL KBs, any information about a function's value for which there is no information does not appear as an axiom. That is, by default, any function call whose value is completely unknown is missing from the corresponding KB. In other situations, there may be only partial information about a function call's value. In this case, the information that *is* known is represented in different forms. In cases where the unique value of a function is known to exist, but the identity of this entity is unknown, information about its possible values can be represented by a disjunctive axiom, using the Xor operator. For example, if `MotherOf[John_Smith]` is unknown, but constrained to be one of a small number of possibilities, then these possibilities can be represented in SL by the axiom:

```
Xor[MotherOf[John_Smith]= Mary, MotherOf[John_Smith]= Sue,…]
```

This axiom acts as a constraint on the possible values for `MotherOf[John_Smith]`.

In addition to constraints on individual function-call values, there may also be constraints between different function-call values. For example, in the Map Coloring problem (See Section 17.2.1), there is a constraint that any two states that border each other cannot have the same color, in addition to the constraint that any state can only have one of 3 different colors. The set of mutually constrained values constitute a constraint satisfaction

problem (CSP) and such CSPs may have no solution: (the constraints are inconsistent) or one solution: (the values are then known), or many solutions, so there is a conjunction of possible values. Another possible representation of incomplete knowledge is in the form of conditional values. For example, the result of evaluating the function call: `Integrate[x^n, x]` would be the conditional result:

```
Integrate[x^n, x] = And[ ((n = -1) ==> (ans = Log[|x|]) ),
     ((n != -1) ==> (ans = x^(n+1)/(n+1)) ]
```

13 Higher–Order and Meta-Logic in SetLog

Sets, numbers and functions are higher-order entities in SetLog, and since sequences are really functions, sequences are also higher-order. SetLog is a higher-order logic because it can reason about these entities directly. Previously in this paper, higher-order concepts have been discussed repeatedly (for example see Section 6.9, Section 6.11, Section 8.6, and Section 11.1.2).

This use of higher-order reasoning is distinguished from meta-logic. Meta-logic is a process by which axioms in a KB become entities in a meta KB. For example, in order to discuss attributes of axioms such as the number of variables, it is necessary to talk about the axiom in structural terms. This requires reification, by which the axiom is converted to a tree structure. Reference to a particular variable in an axiom then becomes a reference to a particular structural component. Any given KB has a meta-KB that names and describes its axioms. For example, given the SL axiom:

```
Equals[FunctionName[arg], Value];
```

This axiom might be named `Axiom3`, which would literally be a tree, represented as labeled sequences of further meta-descriptions. `Axiom3[1]` would be a sequence representing "`FunctionName[arg]`", and `(Axiom3[1])[1]` would be a leaf node representing "`arg`".

Concepts such as pattern matching exist at the meta-level, because they refer to invididual syntactic components of expressions and not their logical meaning. The inference agent (IAgt) effectively operates at this level to bind variables in function calls (see Appendix 7).

Consider the axioms and type-axiom for `Factorial` presented earlier in Section 6.3.1:

```
(Factorial[n] = n*Factorial[n-1]) <==> (n > 1);
Factorial[1] = 1;

## Type-axiom for Factorial:
NatNumP[n] ==> NatNumP[Factorial[n]];
```

This states that `Factorial` is a function which takes a single natural number as argument. When the IAgt is presented with a logical evaluation of `Factorial[7]`, it does a pattern match of against the defining axioms for `Factorial`, coupled with the type-axiom for `Factorial`. These yield two patterns. The first is a single natural number argument with a

condition that the argument be greater than one, and the second is a single natural number argument which is the constant `1`. Of these two, the first matches the requested evaluation of `Factorial[7]`, and the IAgt does a substitution of `n → 7`, and applies this to the function definition. Then `n*Factorial[n-1]` becomes `7*Factorial[6]`. As described previously in Section 6.3.1, the process then recurses, to yield the eventual result of `5040`.

This functionality could be exposed directly with meta-KB functions. For example, `PatternMatchP[expr,pattern]` would be a predicate that matches a pattern to an expression, and `PatternRules[expr, pattern]` would return a list of variable substitutions – expressed in terms of the meta-KB structural specifications.

14 Uncertainty in SetLog

The ability to represent and reason under uncertainty is essential for AI systems. While alternative frameworks such as fuzzy logic (Zadeh, 2023), non-monotonic logic and Dempster–Shafer theory (Shafer, 1992) have been proposed, our system adopts Bayesian probability, aligning with the mainstream approach as presented in (Cheeseman, 1985) and AI textbooks like (Russell and Norvig, 2016).

Uncertainty arises from incomplete, vague, noisy, or ambiguous information. Bayesian probability offers a quantitative framework for representing and reasoning about such uncertain information, but in many cases Bayesian inference must be augmented by the principle of maximum entropy (maxent) to formalize independence assumptions and establish uninformative prior probabilities (entropic priors). This augmented framework provides a basis for uncertainty representation and reasoning, decision making, belief updating, and learning under uncertainty. This section outlines SL's approach to representing probability. For information on probabilistic inference, see Section 17.4.

Bayesian probability has been under development since the mid-18th century, beginning with Reverend Thomas Bayes (Bayes, 1763), and has since evolved into a mathematically rigorous and widely applicable theory of uncertain representation and inference. Meanwhile, modern logic – developed independently, starting with George Boole (Boole, 1847) – has become the standard language for representing and reasoning about information with no uncertainty. One of the core challenges for knowledge representation and reasoning in AI is integrating logic and probability into a single, coherent system. This section shows how probabilistic axioms are integrated into SL.

14.1 Representation of Probabilistic Information

In SetLog (SL), all probabilistic axioms are expressed using a standard axiom form:

```
Prob[agent, Proposition, Evidence, Assumptions] = PDF
```

- `agent` is the agent that believes the proposition to the degree given by the PDF (Probability Density Function). This explicit agent argument allows reasoning about

other agent's beliefs. If the agent is the system itself, then it may be omitted by default.
- `Proposition` is the target proposition whose probability is being asserted.
- `Evidence` is a Boolean expression that represents the conditioning evidence.
- `Assumptions` is the set of assumptions used in deriving the PDF. Typically, these are some sort of independence assumptions.
- `PDF` is a probability density function, which, for the given `agent`, represents the degree of belief in `Proposition`, given `Evidence`.
- `Prob` is the probability function that evaluates to a PDF

This form ensures that all probability statements in SL are dependent on the evidence and assumptions, reflecting the Bayesian view that all beliefs of an agent are degrees of belief of that agent given specific evidence and assumptions. Since the probability axiom form is a Boolean expression, this method of representing probability allows reasoning with the `Prob` function the same way as for any other SL function. This means that our method for integrating probabilistic information with logic does not require changing the underlying logic (SL), but it does require additional inference rules specific to probability (see Section 17.4). Prior methods for integrating logical and probabilistic information, such as probabilistic logic (Nilsson, 1986), and Markov Logic Networks (Richardson & Domingos, 2006), do modify the underlying logic. We note that the Prob function is a special type of function that quotes its arguments so that they are not evaluated during inference.

A probability axiom in standard probability theory has the form:

```
Prob[PropositionP | EvidenceP] = n;   0 <= n <= 1
```

This form differs from the SL form in three major ways:

1. The SL form has the *agent* whose belief is represented as an argument in the probability function.
2. The SL form separates out the *assumptions* from the *evidence* used in determining the PDF.
3. The SL form has a PDF as the value of the Prob function, not a single number as in the standard form. The SL PDF form expresses not just the estimated probability value, but also the uncertainty in that estimate. The use of a PDF as the value of a probability statement was previously advocated by (Ng & Lloyd, 2009).

The SL probability format allows an agent to axiomatize other agents' beliefs, and so supports multi-agent probabilistic reasoning. This agent dependency makes it possible to construct nested agent belief statements, such as:

"Agent A1 believes that Agent A2 believes that Agent A3 does not believe that…"

These nested beliefs are expressed using nested probability axioms, enabling the system to reason about chains of belief, belief disagreement, deception, and other complex multi–agent phenomena. When the agent is omitted in a probability axiom, SL assumes that the AI itself is the default agent.

Because the SL `Prob` function has the evidence and assumptions as arguments, `Prob` axioms with the same `PropositionP` but different evidence or assumptions are not contradictory. This property ensures that our representation of probability axioms preserves logical monotonicity.

As a simple example of the use of SL's representation, consider the standard coin flip situation, where the coin may be biased. In this case, a SL representation of the evidence is: `EvidenceP = FlipDataP`, where `FlipDataP = And[Flip[1] = Head, Flip[2] = Tail,…,Flip[n] = Head]` – i.e. a conjunction of coin flip results. The assumption that the coin flip data is conditionally independent given θ, is represented as `CondIndepP[FlipData, θ]`, where θ is the parameter representing the unknown probability of a random flip resulting in Heads.

In the SL probability axiom form, the target proposition need not be ground – it may be quantified or a conjunction. For universally quantified propositions, probability can be interpreted in two ways: (1) the probability that the proposition is true for all values of its universal variable(s), or (2) the probability that a randomly chosen outcome (e.g., flip[n] = Head) is true. These are distinct questions with different answers. In SL's probability form, they are distinguished by whether the proposition contains universally quantified variable(s) (the *all* case) or an existentially quantified variable (the *any* case).

14.2 What is a Probability Density Function (PDF)?

A probability density function (PDF) describes a distribution over possible values of a quantity–in this case, the (unknown) probability of a proposition given the evidence. For example, a common PDF in Bayesian modeling is the Beta distribution, parameterized by α and β, and used to represent degree of belief over binary outcomes. The Beta distribution is especially useful when the probability is learned or inferred from binary observations (e.g., success/failure data), but the generalized multivariate Dirichlet distribution is also commonly used for non-binary outcomes. Although a PDF can be an arbitrary numeric function, in practice, only a small number of parameterized functions are used, such as the beta distribution, normal distribution, LogNormal distribution, Poisson distribution, etc. Their parameters give a convenient, compact representation compared to an arbitrary PDF. For discreet values, a probability mass function (PMF) should be used instead of a PDF.

In SL, we use a PDF rather than a single number because it reflects both the agent's measure of belief in the target proposition, as well as the uncertainty about the exact value of this measure. This encoding allows easy update of the distribution when new evidence becomes available by using Bayes rule.

In some cases, it is useful to approximate a PDF with a single number, which can be interpreted either as the mean of the PDF or as a delta function centered at that value (i.e., assuming all probability mass is concentrated at a single point). This approximation is accurate in decision-making contexts when the utility function is linear or approximately linear over the range where the PDF has significant mass. In such cases, computing

expected utility using only the mean probability gives a good approximation of the full expected utility obtained by integrating the utility function over the PDF.

14.3 Probability Axioms as Meta-Reified Axioms

It's important to note that SL's probability axioms are reified meta-logical statements: they involve Boolean expressions as arguments to functions, rather than simply expressing facts about a domain. The essential reification step is to allow axioms themselves to become entities in the domain of discourse, and so can be arguments in probability axioms. This step not only allows probability axioms to be represented, but also allows any other information about axioms in a KB to be represented. This includes information such as the provenance of an axiom, when it was added, by whom, how often it is used, etc. Apart from the extra-logical reification step, the resulting meta-level logic does not require any modification to the logic to represent probabilistic information.

For more in-depth discussion of representation of uncertainty using probability, see Appendix 2.

15 Temporal Representation in SetLog

Any AI system needs to be able to represent and reason about dynamic systems. This section describes how this can be done in SetLog (SL). Our approach uses two basic concepts: events and time dependent state representation, as explained in the following subsections. For information on prior approaches to temporal reasoning, see Appendix 3.

15.1 Events

Events are a type of entity in SL that occur in time. Events may be agent-caused (e.g. giving or throwing) or spontaneous (e.g earthquakes). Events mark a change of state from before to after the event. In internal SL, all event entities have a unique name, such as the event: `Coronation_LouisXIV`. Information about this event can be provided by using this event as an argument in qualifying functions, for example:

Example 16.1.1

```
CoronationEventP[Coronation_LouisXIV];
YearOf[Coronation_LouisXIV]= 1654;
MonthOf[Coronation_LouisXIV] = June;
DayOf[Coronation_LouisXIV] = 7;
LocationOf[Coronation_LouisXIV] = Rheims;
MonarchOf[Coronation_LouisXIV] = Louis XIV;
…
```

The information given by the `YearOf`, `MonthOf` and `DayOf` functions can also be represented by a single structured entity:

```
DateOf[Coronation_LoursXIV] = <Date, 7, June, 1654>;
```

An event can be part of an event type-hierarchy:

```
CoronationEventP[event] ==> PoliticalEventP[event];
PoliticalEventP[event] ==> EventP[event];
```

This hierarchy allows type-inferences such as:

```
EventP[Coronation_LouisXIV]
```

Louis XIV's coronation is represented in Example 16.1.1 as an instantaneous event (i.e., having no duration). The choice between representing an event as instantaneous or durational depends on both the available information and the intended use of that event. For distant historical events, only coarse temporal details, such as the century, may be known. The `Reign_LouisXIV` event describing the full reign of LouisXIV, has a duration:

Example 16.1.2

```
ReignEventP[Reign_LouisXIV];
YearOfAccession[Reign_LouisXIV] = 1654;
YearOfDeath[Reign_LouisXIV] = 1715;
MonarchOf[Reign_LouisXIV] = Louis XIV;
DurationYearsOf[Reign_LouisXIV] = 72;
```

Choosing the appropriate level of temporal granularity is a pragmatic and context-dependent aspect of knowledge representation, guided by the precision of available time information and the need for accurate temporal localization. A single event may be modeled as instantaneous in one context (e.g., a birth known to have occurred on a specific day) and as durational in another (e.g., a labor lasting several hours), depending on the informational goals and the resolution of available evidence.

15.2 Time Dependent State Representation

In dynamic systems, some components of a state description can change over time. Time dependent state description axioms are referred to as "fluents" in the literature (McCarthy & Hayes, 1969), the rest are time invariant. We represent fluents by adding a time argument to the corresponding axiom, typically as the last argument, allowing state values to be expressed as functions of time. For example, the position of the white queen in a chess game can be represented as:

```
And[t >= 0, t <= 10] ==> (PositionOf[WhiteQueen, t] = Sq1D);
(t >= 11) ==> (PositionOf[WhiteQueen, t] = Sq4A);
```

The second axiom above does not have an upper bound on `t`, so if the queen is moved again at say `t = 13`, the second axiom must be modified to:

```
And[t >= 11, t <= 13] ==> (PositionOf[WhiteQueen, t] = Sq4A);
```

and the new axiom:

```
(t > 13) ==> (PositionOf[WhiteQueen, t] = Sq5C);
```

must be added.

In this example, t is a universal variable, so these axioms allow specific deductions such as:

```
PositionOf[WhiteQueen, 3] = Sq1D
```

By adding the appropriate temporal axiom every time the white queen is moved, the entire positional history of the queen can be represented.

In the above example, t is a discrete turn variable, represented by a natural number. It can also be regarded as a relative time variable. If the chess game is being played with a chess clock, the moves can be represented using t in continuous metric time. Such a change allows more detailed information to be deduced, such as: “where was the white queen at a particular clock time”.

A quantitative use of metric time is illustrated by the following example representing the position over time of a “throw ball” event.

```
ThrowEventP[Throw1] = True;
InitialSpeedOf[Throw1] = <Speed,10, UnitDiv[M,Sec]>;
ThrowAngleOf[Throw1] = <Angle,0.6, Radians>;
StartTimeOf[Throw1] = <StopwatchTime,0,Sec>;

## Type-axiom for EndTimeOf:
ThrowEventP[event] <==> StopWatchTimeP[EndTimeOf[event]];
## Definition of EndTimeOf:
EndTimeOf[throwEvent] = StartTimeOf[throwEvent] +
     ½*G/(InitialSpeedOf[throwEvent] *
          Sin[ThrowAngleOf[throwEvent]]);

## In EndTimeOf above, the '+' operator above is overloaded
## to work with StopwatchTime structured entities.
## Likewise, the '*' operator is overloaded to work with
## Speed structured entities, and the Sin function is overloaded
## to work with Angle structured entities.

## MovingP predicate tests if ball is moving

## Type-axiom for MovingP:
And[ThrowEventP[event],StopWatchTimeP[t]] <==>
     BoolP[MovingP[event,t]];

## Definition of MovingP
MovingP[throwEvent,t] = And[
     (t >= StartTimeOf[throwEvent]),
     (t <= EndTimeOf[throwEvent])];

## In MovingP above, the argument 't' is a StopwatchTime
## structured entity, and the comparison operators '>=' and
## '<=' are overloaded to compare StopwatchTime entities.
```

```
## Type-axiom for XCoordOfThrow:
And[ThrowEventP[event],StopwatchTimeP[t]] <==>
     RealP[XCoordOfThrow[event,t]];

## Definition of XCoordOfThrow
( XCoordOfThrow[throwEvent, t] =
          InitialSpeedOf[throwEvent] *
          Cos[ThrowAngleOf[throwEvent]] *
          (t - StartTime[throwEvent]) ) <==>
     MovingP[throwEvent,t];
## Type-axiom for YCoordOfThrow:
And[ThrowEventP[event],StopwatchTimeP[t]] <==>
     RealP[YCoordOfThrow[event,t]];

## Definition for YCoordOfThrow
( YCoordOfThrow[throwEvent, t] =
          InitialSpeed[throwEvent] *
          Sin[ThrowAngle[throwEvent]] *
          (t - StartTime[throwEvent]) -
          ½*G*(t - StartTime[throwEvent])^2 ) <==>
     MovingP[throwEvent, t];

## In XCoordOfThrow and YCoordOfThrow above, the argument 't'
## is a StopwatchTime structured entity, and the various
## functions are overloaded as before to support the Angle,
## Speed, and StopWatchTime structured entities.
```

`G` is the earth gravitational constant. In this example, we have used the structured entity representation for units, such as `<Angle,0.6, Radians>`, as explained in Section 16.1. The above SL "throw ball" SL axioms correspond to a physics representation:

x= v*Cos(angle)*t and y = v*Sin(angle)*t – (1/2)*g*t^2.

15.3 Combining Event and Temporal State Representation

Any causal sequence through time can be represented as an alternating sequence of event-state pairs, starting with an initial state. If the effect of the events in this sequence is deterministic, the successive state information can be inferred from the event sequence. For example, a chess game is often recorded as a sequence of moves, leaving the corresponding board state to be inferred. This is not true in general, so a dual event/state representation is usually necessary to represent a temporal sequence.

When SL is used to project potential future event/state information, unless the effects of the events are deterministic, any temporal projection will be a branching tree of possibilities. If the uncertainty created by this branching is sufficient, the current state gives little or no information about future state after many steps. In real-world systems, this loss of information is typically compensated for by sensing.

Because time is a one-dimensional ordered quantity, many different temporal relationships can be defined over time. For example, two durational events can be disjoint, overlap, or one can be contained in another, and so on. These temporal relationships can be defined in SL. This is supported by the *Duration* structured entity, with structured entity recognition predicate:

```
DurationP[d] <==>
     And[SeqP[d], LengthOf[d]=4, d[1]=Duration,
          ClockTimeP[d[2]],ClockTimeP[d[3]],TimeUnitP[d[4],
           d[2] < d[3]];
```

and supporting functions

```
## Type-axiom for StartTimeOf:
DurationP[d] <==> ClockTimeP[StartTimeOf[d]];

## Definition of StartTimeOf:
StartTimeOf[d] = d[2];

## Type-axiom for EndTimeOf:
DurationP[d] <==> ClockTimeP[EndTimeOf[d]];

## Definition of EndTimeOf:
EndTimeOf[d] = d[3];
```

Given the above, the concept of two durations overlapping can be described by:

```
OverlapP[duration1, duration2] <==>
     StartTimeOf[duration2] < EndTimeOf[duration1];

## Type-axiom for OverlapP:
And[DurationP[d1],DurationP[d2]] <==> BoolP[OverlapP[d1,d2]];
```

where the '<' operator has been overloaded for *ClockTime* structured entities.

15.4 Current Time

A common way of measuring time for humans is to fix it relative to the current time. We often say things like, it happened an hour ago, or right now. The problem with this measure of time in formal knowledge representation systems is that t_{now} is not a constant, and depends on the time the inference is being done or the axiom is being entered into the KB. What's more, one cannot logically deduce what t_{now} is unless it is concurrent with another event for which the time is known. In these cases, the agent must go outside the logic to check a clock that keeps track of the time. This is an example of a special kind of instantaneous event called a sensing event.

```
EventP[ReadClock1];
EventTypeOf[ReadClock1] = SensingEvent;
SenseResultOf[ReadClock1]] = < <ClockTime,9,Hour,GMT>,
     <Date,16,January,2025> >;
```

Here the SenseResultOf function returns an ordered sequence consisting of first a *ClockTime* entity and then a *Date* entity.

The result of this ReadClock1 event would allow for one to define t_{now} for inferences done at that time.

16 Miscellaneous Knowledge Representation in SL

16.1 Units

Quantitative properties frequently have units, e.g. mass => Kg, length => miles, etc. SL can represent these via structured entities that combine a numeric value with a unit (see Section 5.3). For example, a MassOf function can take an object and return a structured entity that includes the units as an argument. Consider:

```
MassOf[RosettaStone, Kg] = <Mass, 760, Kg>;
```

The general SL axioms that support this approach are:

```
## Type-axiom for MassOf
PhysicalObjP[obj] ==> MassP[MassOf[obj,u]];

## Definition of MassUnits$ set
MassUnits$ = {Kg, Pounds, Ounces, Grams,…};

## Type-axiom for MassUnitP
BoolP[MassUnitP[u]]; # no constraints on argument

## Definition of MassUnitP predicate from MassUnits$
MassUnitP[u] <==> MembP[u, MassUnits$];

## Type-axiom for MassP
BoolP[MassP[m]]; ## no constraints on argument

## Definition of MassP predicate
MassP[m] <==> And[SeqP[m], LengthOf[m]=3, m[1]=Mass,
     PosNumberP[m[2]], MassUnitP[m[3]];
```

Taking this further, the equation for force (without units) is Force = Mass * Acceleration. This can be represented in SL by using universal variables for the units as needed:

```
Times[<Mass, x, u1>, <Accel, y, u2>] =
     <Force, x*y, UnitTimes[u1, u2] >;
```

The two structured entities *Accel* and *Force* are defined together with predicates AccelP and ForceP, similarly to *Mass* and MassP. These structured entities contain the unit specification. The polymorphic Times function is extended (by this axiom) with structured entities to handle unit multiplication (e.g. N = kg*m/ sec^2), making use of the UnitTimes function which handles multiplication of the units. Note that this approach allows dimensional analysis, since it abstracts away the explicit choice of units.

Conversion between different units is straightforward. Define a function "ConversionFactor" that takes two compatible quantity units and returns the corresponding conversion factor. For example, for *Mass*:

```
## Example conversion
ConversionFactor[Kg, Pound] = 0.453592; ## kg to pounds

## General mass conversion
<Mass, x, u1> = <Mass, x * ConversionFactor[u1, u2], u2>;
```

The inverse conversion is defined as:

```
ConversionFactor[u1, u2] = 1/ ConversionFactor[u2, u1];
```

(note the u1, u2 positions are reversed in the second argument list).

The incorporation of the choice of unit into the structured entities means that two different structured entities can be equivalent. For example, <*Mass*, 1, Pound> is equivalent to <*Mass*, 0.453592, Kg>.

A second way of representing units is permitted in SL. Instead of creating a structured entity such as the MassOf[obj, u] = <*Mass*, n, u> example above, a unit specific function can be used. To measure the mass in kilograms of an object one would write:

```
MassValOf[obj, Kg] = n
```

As the same information is contained in both forms, the question of which to use is pragmatic and depends upon the context in which the user is employing the axiom.

A further generalization can be made concerning the dimensions these unit axioms measure:

```
AllUnits$ = {MassUnits$, LengthUnits$, TimeUnits$, …}
```

The various types of units can be gathered together into a general Quantities$ set:

```
Quantities$ = {Mass, Length, Time, Density, …};
QuantitiesP[q] <==> MembP[q[1],Quantities$];
```

16.2 Frame of Reference

Some measurements such as velocity or weight are relative quantities. A car may be moving at 75 km/h relative to the ground, but relative to the sun is traveling at a whopping 107,283 km/h.

Example 17.2.1

```
SpeedOf[CarA, KPH, Ground]= 75;
SpeedOf[CarA, KPH, Sun]= 107283;
```

For such relative quantities, yet another additional argument is needed to note the frame of reference. If all the axioms in a KB take place in a common frame of reference, the addition of the frame of reference argument can be inferred by default. The translator

which takes input SL and turns it into internal SL will add any default frames of reference (see next section).

Like transformation between units, there are similar rules to transform the representation of points, lines, etc., from one frame of reference to another.

16.3 Default Values

The Input language of SL is engineered for readability. It achieves this, in part, by supporting infix notation with parentheses for disambiguation and permitting flexible naming conventions for commonly used functions. An additional mechanism is the use of default values to avoid unnecessary duplication. In many application domains, functions require multiple parameters to be fully specified. In cases where all or nearly all axioms in the KB assume the same value, SL allows an omission of this argument with a general axiom that directs the translator to add the value back in to all the axioms in the internal SL representation. This reduces the need for explicit specification, streamlining function declarations and enhancing input clarity.

Consider the first statement from Example 17.2.1:

```
SpeedOf[CarA, KPH, Ground] = 75;
```

Many functions share that same frame of reference, so an underscore can be used a stand–in for all axioms using this same frame of reference.

```
SpeedOf[CarA, KPH] = 75;
```

However in cases where the frame of reference differs from the common one, it must be specified explicity. For example,

```
SpeedOf[CarA, KPH, Sun] = 107283;
```

Note that default values only exist in input SL. When the SL expressions are passed to the Translator (see Section 2) the omitted arguments are added using deterministic substitution rules. Furthermore, more than one default value may be used in the same function.

More generally, a default value is analogous to the background assumptions that humans make in communicating with each other. To fully specify every aspect of every concept would be tedious and time consuming, so using defaults reduces the communication overhead. Although similar in name, SL defaults are distinct from Default Logics (Reiter, 1980), since SL defaults are just a shorthand for the full logical form, whereas Default Logics are a poor substitute for representing incomplete or uncertain information.

16.4 Representation of Vagueness (Semantic Uncertainty)

The representation of vague or imprecise information is a significant challenge in translating natural language. While human language is naturally tolerant of inexactness, formal logic systems require precise, well-defined inputs. The issue of representation of vague information and inference using such vague information is well understood in the

field of knowledge representation. Various approaches have been proposed to address it, such as fuzzy logic (Zadeh, 2023), but we have chosen to use probability as our approach to this problem.

Vagueness refers to predicates or attributes lacking clear boundaries – e.g., the statement *"He's tall."* stated in the context where "He" refers to an adult male. Such a description is too indeterminate to be represented directly as a logical axiom. However, such vague information can be modeled using the appropriate conditional probability functions. For example, the "Tall" statement above can be represented by the probability function: P["Tall" | height, adult-male] = f[height], shown in Fig. 2. This graph represents a population consensus view of the probability that a typical member of the population would use the descriptor "Tall" to describe an adult male as a function of that person's height. In principle, each height would yield a PDF, not a single number. Note that this is a probability function (of height), not a probability density function (PDF), so the area under the curve has no meaning. Additional contextual parameters, such as the individual's age, geographical region, or historical period, could further refine this probability estimate.

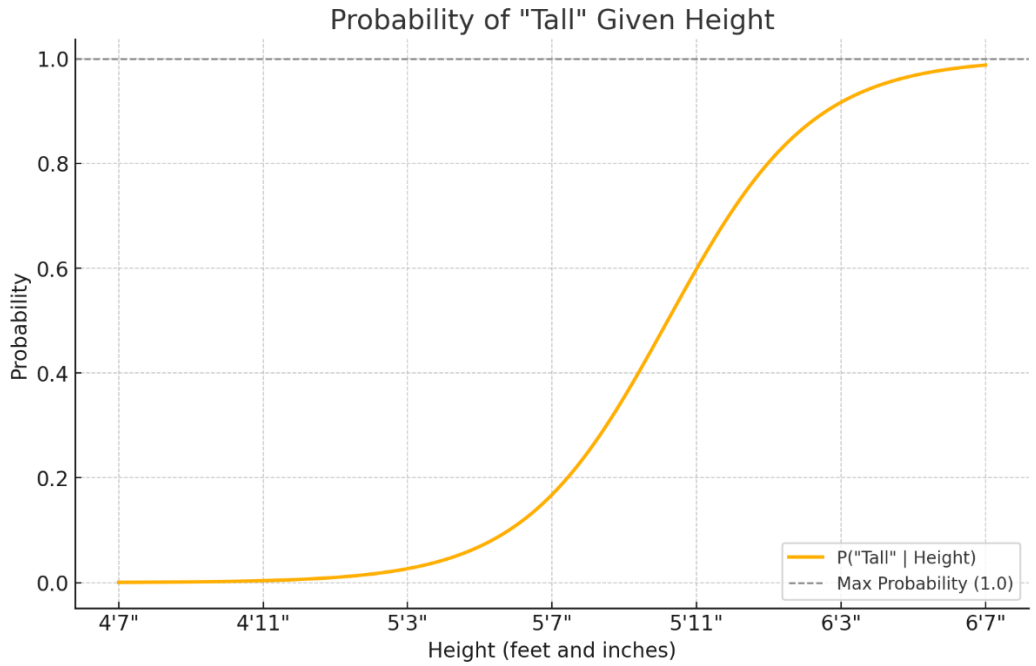

Fig 2 The probability function: (P["Tall" | height, adult-male]). This curve is the result of a statistical average of instances where adult males were described as "Tall" or "Not Tall", given their height.

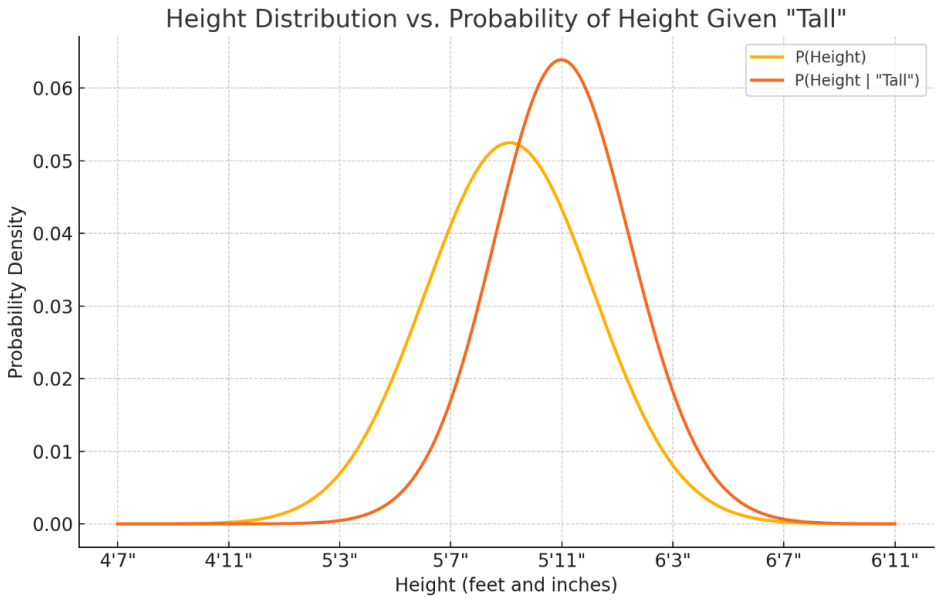

Fig 3 The height distribution of US adult males (in orange) is shifted by the "Tall" description using Bayes' theorem. The red curve is the posterior probability based on the prior yellow height distribution and the likelihood function curve from Fig 2.

To apply Bayes' theorem, the likelihood function shown in Fig. 2 and the prior PDF of an adult's male's height (without the "Tall" information, shown in yellow in Fig. 3) are needed. The result of applying Bayes' theorem to this "Tall" example is:

P[height | "Tall", adult-male] $\propto$ P["Tall" | height, adult-male] * P[height | adult-male]

This formula yields an unnormalized PDF. To convert it to a normalized PDF, the curve needs to be multiplied by a factor that makes the area under the curve = 1, with the resulting posterior PDF shown in red. As can be seen from Fig. 3, the result of the "Tall" information is to shift the height probability density towards greater heights, as well as narrowing the height range. This means that the "Tall" information has given real information but not definitive information about the individual's height.

Other vague descriptors can be treated in a similar manner to the "Tall" example, thus allowing such linguistic descriptors to give information about the underlying quantity in the form of a more informed posterior PDF.

Using SL's probabilistic representation, uncertainty about existence, identity, meaning, etc. can also be represented.

16.5 Missing Information

When dealing with real-world datasets, missing information is endemic. In virtually all the KBs, there will be information that is not explicitly known. In a genealogy KB, for example, if one goes far enough back in time, unknown individuals become common. We know that

some of these individuals had to exist as everybody has a father and mother, even if we don't have identifying information for them. In some cases, we can refer to these individuals by the relations we do know about. For instance, if the mother of `John_Smith` is unknown (due to missing data), we would refer to that individual using the entity: `MotherOf[John_Smith]`. Because `MotherOf[John_Smith]` is unknown, the inference agent would return `MotherOf[John_Smith]` unevaluated or a set of possible values. Even if `MotherOf[John_Smith]` is not identified, she may be constrained by other axioms (e.g. she must be older than `John_Smith`.) This makes `MotherOf[John_Smith]` a constraint defined function.

The result of a query evaluation by the IAgt given a KB is known if the result is a definite answer. Otherwise, it is unknown or partially known (i.e. constrained). There are two cases for failure to answer a query. First, the necessary information might be missing from the KB. Second, there may be insufficient compute resources to arrive at a definite answer (although in principle the necessary information might have been present in the KB).

While in many cases missing information hinders inference, there are instances where the absence of information allows us to draw conclusions. For example, if a home alarm system does not go off, it is good evidence that nobody has entered the house.

16.6 Agent Beliefs

16.6.1 Knowledge vs. Belief

In SL, knowledge denotes statements that are logically entailed or implied by the KB under the SL proof theory; they are categorical and truth-valued. Beliefs capture uncertainty and are modeled probabilistically, consistent with subjective Bayesianism's treatment of rational degrees of belief (Joyce, 2011). Thus, while knowledge is closed under logical consequence, beliefs are graded and update under evidence.

16.6.2 Agent-indexed beliefs

Probability propositions in SL are explicitly agent-relative. Canonically, a belief about a formula ϕ held by agent a under background context/evidence E and assumption set A$, is written as a probability term that returns a PDF (see Section 14.1) – that is:

ProbP[a, ϕ, E, A$] = PDF

Because the agent is an argument, beliefs are inherently agent dependent: the same ϕ may receive different distributions for different agents (or for the same agent at different times) without rendering the global KB inconsistent.

16.6.3 Summarization and acceptance

An inference agent may employ a decision/acceptance policy that maps graded belief to categorical commitments when uncertainty is sufficiently low. A simple, explicit policy is:

- Accept as true for agent a if $u \geq 1 - \varepsilon_T$
- Accept as false if $u \leq \varepsilon_F$

We refer to this as the logical leap. Practically, the acceptance is agent-scoped (i.e., in the agent's working axiom set) rather than merged into the shared, logically entailed core; this preserves global consistency while enabling efficient downstream reasoning. Thresholds ε_T and ε_F can be tuned for risk tolerance, and acceptance is retractable under subsequent updates to ProbP.

16.6.4 Updating beliefs

As new evidence arrives, SL can update the distribution produced by ProbP using the update operators defined in Section 14 (e.g., Bayesian conditioning when evidence is event-valued; Jeffrey-style probability kinematics when evidence is uncertain or comes as constraints on marginals). This keeps belief dynamics aligned with standard probabilistic coherence principles (Jeffrey, 1965; Levi, 1980).

16.6.5 Multi-Agent and Higher-Order Beliefs

Because beliefs are agent-indexed, modeling another agent's beliefs is direct. In Example 16.6.5.1 we show how Agent 1 can model a belief of Agent 2 who is using one evidence and assumption set, where Agent 2 is using a different evidence and assumption set.

Example 16.6.5.1

```
Prob[Agent1, Prob[Agent2, PropositionP, E1, A$], E2, B$]] = PDF;
```

Such higher-order constructs support theory-of-mind tasks (e.g., negotiation, explanation) while keeping contradictions localized to agents rather than infecting the KB.

16.6.6 Contrast with Modal Belief Logics

Classical doxastic logics (e.g., Ba ϕ; Hintikka, 1962) and dynamic epistemic logic (van Ditmarsch et al., 2007) treat belief via modal operators over possible-world models. These frameworks excel at introspection schemas and action-based updates but, in their standard forms, do not encode degrees of belief. SL's probabilistic stance renders graded commitment native: the "strength" of belief is quantified by the returned distribution, enabling statistical decision rules, calibration, and learning. Nothing prevents embedding modal operators into SL if desired; see Section 16.7 for a comparison mapping.

16.6.7 Consistency and Contradiction Management

Because statements of the form ProbP[a, ϕ, E, A$] are distinct from categorical knowledge, two agents a and b may rationally hold opposing high-confidence beliefs about ϕ based on different evidence. Because each agent maintains their own separate KBs, this belief conflict does not produce contradictions

In summary, SL distinguishes categorical knowledge from agent-indexed, probabilistic beliefs. Beliefs return distributions, support principled updates, and permit agent-scoped acceptance ("logical leap") when uncertainty is low. This captures the graded nature of belief championed in Bayesian epistemology (Joyce, 2011) while maintaining global logical hygiene, and it affords straightforward modeling of multi-agent and higher-order beliefs –

capabilities that are cumbersome in purely modal treatments (Hintikka, 1962; van Ditmarsch et al., 2007).

16.7 Modal Logics and SL

SetLog does not adopt modal operators. Instead, modal notions such as necessity, belief, and possibility are expressed directly as axioms, using the same syntax as all other SL axioms. The motivation for this design choice is practical: introducing additional operators beyond ∀ and ∃ increases the complexity of inference and imposes computational overhead across the entire knowledge base, even though only a small fraction of axioms would exploit these operators. In standard modal logics, two operators are typically introduced: necessity (□) and possibility (◇). Formally, their semantics are defined relative to an accessibility relation R between possible worlds. Necessity is expressed as:

$$\Box\phi(w) \equiv \forall v\,(R(w,v) \rightarrow (\phi,v))$$

which states that ϕ holds in all worlds v accessible from world w. Its dual, possibility, is expressed as:

$$\Diamond\phi(w) \equiv \exists v\,(R(w,v) \wedge (\phi, v))$$

which states that there exists at least one accessible world v in which ϕ holds. The structure of the accessibility relation determines which modal system applies. For example, System T requires reflexivity of R (∀w R(w,w)), ensuring that if something is necessary, then it is true in the actual world. System S4 requires both reflexivity and transitivity (∀u,v,w (R(u,v) ∧ R(v,w) → R(u,w))), ensuring that what is necessary remains necessary in further accessible worlds. System S5 requires reflexivity, symmetry, and transitivity, collapsing all worlds into a single equivalence class of accessibility, so that necessity means "true in all worlds" without restriction.

Two common use cases for modal logic illustrate this framework. The first is branching time reasoning, where each world corresponds to a temporal state and R(w,v) indicates that v is a possible successor of w. Here, ◇Event expresses that "it is possible that an event occurs in the future," while □Event asserts that "the event necessarily occurs in all future timelines." The second is epistemic reasoning about agents' beliefs, where $R_a(w,v)$ encodes the set of worlds agent a considers possible from world w. In this setting, $\Box_a\phi$ means "agent a believes ϕ," since ϕ holds in all worlds accessible to a. For instance, □Alice ϕ states that Alice believes ϕ, while ◇Bob ψ indicates that Bob considers ψ possible in at least one world consistent with his knowledge.

SetLog avoids the introduction of □ and ◇ by anchoring inference to a single knowledge base (see Section 12). Once the domain of discourse is fixed to the entities defined in that KB, the distinction between universal quantification within a world and modal quantification across worlds collapses. Moreover, because all definitional axioms in SetLog are stipulated as true within the KB, they are necessarily true by construction: there is no distinction between "true in this world" and "true in all accessible worlds." Modal concepts such as necessity, possibility, or belief can therefore be represented directly as

axioms, without additional operators, while still allowing branching-time or epistemic structures to be modeled explicitly by axiomatizing accessibility relations when needed.

An example of SL axioms representing modal necessity is:

Example17.7.1
Proposition: Checkmate is not possible given a board state (e.g., King and Knight vs. King).

English: No checkmate is possible.
Modal Logic: □NotMateable(CurrentState)
Predicate Calculus: ∀BoardStates (R(BoardStates, CurrentState) → NotMateable(BoardStates, CurrentState))
SetLog:
```
LegalMoveP[boardstates, CurrentState] ==>
    NotMateableP[boardstates];
```

In modal logic, the necessity operator (□) applied to the predicate expresses that there is no accessible world in which NotMateable is false given the current game state. In other words, across all reachable worlds (board states), checkmate cannot occur. Predicate calculus requires the accessibility relation to be represented explicitly as *R*. Universal quantification then ranges over all possible worlds (i.e., all board states). The formula states that for every board state, if there is an accessibility relation between that state and the current one, then NotMateable holds in that state. For possibility, ◇, a similar construction can be written to assert that a Mate is possibility in at least one world.

In SetLog, the generalized accessibility relation *R* is replaced by the domain-specific predicate `LegalMoveP`, which encodes the rules of chess governing which states can follow from a given state. The axiom then reads: if a board state results from a legal move applied to the current state, then checkmate is not possible in that resulting state. This illustrates SetLog's design philosophy of avoiding modal operators by embedding accessibility directly in domain-specific predicates, thereby reducing inference complexity while retaining expressive power.

16.8 Separation of Logical and Control Knowledge

Logic programming and formal systems, such as typed lambda calculus implicitly assume that they are both a logic and a programming language. In SL, control knowledge is logically separate from domain knowledge. The typed lambda calculus does not explicitly represent this control information and so expressions in the typed lambda calculus cannot be interpreted as programs, unless a particular evaluation procedure is assumed. Typed lambda calculus and logic programs are neither logic nor a complete model of computation and so are essentially of limited utility for each role separately or together. We avoid this issue in our approach to SL by completely separating logical information from the control information. It is at the computational level that logic and control knowledge is used to compile executable code.

16.9 Properties of Higher Order Entities in SL

LengthOf returns the element count of a Sequence, while **SizeOf** returns the element count of a Set (this is "top level" only, not counting elements contained in nested levels).

```
LengthOf[<A, B, C>] = 3;
SizeOf[{A, B, C, D}] = 4;

## Type-axiom for LengthOf
SeqP[s] <==> PosIntegerP[LengthOf[s]];

## Type-axiom for SizeOf
SetP[s] <==> PosIntegerP[SizeOf[s]];
```

In internal SL, all functions have one argument. However, some functions have a sequence as the argument, which corresponds to functions with multiple arguments in input SL. The function **ArityOf** returns the effective number of arguments. That is, in the internal SL, it returns 1 for functions without a sequence argument, or the length of that sequence argument otherwise. For example,

```
## SalesOf[Celica, 2030]
## Type-axiom for SalesOf:
And[CarModelP[model],YearP[year]] <==>
     IntegerP[SalesOf[model,year]];

ArityOf[SalesOf] = 2;
```

16.10 Minsky's Frames

Minsky introduced the idea of frames (Minsky, 1974) to represent knowledge that didn't fit into the knowledge representation paradigms of the time. The frame-based approach has been widely used in subsequent AI KRR (Lassila & McGuinness, 2001), but the relationship between frames and logic-based knowledge representation is not clear from the literature. Here, we show how the information in frame-based representations can be represented in SL as higher-order axioms.

An example to illustrate the use of frame-based representations, consider a restaurant event entity: ResturantVisit6. Information about this event can be represented in SL by qualifying axioms. For example:

```
TimeOf[ResturantVisit6] = <ClockTime,18,Hours,GMT>;
RestaurantOf[ResturantVisit6] = "Cove Café";
CostOf[ResturantVisit6] = <Price,126,USD>;
```

The corresponding restaurant Minsky frame is a generalization over specific restaurant visit events. In SL, the specific case above can be represented as a structured entity (see Section 5.3):

```
<RestaurantVisit,<ClockTime,18,Hours,GMT>,"Cove Café",
     <Price,126,USD> >
```

with the corresponding structured entity recognition predicate

```
RestaurantVisitP[rv] <==> And[SeqP[rv], LengthOf[rv]=4,
     rv[1]=RestaurantVisit, ClockTimeP[rv[2]], StringP[rv[3]],
     PriceP[rv[4]]>];
```

This predicate describes the Minsky restaurant visit frame, specifying the type of the arguments (slots) in a specified order that can occur in describing any particular restaurant visit event. This example makes clear that a Minsky frame in SL is a structured entity recognition predicate.

This representation allows hierarchy, since structured entities can contain other structured entities.

A problem with both the Minsky frames and any single structured entity representation is how they represent missing information. In the SL atomic axiom approach, if the cost of `RestaurantVisit6` is not known, then no `CostOf` axiom is added to the KB. In the frame approach, there is a "slot" for the cost information, whether it is known or not. A commonly used ad hoc solution in a frame representation is to use a special symbol in any slot where the expected information is missing. This approach requires special handling by any IAgt to respond appropriately to such a special symbol.

The concept of Minsky frames includes other generalized information, such as how probable it is that the information for a particular slot is available. Another form of probabilistic information that can be represented in a Minsky frame is distribution over the values for a particular slot, such as the distribution over Price inferred from specific Prices. For example,

```
RestaurantVisitStatsP[rv] <==> And[SeqP[rv], LengthOf[rv]=4,
     rv[1]=RestaurantVisitStats, FrequencyP[rv[2]],
     FrequencyP[rv[3]], FrequencyP[rv[4]]>];
```

where `FrequencyP` is a predicate recognizing a frequency value.

17 Inference in SetLog

At the **logic level**, inference consists of an Inference Agent (IAgt) applying rules of inference to generate new axioms until the desired result is found or the attempt to prove the target proposition fails. Automated theorem provers work this way. All inference agents have a major search problem because of the large number of possible inference rules that could be applied at each step. At the **procedural level**, inference is achieved by the execution of code compiled from the axioms (see Fig. 1 and Section 17.3). This code is guaranteed by the compiler to produce the same answer as would be obtained by doing inference at the logic level, but does so more efficiently.

Inferring *probabilistic* information from a KB uses the same rules of inference as for logical deductions, but these rules are augmented with additional probabilistic inference rules. In particular, the IAgt uses Bayes Rule and the principle of maximum entropy, as described in Section 17.4, to do probabilistic inference.

17.1 Logical Inference Overview

At the logic level, all inference based on SL can be summarized by the following schema:

Inference Schema: `Eval[KB ==>(Fn[arg] = ans)] = True]`

where `KB = And[axiom(1),…,axiom(n)]`, and all axioms are Boolean.

The inference agent is tasked with finding the simplest possible value for the unknown variable `ans`, as implied by the KB. Evaluating the inference schema returns the value(s) for `ans` that makes the entire conjunction `True`. Any newly inferred axioms/theorems generated by the IAgt can be added to the KB as additional axioms for use in future inference steps, even though they are redundant. Our schema is more general than theorem proving because `ans` could be `True` or `False` or any other type of value as dictated by the return type of query function. In principle it is possible to have non-conjunctive logical forms for a KB, but any such alternative forms are assumed to have been transformed into conjunctive form when building a KB, by using the standard logical equivalence rules.

In this schema, the KB does not have *be* true in any absolute sense–the schema merely makes the inferred value of `ans` conditional on the truth of the KB. This makes hypothetical inference possible by *assuming* a given KB is true. The inference schema is meta-level because the KB is treated as an entity. This schema also hides the fact that inference is inherently a procedure executed by an IAgt, thus making inference inherently temporal. There must also be a querying agent that specifies the KB and target function to be evaluated by the IAgt. All formalizations of inference that we are aware of do not include these agents or the temporal dependence as part of the inference formalization, yet they are an essential part.

In some cases, there is insufficient information in the KB to deduce a definite answer, but partial knowledge may be uncovered by the inference process. A possible return value in such cases is a disjunctive answer. For example, `FatherOf[Bill]` might return: `Xor[ans = John, ans = Simon]`. In other cases, a conditional answer may be the most informative answer that can be returned. For example, the query:

```
Integrate[x^n, x] = ans
```

would return the conditional result:

```
And[((n = −1) ==> (ans = Log[|x|]) ),
     ((n != −1) ==> (ans = x^(n+1)/(n+1))]
```

In the extreme case where the KB has no information about the query, the IAgt returns the query unevaluated. For example, if nothing is known about `FatherOf[Bill]`, the IAgt returns `FatherOf[Bill] = ans`. Such an unevaluated answer is a signal to the querying agent that the KB is uninformative about the query, or at least that the IAgt is unable to find an informative result. In all cases, the IAgt returns the most specific answer it can find given the amount of search it devoted to the task.

17.2 Methods for SL Inference at the Logic Level

In this section, we outline, by example, how a SL inference agent (IAgt) would operate.

Example 17.2.1: Given a particular genealogy KB and the query: `MaleP[Robin] = ans`, the IAgt might match this query against the axiom: `MaleP[Robin] = True.` It would then add the new axiom: `ans = True` to the KB, then terminate. If no match is found, the input query would be returned with no new axiom for `ans`. This inference is a simple example of constraint satisfaction, where the unknown variable (`ans`) is constrained by the KB to the value `True`. We note that if the KB contained the axiom `MaleP[Robin] = False` instead, the new axiom would be: `ans = False`, showing that our IAgt is not trying to prove: `MaleP[Robin] = True`, as in standard theorem proving, but instead is more generally trying to determine the truth value of the Boolean query as determined by the KB – i.e., `True, False or Unknown.`

Example 17.2.2: Consider giving the IAgt a genealogical KB and the query: `FatherOf[Bill] = ans`. Again, the IAgt tries to match the query against the KB and we assume that the match: `FatherOf[Bill] = John` is found. As a result of this match, the IAgt adds the axiom: `ans = John` to the KB and stops. In this example, the result of the inference is a person, not a truth value, as in Example 18.2.1. This is possible because SL is a function-based language, and functions can have values other than truth values. More complex inferred values include sets, plans, programs, proofs, etc. These complex inferred entities are assigned a name (`ans`) and the structure of these complex entities is defined by the axioms added to the KB during the inference process. These additional axioms can define arbitrarily complex structures, so there is no restriction on the complexity of an inferred answer.

Example 17.2.3: [Classic syllogism]: Assume the KB contains the general axiom : `((ManP[x] = True) ==> (MortalP[x] = True))`, and the ground axiom: `ManP[Socrates] = True`. If the query is: `MortalP[Socrates] = ans`, the IAgt's pattern matcher matches this query with the general axiom to give the binding: `x` → `Socrates`. With this binding, the IAgt creates a new goal: `ManP[Socrates] = True` from the implication. This new goal matches the second axiom, making the implication true, with the result: `ans = True`. This is a simple example of modus ponens chaining, where one goal creates one or more new goals which are evaluated in turn. In the logic literature, such an inference is described by the meta-logical expression: `KB ⊢ (MortalP[Socrates]= True)`. Here, the ⊢ symbol is used to denote what logicians call syntactic inference.

Example 17.2.4: Assume the IAgt is given a genealogical KB and the query: `((MembP[x, GreatGrandParentsOf$[Bill]] ==> (AgeAtDeathOf[x] < 50)) = True)= ans)` – i.e. did all of Bill's great grandparents die before 50 years of age? In this case, the query involves the universal variable `x`, so the IAgt has to test every possible value of `x`. If `x` is not a great grandparent of Bill, then the implication is automatically true, so only those values of `x` which *are* great grandparents of Bill need to be considered by the IAgt. Each such value of `x` generates a ground implication which is

true iff that great grandparent died aged less than 50 years old. A single exception means that `ans = False`. If all the ground implications evaluate to True, then `ans = True`. In performing this inference, the IAgt uses repeated applications of modus ponens for each ground axiom, as well as using model checking to determine if the universally quantified query is true for all possibilities.

If, for example, the age at death for Bill's great grandparent Mary is Unknown, then the IAgt returns the conditional answer: `( AgeAtDeathOf[Mary] < 50) ==> (ans = True) & (AgeAtDeathOf[Mary] >= 50) ==> (ans = False).`

17.2.1 Hidden Constraint Satisfaction Problems

An example of a hidden constraint satisfaction problem is "The Australian State Coloring Problem" from: Artificial Intelligence: A Modern Approach (Russell & Norvig, 2016). Here is a complete SL representation of this problem:

```
## Map Colors
Colors$ = {Red, Green, Yellow};
ColorP[x] <==> MembP[x, Colors$];

## Type-axiom for ColorP
BoolP[ColorP[x]];

## States
States$ = {WA, SA, NT, Vic, NSW, QLD, Tas};
StateP[x] <==> MembP[x, States$];

## Type-axiom for StateP
BoolP[StateP[x]];

## Borders
Borders$ = {{WA, SA}, {WA, NT}, {SA, NT}, {SA, Vic}, {SA,
     NSW},{SA, QLD},{NT, QLD}, {QLD, NSW}, {NSW, Vic}};
BorderP[x] <==> MembP[x, Borders$];

## Type-axiom for BorderP
And[SetP[x], (SizeOf[x] = 2), (And[MembP[u,x], MembP[v,x]]  ==>
     And[StateP[u],StateP[v]])] ==> BoolP[BorderP[x]];

## Bordering Test
BorderingP[x, y] <==> MembP[{x, y}, Borders$];

## Type-axiom for BorderingP
And[StateP[x], StateP[y]] <==> BoolP[BorderingP[x, y]];

## ColorOf function
ColorOf[Vic] = Green;

## Type-axiom for ColorOf
StateP[x] <==> ColorP[ColorOf[x]];
```

```
## ColorOf constraint
BorderingP[x, y] ==> (ColorOf[x] != ColorOf[y]);
```

Assume that the IAgt is given the goal: Eval[`ColorOf[WA]`]. The only matching axiom for this goal is the last one above. Matching the goal against this axiom gives the new axiom:

```
BordersP[WA, y] ==> (ColorOf[WA] != ColorOf[y])
```

where y is a universal variable. For this axiom to give information about the goal, the condition must be true. Given the bordering relations, the only values for y for which this is true are `y=SA` and `y=NT`. This is an example of two applications of universal instantiation by the IAgt. With these substitutions, two specific unconditional constraints are added to the KB as new axioms: `ColorOf[WA] != ColorOf[SA]` and `ColorOf[WA] != ColorOf[NT]`.

Next the IAgt tries to evaluate: `ColorOf[SA]` and `ColorOf[NT]` because this information is needed to constrain the initial goal. Since the only matching axiom for these goals is again the main constraint, each of these goals will, in turn, generate more constraint axioms: `ColorOf[SA] != ColorOf[NT]`, `ColorOf[NT] != ColorOf[QLD]`, ……, `ColorOf[SA] != ColorOf[Vic]`. Since the color of Vic is known (green), the last axiom constrains the possible colors of the neighboring states. At this point of the inference procedure, the IAgt has used modus ponens and universal instantiation to uncover a discrete constraint satisfaction problem (CSP). We refer to the original problem as a hidden CSP, because inference was required to uncover the relevant ground constraints. Usually, in CSPs, the constraints are given, not uncovered. A similar process of uncovering and solving CSPs is used in constraint logic programming.

Once the IAgt has exposed a CSP, it must try to solve it. In this case a solution would bind `ans` to the color of WA revealed by the solution. One possible solution method is for the IAgt to call an existing CSP solver to return the desired answer. An alternative solution method is for the IAgt to apply standard inference rules to the CSP. These inference rules include constraint simplification/manipulation. Gaussian elimination is an example of such a method. Even if the constraint simplification does not produce an answer, it can reduce the cost of applying subsequent assignment methods. Partial assignment methods use the case splitting rule. This rule breaks a problem into a disjunction of subproblems by assigning some variables a possible value and creating new reduced CSPs – those with the assigned value(s). This process can be applied recursively to solve the original problem.

An extreme version of the case splitting method is to assign a value to all unknown variables then check the resulting substituted constraints for consistency. As we have mentioned above, KBs represent what is known about the subject being modeled thus no unknown variables occur. However, when an agent is doing inference, unknown variables can occur while evaluation is taking place. Not all evaluations will provide complete knowledge. In these cases only partial knowledge will be added to the KB. This is repeated exhaustively for all possible combinations of assignments to the unknown variables. If any of these assignments make all the constraints true, then the CSP is satisfied by the

corresponding assignment. This extremely inefficient method is called model checking or semantic[6] inference in the logic literature and is given its own symbol (⊨) to distinguish it from other ("syntactic") methods of inference represented by the symbol (⊢). The partial assignment method outlined above mixes constraint simplification (syntactic inference) with case splitting assignments, to give an efficient inference method that uses both syntactic and model checking methods. In our view, *all* logical inference is application of inference rules to symbolic expressions, so having separate symbols (⊨ and ⊢) for particular procedures seems pointless.

Another inference rule we use is the process of elimination. When an unknown variable is constrained, this reduces the set of its possible values. If the possible values are reduced to a singleton set, then the sole value in this set can be returned as the answer. The process of elimination is relatively simple in SL because it includes sets as primitive entities.

17.2.2 Inference Termination

An important question about logical inference is what criterion should the IAgt use to decide when to stop? One criterion is that `ans` should be as simple as possible. Unfortunately, there is no universally agreed definition of "simplicity". One version of simplicity is "syntactic" simplicity – that is the `ans` with the fewest number of symbols in its syntactic form. While this criterion agrees with intuition in many cases, it is not universal. For example, consider the query:

```
Integrate[Sqrt[x^2 + 1], x]
```

with the result

```
(1/2)*x*Sqrt[x^2+1] + (1/2)*Log[Abs[x + Sqrt[x^2+1]]]+C.
```

Most would not regard this result as syntactically simpler than the unevaluated query, but it is what is normally regarded as the desired answer. This example points to a more nuanced definition of syntactic simplicity, where some symbols are weighted more heavily than others in the symbol count. Here, the symbol `Integrate` would have a large weight, so that the given result would be regarded as simpler.

Another possible termination criterion for the IAgt is finding the `ans` that minimizes expected computation in future uses of `ans`. Unfortunately, this criterion is hard to operationalize as the future uses of `ans` are unknown in many cases. Other approaches are to maximally reduce Shannon entropy or to maximize the informativeness of the result, but again, this criterion is difficult to operationalize.

Yet another termination criterion is to terminate when the IAgt reaches a fixed point – i.e. where the IAgt has exhausted the inference rules that could further reduce the target

[6] Even though logicians refer to model checking as "semantic" inference, it is in fact purely syntactic (symbol manipulation) and has nothing to do with meaning. We believe that the only way to connect symbols in the computer to the real world is via sensing, an operation we do not address here.

query. This is the idea behind the lambda calculus as a model of computation. A related stopping criterion is when rule application produces cyclic behavior. Such behavior may be an indication of circular definitions or a signal that no further reductions are possible.

We find none of these criteria entirely satisfactory, so we push the inference termination criterion onto future IAgt implementers.

In summary, logical inference in SL, given a KB, consists of repeated application of inference rules until the desired result is reached or the inference attempt fails. The inference rules are the same as in standard logic, but with two new rules: case splitting and process of elimination. The lack of search guidance for any such IAgt makes fully automated inference infeasible. For this reason, we have developed a compiler (see Section 17.3) that adds control knowledge to the logical specification to speed up inference.

17.2.3 Separation of Logical and Procedural Inference

In our approach to KRR we assume the total separation of logical information (as axioms in KBs) from procedural information (embedded in executable code). This approach separates the "what" from the "how", and allows logical reasoning without procedural considerations interfering with logical inference. The Logic Programming community fails to make this separation, with the result that a logic program is neither logic, nor does it yield maximally efficient procedures.

An important property of this separation is that control knowledge itself can be represented declaratively using a combination of logic and probability. For example, the order of evaluation of the arguments in a conjunction can have a major effect on the evaluation cost of that conjunction, so control knowledge would specify an optimal order. Because control knowledge is represented in SL the same way any other knowledge is represented, it can be learned and reasoned about using the same methods as for any other KB. This control knowledge is typically used by an interpreter or compiler (see next section and Fig. 1).

17.2.4 Inference with Incomplete or Missing Information

An Inference Agent (IAgt) must be capable of explicitly representing its epistemic state – what it knows and what it does not know – at every stage of the inference process. This requirement holds even when the underlying knowledge base (KB) is complete: during inference, some function calls may be provisionally marked as unknown, but later in the inference process become known. Consequently, the IAgt's knowledge is dynamic, evolving as new information is uncovered during the inference process. In many cases the outcome of inference is not complete certainty but partial knowledge, which must be both conveyed to the user and, where appropriate, stored in the KB (see Section 12.10).

17.3 SetLog Compiler / Procedural Inference

Syntactic inference at the logic level is inherently slow. The SL compiler accelerates this inference by generating efficient code in a target language such as C or LISP. In essence, it

applies rewrite rules to SL axioms, transforming logical specifications into procedural code. Multiple definitional axioms for a function are merged into a single runtime procedure. Examples of rewrite rules include turning implications into if-then-else statements, turning logical expressions with multiple universal variables into nested iterations over domains inferred from type- axioms, and choosing the order of evaluation of components of a conjunction or disjunction.

At a higher level, the rewrite rules embody control knowledge, producing deterministic code from purely logical descriptions. Examples of control knowledge include heuristics for guiding search, efficient order of evaluation of disjuncts and conjuncts, and turning recursion into iteration. The SL compiler also makes use of standard compiler optimizations such as inlining, memorization, etc. This compilation of strictly logical specifications into procedural code is a distinctive feature of SL.

Not all functions have explicit definitional axioms. Some are defined indirectly through constraints (see Section 17.2.1). These cases also fall within the SL compiler's scope, which can invoke a CSP solver when needed.

In principle, the logic used to convert axioms into procedures can itself be expressed in SL. Explicit representation of control knowledge in SL allows learning and inference about control independent of its embedding in code. This would enable the compiler to compile itself – opening the door to recursive self-improvement.

17.4 Probabilistic Inference

Probabilistic inference is fundamentally different from logical inference. In particular, logical inference can apply millions of logical steps and still produce a true or false result. In contrast, probabilistic inference probabilities typically converge to prior probability values after a few probabilistic inference steps. Also, a single negative case will make a universally quantified logical expression false, but in probability it may only lead to a small adjustment of a probability value.

As described in Section 14.1, a probability proposition in SL is of the form:

```
Prob[agent, Proposition, Evidence, Assumptions] = PDF
```

The IAgt can use such Prob axioms if the arguments match, or it could be given a Prob query. In the query case, the IAgt may have to choose the evidence and assumptions to get an answer, and the results of such an inference can be added to the KB in the Prob format above. In this section we outline how an IAgt can derive such probabilistic answers.

Since the Prob function is a function like any other in SL, the logical rules of inference apply to it also. This means that all the machinery for pattern matching, term rewriting etc. that are used in inference apply to Prob function inference as well. In addition, the following axioms become rule of inference applicable to Prob functions only:

1. **Bayes' Theorem**: This fundamental rule allows for the updating of probabilities based on new evidence. It states that: $P(A|B) = P(B|A) \cdot P(A) / P(B)$ where $P(A|B)$ is the

posterior probability, P(B|A) is the likelihood, P(A) is the prior probability, and P(B) is the marginal likelihood.

2. **Law of Total Probability**: This rule helps in calculating the total probability of an event based on different conditions: $P(B)=\sum_i P(B|A_i)\cdot P(A_i)$ where A_i are mutually exclusive events that cover the entire sample space.

3. **Independence**: If two events A and B are independent, the probability of both occurring is the product of their individual probabilities: $P(A \text{ and } B)=P(A)\cdot P(B)$ This simplifies calculations in many probabilistic models.

4. **Marginalization**: This involves summing or integrating over a variable to find the marginal probability of another variable: $P(X)=\sum_Y P(X, Y)$ or for continuous variables, $P(X)=\int P(X,Y)dY$

5. **Chain Rule**: This rule is used to express the joint probability of a set of variables as a product of conditional probabilities:

   $P(X1,X2,\ldots,Xn)=P(X1)\cdot P(X2|X1)\cdot P(X3|X1,X2)\cdots P(Xn|X1,X2,\ldots,Xn-1)$

6. **Conditional Independence**: If two events A and B are conditionally independent given a third event C, it means knowing B does not provide any additional information about A when C is known: $P(A|B,C)=P(A|C)$ or $P(A,B|C) = P(A|C)*P(B|C)$.

7. **Principle of Maximum Entropy (Maxent)**: Find the joint probability distribution that maximizes the entropy of the joint subject to any constraints. Using Maxent as a probability inference method makes the assumption that the constraints used in the inference are complete – i.e. no other constraints are operating. If this assumption is false, the Maxent predictions will deviate significantly from their Maxent value, signaling that there are missing constraint(s).

In Prob cases, the IAgt has to make decisions on what evidence to use and what assumptions to make in order to apply rules like Bayes rule and Maxent. Generally, the more relevant evidence that is used, the more the Prob estimate is likely to match reality. However, there is a cost to obtaining evidence and computing Prob estimates from this evidence, so the IAgt has to make decisions that trade off cost against accuracy.

A set of Prob axioms can be combined into a structure known as a Bayes net. This network representation allows inference about the probability of Probs (nodes) in the network given any combination of evidence, and the network itself encodes information about conditional independence between the Probs. Algorithms for learning and doing inference on Bayes nets are well studied (see Jordan, 1999). These nets can be thought of as compiled probabilistic information in a form suitable for probabilistic inference.

Having answered a Prob query, the IAgt can add the resulting Prob proposition to the KB as a new axiom, to avoid having to recompute it. If this information is stored in a dynamic KB, new evidence may make prior Prob estimates obsolete. However, Prob axioms for the same target proposition, but with different evidence are not contradictory. Such cases

raise the problem of how to combine such separate probability axioms into a single axiom that takes all the evidence into account.

Another issue that is fundamental to probabilistic inference is that in practice, there is insufficient evidence to compute simple conditional probabilities. For example, to compute the conditional probability that a particular patient would survive a particular operation, there would have to be a matching set of patients of the same age, medical condition, gender, medical history, weight, etc. Typically there are insufficient matching patients with all the same characteristics to compute reliable conditional probabilities. To cope with this fundamental problem, it is necessary to make conditional or marginal independence assumptions. The graphical structure in Bayes nets is an example of a method for making conditional independence assumptions in a form that allows probability inference. Maxent is a generalized principle of independence that reduces to marginal or conditional probability independence forms when the dependencies form a tree-like structure, but can produce joint probability estimates numerically for arbitrary dependency structures (see Cheeseman, 1983). In essence, maxent distributes the uncertainty as evenly as possible given the constraints.

18 Innovations in SetLog

Standard logic was developed by logicians mostly concerned with meta-logical or foundational questions, such as soundness and completeness of the logic. To simplify these meta-logical proofs, the logics that were developed were stripped-down to what was regarded as the minimum necessary. When AI began using logic as a representation and reasoning language, it adopted the formalism previously developed by logicians, even though the stripped-down nature of this formalism is not well suited to the task. In particular, FOL, the language of choice for AI (Russell & Norvig, 2016) does not include sets or numbers as primitives but instead treats these as ex-logical constructs.

This section summarizes SL's innovations as described in detail in the body of this paper. These innovations are driven by practical needs in using SL as a knowledge representation language. Although these innovations are new to us, we would not be surprised if in some cases we have sometimes re-invented the wheel. These innovations in no particular order are:

1. **Integration of probability with logic** (see Section 14). Probability propositions are meta-logical axioms in SL, and the inference agent is augmented with additional rules of inference, particularly Bayes rule and Maximum Entropy, (see Section 17.4).
2. **Sets and numbers as primitive entities**, along with the corresponding primitive functions, such as Union, Subset, Size, Add and Multiply. These additions make quantitative representation and reasoning relatively easy.
3. **Set-Predicate duality** (see Section 8.2). Sets are defined by their membership function, and the members are those for which the defining predicate is true. This means that sets, such as "people born after WW2" will be different in different

genealogy KBs. This differs from the standard logical view that sets are defined by their members, so that these WW2 sets would be regarded as different. In SL, sets from different KBs are unrelated, but may be generated from the same general logical definition.

4. **Open and Closed Sets** (See Section 8.3). SL allows complete or partial knowledge about set membership to be represented and used in inference.
5. **SetLog is a higher Order Logic**. In SL, sets and functions are 1st class entities and so can be quantified over. This greatly enhances the expressiveness of SL relative to FOL.
6. **No Distinction between Predicates and Functions** (see Section 6.1). In SL, predicates are just functions that return Boolean values. This differs from FOL, where predicates are distinguished from functions, but similar to Boolean algebra.
7. **Interconversion of Functions and Relations** (see Section 6.2). This conversion is just syntactically picking out some args of a relation as input, and others as output. Regarding some arguments as inputs and the rest as output is a procedural interpretation, but logically there is no input or output, just syntactic relationships between arguments.
8. **Sequences as Functions** (see Section 6.4). Sequences in SL are just functions with an integer argument, where the integers are sequential. The ordering information in a sequence function is a result of the ordering of its integer argument. In contrast, FOL needs extra-logical information to represent order.
9. **Explicit Quantifier Elimination** (see Section 10). The symbols ∀ and ∃ are eliminated from SL syntax. In order to do this elimination with full generality, existential quantification is split into two forms: unique existence and general existence. In SL, these two forms of existence are represented differently.
10. **Relativization of Universal Quantification** (see Section 10.2). In standard FOL, universally quantified variables range over the entire domain of discourse. In SL, universally quantified variables only range over a defined subset of the domain of discourse. This is similar to Henkin Semantics (Henkin, 1973).
11. **Eliminating Modal Operators:** (see Section 16.7). SetLog's design removes the need for quantifying over accessible possible world by using KBs to localize inference. This provides several useful benefits. First, it eliminates the need for Kripke semantics as all axioms are localized to a given possible world defined by KB the axioms reside in. Second, in combination with type-axioms, KBs provide a well-defined range of values for universal variables to iterate over.
12. **Type Information in SetLog**: (see Section 7). Unlike Simple Type Theory as used in system like Bach (Lloyd, 2007), type information in SL is in the form of type-axioms using the same syntax as for regular axioms. This means that type inference uses the same inference machinery as any other inference in SL.

13. **Organization of all Knowledge into Knowledge Bases (KBs):** (see Section 12). This organizing principle has several important consequences:
    - All KBs are interpreted. This means that there are no free variables, and no purely uninterpreted syntactic systems.
    - No fundamental distinction between syntactic inference (KB ⊢ θ) and semantic inference (KB |= θ). All inference is essentially solving hidden CSPs.
    - Abstraction mappings between KBs allows a single abstraction KB to represent information abstracted from multiple specific KBs.
    - Only information necessary for performing specific inferences needs to be loaded into the working KB from the relevant KBs stored elsewhere.
    - The general KB is always loaded into working memory and only contains information that is universally true.
14. **Structured Entities** (see Section 5.3): These are entities whose structure carries information about the entity.
15. **Virtual Entities** (see Section 5.4): Virtual entities allow for very large domains of discourse without cluttering up the namespace with entities that are not referred to.
16. **Separation of Logic from Control** (see Section 16.8): Logical and control information is represented separately as axioms in KBs. The compiler combines both to generate efficient code.
17. **Ability of Inference to Return values other than T or F.** This allows the IAgt to answer queries with complex data structures rather than yes/no, greatly enhancing the reasoning capabilities of the IAgt.

19 Comparison to Prior KR Languages

Many languages for knowledge representation in AI have been developed over the course of its history, but no one language has emerged as the universal choice. There are many reasons for this lack of standardization, but one reason is that different languages were developed for restricted purposes and so fail as a general knowledge representation language. The following subsections highlight some of the major language choices and their limitations relative to SL.

19.1 SetLog vs. LLMs

SetLog (SL) is fully interpretable. By contrast, large language models (LLMs) represent their knowledge through high-dimensional embeddings and opaque parameter interactions. Every element of an SL knowledge base is expressed explicitly as a logical axiom which can be directly inspected, understood, and modified by human or computer users. This transparency enables precise traceability of reasoning steps: given any conclusion, one can reconstruct the axioms and inference rules that produced it. By comparison, the internal workings of LLMs are a black box, making it difficult to explain why a particular response was produced. Because of the statistical nature of LLMs, they can generate false outputs (hallucinations). This fundamental difference highlights the complementary roles

of the two approaches: SL provides semantic clarity and logical accountability, whereas LLMs provide statistical fluency without interpretability. See Appendix 5 for more details.

19.2 Separation of Knowledge from Control

SetLog (SL) cleanly distinguishes between the representation of knowledge ("the what") and the control information required for inference ("the how"). By contrast, Prolog combines control knowledge with declarative domain knowledge, producing a program in which the two are intermixed. This mixed representation makes it difficult to reason about domain knowledge independently of the control strategies. As Kowalski stated:

"Algorithm = Knowledge + Control" (Kowalski, 1979).

However, focusing solely on the algorithm obscures the separation between domain knowledge and control knowledge.

In Prolog, the declarative semantics of Horn clauses are tightly bound to the operational model of depth-first search with backtracking. This intertwining obscures the distinction between logical truth and computational procedure, and can introduce artifacts such as non-termination or rule order-dependence. SL eliminates this conflation by enforcing a strict separation: axioms in SL encode only logical or probabilistic truths within the knowledge base, entirely independent of any reasoning strategy. Inference procedures can therefore be designed, analyzed, or replaced in a modular fashion without affecting the semantics of the underlying representation. This separation improves clarity, allowing truth to be evaluated without procedural bias. It also increases flexibility, since diverse inference engines (deterministic, probabilistic, temporal, etc.) can operate over the same SL knowledge base without reformulating its content. This not only improves interpretability and reproducibility, but also supports extensibility, since new inference strategies can be incorporated without altering the logical structure of the knowledge base itself.

A further distinguishing feature of SetLog (SL) is its explicit treatment of control knowledge. In SL, guidance for inference – such as preferences of rule application order, search-space pruning, or prioritization of constraints – can be expressed declaratively within the control-knowledge KB, on equal footing with other forms of knowledge. This makes control strategies transparent and subject to inspection, rather than being embedded implicitly in the reasoning engine. When SL inference agents perform compilation, declarative control specifications are automatically translated into optimized executable code. This dual representation (declarative + procedural) provides the best of both worlds: human-readable and modifiable control rules at the declarative level, and computational efficiency at the procedural level. By compiling control knowledge into efficient functions, SL avoids the inefficiencies of purely declarative reasoning while preserving the ability to analyze, revise, and extend the reasoning process at the semantic level. This architecture enables SL to scale more effectively than systems in which control strategies are either opaque (as in many heuristic-driven systems) or purely declarative without performance optimization.

While SL enforces a clean separation between declarative knowledge and procedural inference, Cyc exemplifies a different design philosophy in which logical assertions are deeply intertwined with a large suite of heuristic inference modules. Cyc's knowledge base is expressed in first-order logic with numerous extensions, but the process of deriving conclusions often relies on specialized inference heuristics and rule priorities that embed procedural assumptions alongside declarative knowledge. This integration makes it difficult to disentangle what is logically entailed from what is heuristically inferred. As a result, Cyc's reasoning can appear opaque. SL avoids this entanglement.

19.3 Quantitative Reasoning

Quantitative reasoning in SetLog (SL) is facilitated by the fact that both numbers and sets are first-class primitives within the language. Unlike many traditional logic-based systems, where arithmetic must be encoded indirectly through axioms or external libraries, SL provides native constructs for numerical operations, set formation, and cardinality. This design choice enables concise representations of statements that mix logical and quantitative conditions, such as constraints on the size of sets, comparisons between numerical values, or probabilistic weightings attached to events. For example, the cardinality of a set can be directly queried or equated with a numerical variable without requiring auxiliary definitions. This tight integration reduces representational overhead, eliminates a common source of syntactic complexity, and allows inference engines to operate more efficiently over mixed symbolic/numeric domains. By treating quantitative reasoning as a core feature rather than an addon, SL lowers the barrier to modeling domains where logical structure and numerical constraints must interact, such as temporal reasoning, probabilistic inference, and scientific knowledge representation.

19.4 SetLog and Higher-Order Logic

SetLog includes higher-order entities in its domain of discourse such as sets, sets of sets and functions. As a result, SL can quantify over these higher-order entities. For example, consider the `Sort` function which orders a set (or multiset) according to an ordering function such as alphabetical or ascending orders. Sort is a higher-order function as it takes the name of another function as one of its arguments, as shown in the following example:

Example 19.4.1

```
SortedPrimesUnder100 = Sort[GreaterThan, PrimesUnder100$];
```

For a given set of prime numbers under 100, this function creates a sequence where the members of the input set are placed in numerical order. `GreaterThan` is a binary comparison function being passed as an argument to `Sort`.

In first-order logic (FOL) by definition, such higher-order entities are not part of the domain of discourse, and therefore can not be quantified over. Previous efforts often restricted themselves to FOL because of theoretical concerns about completeness and decidability. In practice, no logic-based inference is guaranteed to terminate in a time frame of interest, making concerns about completeness and decidability moot. Another problem

encountered with higher-order entities is increased overhead, due to the enlargement of the domain of discourse. SL partially avoids this problem by only quantifying over a subset of the domain of discourse. This corresponds to Henkin semantics (Henkin, 1973). Henkin's idea was to relax the requirement that higher-order variables range over all sets. Instead, in Henkin semantics, they range over some designated collection of sets/functions provided in the model. Similarly, higher-order function variables range over some chosen set of functions, not necessarily all possible ones. This makes higher-order logic behave more like first-order logic in disguise as one can encode Henkin models as first-order structures. And crucially, Gödel's completeness theorem holds: if something is true in all Henkin models, then it is provable.

19.5 Quantifiers in SetLog

SetLog eliminates explicit quantifiers without any loss of generality. In predicate calculus, universal and existential quantifiers are introduced syntactically, resulting in complexity in parsing and reasoning. In SL, all variables are implicitly universally quantified, and the scope of such a variable is limited to the axiom in which it occurs. Existential quantifiers in SL are replaced by two different representations depending on whether unique or general existence is intended, as explained in Section 10.3. The representation of general existence is only possible in SL because sets and numbers are primitive to the language.

19.6 Simplicity of Syntax

Another important advantage of SetLog (SL) is the extreme simplicity of its syntax. Unlike modal logics and other extensions of classical logic, which may require the introduction of exotic operators (e.g., $\Box$, $\Diamond$) to capture modality, temporality, or type constraints. Even the standard quantifiers $\forall$ and $\exists$ are eliminated in SL, thus avoiding any issues related to quantifier scope. Table 20.1 summarizes the key differences between SL and other knowledge representation systems.

Table 20.1

System	Interpretability	Declarative Model	Handles Probability	Quantitative Reasoning	Efficiency	Syntax Simplicity
Resolution Theorem Provers	4/5	4/5	1/5	2/5	1/5	4/5
Prolog	4/5	4/5	1/5	2/5	3/5	3/5
Cyc	4/5	5/5	3/5	2/5	1/5	5/5
Description Logics (DL/OWL)	4/5	5/5	1/5	2/5	3/5	4/5
Markov Logic Networks (MLNs)	2/5	4/5	4/5	3/5	2/5	4/5

System	Interpretability	Declarative Model	Handles Probability	Quantitative Reasoning	Efficiency	Syntax Simplicity
Proof Assistants (Coq, Isabelle, Lean)	5/5	5/5	1/5	3/5	2/5	5/5
OpenCog/Hyperon (MeTTa)	3/5	4/5	3/5	3/5	2/5	4/5
LLMs (GPT, Claude, etc.)	1/5	1/5	4/5	2/5	5/5	1/5
SetLog (SL)	5/5	5/5	5/5	5/5	4/5	5/5

20 Summary

20.1 Introduction

SetLog (SL) is a logic-based language designed for knowledge representation and reasoning (KRR) with a central goal of enabling transparent, inspectable, and verifiable artificial intelligence. Unlike large language models (LLMs), which encode knowledge implicitly in high-dimensional vectors, SL encodes knowledge explicitly as axioms in a logical system. This makes reasoning both explainable and computationally efficient. The motivation behind SL is to provide a rigorous foundation for Artificial General Intelligence (AGI) by combining symbolic logic with probability, arithmetic, sets, and time.

20.2 Core Foundations

- Logical Basis**:** SL extends higher-order predicate calculus with numbers, sets, probability, and temporal reasoning as primitive constructs.
- Knowledge Bases (KBs)**:** All knowledge is organized into KBs, expressed as axioms. These KBs support structured, incomplete, probabilistic, and temporal knowledge.
- Functions are core constructs**.** Unlike Prolog-style languages, SL is function-oriented.
- Predicates are boolean-valued functions.
- Functions m ay accept or return other functions (higher-order).
- Polymorphism and multiple inheritance are supported through type-axioms.
- Entities**:**
    - Elementary entities**:** constants or symbols.
    - Structured entities: tuples, points, line segments, etc.
    - Virtual entities**:** large or implicit collections (e.g., voxel grids).

20.3 Syntax and Semantics

- Dual Syntax:
  - Input SL: human-friendly, flexible notation.
  - Internal SL: canonical prefix form used internally for reasoning and compilation.
  - A bidirectional translator ensures equivalence.
- Quantification: Explicit ∀ and ∃ are eliminated. Instead:
  - Universal variables implicitly express ∀.
  - Existential quantifiers are replaced by two forms of existential representation. One for unique existence using Skolem functions and the other for general existence using set size.
- Sets and Multisets: Sets are primitive. Membership axioms define extensional or intensional sets. Multisets (with repeated elements) are also supported.
- Sequences: Represented as functions from indices to values, generalizing arrays or lists.

20.4 Uncertainty and Probability

- Probability Axioms: Reified into higher-order logic, allowing probability distributions to be reasoned about within the system.
- Interpretation of Universals: The probability of universally quantified statements can be treated as “all” or “any,” depending on context.
- Integration: Probabilistic and logical inference coexist within the same formalism.

20.5 Temporal and Dynamic Reasoning

- SL incorporates time, events, and state evolution as first-class constructs.
- Entities and functions may be time-dependent, allowing reasoning about change and processes.
- Both discrete events and continuous time reasoning are supported, providing flexibility for modeling real-world dynamics.

20.6 Efficiency and Compilation

- SL Compiler:
  - Converts logical axioms into efficient procedural code (e.g., C or Rust).
  - This transformation preserves correctness but improves runtime performance.
- Control vs. Logic Separation**:** Unlike Prolog, where execution order affects logical outcomes, SL cleanly separates declarative logic from control knowledge.
- Constraint Solving**:** Functions can be defined indirectly by constraints, with CSP solvers integrated into inference.

20.7 Inference Mechanisms

- Logical Inference: Classical deductive reasoning extended to higher-order logic.
- Probabilistic Inference: Uses probability axioms for reasoning under uncertainty.
- Constraint Inference**:** Invokes CSP solvers for problems not definable by direct functional axioms.

20.8 Significance and Applications

SetLog aims to address the opacity, hallucination, and unverifiability problems of modern AI systems. Its unification of logic, probability, computation, and temporal reasoning makes it a candidate foundation for transparent AGI. Applications include:

- Knowledge engineering: Building inspectable KBs in scientific, military, and industrial domains.
- Reasoning under uncertainty: Integrating probabilistic and logical models.
- Self-improving systems: Recursive compilation and meta-reasoning.

20.9 Conclusion

SetLog represents a shift from opaque statistical models toward a clear, axiomatic, and compilable framework for intelligence. Its design combines the rigor of mathematical logic with practical tools for efficient computation. By embedding sets, numbers, probability, and time directly into the language, SL provides a foundation for scalable, interpretable, and general-purpose AI systems.

References

1. Akama, K., & Nantajeewarawat, E. (2011, October). Meaning–preserving skolemization. In *International Conference on Knowledge Engineering and Ontology Development* (Vol. 2, pp. 322–327). SCITEPRESS.
2. Allen, J. F., & Ferguson, G. (1994). Actions and events in interval temporal logic. *Journal of logic and computation*, *4*(5), 531–579.
3. Baader, F. (Ed.). (2003). *The description logic handbook: Theory, implementation and applications*. Cambridge university press.
4. Baier, C., & Katoen, J. P. (2008). *Principles of Model Checking*. MIT Press.
5. Banieqbal, B., & Barringer, H. (1986). *Temporal logic with fixed points*. In Temporal Logic in Specification (pp. 62–74). Springer.
6. Barwise, J., Etchemendy, J., Allwein, G., Barker–Plummer, D., & Liu, A. (2002). *Language, proof and logic* (p. 598). Stanford, USA: CSLI publications.
7. Besnard, P. (1989). *An introduction to default logic*. Springer Science & Business Media.
8. Bayes, T. (1763). LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, FRS communicated by Mr. Price, in a letter to John

Canton, AMFR S. *Philosophical transactions of the Royal Society of London*, (53), 370–418.
9. Boole, G. (1847). *The mathematical analysis of logic*. Philosophical Library.
10. Cheeseman, P. C. (1985, August). In Defense of Probability. In *IJCAI* (Vol. 85, pp. 1002–1009).
11. Carta, S., Giuliani, A., Piano, L., Podda, A. S., Pompianu, L., & Tiddia, S. G. (2023). Iterative zero-shot llm prompting for knowledge graph construction. *arXiv preprint arXiv:2307.01128*.
12. Cheeseman, P. C., A method of computing generalized Bayesian probability values for expert systems, IJCAI, 1983.
13. Clarke, E. M., and Emerson, E.A. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings of the IBM Workshop on Logics of Programs. Lecture Notes in Computer Science, vol. 131. Springer–Verlag, New York, 1981, pp. 52–71.
14. Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM Transactions on Programming Languages and Systems, 8(2), 244–263.
15. De Moura, L., Kong, S., Avigad, J., Van Doorn, F., & von Raumer, J. (2015). The Lean theorem prover (system description). In *Automated Deduction–CADE–25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1–7, 2015, Proceedings 25* (pp. 378–388). Springer International Publishing.
16. van Ditmarsch, H., van Der Hoek, W., & Kooi, B. (2007). *Dynamic epistemic logic* (Vol. 337). Springer Science & Business Media.
17. Doyle, J. (1979). A truth maintenance system. *Artificial intelligence*, *12*(3), 231–272.
18. Emerson, E. A., & Halpern, J. Y. (1986). “Sometimes” and “not never” revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, *33*(1), 151–178.
19. Gabbay, D. M. (2003). Many–dimensional modal logics: theory and applications.
20. Gallardo-Delgado, A.A., Nolazco-Flores, J.A., Priego-Sánchez, Á.B. *et al.* Large Language Models and Knowledge Graphs: A State-of-the-Art Exploration. *SN COMPUT. SCI.* **6**, 752 (2025). https://doi.org/10.1007/s42979-025-04277-7
21. Galton, A. (1990). A critical examination of Allen's theory of action and time. *Artificial intelligence*, *42*(2–3), 159–188.
22. Goertzel, B. (2021). The general theory of general intelligence: a pragmatic patternist perspective. *arXiv preprint arXiv:2103.15100*.
23. Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, *42*(1–3), 335–346
24. Henkin, L. (1973). Internal semantics and algebraic logic. In *Studies in Logic and the Foundations of Mathematics* (Vol. 68, pp. 111-127). Elsevier.
25. Hindley, J. R. (1997). *Basic simple type theory* (No. 42). Cambridge University Press.
26. Hintikka, K. J. J. (1962). Knowledge and belief: An introduction to the logic of the two notions.

27. Horrocks, I., Patel–Schneider, P. F., & Van Harmelen, F. (2003). From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of web semantics*, *1*(1), 7–26.
28. Huber, F. (2007). The plausibility-informativeness theory.
29. Jeffrey, R. C. (1965). *The logic of decision*. McGraw-Hill. *(If you used the second edition, cite instead as: Jeffrey, R. C. (1983).* The logic of decision *(2nd ed.). University of Chicago Press.)*
30. Jordan, M. I. (Ed.). (1999). *Learning in graphical models*. MIT press.
31. Joyce, J. M. (2011). The development of subjective Bayesianism. In *Handbook of the History of Logic* (Vol. 10, pp. 415-475). North-Holland.
32. Jun, Y., Michaelson, G., & Trinder, P. (2002). Explaining polymorphic types. *The Computer Journal*, *45*(4), 436–452.
33. Kamp, H. (2013). Events, instants and temporal reference. In *Meaning and the Dynamics of Interpretation* (pp. 53–103). Brill.
34. Kowalski, R. (1979). Algorithm= logic+ control. *Communications of the ACM*, *22*(7), 424-436.
35. Kowalski, R. A. (1990, November). Problems and promises of computational logic. In *Computational Logic: Symposium Proceedings, Brussels, November 13/14, 1990* (pp. 1–36). Berlin, Heidelberg: Springer Berlin Heidelberg.
36. Kowalski, R., & Sergot, M. (1986). A logic-based calculus of events. *New generation computing*, *4*, 67–95.
37. Kyburg, H.E. (1961). *Probability and the Logic of Rational Belief*, Middletown, CT: Wesleyan University Press.
38. Lassila, O., & McGuinness, D. (2001). *The role of frame-based representation on the semantic web*. Linköping University Electronic Press.
39. Levi, I. (1980). *The enterprise of knowledge: An essay on knowledge, credal probability, and chance*. MIT Press.
40. Lewis, C. I. (1932). Alternative systems of logic. *The Monist*, 481-507.
41. Lloyd, J. W. (2007, October). Declarative programming for artificial intelligence applications. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming* (pp. 123–124).
42. Manna, Z., & Pnueli, A. (1992). *The temporal logic of reactive and concurrent systems: specifications* (Vol. 1). Springer Science & Business Media.
43. McAllester, D. A. (1990, July). Truth Maintenance. In *AAAI* (Vol. 90, pp. 1109–1116).
44. J. McCarthy and P. Hayes (1969), Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, Machine Intelligence 4, chapter 26, pages 463–502. Edinburgh University Press, Edinburgh.
45. Minsky, M. (1974, June). *A framework for representing knowledge*.
46. Newell, A., & Simon, H. (1956). The logic theory machine—A complex information processing system. *IRE Transactions on information theory*, *2*(3), 61–79.
47. Nilsson, N. J. (1986). Probabilistic logic. *Artificial intelligence*, *28*(1), 71–87.
48. Ng, K. S., & Lloyd, J. W. (2009). Probabilistic reasoning in a classical logic. *Journal of Applied Logic*, *7*(2), 218–238.

49. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., & Rosati, R. (2008). Linking data to ontologies. In *Journal on data semantics X* (pp. 133–173). Springer Berlin Heidelberg.
50. Prior, A. N. (1969). Recent advances in tense logic. *The monist*, 325–339.
51. Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine learning*, *62*, 107–136.
52. Reiter, R. (1980). A logic for default reasoning. *Artificial intelligence*, *13*(1–2), 81–132.
53. Robinson, J. A. (1965). A machine–oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, *12*(1), 23–41.
54. Russell, B., & Moore, G. H. (2015). The Paradoxes of Logic [1906]. In *The Collected Papers of Bertrand Russell, Volume 5* (pp. 273–296). Routledge.
55. Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Pearson.
56. Shafer, G. (1992). Dempster–Shafer theory. *Encyclopedia of artificial intelligence*, *1*, 330–331.
57. Shanahan, M. (2001). The event calculus explained. In *Artificial intelligence today: Recent trends and developments* (pp. 409–430). Berlin, Heidelberg: Springer Berlin Heidelberg.
58. Shoham, Y., & McDermott, D. (1988). Problems in formal temporal reasoning. *Artificial Intelligence*, *36*(1), 49–61.
59. Stanford University. (2012, Winter). *Condensed slides: Lecture 5* [PDF slides]. CS 103: Mathematical Foundations of Computing. https://web.stanford.edu/class/archive/cs/cs103/cs103.1242/lectures/05/Condensed%20Slides.pdf
60. Vectara. (n.d.). *Hallucination leaderboard* [GitHub repository]. GitHub. Retrieved August 5, 2025, from https://github.com/vectara/hallucination-leaderboard
61. Williamson, J. (2004). *Bayesian nets and causality: philosophical and computational foundations*. Oxford University Press.
62. Zadeh, L. A. (2023). Fuzzy logic. In *Granular, fuzzy, and soft computing* (pp. 19–49). New York, NY: Springer US.

Appendix 1: Logical Equivalences

P, Q, and R represent atomic propositions.

1. Biconditional elimination
   ```
   P <==> Q = And[(P ==> Q), (Q ==> P)]
   ```
2. Implication elimination
   ```
   P ==> Q = Or[Not[P], Q]
   ```
3. Commutativity of And
   ```
   And[P, Q] = And[Q,P]
   ```
4. Commutativity of Or
   ```
   Or[P, Q] = Or[Q,P]
   ```
5. Associativity of Set And
   ```
   And[{P, And[{Q, R}]}] = And[{P, Q, R}]
   And[(And[P, Q]), R] = And[{P, Q, R}]
   ```
6. Associativity of Set Or
   ```
   Or[P, (Or[Q, R)] = Or[P, Q, R]
   Or[(Or[P, Q]), R)] = Or[P, Q, R]
   ```
7. Associativity of Binary And
   ```
   And[And[p, q], r] = And[p, And[q, r]]
   ```
8. Double Negation Elimination
   ```
   Not[Not[P] = P
   ```
9. Contraposition
   ```
   P ==> Q = Not[Q] ==> Not[P]
   ```
10. De Morgan's Law (And)
    ```
    Not[(And[P, Q])] = Or[Not[P], Not[Q]]
    ```
11. De Morgan's Law (Or)
    ```
    Not[(Or[P, Q])] = And[Not[P], Not[Q]]
    ```
12. Distributivity of And over Or
    ```
    And[P, (Or[Q, R])] = Or[(And[P, Q]), (And[P, R])]
    ```
13. Distributivity of Or over And
    ```
    Or[P, (And[Q, R])] = And[(Or[P, Q]), (Or[P, R])]
    ```

Appendix 2: Probability, Logic, and Decision Theory

1. Decision Theory and Value of Information

The primary reason that agents need to represent probabilistic information is to make rational decisions under uncertainty. A posterior probability density of a proposition given evidence is needed for computing expected values, particularly expected utility.
From a decision-theoretic perspective, more accurate probability estimates generally require more evidence. However, collecting more evidence usually incurs costs – computational, temporal, or physical. The SL probability approach supports value-of-information reasoning, enabling the system to evaluate whether the expected improvement in decision quality justifies the cost of gathering additional information. This

allows the IAgt to make rational higher-order decisions about whether to act on current knowledge or seek further evidence.

2. Conditional Independence and Maximum Entropy (maxent)

In many practical reasoning scenarios, direct use of conditional probability quickly runs into a severe limitation: the combinatorial explosion of conditioning variables relative to the available data. For example, estimating the probability that a specific patient will survive a scheduled heart operation would, in principle, require identifying previous cases with an identical configuration of relevant factors – such as age, weight, blood pressure, smoking history, lab results, and more. Due to the exponential growth in possible factor combinations, very few (if any) exact matches exist in the historical data, making direct estimation of conditional probabilities infeasible.

The standard solution to this combinatorial problem is to make either marginal or conditional independence assumptions. The principle of maximum entropy (maxent) is a generalized method for making independence assumptions subject to the known constraints. In simple situations, maxent reduces to marginal or conditional independence, but in more complex situations it numerically gives the maximally independent probability distribution, even when there is no closed-form formula. In essence, maxent distributes the uncertainty over the joint distribution as evenly as possible while still conforming to the given constraints. Maxent is sometimes referred to as the principle of least information as it gives the distribution with the minimum information (in the Shannon sense) subject to the constraints. Any other distribution would be assuming more information than is given. This description of maxent is somewhat misleading as maxent inference is making the strong assumption that the given constraints are the only ones operating in the domain. If this is not the case, then the maxent probability values will deviate significantly from observed values. If such deviations are detected, then it is a signal to search for the missing constraint(s).

Maxent fits neatly with logic when the information in the KB is incomplete. In such cases there is a set of possible worlds consistent with the axioms. With incomplete KBs, logical inference can only return “unknown” for some queries when a particular target proposition is true in some worlds and false in others. Maxent says that in such cases assume that all possible worlds are equally likely (i.e. maxent becomes the principle of indifference), and so the probability of the target proposition is just the proportion of worlds in which it is true.

A common question about using maxent prior probabilities is why they should be preferred over other methods for assigning priors. Generally, there is no good reason for assuming that the domain probabilities should have their maxent values. Instead, the justification for maxent priors is that they provide a neutral background for detecting real effects. In the case of KBs representing real world domains, the second law of thermodynamics provides an additional reason why the domain probabilities would match the maxent expectations: evolution of physical states converge on the maxent values, subject to any constraints that they must obey. In other words, as a result of physical world’s tendency to evolve toward a

state of maximum entropy, assuming maximum entropy distribution has a reasonable probability of being correct.

3. Maxent vs. Bayesian Reasoning (and how they fit)

Maxent and Bayes are complementary uncertainty reasoning formalisms. Maxent is a principle for selecting a distribution given constraints, usually used to *choose* priors when the available information is underspecified. Bayes is the principle for *updating* those prior beliefs with evidence. A common workflow is: (i) encode whatever structural/expectation constraints you legitimately have; (ii) use maxent to derive a prior that honors *only* those; (iii) update with data via Bayes' rule. This approach raises two main questions:

1. **How much weight should the maxent prior carry?**
Answer: encode its strength explicitly. For a Bernoulli parameter θ, let the maxent prior imply a pseudo-count (equivalent sample size) W around a maxent center p_0 (often 1/2 under symmetry), i.e., Beta(α_0, β_0) with $\alpha_0 = Wp_0$, $\beta_0 = W(1 - p_0)$,
where Beta is the well known beta distribution from probability theory. Then data (k heads of n) yield the posterior Beta$(\alpha_0 + k, \beta_0 + n - k)$ and the posterior-predictive distribution for the next flip is:

$$\Pr(H \mid D_n) = \frac{\alpha_0 + k}{\alpha_0 + \beta_0 + n}$$

Choosing W sets exactly "how much weight to give the initial maxent value."

2. **Why compute probabilities when no utility function is present?**
Answer: because beliefs are reusable inputs for later decisions. Science routinely estimates parameters (e.g., the speed of light) without any particular downstream use in mind. Generally, probabilistic axioms are stored in the relevant KBs and used when needed. In summary:

- Use maxent to set priors that reflect only warranted constraints.
- Control prior influence with an explicit strength parameter.
- Use Bayes to update, letting past data inform the future through θ (or other latent parameters).
- Cache beliefs now; plug utilities in when they're available.
- When maxent predictions do not match the data it's a signal to look for additional constraints.

4. Conditioning Strategy: What to Condition On?

In principle, conditioning on all available knowledge should yield the most accurate probability estimates. However, in practice, this is often computationally infeasible. SL does not enforce a fixed conditioning policy. Instead, it supports a flexible conditioning mechanism, allowing developers to determine what evidence to include based on domain constraints, computational budget, and inference objectives. There is no universally agreed-upon theory for optimal conditioning; pragmatic trade-offs are always necessary.

Appendix 3: Temporal Logic Review

Temporal logic has been significantly advanced by numerous researchers, each introducing distinct approaches to representing and reasoning about time. Arthur Prior laid the foundation in the 1950s and 1960s by formulating tense logic, explicitly introducing temporal operators such as "F" (future) and "P" (past) (Prior, 1969). Prior's contributions provided the initial framework upon which subsequent extensions were built, emphasizing linear progression of time with discrete, clearly defined temporal points. An example of Prior's notation is shown in Example A3.1:

Example A3.1

```
q ==> F(P(q))
```

The above example represents the tautological point that if a proposition q is true now, then at a future time it is the case that it was true at some time in the past.

Building on this foundational work, Hans Kamp introduced operators "Since" and "Until," enriching temporal logic's ability to represent intervals and complex temporal dependencies (Kamp, 2013). His approach allowed for more nuanced reasoning about events extending over intervals rather than mere points in time. Similarly, Allen and Ferguson (1994) significantly influenced temporal reasoning with Interval Temporal Logic, introducing a robust framework explicitly representing intervals and their relationships, such as "meets," "overlaps," and "during." Allen's interval logic provided practical utility, especially in artificial intelligence applications where events frequently occur over extended durations. Example A3.2 shows how Kamp's operators can be used.

Example A3.2

```
(p U q) ==> F(q)
```

The interpretation of the example above is that if proposition p holds (is true) continuously until proposition q holds (becomes true) then proposition q becomes true at some point in the future. At first glance, the consequent may not seem to logically follow, but the existence of q being true at some point is entailed in the Until operator (U).

Furthering the computational aspects, Manna and Pnueli developed Linear Temporal Logic (LTL), introducing operators such as "O" (next), "□" (globally), "◇" (eventually), and "U" (until) (Manna & Pnueli, 1992). These operators enabled precise specifications and verification of computer programs and systems by modeling temporal behaviors explicitly. Meanwhile, Emerson, Halpern, and Clarke extended temporal logic into branching structures with Computational Tree Logic (CTL), introducing path quantifiers (e.g., "A" for all paths, "E" for some path) to reason about multiple potential future states and model checking (Clarke & Emerson, 1981, Emerson & Halpern, 1986).

Example A3.3

□ ϕ ==> ◇ ϕ

The above example, in Manna and Pnueli's notation, illustrates the tautology that if proposition ϕ is always true (true in all states), then ϕ is eventually true (true in final state).

John McCarthy and Patrick Hayes introduced Situation Calculus in the 1960s, providing a formalism specifically designed for reasoning about actions and state changes within dynamic systems. This system introduced several temporal functions such as At, Holds, and End. The focus of this schema is concerned with modeling actions in the real world. Another key innovation of McCarthy and Hayes is the introduction of the idea of fluents. McCarthy and Hayes used the term fluent to represent conditions or propositions whose truth values can change over time or between different situations. Their work explicitly identified and addressed the frame problem, which concerns efficiently representing properties that remain invariant despite actions. This formalism laid crucial foundations for subsequent reasoning frameworks used in artificial intelligence. (McCarthy and Hayes, 1969)

Example A3.4

At(robot,y,do(move(x,y),s))

The interpretation for the above expression is "After performing the action move(x, y) in situation s, the robot will be at room y.

Yoav Shoham and Drew McDermott approached temporal logic from the perspective of reasoning about actions and causality. Shoham developed the Temporal Logic of Actions (TLA), while McDermott contributed significantly by addressing the frame problem, introducing specialized constructs and logical formalisms to manage implicit state persistence (Shoham & McDermott, 1988). Concurrently, Robert Kowalski and Marek Sergot (1986) proposed the Event Calculus, directly tackling the frame problem through fluents and built-in inertia axioms, effectively modeling persistence and state changes over time and simplifying reasoning about actions and events.

Example A3.5

Occurs(StartEngine, t) ==> Holds(EngineRunning, t + ϵ)

In Shoham and McDermott's formalism the above example's interpretation is: If the action of starting the engine occurs at a time t, then it follows logically that the fluent ("condition") EngineRunning will hold at some small instant of time (t +ϵ) immediately afterward.

Example A3.6

Initiates(TurnOn, LightOn, t)
Terminates(TurnOff, LightOn, t)

In the Event Calculus these expressions state that turning on the light causes LightOn to become true, and turning it off causes it to become false. Some other key predicates in the Event Calculus are:

Happens(e, t): Event e occurs at time t.
Initiates(e, f, t): Event e at time t causes fluent f to start holding.
Terminates(e, f, t): Event e at time t causes fluent f to stop holding.
HoldsAt(f, t): Fluent f holds at time t

These provided a much more expressive language for reasoning about actions and their corresponding changes on state.

Finally, Dov Gabbay and Antony Galton expanded temporal logic by integrating nonmonotonic reasoning (Gabbay, 2003), defaults, and metric interval–based reasoning (Galton, 1990), respectively. Gabbay's logics allowed handling of defaults and exceptions, crucial in realistic temporal reasoning contexts, while Galton provided formal structures for reasoning about continuous and metric temporal phenomena. Collectively, these researchers' diverse contributions and many others have provided temporal logic with rich, expressive capabilities, supporting a wide range of theoretical and practical applications.

Appendix 4: Return Types for Structured Entities

Suppose the structured entity `<Price, quantity, currency>` is defined with `quantity` as an integer, and `currency` as an entity drawn from a set of possible currencies. Addition of two *`Price`* entities can be defined, with a type-axiom that requires that the currencies of the summands must match, and additionally requires that the currency of the sum must match as well. `PriceP` is a structured type recognition predicate, that requires `quantity` to be an integer, and `currency` to satisfy a `CurrP` predicate.
There are three ways to implement this example in SL. The first two ways make use of the standard type-axiom schema, in which the return value predicate takes a single architecture. This schema looks like:

```
ARG-TERM <==> RETURN-PRED[PriceAdd[p1,p2]];
```

where `ARG-TERM` is:

```
And[PriceP[p1],PriceP[p2],p1[3] = p2[3])]
```

The first implementation for the return type predicate is an anonymous inner predicate which enforces the currency match with the arguments. Specifically, the return type constraint predicate would be:

```
And[PriceP[@], @[3]=p1[3])]
```

In words, this says that the return value must be a *`Price`* structured entity (assuming a recognition predicate `PriceP`), and that the currency must match the currency of the arguments. Here the recognition predicate for *`Price`* constrains the first argument n to be an integer, and the `ARG-TERM` above requires that `p1[3]=p1[2]`.
The second implementation avoids the use of an anonymous inner predicate by the use of a function factory, that is a function returning a function (see Section 6.11, Functions Returning Functions). This function factory is defined in a separate axiom, and essentially returns the predicate described by the anonymous inner predicate above. Naming it `"PricePredFactory"`:

```
(PricePredFactory[c] = pred) <==>
     (pred[x] <==> And[PriceP[x], x[3]=c)]));
```
Then the RETURN-PRED above would simply be
```
PricePredFactory[p1[3]]
```
Putting all of this together, the type-axiom using the function PricePredFactory would be:
```
And[PriceP[p1],PriceP[p2], p1[3] = p2[3]] <==>
     (PricePredFactory[p1[3]])[PriceAdd[p1, p2]];
```
The third implementation makes use of an alternate type-axiom schema in which the return value predicate takes two arguments:
```
ARG-TERM ==> RETURN-PRED[PriceAdd[p1, p2], p1[3]];
```
Now the RETURN-PRED is defined as a named predicate in a separate axiom, termed here as PriceValuePred:
```
PriceValuePred[x, curr] <==>
     And[PriceP[x], x[2] = curr]];
```
Here, PriceP is a structured entity recognition predicate that ensures x is a <Price,quantity,currency> as described above. This named predicate can now be used as the RETURN-PRED:
```
And[PriceP[p1], PriceP[p2], p1[3] = p2[3]]
     <==> PriceValuePred[PriceAdd[p1, p2], p1[3]];
```

Appendix 5: SetLog vs. LLMs: A Comparison

Here's a side-by-side comparison chart highlighting the differences between SetLog and Large Language Models (LLMs), tailored for presentations or discussions with collaborators and funders:

Dimension	SetLog (SL)	Large Language Models (LLMs)
Knowledge Representation	Explicit axioms in a formal logic language (transparent, inspectable).	Implicit statistical embeddings (opaque, uninterpretable).
Reasoning	Deductive, probabilistic, and constraint-based inference; guaranteed logical soundness.	Pattern-matching and statistical correlation; no guarantee of logical validity.
Uncertainty	Native probability axioms integrated into logic.	Learned heuristics; no explicit probability semantics.
Quantification	Eliminates ∀ and ∃ in favor of universal variables, Skolem functions, and sets.	No formal quantification; relies on statistical associations.
Functions & Polymorphism	Function-oriented design; supports higher-order functions, polymorphism, multiple inheritance.	No true function semantics; operates on token sequences.

Dimension	SetLog (SL)	Large Language Models (LLMs)
Entities	Supports elementary, structured, and virtual entities (e.g., voxel grids).	No explicit notion of entities; everything reduced to token co-occurrence.
Sets & Sequences	Sets and multisets are primitives; sequences are functions from indices.	No primitives for sets or sequences beyond text simulation.
Control vs. Logic	Clean separation; compiler translates axioms into efficient procedural code (C, Rust, etc.).	Entangled; reasoning inseparable from training data and statistical inference.
Transparency	Fully explainable: every inference traceable to axioms.	Opaque: "black-box" behavior, prone to hallucination.
Self-Improvement	Can represent and compile its own inference rules, enabling recursive self-improvement.	Limited: improvements require retraining with external data.
Applications	Inspectable AGI, knowledge engineering, scientific/industrial reasoning, defense applications.	Conversational AI, content generation, autocomplete, general assistance.

Key Takeaways:

- **Transparency**: SetLog solves the "black box" problem - every decision is inspectable.
- **Reliability**: Logical inference prevents hallucinations common in LLMs.
- **Integration**: Combines logic, probability, sets, and computation in one framework.
- **Future Potential**: Provides the foundation for scalable, self-improving AGI, where LLMs cannot.

Appendix 6: Probabilistic Inference in SetLog

Classical logic provides a framework for reasoning under certainty: propositions are either entailed (true) or refuted (false). However, most practical domains require reasoning under uncertainty. Probability theory extends logic by assigning *graded degrees of belief* to propositions, conditional upon available evidence and explicit assumptions. Within SetLog (SL), probability terms are represented as first-class expressions and are subject to the same logical manipulations as other axioms. Probabilistic statements are expressed in SL as functions of a target proposition, evidence, and explicit assumptions – i.e. `Prob[φ, Evidence, Assumptions]= PDF.`

Logical inference rules can be used to reduce or simplify the propositional structure, while probabilistic axioms compute or transform the associated values. As an example of

probabilistic inference, consider the beta-Bernoulli model; one of the most commonly used probability models for binary events.

Consider repeated coin flips modeled as random variables `Flip[i] ∈ {H,T}`, assumed conditionally independent and identically distributed outcomes given a parameter $\theta \in [0,1]$, denoting the unknown probability of heads. Let the prior distribution over θ be $\mathrm{Beta}(\alpha_0, \beta_0)$, where Beta is a well-known probability PDF, and the dataset consists of K trials with k successes, the posterior distribution and posterior-predictive probability is then:

$$Prob[\theta, (\alpha_0, \beta_0, k, K), CondIndep] = \mathrm{Beta}(\alpha_0 + k, \beta_0 + K - k) = \frac{\alpha_0 + k}{\alpha_0 + \beta_0 + K}$$

where α_0 is the prior number of successes, β_0 is the prior number of failures.

A common case for the prior is the uniform prior with $\alpha_0 = 1,\ \beta_0 = 1$, giving the posterior mean value of $(1+k)/\ (2 + K)$. Here, the data values *k* and *K* can be extracted from trial data, represented as a set of outcome propositions. In the case of coin flip data, the evidence can be represented in SL as:

```
Trialset$ = {<Flip[1], H>, <Flip[2], T>, …, <Flip[n], H>};
```

As K increases, the posterior density concentrates around the true value of θ. However, no finite amount of evidence yields certainty (0 or 1 probability). SL is well suited to probabilistic representation and reasoning because sets and numbers are native to the language, as indicated by the numerical parameters such as k and K, and data sets, such `Trialset$`. In addition to the logical rules of inference that any IAgt uses, the following additional rules (axiom schemas) can be used for inference with probability axioms:

- **Chain rule (Bayesian expansion):**
$$\Pr(A \wedge B \mid C) = \Pr(A \mid C)\Pr(B \mid A \wedge C).$$
- **Marginalization:**
$$\Pr(A \mid C) = \sum_x \Pr(A \mid x, C)\Pr(x \mid C).$$
- **Negation:**
$$\Pr(\neg A \mid C) = 1 - \Pr(A \mid C).$$
- **Disjunction:**
$$\Pr(A \vee B \mid C) = \Pr(A \mid C) + \Pr(B \mid C) - \Pr(A \wedge B \mid C).$$
- **Independence:**
$$Indep(A, B; C) \Rightarrow \Pr(A \wedge B \mid C) = \Pr(A \mid C)\Pr(B \mid C).$$
- **Conditional independence:**
$$CIndep(A, B \mid Z; C) \Rightarrow \Pr(A \wedge B \mid Z, C) = \Pr(A \mid Z, C)\Pr(B \mid Z, C).$$

In SL, these axioms are represented as guarded equivalences, with independence predicates recorded explicitly in the assumptions set. Determining the appropriate set of conditioning variables is a fundamental open problem in probabilistic inference. A

pragmatic approach is to rely on causal models: conditioning on direct causes rather than arbitrary correlates. Causal models can be used to suggest what information might be probabilistically informative in particular situations. In SL, such modeling choices are

Summary:
In SL, probabilistic reasoning is tightly integrated with logical reasoning. Logical inference manages propositional structure, while probabilistic axioms govern uncertainty. Independence assumptions and the MaxEnt principle provide mechanisms for tractable inference when complete probabilistic structures are unavailable. Bayesian updating supplies the unique coherent method of belief revision. Collectively, these elements render SL a unified framework for reasoning about both certainty and uncertainty, supporting the ultimate goal of principled decision-making.

Appendix 7: Pattern Matching and Inference

Pattern matching is the essential operation involved in doing inference. Consider the axioms and type-axiom for `Factorial` presented earlier in Section 6.3.1:

```
(Factorial[n] = n*Factorial[n-1]) <==> (n > 1);
Factorial[1] = 1;

## Type-axiom for Factorial:
NatNumP[n] ==> NatNumP[Factorial[n]];
```

This states that `Factorial` is a function which takes a single natural number as argument. When the IAgt is tasked with a logical evaluation of `Factorial[7]`, it does a pattern match against the defining axioms for `Factorial`, coupled with the type-axiom for `Factorial`. These yield two patterns. The first is a single natural number argument with a condition that the argument be greater than one, and the second is a single natural number argument which is the constant `1`. Of these two, the first matches the requested evaluation of `Factorial[7]`, and the IAgt does a substitution of `n → 7`, and applies this to the function definition. Then `n*Factorial[n-1]` becomes `7*Factorial[6]`. As described previously in Section 6.3.1, the process then recurses, to yield the eventual result of `5040`.

In some cases, matching may become a significant computational task. For example, a KB for computing mathematical integrals will have many patterns to match against a given integrand (posed as a SL expression). The IAgt can identify such situations and construct efficient tree searches.

Pattern matching is recursive when sequences such as structured entities are involved. For example, in Section 5.3 a *`Point2D`* is defined as a sequence that looks like `<`*`Point2D`*`, Int, Int>`. An addition function for these points can then be defined:

```
VectorPointAdd[<Point2D, i, j>, <Point2D, l, m>] =
     <Point2D, i+l, j+m >;
```

with type-axiom:

```
And[Point2DP[a],Point2DP[b]] ==> Point2DP[VectorPointAdd[a,b]];
```

(See Section 5.3 for the definition of `Point2DP`).

Suppose the IAgt needs to evaluate

`VectorPointAdd[<`*`Point2D`*`, 4, 3>, <`*`Point2D`*`, -1, 8>]`

The top level of matching looks for two arguments, each of which must satisfy the `Point2DP` predicate. A second level of matching applies the Point2DP predicate to each argument, first checking to see if they are sequences, then of length 3, then with first element *Point2D*, then with the remaining elements integers. The pattern match produces substitutions `i` → 4, `j`→3, `l`→-1, and `m`→8.

In SL, pattern matching can be controlled by conditions. For example, a function definition could look like:

```
(n != -1) => (Integrate[x^n, x] = (x^(n+1))/(n + 1));
```

In this case, the matching '`n`' inside an integrand must pass the additional test: `n != -1`.

More simply, revisiting the example from Section 6.3:

```
(RowNumber[squareID] = 1) <==> And[squareID > 0,squareID < 4];
(RowNumber[squareID] = 2) <==> And[squareID > 3,squareID < 7];
(RowNumber[squareID] = 3) <==> And[squareID > 6,squareID < 10];
```

The evaluation of `RowNumber[7]` involves matching `7` against `squareID` (after verifying that `7` satisfies the type-axiom for `RowNumber`). However, this pattern match is guarded by the conditions requiring `squareID` to lie in a certain range, so the match will fail. This is pattern matching with attached conditions.