CREATING A UNIVERSAL INFORMATION LANGUAGE

OSCAR BENDER-STONE

ABSTRACT. Welkin is a formalized programming language to store information. We introduce its use cases and rigorously define its syntax and semantics. From there, we introduce the bootstrap, making Welkin completely self-contained under the meta-theory of Goedel's System T (equi-consistent to Peano Arithmetic).

1. INTRODUCTION

Humanity produces a colossal amount of data each year. According to the International Data Corporation, there is currently 163 zettabytes of digital data in the world. Given that the average human can read, on average, 200 words per minute, and approximating a word as 8 bytes, this would require *billions* of years ($7.8675 * 10^9$ minutes). Even in restricted areas, such as academia, the amount of data available cannot be consumed individually. On JSTOR alone, there are over 2800 journals, translating to around 12 *million* articles. Taking the average size of an article to be 5000 words, this amounts to *hundreds* of years ($3 * 10^8$ minutes) for a *single* journal provider. These current estimates, as well as exceedingly large predictions, represents the sheer complexity of digital data.

In an attempt to tame these large data sets, a key concept called *information* emerged. Within modern databases, this is pronounced in the way data is organized and the relations between them, with recent integration with AI systems. The most prominent ones are **ontologies**, particularly OWL, or **knowledge graphs**, e.g., John Sowa's Conceptual Graphs. More recently, AI systems are being more deeply integrated with databases, providing an easier way for users to query across a large amount of websites or resources at a time. However, most approaches focus on information via a *theory*, connecting data by true relations and facts. The potential for contradictory facts can compromise the reliability of a knowledge base, though there are some promising approaches, including using paraconsistent logics. Additionally, research done in AI aims to provide a good "average"; the amount of fake data produced is a concern, as noted in [1].

Analyzing this problem from a theoretical lens, the natural question arises: *why* is there so much information present? Could it be *compressed* into a smaller form? The leading two theories on the matter provide hard limitations on compression, each with their own notion of "information" that face certain limitations:

- **Shannon entropy:** Claude Shannon founded information theory and defined information as the "reduction of uncertainty", measured in a probablistic setting. However, this applies to *noisy channels*, such as passing 010 to a receiver (with a probability of success). This is distinct from the *structural* view of information, such as a deterministic total of a data set.
- **Kolmogorov complexity:** Andrey Kolomogorov founded Algorithmic Information Theory, independently connecting Shannon's work to computability. This is defined on the **Minimum Description Length (MDL)** principle, that the best representation of a string is the smallest one possible. Kolmogorov formalized this idea with Kolmogorov complexity, the size of the smallest Turing machine that accepts a string. While this is more structural in nature

- **Scott domains:** Dana Scott introduced an algebraic structure to represent information and described how information can be consistent with one another. These structures come close to providing a semantic basis, but ultimately focus on the *hierarchies on pieces of information*, and not the *connections between information*.

This thesis proposes to re-examine the notion of information. Using the etymology in Latin, "to form", this thesis develops a formalized programming language to organize information.This provides the missing link to the *semantics* of information, more so than labels within Knowledge Graphs or ones commonly used in AI.

1.1. **Goals.**

The aim of this thesis is to create a formalized programming language, called Welkin, originating from the German word *wolke* (cloud) [2]. The following goals resolve the deficiencies observed in the formalisms above:

- **Universality (G1):** we provide a theory to work with *any* partial computable function. Thus, we ensure that checking a certificate is always decidable, and encode appropriately (for set theory, type theories, etc.).
- **Standardization (G2):** the language must be specified by an unambiguous standard and have a reliable **Trusted Computing Base**. To ensure this, the language must have an unambiguous syntax and semantics, as well as a reliable meta-theory.
- **Encoding (G3):** information must be encoded in an optimal yet efficient way (effective linear-time checking). We provide a 64-bit hashing scheme for most implementations, as this ensures enough unique IDs while being efficient on modern hardware.

1.2. **Outline.**

This thesis is organized linearly, shown in Table 1. Note that this thesis serves as a *high-level*, but precise, guide to the Welkin language. The *precise* specification is the Standard itself, with *minimal* abbreviations and all details explicitly mentioned.

Section	Description
Section 2	Discusses foundations, providing the meta-theory based on Gödel's System T to more easily encode computability and have a theory equi-consistent to Peano Arithmetic.
Section 3	Defines information and the optimal encoding proven in this thesis. This is optimal w.r.t. to a certain efficiently computable class of encodings.
Section 4	Introduces the LL(1) grammar for Welkin, a graph-based language that naturally encodes lambda terms.
Section 5	Explains the semantics of terms in the theory and finalizes the encoding. This also introduces the 64-bit hashing scheme.
Section 6	Reviews the Welkin Standard and justifies why Welkin is "strong enough" to "encode itself".
Section 7	Reviews the work in this thesis and provides insights to applications for formal verification and creating custom languages that compile into Welkin.

TABLE 1. Outline of the thesis.

2. FOUNDATIONS

We introduce the base theory needed for this thesis. This theory embodies a unifying concept for formal systems: computability. We capture this through a suitable simple type theory.

We will keep this self-contained; additional references will be provided in each subsection. For general notation, we write $:=$ to mean "defined as".

2.1. **Base Notions.**

Before continuing, we must introduce some fundamental notions.

Definition 2.1.0. The set of **hexadecimal digits** is the set of symbols given by $\mathbf{Digit}$, shown in Table 2. The **binary digits (bits)** are $\mathbf{Bit} := 0\ 1$.

$\mathbf{Digit} :=$	0	1	2	3	4	5	6	7	8	9
	zero	**one**	**two**	**three**	**four**	**five**	**six**	**seven**	**eight**	**nine**
	A	B	C	D	E	F				

TABLE 2. Hexadecimal digits.

The **binary digits (bits)** are $\mathbf{Bit} = 0\ 1$.

Definition 2.1.1. The **language of words** is given by $\mathcal{L}_W$ in Table 3. A **word** is defined by the judgments in Figure 1.

		Symbol	**Name**
$\mathcal{L}_W$	$:=$	Digit	
		.	**Concatenation**
		$=$	**equality**
		not $=$	**inequality**

TABLE 3. Language of words

$$\frac{\Gamma \vdash w : W}{\Gamma \vdash w.0 : W, w.1 : W}$$

FIGURE 1. Recursive definition of words.

Definition 2.1.2. Equality on binary strings is defined recursively:

- **Base case:** $0 = 0$ and $1 = 1$, but *not* $0 = 1$ (which we denote by $0 \neq 1$).
- **Recursive step:** let w, w' be binary strings. Then if $w = w'$, then $w.0 = w'.0$ and $w.1 = w'.1$

Whenever two binary strings w_1, w_2 are not equal, we write $w_1 \neq w_2$.

Remark 2.1.3. The definition for binary strings, as the remaining recursive definitions, serves as a suitable *uniform* abstraction for data. From a physical viewpoint, we cannot *verify* each finite string, a phenomena related to the notion of "Kripkenstein" [3]. However, we *can* provide the template and is more suitable as a definition, and we presume these definitions are completely contained (i.e., binary strings are defined by a finite combination of *only* the rules above). On the other hand, proof checking will be done in an ultra-finitistic setting and is addressed in Section 6.

For simplicity, our primary encoding uses binary. We directly use this in the notion of a variable in the next section. We review the primary number systems natively supported by Welkin.

2.2. **System T.**

The choice for foundations go back to the very foundational crisis of the 20th century. During this time, logicians sought a rigorous underpinning of mathematics,

responding to uncertainty in the past with the introduction of the real numbers. The predominant system that remains today is ZFC. We explore an alternative well known in computer science: type theory. Starting in 1932, Alonzo Church introduced his original untyped lambda calculus [4]. However, it was quickly shown to be inconsistent, via the Kleene-Rosser paradox, but was fixed in 1936 with a revision [5]. He then restricted it further in 1940 with simple type theory [6], which is the basis today for most proof assistants. For additional context, please consult [7].

The simply typed lambda calculus on its own is too weak for proofs on computability. The solution is to augment this with induction, via Kurt Gödel's System T [8]. We closely follow the presentation from [9]. This theory consists of three key parts:

- **Types:** Definition 2.2.4.
- **Terms:** Definition 2.2.6.
- **Operational Semantics:** Definition 2.2.8.
- **Normal forms:** Definition 2.2.7.

Definition 2.2.4. The **base language of System T** consists of symbols $L_{BT} = \{\top \perp 0\, 1 + * \mathbb{B}\, \mathbb{N}\, x : \to \times \lambda\, D\, R \langle \rangle\, (\,)\}$. The **types** of System T are defined recursively:

- **Base case:** These are called **base types**.
  - $\mathbb{B}$ is a type, called the **boolean type**.
  - $\mathbb{N}$ is a type, called the **natural numbers type**.
- **Recursive step:** let U and S be types. Then $U \to S$ is a **function type** and $U \times S$ is a **product type**. Moreover, we set $(U) \equiv U$.

Definition 2.2.5. The **(complete) language** is $L_{ST} = L_{BT} \cup \mathbf{Var}$, where $\mathbf{Var}$ consists of the **variables**, symbols x_i^S for each binary string i (the **index**) and type S. A recursive definition can be adapted from Definition 2.1.2 and the one above.

Definition 2.2.6. The **terms** of System T are defined recursively.

- **Base case:**
  - Each variable x_i^S is a term of S.
  - $\top$ and $\perp$ are the *only* terms of $\mathbb{B}$.
  - 0 and 1 are terms of $\mathbb{N}$.
- **Recursive step:**
  - If u is a term of U and v is a term of V, then $\langle u, v \rangle$ is a term of $U \times V$.
  - Given a term $\langle u, v \rangle$ of $U \times V$, then u is a term of U and v is a term of V.
  - If t has type τ and f has type $S \to U$, then $f(t)$ has type U.
  - For each variable x_i^S of type S, if $f(s)$ has type U, then $\lambda x_i^S . f\big(x_i^S\big)$ has type $T \to U$.
  - If $u : U$, $v : U$ and $t : \mathbb{B}$, then $D\, u\, v\, t$ has type U.
  - For each term n of $\mathbb{N}$, $n + 1$ is a term of $\mathbb{N}$.
  - Let $u : U$, $v : U \to (\mathbb{N} \to U)$, and $t : \mathbb{N}$. Then $R\, u\, v\, t : U$.

Definition 2.2.7.

Definition 2.2.8.

Remark 2.2.9. For notational ease of use, we will add several conventions:

- Variables will be denoted with letter names $a, b, ..., z$ with an implicit index.
- We define new notation with parantheses via recursion (with the base case already set above): let σ, τ, ρ be types.
  - We set products to be **left-associative**: $\sigma \times \tau \times \rho \equiv (\sigma \times \tau) \times \rho$.
  - Similarly, $+$ and $*$ on $\mathbb{N}$ are left-associative.
  - We add **greater precedence** for $\to$ over $\times$: $\sigma \to \tau \times \rho \equiv (\sigma \to \tau) \times \rho$ and $\sigma \times \tau \to \rho \equiv \sigma \times (\tau \to \rho)$.

2.3. **Serial Consistency.**

As mentioned, System T is closely related to Peano Arithmetic. Specifically, Gödel proved that Peano Arithmetic ($\mathbf{PA}$) is equi-consistent to System T. He did this to base the former on an intuisonistic logic via Heyting Arithmetic ($\mathbf{HA}$), further compounded by a theory of functionals. The proof is extremely technical and out of the scope of this thesis; we refer to [8] for details.

Theorem 2.3.10. *The consistency of the following theories are equivalent:*

- *System T*
- $\mathbf{HA}$
- $\mathbf{PA}$

Due to Gödel's second incompletness theorem, none of these theories can prove that *any* of them are consistent. However, a weaker property is *is* provable within these systems and represents a revival of Hilbert's program. This was discovered by Sergei Artemov, outlined in multiple papers [10]. We refer to his latest one but recommend the others.

Theorem 2.3.11. $\mathbf{PA}$ *proves that there exists a primitive recursive function (PRF)* s *such that, given a proof of* D*, verifies that it contains no contradictions. We call* s *a* ***selector*** *and say that* $\mathbf{PA}$ *is* ***serial-consistent****.*

Note that Artemov's condition is distinct from the normal definition of consistency, that there is a *single* proof to demonstrate this consistency. Thus, this does *not* contradict Gödel's incompleteness theorems, and in fact underlies an important principle, closely aligning to Artemov's views:

> **(P1)** ***Finitistic consistency relies on inductively verifying proof-checkers as combinatorial objects.***

This is closely tied to an equivalent notion: proofs on Σ_1^0 statements. This involves the arithmetic hierarchy, but in short, this is the set of *partial computable statements* we wish to verify. With Sergei's theorem, we obtain:

Corollary 2.3.12. $\mathbf{PA}$ *is* ***serial-***Σ_1^0***-sound****, which means that there is a PRF* t *that, given a proof of* Σ_1^0 *statement in* $\mathbf{PA}$*,* t *returns a* $\mathbf{PA}$*-proof of this statement.*

With this corollary, in combination to Theorem 2.3.1, we can *definitively claim* which statements are proven correctly in the realm of computatbility. This is key for the reliablity of the method presented in the remaining sections of this paper. It suffices to examine serial-consistency for this claim, so we will impose a *constructive proof* of

serial-consistency for theories at least as strong as $\mathbf{PA}$. Computationally, this corresponds to *correctly* providing proofs of termination of lambda terms.

3. Information Systems

We introduce the bulk of this thesis: providing an optimality criterion for an informatoin system and deriving the best one in terms of explicitly comptuable structures. We first introduce structures, followed by their representations. We then prove that the study to convert betweewn representations is RE-complete, ensuring the theory is sufficiently expressive.

3.1. **Information.**

In FOL, the usual definition of a structure relies on a tuple of finitely many relations and function symbols on a domain. We simplify the definition; this will be expanded upon in Section 5.

Definition 3.1.0. Data is a binary string. A **domain** X is a function $X : \mathbb{N} \to \mathbb{B}$, i.e., a "set" of binary strings.

Definition 3.1.1. Information is a **bigraph**, which is a tuple $\langle X, T_X, G_X \rangle$, where

- $X : \mathbb{N} \to \mathbb{B}$ is the domain.
- T_X is the **hierarchy** on X, which is a tree. More precisely, this is a pair $\langle \bot, p \rangle$, where $\bot$ is the **root** and $p : X \to X + \{\bot\}$ is the **parent function**.
- G_X is the **hypergraph** on X.

Example 3.1.2.

- Consider labels $\{\mathrm{dog}, \mathrm{cat}, \mathrm{mammal}, \mathrm{animal}\}$. We can define a bigraph with only a tree-structure:
  - $p(\mathrm{cat}) = \mathrm{mammal}$, $p(\mathrm{dog}) = \mathrm{mammal}$ (cats and dogs are mammals).
  - $p(\mathrm{mammal}) = \mathrm{animal}$ (mammals are animals).
- Consider grammatical elements to describe a story: $\{\mathrm{tortoise}, \mathrm{beats}, \mathrm{hare}\}$. We could write, $\mathrm{tortoise} - \mathrm{beats} \to \mathrm{hare}$. This corresponds to usual relations in OWL and other ontological frameworks.
- A more unique example is treating connections syntactically: consider $\{+, *, 1, 0\}$. We can create a graph with $1 - * - 0$ to denote the string $1 * 0$.

We show that this definition encompasses all of ontology in two ways:

- This notion is powerful enough to encode computation. This is already apparent with Turing machines, but we will directly spell this out.
- This encompasses the current ontological systems in use today. This will be shown with reductions from OWL and Conceptual Graphs, focusing on the *form* itself and *not* truth values.

3.2. **Bases For Turing Machines.**

Let $\mathcal{M}$ be the set of all Turing machines. We examine a suitable lattice-structure on this set provide a semi-lattice as a step towards organizational optimality.

Definition. Let $\mathcal{A} \subseteq \mathcal{M}$. We say $\mathcal{A}$ **spans** $\mathcal{M}$ if there is an explicitly computable, surjective $f : \mathcal{M} \to \mathcal{F}(\mathcal{A})$ such that $\mathcal{A} = \{M \mid f(M) = \{M\}\}$. In this case, we call f an **analyzer** of $\mathcal{A}$.

Analogous to group theory, bases enjoy a computable version of the First Isomorphism Theorem.

Theorem. *Suppose $(\mathcal{A}, f)$ is a basis for $\mathcal{M}$. The following hold:*

- *Let ρ be the function that takes Turing machines to the smallest Turing machine M' such that $f(M) = f(M')$. Then $g = f \circ \rho$ is an analyzer for $\mathcal{A}$. We call g the **canonical analyzer** w.r.t the basis.*
- *The inverse $f^{-1} : \mathcal{F}(\mathcal{A}) \to \ker(f)$ is explicitly computable; we call this a **synthesizer**. In particular, so is g^{-1}, which we call the **canonical synthesizer** (w.r.t. the basis).*
- *Define the following operations on Turing machines:*
  - $M_1 \sqcup M_2 = g^{-1}(f(M_1) \cup f(M_2))$
  - $M_1 \sqcap M_2 = g^{-1}(f(M_1) \cap f(M_2))$

  *Then $\sqcup$ and $\sqcap$ are explicitly computable, and $(M_1, \sqcup, \sqcap)$ is a semi-lattice. We call the induced partial order $M_1 \sqsubseteq M_2 \Leftrightarrow M_1 = M_1 \sqcap M_2$ a **part-hood** relation, and the system $(\mathcal{M}, \sqcup, \sqcap)$ the **Mereological System** of the basis.*

Proof ■

3.3. **Mereological Rewrite Systems.**

We generalize a basis to include rewrite components. This will be the starting point for discussing the optimality of the semantics.

Definition 3.3.3. Let (A, f) be a basis on $\mathcal{M}$. The **Mereological Category** $\mathcal{C}(A, f)$ is the largest category closed under explicit transformations on $\mathcal{F}(\mathcal{A})$. In detail, it contains:

- Objects: $\mathcal{F}(\mathcal{A})$.
- Morphisms: $\mathrm{Hom}(A_1, A_2) = \mathcal{E}(g^{-1}(A_1), g^{-1}(A_2))$. If $\mathrm{Hom}(A_1, A_2) \neq \varnothing$, then we write $A_1 \to A_2$.

Definition 3.3.4. Definition. A **progress theorem** is a proposition p of the form $\forall A.\exists D_A.\forall B.D_A(B) \Rightarrow B \to A$ where D_A is a family of explicitly computable unary predicates called the **progress predicate of p**. We say a Mereological Category has **progress p** if it satisfies p. Note that A may be free in D_A. We write $\mathrm{Prog}(\mathcal{C})$ for the set of progress theorems satisfied by $\mathcal{C}$.

Definition 3.3.5. A Mereological Category has **universal progress** if the **Universal Progress Theorem (UPT)** holds: for every $A \in \mathcal{A}$, there is a N such that, for every B with N elements, $B \to A$. This ensures the existence of $m : \mathcal{F}(\mathcal{A}) \to \mathbb{N}$, given by $m(A) = \min\{N \mid \forall B, |B| \geq N.B \to A\}$.

Definition 3.3.6. The category of Mereological Categories with universal progress consists of:

- Objects: Mereological Categories.
- Morphisms: $\sigma : \mathcal{C}(A_1, f_1) \to \mathcal{C}(A_2, f_2)$ are the faithful functors.

Our goal is to find the final object in this meta-category above with universal progress.

4. Syntax

5. Semantics

5.1. **Terms.**

We expand upon to anaylze the ASTs generated from Section 4.

6. Bootstrap

7. Conclusion

References

1. Romano, A.: Synthetic geospatial data and fake geography: A case study on the implications of AI-derived data in a data-intensive society. Digital Geography and Society. (2025)
2. Dictionary, O.E.: welkin, n., https://www.oed.com/dictionary/welkin_n
3. Loar, B., Kripke, S.A.: Wittgenstein on Rules and Private Language. Noûs. 19, 273 (1985)
4. Church, A.: A Set of Postulates for the Foundation of Logic. Annals of Mathematics. 33, 346 (1932)
5. Church, A.: An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics. 58, 345 (1936)
6. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic. 5, 56–68 (1940)
7. Laan, T.: The evolution of type theory in logic and mathematics. Presented at the (1997)
8. Gödel, K.: Über Eine Bischer Noch Nicht Benützte Erweiterung Des Finiten Standpunktes. Dialectica. 12, 280–287 (1958)
9. Girard, J.-Y., Taylor, P., Lafont, Y.: Proofs and types. Cambridge University Press, Usa (1989)
10. Artemov, S.: Serial properties, selector proofs and the provability of consistency. Journal of Logic and Computation. 35, exae34 (2024). https://doi.org/10.1093/logcom/exae034

Department of Mathematics, University of Colorado at Boulder, Boulder, CO
Email address: oscar-bender-stone@protonmail.com