Welkin
An Information Language

Oscar Bender-Stone

October 2023

Preface

To be written...

Chapters 1 and 2 provide the philosophical basis for Welkin. Chapter 3 contains the Welkin Standard, a formal specification describing high level, algorithmic properties in compliant Welkin interpreters. Chapter 4 explores several ways to use Welkin. Chapter 5 concludes with the future of digitally storing mathematical documents, synthesized programs, and the overall organization of ideas.

Chapter 1

Introduction

Startng Outline:

- Describe the ways philosophers and mathematicians have approached organizing the world around them.
  - Look at Cantor, Dedekind, Frege, Russell, Goedel, and up to the present.
  - Look at modern philosophers
- Reflect on two major shortcomings of these approaches: having little metaphysical explanation about *how* things can be combined, and a lack of generality for several concepts
  - No explanation for what *is* a set fundamentally, or, more importantly, how two entities can be combined into one.
    - Include formal and informal definitions
    - Recognize the prior *tool* used to start math in the first place (again, the ability to combine objects or ideas)
    - Explain that, while category theory may make these constructions more precise, it *still relies on predetermined tools.* These tools are physiological in nature, but they are *ultimately metaphysical.*
  - Lack of generality for: structures, the real numbers, logic (particularly paraconsistent logics). Specifically, mention:
    - Cryptomorphisms and Voldemort's Theorem. This is not part of official literature, but has been given in an extensive document. It serves an important philosophcial point (which will be rexammined in Chapter 2)
    - Mention the number of models for the reals, and how it still has not been decided what the official model is. Mention constructive concerns (overarrching with generality). (Cite Lesnik's paper as a step in the right direction, but critique the lack of generality)

        * Mention the role of bifrucation, and that there is opportunity to expand upon it more (into a generalized fuzzy logic)

- Explain the essential goals of Welkin, built upon CFLT
- Describe target audience
    - This document is geared towards philosophically or mathematically inclined persons. The official version is in English, but, in time, translations will be added.
    - In a different document (separate from this repo), there will be a more accessible version as a part of my program on "humanistic logic".

Chapter 2

Background

Possible beginning quote (at least for Western philosophers). "The Fold: Leibniz and the Barouqe"

> the two instances . . . have no windows, . . . for Leibniz, [this! is because the monad's being-for the world is subject to a condition of closure, all compossible monads including a single and same world. For Whitehead, on the contrary, a condition of opening causes all prehension to be already the prehension of another prehension. . . . Prehension is naturally open, open to the world, without having to pass through a window.

2.1 Western and Eastern Thought

- Contrast the distinct philosophies pervasive in Western and Eastern philosophy. Some key points to emphasize:
  - Mechanistic vs spiritual leaning
  - Classical logic vs Different Logics (particularly involving contradictions)
  - Brief mention in other philosophies (possibly ethics or other non-epistemology related fields?)

2.2 Sets: Georg Cantor and Others

- Cover most of Cantor's writings, including "Beiträge zur Begrundung der transfiniten Megenlehre".
  - Touch upon other thinkers that built upon Cantor, including Frege, Dedekind, etc. Dedekind particularly has a section on systems that he later uses to define naturals (and is extremely relevant to Cantor's definition).

  - Cover the core issues behind Cantor's original definition, including pyschological components. Use modern crtiqiues in current literature. (See references in: "New Essays On Peirce's Mathematical Philosophy")
  - Poin

- Explain the overreaching implications of set theory, as well as its materialistic issues. Briefly explore the possible resolutions to this (e.g., type theory), and demonstrate that they do not sufficiently resolve the materialistic roots.

- Explore the issue of defining an object, claiming a more general theory is possible. Argue that Cantor and others have used foci

- Touch upon the formalist response to set theory, and rebuttal against the claim that formalism alone can be used for mathematical thought.

- Helpful: look at the references in the metamath book

2.3 The Role of Foci

- Explore the issue of defining an object, claiming a more general theory is possible. Argue that Cantor and others have used foci (in a specialized form, collections)

- Justify the existence of different paradoxes the applicability of non-classical logics (particularly paraconsistent logic)

- Explain the shortcomings of foci alone in explaining both new phenomena and generality

2.4 Early Attempts at Continua: Cantor, Dedekind, and Others

- Revisit Cantor, particularly with Cantor's Theorem. Explore the different models of the real numbers.

- Referencing foci, explain how these models do not completley generalize the reals. Eve mention how category theory does not resolve this either (in both a mathematical and philosophical sense).

- Analyze Zeno's paradoxes and Aristotle's original definition of a continuum. Explain set-theoretic continua do not constitute full fleged continua (and are missing key metaphysical properties)

2.5 Continua: Charles Peirce

- Discuss Peirce's role in mathematics, logic, and ontology.
- Briefly mention Pragmatism and Peirce's search for its proof.
- Explain what Perice thought about Cantor's theory, and introduce his concept of multitudes
  - Elaborate on his initial readings of Cantor's theorem and definition of sets
  - Explore Peirce's shortcomings in his definition of collection
  - Mention some of his misconceptions, e.g., his presumption that the Continuum Hypothesis must hold
- Informally justify the mathematical value of Peirce's writings, arguing that more concrete axioms need to be developed from his ideas. In particular, explain why Perice was closer than Cantor to getting to a definitin of a bona fide continuum.

2.6 Folds: Deleuze

Possible quote to include about Deleuze, "The Fold: Leibniz and the Baroque", page 81 (maybe this is better suited for the introduciton?)

> For Leibniz... bifrucations and divergencies of series are genuine borders between incompossible worlds, such that the monads that exist wholly include the compossible world that moves into existence. For Whitehead (and for many modern philosophers)... bifrucations, divergences, incompossibilites, and discord belong to the same motley world that can no longer be included in expressive units, but only made or undone according to prehensive units and variable configurations or changing captures. In a same chaotic world divergent series are endlessly tracing bifurating paths. It is a "chaosmos" of the type found in Joyce, but also in Maurice Leblanc, Borges, or Gomrowicz.

- Describe Deleuze's work in epistemology and relevance throughout modern philosophy.
- Connect Deleuze to some concepts in Eastern thought, particularly tying his idea of folds with origami. Also mention how he challenged Leibniz's metaphysics.
- Explain how Deleuze's concept of a fold resolves Peirce's previous inquries into collections.

Chapter 3

The Welkin Standard

3.1 Preliminaries

3.1.1 Character Encodings

In a formalist fashion, our BNF variant leaves encodings as an generalized notion. Let Char be an arbitrary, finite set. An **encoding** is a mapping $\mathcal{E} : \mathbb{N} \to$ CHAR. We denote CHAR $= \mathcal{E}$(CHAR) and CHAR* as the Kleene-closure of CHAR. We also recognize a distinguished (possibly empty) subimage NUMBER $\subset$ CHAR.
Character encodings may be given as finite tables. Several major encodings are formally defined in the following sources.

- ASCII []
- UTF-8 []
- UTF-16 []

As an auxiliary set, define

$$\text{STRING(DELIMITERS)} = \{s \in \text{CHAR}^* \mid s = d_1 u d_2, u \in (\text{CHAR}\} \setminus \text{DELIMITERS})^*\},$$

where DELIMITERS $\subset$ CHAR^2. Strings may include their delimiters by escaping them, i.e., using a sufficient prefix or suffix to distinguish them from string boundaries. Any Welkin implementations may choose how to escape their strings by explicitly setting the (default) DELIMITERS. For the rest of this document, we will denote STRING ::= STRING(DELIMITERS).
We implicitly assume that $CHAR*$ does not conflict with literals defined in a given grammar, and that these literals can be escaped with a character $ESCAPE$.. Setting $ESCAPE = \varepsilon$, means there is no way to include predefined literals in CHAR.

3.1.2 EBNF Variant

Our variant of EBNF uses the following notation:

- ::= denotes rule assignment,
- `term ::=` $S \subseteq \text{CHAR}^*$ means `term` $\in S$,
- | denotes alternation,
- In a given choice, an arrow $\rightarrow$ denotes a new rule name. For example, the rule

```
term  ::=  'A' → A
       |   'B' → B
```

  is equivalent to

```
term  ::=  A | B
A     ::=  'A'
B     ::=  'B'
```

- `term*` means 0 or more instances and is shorthand for

```
terms ::= term terms | term | ε,
```

- `term+` means 1 or most one instances and is shorthand for

```
terms ::= term terms | term,
```

- Elements of CHAR* are included in quotes. To avoid confusion, literal quotes are denoted with [’].
- A production between a terminal and non-terminal, such as *term*”*A*”, means there is no whitespace character that appears between them. All other productions, in the form `term1 term2` any number of whitespaces may be included.

3.2 The Welkin Language

There are three fundamental variants of Welkin that define the foundation for the language:

- Base Welkin, a language mirroring the key properties of the core data structure;
- Standard Welkin, which extends Base Welkin with attributes. Attributes are a limited type of directive that can customize how the interpreter accepts or presents data;

- Binder Welkin, which enables arbitrary evaluation of Welkin files and access to Operating System resources. This is equivalent to Attribute Welkin with two new directives: @*eval* and @*exec*. See Section ?.? for more details.

Each of these variants can be parsed with LALR parsers and fundamentally have the same semantics. However, the third requires the highest level of privileges and, as proven in Section ?.?, makes the interperter Turing complete. All official implementations will include Binder Welkin as a separate, optional component.

Syntax

```
terms        ::=  term*
term         ::=  graph | connections | member | unit
graph        ::=  unit?  '{' terms '}'
connections  ::=  term (connector term)+
connector    ::=  '-' term '-' → edge
                |  '-' term '->' → left_arrow
                |  '<-' term '-' → right_arrow
member       ::=  unit?  ('.'(ident | string)?  | '#'num)+
unit         ::=  ident | string | num
ident        ::=  CHAR*
string       ::=  STRING
num          ::=  NUMBER
```

Base Welkin is given by the grammar in Figure ??. Note that if NUMBER $= \emptyset$, any instances of the rule `num` must be excluded from the parser. This grammar defines the text-based interface to the Welkin Information Graph (see Subsection 2.? for more details). Implicitly while parsing, we enforce the fact that all identifiers may not solely contain the symbols `, ., -, ->`. We enforce this by matching these symbols first.
Throughout this document, Welkin documents are formatted with the following convention: the ASCII sequence `->` is rendered as → (A graphical user interface may support this rendering via glyphs).

Semantics

A **scope** is recursively defined and intutively is a level of terms. A **valid** base Welkin file consists satisfies a unique naming rule: in every scope, there are no name collisons. In particular, every graph must **only be defined once.**
We first form an Abstract Syntax Tree (AST), from which we form the final stored data in a **Welkin Information Graph.**

Definition 3.2.1. Base Welkin is parsed into the following AST $\mathcal{A}$.

- Every term is a new subtree with its contents as children.
- Every graph is an ordered pair of its aliases and list of children.

- Every connection is an ordered pair:
  - Left arrows $u \to v$ correspond to a pair (u, v);
  - Right arrows $u \leftarrow v$ correspond to the pair (v, u);
  - Edges correspond to both a left and right arrow.
- Every atom is converted into its corresponding encoding via the function $\mathcal{B} : \text{CHAR} \to \text{BYTES}$

Definition 3.2.2. A **Welkin Information Graph (WIG)** is a tuple $G = (V, E, A)$, where

- V is a set of **vertices** or **units,**
- $E \subseteq V^2$ is a set of **(directed) edges,**
- $A : \text{CHAR}* \to V \cup E$ is an **alias function.**

Notice that not every vertex or edge in a WIG have an alias, as opposed to a colored graph. There are several examples demonstrating that, under a suitable transformation, a normalized WIG may contain new structures not found in a Welkin file. See Example ??.

Lemma 3.2.1. The conversion from ASTs to Welkin Information Graphs is valid.

The final output of parsing is a normalized WIG. We define Welkin Normal Form in the following fashion.

Definition 3.2.3. A WIG is in **Welkin Normal Form (WNF)** if ...

Based on this form, we have chosen a unique way to represent Welkin files. In particular, there is a representation under WIG (generalized) homotopies. We prove that there is a polynomial (or exponential?) algorithm to convert any WIG into WNF.

3.3 General Application Behavior

3.3.1 Customization

All Welkin files are infinitely customizable via the welkin config file, which is written in standard welkin.

Every .welkin file has a corresponding configuration. It may either be put as a comment or (preferred) written in a separate file, which, by default, is called config.welkin. It has the following format:

- Encoding
  - Options: ascii, utf-8, utf-16, other

  - In the case of other: we need to specify how to define an encoding. (We need a light-weight API for implementations)

- Grammar
  - Strength: bounded (only finitely nested graphs with a given nesting limit, no recursion), no-self (arbitrary nesting limit, but no recursion), full (recursion allowed)
  - Customized: use a builtin template or custom welkin file. These can be used to change any part of the grammar, including adding keywords, the symbols used, adding new symbols, etc. Essentially, this will be a way to built new grammars from the original specification; we will need a separate parser for this (i.e., a parser of BNF/Welkin accepted notation).

- (Optional) Language
  - Defaults to English. Can be written in the writer's desired language (as long as it has been configured in Encoding above)

Alternatively, a file with a different name may be used; see th

- Current problem: we need a way to recognzie which Welkin modules (graphs) define grammars. (Then we can figure out how one can customize *any* aspect of Welkin). Here are some possible solutions (and evaluations thereof):
  - Introduce a new meta-arrow `=>` designed specifically to be detected by Welkin (the left hand side denotes any terms in the user file; the right hand side must be terms that are defined by the Welkin interface). Using this symbol may initially work, but what if a user would like to use this meta-arrow in their grammar? How do we control how much the meta-arrow is used? I feel as though this could be dangerous, if not managed. Moreover, logistically, *how* do we inform users of the terms they may bind to? For example, in the long-term, how do we make it clear that there is a ident for, say, num (standing in for general int/float)?
  - Create a different welkin file, such as welkin.config, that only uses minimal syntax (or a slightly extended version). This would ensure that there is a distinction between the information in welkin files and the grammars that they utilize. However, we want to makek Welkin infinitely customizable, so I aim not to use this solution, if at all possible.
  - Preferred, but still in the works: bootstrap interfaces. We would probably want to add this subsection later on, but definitely mention it while customizing grammars. *Actually, customizing grammars is NOT a part of the standard, but rather, it is a part of the API. We will include EBNF and the custom grammar for standard Welkin*

  - I am aiming to use the last solution, but what should the standard Welkin language be? Should it be aimed for programmers? Should it be aimed for general information preservation?

- At this point, I believe I am aiming to use the first solution; it would be benefitcial for Welkin itself to intrepret what Welkin files do, not *directly* the other way around. At its core, Welkin files store information; it is up to an external program (such as Welkin) to figure out *how* those files may be interpreted.

  - Moreover, doing the other way around directly would prove formally difficult. For correctness, we would have to worry about having a correct model for C, the operating system, etc.. But, part of this project is building the infrastructure for *getting and organizing* that specification! For initial implementations, Welkin need not *have* a low-level specification, so neither should this standard. That should be taken care of in a formally verified programming language (or formally verified binaries).
  - I hope that using the meta arrow could be optional in time; there could easily be ambigiuity if care is not taken. For example, the rule (test1 -> test2 -> graph) is ambiguous because of the ordering of the parantheses; it is unclear what is ultimately turned into a rule.
  - However, if the lefthand side only uses terms in the minimal grammar, this would not be ambigiuous. That could certainly work. Regardless, *we need to make it clear HOW Welkin, the program, is involved. Using the meta arrow is probably the least intrustive way to go about this, and can easily be customized by the user.*

- Rephrasing above problem: how do we make it clear when we are defining a new semantics?

  - A key realization: Welkin *is a semantics language.* You can define anything in it! It suffices to show that we can embed Welkin into discrete foci (or, more directly, can define any Chomsky grammars. This part is pretty clear from the semantics given to base welkin (or for graphs, and for connections, not for a reference to self + ?))
  - We could try making code blocks that specifically contain bindings. This would be a fancier alternative to restricting names (or coming up with a system, such as names prefixed with). My hesitance to go with this solution is flexibility; should we force the users to do this?
  - Here is a possible question we can ask: how do we distinguish between regular and *interfaced* Welkin?
  - The goal is to have the *guest* tackle how to interpret Welkin... but we still want to put the guest in Welkin!
    * How do we gain "awareness" about Welkin's interfaces?

    * We need a place to talk about *interpretations.* We know Welkin can talk about it, but how do we **reference external things? (Files, programs, even a simple microcontroller?)**
        · Do we directly include a pointer to the file? Is that strictly necessary8? It could definitely break on different machines. We would have to keep some config data in mind.
        · We know that any possible import symbols, as per compiler theory (and general humanb comprehension) need to be applied directlyat the *top* of a page. But how do we define the import statement itself?
        · Can we bootstrap an interface mechanism?

- Current (recommended) solution: we define standard Welkin to have direct access to the interpreter, similar to prolog. *Later on, we will make it clear that standard Welkin can be queries, but can do more.* Ultimately, *it is up to the individual to decide whether they include programming elements into their own welkin files. This is the key thing!* All the user needs to do is define their EBNF in standard Welkin
    – Here's a big idea: we can go back and *refine* the base interface through standard Welkin! That's the powerful of Welkin, after all!
    – More precisely, we need to implement Vero, the Welkin interface for stating and verifying formal properties. *We only want to make this through the APIs specified by Welkin. The implementation should NEVER matter for verification.*
    – There should be a separate document outling the thought process behind Vero, but for the most part, Vero should be defined *with Welkin alone.* It will be the big first test for Welkin to make sure Welkin's claim for any layer of abstraction works well.
    – Vero comes with its own grammar which is fully customizable (some mathematicians/logicians/etc may want to change it, for their personal preference). We also have certain semantics that impose checks on graphs. This is the key thing here: *we want to make checks/formal properties explicit.*
    – Main takeaway: *the interface can always be defined first. It need not be perfect. We can start with any interface and clarify it with Welkin. We can delete things, but we NEVER have to. We can add onto it instead*
    – Moreover, using standard Welkin for *all implementations* is a good sanity check; how can we check different configs with completely different grammars! At some point, *we need to use the same language to customize things!* Starting with this idea, it is straight forward to make custom interpreter settings *in your own language.* You just have to invoke it in standard welkin at some point.

* In fact, some core plugins will use this idea as well! One of the big things to have is a suitable *build system.* That will be included and have a straightforward interface.
* Other key starter grammars include: bullets (for bullet point type notes) and literate (for a starting point with code blocks). (As an implementation detail, these should be *optional* packages the user can be install; they should be more straightforward to get setup)

In Welkin, we informally write the BNF above as follows:

```
term -> { graph connection ident string}
graph -> {{ident {}}->‘{’-term-‘}’}
connection -> {term-connector-term}
connector -> {edge arrow}
edge -> ‘-’
arrow -> ‘->’
ident -> CHAR*
string -> ‘“’ CHAR* ‘“’ | ‘`’ CHAR* ‘`’
```

3.4 Core Algorithms

3.4.1 Graph Encoding