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

We address several issues that arise from organization, specifically within mathematics and computer science. We stress that they are not restricted to these fields and are widely applicable to all others.

Abstractions

Fragmentation

Limitations in Extensions

Complexity

Cite sources of lines of code, mathematical documents, etc.

Cite: “How do Committees Invent”, Conway: “https://www.melconway.com/Home/pdf/committees.pdf” (While the sources in this paper have NOT been formally justified, we are addressing their concerns.)

Chapter 3

The Welkin Standard

This standard requires a cursory background in discrete mathematics and Context Free Grammars. A reading of [] and [] suffices to understand this document.

3.1 Preliminaries

3.1.1 Character Encodings

In a formalist fashion, we define text, character encodings, and character decodings as generalized notions. The discussion here may be carried out with bytes and specific data formats, but these concepts are beyond the scope of this standard.

Let Char be an arbitrary, finite set. An **encoding** is an injective mapping $\mathcal{E} : \mathbb{N} \to \text{Char}$. The associated **decoding** is the left-inverse $\mathcal{D} : \text{Char} \to \mathbb{N}$ of $\mathcal{E}$. There is a natural extension $\mathcal{D}^* : \text{Char}^* \to \mathbb{N}^*$ that maps sequences in Char pointwise to sequences in $\mathbb{N}^*$.

Character encodings may be given as finite tables, matching natural numbers with characters. Several major encodings are formally defined in the following sources.

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

We denote $\text{CHAR} = \mathcal{D}(\text{CHAR})$ and CHAR* as the Kleene-closure of CHAR, whose elements are called **words.** [1] We recognize distinguished (possibly empty) subimages $\text{NUMBER}, \text{WHITE\_SPACES} \subset \text{CHAR}$, as well as a set $\text{DELIMITERS} \subset \text{CHAR}^2$. A **string** s is a word such of the form

$$s = d_1 u d_2, u \in \text{CHAR} \setminus \text{DELIMITERS}, (d_1, d_2) \in \text{CHAR}^2.$$

[1] Traditionally, the Klenne-closure of a set A is denoted by A^*. However, to ensure our BNF can be written in pure ASCII, we append $*$ without a superscript on CHAR and its subsets.

We call u the **contents** of s.We define STRING := STRING(DELIMITERS) as the set of strings over DELIMITERS.

Every standard Welkin grammar is written in ASCII, but the interpreter may support any additional character encoding. (See Section ?.?).

As an auxiliary set, define

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

where DELIMITERS $\subset$ CHAR^2. Strings may include their delimiters by escaping them, i.e., using a sufficient prefix or suffix DELIMITER_ESCAPE to distinguish them from delimtiers.

We implicitly assume that $CHAR*$ does not conflict with literals defined in a given standard grammar. In terms of the recommended LALR parser, this means that literals are matched first, not identifiers containing those characters. However, this can be changed by creating a custom grammar (see Section ?).

3.1.2 EBNF Variant

Our variant of EBNF uses the notation in Table ?.?.

Concept	Notation	Example
Rule Assignment	$=$	$term = atom$
Empty String	$\varepsilon := \emptyset \in$ CHAR$*$	$term = \varepsilon$
Concatenation (No white space)	"."	
Concatenation (white space)	Separated by whitespace	$term = AB$
Alternation	$\vert$	$term = A \vert B$
Zero or more instance	$*$	$terms = term*$, equivalent to $terms = termterms \vert term \vert \varepsilon$
One of more instance	$+$	$terms = term*$, equivalent to $terms = termterms$
Literal characters	"chars", with $chars \in$ CHAR$*$	$list =$ "["$terms$"]"

Each BNF has an associated subset RESERVED $\subseteq$ CHAR$*$ for any literals that appear in the grammar. We will explicty state these for standard Welkin grammars in the next section.

3.2 The Welkin Language

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

- Base Welkin, mirroring the key properties of the core data structure,

- Attribute Welkin, extending Base Welkin with attributes. Attributes are a limited type of directive that can customize how the interpreter accepts or presents data,
- Binder Welkin, enabling arbitrary evaluation of Welkin files and access to the Operating System. 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, in Binder Welkin, @*eval* makes the interpreter Turing complete (see Section ?.?), and using @*exec* can significantly impact the user's system. For this reason, Binder Welkin is a separate, optional component, as detailed in the Section ?.?

Syntax

Set RESERVED = {`{, }, ., -, ->, <-`}.

<table>
<tr><th>Variant</th><th colspan="3">Grammar</th><th>No</th></tr>
<tr><td rowspan="19">Base Welkin</td><td>terms</td><td>::=</td><td>term*</td><td rowspan="19">If NUMBER = ∅, any instance of `nu`</td></tr>
<tr><td>graph</td><td>::=</td><td>unit? ‘{’ terms ‘}’</td></tr>
<tr><td>connections</td><td>::=</td><td>term (connector term)+</td></tr>
<tr><td>connector</td><td>::=</td><td></td></tr>
<tr><td></td><td>|</td><td>‘-’ term ‘-’ → edge</td></tr>
<tr><td></td><td>|</td><td>‘-’ term ‘- ’ → left_arrow</td></tr>
<tr><td></td><td>|</td><td>‘<-’ term ‘-’ → right_arrow</td></tr>
<tr><td>member</td><td>::=</td><td>unit? (‘.’(ident</td></tr>
<tr><td></td><td>|</td><td>string)?</td></tr>
<tr><td></td><td>|</td><td>‘#’num)+</td></tr>
<tr><td>unit</td><td>::=</td><td>ident</td></tr>
<tr><td></td><td>|</td><td>string</td></tr>
<tr><td></td><td>|</td><td>num</td></tr>
<tr><td>ident</td><td>::=</td><td>CHAR ×</td></tr>
<tr><td>string</td><td>::=</td><td>STRING</td></tr>
<tr><td>num</td><td>::=</td><td>NUMBER</td></tr>
</table>

Base Welkin is given by the grammar in Figure ??. Note that if NUMBER = ∅, 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). Recall Rule ?.? that specifices which characters are forbidden in identifiers.

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). Attribute Welkin is given in Figure ?.?

Finally, Binder Welkin is given by the BNF in Attribute Welkin composed by two new directives. In BNF, these are appended to the directives rule:

- `eval ::= 'eval'.'(' unit ')'`,
- `exec ::= 'exec'.'(' string ')'`.

We explain the semantics for these directives in Section ?.?

Semantics

We break down our semantics first by terms. Directives are handled separately in the next section.

Definition 3.2.1. Equality of terms.

- **Basis.** Two units are equal if they are the same kind and obey one of the following.
  - `ident` terms are equal if their corresponding characters are equal,
  - `string` terms are equal if their corresponding contents are equal. Thus, `"A"` coincides with `'A'`,
  - `num` terms are equal if they represent the same value. Thus, `1` coincides with `10E`.
- **Recursion.**
  - Equality of connectors
    -
    -
    - `edge` terms are equal if they correspond as both left and right arrows.
  - Two connections are equal if they connect the same terms and have equal connectors.
  - Two graphs are equal if they contain the same terms.
-

A **scope** is recursively defined and intutively is a level of terms.

Definition 3.2.2. (Scope) Let t be a term.

- If t is not contained in a graph, then scope(`term`) $= 0$,
- If $G' \in G$ are both graphs and $\text{scope}(G') = n$, then for all $t \in G'$, $\text{scope}(t) = \text{scope}(G) + 1$.

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.** Note that, by the way equality was defined between two numbers, there can only be one representation of a given number in a scope. For example, using 1 and $10 * E^{-1}$ in the same scope would produce a name collison.
We first form an Abstract Syntax Tree (AST), from which we form the final stored data in a **Welkin Information Graph.**

Definition 3.2.3. 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 $u - a \to v$ correspond to a triple (u, a, v);
  - Right arrows $u \leftarrow a - v$ correspond to the triple (v, a, u);
  - Edges correspond to both a left and right arrow.
- Every unit is converted into its corresponding encoding via $\mathcal{E}^*$.

Definition 3.2.4. A **Welkin Information Graph (WIG)** $\mathcal{G}$ consists of

- sets $G_0, G_1, \cdots, G_n$ of **layers**,
- for each $0 \leq k \leq n$, functions $s_k, t_k : G_{k+1} \to G_k$ called the i-th **source** and **target** maps, respectively,
- for each k, an injective function $i_k : G_k \to G_{k+1}$,
- a set $\mathcal{L}$ of **labels** or **aliases**, and
- for each k, an injective function $l_k : \mathcal{L} \to G_k$ called the k-th **labeling map,**

that obey the following equations:

- $s_k \circ s_{k+1} = s_k \circ t_{k+1}$
- $t_k \circ s_{k+1} = t_k \circ t_{k+1}$
- $s_k \circ i_k = \mathrm{id} = t_k \circ i_k$
- (Condition to allow edges to be the "intervals" of connectors)

We may illustrate the above definition via the following diagram.

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 ??.
To select a single graph from a layer, we use the following definition.

Definition 3.2.5. (Abstract version of graphs) ...

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 Canonical Form in the following fashion.

Definition 3.2.6. A WIG is in **Welkin Canonical Form (WCF)** 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

Note that all apparent structures may be adjusted under cryptomorphism.

Directives

Each directive relies on the following components.

- Parser: takes in a Welkin file and generates an AST,
- Validator: ensures that the AST is valid, raising an error that directly points to a violation,
- From here, an AST may be processed by three different means:
  - Recorder: takes the AST, converts it into a WIG in WCF, serializes the data,
  - Attributor:
  - Binder:

Directive	Definition	Example
`import`	Concatenates the file	Example

3.3.1 Customization

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

Attribute Welkin

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